8-29-2019 2:30 PM

New Algorithms for Computing Field of Vision over 2D Grids

Evan Debenham
The University of Western Ontario

Supervisor
Solis-Oba, Roberto
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

Abstract

In many computer games checking whether one object is visible from another is very important. Field of Vision (FOV) refers to the set of locations that are visible from a specific position in a scene of a computer game. Once computed, an FOV can be used to quickly determine the visibility of multiple objects from a given position.

This thesis summarizes existing algorithms for FOV computation, describes their limitations, and presents new algorithms which aim to address these limitations. We first present an algorithm which makes use of spatial data structures in a way which is new for FOV calculation. We then present a novel technique which updates a previously calculated FOV, rather than re-calculating FOV from scratch. We then compare our algorithms to existing FOV algorithms and show that they provide substantial improvements to running time and efficiency of memory access.

Keywords

Field of Vision (FOV), Computer Games, Visibility Determination, Algorithm Analysis.

Summary for Lay Audience

In many computer games checking whether one object is visible from another is very important. Field of Vision (FOV) refers to the set of locations that are visible from a specific position in a scene of a computer game. The scene may contain vision-blocking objects, which prevent some locations within the scene from being visible. Once computed, an FOV can be used to quickly determine the visibility of multiple objects or locations from a given position.

This thesis summarizes existing algorithms for FOV computation, describes their limitations, and presents new algorithms which aim to address these limitations. We first present an algorithm which uses a more efficient way to store and access vision-blocking objects within a scene. We then present a novel technique which updates a previously calculated FOV, rather than re-calculating FOV from scratch. We then compare our algorithms to existing FOV algorithms and show that they provide substantial improvements to running time and efficiency of computer memory access.

Acknowledgements

I would like to first express my deepest gratitude to my supervisor, Professor Roberto Solis-Oba. He has given me constant feedback and guidance throughout the process of researching and writing this thesis. He did this while offering me the freedom to pursue a topic outside of his immediate focus of research. I owe a great amount of the final product of this thesis to his continual commitment and support.

I am also grateful for the help received from other faculty members of the Department of Computer Science at The University of Western Ontario. In particular I would like to thank Professor Michael James Katchabaw, whose reading course was a great help towards the research for this thesis.

I would also like to thank the community of RogueBasin.com for building a collection of existing open-source work on FOV calculation. Their collection served as the starting point for this thesis.

Lastly, I would like to thank my parents for their support and advice throughout my study.

Table of Contents

Abstract ii

Summary for Lay Audience iii

Acknowledgements iv

Table of Contents v

List of Tables vii

List of Figures viii

Chapter 1 1

1 Background 1

1.1 An Introduction to Field of Vision 1

1.2 Other Visibility Techniques in Computer Games 5

1.3 Existing FOV Algorithms 12

1.4 Analysis of Existing FOV Algorithms 21

1.5 Correctness Issues with Recursive Shadowcasting 28

Chapter 2 30

2 Improving FOV Calculation 30

2.1 Grouping Vision-Blocking Cells 30

2.2 Splitting Vision-Blocking Groups into Rectangles 33

2.3 Storing Rectangles in a Quadtree 36

2.4 Handling Cases Involving Multiple Rectangles 38

2.5 Ordering Rectangles and Calculating the FOV 42

2.6 A Brief Evaluation of Rectangle-Based FOV 44

Chapter 3 48

3 Updating an Existing FOV 48

3.1 An Overview of FOV Updating 48

3.2 Inverting Cones to Update an FOV 51

3.3 Ordering Cones for Inversion 54

3.4 Checking the Visibility of Cones 57

3.5 Rectangle Intersections with a Cone 59

3.6 The Cone Inversion Algorithm 64

Chapter 4 68

4 Experimental Evaluation of Our FOV Algorithms 68

4.1 Environments 1 to 4 68

4.2 Environments 5 to 8 77

Chapter 5 87

5 Conclusions and Future Work 87

5.1 Conclusions 87

5.2 Potential Future Work 88

References 91

Curriculum Vitae 93

List of Tables

Table 1: Mean running times of our algorithm implementations in environment 1 24

Table 2: Mean running times of Doryen implementations in environment 1 24

Table 3: Mean running times of our algorithm implementations in environment 2 25

Table 4: Mean running times of Doryen implementations in environment 2 25

Table 5: Mean running times of our algorithm implementations in environment 3 26

Table 6: Mean running times of Doryen implementations in environment 3 26

Table 7: Mean running times of Shadowcasting and Rectangle FOV for Figure 22. 45

Table 8: Mean running times for Environment 1 70

Table 9: Mean running times for Environment 2 70

Table 10: Mean running times for Environment 3 71

Table 11: Mean running times for Environment 4 71

Table 12: Mean performance statistics for Environment 1 73

Table 13: Mean performance statistics for Environment 2 74

Table 14: Mean performance statistics for Environment 3 75

Table 15: Mean performance statistics for Environment 4 75

Table 16: Running times for Environment 5. 81

Table 17: Running times for Environment 6 82

Table 18: Running times for Environment 7 83

Table 19: Running times for Environment 8 84

List of Figures

Figure 1: An example of FOV in a game with simple 2D graphics. 1

Figure 2: An example of fog of war in League of Legends before image filtering..... 4

Figure 3: A demonstration of shadow mapping..... 9

Figure 4: A simple grid with a correctly calculated FOV for strict (left), shadowcast (middle), and permissive (right). 13

Figure 5: Lines of visibility for shadowcast FOV (left) and strict FOV (right). 13

Figure 6: An example of Mass Ray FOV on a simple grid. 14

Figure 7: An example of Perimeter Ray FOV on a simple grid. 15

Figure 8: An example of Shadowcasting octants, numbered 1-8. 16

Figure 9: An example of Recursive Shadowcasting on a simple grid. 17

Figure 10: An example of Precise Permissive FOV on a simple grid. 19

Figure 11: Examples of each testing environment in a simple 9x9 grid..... 23

Figure 12: An example of Recursive Shadowcasting producing incorrect output. 28

Figure 13: How cells within the region occluded by one vision-blocking cell would be assigned not visible status..... 31

Figure 14: Cell traversal order of Recursive Shadowcasting for each octant..... 32

Figure 15: Vision blocking cells (left) being transformed into a rectilinear polygon (center), and then a set of rectangles (right). 33

Figure 16: A figure showing the segmentation of a rectilinear polygon. 34

Figure 17: Rays cast from relevant points for two separate source positions. 35

Figure 18: A grid with two vision-blocking cells (left), its quadtree representation with L=1 (right), and the space represented by each node (middle). 36

Figure 19: An example of two adjacent rectangles 38

Figure 20: The example from Figure 19, now adjusted to give correct output. 39

Figure 21: A figure showing rectangle occlusion and shrinking. 41

Figure 22: An example of our testing environment on a 13x13 grid. 45

Figure 23: Cones made by a rectangle, S_1, and S_2. Origins are shown with a dot. 49

Figure 24: Rectangle B has cones that are not visible (left), transitioning visible (center), and visible (right). 51

Figure 25: A height 2 binary tree of rectangles, approximating a cone. 59

Figure 26: Example of a rectangle intersecting both edges of a cone (left), intersecting one edge of a cone (center), and intersecting no edges of a cone (right). 62

Figure 27: a primarily horizontal cone with numbered columns 63

Figure 28: An example of a rectangle r which may have intersected a binary tree, but not the cone itself 64

Figure 29: An example of Environments 1, 2, 3, and 4. 69

Figure 30: A simple FOV grid with a single vision-blocking rectangle R. The grid is of size 5x5 on the left and of size 15x15 on the right. 78

Figure 31: Environments 5, 6, 7, and 8 on a grid of size 128*128. 80

Figure 32: A simple environment with FOV calculated according to the shadowcast FOV definition(left), and the strict FOV definition(right). 88

Chapter 1

1 Background

This chapter gives a background on Field of Vision (FOV) and other visibility techniques in computer games. This includes an explanation of what FOV is, descriptions of several FOV algorithms, and a brief comparison of them.

1.1 An Introduction to Field of Vision

A field of vision is the set of locations that are visible from a specific position in a scene of a computer game. FOV is calculated over a two-dimensional finite grid, referred to as the *FOV grid*. One grid cell is specified as the source of vision and is referred to as the *FOV source cell*. Some grid cells are also specified as representing vision-blocking objects in the game. An FOV algorithm must determine which cells are visible from the source and which cells are not visible based on the cells that are vision-blocking. The resulting grid with cells labelled as visible and non-visible is called the field of vision.

Figure 1 gives an example of FOV in a game. The scene of a simple 2D game is shown on the left with the FOV grid superimposed in pink. On the right a representation of the FOV grid is shown: the source cell is marked with an S, vision-blocking cells are marked with a pattern, and non-visible cells are darkened. FOV grids are usually calculated at a relatively low resolution, as this provides better performance. In Figure 1 each grid cell corresponds to a 48x48 pixel region

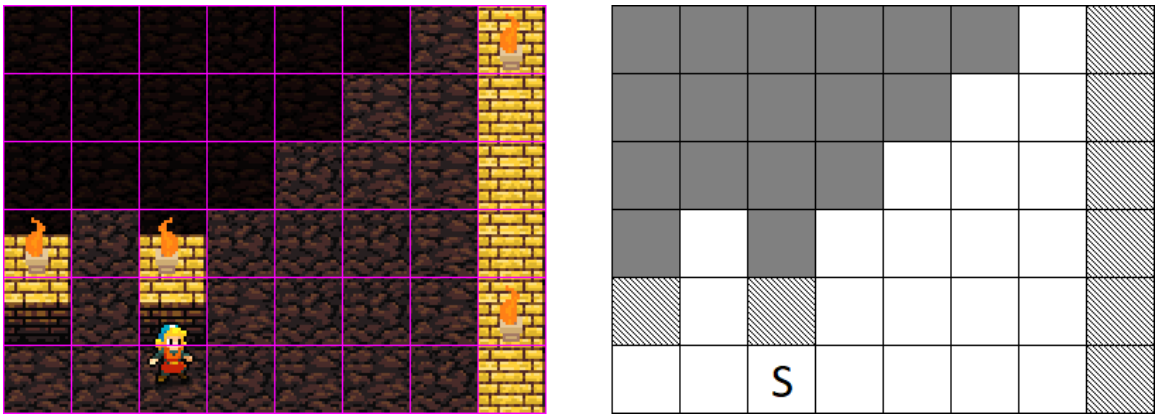

Figure 1: An example of FOV in a game with simple 2D graphics [1].

Calculating an FOV is useful in computer games with a top-down perspective. In these games the player views the game world from above and thus sees much more of the game world than an individual character inside the game. This influences game design, and results in several situations where FOV is useful. For example, these games may wish to convey to the player which areas of the world their character cannot see by visually darkening them. This visual effect is referred to as a *fog of war* and is calculated using FOV (see Figure 1).

An example of games which use fog of war is the action real-time strategy genre. These games place two teams of players on a large map with each player controlling one character. Fog of war is an extremely important visual effect in these games as a player's strategy is very dependent on which areas of the game map their character can see. Fog of war allows a player to quickly see which areas are not visible and make decisions based on that information. Action real-time strategy games are very popular, the most significant games in the genre are League of Legends [2] and Defense of The Ancients 2 [3], each with tens of millions of active players.

FOV is also useful for determining visibility, which is a common task for most computer games. A game environment will likely have objects which obstruct movement and vision, such as walls or trees, and it is important that game actors interact with these objects realistically. Actors may need to take visibility into account when making decisions, such as an enemy checking if it can see the player before attacking them.

The simplest approach to determining visibility is through a line of sight check. For two points A and B, if the straight line connecting A to B does not intersect any vision-blocking objects then A can see B. A line of sight check has a non-trivial performance cost, as there may be many objects within a scene that would need to be checked for intersection. Despite this, lines of sight can work well in applications where a relatively small number of visibility checks are needed.

Line of sight checks are adequate for calculations involved in actor decision-making, in games that have a relatively small number of actors who do not need completely accurate visibility information. As an example, if an enemy turns a corner and wishes to attack the player, it would not be realistic for them to attack the instant they saw any part of the player. A game could, for example, perform a single line of sight check from the enemy to the player once every 250 milliseconds while still maintaining perfectly believable enemy behavior.

However, games with a top-down perspective may have many actors which need rapid and accurate visibility information. FOV is useful to these games as it allows visibility information to be quickly referenced from the FOV grid itself, rather than repeatedly performing line of sight checks.

Classical real-time strategy games are an example of a genre of game where many visibility calculations are needed, and line of sight checks are not sufficient. In these games players control armies of up to hundreds of characters in real-time, and they are expected to react to player input nearly instantly. Characters may be given an instruction such as ‘attack the first enemy you see’ and will be expected to immediately attack the first visible enemy, the moment they become visible, from among hundreds of potential enemies. Field of vision allows a real-time strategy game to compute the area which a given character can see once and then check if any enemies are within it, instead of performing many line of sight checks for each character.

League of Legends by Riot Games [2] is an excellent example of how FOV grid resolution affects game quality. The game uses an FOV grid size of 128x128 cells, which spans the entire game map. This results in a fog of war which is adequate for gameplay but visually blocky and unrealistic (see Figure 2). The game was originally released in 2009 and has undergone significant development since then, including a visual overhaul in 2016. As part of that overhaul Riot Games wanted to improve the visual quality of their fog of war. They experimented with increasing the FOV grid size but deemed it to be too large of a performance bottleneck [4]. They instead opted to treat the FOV as an image, and upscale/blur it to increase the perceived quality.

Figure 2 gives an example of FOV in League of Legends before image filtering is applied. The character at the top-left cannot see the region to the bottom-right because it is out of its range of visibility. The visible region is circular, but because of the low FOV resolution the edge of the region appears as a series of jagged lines, rather than a smooth curve. This is in contrast to the other visual elements of the game, which are well detailed. This clearly highlights the performance constraints of existing FOV algorithms, especially as demand for realism in games increases.

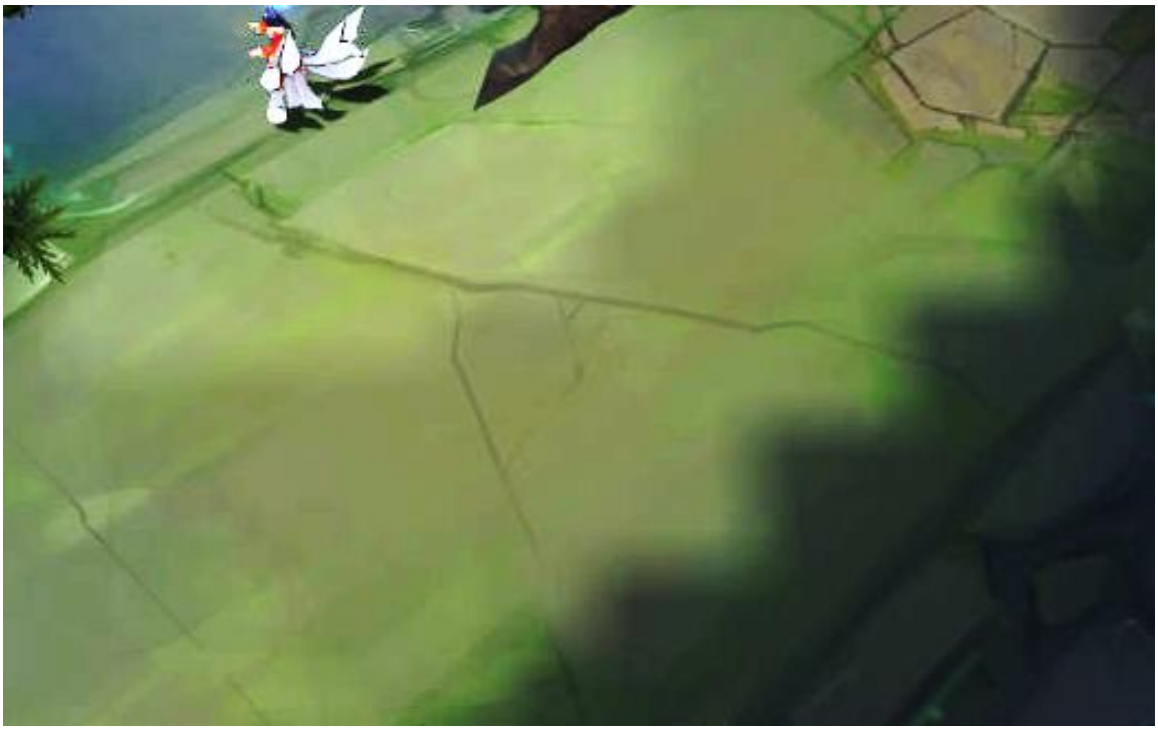

Figure 2: An example of fog of war in League of Legends before image filtering

1.2 Other Visibility Techniques in Computer Games

There are many other visibility techniques used in computer games which are conceptually similar to FOV. These problems relate to drawing objects within a game to the screen. We provide a brief survey of these problems and some of their solutions and explain why these solutions are not applicable to actor decision-making, where FOV is useful.

The data which comprises an image displayed on a computer screen is referred to as a *frame buffer*. A frame buffer is a two-dimensional array with width and height matching the resolution of the screen. Each location within this array stores information about one pixel of an image. Every pixel includes a red, green, and blue component, which combine to specify the color of that pixel.

Creating a complete frame buffer is a very computationally intensive process, so much so that specialized hardware is widely available for this sole purpose. This hardware is referred to as a graphics card, and it contains its own memory and graphics processors (GPUs) which are designed to rapidly render objects to the frame buffer and then display that frame buffer on the screen. Graphics cards have led to massive leaps in the number and complexity of objects which can be rendered on a screen, but their specialized nature means that they are only suitable for scene rendering within a game. They are not suitable for processing other aspects of a game, such as actor decision-making, which must be performed by a computer's central processor (CPU).

Graphics rendering requires many visibility calculations, and line of sight checks do not give adequate performance for this task. For instance, if a shadow needs to be drawn, the computer cannot calculate it with acceptable speed using line of sight, as a line of sight check would need to be performed for each pixel on the display for every rendered frame. The most common monitors can display over 2 million pixels at 60 frames per second, which means that accurately rendering a single shadow would require a ridiculously large number of line of sight checks.

There is a large body of academic work on visibility determination techniques designed for the graphics card which address various graphical visibility challenges. Examples of this work include hidden surface removal and shadow calculation. These techniques are briefly explained below.

If one object is obscured behind another, it is important that the further object is not rendered to the frame buffer on top of the closer object. Hidden Surface Removal (HSR) techniques are used to solve this problem, by ensuring that the occluded region of the further object is not included in the framebuffer for the scene.

The classic approach to correctly rendering objects of varying distances is the painter's algorithm [5]. When rendering a scene using the painter's algorithm, the objects in the scene are rendered to the frame buffer in order from furthest to closest to the viewpoint. By rendering objects from furthest to nearest, the painter's algorithm ensures that closer objects are always rendered on top of further objects. This approach works well for simpler scenes but is not useful in modern computer graphics.

Modern GPUs are optimized to render large batches of objects at once, and not one object at a time. Rendering objects in large batches results in substantially better performance than rendering individually. Complex scenes are made of many objects, and so batching them into as few GPU operations as possible is important for performance. This makes the painter's algorithm unsuitable for these cases, as it requires that everything be sorted and then rendered individually.

Modern computer graphics also make use of moving objects and changing terrain. The painter's algorithm requires that the objects in a scene be stored in a data structure which allows for rapid sorting based on the position of the viewpoint, but such data structures are very slow to build or update. Because of this, the painter's algorithm is also unsuitable for scenes with moving objects, as the data structure which stores them would need to be constantly updated.

The more modern approach to correctly and efficiently render objects of varying distances is to add a per-pixel depth value to the framebuffer, which is called a *z-buffer* [6]. When deciding whether to write color values for a given pixel, the GPU compares the existing z-buffer value for that pixel with the incoming z-buffer value. If the current z-buffer value is less (i.e. closer) than the incoming value, that pixel is not written to. Using a z-buffer ensures that distant objects are not rendered over near ones regardless of the order geometry is rendered in. Z-buffer use is nearly ubiquitous in modern computer graphics, and it is a standard feature of both GPU hardware and graphics programming languages [7].

Hidden Surface Removal (HSR) techniques can also improve the performance of rendering of a 3D scene by identifying which objects in a scene are not visible from the viewpoint from where the scene is being rendered. As non-visible objects do not affect the rendered output, skipping the rendering of them improves performance with no change to the resulting frame buffer. On large complex scenes this provides a substantial performance improvement over rendering everything regardless of visibility. These HSR algorithms often make conservative approximations, as it is better to render some small portion of non-visible objects if it means an HSR algorithm can be much faster. There are many different HSR algorithms for dealing with different situations in which all or part of an object may not be visible.

The most simple HSR technique which improves rendering performance is view frustum culling [8]. The *view frustum* is the region of space which is visible from the viewpoint from where the scene is being rendered. Objects which do not intersect the view frustum can be skipped, as they are guaranteed to not be visible. When testing for intersection with the view frustum, the shape of objects is often approximated to increase speed, with the trade-off that objects which are just outside of the frustum will occasionally be rendered. View frustum culling is universally useful but does not completely remove non-visible objects on its own. In a scene where closer objects (such as a wall) occlude further objects, view frustum culling will not be useful for skipping those further objects.

HSR techniques which concern themselves with skipping the rendering of objects which are occluded behind other objects are called occlusion culling techniques. These techniques are more complex than techniques like view frustum culling but are very important for performance in denser environments where a small number of objects may occlude a large number of objects.

One occlusion culling technique is portal-based occlusion culling [9]. Portal-based occlusion culling represents a game environment as a number of “rooms” connected via “portals”. Rooms may be any region in 3D space, and Portals are the boundaries between these regions. This representation can be most intuitively used indoors, where rooms are actual rooms and portal are doors or windows. These rooms and portals can then be represented as a graph where rooms are nodes and portals are edges. These rooms are then pre-processed to generate a ‘potentially visible set’ for a given room R. This potentially visible set is a collection of all rooms which are visible from at least one point in R. The set includes at least all nodes adjacent to R in the graph. Then, when rendering graphics, all rooms which are not in the potentially visible set can be skipped.

Visibility calculations are also useful for rendering shadows in 3D graphics. In computer graphics, an area is considered to be in shadow if it is not visible from a light source. As discussed previously, lines of sight are not adequate for this purpose, and so various techniques exist for the purpose of generating shadows.

Shadow Mapping [10] is the most common technique used for creating shadows cast from light sources in 3D. To create a shadow map, a z-buffer is rendered from the perspective of the light source. This z-buffer is then referred to as that light source's shadow map. For each pixel of a given surface, its distance from the light source is compared to the corresponding value on the shadow map, and if the shadow map has a smaller distance value then that pixel is rendered as being in shadow. A visual example of this process is shown in Figure 3. This technique is popular due to its performance, but attaining that performance requires rendering the shadow map at a reduced resolution, which results in somewhat blocky and imprecise shadows.

Shadow Volumes [11] are an alternative approach to creating shadow effects. A shadow volume is a representation of the space a shadow occupies, which is created using the shape of the object casting the shadow. This shadow volume is then used to determine whether a given surface is in shadow or illuminated. This results in precise (not blocky) shadows at better performance than high resolution shadow maps, but worse performance and higher complexity than lower resolution shadow maps.

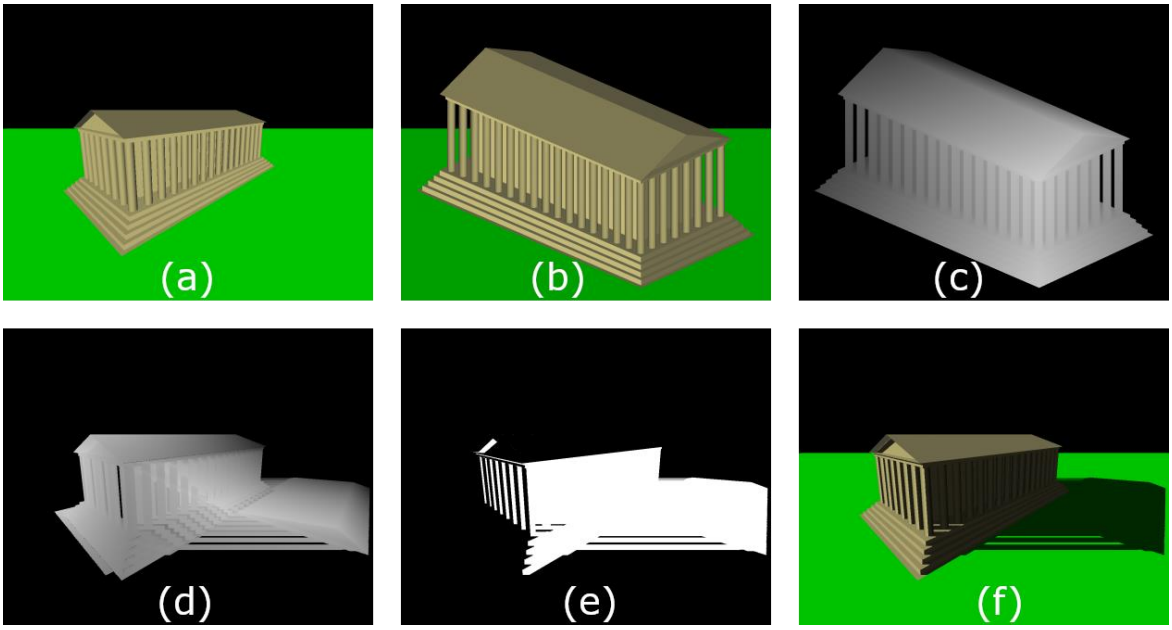

Figure 3: A demonstration of shadow mapping. The images represent: (a) the scene without shadows, (b) the scene from the perspective of the light source, (c) a visualization of the shadow map (lighter = closer), (d) the shadow map projected onto the scene, (e) a visualization of which pixels are in shadow (white = shadowed), and finally (f) the scene with mapped shadows applied.

Intuitively, it may seem that the above techniques could be suitable for calculating an FOV. Shadow mapping seems especially suited to this, as it is also a grid-based representation of visibility from a given source position. However, because these techniques are designed for GPUs, they are not suitable for processes such as the computation of actor behavior, which is performed by the CPU.

Algorithms designed for GPUs take advantage of the specialized nature of these processors and so they would be much slower if they ran on the CPU. Even assuming these algorithms could produce an FOV, they would not offer competitive performance when run on the CPU. The output from an FOV algorithm must be available to the CPU, as it will need to be referenced by other algorithms which run on the CPU, such as those that control actor behavior.

If algorithms designed for GPUs must be executed on the graphics card, and an FOV must be available to the CPU, could an FOV be computed on the graphics card, and then transferred to the CPU? Unfortunately, this will also not offer competitive performance due to the nature of CPU and graphics card interaction. Graphics cards accept rendering commands from the CPU, but do not necessarily execute them immediately as they are received. The CPU does not need to wait while the graphics card does work, so commands that the CPU issues enter a queue for the graphics card to execute [12]. The CPU has no access to this queue and cannot know what specific tasks the graphics card is currently performing. The graphics card may even choose to wait until the queue has many commands in it before starting execution. Because of this, if the CPU requests an FOV from the graphics card it has no assurance that the FOV calculation will start in a timely manner. The CPU may end up waiting until an entire frame buffer is finished before the FOV is computed and sent to it.

It should be noted that graphics cards can be used for more than just rendering graphics to a display. Technologies such as Nvidia's CUDA [13] have enabled graphics cards to run general-purpose algorithms. These algorithms are faster when run on the graphics card as they take advantage of its many GPUs. CUDA and similar technologies have enabled graphics cards to benefit specific areas of Computer Science in a way similar to how they benefit graphics rendering. Unfortunately, these technologies do nothing to address the concerns raised previously for FOV calculation using GPUs. This is because there is still no assurance of when FOV calculations will start, as the graphics card might be busy with rendering queued graphical operations.

Because of the above limitations, graphical visibility techniques are not able to effectively calculate an FOV and therefore cannot effectively solve the same problems which FOV solves. While there is certainly a conceptual overlap between FOV and these techniques, FOV calculation is a distinct subject with specific applications to processes which are executed by the CPU.

1.3 Existing FOV Algorithms

There are several algorithms for calculating FOV. However, these algorithms have not been formally analyzed to prove their correctness and to compute their complexities. They are designed by implementors whose primary goals were to produce a game, not research FOV. To the best of our knowledge this paper is the first effort to perform a systemic evaluation of these algorithms. Unless otherwise noted, the discussed algorithms are part of programming folklore, and so have no known author. We summarize the most popular FOV algorithms and the ones most relevant to our discussion of FOV.

Perhaps the most obvious method for determining the visibility of a cell is to trace a line from the FOV source to that cell and check for intersection with vision-blocking cells. Lines cast from the FOV source in a specific direction are a building block of all FOV algorithms. We refer to these lines as *visibility rays*, or simply *rays*.

When describing an FOV algorithm, a rule must be established which dictates the specific circumstances in which a grid cell is visible from the source cell. By following such a rule, we can determine whether a calculated FOV is correct. This is important as a grid discretizes 2D space and implementors may have differing ways that they wish to define visibility. We refer to these rules as *visibility definitions*. Consider a source cell S and destination cell D. We consider the three most common definitions of visibility:

Strict FOV defines that D is visible from S if a ray can be traced from the center of S to the center of D without intersecting any vision-blocking cells. Many implementors find this definition overly restrictive, as it results in many non-visible cells.

Shadowcast FOV defines that D is visible from S if a ray can be traced from the center of S to anywhere on D without intersecting any vision-blocking cells. This definition is popular as it results in more visible cells than strict FOV and is used by the popular Recursive Shadowcasting algorithm.

Permissive FOV defines that D is visible from S if a ray can be traced from anywhere on S to anywhere on D without intersecting any vision-blocking cells. This definition is useful because it is symmetrical. FOV symmetry is explained on the next page.

It should be noted that the shadowcast and permissive FOV definitions define ‘anywhere on a cell’ in a way which may not be intuitive. If a ray can reach any point on or in a cell, including just grazing its edge or corner, then that cell is visible. This behavior comes out of the desire to have vision-blocking cells be visible, to simulate seeing the face of a wall or similar vision-blocking object. Strict FOV also accomplishes this by always considering the destination cell to not be vision-blocking.

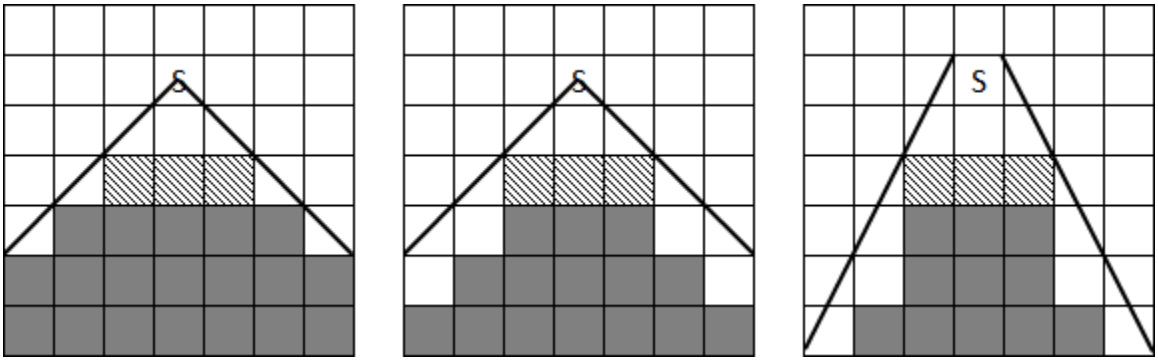

Figure 4: A simple grid with a correctly calculated FOV for strict (left), shadowcast (middle), and permissive (right). The rays which define the bounds of visible space are shown with bolded black lines.

Another important property of an FOV definition is FOV symmetry. An FOV is symmetrical if visibility, or lack of visibility, is always shared between any two cells (see Figure 5). This is important for some implementors. Of our three visibility definitions, strict and permissive are both symmetrical, and shadowcast is not.

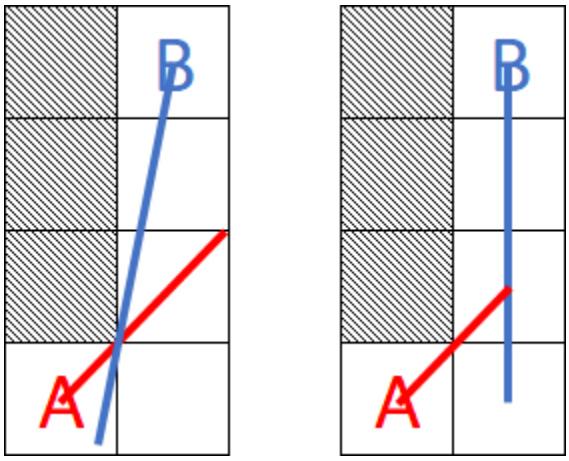

Figure 5: Lines of visibility for shadowcast FOV (left) and strict FOV (right). The asymmetry of shadowcast FOV is shown, as B can see A but not vice-versa.

The concept of visibility rays leads to an obvious first FOV algorithm: *Mass Ray FOV*. For every cell D within the grid, this algorithm traces a ray from the center of the FOV source cell to the center of D (see Figure 6). If that ray intersects no vision-blocking cells, then D is set to visible, otherwise it is set to not visible. Note that a ray is not considered to intersect with a cell if it just grazes its corner. This produces a correct FOV for the strict FOV definition.

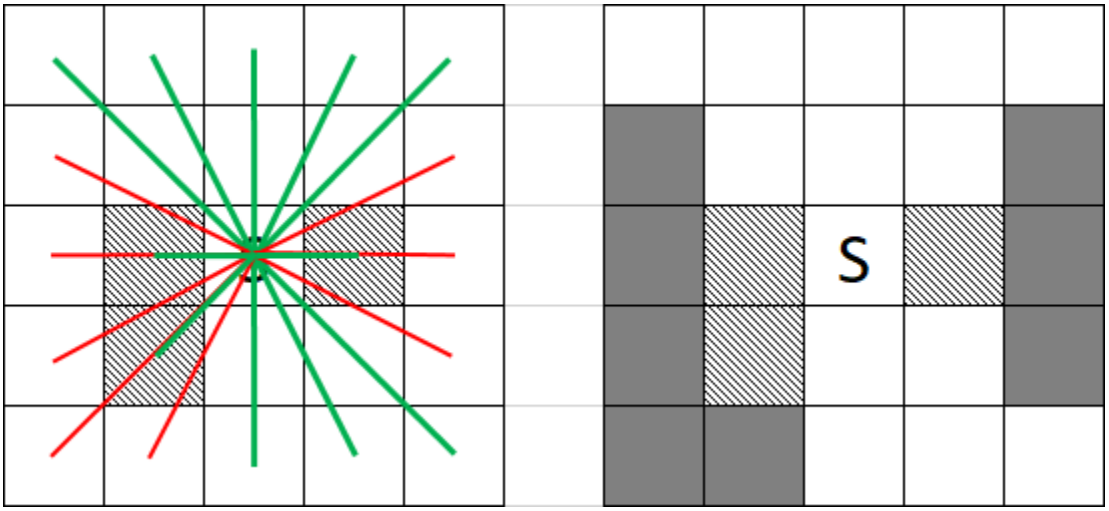

Figure 6: An example of Mass Ray FOV on a simple grid. Rays cast are on the left, and the calculated FOV grid is on the right. Unobstructed rays are shown in green, obstructed rays are shown in red.

Mass Ray FOV directly checks for visibility on a per-cell basis. This is in many ways equivalent to performing a line of sight check for each cell. This leads to very poor performance, as for an n*n grid, n^2 cells must be considered. The number of cells that a ray intersects increases linearly with n. This gives a total time complexity of $O(n^3)$ for Mass Ray FOV.

A straightforward optimization to Mass Ray FOV is to assign visibility values to many cells which intersect a given ray, instead of just the destination cell. An algorithm based on this approach is *Perimeter Ray FOV*. Perimeter Ray FOV casts a ray to every cell along the perimeter of the grid and sets to visible every cell intersected by the ray that is located between the source and the first vision-blocking cell the ray intersects.

Perimeter Ray FOV has much better performance than mass-ray FOV, as now only 4n-4 rays are cast, instead of n^2. This leads to an $O(n^2)$ time complexity. However, the algorithm does check some cells several times. Specifically, cells close to the FOV source will be set to visible numerous times. This is a large improvement over mass ray FOV, but shows that there are still ways to improve performance further.

This algorithm produces a result similar to the shadowcast FOV definition, but it is unfortunately not the same. It is common for there to be some portion of a cell which is visible from the source, but for the algorithm to not cast a ray in a direction which finds that visible portion of the cell. This results in Perimeter Ray FOV producing an incorrect output for the shadowcast FOV definition in many cases (see Figure 7). Increasing the number of perimeter rays (e.g. one ray cast to each of a cell's four corners) would reduce this inaccuracy but would not eliminate it.

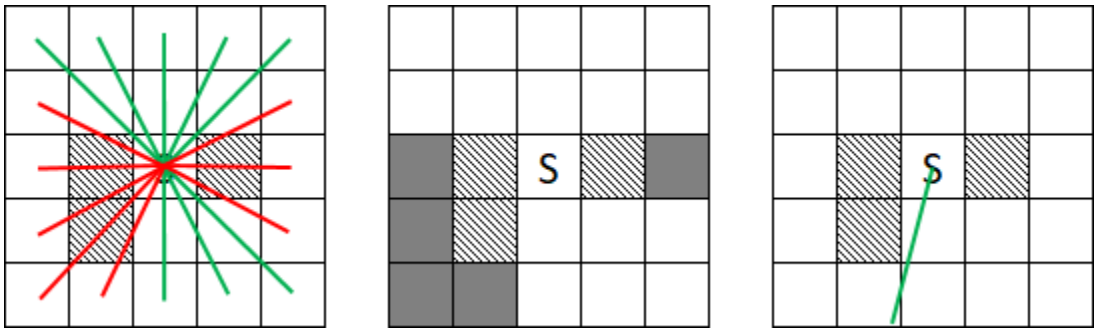

Figure 7: An example of Perimeter Ray FOV on a simple grid. Ray casts are on the left, the calculated FOV is in the middle. On the right there is an example of a visibility ray reaching a cell which was incorrectly set to not visible.

As the previous two algorithms have demonstrated, determining visibility by directly casting a ray to a cell is not ideal. This is not the only way to make use of rays however. More intelligent FOV algorithms cast rays to the edges of vision-blocking cells, so that these define the boundary between visible and non-visible space. These algorithms then traverse through visible cells in increasing order of distance from the FOV source, using these rays to determine when to stop traversal. This results in few rays being cast and reduces the number of duplicated cell traversals.

One algorithm based on this approach is *Recursive Shadowcasting* by Björn Bergström [14]. This algorithm computes the FOV in 8 iterations, each handling one 45 degree octant of the FOV grid (see Figure 8). The algorithm described below is written for octant 1, and mirroring operations are performed when accessing cells to process the remaining 7 octants.

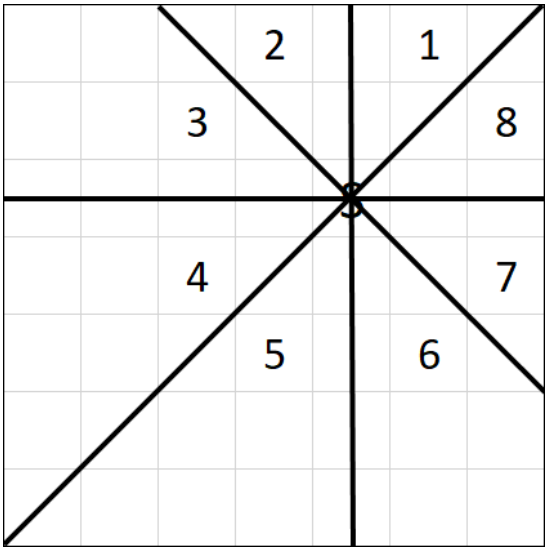

Figure 8: An example of Shadowcasting octants, numbered 1-8.
Note that the octants are centered on the FOV source, not the center of the grid.

The algorithm for octant 1 processes the cells by rows moving away from the FOV source. Each row starts at the cell touching the leftmost part of the octant and ends with the cell touching the rightmost part. When vision blocking cells are encountered, the algorithm moves the left or right edge of the octant inward for all future rows (see Figure 9). This correctly handles visibility shrinking as vision-blocking cells are encountered. If the blocking cell is in the middle of a visible region, the algorithm recursively calls itself to handle the two new visible regions.

Figure 9 shows an example of Recursive Shadowcasting. The algorithm first processes rows 1 to 3 (shown with blue arrows) without encountering any vision-blocking cells. On row 4, two vision-blocking cells are encountered, splitting the visible region in two and causing the algorithm to recursively call itself. The recursive call then processes the visible region to the left (shown in pink) while the main iteration of the algorithm continues processing the visible region to the right. Recall that even if a ray just grazes the edge of a cell, that entire cell must be set to visible.

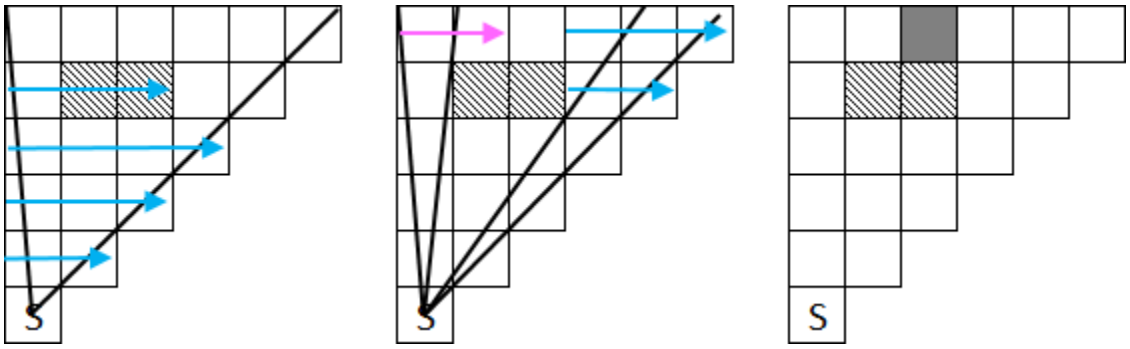

Figure 9: An example of Recursive Shadowcasting on a simple grid.

The algorithm for octant 1 is given below:

```
Algorithm: recursiveShadowcasting (G, S, T, left, right)
Input: FOV grid G, source cell S, distance integer T, visibility rays left & right
When first called: cells in G are not visible, T is 1, left & right are edges of the octant
Result: cells in G which are visible from S are set to visible

boolean inBlocking = false

for each row R in the part of G between left and right,
                    starting from T rows away from S {

    increment T by 1

    for each cell C in R, starting from the cell intersecting left,
                          and ending with the cell intersecting right {

        set C to visible

        if C is vision-blocking and inBlocking is false then {
            inBlocking = true
            recursiveShadowcasting (G, S, T, left,
                ray from center of S to top-left corner of C)

        } else if C is not vision-blocking and inBlocking is true then {
            inBlocking = false
            left = ray from center of S to bottom-left corner of C
        }
    }

    if inBlocking is true then return
}
```

Another algorithm is *Precise Permissive FOV* by Jonathon Duerig [15]. This algorithm produces output for the permissive FOV definition, which makes it an important alternative to Recursive Shadowcasting for implementors that desire FOV symmetry.

The algorithm has a similar approach to Recursive Shadowcasting. It processes cells contained within a left and right ray, changes the rays based on encountered vision-blocking cells, and makes recursive calls when visibility is split. It differs from Shadowcasting in a few keys ways however:

- Precise Permissive FOV operates on quadrants of the FOV grid, rather than octants. This means fewer mirroring operations are required.
- Instead of processing the cells by rows, the algorithm uses diagonal lines (see Figure 10).
- The algorithm performs a recursive call for every blocking cell which is fully contained between the left and right rays, whereas Shadowcasting performs a recursive call once for each group of consecutive blocking cells in a row.
- The algorithm casts rays from the edges of the source cell, instead of from the center, to match the permissive FOV definition.

Figure 10 shows an example of Precise Permissive FOV. First the algorithm moves though diagonals until it encounters a vision-blocking cell that is fully contained between the left and right rays. Then, the algorithm recursively calls itself, each iteration now has its own left and right ray which bound the set of cells that it processes. The final computed FOV is shown on the right.

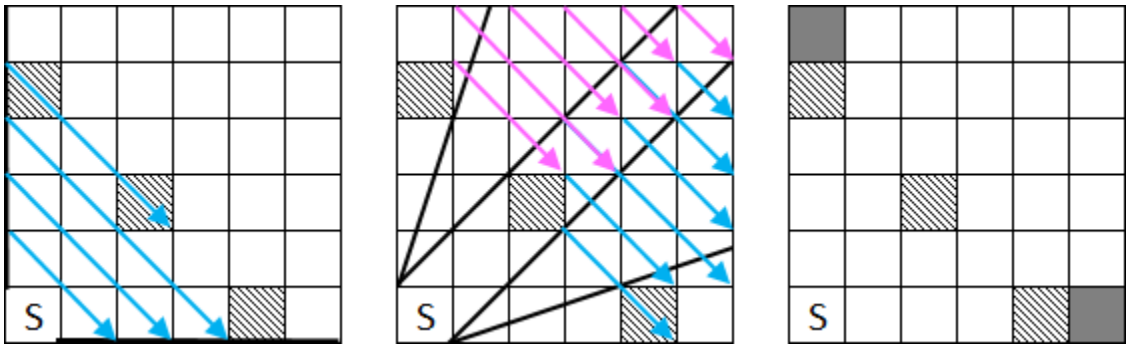

Figure 10: An example of Precise Permissive FOV on a simple grid.

The algorithm for quadrant 1 is given below:

```
Algorithm: precisePermissiveFOV (G, S, T, left, right)
Input: FOV grid G, source cell S, distance integer T, rays left & right
When first called: cells in G are not visible, T is 1, left & right are edges of the quadrant
Result: cells in G which are visible from S are set to visible

for each diagonal line L in the part of G between left and right,
                                    starting from T lines away from S {

    increment T by 1

    for each cell C in L, starting from the cell intersecting left,
                                    and ending with the cell intersecting right {

        Set C to visible

        if C is vision-blocking then {
            if C is the only cell in L then {
                return
            } else if C is the first cell in L then {
                left = ray from top-left corner of S to bottom-right corner of C
            } else if C is the last cell in L then {
                right = ray from bottom-right corner of S to top-left corner of C
            } else {
                precisePermissiveFOV (G, S, T, left,
                    ray from bottom-right corner of S to top-left corner of C )

                left = ray from top-left corner of S to bottom-right corner of C
            }
        }
    }
}
```

1.4 Analysis of Existing FOV Algorithms

We now analyze the performance of the four FOV algorithms discussed in Chapter 1.3. This analysis will help highlight the performance characteristics of these algorithms, so that we may then propose improvements. The existing research on FOV algorithms is very limited. To the best of our knowledge, there is a single study of FOV algorithms made by Jice in 2009 [16]. This study tests several FOV algorithms in a variety of cases but has several shortcomings which we address below.

Firstly, Jice ran each FOV algorithm multiple times with the same input and reported statistics on the performance results. Repeatedly running an FOV algorithm may be problematic because the performance of each run of the algorithm will be affected by the CPU cache. The CPU cache is a relatively small amount of extremely fast memory which the computer attempts to populate with recently referenced data. The FOV algorithms which we have described in Chapter 1.3 perform large numbers of memory accesses, and so having some or all of the FOV grid stored in the cache will substantially enhance their performance. However, in a computer game the FOV will be computed as needed, in between many other computations, and so the FOV grid would not be consistently stored in the cache. Running FOV algorithms many times without ensuring the grid is not present in the cache will result in unrealistic performance data. In our analysis each run of an FOV algorithm uses an entirely new grid. This ensures that the CPU cache will be filled with old FOV grids which the algorithms are no longer using, thus effectively clearing it.

The work in [16] also compares the differences in the visibility grid computed by the tested FOV algorithms. Such an analysis may be helpful to implementors who wish to decide on a visibility definition, but it is not useful for comparing the performance of FOV algorithms. Jice assigns a score to each algorithm based on its visual output, but even Jice admits that the scoring system is largely arbitrary. This is why in our analysis we use a limited number of visibility definitions and do not attempt to rank them, so as to focus our analysis on the properties of the algorithms.

The work in [16] uses FOV algorithm implementations present in the Doryen library [12] (also referred to as LIBTCOD). This library provides many game related functions, but it is not specifically focused on providing a lightweight or efficient implementation of FOV algorithms. In [16] no comparisons are made between the implementations in the Doryen library and other implementations of FOV algorithms. Because of this, the results presented in [16] may be influenced by inefficiencies present in the Doryen library. We compare the Doryen library's implementation of FOV algorithms to our own implementations to determine if such inefficiencies exist.

Finally, while [16] presents overall performance statistics for each FOV algorithm studied, it does not examine what causes differences in performance between the algorithms. Some algorithms are shown to have superior performance in certain situations, but [16] makes no attempt to determine why. We will choose test cases which highlight specific performance characteristics in order to better understand differences between each FOV algorithm.

We tested all four algorithms described in Chapter 1.3. We used our own implementations of each algorithm as well as the Doryen implementation of Perimeter Ray FOV, Recursive Shadowcasting, and Permissive FOV. The Doryen library does not include an implementation of Mass Ray FOV. For all algorithms we did not consider the time spent initializing the FOV grid to not visible.

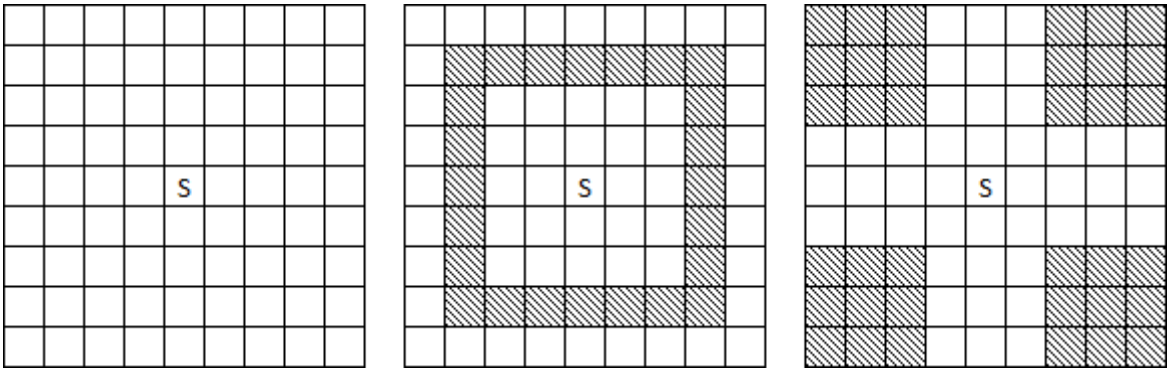

Figure 11: Examples of each testing environment in a simple 9x9 grid

We tested using three environments, shown in Figure 11. Each environment was tested at grid sizes ranging from 128*128 to 4096*4096. These sizes cover a realistic range of values which game implementors may choose to use. Square grids were chosen as they are the most common environment shape used by games which use FOV. This range includes the current grid size of League of Legends [2] (128*128) and the size that League of Legends visually upscales its grid to (512*512) [4].

Note that while monitors commonly have a display resolution below 4096*4096, game environments may be much larger in size than the area visible on screen. This means that if the FOV grid has a size greater than the monitor's resolution, only a region of the grid will be visible.

The first environment is an entirely empty FOV grid, which is the worst-case with respect to the number of cells which must be set to visible. This environment is purely a test of how efficiently each algorithm assigns visibility statuses to cells.

The second environment is a 5x5 enclosed space with the FOV source in its center. This is almost a best-case scenario with respect to the number of cells which must be set to visible. This effectively tests how each algorithm performs when the number of cells which are visible is very small, regardless of the size of the FOV grid.

The third environment is a mostly enclosed FOV grid with a three cell wide corridor extending in each cardinal direction. This environment is designed to be close to a worst-case scenario for Recursive Shadowcasting and Permissive FOV when compared to other FOV algorithms such as Perimeter Ray FOV.

Table 1: Mean running times of our algorithm implementations in environment 1

Grid Size	Mass Ray	Perimeter Ray	Shadowcasting	Permissive
128*128	5,160 µs	257 µs	75 µs	95 µs
256*256	37,965 µs	1,004 µs	343 µs	317 µs
512*512	291,654 µs	4,161 µs	1,744 µs	2,022 µs
1024*1024	2,319,475 µs	21,444 µs	11,720 µs	9,341 µs
2048*2048	19,257,575 µs	112,544 µs	49,381 µs	47,443 µs
4096*4096	178,968,245 µs	636,215 µs	242,105 µs	394,429 µs

Table 2: Mean running times of Doryen implementations in environment 1

Grid Size	Perimeter Ray	Shadowcasting	Permissive
128*128	785 µs	299 µs	728 µs
256*256	3448 µs	1,204 µs	2,911 µs
512*512	13,993 µs	4,566 µs	11,913 µs
1024*1024	65,036 µs	19,592 µs	50,597 µs
2048*2048	264,784 µs	80,121 µs	180,286 µs
4096*4096	1,065,233 µs	339,157 µs	710,267 µs

The $O(n^3)$ time complexity of Mass Ray FOV is clearly shown in Table 1: the algorithm is substantially slower than all others and its running time increases by roughly a factor of 8 each time the dimensions of the FOV grid are doubled. The other algorithms all demonstrate an $O(n^2)$ time complexity, by roughly quadrupling their running time when the dimensions of the grid double.

Shadowcasting and Permissive FOV have similar performance, with Shadowcasting performing best at high grid sizes. Both Shadowcasting and Permissive FOV are much faster than Perimeter Ray FOV in all cases. This difference is explained by the lower number of duplicated cell assignments performed by Shadowcasting and Permissive FOV.

The Doryen library implementations of these algorithms exhibit the same $O(n^2)$ time complexity but they are slower than our implementations. In particular, the Doryen implementation of Permissive FOV is very slow, making it appear much worse than Recursive Shadowcasting.

Table 3: Mean running times of our algorithm implementations in environment 2

Grid Size	Mass Ray	Perimeter Ray	Shadowcasting	Permissive
128*128	519 µs	18 µs	1.2 µs	10.2 µs
256*256	1,953 µs	35 µs	1.2 µs	10.2 µs
512*512	7,711 µs	77 µs	1.2 µs	10.2 µs
1024*1024	30,999 µs	139 µs	1.2 µs	10.2 µs
2048*2048	130,333 µs	280 µs	1.3 µs	10.3 µs
4096*4096	502,211 µs	555 µs	1.5 µs	10.8 µs

Table 4: Mean running times of Doryen implementations in environment 2

Grid Size	Perimeter Ray	Shadowcasting	Permissive
128*128	112 µs	2 µs	50 µs
256*256	356 µs	2 µs	183 µs
512*512	1,633 µs	1.9 µs	761 µs
1024*1024	8,035 µs	2 µs	3835 µs
2048*2048	33,064 µs	2.1 µs	27.9 µs
4096*4096	144,008 µs	2.3 µs	29.4 µs

While Mass Ray FOV and Perimeter Ray FOV do benefit from only having to assign visibility statuses to cells in a small area, their running times still increase as the grid size increases. This is because the number of rays that these algorithms cast is dependent on the grid size and is not affected by the vision-blocking cells.

The running times of Recursive Shadowcasting and our implementation of Permissive FOV remain constant as the size of the grid increases because the block of visible cells which they traverse does not change. For these algorithms an arbitrarily large FOV grid will produce almost the same running time as the smallest possible grid which is able to contain all visible cells.

The Doryen implementation of Permissive FOV exhibits unusual behavior in this environment. Its running time increases as the grid size increases up to 1024*1024 because it allocates and initializes an amount of memory that depends on the size of the FOV grid. The more memory that is allocated, the larger the running time of the algorithm becomes. However, the amount of this memory which the algorithm actually uses depends on the number of visible cells, and so most of the memory is unused in this environment.

At grid sizes of 2048*2048 and above the running time of Doryen Permissive FOV decreases. We suspect that this is caused by ‘lazy allocation’, which is a technique where a computer will only allocate or initialize memory when a program actually uses it. At higher grid sizes the amount of unused memory is large enough for lazy allocation to trigger, which causes a reduction in the running time of the algorithm.

Table 5: Mean running times of our algorithm implementations in environment 3

Grid Size	Mass Ray	Perimeter Ray	Shadowcasting	Permissive
128*128	908 µs	37 µs	19 µs	59 µs
256*256	4,994 µs	115 µs	56 µs	127 µs
512*512	16,758 µs	298 µs	225 µs	327 µs
1024*1024	72,343 µs	845 µs	608 µs	852 µs
2048*2048	333,513 µs	2,545 µs	2,066 µs	2,960 µs
4096*4096	1,450,916 µs	9,470 µs	7,588 µs	8,857 µs

Table 6: Mean running times of Doryen implementations in environment 3

Grid Size	Perimeter Ray	Shadowcasting	Permissive
128*128	166 µs	351 µs	533 µs
256*256	482 µs	1,207 µs	2205 µs
512*512	1,697 µs	4,922 µs	8,305 µs
1024*1024	8,847 µs	20,103 µs	32,638 µs
2048*2048	34,033 µs	72,732 µs	115,242 µs
4096*4096	161,096 µs	306,555 µs	471,515 µs

Environment 3 was chosen specifically to make Shadowcasting and Permissive FOV perform a large number of ray casts. Both Shadowcasting and Permissive FOV have much worse performance when compared to Perimeter Ray FOV in this environment than in environment 1. The more efficient cell traversal of Shadowcasting and Permissive FOV is less effective here, as the performance of these algorithms is reduced due to the number of rays they must cast.

The Doryen library implementations of Shadowcasting and Permissive FOV performed very poorly here. Clearly the Doryen implementation of these algorithms is managing rays and ray casting in an inefficient manner, as the algorithms are slower than our implementation by a factor of up to 50. The relative difference in running time increases as grid size increases, because a higher size will result in more rays being cast.

From these three test cases, we can make the following conclusions:

In terms of running time Recursive Shadowcasting is the most efficient algorithm, however Permissive FOV is competitive with it in most cases. Perimeter Ray FOV generally performs poorly but becomes competitive with Shadowcasting and Permissive FOV when those algorithms must cast many visibility rays. Mass Ray FOV performs extremely poorly in all cases and is clearly not a useful FOV algorithm.

The study given in [16] is significantly affected by inefficiencies present in the Doryen library. The Doryen implementations of FOV algorithms perform almost universally worse than our own, sometimes by large margins, and in some cases exhibit unusual behavior which is inconsistent with how the FOV algorithms work. While [16] would be useful to an implementor who plans to work with the Doryen library, it is not a useful experimental evaluation of the FOV algorithms themselves.

Existing FOV algorithms are very well suited to environments with few visible cells but struggle when many cells are visible. The most efficient algorithms are almost completely unaffected by the size of the FOV grid in environment 2 but have running times which scale quadratically with grid size in environments 1 and 3. This makes sense based on the design of Shadowcasting and Permissive FOV, as these algorithms only scan cells which they will set to visible. As a result their performance is primarily dependent on how many cells will be visible in an environment.

As performance is most dependent on cell visibility assignments, the algorithms do not scale well to higher grid sizes. In realtime applications such as computer games, an algorithm which takes even a few milliseconds to complete may have a negative impact on the gameplay experience. In the worst-case scenario of environment 1, Recursive Shadowcasting becomes problematic at grids of size 512*512 and would certainly be unusable at grids of size 1024*1024 and above. A better FOV algorithm must improve the process of assigning visibility values to cells, either by assigning fewer visibility statuses, or assigning statuses more efficiently.

1.5 Correctness Issues with Recursive Shadowcasting

While developing and testing our own implementation of the Recursive Shadowcasting algorithm, we noticed certain cases where the algorithm as described in [14] produces incorrect output. We describe this issue and a solution below.

When Recursive Shadowcasting traverses a row of the grid, it scans all cells in that row which intersect the area defined by its visibility rays. If a cell even just grazes a visibility ray, it must be set to visible. However, in certain cases a visibility ray will intersect a vision-blocking cell and become blocked. In the example provided in Figure 12, the visibility ray on the right intersects a vision-blocking cell at point A and hence it should not be extended beyond that point. However, the algorithm as described in [14] would continue the ray all the way to point B and thus incorrectly set cell C to visible. It is therefore important for an implementation of Recursive Shadowcasting to check for intersections and handle them appropriately. In the example provided in Figure 12, when traversing the topmost row the algorithm must stop at the cell intersecting point A, instead of the cell intersecting point B.

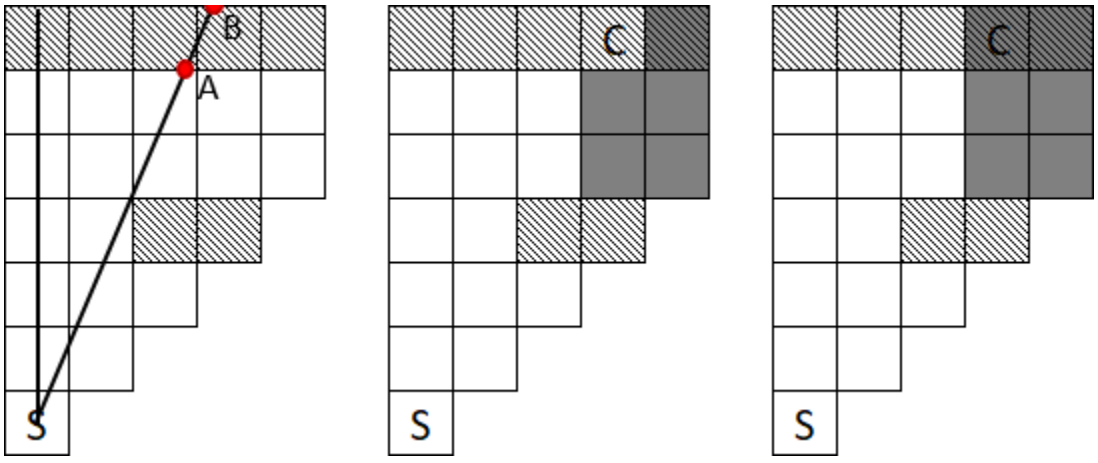

Figure 12: An example of Recursive Shadowcasting producing incorrect output. The visibility rays are shown on the left. The resulting FOV output is shown in the center. The correct output is shown on the right.

This error will sometimes cause a single cell to be incorrectly set to visible. To the best of our knowledge no existing implementation of Recursive Shadowcasting addresses this problem. This is likely because the error will only affect one vision-blocking cell which is at the edge of a visible region, and so the error is not easily noticed.

This error can be fixed by modifying the Recursive Shadowcasting algorithm: Rows are traversed as described in Chapter 1.3, except an additional check occurs when the final cell of a row is reached. When the algorithm is considering the final cell, before assigning it a visibility status, it checks if the previous cell (i.e. the second to last cell in the row) is vision-blocking. If the previous cell is vision-blocking, then the algorithm checks if the visibility ray intersects it, and if so the algorithm does not assign a status of visible to the final cell, but ends its current recursive iteration. This check is not performed if the row being scanned only includes one cell.

Note that while our example of this error uses octant 1, where the algorithm traverses by rows from left to right, this error may occur in any octant. Because of this, this check must be performed while calculating the FOV for the other 7 octants as well. In some octants this traversal will be by columns instead of rows.

Adding this check to the Recursive Shadowcasting algorithm does not significantly affect performance and ensures that the algorithm always produces correct output according to the definition of shadowcast visibility. Our implementation of Recursive Shadowcasting that was tested in Chapter 1.4 includes this check.

Chapter 2

2 Improving FOV Calculation

This chapter covers our first new approach to FOV calculation: an FOV algorithm based on a compact and efficient representation of vision-blocking cells.

2.1 Grouping Vision-Blocking Cells

The FOV algorithms which we have discussed perform two essential operations: determining which cells are visible from the source, and storing this information in the FOV grid. For both operations, the algorithms scan the entire FOV grid and therefore have time complexities which at best depend linearly on the number of grid cells. This results in poor performance at large grid sizes, as the number of cells depends quadratically on the size of the grid.

We know that each time an FOV algorithm is run the visibility status of some cells must be different, as otherwise there would be no reason to calculate a new FOV. However, the positions of vision-blocking cells within the grid can be expected to change infrequently, or not at all. Therefore, a compact representation for vision-blocking cells could be computed once and used for many FOV calculations.

We can process vision-blocking cells in an efficient manner by grouping adjacent vision-blocking cells. The time complexity of determining which areas are visible and which are not will then depend on the number of vision-blocking groups, and not necessarily on the number of individual vision-blocking cells. In most environments increasing the FOV grid size will increase the number of vision-blocking cells but will increase the number of vision-blocking groups by a smaller amount. Therefore, as grid size increases an algorithm whose performance depends on the number of vision-blocking groups will likely have better performance than an algorithm with performance depending on the number of vision-blocking cells.

This grouping of vision-blocking cells allows us to assign visibility statuses to cells efficiently as well. Existing FOV algorithms assign visibility status on a per-cell basis. By grouping vision-blocking cells and computing the area of the FOV grid that the group occludes, we can determine the visibility of a large region of the grid at once. We can then store visibility statuses of cells in this region in whichever order we like.

This is important because, as previously discussed in Chapter 1.4, the efficiency of algorithms which frequently access memory is affected by the CPU cache. In addition to storing recently accessed data the CPU cache will also store data that is located nearby, this is called *spatial locality*. If a program accesses memory in a manner that takes advantage of this property of caching, it will be significantly faster, and is said to be taking advantage of spatial locality.

Typically cells in an FOV grid will be laid out from left to right and top to bottom in adjacent memory locations. In other words, the first row of cells in the grid would be stored from left to right in consecutive memory locations, then the second row would be stored immediately after, and so on. This means that if we assign visibility status to cells by rows from left to right we will take maximal advantage of spatial locality, which will significantly accelerate the process of writing visibility statuses to cells (see Figure 13).

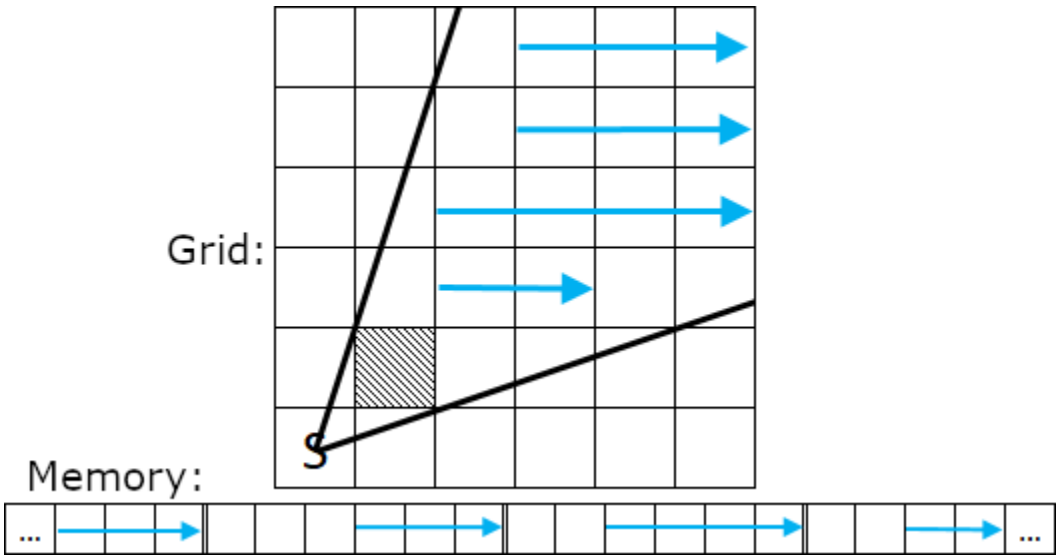

Figure 13: How cells within the region occluded by one vision-blocking cell would be assigned not visible status. Cells are shown in the grid above and are shown laid out in memory below. A double line in memory indicates when a new row begins.

Some existing FOV algorithms make some use of spatial locality, but only in a limited way. Mass Ray FOV and Perimeter Ray FOV both access grid cells which intersect the rays that they cast and so will only incidentally benefit from spatial locality when those rays happen to intersect cells in the same order in which they are stored in memory. Permissive FOV traverses the cells of the grid by diagonals, and so it does not significantly benefit from spatial locality either.

Recursive Shadowcasting does take advantage of spatial locality however. As already discussed in Chapter 1.3, Recursive Shadowcasting moves by along the cells of the grid by rows from left to right when processing Octant 1. However, when the Recursive Shadowcasting algorithm is mirrored on the other 7 octants the cell traversal order changes as well (see Figure 14). For four of the eight octants the algorithm will traverse cells by columns and will therefore not take advantage of spatial locality.

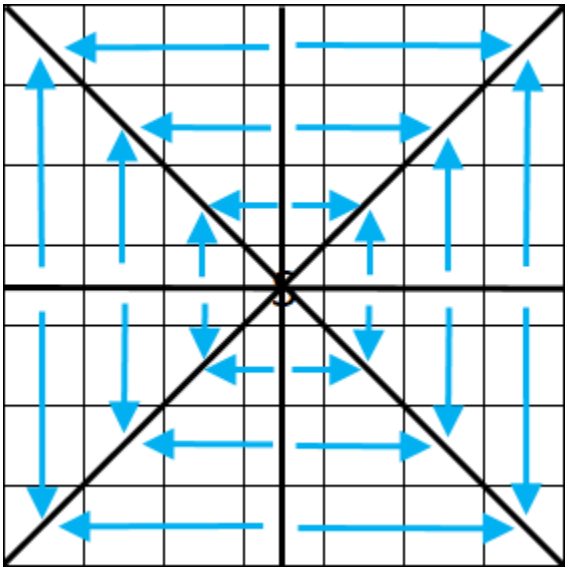

Figure 14: Cell traversal order of Recursive Shadowcasting for each octant.

Additionally, while Recursive Shadowcasting will make use of spatial locality when it traverses by rows, this traversal will be split when the algorithm encounters vision blocking cells and recursively calls itself. This splitting will reduce the spatial locality of the algorithm.

2.2 Splitting Vision-Blocking Groups into Rectangles

As shown in Figure 15, adjacent vision-blocking cells form vision-blocking rectilinear polygonal regions. If any holes are inside of these polygonal regions, we can fill them with vision-blocking cells without affecting the resulting FOV. A rectilinear polygon without holes is called a simple rectilinear polygon. These simple polygons can be dissected into rectangles. We show an FOV algorithm that can use these vision-blocking rectangles to efficiently determine which regions of the grid are non-visible.

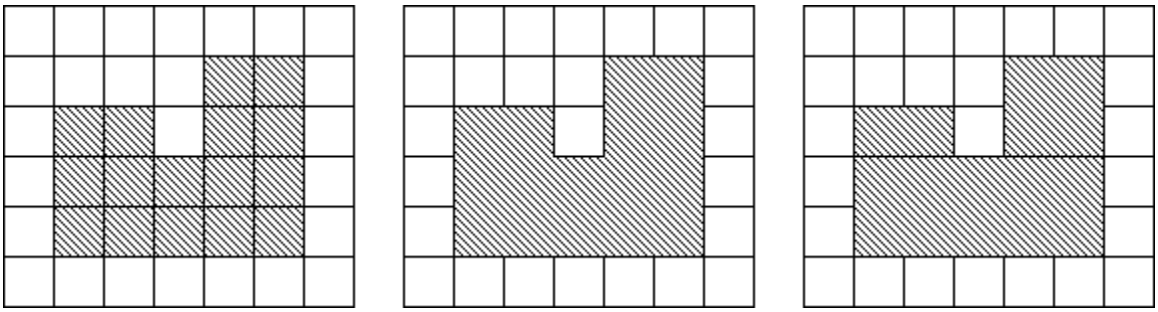

Figure 15: Vision blocking cells (left) being transformed into a rectilinear polygon (center), and then a set of rectangles (right).

There are many ways to partition rectilinear polygons into rectangles, but we want to do so in a way which minimizes the number of rectangles. This is a well-studied problem which can be solved in polynomial time. We summarize one way to partition simple rectilinear polygons into a minimal number of rectangles, first described in [18].

All vertices of a rectilinear polygon can be separated into two categories: concave and convex. A vertex is concave if its internal angle is 270 degrees and convex if that angle is 90 degrees (see Figure 16 (a), where concave vertices are highlighted in red). The internal angle of a vertex is the angle inside the polygon which is formed by the two edges touching that vertex. A horizontal or vertical line which is entirely within a rectilinear polygon and which connects exactly two concave vertices is a *chord* of that polygon (see Figure 16 (b), where chords are labelled and colored).

We construct a graph based on all the chords of a given rectilinear polygon: Each chord is a node of the graph, and chords which intersect each other are connected by an edge. All horizontal chords can only intersect vertical chords, and vice-versa, which means the resulting graph will be bipartite. A bipartite graph is a graph where the nodes can be separated into two groups, such that no nodes in the same group are connected by an edge.

We then determine a maximum independent set of the bipartite graph, which is the largest set of nodes no two of which are adjacent to each other (see Figure 15 (c), which shows a bipartite graph of chords with a maximum independent set highlighted in red). The problem of finding a maximum independent set of an arbitrary graph is NP-hard, however for bipartite graphs a maximum independent set can be found in polynomial time, such as with an algorithm by Hopcroft and Karp [19]. The polygon is then cut along the chords that are part of the maximum independent set, which creates the smallest number of rectilinear polygons that do not have any chords [18] (see Figure 16 (d), which shows only the chords that cut the polygon).

Finally, these chord-less rectilinear polygons are partitioned into rectangles using their concave vertices. For each concave vertex, a polygon is cut along a horizontal or vertical line which extends from that vertex to the other side of the polygon (see Figure 16 (e), which shows these final cuts in green). The choice between horizontal or vertical is arbitrary, either will result in the same number of rectangles. The polygons which result from this final process will all be rectangles, and as shown in [18] it is not possible to partition the original polygon into fewer of them.

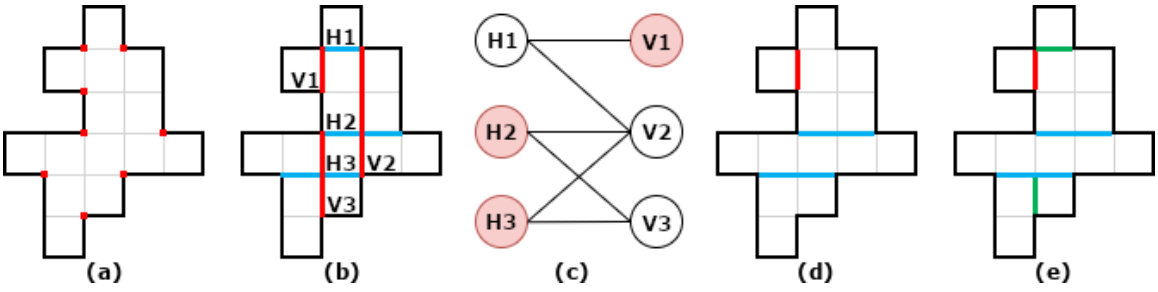

Figure 16: A figure showing the segmentation of a rectilinear polygon.

Having grouped vision blocking cells into rectangles, we now discuss how to use them to calculate the FOV. Of the four corner points of a rectangle, two are relevant to determining the visible space from a given FOV source. We refer to these two points as the *relevant points* of a rectangle. The relevant points of a rectangle are the two points which are farthest apart from each other, among all points which are not occluded behind that rectangle (see Figure 17). Rays traced from each relevant point away from the FOV source define the boundary between space which is occluded behind the rectangle, and space which is not. The area between, but not including, both rays and the visible faces of a rectangle is all space which is not visible because of that rectangle.

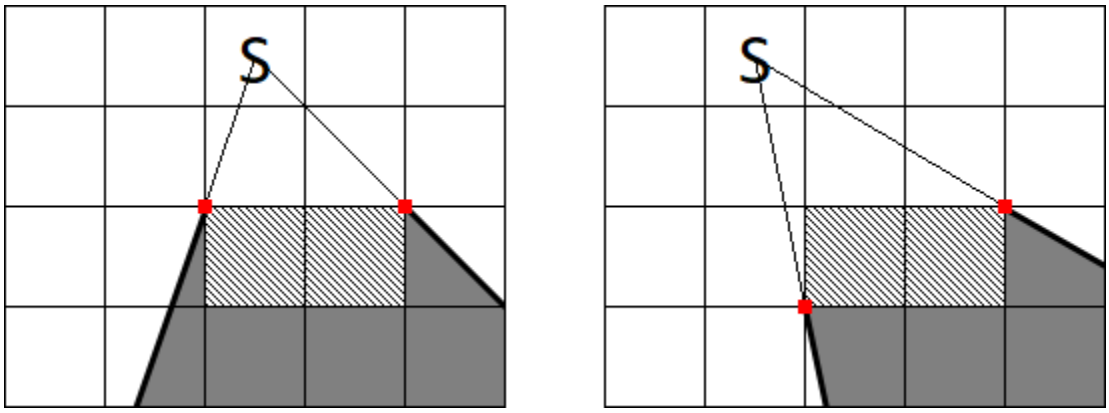

Figure 17: Rays cast from relevant points for two separate source positions. The area occluded by the rectangle is darkened. Two corners of the rectangle are occluded on the left, while only one is on the right. Relevant points are highlighted in red.

We use rectangles to represent groups of vision-blocking cells instead of other more complex polygons because rectangles are the only convex polygons which can be accurately represented on a grid. Convex polygons are useful for representing vision-blocking cells because, as we have shown, they can easily be processed to determine relevant points and grid cell visibility.

2.3 Storing Rectangles in a Quadtree

Vision-blocking rectangles need to be stored in a data structure that allows us to process them efficiently. We must use a data structure which allows us to represent rectangles within 2D space. Such data structures are called spatial data structures. The spatial data structure we have chosen to use is the quadtree [20].

Each node of a quadtree represents a region of the FOV grid and it contains all rectangles that intersect that region. The quadtree has a parameter L that bounds the minimum number of rectangles that must intersect the region represented by a node that would force that region to be split into four quadrants of the same size. The first node of a quadtree represents the entire grid and is referred to as the root node. If more than L rectangles are within the grid, the grid is split and the root is given four child nodes which each represent a quadrant of the grid (see Figure 18). If a node represents a region of the grid that intersects more than L rectangles, that node is given 4 children each representing one quadrant of the region represented by the node. Nodes are added to the quadtree and regions are split into smaller regions until all regions intersect at most L rectangles.

A node which has children is referred to as an internal node, and a node with no children is a leaf node. Rectangles are stored within leaf nodes; each internal node stores its four child nodes. It should be noted that a rectangle can be contained within multiple leaf nodes, as it can intersect more than one leaf node's region.

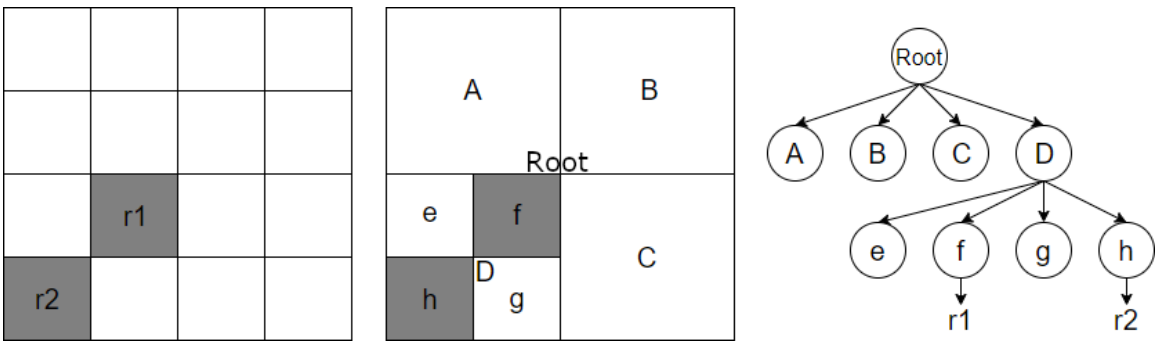

Figure 18: A grid with two vision-blocking cells (left), its quadtree representation with L=1 (right), and the space represented by each node (middle).

Quadtrees can be built and updated quickly. The ability to efficiently update a quadtree is important, as game implementors may want to be able to change vision-blocking terrain as a game progresses without having to rebuild the entire spatial data structure.

The algorithm for building a quadtree from a grid of vision-blocking cells is shown in pseudocode below:

```
Subroutine: buildQuadtree(N, G, L)
Input: quadtree node N, FOV grid G, bound L
When first called: N will be the root node representing the entire FOV grid
Output: N will be the root of a quadtree that contains all rectangles in its region

If the region that N represents intersects more than L rectangles then {
    Add 4 child nodes to N, each representing one quadrant of N's region.
    for each child node C of N {
        buildQuadtree(C, G, L)
    }
} else {
    Store in N all rectangles in G which intersect N's region
}
```

2.4 Handling Cases Involving Multiple Rectangles

The description in Chapter 2.2 of how to determine the region occluded by a rectangle only considers one rectangle in isolation. However, in order to accurately and quickly calculate the entire FOV there are certain cases where we must consider multiple rectangles. We describe two cases where several rectangles must be considered when processing a vision-blocking rectangle. The first case is important for correctness and the second is important for performance.

Note that when we state that a cell is within the occluded region of a rectangle, we mean that the entire cell is within that occluded region. This is consistent with the shadowcast FOV definition.

Consider two adjacent vision-blocking rectangles which share part of a side (see Figure 19). There are some cells which are not fully contained in either of their occluded regions. If these rectangles are processed individually these cells will remain as visible, which will result in an incorrect output.

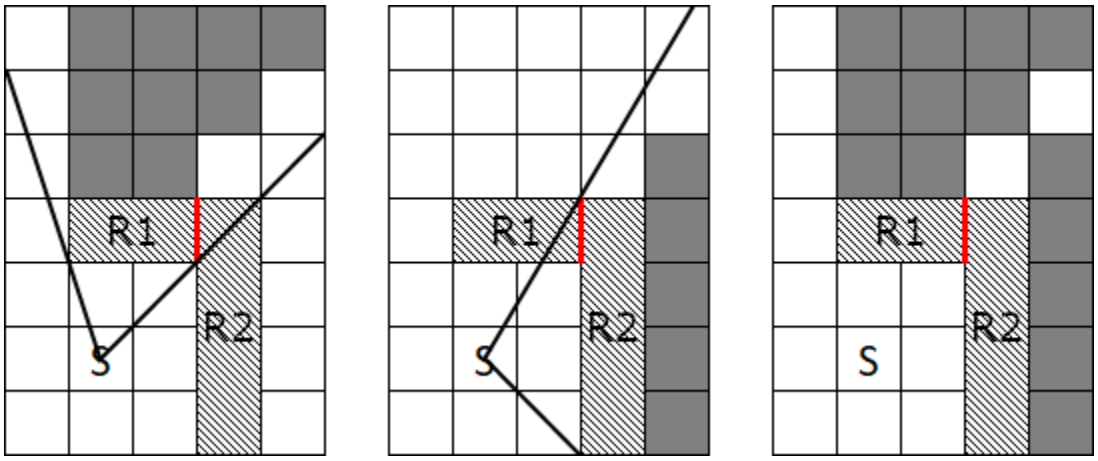

Figure 19: An example of two adjacent rectangles with their touching sides highlighted in red. The area occluded behind each rectangle is shown on the left and center, and the incorrect FOV which will result from considering them individually is shown on the right.

To resolve this issue, we extend the size of adjacent vision-blocking rectangles, so that they overlap. We refer to this process as *extending* a rectangle. This will ensure that no occluded cells will be set as visible. Note that a rectangle is only extended for the purposes of visibility calculation and this does not affect the size of the rectangle stored in the quadtree.

To extend a rectangle R1, we first determine its relevant points. For each relevant point P, we check if P also belongs to any other rectangle. This check can be efficiently performed using the quadtree. We recursively traverse the nodes of the quadtree which represent those regions containing P until we reach a leaf node. Upon reaching a leaf node we check all rectangles stored in it (except R1) to see if any of them contains P. If some rectangle R2 contains P, we check if R2 occludes P, i.e. if P is behind R2. If P is not occluded, we extend the size of R1 by one row or column so that it overlaps with R2.

In the example in Figure 19, we extend the rectangle R1 by one column to the right as its rightmost relevant point is visible and it intersects rectangle R2. However, we would not extend R2, as the relevant point at the top-left corner of R2 is occluded behind R1.

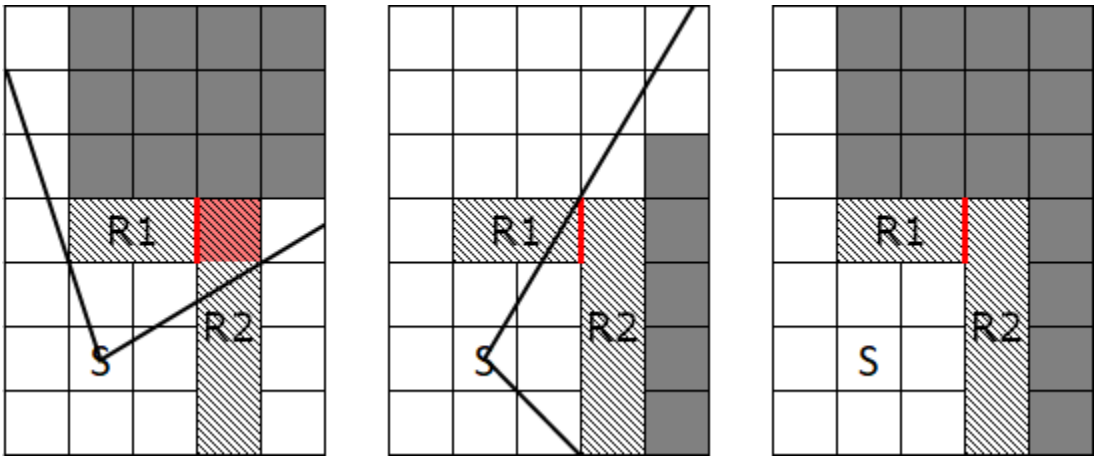

Figure 20: The example from Figure 19, now adjusted to give correct output. The expanded region of R1 is shown in red on the left.

If a rectangle R1 is partially or completely occluded behind another rectangle R2, then their occluded regions will intersect and processing them will result in some cells being set to not visible twice. This does not affect the correctness of our algorithm but it is an important performance consideration as there may be a large number of occluded rectangles. We can reduce the number of cells whose visibility statuses are computed more than once by ignoring the cells within a rectangle R1 which are entirely occluded behind another rectangle R2. We refer to this as *shrinking* R1. Note that a rectangle is only shrunk for the purposes of visibility calculation and this does not affect the size of the rectangle stored in the quadtree.

Directly determining the area of a rectangle R which is occluded behind other rectangles is computationally intensive. We would need to test for intersection between R and the regions occluded by all other rectangles. However, instead of doing this, we use the visibility statuses of the cells of the FOV grid that have already been determined. Once a rectangle has been processed, the FOV grid will contain visibility statuses for the cells in the area that the rectangle occludes. Therefore, to be able to take advantage of the cell visibility statuses already stored in the FOV grid we must order the rectangles such that a given rectangle is only processed after having processed all the rectangles which occlude it. We will discuss later how to order rectangles in this way; for now let us assume that we are processing rectangles in this order.

To shrink a rectangle R, we first determine its relevant points. For each relevant point P we consider the FOV grid cell inside of R which contains P. If that cell is currently set to not visible, we reduce the size of R by one row or column such that R no longer includes that cell. The decision between removing a row or column is always made to minimize the number of cells that are removed from the side of the rectangle which is closest to the FOV source. An example of shrinking a rectangle is given below. We repeat this process of finding relevant points and removing rows or columns from a rectangle until the rectangle is either reduced to nothing, or it is not possible to shrink it any further.

Figure 21 describes the rectangle shrinking process. On the left, part of R1 is occluded behind R2 and there is significant overlap between the regions occluded by both rectangles. If this grid were to have many more rows above the ones shown in the figure, the number of duplicated cell visibility computations would be quite significant. We compute the relevant points of R1 and find that the bottom-left corner cell of R1 is not visible, so we remove the leftmost column of R1 so that it no longer includes that cell. This process is repeated once more, as R1's new bottom left corner is also not visible. In total we remove 4 cells from R1. This shrunk version of R1 is shown on the right, where the overlapping of the occluded regions has been reduced significantly.

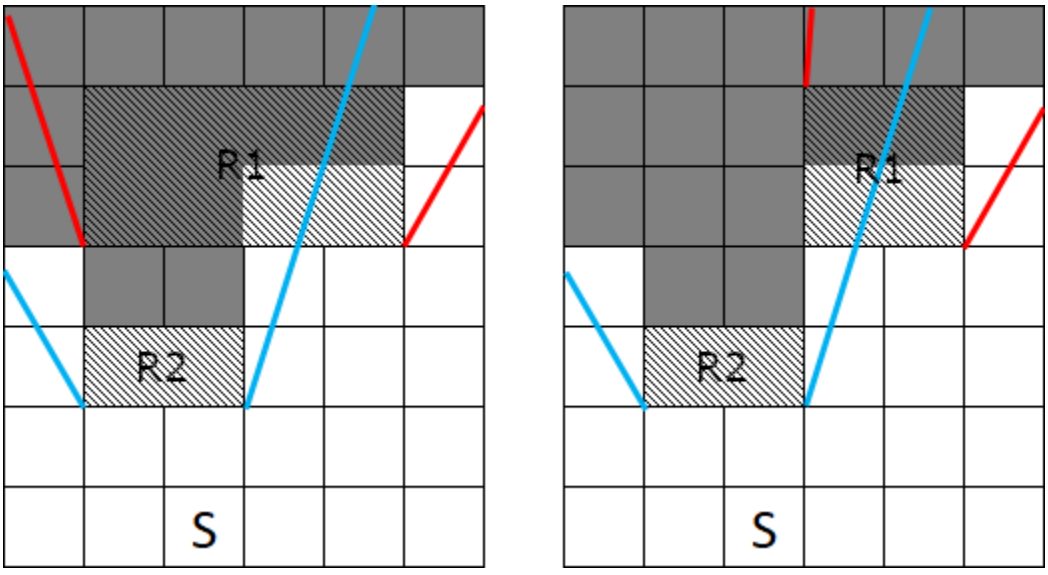

Figure 21: A figure showing rectangle occlusion and shrinking.

2.5 Ordering Rectangles and Calculating the FOV

In order to shrink a rectangle as described in Chapter 2.4, we must process a rectangle only after having processed all rectangles which occlude it. Unfortunately, it seems that the only way to order rectangles in this manner is to directly check for intersection of a rectangle with the area occluded by every other rectangle, which would be computationally expensive. However, the quadtree can be used to quickly generate an ordering for rectangles that will ensure we will process most rectangles only after processing all rectangles which occlude them. This will allow our algorithm to shrink many rectangles. As shrinking rectangle is purely a performance optimization, not being able to shrink all rectangles will not affect the correctness of the algorithm.

To produce this ordering, we must calculate the distance between the FOV source and a vision-blocking rectangle and also the distance between the FOV source and the region represented by a node of the quadtree. Rectangles and node regions are both axis-aligned rectangular areas. We define the distance between the FOV source and a rectangular area as the distance between the source and the closest point to the source in that area.

Determining which point within a rectangular area R is the closest to the source can be done quite easily. Let (X_1, Y_1) and (X_2, Y_2) be the corners of R which are closest to the FOV source. Without loss of generality we may assume that $X_1 \leq X_2$ and $Y_1 \leq Y_2$. Let (X, Y) be the coordinates of the FOV source. We then consider two cases:

If $X_1 < X_2$ **then {**
    if $X < X_1$ **then** (X_1, Y_1) is the closest point of R to the FOV source
    else if $X > X_2$ **then** (X_2, Y_2) is the closest point of R to the FOV source
    else (X, Y_1) is the closest point of R to the FOV source
} else if $Y_1 < Y_2$ **then {**
    if $Y < Y_1$ **then** (X_1, Y_1) is the closest point of R to the FOV source
    else if $Y > Y_2$ **then** (X_2, Y_2) is the closest point of R to the FOV source
    else (X_1, Y) is the closest point of R to the FOV source
}

We traverse the quadtree to produce an ordering of rectangles, as follows. For an internal node of the quadtree, we process its children in order from closest to furthest from the FOV source. For a leaf node, we process its rectangles in order from closest to furthest from the FOV source. As a rectangle can be stored in more than one node, we only process each rectangle the first time it is encountered.

The algorithm for traversing the quadtree and processing rectangles in the order specified above to compute the FOV is given below:

```
Algorithm: rectangleBasedFOV(N, S, G)
Input: quadtree node N, FOV source cell S, FOV grid G
When first called: N will be the root node of the quadtree, all cells in G are set to visible
Output: cells in G which are not visible from S are set to not visible

If N is a leaf then {
    for each rectangle R in N, from closest to farthest from S {
        if R has not already been processed then {

            Extend R if its visible relevant points overlap with another rectangle
            Shrink R if it is occluded by another rectangle
            Let E = region occluded behind R

            for each row X in E {
                Set to not visible the cells in X contained in E, from left to right
            }
        }
    }
} else if N is not a leaf then {
    for each child node C of N, from closest to farthest from S {
            rectangleBasedFOV(C, S, G)
    }
}
```

2.6 A Brief Evaluation of Rectangle-Based FOV

Before moving on to describe an algorithm which updates an existing FOV, we will briefly compare Rectangle FOV to Recursive Shadowcasting. We choose to compare to Recursive Shadowcasting because it is the most popular existing FOV algorithm and because our Rectangle-Based FOV algorithm uses the shadowcast FOV definition. Note that this is not meant as a comprehensive analysis of our Rectangle-Based FOV algorithm; we will include that in our full analysis in Chapter 4. Our intention is to get an impression of how the performance of our algorithm scales with grid size and number of vision-blocking rectangles before we move on to describe further improvements to FOV calculation.

When comparing algorithms, it is important to note that Recursive Shadowcasting (and all other existing FOV algorithms) starts with a grid whose cells have been initially set to non-visible, while our algorithm starts with cells initially set to visible. This means that in an environment with many non-visible cells Recursive Shadowcasting will need to modify relatively few values, while the Rectangle FOV algorithm will need to modify many values. In an environment with many visible cells the opposite will be true. We do not want our performance evaluation to be influenced by this, so we will test with an environment where half of the cells are visible. This will minimize the performance impact of the difference in initial visibility statuses between the two algorithms.

More specifically, we tested both our algorithm and Recursive Shadowcasting in one environment where the FOV source is in the center of the FOV grid and is surrounded by vision-blocking rectangles which make up the shape of a square (see Figure 22). The area inside of this square (the visible area) is almost exactly half of the FOV grid. In order to determine how our algorithm scales as the number of rectangles grows, we performed experiments with different numbers of vision-blocking rectangles. In the simplest case, there are four vision-blocking rectangles. In more complex cases there are many vision-blocking rectangles (see Figure 22). Additionally, we tested at various different grid sizes, just as in Chapter 1.4.

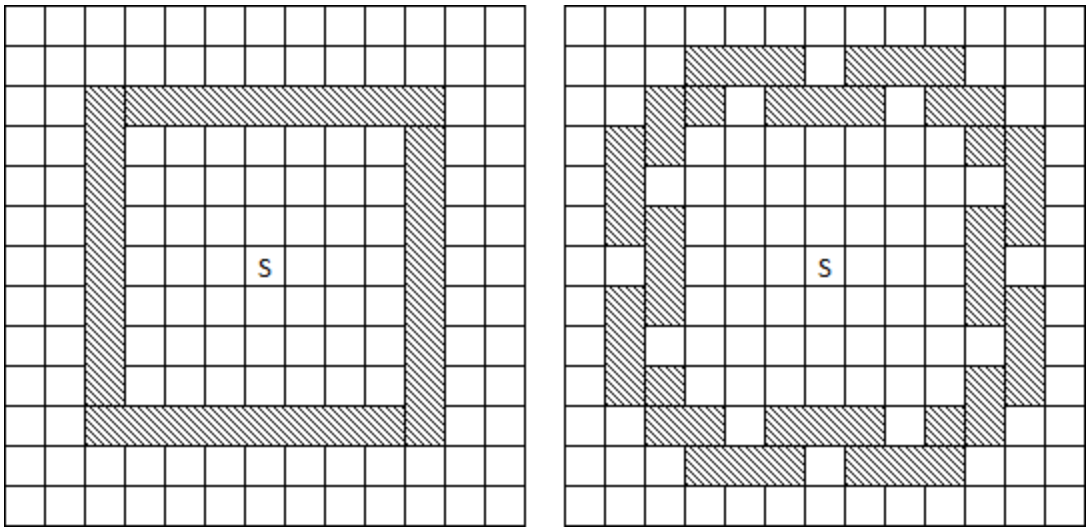

Figure 22: An example of our testing environment on a 13x13 grid, where roughly half of all cells are occluded. The left grid has 4 vision-blocking rectangles, the right grid has 20. As the size of the grid increases, the maximum possible number of vision-blocking rectangles does as well.

Table 7: Mean running times of Shadowcasting and Rectangle FOV for the environments shown in Figure 22.

		Rectangle FOV		
Grid Size	**Shadowcasting**	**4 Rectangles**	**40 Rectangles**	**400 Rectangles**
128*128	37 µs	13 µs	94 µs	N/A
256*256	169 µs	32 µs	127 µs	1,142 µs
512*512	770 µs	96 µs	194 µs	1,552 µs
1024*1024	5,652 µs	293 µs	494 µs	2,391 µs
2048*2048	24,090 µs	1,000 µs	2,053 µs	5,869 µs
4096*4096	112,008 µs	3,499 µs	7,454 µs	16,064 µs

Note that Recursive Shadowcasting is not affected by the number of vision-blocking rectangles. Additionally, 400 rectangles were not used at a size of 128*128, as it is not possible to represent our testing environment with that many rectangles at that number of cells.

The running time of Recursive Shadowcasting on this environment is roughly half of the running time observed for the open environment in Chapter 1.4. This is because there are almost exactly half as many cells to scan and set to visible. The performance of Recursive Shadowcasting is primarily determined by the number of visible cells, and it is not affected by the number of vision-blocking rectangles.

When using only four vision-blocking rectangles, Rectangle FOV is much more efficient than Recursive Shadowcasting. The superior performance of Rectangle FOV is due to its more efficient manner of assigning visibility statuses to grid cells. As discussed previously, Rectangle FOV makes use of spatial locality by assigning cell visibility statuses by grid rows from left to right. Assigning statuses to cells in a sequential manner is fast because the CPU cache stores both memory that is being accessed, and memory nearby. Rectangle FOV benefits the most from spatial locality when the area occluded by a rectangle is very large, as this ensures that a large number of cells occupying consecutive memory locations will be assigned a visibility status. This means that when only four vision-blocking rectangles are present in our test environment, the benefit of spatial locality is very significant. This is unlike Recursive Shadowcasting, which only assigns cells by rows on four of its eight octants.

In addition to the use of spatial locality, Rectangle FOV also needs to perform fewer CPU calculations than Recursive Shadowcasting when assigning cell visibility statuses. Recursive Shadowcasting computes visibility on a per-cell basis, which means that the number of CPU instructions the algorithm executes for each grid cell is relatively large. Rectangle FOV instead determines the cells within each row of a rectangle’s occluded region, and then simply assigns them a visibility status. This means that the number of CPU instructions which Rectangle FOV executes per grid cell will be relatively small, and it will instead execute a larger number of instructions per rectangle. Because of this, if the number of cells being assigned a visibility status in each row of the grid is very large, Rectangle FOV will require fewer CPU instructions to assign visibility statuses to the same number of cells as Recursive Shadowcasting.

As the number of vision-blocking rectangles in the environment increases, the performance benefits of Rectangle FOV lessen. If Rectangle FOV must assign visibility statuses to cells spread over many occluded areas, then the algorithm cannot benefit as much from spatial locality. This is because small occluded regions will result in relatively few cells being assigned a visibility status each time that a row of an occluded region is processed. An increase in the number of rectangles will also result in more CPU instructions, as Rectangle FOV must perform several calculations for each rectangle in order to determine the region it occludes. Even though the performance benefits of Rectangle FOV lessen as the number of vision-blocking rectangles increases, it is still able to outperform Shadowcasting at high grid sizes.

In the environment used for this test (see Figure 22) each rectangle occludes roughly the same area, and as the number of rectangles become large the area occluded by each rectangle becomes relatively small. Additionally, these rectangles only partially occlude each other, so rectangle shrinking will not be able to significantly improve performance. Rectangle FOV may perform better in an environment where a small number of rectangles near the FOV source occlude a large number of rectangles that are farther away from the source. We include such environments in our full analysis in Chapter 4.

Rectangle-Based FOV is able to calculate an FOV more quickly than Recursive Shadowcasting in many cases, but it has some unfortunate shortcomings. The amount of time the algorithm needs per rectangle is significant and can cause the algorithm to perform poorly at low grid sizes compared to Recursive Shadowcasting. Rectangle shrinking helps with this, but unfortunately we cannot guarantee it will be useful in all cases. Most significantly, Recursive Shadowcasting is known to have a very low running time in environments with a small number of visible cells. Environments with few visible cells, such as indoor environments, are very common in computer games. Because of this, in such games we would expect Recursive Shadowcasting to have superior or at least similar performance as Rectangle FOV even at high grid sizes.

Chapter 3

3 Updating an Existing FOV

In this chapter we describe a second new FOV algorithm which attempts to improve performance by updating an existing FOV instead of computing one from scratch.

3.1 An Overview of FOV Updating

In Chapter 2 we represented vision-blocking cells in an efficient manner using a quadtree, which allowed us to efficiently determine and assign grid cell visibility statuses. However, the approach in Chapter 2 has performance which depends on the number of vision-blocking rectangles and may spend a large amount of time computing cell visibility statuses in environments where many cells are not visible. We now describe an FOV algorithm which needs to assign visibility statuses to far fewer cells than any algorithm discussed so far.

When the FOV needs to be calculated in a computer game, most often it is because the FOV source is moving to an adjacent cell. Because of this, it is likely that most FOV grid cells will have the same visibility status between two FOV calculations. Therefore, we may be able to compute the new FOV more efficiently if we update a previously calculated FOV instead of calculating it from scratch.

When updating an FOV, two source points must be considered: S_1, the source for which the FOV was previously calculated, and S_2, the new source for which we must update the FOV. We will consider all cases where S_2 is adjacent to S_1 in a horizontal or vertical direction. Note that that all possible movements of the FOV source can be represented as a combination of single-cell movements which are horizontal or vertical.

A vision-blocking rectangle will have two relevant points when considering S_1, and two relevant points when considering S_2. These are often the same points but can vary depending on how the FOV source moves. We trace rays from each relevant point, away from that point's FOV source. Because the FOV source moves, all four rays will always be different, even if some relevant points are the same, as shown in Figure 23.

These four rays form two ray pairs. Ray pairs are chosen based on the proximity of their relevant points. In most cases at least one pair will be made of rays which share a relevant point. The area between the two rays and possibly the rectangle, is called a *cone* (see Figure 23). Cones represent the space which is occluded behind a rectangle from either S_1 or S_2, but not both. The two cones made by a rectangle represent space where the visibility status may change when updating the FOV.

The point within a cone that is closest to both S_1 and S_2 is said to be the *origin* of the cone.

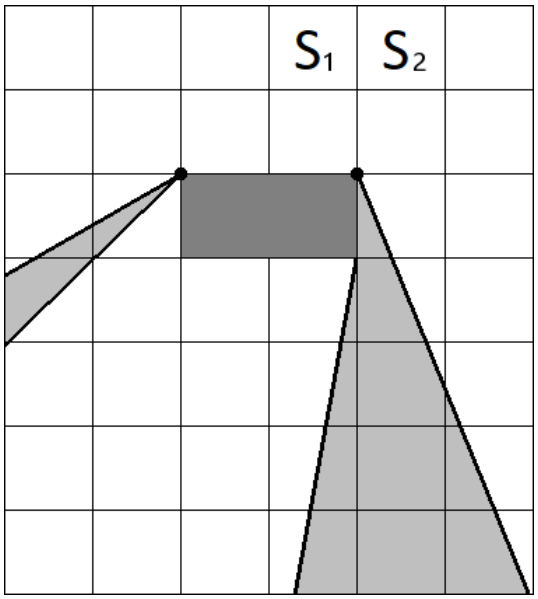

Figure 23: Cones made by a rectangle, S_1, and S_2. Origins are shown with a dot.

In Figure 23, the left cone is formed from two rays which share a point, and the right cone is formed from two rays with close relevant points. In this figure, as the FOV source moves from S_1 to S_2, the left cone defines space losing visibility and the right cone defines space gaining visibility.

An algorithm based on two FOV sources and cones of changing visibility will be more complex than previously discussed algorithms. We provide a summary of this algorithm below, and we devote the rest of Chapter 3 to explaining it in full detail.

Not all cones of changing visibility should be treated equally. Some cones may contain other cones or may be entirely occluded behind rectangles. Chapter 3.2 discusses different cone types and how our algorithm will treat them. This section concludes by explaining that we only need to process cones whose origins are visible from both FOV sources. All other cones can either be ignored or will be processed as a part of processing other cones.

It is important to order cones so they are processed in the correct order to correctly update the FOV. Chapter 3.3 describes a process to order the cones using the quadtree and proves that such an ordering ensures a correct updating of an FOV.

Determining the visibility of a cone from an FOV source may be computationally expensive but is necessary to produce the desired ordering for the cones. Chapter 3.4 describes how the visibility of a cone can be efficiently determined.

Cones may intersect rectangles, and it is important to know when this occurs. For example, if a cone which represents a region that is gaining visibility intersects a rectangle, it is important to stop processing the region of the cone occluded that rectangle so as to avoid setting the cells within that region to visible. Chapter 3.5 describes how to efficiently determine which rectangles intersect a cone.

Chapter 3.6 details the algorithm for processing a cone. Processing every cone in the correct order, while accounting for rectangle intersections, will correctly update the FOV.

3.2 Inverting Cones to Update an FOV

Definition: *Inverting* a cone means to invert the visibility status of all grid cells which are within the cone and are visible from either S_1 or S_2.

Cones must be inverted to update the FOV. Consider all the cones defined by a given set of rectangles for the FOV sources S_1 and S_2. Some cones may be partially or completely hidden from S_1 or S_2. It is helpful to separate cones into three categories as follows:

If the origin of a cone is visible to both S_1 and S_2, that cone is said to be *fully visible*.

If the origin of a cone is visible from S_1 or S_2, but not both, that cone is said to be *transitioning visible* (when moving from S_1 to S_2 the origin either becomes visible or not visible).

If the origin of a cone is neither visible from S_1 nor from S_2, that cone is said to be *not visible*.

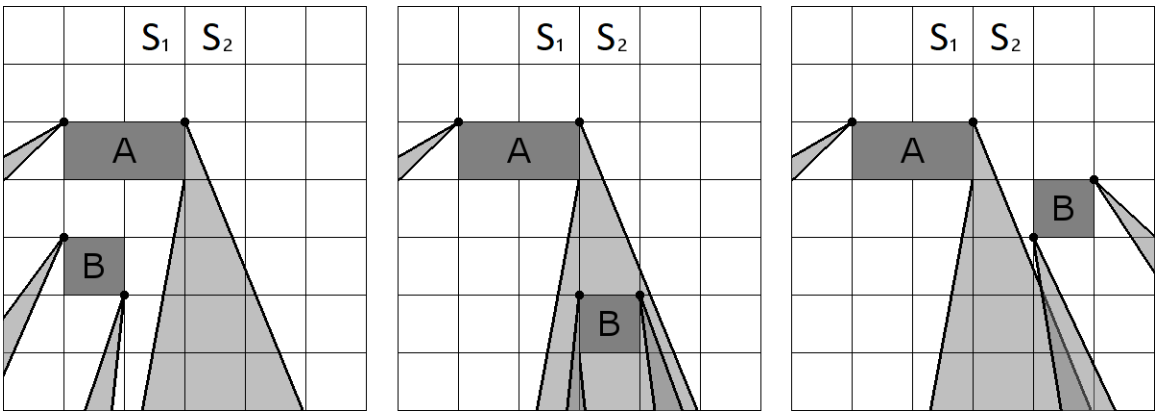

Figure 24: Rectangle B has cones that are not visible (left), transitioning visible (center), and visible (right).

When inverting cones, it is important to consider cases where cones may intersect. If two cones intersect and they are both inverted, some cells may have their visibility status inverted twice. Inverting a cell twice may result in an incorrect visibility status for that cell. Figure 24 contains a few examples of cone intersection.

In the center image of Figure 24, the right cone from rectangle A completely encloses the two cones made by Rectangle B. Inverting both the right cone made by A and the right cone R made by B would incorrectly invert the visibility status of cells in R. However, the left cone L of B only contains space which is not visible to either FOV source, and therefore inverting it and the right cone of A will not result in any cells within L changing visibility status.

In the right image of Figure 24 the right cone of rectangle A and the left cone of rectangle B intersect. The space within this intersection is not visible to either FOV source, which means that no cells within the intersection will have their visibility status changed by the inversion. So, inverting both cones will not result in incorrect visibility statuses, as the visibility status of the intersection of the two cones will not change.

Lemma 1: Inverting non-visible cones does not affect the visibility status of any cells.

Proof: By definition, when a cone is inverted, the only cells whose visibility status is changed are those which are visible from either S_1 or S_2. Since cells in a non-visible cone are not visible to either source, inverting a non-visible cone will result in no cells changing visibility status. □

Lemma 2: Each transitioning visible cone must be completely within a fully visible cone.

Proof: By definition, the origin of a transitioning visible cone is visible from one source and not the other. Therefore, the origin of a transitioning cone T (and thus the entire cone T) must be inside another cone C, as cones by definition define space which is visible from one source and not the other. Cone C cannot be a non-visible cone, as if it were then T would have to be non-visible as well. If C is a transitioning visible cone, then by the same reasoning C must be inside of another cone itself. The only type of cone which can break this chain is a fully visible cone. Therefore, a transitioning visible cone must be within a fully visible cone. □

Lemma 3: When two fully visible cones intersect, the intersection is not visible to either FOV source.

Proof: If two fully visible cones intersect, none of their origins can be in the intersection, as then at least one cone would not be fully visible. Therefore, when two fully visible cones intersect, the intersection must be bounded by one ray cast from S_1 and one ray cast from S_2 (as seen in Figure 24). Rays define the boundaries of visible space from their respective FOV sources, and therefore the area between these intersecting rays is space that is visible to neither S_1 nor S_2. □

Theorem 1: Inverting all fully visible cones will correctly update the FOV when the FOV source moves from S_1 to S_2.

Proof: By Lemma 1 we ignore all non-visible cones, as inversions on them will not affect the visibility status of any cells. By Lemma 2 we ignore all transitioning visible cones, as the cells within them are also within fully visible cones. Thus, correctly updating the visibility status of all cells within fully visible cones will also correctly update the visibility status of all cells within transitioning visible cones.

Let $C_1, C_2, \ldots, C_n$ be the set of all fully visible cones. Any cells outside of $C_1, C_2, \ldots, C_n$ do not change visibility status when the FOV source moves from S_1 to S_2, so only the visibility status of cells in $C_1, C_2, \ldots, C_n$ needs to be changed.

By Lemma 3 we can update the visibility status of the cells in any given fully visible cone C_i independently from all other fully visible cones. A cell in C_i that is visible from S_1 but not visible from S_2 must change its visibility status to not visible, and vice-versa for a cell not visible from S_1 but visible from S_2. These visibility changes are exactly those caused by inverting C_i. Therefore, inverting all fully visible cones will correctly update the FOV. □

In the following sections we concentrate on the specific logic required to invert a fully visible cone.

3.3 Ordering Cones for Inversion

From Theorem 1 we know that we need only to invert fully visible cones to update the FOV when the FOV source moves. Therefore we must check the visibility of a cone's source in order to determine whether it is fully visible or not. Before discussing how visibility can be checked, it is important to note that we can avoid performing this check for transitioning-visible cones if we order the cones in a particular way before inverting them.

By Lemma 2 each transitioning visible cone must be inside a fully visible cone which is closer to the FOV sources. Therefore, if we process cones in increasing order of distance of their origins from the FOV sources, marking all cones found while inverting a fully visible cone, we can easily determine which cones are transitioning visible.

Creating a list of cones sorted by distance from the FOV sources is not necessary however. It is sufficient to ensure the list of cones is partially sorted according to a poset P which satisfies this condition: A fully visible cone c must precede all transitioning visible cones which are inside of it. By processing the cones in an order consistent with P we ensure that any fully visible cone c will be inverted before any cone within c. The quadtree which stores the vision-blocking rectangles can be used to efficiently create a partially sorted list according to P.

When we refer to the distance of a quadtree node to the FOV sources, we mean the distance between the midpoint of the two source points and the closest point to that midpoint within the region represented by the node. Using the quadtree to generate a partially sorted list according to P requires sorting quadtree nodes according to their distance to the FOV sources. Note that the midpoint of the two FOV sources will always be on a point shared by two cells of the FOV grid. This makes it possible for the FOV sources be equidistant to two nodes within the quadtree.

If two nodes are equidistant to the sources, we must temporarily ‘merge’ them for the purpose of generating this list. Merging two nodes means that the children of these nodes or the cones contained within the nodes are processed in increasing order of distance from the FOV sources. More specifically, we need to consider three cases.

- If the two nodes are internal, we traverse all eight of their child nodes in ascending order of their distance from the FOV sources.
- If one of the nodes is a leaf and the other is internal, the leaf node and the four child nodes of the internal node are traversed in ascending order of their distance from the FOV sources.
- If both nodes are leaves, then the cones within both nodes are traversed in ascending order of their distance from the FOV sources.

The algorithm for performing a traversal of the quadtree and generating a partially ordered list of cones according to the poset P is below:

```
Subroutine: findAllConesSorted(N, G, S1, S2, C)
Input: quadtree node N, FOV grid G, FOV sources S1 and S2, set C to store cones
When first called: N is the root node and C is empty
Result: C will contain all cones sorted to satisfy poset P.

 If N is a leaf then {
    for each cone c whose origin is in N, sorted by the distance from c’s origin to S1&S2
        append c to C

} else if N is not a leaf then {
    for each child node N[i] of N, sorted by the distance from N[i] to S1 & S2
        if N[i] and N[i+1] are equidistant to S1 and S2 then {
            temporarily merge N[i] and N[i+1], call the merged node N[i+1]
            increment i by 1   //this skips over N[i], as we have merged it
        }
        findAllConesSorted (N[i], G, S1, S2, C)

}
```

Lemma 5: Traversing a quadtree in the manner described in findAllConesSorted generates an ordering for the cones satisfying the poset P.

Proof: Consider a fully visible cone C and a transitioning visible cone T inside of C. We refer to the origins of these cones as O_C and O_T. Recall that cones are directed away from the FOV sources, so O_T must be further from the FOV sources than O_C.

Consider the nodes of the quadtree which contain O_C, and the nodes which contain O_T. As O_T is further from the FOV sources than O_C, a node which contains O_T must either also contain O_C or it must be at an equal or greater distance from the sources than any node which contains O_C. Therefore, visiting nodes of the quadtree in ascending order by their distance from the FOV sources ensures that a node which contains O_C will not be traversed after a node which contains O_T. If O_C and O_T are within the same leaf node or they are in leaf nodes that are equidistant to the FOV sources, then C and T will be sorted in ascending order by the distance of their origins from the FOV sources and thus C will not appear in the ordering after T. □

3.4 Checking the Visibility of Cones

If a cone has not been marked while inverting a fully visible cone, the visibility of its origin must be checked. This can be done by tracing a line of sight from the cone's origin to either FOV source. If a line of sight intersects any rectangles within the quadtree, then that cone's origin is not visible. The line of sight can be traced to either source, as by definition a fully visible cone must be visible from both sources.

Checking for intersection between a line of sight L and a rectangle R is computationally expensive, however we can simplify this by using minimum bounding rectangles. The minimum bounding rectangle of an object is the smallest possible rectangle which fully encloses that object. To check whether L and R intersect, we first determine if R intersects the minimum bounding rectangle of L. Note that checking if two rectangles intersect can be done quite efficiently, using four integer comparisons at most. This works for both the rectangular region represented by quadtree nodes, and vision-blocking rectangles stored within leaf nodes.

If an intersection between L and any vision-blocking rectangle within the quadtree is found then the cone is known to be not visible, and if no intersections are found the cone is fully visible. Not visible cones can be ignored just as transitioning visible cones, while fully visible cones need to be inverted.

As FOV Update is expected to be run many times with the same environment, it is possible to store the visibility of cone origins to further improve performance. After determining the visibility of a cone origin Q, that information can be stored in a data structure that can retrieve it in constant access time on average, such as a hash table. Note that, as cones define areas gaining or losing visibility, we know that the visibility information stored in the hash table will be accurate so long as Q is not contained in a fully visible cone. If we find Q while processing a fully visible cone, we can simply invert the visibility status we have stored for it in the hash table. Because of this optimization we will only have to directly check the visibility status of Q once.

The algorithm for checking the visibility of the origin of a cone is given below.

```
Subroutine: isLineOfSightInterrupted(N, L)
Input: quadtree node N, line of sight L
When first called: N is the root node
Returns: true if L intersects a rectangle in N, false otherwise

B = minimum bounding rectangle for L

If N is a leaf then {
    for each rectangle r in N
        if r intersects B then
            if r intersects L then
                return true

} else if N is not a leaf then {
    for each child node N[i] of N
        if N[i] intersects B then
            if N[i] intsercts L then
                if isLineOfSightInterrupted(N[i], L) then
                    return true
}

return false
```

3.5 Rectangle Intersections with a Cone

Let b be the line which bisects a given cone c. If the slope of b is between $\frac{-\pi}{4}$, and $\frac{\pi}{4}$, or between $\frac{3\pi}{4}$, and $\frac{5\pi}{4}$, then c is said to be *primarily horizontal*. Otherwise c is said to be *primarily vertical*.

When inverting a cone c, it is necessary to determine if each cell within the cone is visible from S_1 or S_2. Cells within a fully visible cone will always be visible unless there is a rectangle intersecting the cone that blocks the visibility of some cells. Directly checking whether every rectangle intersects c is expensive, however the shape of c can be approximated with a binary tree B of rectangles to allow this checking to be performed efficiently.

The root node of B represents the minimum bounding rectangle for c. The root of B will have two children which each represent the minimum bounding rectangle of one subregion of c(see Figure 25). Subregions of c are created by splitting c at an even interval along the y-axis if the cone is primarily horizontal, or the x-axis if it is primarily vertical. Every internal node of B will have two children, which represent subregions of the region represented by their parent. This continues until the tree reaches some predetermined height, at which point the nodes will be leaves.

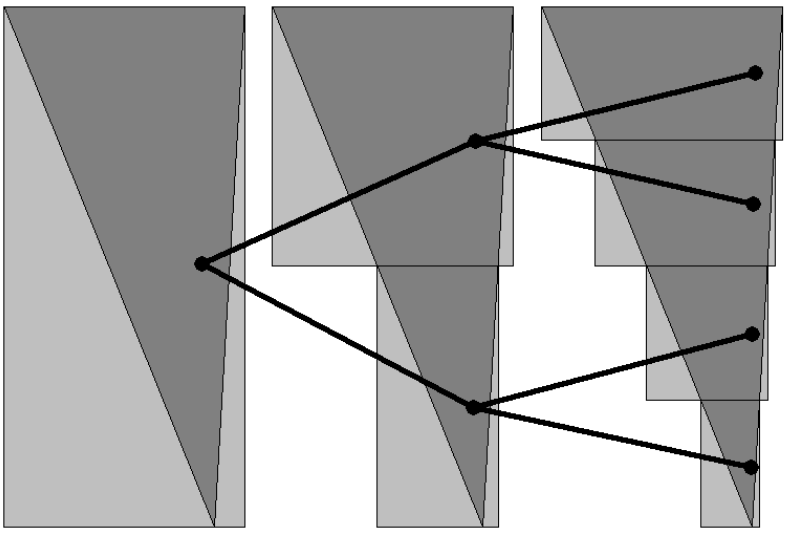

Figure 25: A height 2 binary tree of rectangles, approximating a primarily vertical cone.

The algorithm for building a binary tree to approximate a cone is given below. Note that the depth of a node is equal to the number of nodes between that node and the root, including the root itself. The root of the tree has a depth of 0. The height of a tree is equal to the highest depth value among all nodes in the tree.

Subroutine: buildBinaryTreeForCone(B, C, H)
Input: binary tree node B, cone C, predefined tree height H
When first called: B is the root node representing the minimum bounding rectangle of C
Output: B will be the root of a binary tree of height H that approximates the shape of C

```
If the depth of node B is less than H then {

    Add 2 child nodes to B, each representing the minimum bounding rectangle
    of one subregion of the region of C which B represents.

    for each child node N of B {

        buildBinaryTreeForCone(N, C, H)

    }

}
```

When checking if a rectangle R intersects the cone represented by B, we first check if the rectangle intersects the regions represented by the root node of B. If it does, then we check if R intersects either child node of the root. We then consider the child node which R intersects, or the child closer to the cone's source if R intersects both. We repeat this process until we find intersection with a leaf node, or no intersection can be found with either child node. If R intersects a leaf node, then we consider it to be intersecting the cone. Note that because B approximates the cone it is possible for rectangles to intersect leaves of the tree but not the cone itself. The cone inversion logic which we describe in Chapter 3.6 accounts for the possibility.

The algorithm below finds rectangles which intersect a given binary tree representing a cone.

```
Subroutine: findRectsIntersectingCone(Q, B, R)
Input: quadtree node Q, binary tree B which represents a cone, set R of rectangles.
When first called: Q is the root node of the quadtree of rectangles, R is empty.
Result: R will contain all rectangles which intersect the leaves of B.

If Q is a leaf then
    for each Rectangle r in Q
        if r intersects a leaf of B then
            Insert r into R

else if Q is not a leaf then
    for each child node n of Q
        if n intersects at least one leaf of B then
            findRectsIntersectingCone (n, B, R)
```

When inverting a given cone c, we iterate though its cells either by rows or columns. If c is primarily horizontal, iteration will occur by columns, if c is primarily vertical iteration will occur by rows. We traverse cells in this way to ensure that if a given cell is part of a vision blocking rectangle, it will be processed before any of the cells it may occlude. This could also be achieved by traversing cells based on their distance from c's origin but doing so would involve additional unnecessary calculations.

After a row/column in c is inverted, we check the list of rectangles computed by algorithm findRectsIntersectingCone to determine if any rectangles intersect the cone at that row/column and would therefore occlude cells in future rows/columns. There are three cases of rectangle intersection that need to be handled, each shown in Figure 26:

If a rectangle intersects both edges of c, then the rectangle entirely blocks further row/column inversions, so there are no more cells left to invert.

If a rectangle intersects only one edge of c, then the traversal of further rows/columns should be shrunk in accordance with the space that is occluded behind that rectangle.

If a rectangle does not intersect either edge of c, then the rectangle is completely within c, and effectively splits the visible space into two separate regions. These two sub-regions must be traversed, as they represent the visible space on either side of the rectangle.

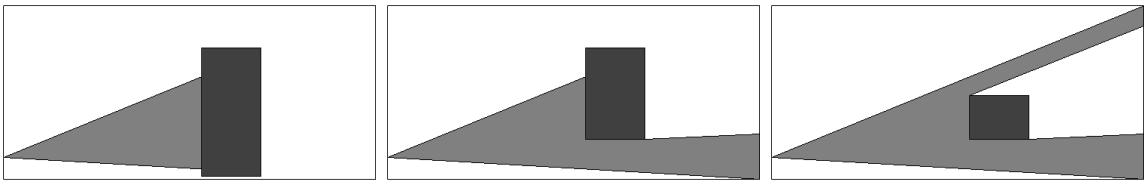

Figure 26: Example of a rectangle intersecting both edges of a cone (left), intersecting one edge of a cone (center), and intersecting no edges of a cone (right).

By iterating through rows/columns as described above and adjusting the size of the traversal as rectangle intersections are encountered, a cone-inverting subroutine will be able to avoid inverting cells which are neither visible from S_1 nor S_2 without having to directly determine the visibility for each cell.

Because a cone c is traversed by either rows or columns, cells within c are visited in order with respect to their distance from c's origin along either the x or y axis. This is important for rectangle processing, as rectangles should be sorted in a manner consistent with when they will be reached by the inversion algorithm. The algorithm moves along the x-axis to process columns for a primarily horizonal cone, and along the y-axis to process rows for a primarily vertical cone. We refer to the axis the algorithm will move along as a cone's *axis of traversal*.

Figure 27 shows that with a primarily horizontal cone, the cone will be traversed by columns, so the x-axis is the axis of traversal. The columns are numbered based on their distance from the cone's origin along the axis of traversal.

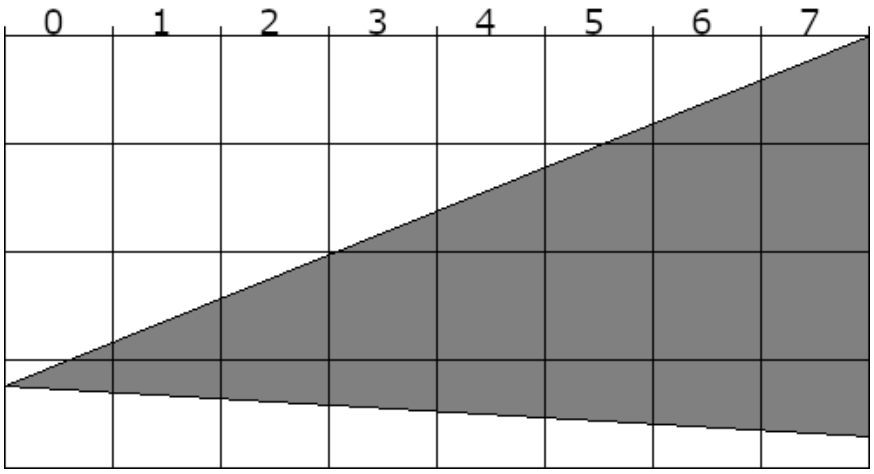

Figure 27: a primarily horizontal cone with numbered columns

The list R of rectangles computed by findRectsIntersectingCone for a cone c should therefore be sorted by their distance from c's origin on c's axis of traversal. The number of intersecting rectangles is not likely to be large, so this sorting should not be a performance constraint.

3.6 The Cone Inversion Algorithm

If the list of rectangles R computed for a cone c is sorted with respect to distance along c's axis of traversal, it is possible to avoid checking for intersection of every rectangle with the cone at each row/column. Rather than checking every rectangle, it is possible to only check a subset of R for each row/column. This is accomplished by performing two additional tests before a rectangle r is checked for intersection:

If the current row/column has a greater distance from c's origin along c's axis of traversal than any cell in r, then r can be removed from R. This is because the current row/column, as well as all future rows/columns cannot intersect with r, so r can simply be ignored.

If the current row/column has a shorter distance from c's origin along c's axis of traversal than any cell in r, then r, as well as all remaining rectangle in R can be ignored for that row/column. This is because r cannot intersect the current row/column, and due to the sorted nature of the list all remaining rectangles in R also cannot intersect.

Figure 28 shows an example of a primarily-horizontal cone which contains a total of 8 columns numbered 0 to 7. There is a rectangle r being considered for intersection, but it does not quite intersect the cone. column 0 is entirely before the rectangle, columns 3 and up are entirely past it. Therefore, only the inversions of columns 1 and 2 will test this rectangle for intersection.

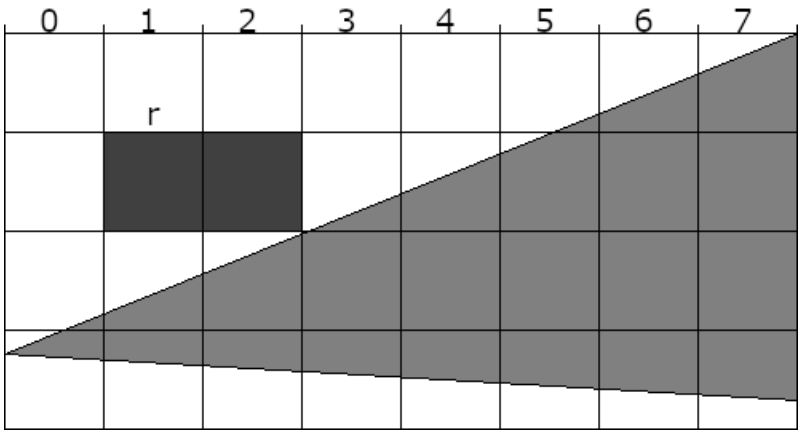

Figure 28: An example of a rectangle r which may have intersected a binary tree, but not the cone itself

The algorithm for inverting a cone takes into account rectangle intersections and is shown below:

Subroutine: invertCone(c, R, G, i)
Input: Cone c, set R of potentially intersecting rectangles, FOV Grid G, integer i
When first called: R is the output of findRectsIntersectingCone and i is 0
Result: the visibility status of cells within G are inverted according to c

Sort rectangles in R based on their distance from c's origin along C's axis of traversal

//Note that the row/column at the origin of c is considered to be the 0th row/column
if c is primarily horizontal **then {**
    set T = all columns in G which c intersects,
    starting from the i-th column from c's origin
} else
    set T = all rows in G which c intersects,
    starting from the i-th row from c's origin
}

for each column/row t **in** T **{**
    Invert all cells within t which are also within c
    i = i + 1
    For each rectangle r **in** R **{**
        // Δc(X) = distance of X from c's source position along c's axis of traversal
        if Δc(t) is greater than Δc(p) for all cells p in r **then** remove r from R
        else if Δc(t) is less than Δc(p) for all cells p in r **then** break
        //continued on next page

```
        else if c intersects r at t then {
            Remove r from R
            Mark the relevant points of r that are within c as handled
            if r intersects both edges of c then {
                return   //cone fully intersected, inversion is finished
            } else if r intersects one edge of c then {
                //see below for further detail
                Adjust the angle of the intersecting edge of c toward the opposite edge,
                  such that c no longer intersects r
            } else {  //intersects neither edge
                // traversal will split in two
                c2 = clone of c
                Adjust the angle of one edge of c2 toward the opposite edge,
                  such that c2 no longer intersects r
                invertCone (c2, clone of R, G, i)
                Adjust the angle of the other edge of c toward the opposite edge,
                  such that c no longer intersects r
        }
    }
}
```

Edges of a cone c which intersect a rectangle r must have their angle adjusted, such that a new smaller cone is formed which no longer intersects r. This results in a smaller cone which is then used for further row/column inversions and rectangle intersection checks. If neither edge of c intersects r, because r is inside of c, then two sub-cones must be formed. These two sub cones are each formed by the adjusting of one of the original cone's edges. In all cases the edge being adjusted has its angle pulled toward the edge which is not being adjusted, until the exact moment an intersection no longer occurs with the rectangle. This is shown in Figure 26.

Lastly, the main FOV-update algorithm combines all the previously discussed subroutines to completely update the FOV when the FOV source moves from S1 to S2:

```
Algorithm: FOV-update (S1, S2, G, T, H)
Input: Grid Cells S1 & S2, FOV Grid G (containing FOV from S1), quadtree T, integer H
Result: the FOV Grid G will contain the FOV from S2

C = new set of cones
findAllConesSorted(root of T, G, S1, S2, C)

for each cone c in C {

    //skip inversion if the cone has been marked as transitioning visible
    if c has not been marked as handled then {

        Line segment L = segment connecting c's origin and S1

        //also skip if the cone is not visible
        if isLineOfSightInterrupted(root of T, L) is not true then {
            R = new set of rectangles
            Create binary tree B of height H which approximates the shape of c
            findRectsIntersectingCone (root of T, B, R)
            invertCone (c, R, G, 0)

        }
    }
}
```

Chapter 4

4 Experimental Evaluation of Our FOV Algorithms

In this chapter we present an experimental evaluation of our implementations of Recursive Shadowcasting, Rectangle-Based FOV, and FOV Update.

4.1 Environments 1 to 4

We first tested using four environments designed to highlight specific properties of the algorithms. Note that, unlike in our previous analysis, we included the time our FOV calculation algorithms take to clear the data structure used to store the visibility values of the grid cells. We included this ‘grid resetting’ procedure in our running times for this analysis because FOV Update does not reset the grid, and so including the time needed to reset the grid more accurately shows the performance difference between our algorithms.

The four testing environments are:

1. An empty FOV grid, matching Environment 1 from Chapter 1.4. This environment will cause Shadowcasting to assign many cell visibility statuses and will cause Rectangle FOV and FOV Update to assign few cell visibility statuses.
2. A 5x5 enclosed space made from four rectangles, matching Environment 2 from Chapter 1.4. This environment will cause Rectangle FOV to assign many cell visibility statuses and will cause Shadowcasting and FOV Update to assign few cell visibility statuses.
3. A square shaped border comprised of 400 rectangles which occludes ~50% of the cells in the FOV grid. This matches the test environment from Chapter 2.6. This environment tests how FOV Update and Rectangle FOV perform when many rectangles are visible.
4. A square shaped border comprised of four rectangles which occludes ~50% of the cells in the FOV grid, plus 396 non-visible rectangles positioned outside the border. This environment has roughly the same number of visible cells as Environment 3, and tests how FOV Update and Rectangle FOV perform when many rectangles are non-visible.

For these environments we tested with the FOV source in the center of the grid, and ensured each algorithm produced the same output. For FOV Update, we first calculated the FOV for the center of the grid, and then measured the running time of updating the FOV for the source moving one cell upward. The running time of FOV Update is not significantly affected by the direction in which the source moves from the center in these environments.

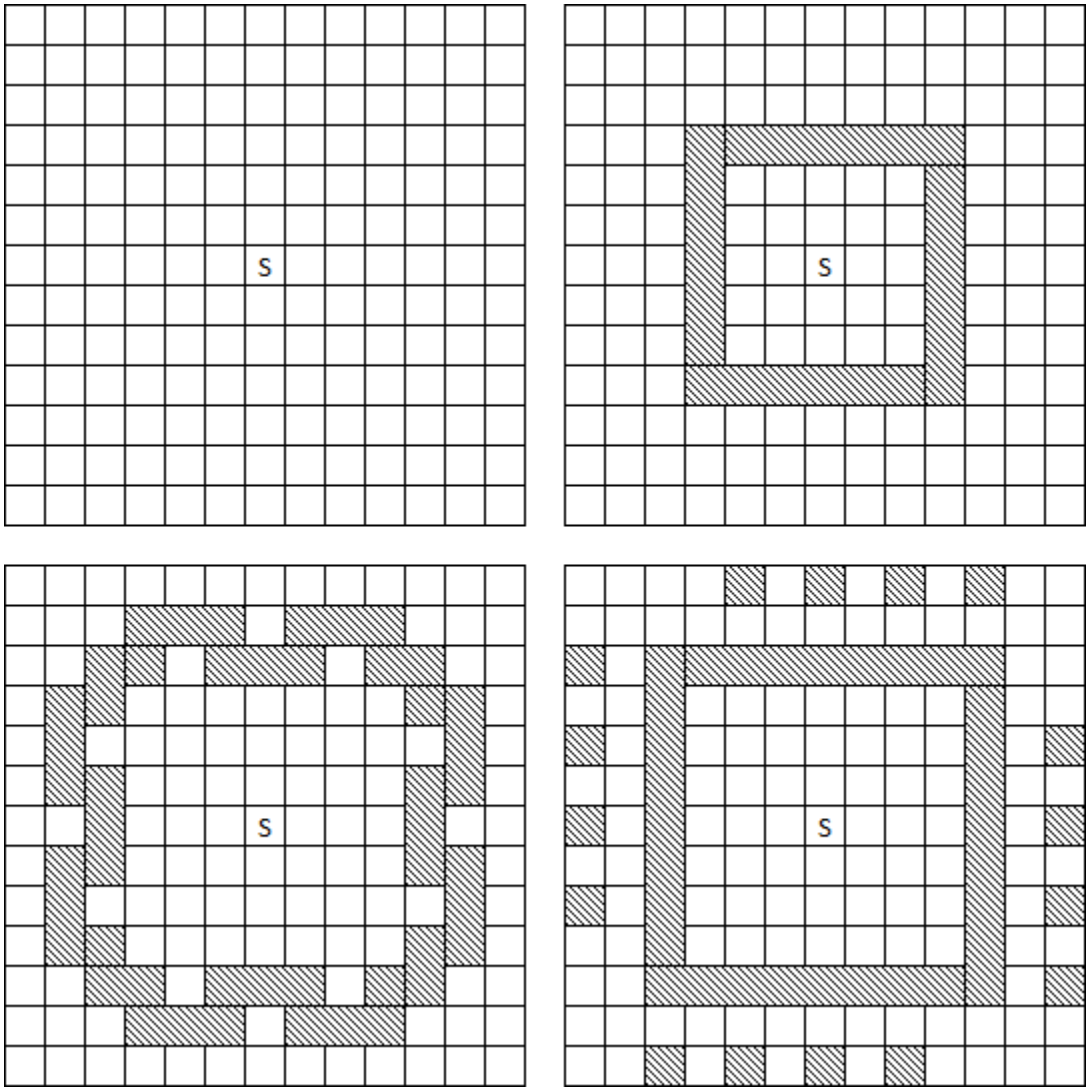

Figure 29: An example of Environments 1, 2, 3, and 4. The environments are shown at a size of 13*13 as individual rectangles cannot be seen at higher grid sizes. Note that environments 3 and 4 are made of 20 rectangles in this example, instead of 400.

Table 8: Mean running times for Environment 1

Grid Size	Shadowcasting	Rectangle FOV	FOV Update
128*128	75 µs	3.0 µs	0.3 µs
256*256	329 µs	11 µs	0.3 µs
512*512	1,526 µs	44 µs	0.3 µs
1024*1024	11,224 µs	173 µs	0.3 µs
2048*2048	46,387 µs	692 µs	0.3 µs
4096*4096	241,818 µs	2,762 µs	0.3 µs

Table 9: Mean running times for Environment 2

Grid Size	Shadowcasting	Rectangle FOV	FOV Update
128*128	3.9 µs	15 µs	2.4 µs
256*256	13 µs	31 µs	2.4 µs
512*512	45 µs	96 µs	2.4 µs
1024*1024	176 µs	339 µs	2.4 µs
2048*2048	695 µs	1,300 µs	2.4 µs
4096*4096	2,808 µs	5,020 µs	2.4 µs

Shadowcasting performs very poorly in Environment 1, as it assigns a visibility status to every cell, and Shadowcasting uses a rather inefficient algorithm to assign visibility statuses to the cells. Rectangle FOV performs very well in Environment 1, as it has no rectangles to process. The performance of Rectangle FOV in Environment 1 is almost entirely determined by the time taken to reset the FOV grid. FOV Update completes almost instantly in Environment 1, as it has no cones to process and does not need to reset the FOV grid.

Shadowcasting performs well in Environment 2, as it assigns a very small number of cell visibility statuses after it resets the FOV grid. Rectangle FOV has a longer running time in Environment 2 than in Environment 1, as it assigns a visibility status to almost every cell in the FOV grid. The running time of Rectangle FOV in Environment 2 is almost exactly twice as long as in Environment 1, so therefore the algorithm assigns cell visibility statuses when processing rectangles at roughly the same speed at which it assigns visibility statuses when it clears the grid. FOV Update completes almost instantly in Environment 2, as all cones are occluded and thus no cell visibility statuses are changed.

Table 10: Mean running times for Environment 3
Note that this environment cannot be represented at a grid size of 128*128

Grid Size	Shadowcasting	Rectangle FOV	FOV Update
256*256	169 µs	1,293 µs	1,311 µs
512*512	797 µs	1,709 µs	1,475 µs
1024*1024	5,674 µs	2,511 µs	1,439 µs
2048*2048	23,803 µs	4,800 µs	1,427 µs
4096*4096	114,881 µs	12,435 µs	1,438 µs

Table 11: Mean running times for Environment 4

Grid Size	Shadowcasting	Rectangle FOV	FOV Update
128*128	42 µs	335 µs	258 µs
256*256	169 µs	357 µs	258 µs
512*512	797 µs	428 µs	258 µs
1024*1024	5,674 µs	654 µs	258 µs
2048*2048	23,803 µs	1,455 µs	258 µs
4096*4096	114,881 µs	5,004 µs	258 µs

Environments 3 and 4 test how efficiently Rectangle FOV and FOV Update handle many rectangles in the environment. In Environment 3 all 400 rectangles are visible, whereas in Environment 4 only four rectangles are visible. The number of cell visibility statuses which are assigned by the algorithms is identical in both environments, which is why Recursive Shadowcasting has the same running time in both cases.

In Environment 3 all rectangles are visible, and so FOV Update and Rectangle FOV must process all of them. FOV Update exhibits superior performance at higher grid sizes as it does not change any grid cell visibility statuses as the FOV source moves.

Both Rectangle FOV and FOV Update exhibit better performance in Environment 4 than in Environment 3. This is because many rectangles are not visible in this environment, and so the algorithms avoid processing most of them. Rectangle FOV must, for each rectangle, check the visibility status of every cell on the faces of the rectangle that are nearest to the FOV source before determining whether that rectangle is visible or not. FOV Update, by comparison, needs only check the visibility of the origin point of a cone. Because of this, the performance impact caused by the non-visible rectangles increases as grid size increases for Rectangle FOV, but it does not affect FOV Update.

From these four test cases we can make the following conclusions:

The running time of FOV Update is nearly constant in all test cases as the number of visible cells does not change in these environments when the FOV source moves from the center. This allows FOV Update to exhibit constant running time because the only operation within FOV Update which depends on the size of the FOV grid is the inversion of cell visibility statuses. If few or no cells change their visibility status, the performance of FOV Update will not be affected by grid size. This is in contrast to the other two FOV algorithms, whose running times depend on grid size.

FOV Update does not exhibit poor performance when many cells are visible or when many cells are not visible. This is in contrast to Recursive Shadowcasting, which performs poorly when many cells are visible, and Rectangle FOV, which performs poorly when few cells are visible. In fact, FOV Update should perform well in environments where most cells are visible, or where most cells are not visible, as in such environments the visibility status of few cells will change when the source moves.

The number of vision-blocking rectangles significantly affects the performance of both Rectangle FOV and FOV Update. Both algorithms exhibit good performance when rectangles are hidden instead of visible, but the performance is significantly better for FOV Update. This is because the Rectangle FOV process of skipping non-visible vision-blocking rectangles has performance which depends on the size of those rectangles, whereas FOV Update has a cone skipping procedure which is independent of rectangle size. The visibility of cone origin points can also be stored between updates to further improve performance.

Recursive Shadowcasting has superior running time in environments with many vision-blocking rectangles and few visible cells, but when the number of visible cells becomes large it performs poorly when compared to Rectangle FOV or FOV Update. FOV Update performs particularly well at high grid sizes, as it needs to assign relatively few visibility statuses and does not need to clear the FOV grid.

After performing running time benchmarks, we tested our algorithms using these same four environments and collected more detailed performance metrics to help understand why each algorithm ran for as long as it did.

We measured three metrics during these tests: number of processor instructions, cache misses, and cache hits. A cache miss is when the processor attempts to read or write data and that data is not currently stored in the processor's cache. A cache hit is when the processor attempts to read or write to data which is present in the cache. Cache misses have a relatively large performance impact, as data must be fetched from main memory.

We measured these numbers by using the perf utility within the Linux operating system. Just as with the running time measurements, we ran each algorithm many times using separate FOV grids and report the mean results.

Table 12: Mean performance metrics for Environment 1

	Shadowcasting		Rectangle FOV		FOV Update	
Grid Size	**Instructions**		**Instructions**		**Instructions**	
128*128	566,179		1,457		1,759	
256*256	2,188,712		1,835		1,687	
512*512	8,590,273		5,744		1,234	
1024*1024	34,167,155		20,344		1,549	
2048*2048	136,000,513		68,832		1,338	
4096*4096	541,816,169		118,772		1,643	
	Cache Operations					
Grid Size	**Misses**	**Hits**	**Misses**	**Hits**	**Misses**	**Hits**
128*128	271	665	12	259	2	4
256*256	1,167	3,578	23	1,024	2	6
512*512	4,657	32,007	102	4,229	3	8
1024*1024	20,198	1,049,073	456	16,945	2	8
2048*2048	83,016	4,414,279	948	67,132	3	7
4096*4096	618,686	17,587,416	1,280	262,316	4	8

Table 13: Mean performance metrics for Environment 2

	Shadowcasting		Rectangle FOV		FOV Update	
Grid Size	**Instructions**		**Instructions**		**Instructions**	
128*128	4,250		84,252		12,598	
256*256	6,150		124,393		12,573	
512*512	8,657		219,656		11,747	
1024*1024	34,301		462,138		12,364	
2048*2048	46,419		974,566		11,980	
4096*4096	230,012		2,179,053		12,673	
	Cache Operations					
Grid Size	**Misses**	**Hits**	**Misses**	**Hits**	**Misses**	**Hits**
128*128	26	279	4	258	4	6
256*256	72	1,032	18	2,109	3	7
512*512	167	4,252	97	9,069	4	8
1024*1024	677	17,024	474	59,634	5	7
2048*2048	1,566	69,387	804	220,300	4	8
4096*4096	2,233	259,013	12,424	740,182	5	8

The metrics for Environments 1 and 2 match with the observed running times for each algorithm. Shadowcasting has the lowest number of operations in Environment 2 due to the low number of visible cells. Rectangle FOV has the fewest operations in Environment 1 due to the high number of visible cells. FOV Update has the fewest operations in both environments at high grid sizes due to not being affected as strongly by grid size.

Some observations can be made by comparing Rectangle FOV in Environment 2 with Shadowcasting in Environment 1. In each of these cases the algorithms assign a visibility status to roughly all the cells in the FOV grid, but the metrics are quite different. As grid resolution becomes high Shadowcasting performs ~250 times the number of instructions as Rectangle FOV, and ~24 times the number of cache hits, and ~50 times the number of cache misses. This clearly highlights the inefficiency of Shadowcasting's method of computing FOV one cell at a time. Because Rectangle FOV computes FOV for the entire area occluded by a given rectangle, it is able to compute FOV by performing far fewer instructions. Rectangle FOV also has a much lower proportion of cache misses because of its better use of spatial locality.

Table 14: Mean performance metrics for Environment 3
Note that this environment cannot be represented at a grid size of 128*128

	Shadowcasting		Rectangle FOV		FOV Update	
Grid Size	**Instructions**		**Instructions**		**Instructions**	
256*256	1,147,177		8,384,396		5,738,107	
512*512	4,415,166		11,699,557		6,498,240	
1024*1024	17,268,299		17,200,022		6,350,838	
2048*2048	68,385,109		28,257,405		6,289,103	
4096*4096	272,896,449		50,216,815		6,086,486	
	Cache Operations					
Grid Size	**Misses**	**Hits**	**Misses**	**Hits**	**Misses**	**Hits**
256*256	813	2,702	376	3,832	27	3,634
512*512	3,307	18,848	622	13,117	118	4,457
1024*1024	12,807	505,705	1,004	45,543	109	4,709
2048*2048	49,508	2,249,637	2,956	148,371	210	4,305
4096*4096	318,454	9,025,138	6,812	563,293	554	3,789

Table 15: Mean performance metrics for Environment 4

	Shadowcasting		Rectangle FOV		FOV Update	
Grid Size	**Instructions**		**Instructions**		**Instructions**	
128*128	295,556		1,991,961		1,331,668	
256*256	1,112,527		2,126,772		1,340,166	
512*512	4,360,386		2,415,775		1,347,151	
1024*1024	17,200,946		2,975,305		1,336,652	
2048*2048	68,814,365		4,206,014		1,333,260	
4096*4096	275,283,817		6,316,213		1,309,088	
	Cache Operations					
Grid Size	**Misses**	**Hits**	**Misses**	**Hits**	**Misses**	**Hits**
128*128	217	539	28	712	1	82
256*256	816	2,856	83	2,647	2	159
512*512	4,809	19,735	200	10,754	3	325
1024*1024	12,889	503,844	260	41,044	3	342
2048*2048	58,245	2,255,750	1,268	123,543	2	364
4096*4096	355,682	9,111,054	6,144	427,064	4	544

The performance metrics for Shadowcasting in Environments 3 and 4 are roughly one half of the performance metrics seen in Environment 1. This is as expected, as Environments 3 and 4 both have roughly half the number of visible cells as Environment 1.

Compared to the other two algorithms, Shadowcasting performs relatively few instructions at low grid sizes. Rectangle FOV and FOV Update have a larger number of instructions at low grid sizes, because they must process a quadtree of 400 rectangles regardless of grid size. This high instruction count is the main cause of the longer running time of these algorithms at low grid sizes, as their number of cache hits and misses are close to those of shadowcasting. The number of cache misses in particular is smaller than that of shadowcasting even at the lowest grid sizes, indicating that our algorithms manage memory more efficiently despite having a longer running time.

Comparing the metrics for Environments 3 and 4 allows us to see how effectively Rectangle FOV and FOV Update are at skipping occluded rectangles. Both algorithms use significantly fewer CPU instructions when many rectangles are occluded, and this reduction is by roughly the same proportion for both algorithms. Both algorithms also exhibit fewer cache hits and misses, but this reduction is more significant for FOV Update. This is because Rectangle FOV must check many cells along a rectangle to determine if it is occluded, while FOV Update only needs to check the origin point of a cone. This means that Rectangle FOV's process of skipping non-visible rectangles involves a number of memory references which increases with grid size, while FOV Update's process does not.

4.2 Environments 5 to 8

We now show test results for four additional environments which are meant to emulate terrain which may appear in a computer game. This time we did not place the FOV source in the center of the grid. Instead we tested with 25 randomly generated paths of 100 cells each. Each path was generated by randomly selecting the center of a cell as the start of the path, among all cells which are not vision-blocking. An x and y value are then chosen to represent the direction of the path. These values are both uniformly distributed random numbers between -1 and 1. The starting cell and direction values (x, y) define a ray. Cells which intersect this ray are added to the path in ascending order of distance from the starting cell. If the next cell to add to the path belongs to a vision-blocking rectangle, a new random direction (x, y) is generated and cells are added to the path using that new direction. This continues until the path contains 100 cells.

The FOV was calculated for each cell along the path in the order in which the cells appear in the path. We used paths in these test environments to mimic scenarios arising in computer games, where the FOV source will be moving through the grid as a game character moves. In the case of FOV Update, for each path we computed an initial FOV using the Rectangle-Based FOV algorithm, and then measured the running time of updating the FOV for every position of the FOV source along the path. We also verified that for each FOV source position each algorithm produced the same FOV.

Each test environment uses a fixed number of rectangles; as the grid size increases the sizes of the rectangles are increased by the same proportion. This is done to replicate how game implementors will likely choose to scale their environments to higher grid sizes. As performance is significantly affected by the number of vision-blocking rectangles, implementors will want to use as few rectangles as possible to model their terrain. The primary benefit of increasing the FOV grid size will be the increased precision of the visibility statuses stored within the grid cells (see Figure 30).

Scaling the environment in this manner also mimics how the developers of League of Legends[2] attempted to increase the quality of their fog of war[4]. When the game's developers considered increasing the size of their FOV grid, it was to increase the precision of the resulting FOV, not so they could more accurately represent their vision-blocking terrain.

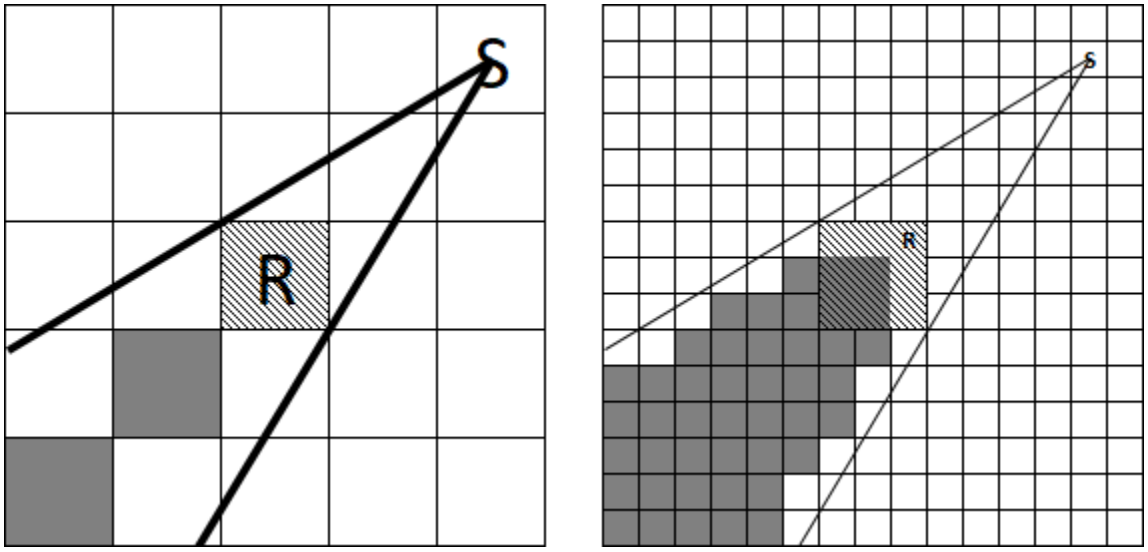

Figure 30: A simple FOV grid with a single vision-blocking rectangle R. The grid is of size 5x5 on the left and of size 15x15 on the right. The area occluded by R is darkened. The FOV on the right grid is more precise due to the increased grid size.

The four additional testing environments are:

5. A fixed indoors environment with 36 square rooms connected by 74 corridors. This environment is constructed such that there is never an alignment of corridors which would allow the FOV source to see across many rooms. This is an enclosed environment where many cells and rectangles/cones will be occluded. This environment is comprised of 160 rectangles.
6. A randomized environment where 200 rectangles of random sizes are placed at random positions. This simulates a more disorganized outdoors environment, such as a forest. Each rectangle has a width and height which are uniformly distributed random values between one and six cells. The position of each rectangle is chosen uniformly at random from all locations that do not intersect another rectangle.
7. A randomized environment where 200 rectangles of random sizes are more densely grouped around the center of the FOV grid and fewer rectangles appear further from the center. This simulates a more organized outdoors environment, such as a town. Each rectangle has a width and height which are uniformly distributed random values between one and six cells. For each rectangle, five x coordinate values and five y coordinate values are chosen uniformly at random. The average value from each set of five coordinates is used as the x and y coordinate of the center point of the rectangle. The position of the center point of the rectangle is then rounded so that it aligns to the grid. If this rectangle position would result in intersection with another rectangle, a new random center is generated as described above. This averaging process results in rectangle positions which are more likely to be near the center of the grid.
8. A fixed environment meant to emulate the visibility grid used in League of Legends [2]. This tests the FOV algorithms using an environment taken from an existing game. The League of Legends map was chosen because of its shape: The League of Legends environment is an organized mix of enclosed spaces and large open pathways, which makes it ideal for thoroughly testing our algorithms. This environment is comprised of 300 rectangles.

Note that all algorithms were tested with the same randomly generated environments and randomly selected FOV source paths. All random calculations used a pseudorandom number generator.

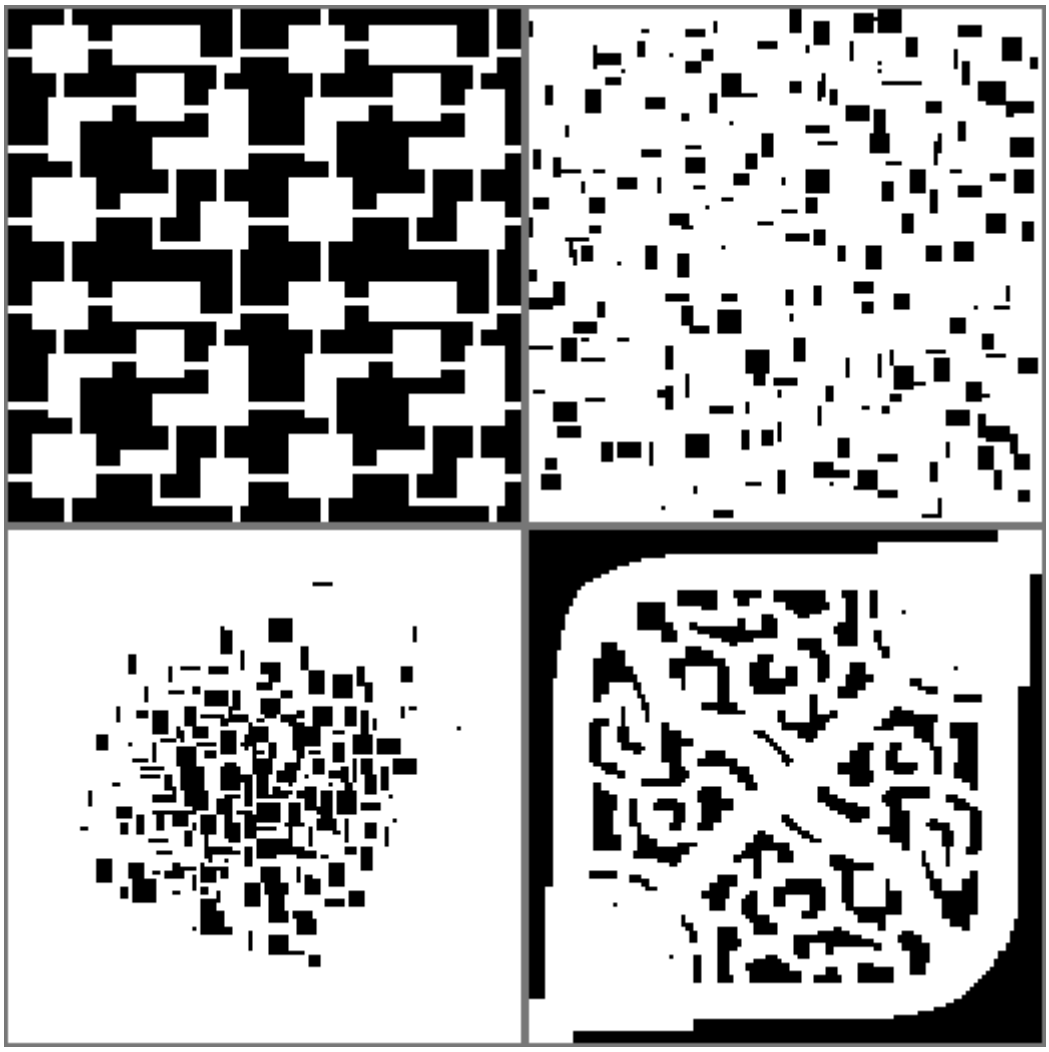

Figure 31: Environments 5, 6, 7, and 8 on a grid of size 128*128. Black cells are vision-blocking, white cells are not vision-blocking.

Table 16: Running times for Environment 5.

	Shadowcasting		Rectangle FOV		FOV Update	
Grid Size	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**
128*128	6.5 µs	1 µs	205 µs	20 µs	170 µs	24 µs
256*256	21 µs	3 µs	259 µs	26 µs	174 µs	25 µs
512*512	80 µs	14 µs	401 µs	39 µs	188 µs	27 µs
1024*1024	290 µs	43 µs	774 µs	68 µs	204 µs	46 µs
2048*2048	1,342 µs	278 µs	2,001 µs	163 µs	249 µs	77 µs
4096*4096	6,665 µs	1473 µs	10,269 µs	765 µs	356 µs	140 µs

Environment 5 has a low number of visible cells because it is very enclosed, which means both Shadowcasting and FOV Update will assign few cell visibility statuses. As the running time of Shadowcasting is primarily dependent on the number of visible cells, it performs relatively well here.

FOV Update performs well at high grid sizes in this environment, as on average very few cells change visibility status each time the FOV is updated. Many cones are non-visible as well, which allows FOV Update to skip processing them.

Rectangle FOV exhibits relatively poor performance in this environment. The high number of non-visible cells means that Rectangle FOV will assign a large number of cell visibility statuses. Additionally, many of the rectangles in this environment are large, which means Rectangle FOV will have to check many cells along the sides of these rectangles to determine if they are non-visible.

All three algorithms exhibit a standard deviation which is only a small percentage of their mean running time in this environment. This is because Environment 5 has a very consistent layout. The difference in the number of visible cells and visible rectangles at any two given source positions will not be very large, and so the performance of the algorithms is not significantly impacted by the position of the FOV source. The one exception to this is FOV Update at high grid sizes, which has a proportionally large standard deviation. This is because most movements of the FOV source will result in few to no cells changing visibility status, but relatively many cells will change visibility status when the FOV source moves around a corner.

Table 17: Running times for Environment 6

	Shadowcasting		Rectangle FOV		FOV Update	
Grid Size	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**
128*128	17 µs	6.5 µs	300 µs	49 µs	468 µs	137 µs
256*256	54 µs	16 µs	358 µs	52 µs	504 µs	135 µs
512*512	201 µs	53 µs	494 µs	77 µs	595 µs	152 µs
1024*1024	777 µs	289 µs	943 µs	172 µs	763 µs	243 µs
2048*2048	3,898 µs	1,747 µs	2,176 µs	277 µs	1,073 µs	366 µs
4096*4096	19,345 µs	8,426 µs	7,347 µs	1,059 µs	1,863 µs	821 µs

The mean running time of Shadowcasting is significantly larger in Environment 6 than in Environment 5. This is due to the increase in the number of visible cells. Its standard deviation is also much larger, which is explained by the random nature of the environment. Some source cell positions will result in significantly more or significantly fewer visible cells than other source positions.

Rectangle FOV performs slightly worse here than in Environment 5, as more rectangles are visible. However, because there is an increase in the number of visible cells, its relative performance versus Shadowcasting is significantly improved from Environment 5. Similarly to its performance in Environment 5, Rectangle FOV exhibits a standard deviation which is a relatively small percentage of its mean running time.

FOV Update performs very well at high grid sizes, but the larger number of visible cones that it must consider affects its performance regardless of grid size. This causes the algorithm to exhibit relatively poor performance at low grid sizes. Higher variance in the number of visible cones also results in FOV Update having a high standard deviation at low grid sizes. However, even in this environment where a large number of visible cones must be considered, FOV Update performs relatively few cell visibility assignments. This allows it to exhibit superior performance as grid size becomes large.

Table 18: Running times for Environment 7

	Shadowcasting		Rectangle FOV		FOV Update	
Grid Size	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**
128*128	25 µs	9.7 µs	272 µs	35 µs	471 µs	138 µs
256*256	83 µs	35 µs	314 µs	43 µs	466 µs	142 µs
512*512	343 µs	169 µs	431 µs	64 µs	489 µs	146 µs
1024*1024	2,132 µs	809 µs	832 µs	117 µs	676 µs	173 µs
2048*2048	11,529 µs	5,592 µs	2,072 µs	226 µs	969 µs	269 µs
4096*4096	46,203 µs	25,962 µs	6,710 µs	1,007 µs	1,331 µs	539 µs

Recursive Shadowcasting exhibits very poor performance in this environment at high grid sizes. Unless the FOV source is near the center of the grid there will be a high number of visible cells, which will result in a long running time. This can be seen in the large standard deviation.

Rectangle FOV performs slightly better in this environment than in Environment 6. This is due to the higher degree of rectangle clustering, which decreases the average number of visible rectangles. Additionally, the number of non-visible cells will be lower than in Environment 6 on average when the FOV source is not near the center of the grid.

FOV Update performs better in this environment than in Environment 6 as grid size becomes large. Because rectangles are clustered together in this environment, it is more likely that cones will be occluded. This means that on average FOV Update will need to consider fewer cones than in Environment 6.

Table 19: Running times for Environment 8

	Shadowcasting		Rectangle FOV		FOV Update	
Grid Size	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**
128*128	13 µs	6.5 µs	403 µs	57 µs	558 µs	220 µs
256*256	46 µs	24 µs	482 µs	78 µs	566 µs	223 µs
512*512	163 µs	75 µs	656 µs	100 µs	590 µs	219 µs
1024*1024	844 µs	468 µs	1,173 µs	210 µs	687 µs	328 µs
2048*2048	4,157 µs	2,780 µs	2,643 µs	472 µs	802 µs	432 µs
4096*4096	22,007 µs	13,698 µs	8,692 µs	1,724 µs	1,247 µs	765 µs

Shadowcasting has a high running time in this environment, as well as a high standard deviation. While this environment is more ‘structured’ than Environments 6 and 7, it is not an indoors environment like Environment 5, and so the running time of Shadowcasting becomes very large as grid size increases. Similarly to Environment 7, the number of visible cells will vary strongly based on where the FOV source is located, which results in a high standard deviation for this algorithm.

Environment 8 has 300 vision blocking rectangles, compared to 200 for Environments 7 and 6 and 160 for Environment 5. This increased number of vision-blocking rectangles negatively affects the performance of Rectangle FOV. This is especially true at low grid sizes, where the performance impact of the number of rectangles is most significant. As grid size increases, the efficiently of Rectangle FOV when processing cell assignments allows it to outperform Recursive Shadowcasting, despite the increased number of rectangles.

FOV Update is also hindered by the larger number of vision-blocking rectangles in Envorinment 8, but only at lower grid sizes. On average there are few changed cell visibility statuses as the source moves, so FOV Update performs better here at high grid sizes than in Environments 6 and 7, despite the higher number of rectangles. In certain cases, such as when the source moves around a corner, a large number of cells may change visibility status, which gives FOV Update a larger standard deviation in this environment.

From these additional four test cases we can make the following conclusions:

At low grid sizes Rectangle FOV and FOV Update are significantly slowed by having to deal with the quadtree of rectangles, while Shadowcasting only has to concern itself with the grid. This allows Shadowcasting to consistently exhibit the best performance at low grid sizes. However, as grid size increases the performance impact of the quadtree becomes less important, and the poor per-cell performance of Shadowcasting causes it to perform poorly in all but very enclosed environments.

Rectangle FOV and FOV Update are not affected as strongly by the shape of an environment as Shadowcasting is. The number of cells that are visible from a given FOV source strongly influences the running time of Shadowcasting, whereas our algorithms are more strongly affected by grid size and number of rectangles. This leads to very high standard deviations for Shadowcasting in some environments. FOV Update also exhibits a proportionally high standard deviation in some environments, as its performance is affected by the number of visible cones and the number of cells which change visibility status.

FOV Update performs very well at high grid sizes in all environments because it assigns few cell visibility statuses. Both Rectangle FOV and Shadowcasting must assign a larger number of cell visibility statuses, which causes their running times to increase more sharply as grid size increases. Rectangle FOV does assign cell visibility statuses more efficiently than Recursive Shadowcasting because of its use of spatial locality, but this is not enough to allow it to compete with FOV Update at high grid sizes.

In Environments 5 to 8, the longest running times were ~46ms for shadowcasting, ~10ms for rectangle FOV, and ~2ms for FOV update. The improvements achieved by our algorithms may not seem significant, as even 46ms is a very small amount of time. However it is important to consider that computer games do not only calculate FOV: many game processes are simultaneously running, competing for system resources, and so a computer can only dedicate a small portion of processor time to FOV calculation. Additionally, games are generally rendered in real-time at 60 frames per second, which allows ~17ms to render each frame. While FOV will not be calculated for every frame, spending dozens of milliseconds to calculate it will result in some frames being skipped, which significantly impacts the smoothness of a game's display. Because there may be little processor time devoted to computing FOV, even spending a few milliseconds computing it may be enough to cause frames to be skipped.

Because of the differing strengths of each of the three tested algorithms, the most effective way to calculate FOV is to use a hybrid approach. Based on our experimental evaluations, we recommend the following:

If an environment has a low number of visible cells, Recursive Shadowcasting is the most effective algorithm for calculating an FOV. An environment may have a low number of visible cells either due to a low grid size or because the environment is indoors. In outdoors environments, which have a larger number of visible cells, Recursive Shadowcasting consistently offers the best performance at grid sizes up to 512*512.

If an environment has a high number of visible cells, Rectangle FOV is the most efficient algorithm for calculating an FOV. An environment may have a high number of visible cells when it is outdoors and grid size is large. Rectangle FOV consistently offers the best performance at grid sizes of 1024*1024 and above.

Once an FOV has been calculated, it may be updated instead of being calculated again from scratch. If an environment has a large grid size, FOV Update is more efficient than calculating an FOV from scratch, regardless of the structure of the environment. FOV Update offers better performance than either calculation algorithms at grid sizes of 1024*1024 and above but must still depend on one of them to calculate an initial FOV.

Chapter 5

5 Conclusions and Future Work

In this chapter we summarize our work and propose avenues for future research.

5.1 Conclusions

In this paper we have examined the existing state of the art for field of vision calculation, described flaws in existing algorithms, and have proposed new algorithms which address these flaws.

We evaluated existing FOV algorithms and concluded that Recursive Shadowcasting has the best performance among them. We also showed performance issues with the doryen implementation of Recursive Shadowcasting and demonstrated an error in the description of the Recursive Shadowcasting algorithm.

We then described an FOV algorithm of our own design: Rectangle-Based FOV. By utilizing a quadtree of rectangles, Rectangle FOV is able to assign cell visibility statuses in an efficient manner.

Finally, we described an algorithm which updates an existing FOV for a new source position, rather than calculating a new FOV from scratch. By updating an existing FOV, our algorithm is able to assign very few cell visibility statuses, even at high grid sizes.

We conducted an experimental analysis which compared Recursive Shadowcasting, Rectangle FOV, and FOV Update. The analysis concluded that no algorithm is the best in all cases, and that an ideal approach is a hybrid of all three algorithms: Recursive Shadowcasting should be used in indoors environments, or at grid sizes up to 512*512, otherwise Rectangle FOV should be used. Once an initial FOV is calculated, FOV Update should be used to update that FOV at grid sizes of 1024*1024 and above.

The implementations of FOV algorithms which were used in this thesis are open source and are provided with compilation instructions at: http://www.csd.uwo.ca/faculty/solis/software/fov/fov.html.

5.2 Potential Future Work

The implementations of the algorithms we have described are available to implementors, but some implementors may have specific requirements or they might know that their games will feature specific types of environments. Such implementors may wish to modify our algorithms to adjust them to their needs. We detail some modifications which implementors could wish to make below.

Both Rectangle FOV and FOV Update use the shadowcast visibility definition. We chose this definition as it is the most popular and is used by the best performing existing FOV algorithm. However, some game implementors may wish to use a different visibility definition such as strict FOV or Permissive FOV.

Adjusting Rectangle FOV or FOV Update to output an FOV according to the strict visibility definition should be relatively simple. An implementor would need to modify how the algorithm determines which grid cells are considered to be within a rectangle's occluded region or within a cone, as strict FOV requires the center of a cell, and not just any point within a cell, to be visible. No changes would need to be made to how rays are cast. One complication is that our algorithms assume that a rectangle cannot occlude part of its own face, yet that is not the case for strict FOV (see Figure 32). This could be addressed by checking which cells on a rectangle’s face are occluded by that rectangle. Adjusting our algorithms to use strict FOV should be the simplest way to ensure they produce an FOV which is symmetrical.

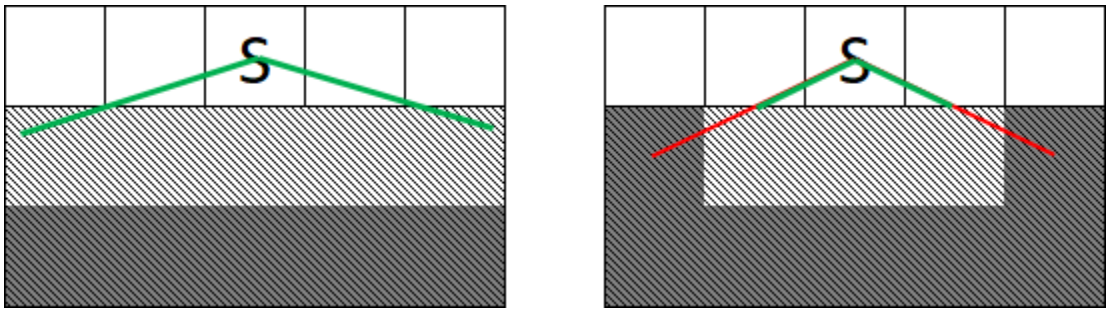

Figure 32: A simple environment with FOV calculated according to the shadowcast FOV definition(left), and the strict FOV definition(right). Visibility rays are drawn to highlight the differences in the FOV produced by each definition.

Modifying our algorithms to produce output according to the permissive visibility definition of FOV is more complicated. No changes need to be made to which cells are considered to be within a cone or occluded region, but changes would need to be made to the location rays are cast from. Permissive FOV does not cast rays from a constant point within the FOV source cell, but instead uses the point within the source cell which results in the most visible space. The points where rays must be cast from can be easily calculated if a single rectangle is considered in isolation, but the process becomes complicated when many rectangles are considered. A point q within the FOV source cell must be chosen such that the visible space is maximized, while also ensuring that the line between q and a rectangle's relevant point does not intersect any other rectangles. This complication does not apply to the other two FOV definitions, which always cast rays from the center of the FOV source cell. The description of Permissive FOV in [15] may serve as a starting point for implementors who wish to pursue this.

If an implementor is able to assume that their game will always take place in enclosed indoor environments, then they could significantly improve the running-time of Rectangle FOV or FOV update by using portal-based culling and potentially visible sets [9]. If an environment can be represented as a series of rooms connected via portals (such as doors or windows) then said environment can be represented as a graph where rooms are nodes and portals are edges. These rooms are then pre-processed to generate a 'potentially visible set' for a given room R. This potentially visible set is a collection of all rooms which are visible from at least one point in R. The set would include at least all nodes adjacent to R in the graph. Portal-based culling is often used as a form of occlusion culling in computer games. When rendering graphics, a game is able to skip the rendering of all objects which are outside of rooms in the potentially visible set [8]. Rectangles outside of the potentially visible set could be ignored, which could result in a significant performance improvement. We chose not to pursue this optimization ourselves as it would add significant complexity to the algorithms and it is ineffective in many environments. We did not want to make specific assumptions about the environments our algorithms may be used in.

Our description of FOV Update does consider changes to vision-blocking terrain in an environment. We chose to focus our research on cases where only the FOV source changes, as this is by far the most common case where FOV must be calculated. In most cases when a game environment does change it changes dramatically, such as when the player moves to a different level within a game. In such cases the previously calculated FOV is not useful and an FOV calculation algorithm must be used.

However, in cases where a game environment only changes slightly, it may be worth updating the previously calculated FOV to accommodate this environmental change. If rectangles are added to an environment, then Rectangle FOV could simply be run only on those rectangles with the previous FOV grid as input (instead of an all-visible grid).

Processing the removal of a rectangle from an environment is more complicated. A process similar to inverting cones would need to be used, where the algorithm does not process regions which are occluded by other rectangles. This would set the region previously occluded by the removed rectangle to visible while ensuring that any regions occluded by other rectangles are not incorrectly set to visible.

Scene changes that require adding and removing multiple rectangles, may result in poor performance versus simply calculating an FOV from scratch. An implementor who is considering modifying FOV Update to work with changing environments should think carefully about whether using FOV Update in these cases will result in improved performance.

References

[1] Brace Yourself Games (2015). *Crypt of the Necrodancer*. Retrieved from: https://braceyourselfgames.com/crypt-of-the-necrodancer/

[2] Riot Games (2019). *League of Legends.* Retrieved from: https://na.leagueoflegends.com/en/

[3] Valve (2019). *Defense of The Ancients 2*. Retrieved from: http://blog.dota2.com

[4] Jung, J. (2016). *A Story of Fog and War.* Retrieved from: engineering.riotgames.com/news/story-fog-and-war

[5] Straßer, W. (1974). *Schnelle Kurven- und Flächendarstellung auf grafischen Sichtgeräten.* Retrieved from: https://diglib.eg.org/handle/10.2312/2631196

[6] Foley, J.D. & van Dam, A. (1982). *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[7] Kronos Group (2018). *Depth Test.* Retrieved from: https://www.khronos.org/opengl/wiki/Depth_Test

[8] Clark, J.H. (1976). *Hierarchical geometric models for visible surface algorithms.* Retrieved from: https://dl.acm.org/citation.cfm?id=360354

[9] Luebke, D.; Georges C. (1995). *Portals and mirrors: simple, fast evaluation of potentially visible sets.* Retrieved from: https://dl.acm.org/citation.cfm?id=199422

[10] Williams, L. (1978). *Casting curved shadows on curved surfaces.* Retrieved from: https://dl-acm-org.ca/citation.cfm?id=807402

[11] Crow, F. (1977). *Shadow Algorithms for Computer Graphics.* Retrieved from: https://dl.acm.org/citation.cfm?id=563901

[12] Kronos Group (2016). *Synchronization*. Retrieved from: https://www.khronos.org/opengl/wiki/Synchronization

[13] Nvidia (2019). *CUDA Zone*. Retrieved from: https://developer.nvidia.com/cuda-zone

[14] Bergström, B. (2001). *FOV using recursive shadowcasting*. Retrieved from: www.roguebasin.com/index.php?title=FOV_using_recursive_shadowcasting

[15] Duerig, J. (2007). *Precise Permissive Field of View*. Retrieved from: https://web.archive.org/web/20170708200308/www.roguebasin.com/index.php?title=Precise_Permissive_Field_of_View

[16] Jice. (2009). *Comparative study of field of view algorithms for 2D grid based worlds* Retrieved from: http://roguecentral.org/doryen/data/libtcod/fov.pdf

[17] Jice, contributors. (2019). *libtcod*. Retrieved from: https://github.com/libtcod/libtcod

[18] T. Ohtsuki. (1982). *Minimum dissection of rectilinear regions*. IEEE Conference on Circuits and Systems

[19] Hopcroft, John E.; Karp, Richard M. (1973), *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*. SIAM Journal on Computing

[20] Raphael, R. & Bentley, J.L. (1975). *Quad Trees: A Data Structure for Retrieval on Composite Keys*. Retrieved from: https://link.springer.com/article/10.1007%2FBF00288933

Curriculum Vitae

Name:	Evan Robert Macleod Debenham
Post-secondary Education and Degrees:	M.Sc. Candidate, Computer Science The University of Western Ontario 2017-present
	B.Sc. Computer Science The University of Western Ontario 2011-2013, 2015
Honours and Awards:	Ontario Graduate Scholarship The University of Western Ontario 2017-2018
	Dean's Honor List The University of Western Ontario 2012-2014, 2016
Related Work Experience	Teaching Assistant The University of Western Ontario 2017-2019