CGI2012 manuscript No.
(will be inserted by the editor)

Real-time Synthetic Vision Visibility Testing on Arbitrary Surfaces

Abstract One of the key features of 2D real-time strategy (RTS) games like *StarCraft II* or serious games for training is visibility testing for each unit. Visibility testing provides functionality like fog of war, non-player character (NPC) activation, and realistic artificial intelligence. With the advent of path planning and crowd simulation algorithms on arbitrary surfaces, the technology is almost available for RTS games to move from the 2D plane to complex surfaces like multi-story buildings, subways, asteroids, space ships, or surfaces where insects swarm, such as a hive or colony. We propose a novel synthetic vision approach to visibility testing that allows features like fog of war for RTS games on arbitrary surfaces. Unlike previous synthetic vision algorithms, our algorithm is specifically designed for non-planar surfaces and surfaces that have not been labeled manually, in addition to meeting the real-time demands of RTS games. Our resulting algorithm does visibility testing for up to 100 or more units in real-time on complex surfaces and allows RTS and serious games to use non-planar surfaces in a manner that has not been possible before.

Keywords visibility testing, synthetic vision, fog of war, 3D crowd simulation

1 Introduction

One of the key elements that computers brought to the gaming world is visibility testing. Unlike board games, which often give both players perfect information about the position of all characters, computer games can limit the visibility of each player, leading to a captivating gaming experience.

Visibility testing is most commonly recognized in a game's fog of war. In a real-time strategy game (RTS) or similar serious game, the player can only see the terrain and enemy characters that are within the line of sight of player-controlled agents. Visibility testing and fog of war create games that mimic the true uncertainty and confusion of search and rescue missions and warfare. This realism leads to a variety of intriguing strategies since players must use resources and agents to disperse the fog of war and may use the fog of war to their advantage against their opponent. Without the fog of war, RTS games would lose their attraction and serious games would lose their ability to model the confusion and uncertainty of the training experience.

The importance of visibility testing has lead to its ubiquity in RTS and serious games. However, most of these games focus on essentially flat, 2D environments since this makes it easy to do path planning, obstacle avoidance, and similar computations in real-time. Unfortunately, restricting RTS and serious games to a plane or a plane plus height map means the virtual environments of these games cannot reflect many of the complex surfaces we walk on every day. For example, a planar environment cannot capture the complexity of architectural structures like multi-story buildings, parking garages, subway stations, and houses; environmental features like intertwining caves, overhangs, and arches; outer-space environments like asteroids and space stations; nor the various surfaces where insects swarm like their colonies or on the inside of houses.

Moving from the 2D plane to fully-3D environments totally revolutionized the first-person shooter experience. Going from the repetitive, flat levels of *Wolfenstein 3D* to the complexity and realism of *Doom* and its successors spawned one of the most successful video games genres in history. With the advent of algorithms

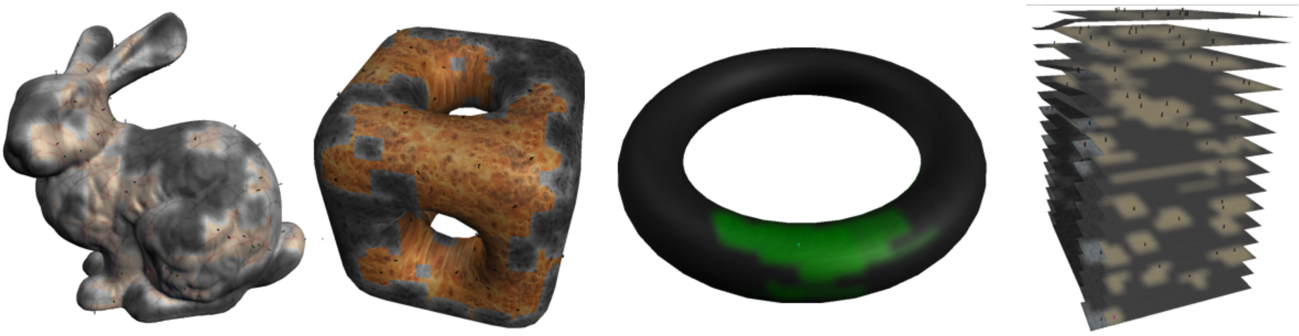

Fig. 1 Examples of visibility testing in the form of fog of war on complex 3D surfaces. Our algorithm uses synthetic vision to do real-time visibility testing for hundreds of agents, allowing for an immersive and realistic gaming experience, even on non-planar surfaces. This example shows results using the Stanford bunny, a highly-convex surface, a torus, and a fifteen-story building all with our algorithm. More examples can be seen in Figure 8. (Stanford bunny data courtesy Stanford University Computer Graphics Laboratory.)

for real-time path planning and crowd simulation on arbitrary 3D surfaces, RTS and serious games are poised to make a similar dramatic change. However, even with new algorithms for moving crowds of people, troops, or animals on complex surfaces, the move from 2D RTS games to 3D ones will be impossible without visibility testing. To remedy this, we propose a real-time method for visibility testing for RTS-style games using synthetic vision.

Our proposed synthetic vision algorithm for visibility testing is unique since it is designed for non-planar surfaces. Unlike other slower like ray-casting, our GPU-based implementation provides both a fast and accurate method for determining visibility. As a result, it handles visibility testing across a wide range of complex 3D surfaces in real-time (see Figure 1). By leveraging the features of modern GPUs, our visibility testing can work with up to 200 agents, creating a dramatic and believable fog of war experience on arbitrary surfaces. Although the focus of this work is on visibility testing for fog of war, the visibility testing algorithm proposed is capable of extension for a variety of other uses like NPC activation and realistic artificial intelligence.

2 Previous Work

To set the context for visibility testing on arbitrary surfaces, we begin with the evolution of path planning and obstacle avoidance algorithms from 2D to 3D followed by specific algorithms for crowds on 3D surfaces like those needed in a 3D RTS game. We then compare and contrast previous synthetic vision algorithms.

2.1 Global Path Planning

Most 2D path planning algorithms have their roots in graph-based path planners like Dijkstra's or A*. These algorithms run quickly, but if left unoptimized the resulting paths are inherently jagged and unnatural. Some algorithms moved to more accurate planners like fast marching methods [28] while others straighten paths using line-of-sight smoothing [30]. As Geraert concludes, even with an accurate path planner, agent motion is most natural when it is a corridor, not just a line [7]. Such algorithms work very well for essentially 2D games, but not for arbitrary 3D surfaces.

Early 3D path planning solutions came from Mitchell et al. [20] and Chen and Han [3], both of which have had countless extensions proposed in the intervening years (with Bose et al. [2] writing an excellent survey). Other unique approaches include Martinez et al. [19] and Kanai and Suzuki [11] who find geodesics by starting with an approximation and iteratively refining the path. Other work breaks down the mesh into quasi-developable regions and then finds paths [31]. A quite powerful and fast discrete geodesic method was proposed by Kimmel and Sethian [13] based on the fast marching method. Lai and Cheng recently proposed an unfolding method for finding discrete geodesics [15] on subdivision surfaces. As we discuss later in Section 3, if these 3D path planning algorithms are not optimized, they unfortunately do not provide natural crowds for RTS or similar serious games on arbitrary surfaces.

2.2 Local Obstacle Avoidance

Another key component of RTS games are local obstacle avoidance techniques that keep agents from running through each other or into walls. Most 2D local obstacle avoidance techniques trace their roots to Reynolds' work on flocking dynamics [26]. Helbing and Molnár [10, 9] extended this to social forces for human crowds on a 2D plane. To address the jamming inherent in spring-based methods, Fiorini and Shiller [5] proposed velocity obstacles as a more realistic method, which was further improved by van den Berg et al. [1] who proposed reciprocal velocity obstacles (RVO) to remove agent oscillations. Recently, Guy et al. [8] changed RVO to optimize over agent effort, which accurately produces global effects like agents walking faster near walls.

Other steering algorithms have been based on observing how human's walk. For example, Lee et al. [17] learned local obstacle avoidance behavior and group interactions from video. Olivier et al. [23] evaluated how people moved in response to a character shown on a video screen which was used to build an anticipation algorithm [24] based on optical flow. Other work has focused on the speed of crowd simulation. For example, Narain et al. [21] simulate tens of thousands of agents at interactive rates in dense settings by combining both Euclidean and Lagrangian methods. Karamouzas et al. [12] achieve similar speed results with their collision prediction model. Pelechano et al.'s HiDAC algorithm [25] produces believable results in huge crowd situations. Reynold's work with the Play Station 3 can simulate 10,000 agents at 60 frames per second [27].

Like path planning algorithms, 2D obstacle avoidance algorithms are mature, but need to be extended to work on 3D surfaces if they are to be used in RTS games on complex surfaces. Fortunately, some recent work has begun to do 3D path planning and obstacle avoidance for crowd simulation on arbitrary surfaces.

2.3 Crowds on Arbitrary Topologies

Building on 3D path planning and 2D obstacle avoidance work, recent years have seen a rising interest in crowd simulation on arbitrary topologies. As mentioned in the introduction, crowd algorithms for arbitrary topologies would be able to model crowds on such relevant and diverse surfaces as office buildings, subways, homes, and stadiums. An effective 3D crowd simulation algorithm could then be used as the backbone for RTS games on arbitrary surfaces by providing realistic path planning and movement controls.

Most algorithms that put crowds on arbitrary surfaces break down the scene into separate 2D pieces that are connected at edges. Shao and Terzopoulos [29] created a model of New York City's original Pennsylvania station where "The representation assumes that the walkable surface in a region may be mapped onto a horizontal plane...thereby enhancing the simplicity and efficiency of environmental queries." More recently Lamarche [16] proposed a method for character motion and animation with advanced features like ducking under ceilings. However, like Shao and Terzopoulos, the algorithm approximates the 3D surface with a 2D plane. Other work breaks down surfaces into 2.5 dimensions [4], but the addition of a height map does not capture the complexity of surfaces like multi-story buildings. Since flattening cannot robustly handle arbitrary surfaces, these algorithms are limited in their ability to be the core of RTS games.

Other work that is not constrained to planar surfaces includes Fischer et al. [6] who propose a planning algorithm for factories with ramps and stairs, but the work's focus on planning for automated factories means it is not immediately applicable to our goal of RTS games with a large number of agents. Levine et al.'s work [18] provides an algorithm for character locomotion through complex environments with dynamic obstacles. Like Lamarche's work, Levine's work produces good results, but it is not designed for the large agent requirements of games.

A more successful and general approach to crowds on 3D surfaces was done by Torchelsen et al. [32]. This work uses a discrete geodesic method to move agents across smooth meshes while doing local obstacle avoidance on the GPU. Unfortunately, this algorithm only allows a limited number of destinations for all the agents. This makes it difficult for RTS games where the player can assign arbitrary destinations to agents. To resolve these issues, we have developed a new 3D crowd simulation algorithm that does not flatten surfaces, works with hundreds of agents in real-time on high-polygon count surfaces, and allows for surfaces with sharp turns and corners. This is the engine we used for moving agents in our RTS scenario, and we describe it in detail in Section 3.

2.4 Previous Synthetic Vision Work

Synthetic vision has been a consistent theme in crowd simulation for over a decade. Unlike most previous work, we are not using synthetic vision for the path planning or local obstacle avoidance parts of our work. Instead, synthetic vision plays a key role in informing the user about the state of the game. For example, [22] use synthetic vision for global navigation via an oct tree and local navigation.

Kuffner et al. [14] use synthetic vision to find and update the positions of obstacles in their environment. Each object has a unique identifier that is precomputed and the synthetic view of each agent is colored based on the id of the object present. The colors are processed

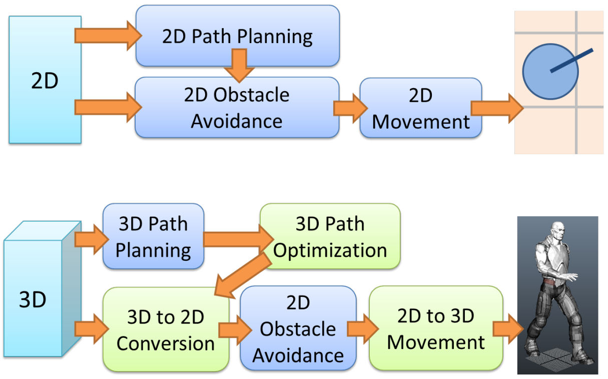

Fig. 2 Classic 2D crowd simulation framework (top) and our proposed framework for 3D surfaces (bottom). This framework provides the backbone to our RTS simulation.

to determine which objects can be seen where, and the memory of the agent is updated accordingly.

3 3D Crowd Simulation Implementation

[cite myself]

The foundation of any RTS game is an engine for agent path planning and obstacle avoidance. In order to have groups of agents on arbitrary arbitrary surfaces, we use our algorithm for 3D crowds simulation, which we describe briefly in this section. (Note that when we say 3D crowd simulation, we mean crowds constrained to surfaces in 3D space, not flock with unconstrained 3D movement.) This 3D framework mirrors the traditional 2D crowd simulation framework shown in Figure 2 with a path planning piece and a local obstacle avoidance piece followed by a movement piece. This algorithm is flexible enough to allow for almost any 3D path planning algorithm and 2D obstacle avoidance algorithm, as long as the results are optimized for crowd simulation. As we discuss in our results section, use of this engine provides believable, real-time movement for our RTS strategy agents.

3.1 Optimized 3D Path Planning

Our simulation engine optimizes 3D path planning algorithms since unoptimized algorithms fail to naturally move agents in real-time. Symptoms of unoptimized path planning include paths that are jagged, agents that get confused around corners, and agents that easily jam. For each of these problems we provide an optimization, as shown in Figure 3. The first problem, jagged paths, comes since not all 3D path planners produce perfectly smooth paths. Needing a fast but slightly inaccurate method would not be surprising given the real-time demands of RTS games. Our first optimization quickly straightens out agent movement even if a path has unnatural turns. The next problem is that an agent can get confused or stall near edges even with a perfect path. For example, in a two-story building every path running up or down the stairs overlaps on the inside of the stairway. This results in stalling as agents try to follow a path right along an edge or wall. Our second optimization analyzes the geometry of the 3D environment to push paths away from edges. The last problem is that around corners and bottlenecks paths from multiple agents tend to lie on top of each other, leading to congestion and jams. Our third optimization resolves this by giving agents corridors, which gives them the additional freedom they need to form lanes and pass naturally.

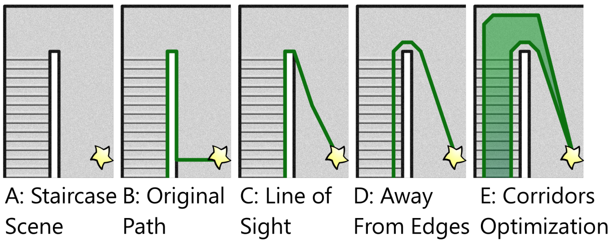

Fig. 3 The three optimizations in the path planning part of our framework. A: A scenario with an agent (star) who needs to go up the stairs. B: The initial 3D path provided by a graph-based algorithm with an unnatural turn. C: The first optimization smooths the path using a line of sight approximation. D: The second optimization moves the path away from the edge by analyzing triangle edges. E: The last optimization provides a corridor for the agent, thus reducing jamming in the presence of other agents.

Empirically, these three optimizations dramatically decrease the amount of jamming and stalling in an environment with edges. In our tests, without the away from edges optimization, almost a full fifth of the agents stalled. With the away from edges optimization, the percent dropped to less than one and a half percent,

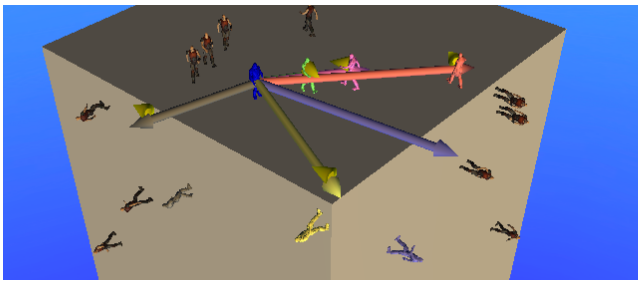

Fig. 4 How our framework converts 3D offsets to 2D values. The agent highlighted in blue responds to agents in front of him, which are highlighted different colors. The colored arrows correspond to the calculated 2D distance and angles to the agent of the respective color and become the 2D values used by the local obstacle avoidance algorithm.

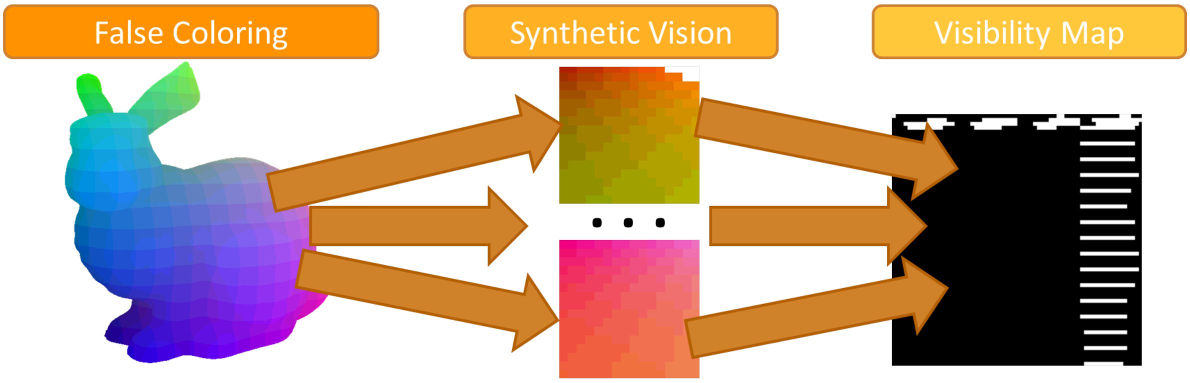

Fig. 5 The flow of our visibility testing algorithm. The viewport of each agent is rendered using our false coloring shader. The visibility texture is then calculated based on these viewports. The visibility texture is then used to render the surface and determine which agents are visible.

but jamming was still noticeable. With all optimizations, the percentage dropped to one tenth of one percent. Both visually and numerically the improvement was dramatic. This meant that the player-controlled characters and NPCs in our RTS game moved naturally and quickly to their destinations.

3.2 Local Obstacle Avoidance

The second major piece of our framework required for our RTS games does local obstacle avoidance. To do this on arbitrary surfaces, our algorithm converts the location and offsets of nearby obstacles around an agent from their true 3D values to simplified 2D values (as shown in Figure 4). The resulting 2D values are fed into any of a number of traditional 2D obstacle avoidance algorithms. The resulting 2D change in position and heading are then optimized for the curvature of the 3D surface. Empirically, this leads to real-time crowd movement even on surfaces with over 100,000 triangles.

All of the 3D path planning and obstacle avoidance optimizations are done without flattening, so the resulting crowds move smoothly and without distortion even around complicated parts of surfaces such as the corners of stairs. For our RTS scenario, we used these optimizations on top of an A* weighted algorithm for 3D path planning and RVO for our local obstacle avoidance algorithm. The resulting crowds moved quickly towards their destinations and had an almost zero percent collision rate.

4 Visibility Testing Implementation

Using our crowd simulation as the engine for moving both player-controlled and computer-controlled characters, we were able to do realistic visibility testing for agents in real-time in an RTS scenario. In this section we discuss our visibility testing algorithm in general, and in the next section (Section 5) we discuss how we used this testing to implement fog of war and other features.

The goal of visibility testing is to determine what player-controller agents can see and what they have been able to see in the past. As previously discussed, this allows for accurate fog of war, NPC activation, and computer artificial intelligence. Some options for doing visibility testing on 2D surfaces, like ray-casting, are too slow in the 3D case since the number of ray casts required for 3D surfaces is too large. Similarly, merely assuming that agents can see everything within a distance threshold is unrealistic since it does not account for self-occluding surfaces like ones with corners, turns, and walls. To resolve these issue of speed and accuracy, we propose a synthetic vision solution that is both fast and accurate.

As an overview of our visibility testing algorithm, see Figure 5. As shown on the left, we use a false coloring shader to color the surface when it is rendered by the virtual camera's place at the heads of player-controlled agents. Shown in the middle are examples of what individual agents may see as rendered to a texture. On the right is shown the resulting visibility map that is generated on the GPU and can be used either on the CPU or GPU for various needs like fog of war, NPC activation, and artificial intelligence. In the next three subsections we discuss each part of this process in turn.

4.1 False Coloring

Synthetic vision allows us to calculate a realistic view of what each agent sees in real-time. To do this we built a

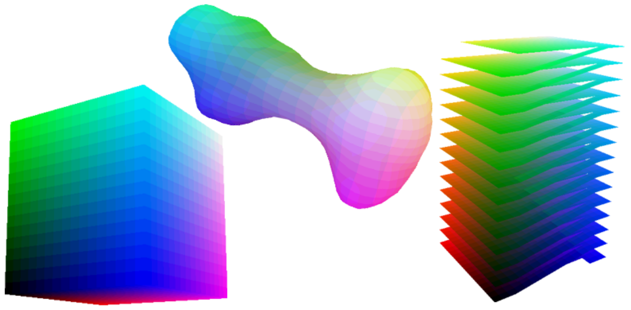

Fig. 6 Three surfaces shaded with our false coloring scheme: a cube, the asteroid 216 Kleopatra, and a 15-story building. Asteroid models courtesy Scott Hudson, Washington State University, http://users.tricity.wsu.edu/h̃udson/Research/Asteroids/models.html

shader that colors each pixel of the surface based on its position in world space. Unlike Kuffner and Latombe [14] and others, we do not label each piece of the surface and render each piece with a unique color since this may require some manual intervention in terms of labeling and since we do not want whole pieces of the surface to become visible if just one corner can be seen. Instead, we use a positional false coloring shader that colors each part of the surface based solely on its position in space.

Our positional shader divides the surface bounding box into a 16x16x16 grid and assigns each voxel in that grid a unique color (see Figure 6). In the pixel shader routine of our false coloring shader, the 3D world space position of the current pixel is stored. This position is converted into percentages in x, y, and z based on the bounding volume that encompasses the surface. Thus, if a pixel represents a part of our scene that is near the bottom left back of our surface, the percentages will all be near 0, and if a pixel is near the top right front of our surface then the percentages will all be near 1. These percentages are then converted into numbers between 0 and 15 inclusively, which give the voxel coordinates of the pixel. This 3D coordinate is converted into 2D space and a lookup is done in a reference texture, which stores the color for that voxel. The pixel is then assigned that color without consideration of lighting, etc.

4.2 Synthetic Vision Shader and Multiple Agents

Key to our algorithm is the synthetic vision piece that renders the viewport of each agent to a texture which is then interpreted by the visibility shader. To create synthetic vision, on each frame a virtual camera was placed above the player-controlled agent pointing slightly down. The world was then rendered using our false coloring shader onto a low resolution texture.

Visibility testing is most interesting and useful when it combines the visibility of many agents working together as a team. For example, in a military RTS game, certain units can be positioned to reveal areas of the map so other units can fire long-range weapons. In a serious game scenario, different units can be used to search different parts of the map for a missing person or a bomb. To reproduce this effect in our algorithm, we allowed multiple user-controlled agents to contribute to the visibility map.

To allow multiple contributing viewports, each player-controlled agent renders its own view. Once all the viewports were rendered, the visibility map shader looped over each viewport looking for visible voxels. This was done in a cumulative way so that if a voxel was visible in any viewport, it would be flagged as visible in the final visibility map.

4.3 Visibility Map

In the next step of our synthetic vision algorithm, a visibility texture is generated, which stores which voxels are visible to any agent. In the simple case, voxels which have been seen at any time are colored white while the rest are colored black. At the beginning of the program the visibility map is initialized to all black. On each frame the visibility shader loops over each agent's viewport. Since each pixel in the visibility map represents a voxel in space, the shader first determines the color that would be present if any of the agents can see that voxel. The shader then samples the viewports looking for that color. If the color is found (or was seen in a previous frame), the current pixel is colored white. If not it remains black.

This process creates a very realistic visibility map in real-time. To determine if a part of the surface is visible, the GPU or CPU only needs to consult this visibility map texture and check for black and white pixels. Note also that this process does not depend on the tessellation of the surface, nor are parts of the surface manually colored or numbered as in other synthetic vision approaches. Additionally, this algorithm accurately calculates visibility even on self-occluding surfaces, such as around corners and walls in a building.

5 Fog of War Implementation

As a direct application of our visibility testing, we used the visibility map to draw fog of war on our surfaces and to determine the visibility of NPCs in our RTS scenario.

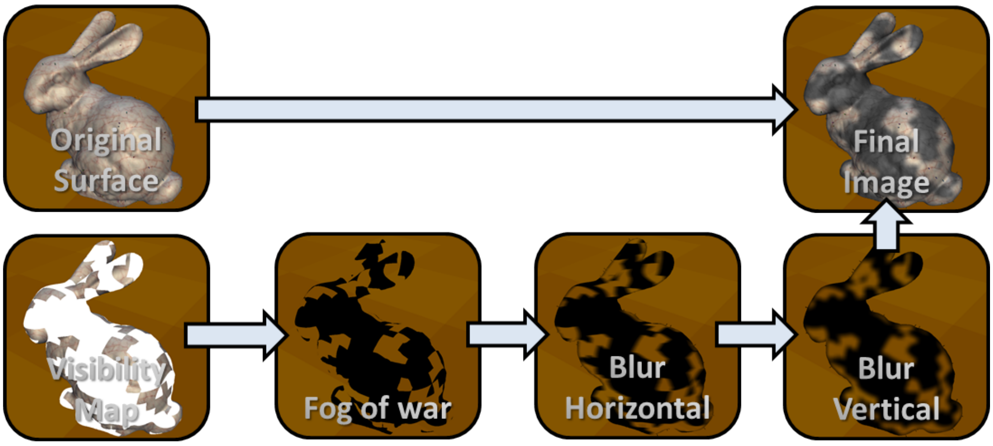

Fig. 7 Process for soft fog of war shadows. We begin by rendering the scene without any fog. We then render the surface again with our fog of war shader. We blur the fog of war horizontally and vertically, using the original surface render as our clipping bounds. The surface, the soft fog of war, and the background are then combined for the final image.

5.1 Fog of War Shader

The final shader in our synthetic vision algorithm shades the surface using a fog of war algorithm. We were faced with two options in this process: rewrite all the surface shaders we already had and change them to consult the visibility texture or find a way to draw the surface twice, once with our unchanged previous surface shaders and once with a new shader. The former idea would be tedious and would not result in a flexible algorithm. Instead, we followed the latter course and developed a final shader for changing the appearance of the surface based on the visibility map.

The fog of war shader takes the surface geometry and the visibility texture and colors the playing surface black if that area is in a black voxel and draws a transparent pixel if the voxel is white. The black pixels obscure the underlying surface where the fog of war is, and the transparent pixels allow the surface to show through where the surface is revealed. To prevent z-buffer fighting, the fog of war is drawn with a small z bias that brings it slightly closer to the viewing plane.

5.2 Softening the Fog of War

This fog of war process is accurate, but the results look unnatural since it has sharp edges where the fog of war starts and ends. To enhance the idea that the fog of war is fuzzy, we provide a series of shaders that soften the fog of war much like shadows are softened by rendering the surface and the fog of war separately (see Figure 7). First, the scene is rendered using the normal shader. In a separate buffer the visibility map is used to render the fog of war. Next, the fog of war is blurred both horizontally and vertically. To make sure the fog does not get softened outside the boundaries of the surface, pixels are not blurred if they are background pixels. Finally, the background is rendered to its own buffer and the surface, the fog of war, and the background are all composed.

Notice that when a similar approach is used for softened shadows in video games, the shadow often struggles with bleeding as the shadow is blurred onto foreground objects. In the context of fog of war, this often is a feature, not a problem. With our blurring shaders, NPC characters that are on the edge of the fog of war are partially obscured by the blur. This is exactly the effect we want since it gives a clear visual cue that these agents are on the boundary of the known and unknown.

5.3 Other Fog of War Types

The fog of war described so far is simplistic in the sense that once a part of the surface has been revealed, it stays revealed permanently. The approach taken in more modern RTS games like *StarCraft II* is to have areas of the surface that have not been visible for a long time slowly fade out and return to being fully obscure. Dealing with this means changing the visibility map from a binary texture to a gray-scale texture. In the gray-scale case black means an area has never been seen, shades of gray means the area has been seen but cannot be seen now, and white means the area is visible on this frame. To calculate gray-scale values, the visibility map from the previous frame is darkened slightly unless a voxel is currently visible, in which case it is colored white. In this way, areas that are not seen for a long time slowly return to a black color. In our fog of war implementation, we either treat dark gray voxels

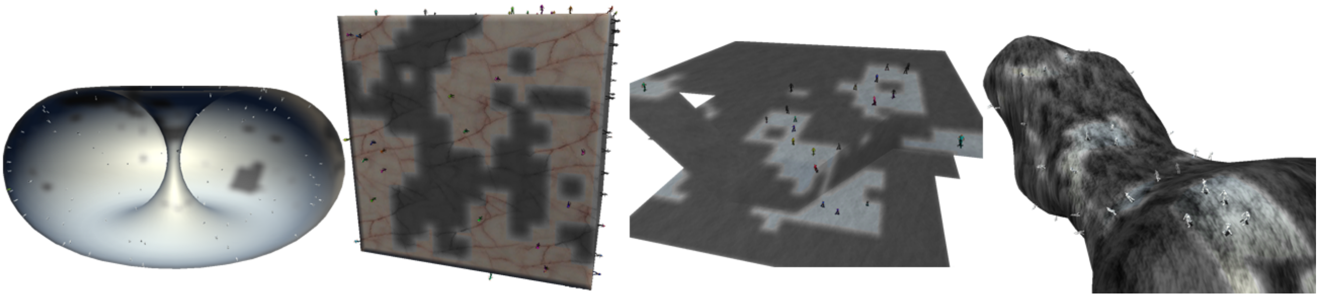

Fig. 8 Examples of our fog of war using our visibility map as a base. These include astronauts inside a space station, agents on a cube, shoppers in a two-story mall, and astronauts on the asteroid Kleopatra. More examples can be seen in Figure 1.

as totally obscure or slowly fade in the fog of war on voxels that are gray scale.

We are also able to easily alter other aspects of the fog of war. For example, in some RTS games the fog of war is totally black and thus obscures the underlying terrain. In other games the fog of war obscures NCPs, but shows the layout of the underlying terrain. We mimic both of these features by tweaking the alpha value of our fog of war: for total obscurity the fog of war has an alpha of 0, for partial terrain visibility the alpha is set to .2. We further emphasized which parts of the surface are obscured by the fog of war by coloring obscured parts of the surface in gray scale.

5.4 Other Features

In addition to revealing visible terrain, our algorithm also uses our visibility testing to determine which NPC characters are visible at a given frame. Unlike our fog of war, this process was done on the CPU, showing our algorithm's flexibility in terms of both CPU and GPU use.

In order to mimic modern RTS games, we added two features for hiding and revealing NPC characters: NPC characters should only be visible if they are within the revealed part of the map and NPC characters on the very border of the fog of war should be partially shaded. To accomplish this first goal, on every frame we query the GPU for the texture data of the visibility map. When each NPC character is drawn, the CPU code finds the NPC's voxel and then finds the color on the visibility map corresponding to that location. If the color is black or gray, the NPC is not drawn. If it is white, the NPC is in a revealed part of the map and it is drawn. As mentioned earlier, the second feature—shading agents on the edge of the fog of war—comes for free since we draw the fog of war after NPCs are drawn. If the NPC is on the edge, it is partially obscured, creating a very believable effect as they walk in and out of the revealed area.

6 Results

To validate our synthetic vision algorithm for visibility testing on arbitrary surfaces, we used it on a suite of surfaces using varying numbers of agents. Some of our results were qualitative, as shown in Figures 1 and 8. These figures show that our fog of war results were accurate, believable, and that NPC agents could only be seen where the map was revealed. The results are even more impressive in a time series as the fog of war gradually dissipates around the player-controlled agents. Another key result is that our algorithm determines visibility using the GPU, but the results can be read in real-time by the CPU. In our algorithm this is seen in the fact that the CPU determines which NPCs are visible based on a GPU calculated texture. This means that an RTS game built on our algorithm does not need to move all computation to the GPU. Additionally, our qualitative results include the fact that our algorithm for crowds on arbitrary surfaces works naturally as the engine for a RTS scenario with believable movement and with practically no jamming. We believe this further validates our 3D path planning and 2D local obstacle avoidance optimizations.

For a more quantitative analysis, we ran our algorithm on a benchmark of surfaces to determine the runtime of our visibility testing algorithm. Figure 9 shows a sampling of the most indicative results. For lower triangle-count surfaces, like the asteroid 216 Kleopatra with a little over 2,000 triangles, we could run our synthetic vision algorithm at 24 frames per second with up to 200 hundred agents. Moving to a higher-polygon count surface like our cube with 19,000 triangles, the run time stayed real-time up to about 150 agents. The Stanford bunny (with over 69,000 triangles) ran near real-time up to 100 agents. Some of our most interesting surfaces (like the 15-story building in Figure 1) had less than a thousand polygons, so even though it was a very large and intriguing playing field, it achieved high frames even with hundreds of agents.

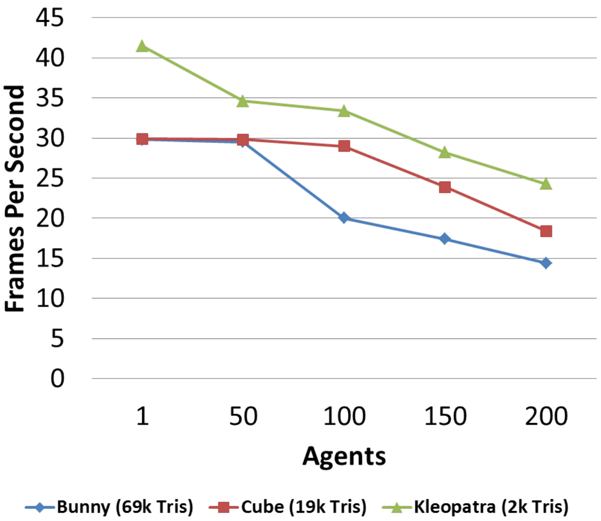

Fig. 9 Graph of our synthetic vision's performance based on the number of agents doing synthetic vision in real-time. Notice that even on high-triangle count surfaces, run-times were at or near real-time with at least a hundred agents.

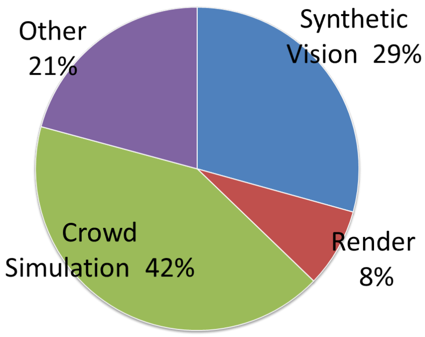

Fig. 10 Pie chart of where computation time was spent using our synthetic vision algorithm on the Stanford bunny with 100 agents. Notice that the synthetic vision piece only required a little more than a quarter of the computation time.

We also profiled our results to see where the computation time of our algorithm is spent. The pie chart in Figure 10 shows profiling results from running our visibility testing algorithm on the Stanford bunny with 100 agents contributing to visibility. Notice that only a little more than a quarter of our run-time was dedicated to the synthetic vision algorithm. This meant that almost 75 percent of the time was left over for other computation like path planning and obstacle avoidance.

All tests were run on our quad-core, 2.33GHz processor with one ATI Radeon HD 4600 series graphics card. All CPU code was written in managed C# and our shaders were written in HLSL. Our synthetic vision viewports were 16x16 and visibility map was 64x64. This meant that we did not come close to using all the available memory on our GPU.

From these results we conclude that synthetic vision provides a fast and realistic method for visibility testing on complex surfaces for RTS games. When used to create fog of war, the results are very believable and can be rendered in real-time.

7 Conclusion and Future Work

We have presented a real-time algorithm for visibility testing on arbitrarily complex 3D surfaces. This technology has the potential to compliment existing 3D crowd simulation algorithms in the creation of RTS and serious games on complex surfaces by allowing fog of war, NPC activation, and line-of-sight based artificial intelligence. Our synthetic vision approach leverages the GPU and could have up to 200 agents contributing to the visibility map in real-time.

There are a variety of ways to speed up our synthetic vision algorithm even further. One approach would be to poll agents every few frames for visibility testing instead of checking each agent on ever frame. Similarly, the full visibility texture could be updated periodically instead of every frame.

Our future work is focused on additional technologies that would aid in the creation of 3D RTS games including intelligent camera movement. In most 2D games the camera moves by panning across a flat plane. With the addition of the 3D dimension to the playing surface simply panning is not sufficient. We are interested in intuitive and easy camera controls that allow the player to move around complex surfaces rapidly. Similarly, many RTS games are augmented with a mini-map that shows a small 2D view of the entire surface. Since most 3D surfaces cannot be flattened without distortion, we are also investigating mini-map technologies for complex surfaces.

References

1. van den Berg, J., Lin, M., Manocha, D.: Reciprocal velocity obstacles for real-time multi-agent navigation. Robotics and Automation pp. 1928–1935 (2008)
2. Bose, P., Maheshwari, A., Shu, C., Wuhrer, S.: A survey of geodesic paths on 3d surfaces. Computational Geometry **44**(9), 486–498 (2011)
3. Chen, J., Han, Y.: Shortest paths on a polyhedron. Computational Geometry pp. 360–369 (1990)
4. Cupec, R., Aleksi, I., Schmidt, G.: Step sequence planning for a biped robot by means of a cylindrical shape model and a high-resolution 2.5 d map. Robotics and Autonomous Systems (2010)
5. Fiorini, P., Shiller, Z.: Motion planning in dynamic environments using velocity obstacles. The International Journal of Robotics Research **17**(7), 760 (1998)

6. Fischer, M., Renken, H., Laroque, C., Dangelmaier, W., Schaumann, G.: Automated 3d-motion planning for ramps and stairs in intra-logistics material flow simulations. Winter Simulation Conference pp. 1648 –1660 (2010)
7. Geraerts, R., Overmars, M.: The corridor map method: Real-time high-quality path planning. Robotics and Automation pp. 1023–1028 (2007)
8. Guy, S., Chhugani, J., Curtis, S., Dubey, P., Lin, M., Manocha, D.: Pledestrians: A least-effort approach to crowd simulation. ACM SIGGRAPH/Eurographics Symposium on Computer Animation pp. 119–128 (2010)
9. Helbing, D., Farkas, I., Vicsek, T.: Simulating dynamical features of escape panic. Nature **407**, 487–490 (2000)
10. Helbing, D., Molnár, P.: Social force model for pedestrian dynamics. Physical Review E **51**(5), 4282–4286 (1995)
11. Kanai, T., Suzuki, H.: Approximate shortest path on a polyhedral surface and its applications. Computer-Aided Design **33**(11), 801–811 (2001)
12. Karamouzas, I., Heil, P., van Beek, P., Overmars, M.: A predictive collision avoidance model for pedestrian simulation. Motion in Games pp. 41–52 (2009)
13. Kimmel, R., Sethian, J.: Computing geodesic paths on manifolds. National Academy of Sciences of the United States of America **95**(15), 8431 (1998)
14. Kuffner, J., Latombe, J.: Fast synthetic vision, memory, and learning models for virtual humans. Computer Animation pp. 118–127 (1999)
15. Lai, S., Cheng, F.: Approximate geodesics on smooth surfaces of arbitrary topology. Computer-Aided Design (2011)
16. Lamarche, F.: Topoplan: a topological path planner for real time human navigation under floor and ceiling constraints. Computer Graphics Forum **28**, 649–658 (2009)
17. Lee, K., Choi, M., Hong, Q., Lee, J.: Group behavior from video: a data-driven approach to crowd simulation. ACM SIGGRAPH/Eurographics symposium on Computer animation pp. 109–118 (2007)
18. Levine, S., Lee, Y., Koltun, V., Popović, Z.: Spacetime planning with parameterized locomotion controllers. ACM Transactions on Graphics **30**(3), 23 (2011)
19. Martínez, D., Velho, L., Carvalho, P.: Computing geodesics on triangular meshes. Computers & Graphics **29**(5), 667–675 (2005)
20. Mitchell, J., Mount, D., Papadimitriou, C.: The discrete geodesic problem. SIAM Journal on Computing **16**, 647 (1987)
21. Narain, R., Golas, A., Curtis, S., Lin, M.: Aggregate dynamics for dense crowd simulation. ACM Transactions on Graphics **28**(5), 1–8 (2009)
22. Noser, H., Renault, O., Thalmann, D., Thalmann, N.: Navigation for digital actors based on synthetic vision, memory, and learning. Computers & graphics **19**(1), 7–19 (1995)
23. Olivier, A., Ondřej, J., Pettré, J., Kulpa, R., Crétual, A.: Interaction between real and virtual humans during walking: perceptual evluation of a simple device. Applied Perception in Graphics and Visualization pp. 117–124 (2010)
24. Ondřej, J., Pettré, J., Olivier, A., Donikian, S.: A synthetic-vision based steering approach for crowd simulation. ACM Transactions on Graphics **29**(4), 1–9 (2010)
25. Pelechano, N., Allbeck, J., Badler, N.: Controlling individual agents in high-density crowd simulation. ACM SIGGRAPH/Eurographics symposium on Computer animation pp. 99–108 (2007)
26. Reynolds, C.: Flocks, herds and schools: A distributed behavioral model. Computer Graphics and Interactive Techniques pp. 25–34 (1987)
27. Reynolds, C.: Big fast crowds on ps3. ACM SIGGRAPH symposium on Videogames pp. 113–121 (2006)
28. Sethian, J.: Level set methods and fast marching methods. Cambridge University Press (2003)
29. Shao, W., Terzopoulos, D.: Autonomous pedestrians. ACM SIGGRAPH/Eurographics symposium on Computer animation pp. 19–28 (2005)
30. Singh, S., Kapadia, M., Faloutsos, P., Reinman, G.: Steerbench: a benchmark suite for evaluating steering behaviors. Computer Animation and Virtual Worlds **20**(5-6), 533–548 (2009)
31. Torchelsen, R., Pinto, F., Bastos, R., Comba, J.: Approximate on-surface distance computation using quasi-developable charts. Computer Graphics Forum **28**, 1781–1789 (2009)
32. Torchelsen, R., Scheidegger, L., Oliveira, G., Bastos, R., Comba, J.: Real-time multi-agent path planning on arbitrary surfaces. ACM SIGGRAPH symposium on Interactive 3D Graphics and Games pp. 47–54 (2010)