Smart Visible Sets for Networked Virtual Environments

FÁBIO O. MOREIRA JOÃO L. D. COMBA CARLA M. D.S. FREITAS
Instituto de Informática - Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 Caixa Postal 15064 CEP 91501-970 Porto Alegre, RS, Brasil
{moreira, comba, carla}@inf.ufrgs.br

Abstract - The real-time visualization of complex virtual environments across the network is a challenging problem in Computer Graphics. The use of pre-computed visibility associated to regions in space, such as in the Potentially Visible Sets (PVS) approach, may reduce the amount of data sent across the network. However, a PVS for a region may still be complex, and further partitions of the PVS are necessary. In this paper we introduce the concept of a Smart Visible Set (SVS), which corresponds to (1) a partition of PVS information into dynamic subsets that take into account client position, and (2) an ordering mechanism that enumerates these dynamic sets using a visual importance metric. Results comparing the SVS and the PVS approach are presented.

1. Introduction

Computer Games are one of the most powerful forces driving graphics development these days. The recent advances in graphics hardware allows the creation of rich and complex scenarios, with advanced texture effects and lighting. One of the most promising aspects of the game business is on-line gaming, where several players interact with each other in a virtual environment across the network. Unlike personal gaming, on-line gaming has the challenge of streaming game information (environment, players positions, actions, etc) to remote clients, usually referred to as the latency problem. This is a hard problem to solve, as streaming complex environments with lots of geometry and texture can be very expensive. Therefore, a trade-off on streaming speed against environment complexity needs to be made, leading most of the times to simpler environments (seen in on-line massive multiplayer games such as Everquest or Ultima Online).

If the networking speed is a limiting factor, one way to increase game complexity is to send information ordered by visual importance to the client. In other words, the complexity is not bound by the environment itself, but from the views that we obtain from it. This is a classical problem in graphics, including visibility and occlusion-culling algorithms, and multi-resolution, level-of-detail (LOD) and image-based representations of geometry. Integrating these solutions to help solve the latency problem is the focus of our investigation.

The pre-computation of visibility information and storage in Potentially Visible Sets (PVS) have been used in many applications and in games such as Quake. A PVS consists of a list of objects (polygons or other regions), representing what can be seen from a region in the environment. The PVS needs to be sent only once for each region, and while the client stays in this region, the PVS of adjacent regions (hopefully predicting the client path) can be sent to maintain full use of network bandwidth, allowing spatial coherence to be explored. If a single PVS is still more than what the network can handle, further orderings or simplification of geometric detail based on an importance metric are alternatives need to be performed.

In this paper we propose the Smart Visible Sets (SVS) approach to allow pre-computed visibility information to be adapted to the needs of a client viewing parameters. Usually, PVS information is not stored in a way that can be efficiently adapted to further processing. In the SVS, objects are first grouped by angle into directions that span the hemisphere of viewing directions. Organizing features by angle allow the ones located along the client's viewing direction to be sent before the ones behind the viewer. The angle break-up of viewing directions can be made as flexible as possible, into as many directions as necessary, and region specific (i.e., adapted to its visibility set). Additionally, each angle group is broken into subgroups with respect to its distance to the PVS region. Having the distance information pre-computed allows faster selection of LOD or image-based replacements to geometry.

Therefore, the SVS represents an indexed data structure to visibility information, classified into angle and distance subgroups. Along the representation, an ordering mechanism that uses a pre-defined importance metric (maybe user-defined), can efficiently enumerate the information in such a way that data can be streamed across the network.

The paper is organized as follows. In the next section, we review past and relevant work. The SVS approach is described in Section 3, presenting several ideas to break viewing directions into groups. Results are discussed in Section 4 while conclusions and directions of future work are presented in Section 5.

2. Previous Work

Techniques for rendering complex scenes in interactive walkthroughs especially in networked virtual environments have been reported in the literature lately [9]. Most of the research addresses acceleration of local rendering, transmission of graphical information over a network and scene simplification (refer to Teler and Lischinski [6] and Pires and Pereira [8] for discussion on these topics, which will not be specifically surveyed here).

One of the problems in the real time rendering of complex scenes is the computation of visibility information, which is a classical problem in Computer Graphics [5]. Our first approach to deal with the visibility problem is based on the computation of PVS.

Potentially Visible Sets can be computed from points or regions (or cells) in a scene. The precomputation of the PVS from a cell is more effective regarding computational costs than that for a point, and is stored readily for usage during rendering in interactive walkthroughs. Although there is an inherent space problem due to the large number of cells in complex scenes, the problem of reducing or simplifying PVS information has not been frequently addressed in the literature. A technique to compress precomputed visibility sets based on the clustering of objects and cells was presented by Van de Panne and Stewart [7] while Gotsman et al encode visibility information in a hierarchical scheme [2]. Cohen-Or et al [3,4] are concerned with the transmission of the visibility sets from server to the client. The same research group [4,1] also discusses the selection of the best cell size depending on the size of the PVS More recent results are reported by Koltun et al. [10]. Instead of storing a PVS for each cell, these authors use an intermediate representation that is used for generating the PVS itself during rendering. This intermediate representation is based on virtual occluders, which are a compact representation of the aggregate occlusion for a given cell.

3. Smart Visible Sets

Smart Visible Sets (SVS) are an alternate form of visiblity storage to PVS. To generate the SVS we break the PVS of each cell into several subsets by using an additional parameter. We have been working with (1) viewing frustum and (2) distance.

Once this structure is in place, data streaming routines can combine the SVS to allow the most relevant data to be sent first, or to allow the culling of the non crucial data.

In this work, we focused on 2.5D environments, mainly cities. In this kind of dataset, our discussion of breaking viewing directions is simplified to a planar problem, but nevertheless has interesting aspects and allows us to explore the SVS concept.

We currently use the Binary Space Partitioning Tree (BSP-Tree) and the PVS generated by the QBSP3 and QVIS applications (developed by ID Software) as a base for our work. The scene data is stored in a BSP-Tree and the PVS stores cell-to-cell visibility information. The PVS for each cell is stored as an array of bits (each ON bit meaning that the leaf is visible) and we store the SVS in the same way.

Like the PVS, the calculation of the SVS is done in a pre-processing step, causing no impact to the performance of the applications that use them.

3.1 Breaking Visibility by Angle

One way to split a PVS into different subsets is to break the hemisphere of viewing directions into groups (viewing frustums). Since we are dealing with 2.5D scenes, we use the azimuth to split the PVS, not worrying about frustum tilt or elevation. Deciding which are the best angles to split the PVS is the question to be answered.

Constant Number of Angles and Orientations

Our initial approach was to use a constant set of splitting frustums. In this approach, the PVS of all cells are broken by the same angles (the angles are chosen by the user).

To split the PVS using a set of angles we need to calculate the volumes that are represented by each cell. Then we build two line equations using the splitting frustum limits and check if the other cells' volumes lie inside or outside these lines. To speed up the calculations we use the original PVS so we only need to check frustum visibility on the cells that were already visible in the PVS.

We use a recursive function to calculate the bounding volumes of all the leaves in the scene. Starting with the bounding volume for the whole scene, we traverse the BSP-Tree left-side first. At each non-leaf node, we split the bounding volume in two (using the partitioning plane stored on the node) and then we start two new recursions using the resulting bounding volumes and the node's sons as parameters. When we reach a leaf we save its bounding volume into an array.

For each cell and each splitting frustum, we determine two line equations. Each line equation is defined based on the x and z coordinates of one point of the source leaf's bounding-box and one of the splitting frustum limits. To determine the correct line equation we must pick the correct point from the leaf's bounding box, based on the angle value and if the angle is the starting or ending limit of the frustum (Figure 1). Notice that we use the bounding-box, not the bounding-volume. Choosing the wrong point or using the bounding volume instead of the bounding-box causes incorrect visibility calculations.

Using those two line equations, we calculate the new cell-to-cell visibility, storing it in a bit array. We

also store information about the viewing frustum limits in the beginning of each SVS.

To solve the new cell-to-cell visibility problem we check if the bounding volume of the target cell lies completely outside of both frustum limits (using the two line equations) in which case that leaf will not be visible. We only need to check cell-to-cell visibility for those cells that are visible in the original PVS.

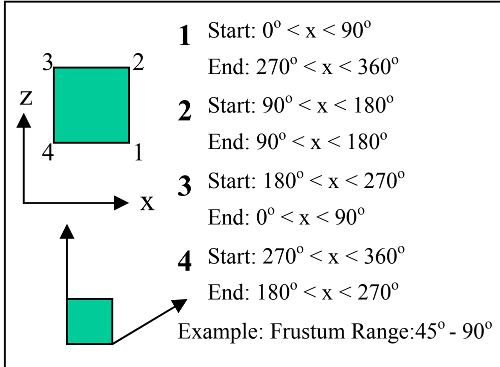


Figure 1: Determining the line equations.

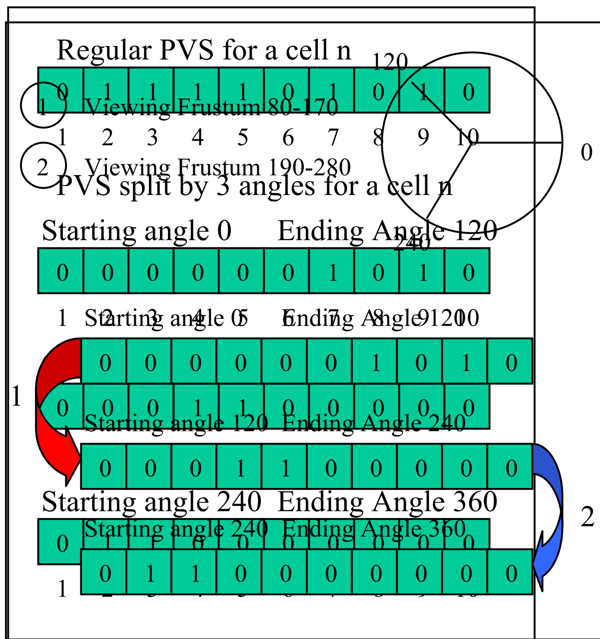
Finally we store the visibility information for each splitting frustum as a bit array (Figure 2). So each leaf will store a set of bit arrays (instead of just one, like in the PVS). The size of the visibility information for each leaf is:

Visibility Size = Number of leaves * Number of splitting angles

Figure 2: Visibility information for each splitting frustum.

To determine the visibility for a given leaf using the default PVS, we simply check which bits are ON in the PVS of that cell. If we want to determine visibility using a SVS split by angles we need to OR the bit arrays whose angles intercept the user viewing frustum. Figure 3 shows how different viewing frustums generate different visibilities in a SVS split by three angles.

Adaptive Number of Angles and Orientations



After extensive testing with the constant frustum approach, where the user chooses the splitting frustums that are used to generate all the SVS, we explored a more adaptive approach.

The main problem with the constant frustum approach is that we often have a SVS storing too many visible cells, while others store just a few. If the SVS belonging to a cell are too unbalanced, they won't be very efficient. The ideal solution would be that each of the cell's SVS stored the same number of visible cells.

In this new approach, we select different splitting frustums for each cell. The user only indicates how many candidates will be tested each time and what the maximum number of splitting frustums is. The algorithm is:

1. Starting with a 360° degree frustum, choose a set of splitting candidates (the number of candidates that are actually tested is picked by the user).
2. Split the frustum using the candidates and choose the best split. The best split is the one that generates the lowest absolute value:

Number of visible cells in the 1st frustum – Number of visible cells in the 2nd frustum value.

3. Recursively choose candidates and pick the best splits using the two halves of the frustum until the maximum number of splitting frustums has been reached.

Figure 3: SVS split by three angles.

Once the splitting frustums are chosen, the cell-to-cell visibility calculation and the storage procedures are the same as in the constant approach.

3.2 Breaking Visibility by Distance

Another way of splitting the PVS into different subsets is to use the minimum distance between the cells. By comparing the faces from the cells' bounding-volumes we are able to calculate the minimum distance between them. A faster way of calculating cell-to-cell distance is to simply calculate the distance between the cell's centers, but this is also a less precise algorithm as we can see in Figure 4, where the dotted line shows an erroneous approximation of cell distance in a top view of the scene.

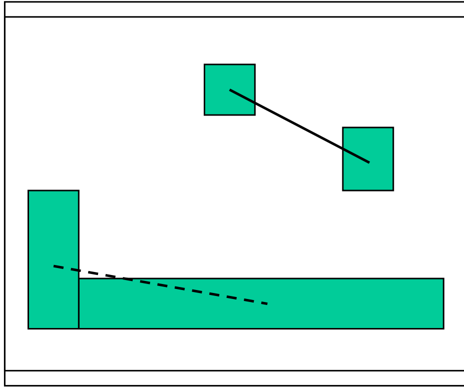


Figure 4: Distance between cell's centers**.**

By setting a few distance ranges, we are able to build a new set of SVS that adds distance information to the visibility information. To make bitwise operations easier, we also store the distance based SVS as bit arrays. Each distance SVS stores a viewing distance and which cells are visible within the distance range for that cell.

Once again we use the original PVS to perform distance calculations only on those cells that were already visible. The distance PVS are stored in an ascending order.

If we want to take into account the distance between the cells we can do an AND between the visibility result (OR of all SVS that intersect the viewing frustum) and the SVS that represents the desired distance range.

By changing the distance SVS that is being ANDed the user can quickly change the rendering quality of the scene. We are also exploring an adaptive solution to generate the best possible rendering quality of the scene while still maintaining a desirable frame rate.

Another possibility is combining distance based SVS and level of detail (LOD) strategies. Checking the distance from objects to the user requires simple calculation of which cell the object is inside. Once we have that information, the SVS can be checked to see what level of detail should be used.

3.3 Visual Importance Metrics

The main use for the SVS is the ordering of information. In a client-server environment, we can order the information that needs to be sent to the client using the SVS, thus sending more important data first. It is easy to see how data that lies inside the client's viewing frustum and closer to the client's position should have a higher priority over data that is far away and outside the viewing frustum.

Using SVS we can determine which cells are closer or/and inside the client's viewing frustum with a small number of bitwise operations and then order the information that needs to be sent across the network according to its visual importance to the client.

Without a defined visual importance metric it's hard to answer questions such as: "Should data that is closer to the client but outside its frustum be sent before data that lies inside the viewing frustum but is farther away?". We are currently working on a graph based metric that will allow quick reconfiguration and testing of the visual quality of the scene.

4. Results and Discussion

In order to test our implementations of these algorithms as well as run benchmarks and check results, we have developed a 3D application. It allows such operations as:

- Loading of scenes
- Visualization of the bounding-boxes of each cell in the scene
- Visualization of the original PVS
- Visualization of SVS (viewing frustum and distance), individually or merged with any number of other SVS
- Cell culling based on the user viewing frustum (using the viewing frustum SVS)
- Cell culling based on the distance from the user cell to other cells (using the distance SVS)
- Creation, storage and loading of walkthrough paths (that are later used for benchmarking)
- A free camera mode and an user camera mode
- Benchmarks that take into account: raw number of cells rendered in each frame and total number of cells seen during a path

The two camera modes were created to make visualization of information easier. All culling and visualizations are done based on the user camera and position. The user himself is represented as tetrahedron. Using the free camera mode, we can walk around the scene without changing the current culling or visualization parameters, making it easier to check if the algorithms are working as intended.

Finally, the possibility of easily changing the rendering parameters and creating walkthrough paths allows the comparison between the SVS and PVS methods and among SVS built with different splitting parameters. The environment was implemented using Visual C++ and OpenGL.

Our first batch of benchmarks tested the times needed to generate different SVS. Table 1 shows these times, for a scene with 600 buildings. The BSP tree for this scene has 1257 leaves. These benchmarks were obtained in an Athlon 1.5 machine with 256 MB RAM, Windows 2000. We include the PVS generation time on this table.

Table 1: PVS and SVS computation times.

Function	Time

PVS	2h 37m 40s
SVS (3 Angles)	14s
SVS (4 Angles)	16s
SVS (4 Angles, 5 Splitters)	71s
SVS (4 Angles, 10 Splitters)	137s
SVS (4 Distances)	12s

Then we did a series of tests to measure the efficiency of different splitting algorithms for the SVS. Results were obtained by running a set of different paths in an environment with 600 buildings. All paths were executed five times using different splitting options, which include: splitting in three angles, splitting in four angles and splitting in four angles using an adaptive algorithm. Path 1 starts in the center of the city and moves to one of its corners; path 2 is a straight line from one side of the city to the other; path 3 starts in the center and moves to one of the sides of the city, and finally path 4 is a square shaped path around the center of the city

During the execution of the path, the visible leaves on each frame were marked. After the path was complete, the total number of marked leaves allowed us to measure how much information would need to be sent to the client using that particular partitioning strategy. Table 2 shows the results.

We also include two pictures of the environment taken during these benchmarks. Figure 5 shows the culling done by a 3 angle SVS. The darker buildings are the ones that are visible in the PVS but have been culled by the rendering engine because they lie in SVS that don't intersect the user's viewing frustum (the user is represented by the tetrahedron located in the middle of the picture).

Figure 6 shows the splitting of a PVS by 4 distances. The large white area is the non visible area. The different shades of gray represent the partitioning of the visible area by different distances. The more distant cells were rendered with darker colors.

Table 2: Number of cells seen on different partitioning strategies.

	Path 1	Path 2	Path 3	Path 4	Totals
PVS	824 100%	918 100%	889 100%	927 100%	100%
SVS 3 Angles	81 98.4%	612 66.6%	638 71.7%	863 93.0%	82.4%
SVS 4 Angles	724 87.8%	775 84.4%	694 78.0%	880 94.9%	88.7%
SVS 4 Angles 5 Splitters	739 89.6%	827 90.0%	614 69.0%	748 80,6%	82.3%
SVS 4 Angles 10 Splitters	647 78.5%	779 84.8%	425 47.8%	747 80,5%	72.9%

5. Conclusions

Client-server applications over the network often have to deal with the problem of having too much information to send to its clients, or too many clients connected. They need fast and reliable ways to cull that information when needed.

In this paper we introduced the idea of Smart Visible Sets, and explained the algorithms used for its implementation. The SVS is a tradeoff between space (both memory and storage) and performance. It can be efficiently used to sort server data according to its visual importance to the client. Then, the more important information for rendering can be sent first, resulting in a smoother experience of the simulation by the user. It can also be used to cull a part of the information (the least important part of it) if the network traffic exceeds a limit (either the server limit or the client limit). In a server with a big number of clients or in slower network environments the use of SVS can represent a big leap in performance.

We also performed a series of tests that show the advantages of the adaptive approach over the constant approach for choosing splitting angles. We have also proved that the SVS generation does not introduce a significant overhead in the pre-computation of visibility information since it's only a small fraction of the time needed for the PVS determination.

We have not executed tests in very large environments yet because the tools we use to generate the PVS perform poorly on large open areas. We are currently studying the use of different PVS generation algorithms.

The next steps in this work are (a) developing a client-server application and comparing the visual results of the SVS and PVS solutions; (b) adding textures to the BSP Viewer Environment so it can look more realistic, and (c) studying the possibility of using 3D-frustum as splitters for the SVS.

Acknowledgments

This work has been supported by CNPq and FAPERGS.

References

[1] B. Nadler, G. Fibich, S. Lev-Yehudi, and D. Cohen-Or, A qualitative and quantitative visibility analysis in urban scenes. *Computer & Graphics,* 23(5):655-666, 1999.

[2] C. Gostman, O. Sudarsky, and J. Fayman, Optimized occlusion culling. *Computer & Graphics,* 23(5):645-654, 1999.

[3] D. Cohen-Or and E. Zadicario, Visibility streaming for networked-based walkthroughs. *Graphics Interface'98*, pp. 1-7, 1998.

[4] D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario, Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum,* 17(3):243-254, 1998.

[5] D. Cohen-Or, Y. Chrysanthou and C. Silva. A survey of visibility fo walkthrough applications. In: *SIGGRAPH 2000 Course Notes - Visibility: problems, techniques and applications*, July 2000.

[6] E. Teler and D. Lischinski, Streaming of complex scenes for remote walkthroughs. *Computer Graphics Forum,* 20(3): C18-C25, 2001.

[7] M. van de Panne and J. Stewart, Efficient compression techniques for precomputed visibility. In *Eurographics Workshop on Rendering*, 1999.

[8] P. Pires and J. Pereira, Dynamic algorithm binding for interactive walkthroughs. *Proceedings of SIBGRAPI 2001*, pp. 154-161, IEEE Computer Society Press, 2001.

[9] S. Singhal and M. Zyda. Networked Virtual Environments. Addison-Wesley, 1999.

[10]V. Koltun, Y. Chrysanthou and D. Cohen-Or, Virtual occluders: an efficient intermediate PVS representation, *Eurographics Workshop on Rendering*, pp. 59-70, Eurographics, 2000.

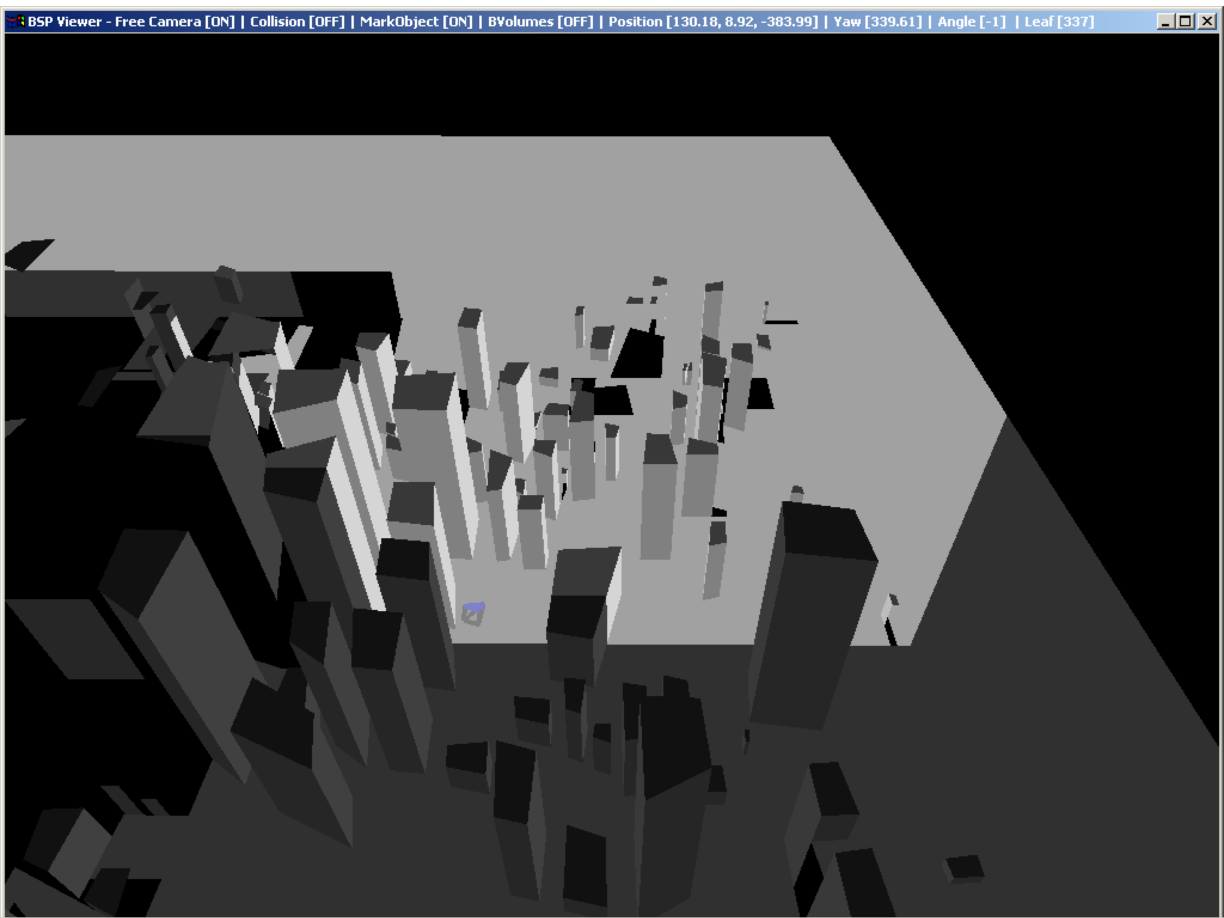


Figure 5: Scene with 600 buildings, SVS split by three angles. Visible cells are light and culled information is dark.

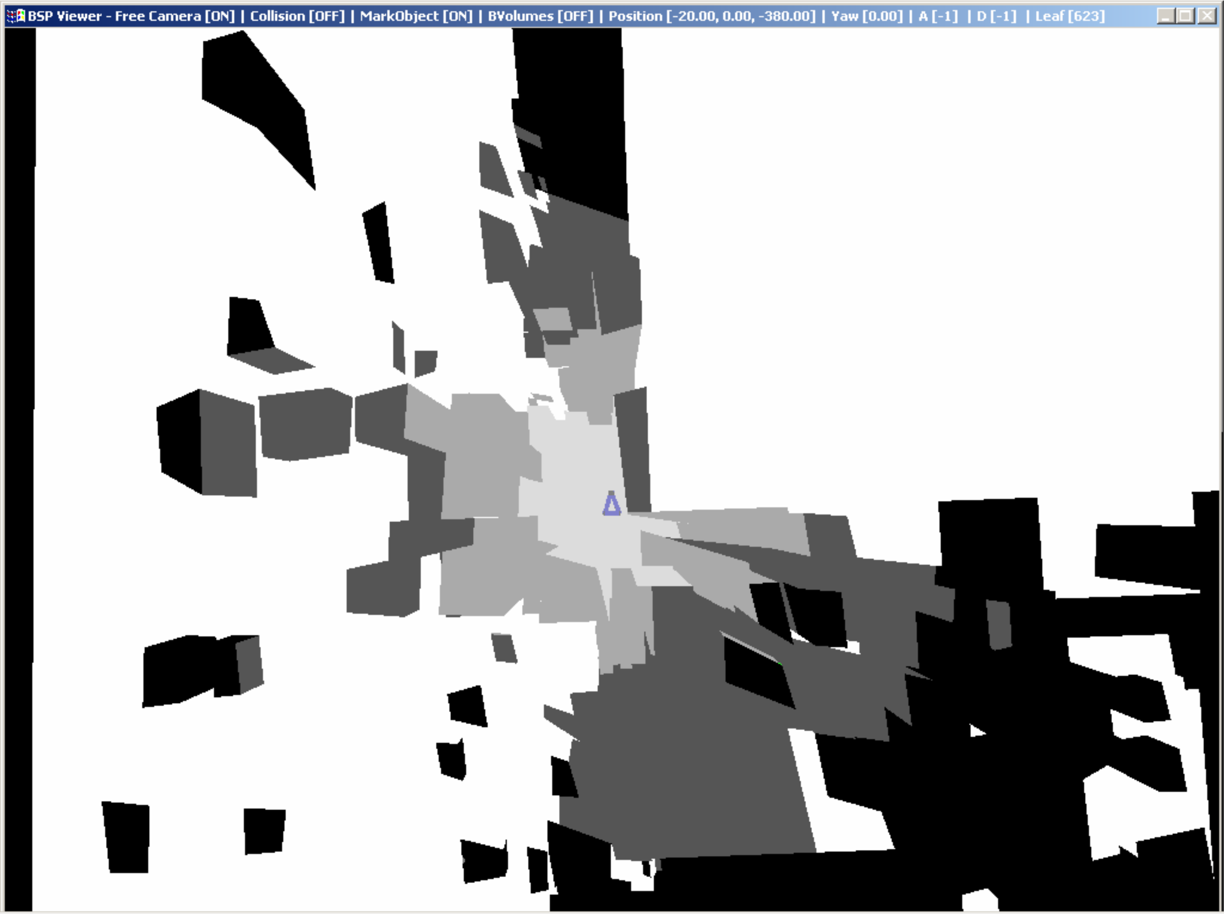


Figure 6: Scene with 600 buildings, SVS split by 4 distances. Darker regions are more distant from the user