Instant Visibility

Peter Wonka, Michael Wimmer, François X. Sillion

EUROGRAPHICS 2001 / A. Chalmers and T.-M. Rhyne
(Guest Editors)

Volume 20 (2001), Number 3

Instant Visibility

Peter Wonka and Michael Wimmer and François X. Sillion

iMAGIS - GRAVIR/IMAG-INRIA and Vienna University of Technology

Abstract

We present an online occlusion culling system which computes visibility in parallel to the rendering pipeline. We show how to use point visibility algorithms to quickly calculate a tight potentially visible set (PVS) which is valid for several frames, by shrinking the occluders used in visibility calculations by an adequate amount. These visibility calculations can be performed on a visibility server, possibly a distinct computer communicating with the display host over a local network. The resulting system essentially combines the advantages of online visibility processing and region-based visibility calculations, allowing asynchronous processing of visibility and display operations. We analyze two different types of hardware-based point visibility algorithms and address the problem of bounded calculation time which is the basis for true real-time behavior. Our results show reliable, sustained 60 Hz performance in a walkthrough with an urban environment of nearly 2 million polygons, and a terrain flyover.

1. Introduction

We are interested in real-time rendering applications such as urban simulation (Fig. 1), flyovers and interactive scene modeling. The scene complexity for these applications usually exceeds the rendering capacity of a single computer. To reduce the load on the graphics hardware, visibility calculations can be used to quickly prune large portions of the scene.

However, the visibility problem is inherently complex[7]. Point-based visibility algorithms, for example, calculate a set of potentially visible objects (PVS) for each viewpoint. The general problem of point visibility algorithms is that they have to be executed for each frame and the renderer cannot proceed until a PVS is available. Their relatively long computation time significantly reduces the time available to render geometry, if not reducing the achievable frame rates below limits acceptable for real-time rendering applications.

Figure 1: *A frame from an urban walkthrough. For true real-time behaviour we aim at refresh rates of 60 Hz.*

Precalculating visibility for a region of space[8, 13, 19, 22] (view cell) reduces almost all runtime overhead. However, current region visibility algorithms suffer from several drawbacks. There is a tradeoff between the quality of the PVS estimation on the one hand and memory consumption and precalculation time on the other hand. Smaller view cells reduce the number of potentially visible objects and therefore improve rendering time. However, smaller view cells also increase the number of view cells that need to be precomputed, which can result in prohibitively large storage requirements and precalculation times for all PVSs. Another problem is that many runtime modifications of the scene cannot be handled, and even small offline changes to the model might entail several hours of recomputation. This makes region visibility a viable choice for certain models (as, for example, in a computer game), but impractical for dynamic systems where changes to the model occur frequently (as, for example, in an urban modeling scenario).

In this paper we address the aforementioned problems. We

show how to achieve a large improvement over previous systems by adding new hardware to the system in the form of an additional machine in the network which is used as a visibility server. We calculate visibility at *runtime*, avoiding memory problems because the PVS need not be stored, but for a *region*, allowing it to be calculated in parallel to the rendering pipeline so that it imposes virtually no overhead on the rendering system. This results in Instant Visibility, a system which calculates a tight PVS with very little preprocessing and practically no runtime overhead on the graphics workstation.

The remainder of the paper is organized as follows: after reviewing some relevant work, section 3 gives an overview of the system, and section 4 gives a detailed description of the various algorithms involved. We discuss integration of existing point visibility algorithms into our system in section 5, and present results in section 6. Section 7 gives a detailed discussion of some important aspects of the system.

2. Previous Work

The idea of efficient visibility culling algorithms for real-time rendering is to calculate a conservative estimation of those parts of the scene that are definitely invisible, leaving final hidden surface removal to a z-buffer. The set of objects remaining after visibility culling is usually called the potentially visible set (PVS).

One class of visibility algorithms calculates visibility from a *point*. A simple and general example is view frustum culling[4], which is applicable to almost any model. To take into account the large amount of occlusion present in many scenes, image-space and geometric occlusion culling methods were proposed. Image space methods render occluders into an occlusion map or a hierarchy of maps. Scene objects, organized in a spatial data structure, are tested against the map(s). Image-based methods have become popular in practice because of their robustness and ability to make use of graphics hardware. Occlusion maps are calculated using either perspective projection (e.g., the hierarchical z-buffer, or HZB[10], and the hierarchical occlusion map, or HOM[23]), or orthographic projection (e.g., the cull map[21]). In geometric methods[6, 3, 12], scene objects are geometrically tested against the shadow volume of selected occluders. The number of occluders that can be considered is limited due to the complexity of geometric shadow computations. Unfortunately, current point occlusion culling algorithms take up a substantial amount of frame time, limiting their use for real-time rendering applications.

Multiprocessing has been explored as a means to speed up visibility culling. In their interactive massive model rendering system, Aliaga et al.[1] combine image-space occlusion culling and several other acceleration techniques into a working system. Occlusion culling is calculated on a separate processor one frame in advance, which introduces a latency of one frame and tightly couples occlusion culling to the frame rate.

A second class of visibility algorithms breaks down the view space into *regions* of space (typically called view cells) and precomputes a PVS for each view cell. After the whole scene has been processed, PVS data for all view cells is stored on disk and retrieved on demand during an interactive walkthrough. Some algorithms require objects to be occluded by a single occluder to be considered invisible[5, 18]. Other algorithms exploit a priori knowledge about the scene structure: indoor scenes (e.g., architectural walkthroughs)[16, 20]) can be partitioned into portals and cells to compute inter-cell visibility, and outdoor scenes can be handled by a 2.5D approach[13, 22]. Algorithms for general 3D scenes have been proposed by Schaufler et al.[19] and Durand et al.[8].

3. Overview

We introduce a new visibility system that allows calculating visibility in parallel to the traditional rendering pipeline. The idea is to calculate a visibility solution (PVS) which is valid for several frames, depending on a maximum observer movement speed. This is achieved by using a standard point visibility algorithm and shrinking the occluders so as to make the resulting visibility solution valid for a certain ϵ-neighborhood around the viewpoint from which the visibility solution is calculated.

The algorithm consists of a short preprocessing phase and an online phase. The following parameters have to be determined in advance:

- Choose a point visibility algorithm.
- Decide on how much time to allot for the visibility solution.
- Set a maximum observer movement speed.

In the preprocessing phase, occluders are generated for the scene and shrunk by an amount determined through the maximum movement speed and the time allotted for the visibility solution. For the online phase, two computing resources are needed:

- one resource to render and display frames with the current PVS
- a second resource to calculate the PVS for the next set of frames

4. Instant Visibility

4.1. The traditional pipeline

The traditional rendering pipeline consists of several steps, where each step depends on the outcome of the previous step. Roughly, we identify two important steps of the pipeline for our discussion:

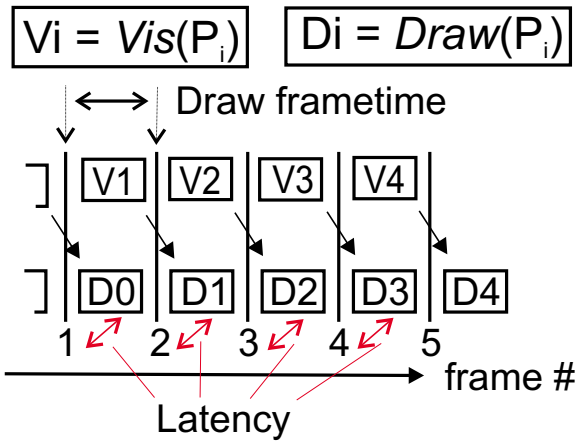

Figure 2: *Parallelization in the traditional pipeline introduces latency and limits the time available for visibility. The figure shows where additional latency is introduced.*

- $Vis(P)$ determines which objects are visible from an observer position P.
- $Draw(P)$ draws (traverses, transforms, rasterizes) the objects identified as visible as seen from the observer at position P.

These two steps communicate via the

- $PVS(P)$, potentially visible set, i.e., the set of objects determined to be potentially visible for a position P.

Efforts to parallelize $Vis(P)$ and $Draw(P)$ (for example, in a manner consistent with the multiprocessing mode of a popular rendering toolkit[11]) suffer from latency, since $Draw(P)$ requires the result of $Vis(P)$ to be able to operate (see Fig. 2).

4.2. Extending PVS validity

Wonka et al. [22] showed that the result of any point visibility algorithm can be made valid for an ϵ-neighborhood around the viewpoint by occluder shrinking. If all occluders in the scene are shrunk by an amount of ϵ, the pipeline step $Vis(P)$ actually computes

$$PVS_\epsilon(P)$$

the set of objects potentially visible from either the point P or any point Q with $\|P-Q\| < \epsilon$. $PVS_\epsilon(P)$ can also be characterized by

$$PVS(P) \subseteq PVS_\epsilon(P)\ \forall Q : \|P-Q\| < \epsilon$$

We observe that the result of $Vis(P_0)$ is still valid during the computation of $Vis(P_1)$, as long as the observer does not leave an ϵ-neighborhood around P_0.

4.3. Parallel execution

We exploit the above observation to remove Vis from the pipeline and instead execute it in parallel to the pipeline (Fig. 3). $Vis(P)$ might even take longer than a typical frame to execute - as long as the observer doesn't move too far away from P. More precisely, it is easy to show the following

Lemma 4.1 Assume a frametime of t_{frame}. Assume also that $Vis(P)$ takes at most a time of t_{vis} to compute, where t_{vis} is a multiple of t_{frame}, and $Vis(P)$ always starts at a frame boundary. Then the time t_ϵ for which the visibility solution $PVS_\epsilon(P)$ computed by $Vis(P)$ has to be valid so as to allow parallel execution of Vis and $Draw$ can be calculated as

$$t_\epsilon = 2t_{vis} - t_{frame}$$

Proof $Vis(P_i)$ takes t_{vis} to calculate. The result has to be valid till the result from $Vis(P_{i+1})$ is available, which takes again t_{vis}. But, the last frame where $PVS_\epsilon(P_i)$ is valid displays an image for a viewpoint at the start of the frame. During the time period needed to render this last frame, no visibility solution is actually needed. So, we have $t_\epsilon = t_{vis} + t_{vis} - t_{frame}$. □

Given a maximum observer speed v_{max}, the amount ϵ by which to shrink occluders can now be readily calculated as

$$\epsilon = t_\epsilon v_{max}$$

If the visibility solution does not take longer to compute than a typical frame (i.e., $t_{vis} = t_{frame}$), this means that $t_\epsilon = t_{frame}$ and ϵ identifies the amount of space the observer can cover in one frame.

The algorithm described here effectively allows near-asynchronous execution of Vis and $Draw$. This makes it possible to achieve high frame rates even if the used visibility algorithm is not fast enough to complete in one frame time. In a typical scenario, the point visibility algorithm can provide results at a rate of at least 20 Hz, and the screen update rate is 60 Hz. Then PVS_ϵ has to be valid for the distance ϵ the observer can go in 5 frames. Assuming a maximum observer speed of 130 km/h, ϵ would be 3 m.

4.4. Networking

Executing visibility in parallel to rendering requires an additional computing resource. If the point visibility algorithm does not need access to graphics hardware itself, it can run on a separate processor. In case the point visibility algorithm does need acces to graphics hardware, multichannel architectures allow the system to run on one machine.

The real strength of the method, however, lies in its inherent networking ability. The low network latency of today's local area networks allows a second machine to be used as a visibility server. At the start of a visibility frame, the current

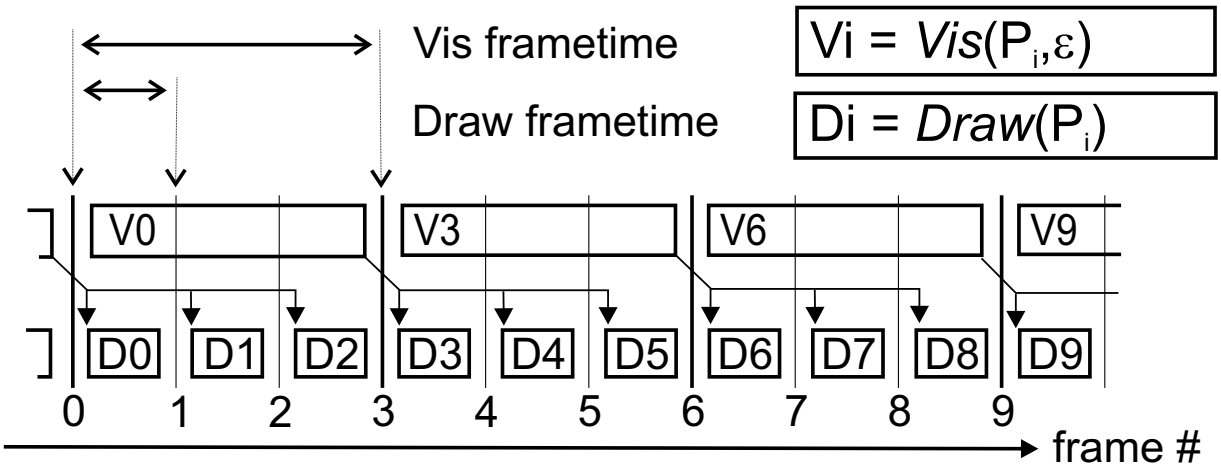

Figure 3: *The new visibility pipeline. Vis can take several frame times, the arrows show for which frames the resulting PVS is used.*

viewpoint is passed to the visibility server. After visibility calculations, the PVS is transmitted back to the graphics workstation.

A PVS typically contains object identifiers of a spatial scene data structure, so the size of a PVS depends on the granularity of this data structure and the amount of occlusion present. It should be noted that PVS-data is well compressible, even standard entropy coding achieves a compression ratio of up to 3:1. To give a practical example, passing a PVS containing 4000 32bit object identifiers on a 100MBit-network takes about 1ms after compression.

Running the point visibility algorithm on a second machine also has the advantage that access to graphics hardware is automatically available, provided the visibility server is equipped with a good graphics card.

4.5. Synchronization

Running the visibility step in parallel to the rendering step requires synchronizing the two. In particular, it is crucial to deal with situations where visibility takes longer than t_{vis} to calculate, because the rendering step cannot continue without a potentially visible set. We list several ways to cope with this problem and discuss their applicability.

The preferred strategy depends strongly on the point visibility algorithm chosen. Many such algorithms consist of two steps: creating an occlusion data structure using a set of occluders, and testing the scene against this occlusion data structure. We assume that the number of occluders to draw determines the running time of visibility, and that the time necessary to test the scene against the occlusion data structure can be determined in advance.

guaranteed visbility Use a point visibility algorithm that has an inherent bound on its running time.

abort visibility Draw only so many occluders that visibility can execute in t_{vis}.

predictive occluder scheduling Determine in advance which occluders to draw so that visibility can execute in t_{vis} and best possible occlusion is achieved. If occluder levels of detail are available, they can be incorporated in a similar fashion to Funkhouser's predictive level of detail scheduling [9].

The next two possibilities are intended as fallback-strategies rather than solutions of their own, in case the other strategies fail, or small errors are not an issue and the chosen visibility algorithm executes fast enough in the majority of cases. They are implemented in $Draw$ instead of Vis.

stall Stall movement to prevent the observer from leaving the ϵ-neighborhood as long as there is no visibility solution available. This always guarantees correct images.

incomplete visibility Let the observer leave the ϵ-neighborhood, but still use $PVS_\epsilon(P)$. Errors in visibility might occur, but observer speed is continuous and unhampered.

5. Integration with current occlusion culling algorithms

In this section we discuss important issues when integrating a point occlusion algorithm into our system.

5.1. Choice of projection

Image-based occlusion culling algorithms rely on projecting occluders and scene objects to a common projection plane.

For the application in urban environments, Wonka et al. [21]

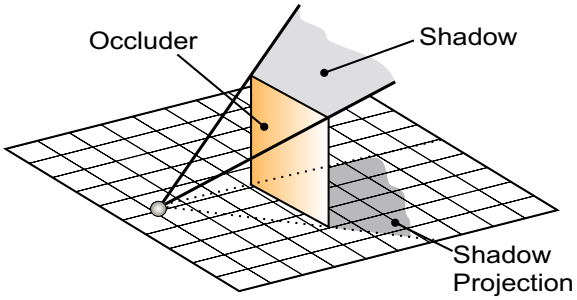

Figure 4: *An orthographic occlusion map. The shadow cast by an occluder is projected orthographically into the cullmap, which stores the depth values of the highest shadow.*

proposed rendering the shadows cast by occluders into an *orthographic* top-down view of the scene, the cull map (Fig. 4). Although the approach is limited to 2.5D scenes, it has two advantages:

- shrinking the occluders also guarantees conservative rasterization of occluder shadows into the cullmap [22].
- It is very easy to calculate occlusion for a whole 360° panorama.

Other common approaches like the hierarchical z-buffer or the hierarchical occlusion maps are based on a *perspective* projection of the occluders and the scene into the current viewing frustum (Fig. 5). The advantage is that arbitrary 3D scenes can be processed. However, visibility is only calculated for the current viewing frustum. Therefore, the viewing frustum (a pyramid) for occlusion culling has to be carefully chosen so as to include all parts of the scene that could be viewed within a time of t_ϵ.

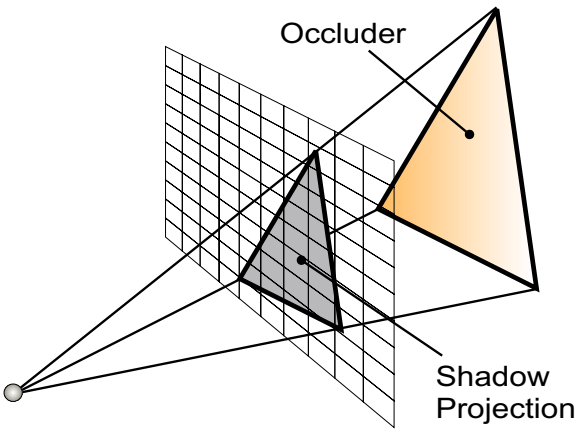

Figure 5: *A perspective occlusion map. Occluders are projected onto a plane defined by the current camera parameters.*

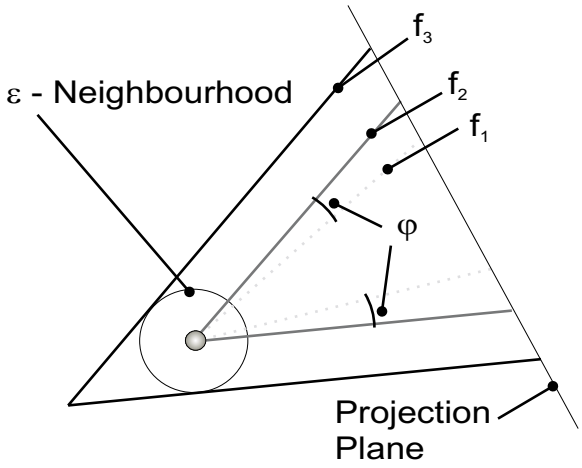

Figure 6: *Adjusting the frustum for perspective projections. The original frustum f_1 is made wider by an angle of φ. The resulting frustum f_2 is then moved back to yield a frustum f_3 tight on the ϵ-neighborhood, to accomodate rotation and movement during the time t_ϵ.*

To make this at all possible, a maximum turning speed ω_ϵ has to be imposed. The viewing pyramid is then made wider at all sides by the angle φ the observer can turn in t_ϵ, given by $\varphi = t_\epsilon \omega_\epsilon$. Finally, the pyramid is moved backwards until it fits tightly on an ϵ-sphere centered on the viewpoint, to account for backward and sideward movement (Fig. 6).

As a simple example, assume visibility and screen update rates of 20 Hz and 60 Hz respectively as above. If $\omega_\epsilon = 180\frac{^\circ}{s}$ (e.g., relatively fast head movement), the pyramid would have to be made wider by 15° on each side. This would enlarge a typical viewing frustum of 60° to 90°.

5.2. Occluder selection policies

The running speed of most current occlusion culling algorithms is mainly determined by the number of occluders they consider. It is thus desirable not to render all occluders for every view point. Typically, a heuristic based on the projected area is used to select a number of relevant occluders. Note that since only visible occluders contribute to occlusion, finding all relevant occluders boils down to solving the visibility problem itself - it is therefore not possible to find exactly all relevant occluders.

We present and discuss 3 different approaches to occluder selection. In the discussion, an occlusion data structure is a perspective or orthographic projection of occluders.

conservative Always render all occluders. Although slow, this approach has the advantage that the time required for calculating visibility can be estimated or even bounded, which is important when determining the number of frames to allot for visibility calculation.

temporal coherence Use the set of occluders visible in the previous visibility step. Since new important occluders can appear even when moving only a small distance, occlusion will not be perfect. This approach is useful in scenarios where visibility takes quite long to compute, but the rendering step is not saturated - it essentially moves load from Vis to $Draw$.

2-pass In a first pass, create an occlusion data structure with the occluders visible in the previous step as above. But instead of the scene, test the occluders against this data structure for visibility. Occluders found visible are used to create a new occlusion data structure in a second pass. The scene is then tested against this new occlusion data structure for visibility.
Like the conservative approach, this approach computes the correct PVS, and like the temporally coherent approach, it reduces the number of occluders to be used, at the expense of having to build an occlusion data structure twice. It is best used when rendering all occluders is expensive compared to testing objects against the occlusion data structure.

5.3. Occluder Shrinking

Theoretically, Instant Visibility works with any point visibility algorithm. In practice, there are restrictions on the type of occluders that can be used. In order to shrink occluders in 3D, they must be of volumetric nature. Furthermore, typical occluding objects should be large enough to ensure that shrunk occluders provide sufficient occlusion.

We propose several methods to actually shrink the occluders in a scene, applicable to different types of input scenes.

In urban environments, occluders will mainly consist of buildings. A simple 2D algorithm can be used to shrink the footprint of a building: triangulate the exterior of the footprint polygon, enlarge each triangle by ϵ, and intersect the footprint polygon with the union of all enlarged triangles. Code for union and intersection operations on polygons is freely available on the internet.

If volumetric objects are given as polyhedra, the basic 2D algorithm can easily be extended to 3D: triangulation is substituted by tetrahedalization, and polygonal union and intersection are replaced by their polyhedral counterparts. Polyhedral set operations have been explored for CSG by applying the operators directly[14] or via BSP-subdivision[2].

Purely geometrical approaches are sometimes prone to numerical robustness and efficiency issues. In an alternative approach based on discretization, Schaufler et al.[19] have shown how to represent the opaque portion of a general scene via an octree, provided the model is *water-tight*. The advantage of using an octree to represent occluders is that the set operations required for shrinking are trivial to implement. After shrinking the octree, it is converted to polyhedral form. To speed up occluder rendering, it is advisable to simplify the occluder mesh using a conservative level of detail algorithm[15].

5.4. Advanced Occluder Shrinking

Up to this point we have used a quite loose definition of occluder shrinking based on ϵ-neighborhoods. It stands to reason, however, that the observer is usually limited more in some directions than in others, and therefore an isotropic ϵ-neighborhood is not flexible enough. In most walkthrough applications, movement in the up/down-direction will occur less frequently and with less speed than movement in the plane.

We therefore introduce a more formal definition of occluder shrinking based on Minkovsky-operators which allows handling anisotropic neighborhoods. The Minkovsky-subtraction of two sets A and B is defined as

$$A \ominus B := \bigcap_{b \in B} \{a + b | a \in A\}$$

The *erosion* of a set A by a set B is defined as

$$E(A, B) := A \ominus (-B)$$

Intuitively, erosion can be interpreted as flipping the shape B around its origin and using this to carve out A, with the origin of B following the surface of A. We make use of erosion for occlusion via the following

Lemma 5.1 Let an occluder O be an arbitrary subset of $\mathbf{R}^3$, and let V be the set of possible movement vectors from the current viewpoint vp (a not necessarily bounded subset of the vector space $\mathbf{R}^3$). Define a shrunk occluder O' via $O' := E(O, V)$. Then any point p occluded by O' seen from vp is also occluded by O when seen from any viewpoint $vp' \in VP$ (where $VP := vp + v, v \in V$).

Proof Interpret occlusion as a shadow problem. Then the space of possible viewpoints VP can be interpreted as a volumetric light source. The question whether p is occluded as seen from all points of VP translates to the question whether p is in the umbra region of the light source VP. Erosion has previously[17] been shown to provide an answer to this problem (the formulation in the previous paper is based on Minkovsky addition, but can easily be shown to be equivalent to erosion). □

The practical implication of this formulation is that occluder shrinking can be adapted to anisotropic observer movement. If, for example, movement is restricted to the ground plane, then objects only have to be shrunk in the two dimensions of the plane. An important optimization results for 2.5D environments: the region of possible observer movements can be approximated by a cylinder with radius ϵ and slightly elevated top. The elevation of the cylinder top is

defined by how far the observer can move up and down in the time t_ϵ.

Implementation of advanced occluder shrinking can proceed exactly as in section 5.3. The new formulation immediately validates the 2D algorithm shown above for 2.5D urban environments, if occlusion is always calculated from the highest possible observer point. For the more general 3D algorithm, the only change is that exterior cells should be expanded by the vectors present in $-V$ instead of a constant ϵ in all directions.

6. Implementation and Results

We implemented an Instant Visibility system and show its applicability on two different test scenes.

The first scene (Fig. 7) demonstrates a walkthrough of an urban environment (2 km x 2 km) consisting of about 1.9 million polygons. An orthographic projection was used for the point visibility algorithm. 1483 occluders (Fig. 9) were generated from the building footprints and shrunk with the 2D algorithm described above. Shrinking the occluder takes not more than 4 seconds.

The second scene (Fig. 8) shows a flyover of a mountain range (4 km x 4 km) populated with trees. The size of the complete database is 872.000 polygons. Orthographic projection was used for point visibility, and 4470 occluders (Fig. 10) were generated from the terrain grid.

We recorded a path through each environment and compared the frame times for rendering without occlusion, rendering using region visibility, rendering using Instant Visibility and the pure visibility calculation times (Tab. 1). For Instant Visibility, we set the rendering frame time to 16 ms and the visibility frame time to 32 ms (= two rendering frames) for both experiments. Head rotation was not restricted, so we calculated occlusion for a full 360° panorama. Viewer movement was restricted to 108 km/h for the first scene (which gives $\epsilon = 1.5$ m) and 720 km/h for the second scene (which gives $\epsilon = 10$ m).

We want to point out the following observations:

- Instant Visibility is slightly better than region visibility.
- The frame rate is always below 16 ms for Instant Visibility. To give a real guarantee for the framerate, LOD switching[9] would have to be implemented. For our results, we only used distance based LODs for the trees in the terrain flyover.
- The times for visibility are well below 33 ms. Although the standard deviation is very small, a real guarantee would require a very careful analysis of the code. The variation is mainly due to the code that tests the scene against the occlusion map.

We additionally measured the latency and throughput of our 100 MB/s network by sending PVS data from the server to the client. We measured 0.741 ms on average for the latency (min: 0.605 / max: 0.898) and 1.186 ms for a PVS of 8 KB (min: 0.946 / max: 1.694). For our examples we did not need more than 4 KB of PVS data.

urban walkthrough method	avg	min	max	std dev.
VFC	207.3	57.6	733.4	131.9
region visibility	8.7	3.1	15.7	2.8
Instant Visibility	7.4	2.7	12.9	2.1
visibility time	19.0	17.6	20.9	0.4
terrain flyover method	avg	min	max	std dev.
VFC	10.8	2.6	23.0	5.2
region visibility	6.2	1.0	14.2	3.3
Instant Visibility	5.5	1.0	13.6	3.3
visibility time	19.1	23.5	17.6	0.9

Table 1: *The table shows the measured times in milliseconds for the two test scenes. We measured rendering times for view frustum culling (VFC), region visibility and Instant Visibility. The last row shows the time required for the visibility calculations for Instant Visibility. Note that the rendering times for Instant Visibility always stay below 16 ms. This is necessary for a 60 Hz simulation.*

Figure 7: *This figure shows an orthophoto of the urban environment. The area is located near the center of Vienna and is about 2 km x 2 km wide.*

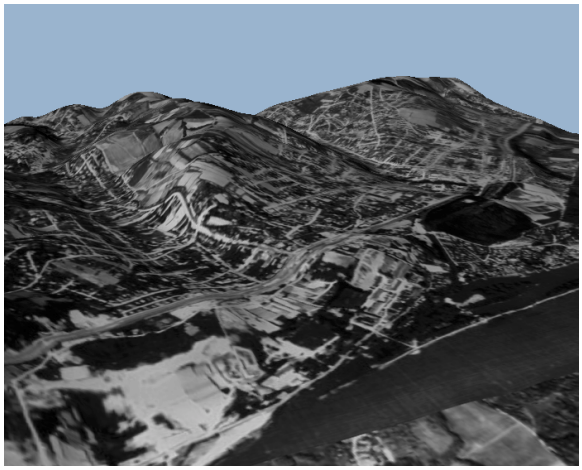

Figure 8: *This figure shows an overview of the terrain. The model covers 4 km x 4 km of the city of Klosterneuburg, a smaller city in the north of Vienna.*

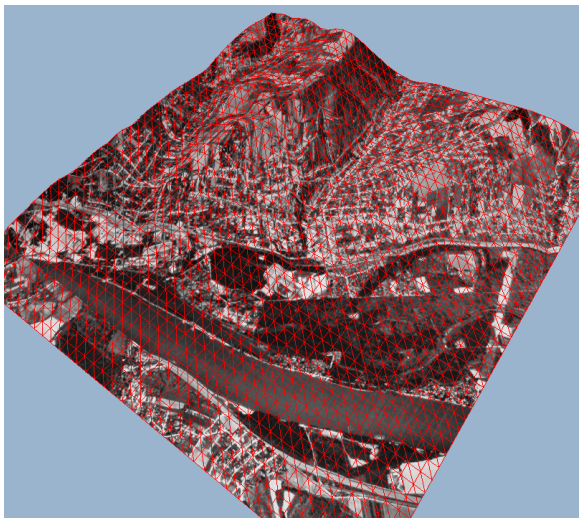

Figure 10: *The occluders used for the terrain flyover are shown in red.*

Figure 9: *The figure shows the building fronts that are used as occluders in the urban walkthrough. The parks are shown as green textured polygons and are not used as occluders.*

Figure 11: *The figure shows a frame of the urban walkthrough. Note that the geometric complexity is high, as all windows are modeled as geometry.*

7. Discussion

We have shown that the Instant Visibility system is able to handle different types of scenes while maintaining high frame rates. An important aspect of a walkthrough system is to make reasonable assumptions about observer movement. Although it might be true that walking speed is limited to several km/h in the real world, it is doubtful that imposing such speed limits would benefit the behaviour of a typical user of a walkthrough system. Especially mouse navigation allows the user to change location quickly and rapidly explore different sections of the environment. We have observed peak speeds of about 300 km/h in the urban scene, and up to 3000 km/h in the terrain scene. The actual limits to be chosen depend strongly on the type of application (100 km/h might be a reasonable limit for an urban car simulation, for example) and user behavior.

Another point to note is that the Instant Visibility system solves the visibility problem by providing a PVS for each frame, but this PVS might still be too complex to be rendered interactively on the graphics workstation. Level-of-detail approaches and image-based rendering methods should be used to further reduce the rendering load. Funkhouser's predictive level-of-detail scheduling provides a way to maintain the desired frame rate during a walkthrough (in this method, levels of detail are selected before a frame is rendered so as to maximize the visual quality while bounding the rendering cost). Terrain rendering should benefit from a multiresolution algorithm.

We would also like to briefly skirt the problem of hard real-time systems. Although the Instant Visibility system tries to maintain a given frame rate, it is not a hard real-time system. It is in general hard to give accurate running times

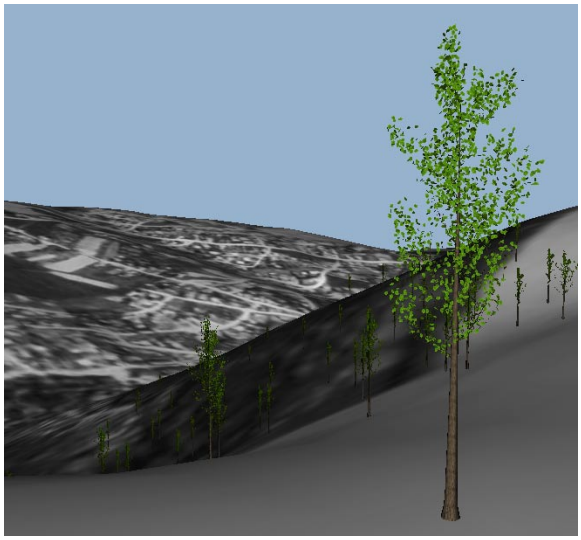

Figure 12: *A typical frame of the terrain flyover. Note that a large portion of the scene is visible.*

and upper bounds for any algorithm, and especially so for rendering and visibility algorithms which depend on graphics hardware and drivers. Therefore, the system can occasionally miss a frame, which we deem reasonable in view of the high costs involved in assuring hard real-time behavior.

The Instant Visibility system works very well if overall smooth system behavior and high frame rates are desired. The advantage over region visibility is that the time required for preprocessing is negligible, so that it is even possible to modify the scene during runtime. If occluders are shrunk separately, rigid transformations do not require any recalculation at all, in all other cases, calculations remain local to the area changed. This was one of the motivations that lead to the creation of the Instant Visibility system - we found that we rarely ever used region visibility because the needed PVS dataset was never available, and if it was, the scene had already changed, making the PVS unusable.

Another advantage over region visibility is that the view cells defined by an ϵ-neighborhood are usually smaller than typical view cells for region visibility, providing for better occlusion.

However, if its significant storage overhead and precalculation time are acceptable, region visibility offers the advantage that difficult visibility situations can be treated with special care. If the PVS is large, the visibility solution can be refined, and alternative representations for objects can be precalculated on a per-view cell basis. This is advantageous for shipping systems where the scene is not going to change, and which should not require more resources than a single machine.

Finally, we discuss the issue of latency and synchronization. The advantage of Instant Visibility over traditional pipeline systems is near-asynchronous execution of visibility and rendering, which is tightly coupled with a reduction in latency. In a traditional pipeline architecture[1, 11], visibility and rendering have to be synchronized, so the rendering frame rate is tied to the time required for visibility (Fig. 2). The latency from user input to the display of an image on the screen is therefore at least twice the time required for visibility (an image is always displayed at the end of a frame interval). In Instant Visibility, on the other hand, the time required for visibility only influences the maximum observer speed. The graphics hardware can render at the highest possible frame rate, and the latency is always one frame interval. At the same time, visibility can be calculated more accurately, because there is more time available for visibility and thus more occluders can be considered.

8. Conclusions

We have introduced Instant Visibility, a system to calculate visibility in real-time rendering applications. Instant Visibility combines the advantages of point and region visibility, in that it is based on a simple online visibility algorithm, while producing a PVS that remains valid for a sufficiently large region of space. We have demonstrated the behaviour of the system in two real-time simulation applications with sustained refresh rates of 60 Hz. We would strongly recommend the use of Instant Visibility whenever a second computer is available as visibility server or in a multiprocessor system like the Onyx2.

Acknowledgements

This research was supported by the Austrian Science Fund (FWF) contract no. P13867-INF and by the EU Training and Mobility of Researchers network (TMR FMRX-CT96-0036) ”Platform for Animation and Virtual Reality”.

References

1. Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Keny Hoff, Tom Hudson, Wolfgang Stürzlinger, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manoclia. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *Proceedings of the Conference on the 1999 Symposium on interactive 3D Graphics*, pages 199–206, 1999. 2, 9

2. J. Amanatides, B. Naylor, and W. Thibault. Merging BSP tress yields polyhedral set operations. In *Computer Graphics (SIGGRAPH 90 Proceedings)*, pages 115–124, 1990. 6

3. J. Bittner, V. Havran, and P. Slavík. Hierarchical visibility culling with occlusion trees. In *Proceedings of the Conference on Computer Graphics International 1998 (CGI-98)*, pages 207–219, 1998. 2

4. James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976. 2

5. Daniel Cohen-Or, Gadi Fibich, Dan Halperin, and Eyal Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–253, 1998. 2

6. Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *1997 Symposium on Interactive 3D Graphics*, pages 83–90, 1997. 2

7. Fredo Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Universite Joseph Fourier, Grenoble, France, 1999. 1

8. Fredo Durand, George Drettakis, Joelle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. In *Computer Graphics (SIGGRAPH 2000 Proceedings)*, pages 239–248, 2000. 1, 2

9. Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics (SIGGRAPH 93 Proceedings)*, pages 247–254, 1993. 4, 7

10. Ned Greene and M. Kass. Hierarchical Z-buffer visibility. In *Computer Graphics (SIGGRAPH 93 Proceedings)*, pages 231–240, 1993. 2

11. J. Hartman and P. Creek. IRIS performer programming guide. *SGI Document 007-1680-020*, 1994. 3, 9

12. T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proc. 13th ACM Symp. Computational Geometry*, pages 1–10, 1997. 2

13. Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. Virtual occluders: An efficient intermediate pvs representation. In *Rendering Techniques 2000*, pages 59–70, 2000. 1, 2

14. D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes. Constructive solid geometry for polyhedral objects. In *Computer Graphics (SIGGRAPH 86 Proceedings)*, pages 161–170, 1986. 6

15. Fei-Ah Law and Tiow-Seng Tan. Preprocessing occlusion for real-time selective refinement (color plate S. 221). In *Proceedings of the Conference on the 1999 Symposium on interactive 3D Graphics*, pages 47–54, 1999. 6

16. David P. Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proc. Symp. Interactive 3-D Graphics*, 1995. 2

17. A. Lukaszewski and A. Formella. Fast penumbra calculation in ray tracing. In *Proceedings of WSCG'98, the 6th International Conference in Central Europe on Computer Graphics and Visualization '98*, pages 238–245, 1998. 6

18. C. Saona-Vázquez, I. Navazo, and P. Brunet. The visibility octree: a data structure for 3D navigation. *Computers and Graphics*, 23(5):635–643, 1999. 2

19. Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François Sillion. Conservative volumetric visibility with occluder fusion. In *Computer Graphics (SIGGRAPH 2000 Proceedings)*, pages 229–238, 2000. 1, 2, 6

20. Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (SIGGRAPH 91 Proceedings)*, pages 61–69, 1991. 2

21. Peter Wonka and Dieter Schmalsteig. Occluder shadows for fast walkthroughs of urban environments. *Computer Graphics Forum (Eurographics '99)*, 18(3):51–60, 1999. 2, 4

22. Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques 2000*, pages 71–82, 2000. 1, 2, 3, 5

23. Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Computer Graphics (SIGGRAPH 97 Proceedings)*, pages 77–88, 1997. 2

Figure 13: *A frame of the urban walkthrough*

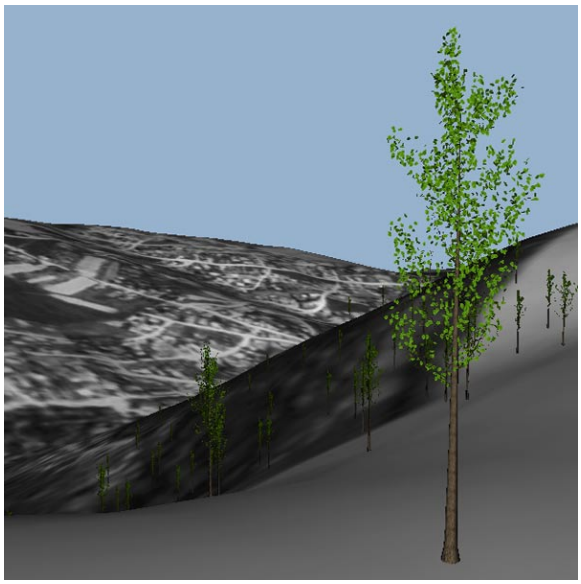

Figure 14: *A frame of the terrain flyover*