2013 13th International Conference on Computer-Aided Design and Computer Graphics

Time and Space Coherent Occlusion Culling for Tileable Extended 3D Worlds

Dorian Gomez[1,2] Mathias Paulin[1] David Vanderhaeghe[1] Pierre Poulin[2]
[1] *IRIT – Université Toulouse III, France*
[2] *LIGUM, Dept. I.R.O. – Université de Montréal, Canada*
{*paulin, vdh*} *@irit.fr* – {*gomezdor, poulin*} *@iro.umontreal.ca*

***Abstract*—In order to interactively render large virtual worlds, the amount of 3D geometry passed to the graphics hardware must be kept to a minimum. Typical solutions to this problem include the use of potentially visible sets and occlusion culling, however, these solutions do not scale well, in time nor in memory, with the size of a virtual world. We propose a fast and inexpensive variant of occlusion culling tailored to a simple tiling scheme that improves scalability while maintaining very high performance. Tile visibilities are evaluated with hardware-accelerated occlusion queries, and in-tile rendering is rapidly computed using BVH instantiation and any visibility method; we use the CHC++ occlusion culling method for its good general performance. Tiles are instantiated only when tested locally for visibility, thus avoiding the need for a preconstructed global structure for the complete world. Our approach can render large-scale, diversified virtual worlds with complex geometry, such as cities or forests, all at high performance and with a modest memory footprint.**

***Keywords*-occlusion culling, visibility, procedural modeling, tiling, PVS, CHC++, BVH**

I. Introduction

Many of today's interactive media outlets, such as video games, immerse users in large virtual worlds. These worlds tend to be diversified, complex, and very extensive. Procedural modeling and tiling sets can be used to create such large virtual worlds at reasonable costs, alleviating the tedious task of modeling them by hand (Figure 2).

Procedural modeling encompasses several kinds of automatic generation methods in order to model synthetic scenes. Some of these methods allow designers to create procedural geometry or textures. Tiling methods can serve as an alternative or supplemental solution to this design, where well-designed portions of a world (enclosed in 3D tiling volumes, as illustrated in Figure 1) are randomly, deterministically, or a combination of both, laid out to give the impression of variations and near-infinite extent.

Procedural generation of such extended scenes is often used as a preprocess with respect to the rendering or interactive walkthrough. By integrating rendering constraints, such as visibility computation, in a tiling-based scene generation process, we can ensure that the generation of only the information necessary for rendering the scene is provided, given a fixed performance budget.

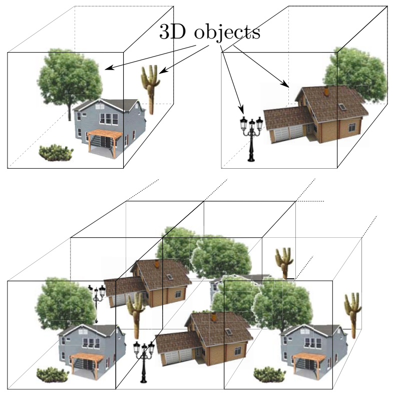


Figure 1: Using a tiling set of two predesigned tiles (top) in a 3D world tiling process (bottom). Objects are placed into these tiles forming the tiling set. These are randomly or deterministically instantiated in 3D space to form the final world geometry.

Despite the fact that we cannot guarantee that all visible polygons will be displayed within a fixed time limit (for instance, in extreme cases, everything could be visible at once), we intend to accelerate visibility determination and minimize the number of occluded polygons sent to the graphics pipeline. Two common solutions consist of performing a real-time occlusion culling or precomputing *potentially visible sets* (PVS) of scene geometry for bounded view regions in order to reduce the costly on-the-fly determination of visible objects.

Our contribution is a tiling creation process that *propagates* the visibility information across tiles from the viewpoint. Local visibility is computed within each 3D tile using CHC++, while global visibility is propagated with extra queries onto neighboring tiles, in order to integrate visibility in the construction of tileable grid-based 3D worlds. The proposed method allows complete freedom in scene design and gives high performance when rendering extended, and potentially infinite, virtual worlds. Because potentially visible tiles are processed in front-to-back visibility order from the viewpoint, fully occluded tiles do not need to be visited,

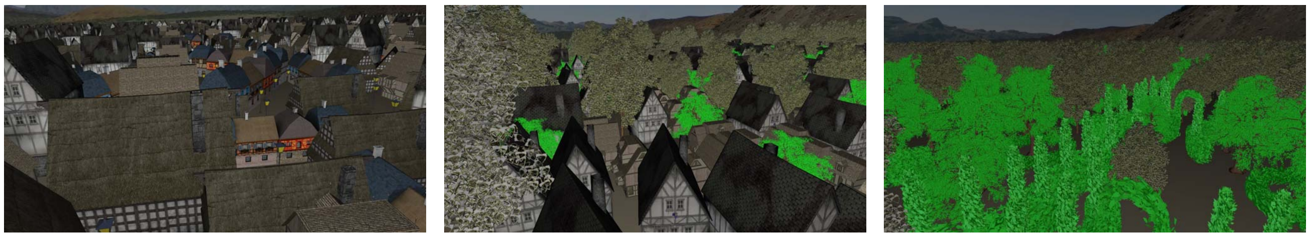

(a) City (b) Hybrid scene (c) Forest

Figure 2: Three tiled scenes generated from three different tiling sets. Our tiling creation process integrates visibility propagation in the construction of tileable 3D worlds allowing for much more interactive rendering. Mountains in the background come from the skybox.

nor even generated. Moreover, no precomputed global data structures for the entire scene are necessary, thus avoiding extreme memory and processing requirements.

Visibility computation, procedural generation of scenes, and tiling methods are well-studied topics in computer graphics (Section II). Building on these, our tiling creation process first computes a per-tile local visibility information using a fast instantiation of a bounding volume hierarchy (BVH; Section III-A) to bootstrap to the CHC++ algorithm. Global visibility computation relies on per-neighbor tile queries using view-frustum culling and GPU occlusion queries, effectively reducing some limitations of CHC++ and adding frame-to-frame temporal coherence to speed up the entire process (Section III-B). Popping artifacts resulting from variation over time and space of the occlusion queries, are efficiently avoided by our method (Section III-C). We analyze our performance on a set of typical procedurally generated scenes (Section IV) and discuss the benefits and limitations of our approach (Section V). Finally, we conclude and offer future perspectives (Section VI).

II. Related Work

A. Visibility

Visibility determination remains one of the most challenging problems in computer graphics, particularly with respect to efficient rendering. For this reason, the literature on this topic is very extensive and we refer readers to surveys [1], [2] for a comprehensive review. We instead restrict ourselves to discussions on visibility (pre)computation methods for interactive rendering.

The two general families of methods used to efficiently display large and complex scenes are PVS and occlusion culling. By neglecting completely occluded objects, these methods can reduce the rendering load on both CPU and GPU.

The concept of PVS, introduced by Airey et al. [3], consists of determining all the polygons that can be potentially visible from a given convex region, called a *view cell*.

In order to increase PVS construction efficiency, an important step consists of computing the fusion of occluders and the aggregation of regions farther away from a view cell. Schaufler et al. [4] use an octree to store opaque volumetric interiors (of city buildings in their application). By conservatively projecting the occluded regions through the octree, they efficiently classify regions as occluded for given view cells. Durand et al. [5] also use the concept of occluder fusion and region aggregation in general 3D scenes, but reproject occluders on successive parallel planes instead. Both methods compute conservative PVS, but their respective preprocessing remains very compute-intensive, and their resulting PVS require large amounts of storage.

Several outdoor 3D worlds can be represented by heightfields. By limiting the maximal displacement speed of an observer in a 2.5D heightfield scene, Koltun et al. [6] achieve interactive rendering with a PVS computed on the fly. Wonka et al. [7] propose a conservative approach also limited to 2.5D, but instead supported by raycasting. Their method is very efficient compared to other geometrical approaches because of the discretization of the scene. It can compute a PVS in only a few seconds. Bittner et al. [8] use sample rays across the scene to simultaneously compute the PVS for an entire set of view cells. Unfortunately this method requires large amounts of storage and cannot be used in a traditional rendering engine.

Occlusion culling consists of quickly testing if an object is occluded by another one, visible from the viewpoint. *Occlusion queries* and *early-z rejection* are commonly used for real-time occlusion culling methods.

Greene et al. [9] traverse an octree in a hierarchical Z-buffer software rasterizer for fast polygon visibility determination with both spatial and temporal coherence. Zhang et al. [10] introduce a similar approach for visibility culling but instead with hierarchical occlusion maps, thus allowing image occluder fusion resulting in significant speedups for interactive walkthroughs.

Staneker et al. [11] use occupancy maps to allow scene graph rendering systems to perform efficient occlusion culling. They organize multiple occlusion queries by priority, even though a scene graph hierarchy is not as efficient as BVHs to perform front-to-back node traversal. They also use temporal coherence to fix a budget on the number of individual occupancy maps tests before GPU occlusion multi-queries. This allows for good spatial and temporal culling coherence if the scene graph remains unmodified.

Because of our use of CHC++, we describe it in more details here. Mattausch et al. [12] present CHC++ using occlusion multi-queries on a BVH representation of the scene. Their algorithm handles millions of polygons while maintaining very high rendering framerates. The CHC++ algorithm recursively queries the BVH nodes to determine whether they are visible from the viewpoint. If a node is visible, CHC++ either displays its geometry if it is a leaf, or recursively traverses its children. If the node is not visible, the node and its subtree are culled. According to the node's polygon count and its spatial extent, it is sometimes preferable to render the entire subtree geometry rather than to determine which children nodes are visible or not. Moreover, the traversal cost, also related to the depth of a subtree, must be limited in order to get CHC++ to work efficiently without too much overhead. CHC++ efficiently exploits temporal coherence, and a node might be marked as visible for a bounded random number of frames. This alleviates the GPU load: the occlusion queries on the BVH nodes are thus spread out across several frames. Nevertheless, as in previous approaches, the scene must remain unmodified to benefit from temporal coherence. In this paper, we name *CHC++ context* the set of resources (query queues, BVH, etc.) used by CHC++ to compute occlusion culling.

Both PVS and occlusion culling methods significantly improve performance. However, for very large worlds, the preprocessing time for building the required data structures is often too long, and the scene must remain static. Moreover, the memory requirements for storing the PVS or the BVH become a limiting factor in real-world rendering engines. Finally, when dealing with interactively and procedurally generated very extended worlds, neither method is convenient due to their precomputation times and memory costs.

B. Procedural Modeling and Tiling

Tiling is often used in procedural modeling to ensure a virtual world's variability over large distances and scales [13]. One compelling example was provided by Peytavie et al. [14] with their aperiodic tiling set of corner cubes to generate piles of rocks structures for landscapes, stone huts, walls, etc.

Despite many existing procedural methods for objects and landscape generation, only a few consider positioning constraints on objects in the scene, and none consider guiding the scene generation with visibility, except for two methods. The first method, from Greuter et al. [15], uses the view frustum to generate a procedural city, but without any other form of culling. The second, from Gomez et al. [16], uses 2.5D visibility to precompute occluding tiles, ensuring a fixed PVS size. However this latter method is limited as scene designers cannot authorize long avenues, where visibility extends far away. In addition, the algorithmic complexity of the method does not scale well to fully 3D scenes.

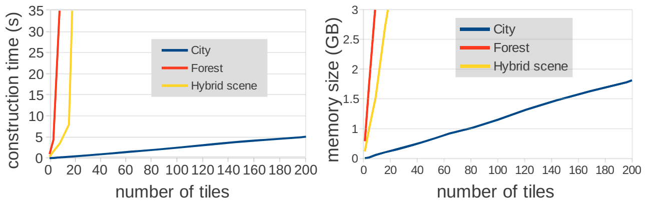


Figure 3: Time and memory measured for the naive building of the BVH of a tiled world. Our method avoids these costs with a low preconstruction time on each tile BVH.

III. Visibility Tiling

Tiling can efficiently generate large 3D worlds based on a set of predefined tiles. However, building a BVH for efficient occlusion culling on such an entire generated scene is impractical as the scene's size increases. Indeed, Figure 3 shows time and memory consumptions for BVH construction using all the objects of a tile-based world. Our method improves the performance of such world generation independently from the tile placement procedure. As such, we do not require the location of each tile to be explicitly stored, e.g., in a tile-instantiation map or by storing neighborhood information.

Our tiling scheme takes a predesigned set of tiles as input (Figure 1 top). These tiles are considered as axis-aligned boxes for simplicity, without loss of generality. In fact, these tiles have no restrictions on their shape. The geometry associated with each tile is defined inside a local scene graph of 3D objects instances freely arranged in the tile. To generate the 3D world, the tiles are instantiated in 3D space, on a 2D surface for our test scenes, to form the final world geometry (Figure 1 bottom), depending on whether they may be visible from the viewpoint according to a two-level occlusion culling algorithm. The first level applies hardware occlusion queries on the tiles' common bounding box, i.e., the union of each individual tile's bounding box (Figure 4) to allow for long-range visibility determination. The second level uses Mattausch et al.'s CHC++ algorithm [12] to determine the local occlusion within each visible tile.

When the extent of the scene is very large, for instance for an entire virtual world, even with instantiated BVHs, the time and memory necessary to build a hierarchical structure, as well as the increased cost of traversing this hierarchy

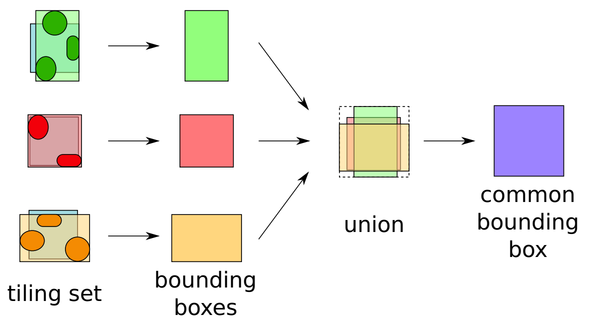


Figure 4: A 2D tiling set consisting of three tiles: their bounding boxes (green, red, orange) are computed with respect to all the tile's objects. Their union, the common bounding box (purple), is the one used for tile occlusion queries.

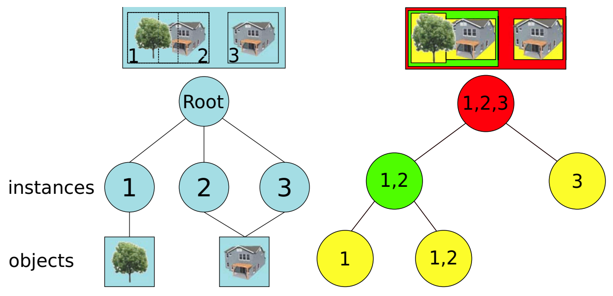


Figure 5: Left : The structure of a scene graph used for instantiation. The numbered nodes correspond to scene graph instances. Right : The BVH structure built for a tile (in red). Each node keeps references to the corresponding instantiation scene graph nodes it belongs to.

for occlusion queries, can become major bottlenecks. Our grid-based front-to-back tile visibility algorithm avoids such difficulties by expanding locally the number of queried tiles without the need to store them in any hierarchical structure.

We first construct a per-tile bounding box and a BVH from the local scene graph associated with the distribution of objects in the tile. The bounding boxes of all tiles in a tiling set are then combined to define the common bounding box of the tiling set (Section III-A). This common bounding box is used to avoid popping artifacts (Section III-C). Using this information, the tiling is generated on the fly, by propagating the visibility from the viewpoint over the tiling until each line of sight is blocked, using our two-level occlusion culling algorithm (Section III-B).

A. Tiling Set Representation

In order to limit memory usage for a large 3D world our method employs object instantiation, represented efficiently with a scene graph. Each tile has its own scene graph and may share objects with other tiles. From this per-tile scene graph (Figure 5 left), we build a per-tile BVH in

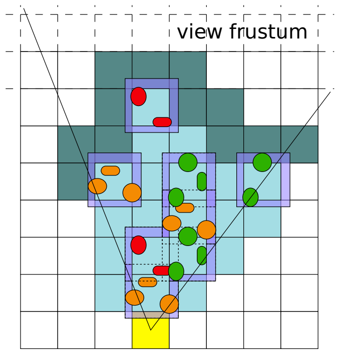


Figure 6: 2D tiling process within the view frustum. Visibility is propagated from the nearest visible tile (yellow). Tiles are shown in light blue if query result is positive, in dark blue if it is negative.

order to compute efficient occlusion queries. Each BVH node contains references to the corresponding scene graph instantiation nodes, but not to the geometry itself. Each BVH is built using the binned SAH heuristic [17], [18]. The recursive BVH construction stops when there is only one object referenced in a leaf node, or when either a maximum depth or a minimum number of triangles per leaf is reached (Figure 5 right). Having a BVH for each tile is important for interactivity and computational efficiency. If a tile contains animated geometries, only its tile's BVH must be recomputed at each frame, which can be done quickly. Animated geometries could also be handled in a separate per-tile BVH, or in a global BVH for the entire world as in traditional implementations.

The BVH's root node defines the tile's bounding box. From each tile bounding box, we compute the tiling set common bounding box, which is unique for the entire tiling set. This bounding box, computed as the union of every tile's bounding box as shown in Figure 4, allows for conservative occlusion determination, independently of the tile, during the tiling process.

B. Visibility Propagation with Tile Queries

For each frame, visibility is computed from the viewpoint and the tiling process propagates visibility from the nearest visible tile to adjacent tiles, and so on, in front-to-back order. The nearest visible tile is found with a search in the tiling domain. The tiling process uses the tiling common bounding box to determine if a tile instance must be rendered or if visibility propagation should stop (Figure 6). We call this step the "tile query". If the tile has to be instantiated, the tiling process selects the associated tile from the set of predesigned tiles. For this selection, we use a function that returns the selected tile from its world position, thus avoiding explicit storage of every instantiated tile position; this is particularly important for scalability to large worlds.

```
1 $\mathcal{Q} \leftarrow \emptyset$ // pending query queue
2 $\mathcal{P} \leftarrow \emptyset$ // postponed query queue
  // retrieve the first tile to render
3 $\mathcal{T} \leftarrow$ closestVisibleTileInFrustum($camera$)
  // propagate visibility
4 while $\mathcal{T} \neq \emptyset$ do
5     renderWithCHC++(fetchBVHInstance($\mathcal{T}$))
6     queryNeighbors($\mathcal{T}$,$\mathcal{Q}$)
      // retrieve next tile to render
7     $\mathcal{T} \leftarrow$ getNextPositiveQuery($\mathcal{Q}$,$\mathcal{P}$)
8 end
9 updatePostponedQueryQueue($\mathcal{P}$)
```

Algorithm 1: Tile Query Main Algorithm

```
Input :   • current tile $\mathcal{T}$
          • pending query queue $\mathcal{Q}$
Output :  • pending query queue $\mathcal{Q}$
  // query neighbor tiles of $\mathcal{T}$
1 foreach neighbor $n$ of $\mathcal{T}$ do
2     if $\neg$ queriedForThisFrame($n$) then
3         if intersectsViewFrustum($n$) then
              // asynchronous query launch
4             $q \leftarrow$ query(commonBboxAtOffset($n$))
5             append($\mathcal{Q}$, ($q$,$n$))
6         end
7         setQueriedForThisFrame($n$)
8     end
9 end
```

Algorithm 2: queryNeighbors($\mathcal{T}$,$\mathcal{Q}$)

In the remainder of this section, we describe the algorithms used by our visibility propagation algorithm. The instructions written in blue are related to temporal visibility prediction. They could be neglected if temporal coherence is not exploited.

The main rendering steps are detailed in Algorithm 1. Starting from the nearest visible tile, the corresponding BVH is instantiated if it has not yet been. The BVH instance is created such that each node can determine its own visibility status (like with CHC++) when rendered with local occlusion culling. When a BVH node is marked as visible, we add the corresponding scene graph nodes instances to the rendering, if they were not already added for the current frame (Figure 5). When the current tile geometry has been rendered with local occlusion culling using CHC++, farther neighboring tiles are queried for visibility (Algorithm 2) using the same common bounding box that is offset at the queried tile position. To benefit from the front-to-back generation of the world and increase the efficiency of our occlusion culling method, tile queries are stored in a queue $\mathcal{Q}$ because hardware query specifications guarantee that occlusion queries results are returned in order.

In Algorithm 3, we send visible tiles to the renderer and predict their visibility for the next frame. If the result of a query is not readily available, we use temporal coherence (line 3): if it was visible in the previous frame, we postpone the query result processing until the end of the current frame (postponed query queue $\mathcal{P}$) and send the tile for rendering. As most of these query results will be available at the end of the frame, the correct visibility status will be determined before the next frame.

```
Input :   • pending query queue $\mathcal{Q}$
          • postponed query queue $\mathcal{P}$
Output :  • pending query queue $\mathcal{Q}$
          • postponed query queue $\mathcal{P}$
          • neighbor to render $n$
  // retrieve first available AND positive query result
1 while $\neg$ empty($\mathcal{Q}$) do
2     ($q$,$n$) $\leftarrow$ pop($\mathcal{Q}$)
3     if $\neg$ isResultAvailable($q$) $\wedge$
      wasVisibleAtPreviousFrame($n$) then
          // postpone query result retrieval
4         append($\mathcal{P}$, ($q$,$n$))
5         return $n$ // send to rendering
6     else
7         waitQueryResult($q$)
          // next frame visibility prediction
8         if isResultPositive($q$) then
9             setVisible(n)
10            return $n$ // send to rendering
11        else
12            setNotVisible($n$)
13        end
14    end
15 end
16 return $\emptyset$ // no neighbor to render
```

Algorithm 3: getNextPositiveQuery($\mathcal{Q}$,$\mathcal{P}$)

```
Input :   • postponed query queue $\mathcal{P}$
1 // get visibility result for next frame
2 while $\neg$ empty($\mathcal{P}$) do
3     ($q$,$n$) $\leftarrow$ pop($\mathcal{P}$)
      // next frame visibility prediction
4     if isResultAvailable($q$) $\wedge$ isResultPositive($q$)
      then
5         setVisible($n$)
6     else
7         setNotVisible($n$)
8     end
9 end
```

Algorithm 4: updatePostponedQueryQueue($\mathcal{P}$)

Due to our front-to-back tiling traversal, the first queries in the query queue $\mathcal{Q}$, that correspond to tiles close to the viewer, are more likely to be predicted as visible, and thus, have their result postponed. Most of the following query results from the query queue will then be available, as queries are launched according to increasing distance from the viewer using a neighborhood relationship. Line 7 of Algorithm 3 is executed when a tile becomes newly visible. This affects the framerate when the number of such tiles is large but the time spent to wait could be used to pre-instantiate the remaining tiles in the pending query queue.

At the end of the main algorithm, we check if the results of all the postponed queries have become available, and update prediction for the next frame (Algorithm 4). At line 4 of Algorithm 4, tile query results that are not available at this moment are those from tiles that are far away from the viewpoint. Predicting these tiles as not visible will force Algorithm 3 to wait at line 7 for the query result retrieval but, as they are far and have a small footprint on screen-space, this cost should be small. We also noticed that the number of such tiles is small: as shown experimentally in our test scenes, the number of queries failing in temporal prediction are only 0.02% (City), 0.04% (Hybrid scene), and 0.14% (Forest) of the total number of queries.

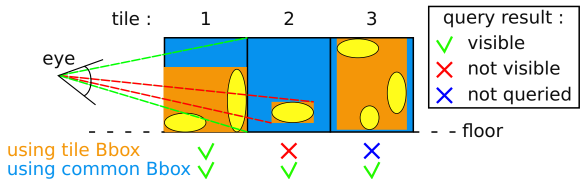


Figure 7: The common bounding box, rather than the tile's bounding box, ensures the correctness during visibility propagation. The query of Tile 2 will return visible, and therefore Tile 3 will be queried for visibility and will not suddenly appear when passing through Tile 1 or when the observer's height increases.

Temporal coherence results in a significant speed-up in terms of framerate. Since queries are performed in a front-to-back order, most of the queries for which results are not yet available are those from tiles that are closer to the viewpoint. As such, most of these tiles were already visible in the previous frame and were therefore predicted to be visible at the current frame.

When a tile instance is marked as visible for the first time, the corresponding BVH instance is kept in a cache to manage its own CHC++ context. Tile query temporal coherence is independent and complementary to CHC++'s own temporal coherence. We keep instantiated tiles in the cache for the subsequent frames, so that we preserve the CHC++ temporal coherence and avoid instantiation overheads from one frame to another. When a tile is not visible for a specified duration, we remove it from the cache, reducing memory usage.

C. *Avoiding Popping Artifacts*

Using simply tiles' bounding boxes and direct neighborhood to propagate visibility across the tiling is not sufficient to ensure that a closer tile completely occludes a farther one. Figure 7 shows an example in which such a simple algorithm would fail in determining potentially visible geometry located farther away from the maximal extent of the instantiated tiling. Using the common bounding box for the tile queries overcomes this problem. As every queried bounding box has the same size, as soon as a query result is negative, we know that farther tiles will not be visible from any viewpoint, i.e., inside or outside of the tiling. Therefore, querying farther neighbor tiles for occlusion will not be necessary and visibility propagation can stop.

IV. Results and Performance Analysis

We tested our algorithm on different tiling sets composed of houses and trees, as shown in Figure 2. The geometries come from *turbosquid* [19]. Each tile holds between 16K and 2.5M polygons divided among all the 3D objects in the tile. No *levels of detail* (LODs) are used on these objects, such that the entire geometries are rendered when declared visible. Computations are performed on an Intel Core i7 with a GeForce GTX 680. The BVH of each tile is constructed with the *embree* parallel implementation of BVH builder [20], using a spatial split heuristic. The BVH construction times, depths, and memory for the tiling sets precomputations are given in Table I.

In the accompanying video, we show three walkthroughs within three tilings, each one set up with a different tiling set. We compare four culling algorithms: tile query + CHC++, tile query alone, per-tile CHC++, and view frustum only. The tile query algorithm alone systematically renders the entire geometry of each visible tile instead of rendering it with CHC++. A direct comparison between our method and the CHC++ algorithm cannot be done since, as said in Section III, building the entire BVH of our very extended scenes is impractical. But in order to show the benefits of our method in a fairer way, we compare it with a per-tile CHC++ algorithm in which we consider, for each frame, each tile BVH within the conservative bounding box of the tiles instantiated using the tile query propagation method (preserving CHC++ temporal coherence). We use the same a priori information for the view frustum culling algorithm alone. In fact, per-tile CHC++ and view frustum culling alone are favored in our implementation in regard to a possible naive implementation. Figures 8 and 9 show the framerates and the numbers of rendered triangles for these walkthroughs.

As we previously mentioned, it is sometimes preferable to render the entire geometry contained in a tile rather than determining its occlusion status. As well, we can see that using the tile query algorithm alone performs sometimes slightly better than using it in conjunction with CHC++ in the City and the Hybrid scenes, where the scenes are not very complex and produce much occlusion (Figures 8a and 8b). Occasionally, the tile query algorithm even performs twice as better, as shown between frames 2100 and 2200 of the City. But tile query alone induces more variance in the framerate.

We can see that for most viewpoints, our algorithm in conjunction with CHC++ keeps better framerates than per-tile CHC++ (Figure 8). This is easily explained by the fact that CHC++ does not handle visibility information or temporal coherence on neighboring tiles, and thus more tiles are instantiated in the tiling, and more time is spent to resolve individual tile queries. The Forest walkthrough is the one for which our hybrid algorithm performs the best in aggregating the occlusion of several tiles (Figure 8c). In each of the three scene walkthroughs, the framerates decrease a lot when the observer's location is high above the ground, looking at the tiling at grazing angles, resulting in many visible tiles thus being instantiated. This slowdown could be reduced using coarser geometry for distant instantiated tiles (LODs), but we decided to leave this for future investigations.

In terms of rendered triangles (Figure 9), our algorithm performs as well as per-tile CHC++, i.e., tile query associated with CHC++ is as much conservative as per-tile CHC++ is in terms of occlusion.

V. Discussion

As explained in Section III-C, using a common bounding box ensures that no popping artifacts occur. But this can affect the framerate in the cases of an overestimation of the visibility of a tile. If the bounding box of a particular tile is small with respect to the common bounding box, this tile will be often marked as visible even if it is not the case. Then, the instantiation and the rendering of the tile using CHC++ will introduce unnecessary computational costs.

As with every occlusion culling method, the efficiency of our algorithm greatly depends on the occlusion in the scene. In our case, as occlusion culling is performed on the tiles, the efficiency depends on the occlusion of each tile. If the tiles do not contain enough occluding geometry, the tiling process will propagate the visibility to many tiles.

As with CHC++, temporal coherence can be increased so that when a tile is marked as visible, we can set it as visible for a random number of subsequent frames, resulting in fewer queries spread over time. We can also use the front-to-back order to decrease the frequency of closer tiles' queries. In our test scenes this did not result in significant speed-up, because our tiles generate enough occlusion.

In our algorithm, the geometry of an instantiated object is completely rendered once determined visible. If this geometry is complex, an efficient LOD system would allow to decrease the number of rendered polygons for farther tiles and improve overall performance.

VI. Conclusion and Future Work

In this paper, we associate visibility and tiling. Within this context, we propose a solution for occlusion culling in 3D worlds generated on the fly. This allows for real-time occlusion culling and low memory BVH instantiation for extended worlds. In our method, tiles are instantiated on a 2D surface but could also be instantiated in 3D. Simply put, only the neighborhood relationship would have to be extended to 3D.

Although BVH computation might not always be achieved on the fly, it is still feasible for reasonable numbers of polygons per tile and depths of the computed BVH. In a multithreaded implementation, a tile BVH could be prefetched in a separate thread, when a neighboring tile is queried, so that such a tiling process could even create the geometry in the tile itself, while instantiating it. Thus, the tiling set would not need to contain predesigned tiles if they do not exceed a given number of polygons, depending of the rendering performance.

Repositioning of procedural geometry according to visibility analysis could also emerge from our structures in order to limit visibility propagation. As a result, level designers could exploit such tools for the fast creation of huge worlds. Moreover, using this method could alleviate the whole PVS construction cost induced by the number of polygons, by dividing it into small computation steps. Finally, we decided to use the CHC++ algorithm, but in fact, this local occlusion culling step could be replaced by any other conservative, exact, or sampling visibility determination method.

For even better occlusion culling, and therefore better rendering framerates, our method can be used in conjunction with hierarchical Z-buffer [9] or occupancy maps [11]. The latter one would use directly the BVH nodes instead of the scene graph nodes to test and update the occupancy map.

We also would like to develop a hierarchical version of this method : visibility would be propagated very quickly over large distances at the top level of the hierarchy.

Acknowledgements

The authors thank the anonymous reviewers for their constructive comments. They acknowledge financial support in Canada from GRAND.

References

[1] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand, "A Survey of Visibility for Walkthrough Applications," *IEEE Trans. Visualization and Computer Graphics*, vol. 9, no. 3, pp. 412–431, 2003.

[2] J. Bittner and P. Wonka, "Visibility in Computer Graphics," *Journal Environmental Planning*, vol. 30, pp. 729–756, 2003.

[3] J. M. Airey, J. H. Rohlf, and F. P. J. Brook, "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments," in *ACM Symposium on Interactive 3D Graphics*, 1990, pp. 41–50, 258.

[4] G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion, "Conservative Volumetric Visibility with Occluder Fusion," in *Proc. SIGGRAPH '00*, 2000, pp. 229–238.

[5] F. Durand, G. Drettakis, J. Thollot, and C. Puech, "Conservative Visibility Preprocessing Using Extended Projections," in *Proc. SIGGRAPH '00*, 2000, pp. 239–248.

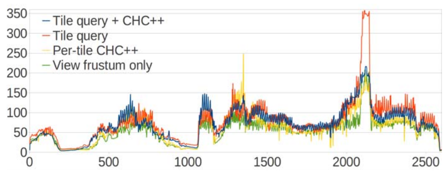
[6] V. Koltun, Y. Chrysanthou, and D. Cohen-Or, "Virtual Occluders: An Efficient Intermediate PVS Representation," in *Proc. Eurographics Workshop on Rendering*, 2000, pp. 59–70.

[7] P. Wonka, M. Wimmer, and D. Schmalstieg, "Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs," in *Proc. Eurographics Workshop on Rendering*, 2000, pp. 71–82.

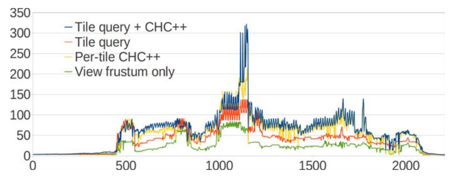
[8] J. Bittner, O. Mattausch, P. Wonka, V. Havran, and M. Wimmer, "Adaptive Global Visibility Sampling," *ACM Transactions on Graphics*, vol. 28, no. 3, pp. 94:1–10, 2009.

[9] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," *Proc. SIGGRAPH*, pp. 231–240, 1993.

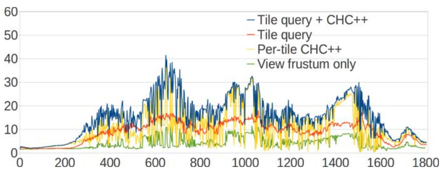
[10] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff, III, "Visibility Culling Using Hierarchical Occlusion Maps," in *Proc. SIGGRAPH*, 1997, pp. 77–88.



(a) City

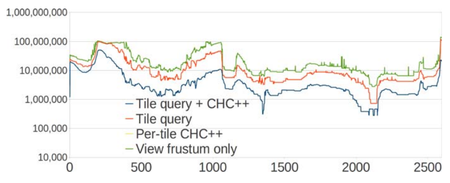


(b) Hybrid scene

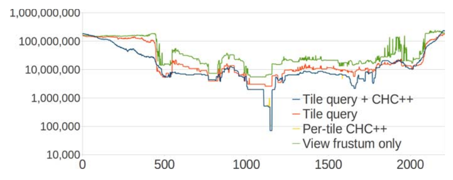


(c) Forest

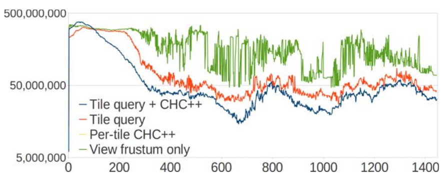
Figure 8: The FPS for the three different walkthroughs in the three different scenes (higher is better).



(a) City



(b) Hybrid scene



(c) Forest

Figure 9: The rendered triangles for the three different walkthroughs (lower is better). Blue and yellow curves overlap.

Table I: BVH construction statistics for the tiles of the three test scenes.

Tile ID	City				Hybrid scene				Forest			
	time (ms)	depth	size (MB)	# polys	time (ms)	depth	size (MB)	# polys	time (ms)	depth	size (MB)	# polys
Tile 0	184	16	24.6	0.768M	251	16	46.6	1.100M	453	16	61.2	2.41M
Tile 1	132	16	27.1	0.728M	181	16	35.9	0.934M	484	16	67.8	2.31M
Tile 2	103	16	20.9	0.580M	391	16	54.9	2.129M	472	16	66.6	2.48M
Tile 3	85	16	16.7	0.474M	184	16	34.0	0.967M	431	16	59.8	1.98M
Tile 4	85	16	16.1	0.674M	114	16	26.8	0.543M	215	16	47.1	1.03M
Tile 5	117	16	21.2	0.363M	125	16	30.3	0.607M	217	16	44.2	1.06M
Tile 6	67	16	15.8	0.276M	45	16	11.2	0.174M	278	16	55.5	1.42M
Tile 7	61	16	14.9	0.580M	96	16	23.8	0.445M	283	16	54.5	1.41M
Tile 8	104	16	19.9	0.050M	192	16	42.2	0.908M	121	16	30.3	0.47M
Tile 9	81	16	15.6	0.459M	111	16	27.6	0.528M	99	16	25.3	0.42M
Tile 10	106	16	27.8	0.562M	10	13	1.3	0.024M				
Tile 11	72	16	18.3	0.365M	164	16	32.8	0.838M				
Tile 12	58	16	12.0	0.313M	74	16	19.9	0.316M				
Tile 13	90	16	18.0	0.494M	184	16	37.7	0.935M				
Tile 14	70	16	14.8	0.369M	94	16	22.5	0.462M				
Tile 15	84	16	18.6	0.472M								
Tile 16	102	16	22.4	0.565M								
Tile 17	57	16	14.9	0.275M								
Tile 18	86	16	19.2	0.472M								
Tile 19	58	16	17.0	0.275M								

[11] D. Staneker, D. Bartz, and W. Straßer, "Efficient Multiple Occlusion Queries for Scene Graph Systems," Universittsbibliothek Tbingen, Tech. Rep., 2004.

[12] O. Mattausch, J. Bittner, and M. Wimmer, "CHC++: Coherent Hierarchical Culling Revisited," *Computer Graphics Forum (Proc. Eurographics)*, vol. 27, no. 2, pp. 221–230, 2008.

[13] A. Lagae, C. S. Kaplan, C.-W. Fu, V. Ostromoukhov, J. Kopf, and O. Deussen, "Tile-Based Methods for Interactive Applications," in *ACM SIGGRAPH 2008 Courses*, 2008.

[14] A. Peytavie, E. Galin, S. Merillou, and J. Grosjean, "Procedural Generation of Rock Piles Using Aperiodic Tiling," *Computer Graphics Forum (Proc. Pacific Graphics)*, vol. 28, no. 7, pp. 1801–1810, 2009.

[15] S. Greuter, J. Parker, N. Stewart, and G. Leach, "Real-time Procedural Generation of 'Pseudo Infinite' Cities," ser. Graphite '03. ACM, 2003, pp. 87–94.

[16] D. Gomez, P. Poulin, and M. Paulin, "Occlusion Tiling," in *Graphics Interface 2011*, May 2011, pp. 71–77.

[17] D. J. MacDonald and K. S. Booth, "Heuristics for Ray Tracing Using Space Subdivision," *Vis. Comput.*, vol. 6, no. 3, pp. 153–166, May 1990.

[18] V. Havran, "Heuristic Ray Shooting Algorithms," Ph.D. Thesis, Dept. of Computer Science and Engineering, Czech Technical University in Prague, 2000.

[19] Turbo Squid Inc. (2013) 3D Models, Textures and Plugins at TurboSquid. [Online]. Available: http://www.turbosquid.com

[20] S. I. Woop. (2012) Embree - Photo-Realistic Ray Tracing Kernels. [Online]. Available: http://software.intel.com