Occlusion Tiling

Dorian Gomez, Pierre Poulin, Mathias Paulin

To cite this version:

Dorian Gomez, Pierre Poulin, Mathias Paulin. Occlusion Tiling. Graphics Interface, May 2011, St. Jonh's, Canada. pp.71-78, 10.20380/GI2011.10 . hal-01518541

HAL Id: hal-01518541
https://hal.science/hal-01518541

Submitted on 4 May 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Occlusion Tiling

Dorian Gomez [1,2] * Pierre Poulin [1] ⋄ Mathias Paulin [2] •

[1] LIGUM, Dept. I.R.O - Université de Montréal [2] IRIT-VORTEX - Université Paul Sabatier, Toulouse, France

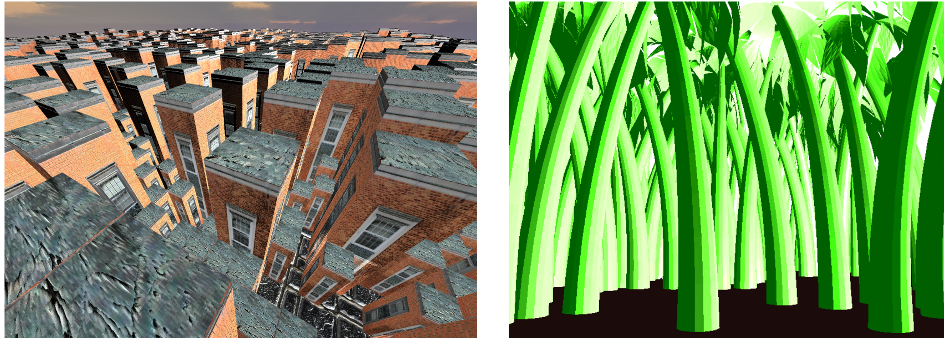


Figure 1: Left : An extruded city from our 2D occluders; the layout of buildings ensures full occlusion of the field of view as long as the observer remains below a specified height. Right : Palm trees positioned by *Occlusion Tiling*. Occlusions are based on an internal square at the base of the tree. The distribution also verifies that two tree trunks are not adjacent to each other.

ABSTRACT

The creation of realistic, complex, and diversified virtual worlds is of utmost importance for video games. Unfortunately the amount of time required to create 3D scene contents can be extremely tedious to graphic artists. While procedural modeling can alleviate this task, it has mostly been developed for specific contexts.

In this paper, we study tiling for synthetic worlds, taking into account visibility between tiles. We propose a method, *Occlusion Tiling*, that precomputes full 2D occlusion caused by tiles in order to ensure that a limited number of tiles can be visible from any viewpoint on the tiling. These tiles are then used as extruded 3D scenes, thus bounding the number of polygons sent to the graphics rendering pipeline for guaranteed throughput.

Keywords: Occlusion Culling, Tiling, Procedural Modeling, Procedural Rendering

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms

1 INTRODUCTION

When virtual worlds need to be tremendous, diversified, and complex in order to spur a user's sense of immersion, artists are left with the tedious task of designing the worlds' variability over wide distances and scales. Procedural modeling can be used to solve this problem by developing automatic generation techniques, but they are often limited to specific world types, such as forests, cities, and landscapes, and they can be difficult to control by artists.

Tiling is another commonly used solution, where well-designed portions of a world are rearranged to give an impression of infinity. Tiling has been frequently used in video games, such as in "Heroes of Might and Magic", "Civilization", and "Sim City".

However, when rendering these large worlds from potentially any viewpoint, a standard z-buffer or ray tracing approach cannot guarantee to treat all objects within a fixed time period. One solution consists of precomputing *potentially visible sets* (PVS) for bounded regions. Several techniques exist to compute PVS, but none has been integrated in scene design with the goal to reduce their sizes.

We present our new *Occlusion Tiling* technique that optimizes occlusion of the visual field and can satisfy other scene properties, by placing objects (therein called *occluders*) on each tile.

Here, a 2D square tile is regularly subdivided into a grid, and filled (occluding) pixels define occluders. A 3D scene corresponds to an extrusion of these grid pixels (e.g., with buildings), or to 3D objects located over them and producing equivalent guaranteed occlusion. Our content-generation method ensures occlusion of each tile along the four canonical directions with a new $\bar{u}$ scheme. The generated tiles can then be independently assembled into a tiling, potentially infinite. The tiling thus limits the number of polygons sent to the graphics pipeline as it ensures that objects located beyond a known distance (expressed as a ring of n tiles) from the view cell are not visible. Our approach can be used as a design tool to automatically generate virtual worlds, to assist an artist in his scene generation, or to optimize the occlusion of previously designed worlds, that in the end guarantee high level rendering performances.

By combining visibility computation in 2D or 3D scenes (Section 2.1) and tiling techniques (Section 2.2), we introduce a new *Occlusion Tiling* method (Section 3), and describe its features, from occlusion computation and to its optimizations. In Section 4, we show how the method can be used for the distribution of city buildings and forest trees, considering constraints on the maximum height of a viewer. Finally, we conclude and propose some research directions that can be explored from our proposition, in Sections 5 and 6, respectively.

*e-mail : Dorian.Gomez@irit.fr
⋄e-mail : poulin@iro.umontreal.ca
•e-mail : Mathias.Paulin@irit.fr

2 Previous Work

2.1 Culling and Visibility Computations

Visibility determination is a broad and complex topic, that extends beyond the scope of our work. The interested reader is referred to the survey by Cohen-Or et al. [5] for an extensive study. Here, we will restrict ourselves to visibility (pre)computation methods that aim at accelerating rendering by limiting the number of polygons sent to graphics hardware.

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

Occlusion culling consists of testing if an object is occluded by another one, visible from the viewpoint. "Occlusion query" and "early-z rejection" are commonly used for real-time occlusion culling methods. Unfortunately, the occlusion tests tend to be too conservative (i.e., they do not cull enough objects), too slow (i.e., they are too costly to compute relative to the cost of rendering), or inexact (i.e., their approximations can produce popping artifacts).

The concept of *PVS*, introduced by Airey et al. [2], consists of determining all the polygons that can be potentially visible from a given convex region, called *view cell*. A PVS is computed for each view cell. PVS methods take advantage of that the rendering engine has only to test the list of polygons corresponding to the current view cell, and to render them possibly in conjunction with other culling methods, such as frustum culling. Nevertheless, despite several optimizations [1, 6, 9, 10, 13, 14, 16, 19], the memory required to store all PVS for all view cells can be huge for large scenes, and scene preprocessing can be very compute-intensive.

In order to increase the efficiency of PVS, an important step consists of computing the fusion of occluders and the aggregation of regions as we consider further distances from a view cell.

Schaufler et al. [14] use an octree spatial representation to store opaque volumetric interiors (of city buildings in their application). By projecting conservatively the occluded regions through the octree, they efficiently classify regions as occluded for given view cells. Durand et al. [9] also use the concept of occluder fusion and region aggregation in general 3D scenes, but reproject them on successive parallel planes. Both methods compute conservative PVS, but their respective preprocessing remains very compute-intensive, and their resulting PVS are very large.

Several methods exploit the coherencies inherent to height fields. By limiting the speed of an observer moving in a $2D\frac{1}{2}$ scene based on height maps, Koltun et al. [10] achieve interactive rendering in such scenes, but the current PVS computed on-the-fly forces the viewer not to switch too fast between view cells.

Wonka et al. [19] also propose a conservative approach in $2D\frac{1}{2}$, but instead supported by raycasting. Their technique is very efficient compared to other geometrical approaches because of the discretisation of the scene. It allows computing a PVS in only a few seconds.

Lloyd et al. [12] and Downs et al. [8] propose two sytems using real-time horizon culling to improve rendering speed respectively for terrain and urban walkthroughs. However, computing horizon culling at run-time may limit performances of the whole rendering process for very complex and extended scenes as the occlusion must be computed for each frame.

All these techniques have improved significantly the rendering speed of scenes. Unfortunately for large worlds, occlusion culling is not efficient enough, and PVS methods for several view cells result in memory requirements much too large. Worst, for infinite worlds procedurally generated on the fly, neither technique offers a satisfactory solution.

2.2 Aperiodic Tiling and Texturing

As a space partitioning method, tiling can occupy space in an optimal way. Furthermore, a small set of primitives can tile an infinite plane in a non repetitive way. Wang [17] proposed a formal system, square compound with *proto-tiles* and simple tiling rules. Berger [3] proved that there exist small sets of *Wang* tiles that can tile the plane aperiodically. This property is a critical aspect to avoid repetition in virtual worlds.

Tiling techniques are often used in computer graphics for texture synthesis. With an eye to produce non repetitive water surface textures, Stam [15] uses *Wang tiling*, precomputing sets of textured tiles. Cohen et al. [4] take a similar approach to synthesize textures with non-periodic tilings.

Lagae et al. [11] and Wei et al. [18] survey several domains in which tiling and texture synthesis are used, e.g., landscape modeling, non photo-realistic rendering, point or object distribution, texture mapping, and surface and ornament modeling.

In our proposal, tiling will help us to create non repetitive and extended worlds.

3 Occlusion Tiling

3.1 A Simple Model

Because of the simplicity of its representation, efficient storage, and generality in multiple computer graphics applications, we consider the square as our basic tile shape. The plane defining our world can thus be fully tiled by identical 2D squares. A tile is subdivided into a regular grid of $n \times n$ pixels (Figure 2a). Each tile pixel can be completely empty or completely occupied (forming an occluder), thereafter illustrated as a white pixel or a dark pixel, respectively. Each tile is also considered as an independent view cell.

The occlusion due to a tile is computed by analyzing the combined occlusions of all its occluding pixels. It is derived for all four canonical directions in the plane (Figure 4a), and its computation is described in Section 3.2.

At lower resolutions, a set of occluding tiles can be automatically generated in a brute-force way by systematically evaluating all possible occluding pixel configurations. Tiles of any resolution can also be generated by randomly setting occluding pixels according to a desired occluding density threshold. However, the computation being $O(n^3)$ for n occluding pixels, the purely random search could prove quite inefficient. We propose a multi-resolution scheme that improves the computations in Section 3.2.4. Finally, in order to give more artistic control over the distribution of occluders, we describe constraints in Section 3.3.

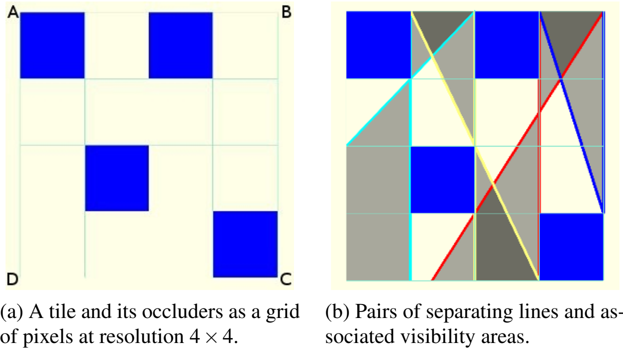
In Section 4 a few configurations of height distribution and multi-tiling are investigated in order to create more variations on the occluder heights.

3.2 Occlusion Properties

Each tile is defined by four vertices in clockwise order: A, B, C, D. A tile oriented as in Figure 2a has its *North* segment $[AB]$, *East* segment $[BC]$, *South* segment $[CD]$, and *West* segment $[DA]$.

We aim at limiting the visibility of an observer located on any tile (view cell) of our tiling up to a maximum distance expressed in tile units. If every view ray emanating from a view cell cannot go through its first-ring neighborhood (i.e., the eight tiles adjacent to the view cell), the field of view is completely obstructed by the occluders located on this 8-neighborhood. We also want each tile to be considered independently from its neighbors when creating a tiling.

Next we study the properties that the eight tiles of the 8-neighborhood must satisfy, without constraining excessively their occluding pixels.



(a) A tile and its occluders as a grid of pixels at resolution 4×4.

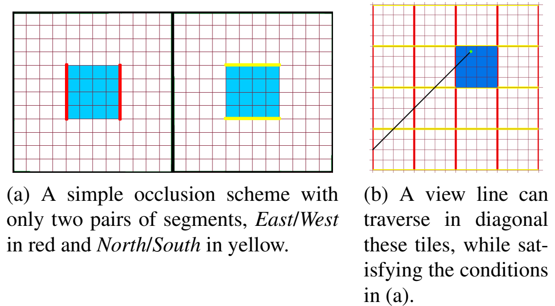
(b) Pairs of separating lines and associated visibility areas.

Figure 2: Visibility information from the top segment of a tile to the bottom segment (actually, a u-shape). Occluding tile pixels are blue, and empty pixels are white.

3.2.1 $\bar{u}$ Occlusion Scheme

A simple occlusion scheme could consider that a tile is fully occluding if all lines between the two pairs of opposite segments (sides) of a tile (North/South and East/West in Figure 3a) are blocked by occluding pixels. Unfortunately, this scheme is not sufficient to verify full occlusion for a set of tiles respecting these conditions. Figure 3b gives a counter-example of how a diagonal line could go through these validated configurations without being detected.

Extending the opposite segment to the other two segments of a tile (opposite segment plus the two entire side segments) would certainly provide sufficient conditions, but this scheme would also force corner pixels adjacent to the *incoming segment* to be occluders. This would result in tiles that are too constrained, with potentially visible patterns.

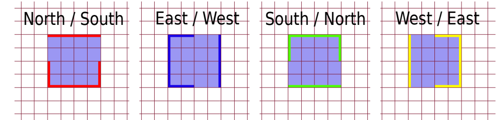


(a) A simple occlusion scheme with only two pairs of segments, *East*/*West* in red and *North*/*South* in yellow.

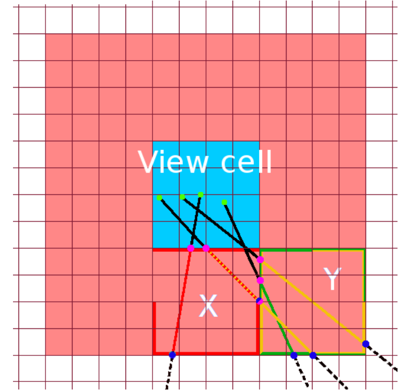
(b) A view line can traverse in diagonal these tiles, while satisfying the conditions in (a).

Figure 3: Insufficient occluding conditions.

In order to ensure full occlusion by the combined occlusion of adjacent pairs of tiles, while allowing for more flexibility in occluders configurations, we propose the conditions shown in Figure 4a. The base of the $\bar{u}$-shape is the receiving segment, augmented with the two sides that have a height of half a tile segment. If all individual tiles respect the $\bar{u}$-shape occlusion for all four directions, it is guaranteed that any view line emanating from a view cell will be blocked by its first-ring of tiles. We can see in Figure 4b that any view line starting from within the view cell and going through tiles X and Y is blocked by the *North/South* occlusion scheme in X, and by the *West/East* or *South/North* occlusion schemes in Y. These four constraints are a necessary and sufficient condition for *Occlusion Tiling* for each tile. Although other occlusion schemes may be used to ensure first-ring occlusion, we found this one quite satisfactory. A big advantage of this scheme is that by respecting these four conditions, each tile can be considered independently of its neighbors when generating a tiling.



(a) $\bar{u}$-shape : The four occlusion constraints (From / To) that must be applied to all tiles of a tiling set.



(b) The occlusion of the view lines beyond the first ring of neighborhood.

Figure 4: Occlusion scheme for each tile of a tiling set.

3.2.2 Computing Constrained Visibility

In order to test whether a tile respects the four occlusion constraints, we use the notion of *separating lines* [7], which is equivalent to separating planes in 3D. This is accomplished for all four conditions by the pseudo-code in Figure 5, provided for the *North*/*South* direction. Figure 2b shows the separating lines generated for this configuration.

This procedure computes all separating lines between every combination of two occluders from the tested tile, and checks if it intersects any (third) occluder. It is adapted and executed for each of the four directions (*North*/*South*, *East*/*West*, *South*/*North*, and *West*/*East*).

For n occluders, the naive algorithm complexity is $O(n^3)$, as it tests each ordered combination of occluders (o, o', o''). Each triplet is tested only once.

Figure 6 shows a particular case where o, o', and o'' are aligned. If o'' is on the same side of the separating line than o', it must be below o to be a valid separating line. If o'' is on the other side than o', it must be above o to be a valid separating line.

To keep the algorithm in Figure 5 simple, some details about other special cases have been left out, for instance, additional visibility tests between vertex A and occluders' bottom left corners, B and bottom right corners, M_{DA} and top left corners, M_{BC} and top right corners, A and M_{BC}, and B and M_{DA}. Nevertheless, these tests do not affect the complexity of the algorithm. These configurations are easily handled if A is considered as a top right corner, B as a top left corner, M_{DA} as a bottom right corner, and M_{BC} as a bottom left corner.

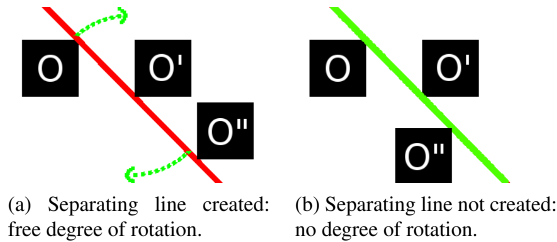
Other visibility tests are needed between A and M_{DA}, and between B and M_{BC}. This is handled by testing if there is any occluder on the borders of the tile between A and M_{DA} or just below M_{DA}, and between B and M_{BC} or just below M_{BC}. Figure 2b shows a border separating line in blue.

3.2.3 Full Resolution Generation

An exhaustive computation of all occluders configurations quickly becomes impossible with an $O(n^3)$ complexity, since even a small

$M_{DA} := middle_of([DA])$;
$M_{BC} := middle_of([BC])$;
Procedure is_occluded
{
for all occluder $o \in ABCD$ **do**
  for all occluder o', $o' \neq o$ **do**
    d_1 = (top_left_corner(o), bottom_right_corner(o'));
    d_2 = (top_right_corner(o), bottom_left_corner(o'));
    {*// if* $d_1 \cap o \mid d_1 \cap o'$, d_1 *is not treated*}
    if d_1 intersects $\bar{u}$-shape **then**
      d_1_blocked = false;
      for all occluder o'', $o'' \neq o$ & $o'' \neq o'$ **do**
        if $d_1 \cap o''$ **then**
          d_1_blocked = true;
          break;
        end if
      end for
      if not d_1_blocked **then**
        return false;
      end if
    end if
    {*// test* d_2 *the same way as* d_1}
    ...
  end for
end for
return true; {// the tile is occluding, cf. Figure 7a.}
}
{Note that for each occluder, only "free" corners are processed (i.e., not shared by another occluder). If all its corners are shared with other occluders, this occluder is skipped.
Also note that paired separating lines between two occluders (as in Figure 2b) are computed separately, and do not result from the same (o,o') combination. }

Figure 5: Procedure for occlusion testing for the North/South direction.



(a) Separating line created: free degree of rotation.
(b) Separating line not created: no degree of rotation.

Figure 6: A particular construction case of separating lines: if a separating line can be created, a hole exists and the tile is not occluding.

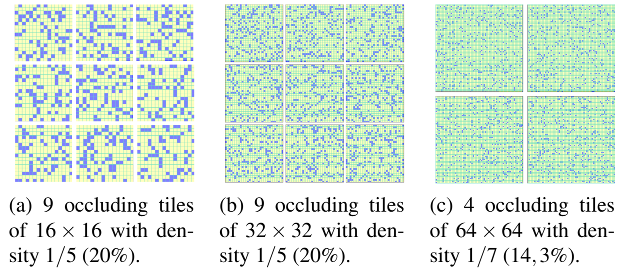
8×8 resolution produces 2^{64} tiles to test. Instead we rely on a stochastic generation of occluders, with a probability assigned to all pixels of a tile (leading to a density of occluders). With a random generator covering the full sequence of binary values (empty or occluded) associated with a configuration, we can investigate different and uncorrolated configurations, leading to more tiles with more variations than with a sequential search.

Some of the results from Table 1 appear in Figure 7. Each test series has been computed in about 90 minutes on an Intel Core 2 Duo CPU running at 2.1 GHz.

One can observe that at a given resolution for the same processing time, increasing occluder density reduces the number of tested tiles, but more occluding tiles are found. Also, for a given occluder density, the number of tiles tested decreases rapidly as tile resolution increases, but more occluding tiles are found. Setting the desired occluder density is therefore key to finding occluding tiles: a density too low will produce too few tiles, and a density too high will lead to a longer time to test tiles.

Resolution	Occluder density	Tiles found / tested
16×16	1/4 (25%)	2,960 / 260,725
16×16	1/5 (20%)	209 / 1,064,670
16×16	1/6 (16,6%)	5 / 2,482,720
16×16	1/7 (14,3%)	0 / 4,399,280
32×32	1/4 (25%)	329 / 801
32×32	1/5 (20%)	112 / 3,710
32×32	1/6 (16,6%)	9 / 19,608
32×32	1/7 (14,3%)	0 / 62,224
64×64	1/4 (25%)	8 / 8
64×64	1/5 (20%)	10 / 10
64×64	1/6 (16,6%)	10 / 20
64×64	1/7 (14,3%)	4 / 59
64×64	1/8 (12,5%)	2 / 244
64×64	1/9 (11,1%)	0 / 909

Table 1: Occluder density and number of tiles found.



(a) 9 occluding tiles of 16×16 with density $1/5$ (20%).
(b) 9 occluding tiles of 32×32 with density $1/5$ (20%).
(c) 4 occluding tiles of 64×64 with density $1/7$ $(14,3\%)$.

Figure 7: A few computed occluding tiles.

Our method gives good results for resolutions lower than 32×32 and occluder density around 20%. For tiles with large resolutions, computing constrained visibility becomes too expensive. While an incremental processing of occluders or a hierarchical classification of occluders would improve the algorithm's complexity, it would not increase the potential for a tile to be an occluding tile.

3.2.4 Multi-resolution Generation

With the aim of reducing generation time, we propose a complementary method to guide stochastic search with a multi-resolution approach.

The four steps to generate derived tiles are:

1. **Generate** a random tile of an $n \times n$ resolution that satisfies occlusion in the four directions.

2. **Double the resolution** ($2n \times 2n$) of the tile; a pixel becomes a set of 2×2 pixels with the same definition.

3. **Mutate** pixels in the $2n \times 2n$ resolution by randomly moving occluders according to a correlation coefficient (set by a user, locally or globally). This correlation coefficient is proportional to the tile size. Each new configuration is then tested for occlusion.

4. **Repeat** steps 2 and 3 with the resulting mutated tiles that passed the occlusion test until the desired resolution is reached.

Figure 8 shows some results for this method. Here we set to 100 the maximum number of mutations to operate on a lower resolution tile before rejecting it, thus allowing for less similar higher resolution tiles. To assess the efficiency of this optimization, an 8×8 occluding tile is found in about one second, and 100 16×16 tiles

are derived from it in less than 36 seconds for a 20% occlusion density, whereas the original method finds 200 occluding tiles in 90 minutes. These operations guide effectively the search of solutions, approximately multiplying by 10 the number of tiles found and dividing by 10 the search time.

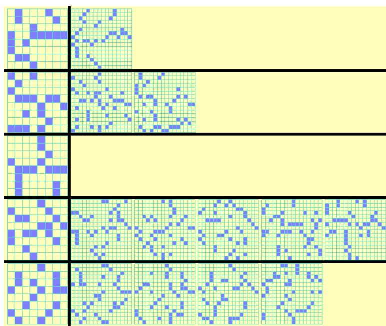


Figure 8: Five 8×8 occluding tiles with derived 16×16 occluding tiles. One 8×8 tile did not result in any valid derived tile.

3.3 Customizing Tiles

Completely randomly generated tiles might have some applications, but this is fairly limited. In order to provide a better controlled generation usable by artists, we use a painting system to set probabilities associated with pixels in specific tiles. In Figure 9a, computed in 14 seconds, the painted image is divided in a set of 7×7 occluding tiles, each of resolution 8×8. White to black pixels are mapped to pixel occluder probabilities linearly scaled from 20% to 100%. The same painted image is used in Figure 9c to generate tiles of resolution 32×32. It is computed in 1765 seconds, with an occluding density linearly scaled from 16,6% to 100%.

4 Application: Extruded Buildings in a 3D City

We used our method to create a city, shown in Figure 10. Each occluding pixel in a tile is instantiated as a building (i.e., a 3D textured box), and visibility is limited within the first ring of tiling, provided that the camera is located below the height of the smallest building. With our method, no PVS is needed for this scene, and all polygons located beyond the first ring of tiling can simply be neglected. Therefore, only the polygons in the current view cell (in blue) and in the first ring of tiles (in red) need to be rendered.

In order to give a more realistic aspect to our cities, additional constraints must be handled. Amongst those, the observer should be able to navigate freely on tiles, and buildings should exhibit different heights.

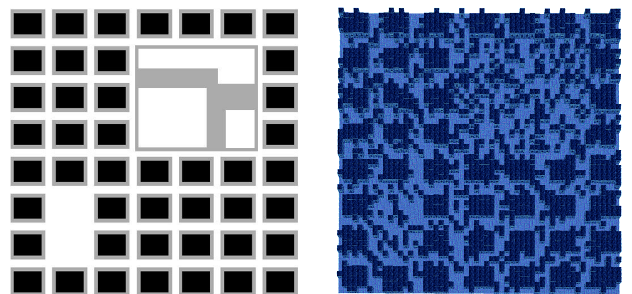
4.1 Navigation

To provide navigation within and between tiles, two constraints are added. First, we ensure that a path exists between opposite sides of a tile, i.e., from *North* to *South*, and from *West* to *East*. Second, we enforce that a tile cannot be added to the set if a border shared by two tiles cannot be traversed. This is expressed as a form of Wang tile neighborhood constraint. Other constraints or properties for certain cities or road systems can easily be added to the above constraints. However, if they severely limit the number of eligible tiles, these constraints should rather be integrated in the rules themselves that generate occluding pixels for increased efficiency.

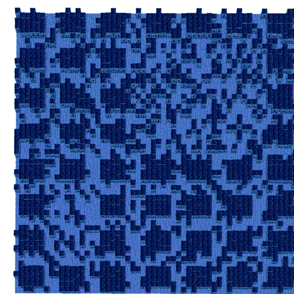
The cities extruded in 3D in Figures 1 and 10 respect these constraints.

4.2 Building Heights

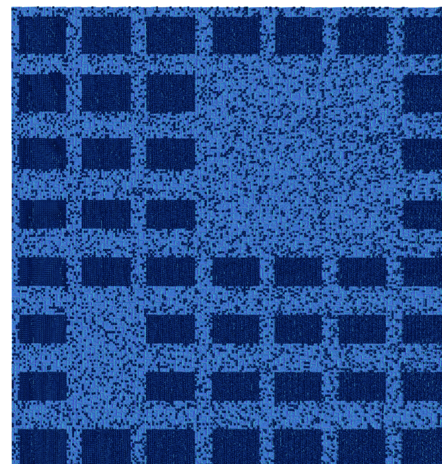
Because occlusion is computed in 2D, building heights are not considered during the tile generation process. However, if the observer navigates at a given maximum height from the ground, variations in building heights are possible within a defined threshold, while still ensuring occluded polygons beyond the first ring from the view cell remain invisible. The formulation for this threshold follows, with the concepts illustrated in Figure 11.



(a) Painted source image.



(b) Resulting 7×7 occluding tiles, each of resolution 8×8.



(c) Resulting 7×7 occluding tiles, each of resolution 32×32.

Figure 9: Customizing the distribution of occluders. Each tile pixel is generated with a probability linearly derived from the source image.

Assume a tile of $n \times n$ pixels, with each pixel of diagonal length D (the tile has thus a diagonal length of nD). Assume also that a building fits perfectly on one pixel. An observer at maximum height H_{obs} moves on a tile, i.e., the view cell. In the worst case, the observer is located at a corner of the tile, the building of smallest height H_{min} is in the opposite corner of the tile, and we want to place a building of height H at pixel $i \in [0, \lceil n/2 \rceil - 1]$ on the most distant tile (belonging to the first-ring neighborhood), but ensure it would be hidden by the smallest building. Note that the limit on i of $\lceil n/2 \rceil - 1$ is due to the fact that the building maximum allowed height is in the central pixel of the tile, and this maximum height diminishes symmetrically until reaching the border of the tile.

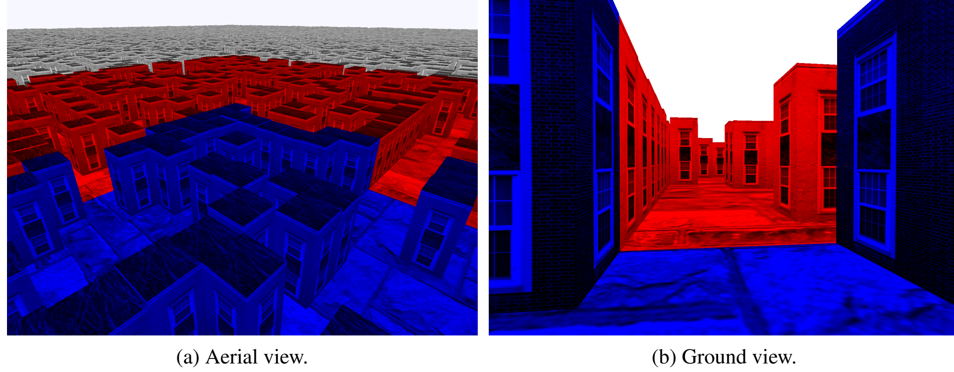
Let $h_{min} = H_{min} - H_{obs}$ and $h = H - H_{obs}$. We have

$$\frac{h_{min}}{(n-1)D} = \frac{h}{nD + iD}$$

which simplifies to

$$h = h_{min} \times \left(\frac{n+i}{n-1} \right). \quad (1)$$

Therefore, any building of height lower than $h + H_{obs}$ will be occluded by the smallest building.



(a) Aerial view. (b) Ground view.

Figure 10: The view cell in blue is surrounded by a first-ring neighborhood of tiles in red. The second ring and further away rings, drawn in grey, are invisible from the view cell if the observer is located between the ground and the top of the smallest building. Here, 10 $\times$ 10 occluding tiles of resolution 16 $\times$ 16 are derived from a single 8 $\times$ 8 tile with a 20% occluder density and a permutation factor of 256. The two connexity constraints are taken into account. The tiles were computed in 36 seconds and the tiling in 1 second.

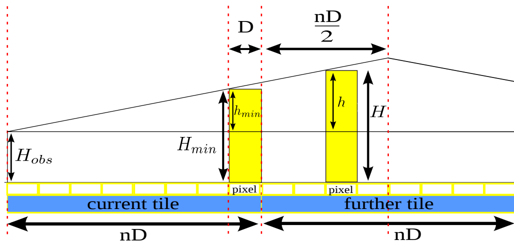
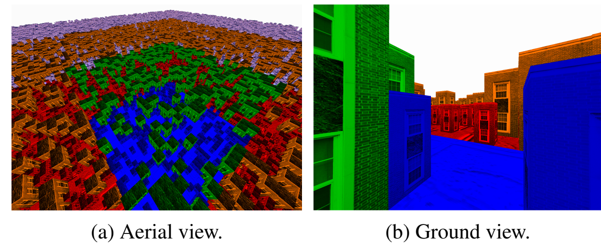


Figure 11: Variations on building height are related to the observer's height and the tiles' size, the latter expressed in number of pixels.



(a) Aerial view. (b) Ground view.

Figure 12: Combining multi-scales in a city: The first level view cell is in blue, the second level in green. The first ring for first level is in red, and for the second level, in orange.

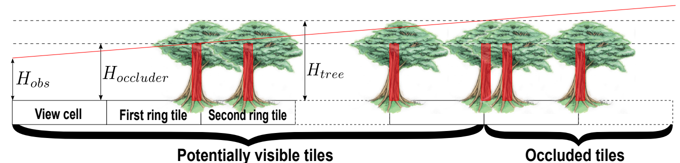


Figure 13: Occlusions are based on an internal square at the base of the tree. We can then compute the maximal extent of visible elements.

4.3 Multi-tiling

Another way to provide more flexibility on the variations of heights for the occluders is to combine tiles of different resolutions. This consists of mixing in the same physical space different grids, while avoiding to have occluders intersect each other.

Four tiles of resolution $n \times n$ fit in one tile of resolution $2n \times 2n$. Our observation is based on the fact that if we keep the same aspect ratio (building height / pixel size), the higher resolution (smaller) pixels provide closer full occlusion for thinner-shorter buildings, while the lower resolution pixels provide full occlusion further away for larger-higher buildings. This is illustrated in Figure 12, for a city made of two different resolutions of tilings. The first level, as in Figure 10, has its view cell in blue and its first ring of tiles in red. The second level has its view cell in green and its first ring in orange. Because in the special case of Figure 10, smaller buildings are completely enclosed by or completely outside of the larger buildings, during rendering, buildings connected to smaller pixels are simply not displayed if they are located in occupied larger pixels.

5 Conclusions

We have presented a technique to generate tiles satisfying occlusion properties in order to respect a bound on the maximum number of objects visible from anywhere on an associated tiling.

The tiles are simple squares subdivided into a grid of occluders. This simple representation proved very efficient for occlusion testing, multi-resolution generation, storage, and multiple constraint satisfaction. With our $\bar{u}$ occlusion testing scheme in the four canonical directions, each generated tile can be used independently from its neighboring tiles, and a simple first ring of occluding tiles bounds the visibility.

Although visibility is computed in 2D, it remains conservative for extruded objects from the occluding tile pixels. We demonstrated its potential with the generation of cities. *Occlusion Tiling* can be applied to other occluder instanciations, such as tree trunks for a forest scene. In Figure 13, we show that the maximal extent of the visible trees for a forest can be computed if we consider as opaque the internal region within tree trunks.

Our algorithm can be integrated in a tool so that artists can easily control tile generation. Simple painted probabilities can provide controls over the general aspects of the tiles. Our method can also be used to optimize object locations or to suggest object displacements in order to increase occlusion. We believe we only scratched the surface of its potential to assist artists modeling complex worlds, while ensuring their efficient rendering and representation.

6 Future Work

While our current prototype shows some potential for real games, we still need to improve generality for practical, more diverse game environments. Extending visibility further than the first ring of tiles could provide more flexibility in generated worlds, at an increased cost of larger potentially visible sets of polygons. On-demand tile generation could also be interesting for worlds with stronger full occlusions.

In the short term, we will improve even more on the efficiency of generating occluding tiles satisfying multiple properties. Our current occlusion computation is mostly brute-force, and improvements are still expected.

Another direction of research will consider how our tiles could be adapted along certain directions of more open visibility, for instance, to simulate straight roads in a city, or large parks. This could be handled by studying how visibility propagates over tile combinations, similarly to horizon culling algorithms.

In our representation, tiles are defined as squares subdivided into grid cells. This structure proved simple, efficient, and general. However, we would like to study other tile shapes and subdivision patterns, thus breaking away from possible grid-like artifacts.

We would also like to investigate how layers of occluding tiles, or even 3D tiles could release some inherent limitations of our current 2D visibility representation.

Acknowledgements

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

References

[1] T. Aila and V. Miettinen. dPVS: An Occlusion Culling System for Massive Dynamic Environments. *IEEE Computer Graphics and Applications*, 24(2):86–97, Mar. 2004.

[2] 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*, pages 41–50, 258, 1990.

[3] R. Berger. *The Undecidability of the Domino Problem*. Memoirs of the Amer. Math. Soc. (66). American Mathematical Society, 1966.

[4] M. F. Cohen, J. Shade, S. Hiller, and O. Deussen. Wang Tiles for Image and Texture Generation. In *Proc. SIGGRAPH '03*, pages 287–294, 2003.

[5] 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*, 9(3):412–431, 2003.

[6] 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):243–253, Aug. 1998.

[7] S. Coorg and S. Teller. Temporally Coherent Conservative Visibility. *Comput. Geom. Theory Appl.*, 12:105–124, Feb. 1999.

[8] L. Downs, T. Möller, and C. H. Séquin. Occlusion Horizons for Driving Through Urban Scenery. In *Proc. Symposium on Interactive 3D Graphics*, I3D '01, pages 121–124, 2001.

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

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

[11] 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.

[12] B. Lloyd and P. Egbert. Horizon Occlusion Culling for Real-time Rendering of Hierarchical Terrains. In *IEEE Visualization*, 2002.

[13] S. Nirenstein, E. Blake, and J. Gain. Exact From-Region Visibility Culling. In *Proc. Eurographics Workshop on Rendering*, pages 191–202, 2002.

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

[15] J. Stam. Aperiodic Texture Mapping. Technical Report R046, European Research Consortium for Informatics and Mathematics (ERCIM), 1997.

[16] S. J. Teller. Visibility Computations in Densely Occluded Polyhedral Environments. Technical Report CSD-92-708, EECS Department, University of California, Berkeley, 1992.

[17] H. Wang. Proving theorems by pattern recognition I. *Commununications of the ACM*, 3:220–234, Apr. 1960.

[18] L.-Y. Wei, S. Lefebvre, V. Kwatra, and G. Turk. State of the Art in Example-based Texture Synthesis. In *Eurographics '09, State of the Art Report (EG-STAR)*, 2009.

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