Université de Montréal

Real-time Rendering of Cities at Night

par
Melino Conte

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Novembre, 2018

© Melino Conte, 2018.

RÉSUMÉ

En synthèse d'images, déterminer la couleur d'une surface au pixel d'une image doit considérer toutes les sources de lumière de la scène pour évaluer leur contribution lumineuse sur la surface en question. Cette évaluation de la visibilité et en l'occurrence de la radiance incidente des sources de lumière est très coûteuse. Elle n'est généralement pas traitée pour chaque source de lumière en rendu temps-réel. Une ville en pleine nuit est un exemple de telle scène comportant une grande quantité de sources de lumière pour lesquelles les rendus temps-réel modernes ne peuvent pas évaluer la visibilité de toutes les sources de lumière individuelles.

Nous présentons une technique exploitant la cohérence spatiale des villes et la cohérence temporelle des rendus temps-réel pour accélérer le calcul de la visibilité des sources de lumière. Notre technique de visibilité profite des bloqueurs naturels et prédominants de la ville pour rapidement réduire la liste de sources de lumière à évaluer et ainsi, accélérer le calcul de la visibilité en assumant des bloqueurs sous forme de boîtes alignées majoritairement selon certains axes dominants. Pour garantir la propagation des occultations, nous fusionnons les bloqueurs adjacents dans un seul et même bloqueur conservateur en termes d'occultations.

Notre technique relie la visibilité de la caméra avec la visibilité des surfaces pour réduire le nombre d'évaluations à effectuer à chaque rendu, et ne calcule la visibilité que pour les surfaces visibles du point de vue de la caméra. Finalement, nous intégrons la technique de visibilité avec une technique de rendu réaliste, *Lightcuts* [26], qui a été mise à jour sur GPU dans un scénario de rendu temps-réel.

Même si notre technique ne permettra pas d'atteindre le temps-réel en général dans une scène complexe, elle réduit suffisamment les contraintes pour espérer y arriver un jour.

Mots clés: Visibilité, rendu temps-réel, villes, CHC, *Lightcuts*, GPU

ABSTRACT

In image synthesis, to determine the final color of a surface at a specific image pixel, we must consider all potential light sources and evaluate if they contribute to the illumination. Since such evaluation is slow, real-time renderers traditionally do not evaluate each light source, and instead preemptively choose locally important light sources for which to evaluate visibility. A city at night is such a scene containing many light sources for which modern real-time renderers cannot allow themselves to evaluate every light source at every frame.

We present a technique exploiting spatial coherency in cities and temporal coherency of real-time walkthroughs to reduce visibility evaluations in such scenes. Our technique uses the natural and predominant occluders of a city to efficiently reduce the number of light sources to evaluate. To further accelerate the evaluation we project the bounding boxes of buildings instead of their detailed model (these boxes should be oriented mostly along a few directions), and fuse adjacent occluders on an occlusion plane to form larger conservative occluders.

Our technique also integrates results from camera visibility to further reduce the number of visibility evaluations executed per frame, and evaluates visible light sources for facades visible from the point of view of the camera. Finally, we integrate an offline rendering technique, Lightcuts [26], by adapting it to real-time GPU rendering to further save on rendering time.

Even though our technique does not achieve real-time frame rates in a complex scene, it reduces the complexity of the problem enough so that we can hope to achieve such frame rates one day.

Keywords: Visibility, real-time rendering, city, CHC, Lightcuts, GPU

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

This thesis concludes my Master's research at LIGUM (Laboratoire d'informatique graphique de l'Université de Montréal). I can say with confidence that I come out of this research with infinitely more knowledge and skills in my field, all thanks to the help of my teachers and colleagues in the lab.

More specifically, I would like to thank my family, whom I love, Ermenegildo Conte, Gildo Conte, Dominic Homo, Louise Guay, Marie-Claude Langlois, and Amélie Jubert, who encouraged and supported me through my progress. From the start of my undergraduate studies to the completion of my Master's degree, I knew I could count on their support to further my education. Your support has made it possible for me to achieve what I have today. A special thanks to Louise and Amélie, who both helped me during the final stages of writing this thesis and to my mom, Marie-Claude, who supported me financially through all these years.

For being always available to provide technical help in the lab, discussing our research, or simply being there as friends, I would like to thank Adrien Dubouchet, Luìs Gamboa, Jean-Philippe Guertin, Étienne Léger, Olivier Mercier, Joël Polard-Perron, Bruno Roy, Arnaud Schoentgen, Jonas Zehnder, and Yangyang Zhao. Furthermore, I would also like to mention my longtime friends, Valérie Bourdeau, Jessica Brière, Maël Costantini, Éléonore Haberer, Emmanuel Iannacci, Alexandre Jubert, and Maxime Marquis, who have supported and believed in me throughout my years at university.

Finally, I want to thank the members of my evaluation committee, Nadia El Mabrouk, Bernhard Thomaszewski, and my Master's director, Pierre Poulin. It is with Pierre that I built this research. I especially thank him for his support, patience and guidance, and feel confident that together, we achieved meaningful work. Thank you again for your thorough corrections, honest advice, and generous recommendations throughout the years. I hope we will pursue our meaningful conversations about computer graphics and Scotch in the years to come.

Thanks Alex for introducing me to computer graphics. You shared your passion with me, and I am grateful to you that I get to work in an area I love, every day. We did it!

CHAPTER 1

INTRODUCTION

1.1 Motivation

When we look at big-budget video games over the last twenty years, we can clearly see the progression in image quality that the industry has achieved. There are many factors that enabled such a visual evolution, and we will take a look at a few of them to compare how they have evolved differently during that period of time.

Geometrical complexity in video game scenes has exploded over the last years. Not only have graphic cards become faster, but they have also become much more adapted to treating large quantities of triangles in less than 30ms (real time). With such power, we now render immense scenes, with stronger occlusion techniques to efficiently determine what we actually see. We even add complexity to certain surfaces through on-the-fly tessellation.

Surface detail is another area where we have seen much progress. We have evolved from static single-color surfaces to textured surfaces, and increasingly detailed rendering algorithms. The rendering quality in video games has evolved to the point that only a trained eye can make the difference between a real digital photo of a wood table or a real-time rendering of one. We have transitioned recently to physically-based materials, giving the games' surfaces, like textiles, properties observed in real-life objects.

Light sources inject depth and complexity in our images from a different standpoint than geometry. Lighting, geometry and material combine to produce an image that we recognize and compare to real-life experiences. Sometimes a single light source is enough to achieve realism, such as the sun in a clear day. Other times, such as in a city at night, we need many more light sources to achieve realism. Whereas previous lighting

and geometry achievements had evolved a lot, the scale reached by the number of light sources is much more limited. In recent video games, such as DOOM (2016), we can find ourselves with a maximum of 256 light sources to shade a single pixel [10]. Such a limit is much too restrictive for a city scene that can be composed of hundreds of thousands of light sources. The reason behind this slower evolution is the inherent impact that light sources have on time spent to evaluate a surface's color. With the most naive algorithm, we would need to sum the incident radiance from all light sources to compute the pixel color, bringing up the complexity from an n frame time to an $n \times m$ frame time for a scene with m light sources.

Such a limit is the motivation behind our research. However, building a pipeline that scales well with many light sources is a daunting task, which comes with many challenges. This can explain why fewer researchers have taken up these challenges. We will present in this document our solution and the simplifications that we introduced to achieve our rendering times.

1.2 Rendering a City at Night

As introduced in the previous section, the geometric complexity and number of light sources in a scene have a multiplicative impact on rendering time. One of the most difficult scenes that we can imagine with that respect is a city at night. Such a scene is composed of many buildings, each building containing many exterior and interior light sources, on top of all street lampposts, traffic lights, lit advertising signs, car headlights and taillights, mirror-like reflective windows, etc. Figure 1.1 shows a photograph of London at night which presents even more different light sources coupled with architectural complexity, such as circular buildings.

In addition to complexity, the viewpoint may change much of the visible portion of the city. For example, at street level we mostly see buildings along the street and taller buildings from neighboring streets. However, if we are flying over the city, the scale of what is visible changes completely, leading to close to visibility of half of the entire scene (the building self-occludes two to three of its facades/top) of all buildings. Further

Figure 1.1 – A photograph of London at night showing many different light sources and unique architectural designs. Image from Pexel [2].

in this document, we present and analyze multiple viewpoints to study scaling for our developed algorithm.

A key aspect of our research is that although the camera viewpoint changes the visible geometry rendered, the camera position does not affect the light sources that shine directly on a building facade. Similarly, light that directly reflects specularly on a surface is constrained by the city occlusions and reflective cone, assuming a cone-like specular lobe. [1]Such properties let us devise a specific visibility technique taking advantage of the 2.5D nature of city buildings while letting the camera move freely in 3D.

1.3 Contributions

To tackle the complexity of rendering a city at night we have developed a pipeline that evaluates camera visibility, identifies the relevant light sources for the visible objects, and finally renders them with the incident illumination from their respective light sources at appropriate cluster levels. The pipeline can be subdivided in three distinct steps, described in the following sections.

1. Although we do not study the case of specular reflections in this thesis, the fundamentals of our technique remain valid for such a case and are further discussed in the potential future work.

1.3.1 Coherent Hierarchical Culling (CHC)

CHC [5] is a CPU-based occlusion technique used to determine, through GPU occlusion queries, objects visible to the camera. CHC is integrated in our pipeline to achieve the first culling of geometry. CHC receives the bounding volume hierarchy (BVH) of our 3D scene along with details about the current camera settings, and then evaluates recursively which nodes in the BVH are potentially visible. While calculating visibility of each node, the technique evaluates an estimate of how many of the pixels the node covers are visible in the final image, and stores this information into the BVH tree. We stop the traversal of the BVH when an analyzed node is not visible or covers fewer pixels than a user-defined visibility threshold.

1.3.2 Facade-Cluster Visibility (FCV)

FCV is our developed visibility technique that evaluates which light sources an object can see in the city. Our technique builds view cells by traversing the BVH nodes from the root until the node's potentially covered pixels is greater than a user-defined visibility threshold. Then, for each view cell, we compute the visibility, according to each cardinal axis, of our precomputed light tree, and assign the visibility result to every object in it.

1.3.3 GPU-based Lightcuts

We adapted the Lightcuts [26] algorithm to a real-time scenario, encoded as a GPU fragment shader. Lightcuts is a many-lights technique that estimates the incident radiance of all the light sources in the scene by traversing its light tree, a binary tree built by clustering together similar light sources until a single node is left. Since GPU shaders are not well adapted to recursion, central in traversing the light tree in the Lightcuts algorithm, we modify the structure of the light tree to linearize its evaluation. Finally we make additional changes in the traversal and clustering functions to improve the speed of the algorithm.

1.4 Outline

In the document we present next the state-of-the-art techniques that tackle challenges or similar problems that we face in our technique. These techniques helped inspire us to develop our method, and are the foundation of this thesis. Then, we detail each step of our algorithm, more precisely our Lightcuts GPU adaptation and our Facade-Cluster visibility technique, and discuss our achieved results. Finally, we present interesting potential future work and conclude the thesis.

CHAPTER 2

STATE OF THE ART

In this chapter we introduce three families of techniques, i.e., visibility, real-time rendering, and many-lights. Each one covers a distinct step in our developed pipeline; it is followed by a summary of advantages and disadvantages of their most notable techniques.

2.1 General Concepts

Frustum culling is the use of the camera's view frustum to cull geometry. When simulating a camera in a rendering engine, the field of view of the camera defines a region of space that may appear in the rendered image. The region is called the view frustum. Typically, cameras in 3D space have a pyramidal frustum which can be defined by six planes. Frustum culling tests if the geometry we want to draw in the image falls inside these six planes. As can be seen in Figure 2.1, geometry fully outside the planes is culled since it cannot be seen by the camera.

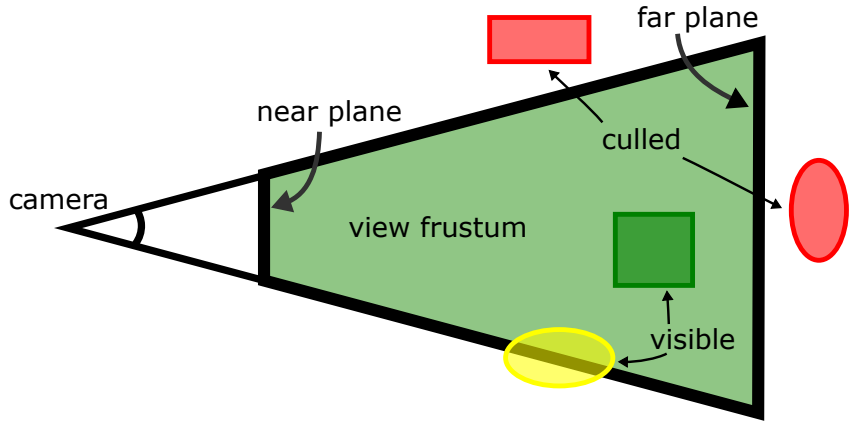


Figure 2.1 – A 2D slice of a camera and its view frustum. Geometry fully outside of the frustum, red in the figure, is culled since it will not be directly visible from the point of view of the camera. Geometry inside or partially inside the view frustum, green and yellow in the figure, is not culled since it could be visible in the rendered image.

A bounding box (BB) is a box defined to enclose an object in a scene. The box being a simple object to define let us conservatively test the visibility of the enclosed object without having to evaluate its potentially complex structure or shape. In this thesis, we only use axis-aligned bounding boxes (AABB) to simplify the computation of our facade-cluster visibility.

A bounding volume hierarchy (BVH), is a binary tree structure over a set of geometric objects in the scene. Every leaf in the BVH represents an object, and the root contains all the objects. The BVH is constructed by repeatedly building and grouping the smallest bounding volume (surface area of a box rather than actual volume) with two nodes. In our technique, the structure is used to quickly traverse the hierarchy of the scene objects to compute visibility. For example, when testing the visibility of a light source, if a BVH node does not occlude the light source, we are guaranteed that the node's children will not either.

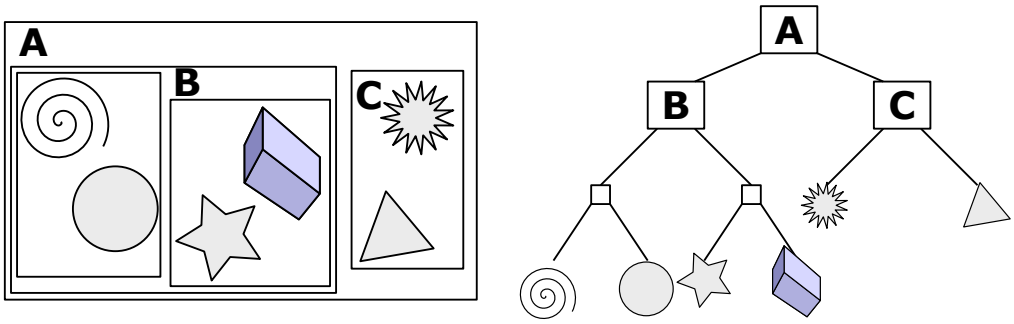


Figure 2.2 – An example of a bounding volume hierarchy. The root of the tree contains all objects, and each leaf represents a single object. Figure from Wikipedia [1].

A mipmap is a precomputed sequence of the same image, each smaller than the previous one by a power of two. This mipmapping process is used to increase performance when sampling texture images while rendering. Nearby and large objects can sample higher resolution images, while far away and small objects, for which a pixel in the image can cover many texels (pixels in the texture image) can sample lower resolution images. This provides a more efficient way of down-filtering than computing the average value by sampling all the texels each pixel covers. As can be seen in Figure 2.3, the cost of a mipmap is a $\frac{1}{3}$ increase in texture memory size.

Figure 2.3 – The mipmap of an image is shown. By separating the red, green and blue channels of the image, we can easily notice the $\frac{1}{3}$ increase in memory size. Image from Wikipedia [3].

Another significant concept for this thesis is that the communication between CPU and GPU hardware is asynchronous. To tell the GPU what to execute, the CPU needs to send it commands and data. When it receives those commands, the GPU executes them in the order they were sent. The *present* command displays the content of an image buffer to the screen, and a frame is the time duration in-between two *present* commands. Since the CPU and GPU are normally not synchronized, the CPU often starts its new frame while the GPU is still rendering the previous one.

In a best-case scenario, the CPU continually sends commands to the GPU before the GPU runs out of commands to execute. This keeps the CPU and the GPU fully occupied. However, depending on the computation duration of the commands sent by the CPU and the speed at which they are sent to the GPU, there are two other possible scenarios. First, if the GPU runs out of commands to execute, it will go idle in a state we call GPU starvation, waiting for the next command. Second, if the CPU needs to fetch the results of a command on the GPU, it must wait until the command has been executed on GPU. Then, when both the CPU and the GPU are synchronized, the CPU can read the results. This synchronization process often results in GPU starvation as the CPU may not be able to issue other commands while waiting for these results.

A hardware occlusion query (HOQ), NVIDIA's occlusion query in particular, is a

GPU visibility query made available by GPU manufacturers to test if an element sent to the GPU would be visible at the current stage of the rendered image. To use such a query, the user sends the mesh to test to the GPU and sets it to execute a HOQ. When the GPU processes the mesh, it rasterizes it and records how many pixels pass (succeed, i.e., are determined visible) the depth test. The HOQ returns this number (the number of pixels considered visible) but the query does not change the depth buffer and does not execute the fragment shader (which calculates and modifies the color of the pixel). On the CPU, to recover this number of pixels, we need to check the state of the query and wait until the GPU sets the query's flag to the finished state. This waiting period on the CPU is one of the major drawbacks of HOQ, as waiting on the CPU for the result of a test before sending the next task can lead to the GPU pipeline being delayed as it also has to wait for the CPU. This drawback can cost many milliseconds over the course of a frame in real-time rendering pipelines where time is vital. An example of such CPU stall, shown in Figure 2.4, is presented by Bittner et al. [5] with their solution.

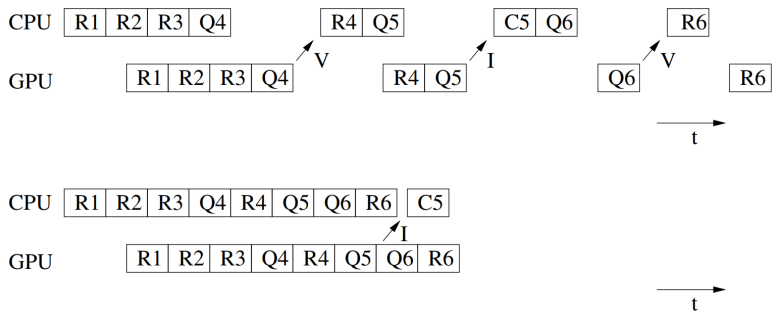


Figure 2.4 – Q_i, R_i, and C_i respectively represent querying, rendering, and culling an object i. An example of CPU stall and GPU starvation (top) followed by a solution where objects 4 and 6 are assumed to be visible (V) and are rendered without waiting for their query results, while object 5 is evaluated as invisible (I) and is culled (bottom). Although the results for both objects are retrieved too late to be useful in the current frame, they are saved and used in the next frame. Figure from Bittner et al. [5].

To work around this CPU stall problem, we can send multiple queries at the same time if they are not dependent on previous results. We can also group together multiple queries by sending their meshes together in the same query. Sending multiple independent queries keeps the GPU working on the next queries while the CPU analyzes the

results of a previous query, adding if necessary new queries to the GPU. However, keeping multiple queries in the GPU pipeline is not always possible, as we will eventually have to wait for the results of our latest query or dependent queries. Sending grouped queries can help culling larger parts of the scene at once if no fragment is visible, but it can also lead to "wasted" queries if there are visible pixels, as we cannot know from which parts of the mesh visible pixels come from. If this information is needed, we must subdivide the previously sent grouped query into multiple independent queries and test them individually.

2.2 Visibility

Visibility techniques are traditionally used to determine which object is the closest in a given direction from a given point. In our technique we have two different cases where a visibility test is needed. Firstly, we must determine which objects are visible from the camera. Secondly, we must evaluate which light sources shine on visible objects. Cohen-Or et al. [8] compiled a very good survey on visibility techniques, even though it is starting to date by now. We classify algorithms in two categories. Point-based techniques perform visibility computations based on a point of view, while from-region techniques evaluate visibility from an entire view cell. The first visibility evaluation can be performed efficiently by both categories of techniques, while our second evaluation tremendously benefits from from-region techniques, as we would otherwise need to do a point-based visibility evaluation for each pixel.

2.2.1 Point-based Visibility

Point-based techniques evaluate visibility from a single point of view, and are well adapted to process visibility from a pin-hole camera. Traditionally, a depth buffer is a 2D texture assigned in the GPU to store depth of previously rendered objects for the current image. The depth buffer is used pixel by pixel to test whether objects are farther away than those currently stored in it. Although it is used by default in most rendering engines and GPU APIs, the depth buffer achieves no culling if objects are sent in order from the

farthest to the closest, relative to the camera, as each object will appear consequently closer than the previous ones (if we ignore the exception illustrated in Figure 2.5), resulting in wasted depth tests and overdraws. The number of overdraws that could happen in an image is in direct relation to the efficiency of a point-based visibility algorithm, as evaluation of pixel color can be a costly operation.

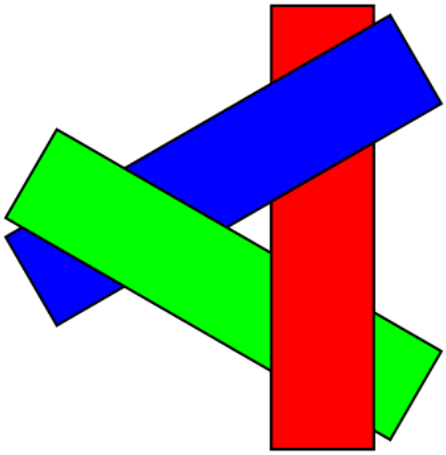


Figure 2.5 – Cyclically overlapping polygons will always occlude one another no matter the order they are being treated. They must be subdivided if one is to skip depth testing. Image from Wikipedia [4].

Coorg and Teller [9] develop a real-time occlusion culling technique that exploits large occluders, which are frequent in urban and architectural models, to quickly and significantly reduce the potentially visible set (PVS). The authors build a visibility oracle that uses runtime table lookups and preprocessing to compute a subset of the supporting and separating planes (see Figure 2.6), formed by an arbitrary occluder and an occludee's axis-aligned bounding box. This subset of planes gives an exact test for full occlusion and a conservative test for partial occlusion. The authors also develop a simple evaluation that detects a subset of nearby large occluders relative to the current viewpoint. Since it would be too costly to evaluate the possible visual interactions, even with a reduced subset of occluders, the authors build a kD-tree with the scene's geometry and instead apply the visibility tests on the tree nodes, traversing the nodes while they are visible. The nodes are kept in cache as long as they are involved in visibility calculation.

By doing so, their technique benefits from temporal coherency when changing the point of view.

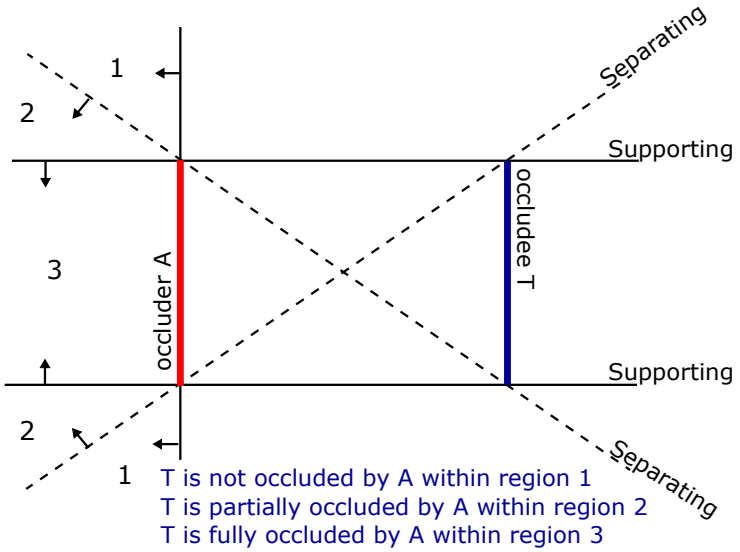


Figure 2.6 – Supporting and separating planes.

Their method performs quick visibility evaluations, but does not support occluder fusion. Such a fusion is vital in presence of many small occluders, as the sum of their projected occlusions can cumulatively hide occludees that were considered visible from their individual projections, as presented in Figure 2.7. The performance of occlusion planes using pre-2000 graphics hardware gives an idea how modern hardware can benefit from such an occlusion method.

Greene et al. [15] build a hierarchy over the depth buffer into a Hierarchical Z-Buffer. Similarly to a mipmap, the technique recursively clusters together 2×2 pixels of the depth buffer. Their algorithm thus builds a depth pyramid that can conservatively cull fragments by choosing the largest value (furthest depth) of 2×2 pixels recursively. To take advantage of the depth pyramid they build an octree hierarchy over the scene's geometry and traverse it from top to bottom, testing each node with the depth pyramid and stopping early the traversal when the node is evaluated as hidden. The technique suffers from the same problems that the basic depth buffer has, notably that sending geometry from back to front order makes the technique useless, and so it must be combined with some ordering of the geometry before sending it to be rendered.

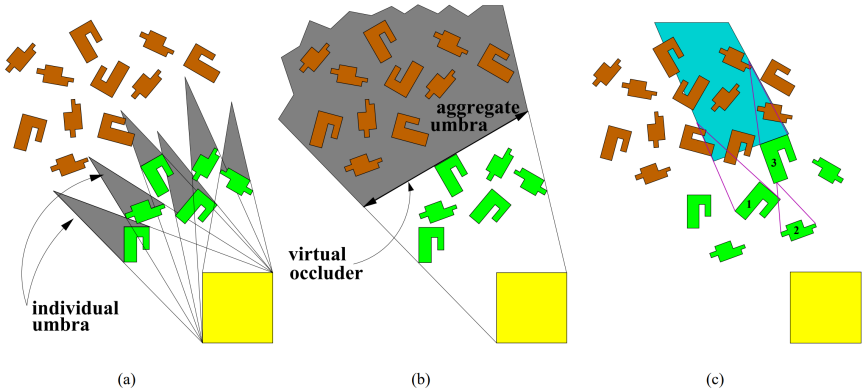


Figure 2.7 – An occluder smaller than a view cell has a vanishing occlusion area that does not fully occlude further than a specific distance. Fusion of occluders represents better the occlusion area but it is much more expensive to evaluate. Figure from Cohen-Or et al. [8].

Bittner et al. [5] introduce the Coherent Hierarchical Culling (CHC) technique. It uses HOQs to determine what is visible from the camera's point of view. To alleviate problems of HOQ, CHC reduces the number of queries sent to the GPU and prioritizes the order of sent queries. Through their pipeline optimizations they achieve real-time rendering and are found to be generally faster than frustum culling evaluated on CPU. The main concepts they implement are presented below.

Grouped queries. CHC builds a bounding volume hierarchy (BVH) over the scene to benefit from multiple concepts in their algorithm. The first usage of the BVH is to query clusters of objects instead of querying each individual object (both clusters and individual objects will be referred to as nodes). As presented before, the concept of grouped queries is beneficial to culling large parts of the scene at once.

Conservative visibility. Instead of querying every node at every frame, CHC assumes that once a node is determined visible, it will remain so for a user-defined number of subsequent frames. This way, only a portion of the visible nodes are queried again at every frame. This is typically valid for walkthroughs where the camera moves forward at a regular speed, like in a video game. To keep queries for visible nodes distributed throughout the frames, CHC adds a random jitter to the frame number where visible nodes must be queried again. In practice, this jitter helps keep the frame rate more sta-

ble. Nodes that were invisible during the last frame are queried during the traversal of the BVH and their results are evaluated in the same frame. However, nodes that were visible during the last frame are queried at the end of the traversal and their results are only evaluated during the next frame.

Reduction of state changes. Changing the GPU from a state where it is rendering objects normally to a state where it is executing queries takes a noteworthy time. Therefore, CHC introduces queues for rendering (r-queue), querying previously invisible nodes (i-queue), and querying previously visible nodes (v-queue). When traversing the BVH, invisible nodes to be tested are sent into the i-queue. When the i-queue reaches a size preset by the user or when the BVH traversal needs the results of the i-queue in order to continue, CHC switches the state of the GPU to querying and sends each query in the i-queue individually. This batching of queries reduces the number of state changes by an order of magnitude.

Estimating coherency of visibility. The visibility history of a node has a strong coherency with its probability to become visible at the next frame. That probability is used to choose previously invisible nodes that should be grouped together in future frames. This idea further reduces queries by creating groups of nodes that are not likely to become visible and for which a grouped query has more potential to be successful.

The technique has been refined by Mattausch et al. [19] as CHC++ to be generally faster. To improve upon CHC, they include the use of queues to batch their queries and draw calls, refine their grouped queries to reduce the number required for previously invisible nodes, and apply a jitter on the number of frames where a node is considered visible without reevaluation to better spread the number of queries sent to each frame.

CHC and CHC++ manage to make use of hardware occlusion queries in a real-time scenario while reducing potential problems that come with them. The integration of temporal coherency into the visibility technique lets it stay conservative while reducing the number of visibility tests performed at each frame, and further reinforces that real-time rendering can benefit from such integration.

Gomez et al. [14] improve the scalability of large virtual worlds by adapting the CHC++ occlusion culling to their tiling scheme. By instantiating the bounding volume

hierarchy (BVH) of tiles during occlusion queries, they remove the necessity for a pre-computed global scene structure. They render the tiles from closest to farthest from the camera, and compute visibility inside a tile by traversing its BVH instance with CHC++.

2.2.2 From-region Visibility

As discussed above, our algorithm must evaluate which light sources shine on surfaces visible from the camera. While both point-based and from-region techniques can find the answer to this question, from-region techniques are generally much faster since the higher cost to evaluate visibility for a region can be spread over multiple frames, as long as the camera stays in the region. A point-based method is generally still necessary as a secondary pass, but on a much smaller set of objects. The two techniques presented here support occluder fusion, which is essential in our algorithm, as our view cells can be larger than occluders. The problem with view cells larger than occluders stems from the fact that the supporting planes between them will necessarily meet behind the occluder, which will render an occluder useless further on (see Figure 2.7). Since we cluster facades and buildings into a single representative facade, and evaluate visibility for it, our algorithm contains many of such exceptions. Therefore, we only look at from-region techniques that support occluder fusion.

Schaufler et al. [24] introduce a volumetric visibility evaluation through the use of voxels to represent occluders and visible space. Their technique discretizes the scene in voxels that are either fully opaque (fully inside an object) or otherwise considered empty. They group together neighboring opaque voxels into effective blockers. They then build a shaft from the view cell to the blockers and extend it behind the blockers. Every voxel is then described as either fully inside, partially inside, or fully outside the shaft. To achieve occluder fusion they not only merge opaque voxels together but also extend blockers into previously fully occluded voxels (see Figure 2.8).

The technique is first presented with a 3D octree and is then transitioned into a 2.5D structure, taking advantage of a simpler representation available for terrains and cities. Their 2.5D representation determines a voxel occluded if it is included in a shaft up to its maximum height, but only considers conservatively the voxel as an occluder up to

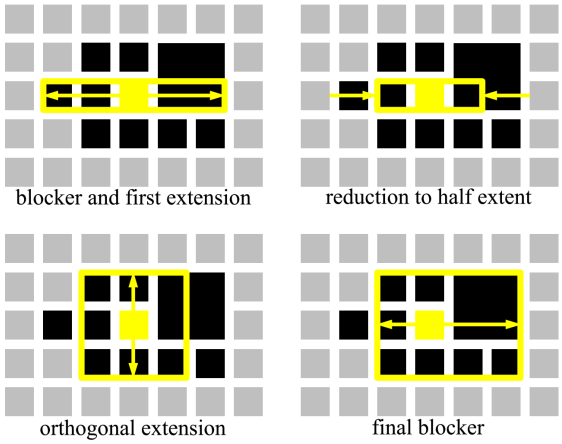


Figure 2.8 – A 2D example of blocker extensions. Figure from Schaufler et al. [24].

its minimum height. The authors also study how to extend blockers while maximizing their subtended solid angle; they have designed the following simple method. Extend the blocker along a first axis, reduce the length on the axis to half, extend along an orthogonal axis, and finally extend again along the first axis. This simple method gives the same results as more complex blocker extension techniques that they tried, since more benefit came from having the occluder fusion in itself compared to optimizing the subtended solid angle.

Finally, they apply their technique to block-based and building-based PVS, and find that the building-based method managed to better reduce the PVS. The technique achieves great performance in visibility tests, but suffers from the memory requirement of its voxel representation of the scene.

Durand et al. [13] present their extended projections to solve visibility for view cells. An extended projection is defined as the projection of an occluder or occludee onto a common projection plane perpendicular to the view direction. The projection is done by intersecting the occlusion area built from the view cell and the occluder to the current projection plane. The occlusion area defined by the interior of the supporting planes behind the occluder is intersected with the projection plane and gives a 2D (in case of 3D visibility) area that represents the area of the projection plane occluded by the occluder from the view cell. A potential occludee is projected in the same way with the

only difference being that the projection plane is located between the view cell and the potential occludee. A potential occludee is then determined potentially visible (or not fully occluded) where its extended projection is not overlapping the extended projections of occluders.

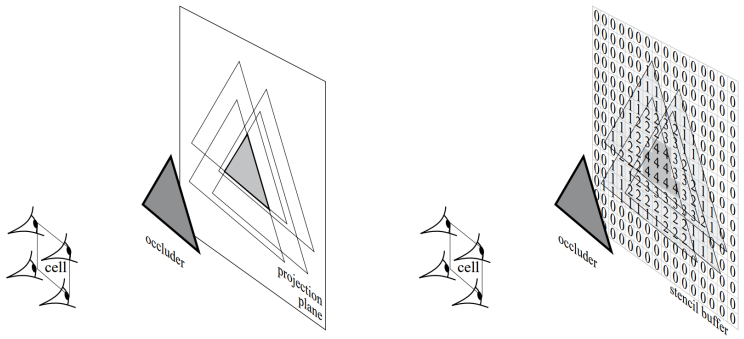


Figure 2.9 – To project an occluder onto a plane from their view cell, Durand et al. [13] compute the intersection of the projections from each corner of the view cell. The figure on the left shows such an example, while the figure on the right shows how they build their Extended Depth Map with rasterization. Image from Durand et al. [13].

To reduce the loss of contributions from occluders smaller than the view cell, the authors propose occluder fusion on the plane, coupled with reprojection of occluders onto further planes. They first discretize the plane into a pixel-based representation called an Extended Depth Map. Figure 2.9 presents how they build the Extended Depth Map by executing projections from the four corners of a flattened view cell. Reprojection of occluders is done by intersecting the occlusion area of their extended projection with a further projection plane, as shown in Figure 2.10b. The authors also show that supporting planes do not handle 3D visibility as well as supporting lines handle 2D visibility. To project occluders for 3D visibility they instead execute twice the 2D visibility and apply a cartesian product between the two. Occluder fusion is achieved since the discretized plane is reprojected onto further planes, grouping together pixels representing occluders.

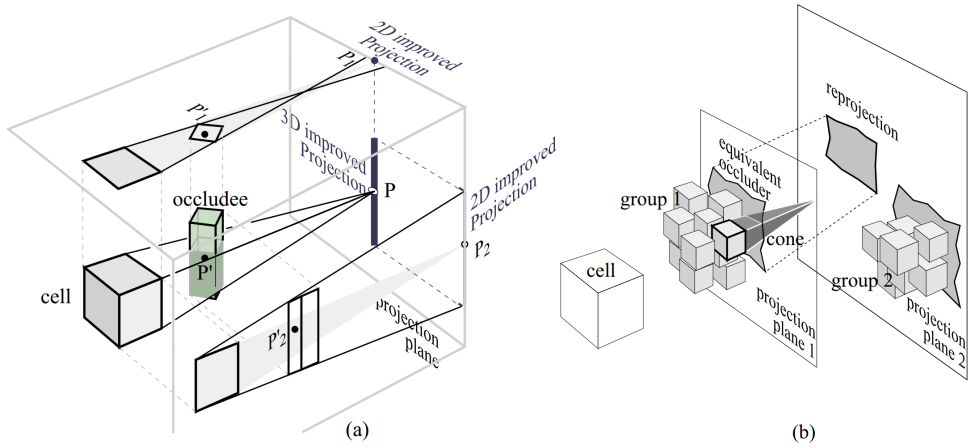


Figure 2.10 – (a) The 3D projection is the cartesian product of two 2D projections. (b) If projection plane 2 is used for re-projection, the occlusion of group 1 of occluders is not taken into account and is instead reprojected from projection plane 1. The occluded-area cone of one cube shows that its projection would be void since it vanishes in front of projection plane 2. Figure from Durand et al. [13].

2.3 Real-time Rendering Techniques

Techniques presented here are frequently used by the video-game industry to accelerate the rendering speed of each frame. These techniques are very specialized and thus, come with downsides.

On the subject of cities at night, Chandler et al. [7] implement a method to procedurally generate window lighting and building interior effects for real-time rendering. Their method is completely implemented in a fragment shader stage which lets it benefit from the parallel computation of the GPU. To generate a building interior they randomly choose a room interior from a list of choices and apply their procedural window lighting to it. They do so by simulating the placement of window curtains and by reducing the intensity of the light source in the room accordingly. Figure 2.11 presents their results when varying the percentage of rooms covered by window curtains and enabling lighting with the presence of interior mapping.

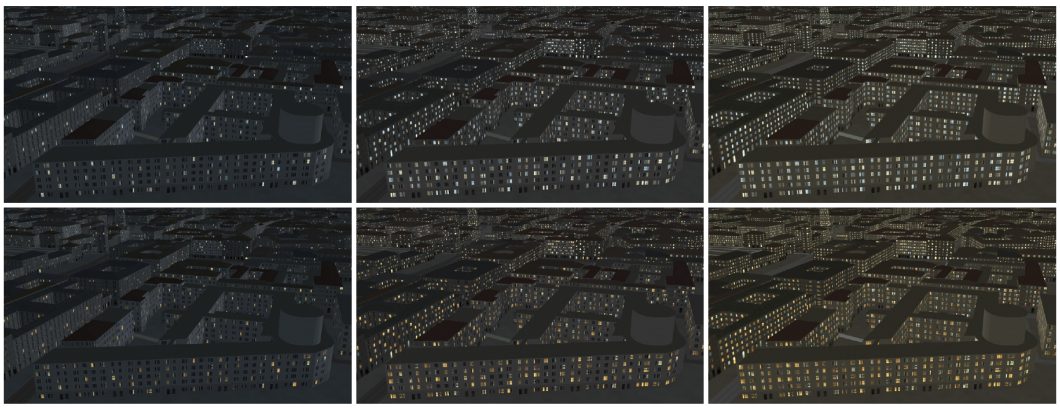


Figure 2.11 – Simulation of window curtain placement and window lighting. The city is rendered with window lighting (top row) and interior room mapping (bottom row). The percentage of curtains opened is controlled by a parameter *thres*. Note that the lit rooms do not contribute to lighting outside their room.

2.3.1 Deferred Rendering

Traditional rendering techniques (from now on referenced as the forward rendering technique) send the scene's geometry to the GPU to be rasterized and tested against the depth buffer, and then shaded when visible. Forward rendering with the use of the depth buffer guarantees that we render only the visible parts (relative to the current depth buffer and subject to its resolution) of the sent geometry. It removes the necessity of sending the geometry from furthest to closest to the GPU, which in turn removes the necessity of ordering the geometry and the dilemma of ordering cyclically overlapping polygons, presented in Figure 2.5, as each pixel is evaluated individually for the depth test.

Deferred rendering changes the usual pipeline to separate the depth test from the shading of the pixel. First introduced by Deering et al. [12] and later refined by Saito and Takahashi [23], a deferred rendering pipeline has two main passes. The scene's geometry is first sent to be tested against the depth buffer. Each pixel passing the depth test then saves its shading information into one or multiple buffers for the second pass. Figure 2.12 shows an example of what information is generally saved in such buffers. Once all geometry has been through the first pass, each pixel then goes through shading using the latest shading information saved for it. The separation of the traditional pipeline in two passes has the drawback that it is harder to implement transparency as

partially transparent objects must be processed after the opaque objects have been tested and shaded. The technique also uses more memory than forward rendering, as we must store the shading information in GPU buffers to transfer it between the two passes. Finally, the separation of the shading pass reduces the number of shaded pixels in a single frame to exactly the number of occupied pixels, and leads to better performance for scenes with multiple light sources, which is a bottleneck for shading-heavy processing.

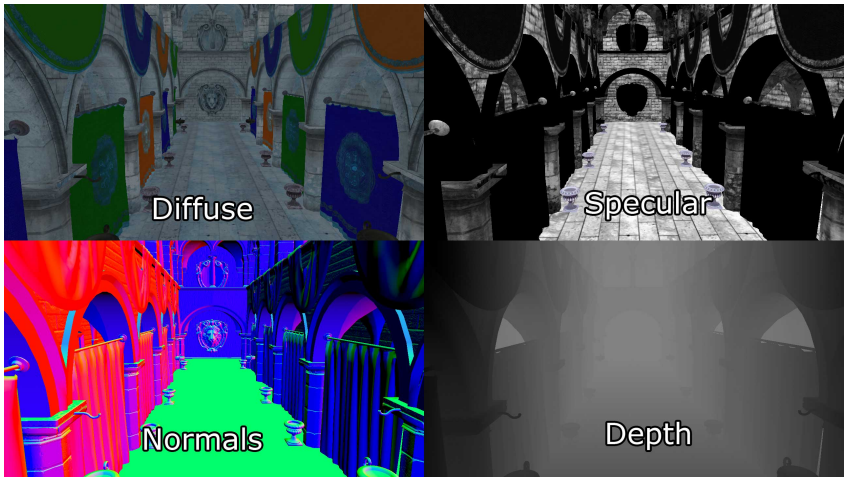


Figure 2.12 – Example buffers used for deferred shading. The diffuse buffer contains the surface color, the specular buffer contains the surface's specular component, the normal buffer contains the surface normal, and the depth buffer contains the depth of the object inside the projection space. Image from van Oosten [25].

2.3.2 Tiled and Clustered Shading

As scenes became more realistic over the years, so did the number of light sources in them. This is also true for simulating global illumination effects using virtual point lights (VPLs). The need to support more light sources in the same frame grew as previous shading pipelines' performance scaled linearly with the number of light sources. Even though deferred shading reduces the number of shaded pixels per frame compared to forward shading, every pixel still has to test which light sources apply to its content. Tiled and clustered techniques aim to reduce the number of times we execute those tests by grouping pixels together.

Tiled shading, introduced by Harada et al. [16], groups pixels in a regular 2D grid of smaller dimension than the image resolution. Each group tests visibility of the scene's light sources and saves the IDs of those light sources that affect at least one pixel in the group. Once all light sources have been evaluated, the pipeline can execute the deferred shading pass and instead of testing all light sources, each pixel only tests the light sources that were not culled for their group.

Clustered shading, introduced by Olsson et al. [20], extends tiled shading to 3D by grouping pixels into cells. The technique is similar to tiled shading in the way that it culls light sources by group, with the difference that pixels in 2D tiles are also separated by their depth. The view-space is subdivided in depth according to a logarithmic distribution to build cells close to cubes.

Tiled and clustered shadings have the potential to cull light sources for multiple pixels at once with the cost of a failed culling test for each light source that affects a cell. The techniques do not change the processing speed of each light source, and therefore do not change the maximum number of light sources that a single pixel can process in a frame. However, they change the number of unique light sources that can contribute to lighting in the same frame, as two different cells can contain the maximum number of light sources but are not forced to contain the same light sources. Neither technique investigate the acceleration in light processing and evaluation, such as in many-lights techniques.

2.4 Many-lights Techniques

In realistic offline rendering, scenes with thousands of light sources are frequent, as one method to approximate global illumination places virtual point lights (VPLs) at every reflection point along light paths first traced from the scene's original light sources. These VPLs are then used to estimate global illumination by evaluating direct illumination on a surface point with all light sources and VPLs. In this way, each VPL represents indirect lighting from a single light path's bounce. Many-lights techniques are designed to optimize the evaluation of such scenes; they have been studied extensively.

We present here two groups of many-lights techniques that tackle the problem from different perspectives. Even though each group can be presented in the perspective of the other, and sometimes referenced each other, they remain distinct on how the methods process their data.

2.4.1 Matrix Sampling

Matrix sampling techniques encode the rendering of an image as a matrix, with each row representing a pixel, and each column representing a light source. The final image could be computed by summing the contributions of all columns together, thus calculating the illumination for each pixel from all light sources.

Introduced by Hasan et al. [17], Matrix Row-Column Sampling (MRCS) is the first method that represents the rendering equation [18] with such a matrix. The method samples the sparse matrix to estimate the sum over the columns. The algorithm, graphically presented in Figure 2.13, starts by rendering randomly selected rows (a row is a pixel with all light sources taken into account individually) using shadow maps on GPU. Then, it assembles the sampled rows into a representative matrix that is partitioned into clustered-reduced columns (columns represent the scene with the illuminance of a single light source). The algorithm then picks a representative column inside each cluster, and renders the whole column. Finally, the algorithm computes a weighted sum of the rendered columns to generate the image. The quality of the result depends heavily on the number of sampled rows and columns, as it determines the quality of the clusters. Considering how the technique clusters light sources, it achieves good results with global light sources (such as the sun) but requires too many samples to effectively reconstruct the illumination of local light sources (such as indirect lighting).

Ou and Pellacini introduce LightSlice [21], an algorithm that further extends the concept of MRCS by grouping similar pixels before trying to resolve the matrix. Where MRCS randomly picked rows, LightSlice groups similar rows of the matrix into slices. The algorithm, presented visually in Figure 2.14, determines similarity through geometric proximity of rows (pixels). The slices are then sampled individually by rendering a representative row inside each slice. Similarly to MRCS, LightSlice then clusters

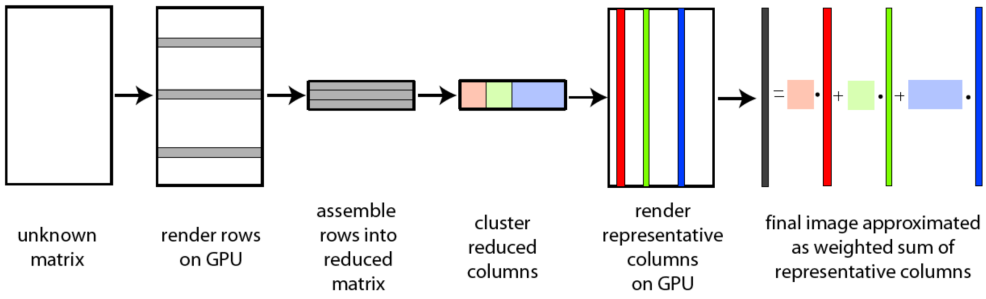


Figure 2.13 – Overview of the Matrix Row-Column Sampling algorithm. Figure from Hasan et al. [17].

columns based on the results of all representative rows. However, they refine column clusters again per slice, based on the results of its own representative row and the neighboring slices' representative rows. Finally, LightSlice renders each slice (equivalent to a group of pixels) by rendering a representative column for each column cluster and applies a weighted sum for all representative columns of the slice. The algorithm suffers from memory usage, as up to 14GB of memory is required to store the data for 800 slices in a scene of 300,000 VPLs. It also suffers from its clustering stage, as its number of clusters is a user-defined value and the efficiency of the algorithm is dependent on choosing the right number of slices for a scene. Furthermore, in addition to the already high memory requirements of the technique, a real-time adaptation could not hope to dynamically evaluate and refine the number of slices and clusters.

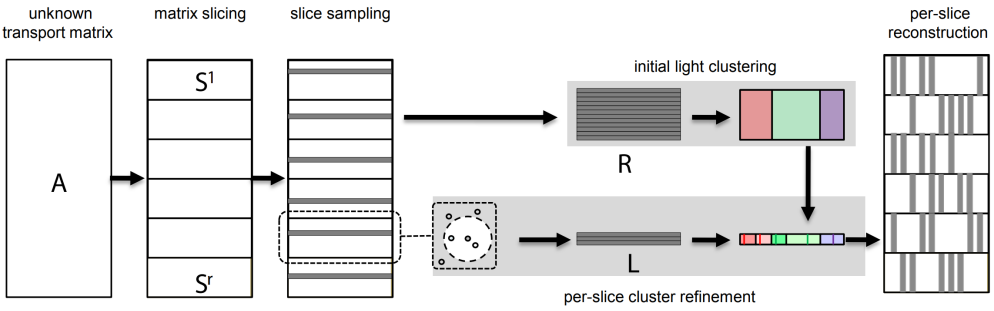


Figure 2.14 – Overview of the LightSlice algorithm. Figure from Ou and Pellacini [21].

2.4.2 Lightcuts

Introduced by Walter et al., Lightcuts [26] is a many-lights technique that focuses on building a binary tree on the scene's light sources, the light tree, traversing it for each individual surface point until the worst potential illumination error is under a user-defined error threshold. The method finally shades the surface point with all the light clusters and light sources that were not subdivided in the light tree. Figure 2.15 presents an overview of the technique with different cuts through a sample tree. The authors approximate direct illumination of a surface point x by a cluster $\mathbb{C}$ by using the representative material, geometric term and visibility term of the entire cluster:

$$\begin{aligned} L_{\mathbb{C}}(x,\omega) &= \sum_{i\in\mathbb{C}} M_i(x,\omega)G_i(x)V_i(x)I_i \\ &\approx M_j(x,\omega)G_j(x)V_j(x)\sum_{i\in\mathbb{C}} I_i \ . \end{aligned} \tag{2.1}$$

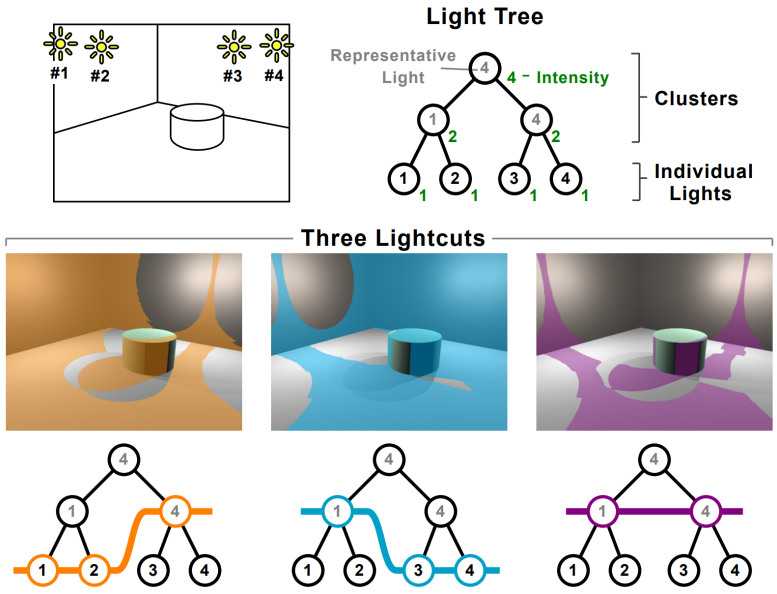


Figure 2.15 – A scene is shown with its light tree, and three sample cuts through it. The colored regions show where each cut manages to keep its error small. Image from Walter et al. [26].

To build the best light clusters, they devise a similarity metric that clusters light sources that are most similar in their material, geometric term, and visibility term, and construct the light tree using a greedy bottom-up clustering. Equation 2.2 presents the metric in which $I_{\mathbb{C}}$ represents the summed intensity of the cluster, $\alpha_{\mathbb{C}}$ the diagonal length of the cluster's bounding box, $\beta_{\mathbb{C}}$ the half-angle of the cluster's bounding cone, and c the diagonal length of the scene's bounding box for oriented lights or zero otherwise. The resulting clusters have an intensity equal to the sum of both its children and a position that is chosen randomly between its immediate children based on their respective intensity.

$$I_{\mathbb{C}}(\alpha_{\mathbb{C}}^2 + c^2(1-\cos\beta_{\mathbb{C}})^2) \tag{2.2}$$

When traversing the light tree for a surface point, the method starts at the root of the tree and subdivides clusters until a satisfying cut is achieved. To determine when to stop subdividing clusters, the method compares the estimated illumination of the cut to the upper bound illumination, and stops when the relative difference between them (presented as the relative error) is lower than the user-defined threshold. The cluster with the worst individual relative error is subdivided and the summed error is calculated again. This repeats until the summed relative error of the cut meets the user threshold. The upper bound of a cluster's illumination is evaluated for each term of the cluster (see the approximation of Equation 2.1) in the following way.

They bound the maximum for the visibility term to the value of 1 as it would be too costly to evaluate even a rough approximation of visibility. The geometric term is dependent of the light source type: The upper bound of a directional light source is set to the value of 1, an omnidirectional (point) light source evaluates $\frac{1}{\|y_i - x\|^2}$ which represents the squared minimum distance between the light cluster's bounding box and surface point x, and an oriented light evaluates $\frac{\max(\cos\phi_i, 0)}{\|y_i - x\|^2}$ which also requires finding the smallest angle ϕ, in relation to the direction of the light, from the cluster's bounding box to the surface point. The material term is equal to the Bidirectional Reflectance Distribution Function (BRDF) times the cosine of the angle between the cluster's representative position and

the surface point. Similarly to the geometric term, to evaluate the upper bound of the cosine, we must find the minimal angle θ, in relation to the inverse of the surface normal between the surface point and the cluster's bounding box.

Lightcuts presents a simple algorithm that scales well with an increasing number of light sources. It differs from other many-lights techniques in the way that visibility is evaluated independently at the time of shading and is not evaluated during tree traversal. While the technique suffers from poor approximation in heavily occluded scenes, it performs particularly well with scenes in free space, that do not contain many occlusions, as it very closely bounds the error of the cut.

As an extension to Lightcuts, Walter et al. present Multidimensional Lightcuts [28], which lets the technique evaluate additional dimensions (such as time, volume, and camera aperture). The extended technique introduces the product graph, which builds a hierarchy over gather points, the graph tree, in addition to building the light tree. For each pixel the method generates a set of gather points, which are then built into the gather tree, and couples it with the light tree. The dual tree represents the association of every gather node with every light node and its traversal selects the appropriate light clusters for the gather points present in the current cut. The technique manages to sample multiple dimensions effectively, but does not take advantage of the similarity of neighboring pixels. Furthermore, the technique suffers from a slow-down on pixels that do not take advantage of the gather tree and its dual traversal.

Davidovič et al. [11] implement the Lightcuts algorithm on GPU, and find that its memory usage is too high for the limited availability on GPU hardware. They present a progressive algorithm using instead a fixed amount of memory, but do not tackle the inadequacy of the Lightcuts algorithm in regard to GPU traversal.

Bus et al. [6] make the observation that Lightcuts and Multidimensional Lightcuts cluster light sources for every pixel when two similar pixels could use the same light clustering. Similarly, they also observe that LightSlice's point clusters (slices) only use the geometric information for clustering and do not adapt to the illumination, possibly introducing artifacts. Their method, Illuminationcut, uses illumination-aware clustering for all pairs of a point and a VPL.

Figure 2.16 – Approximate images rendered as the first phase of the Illuminationcut method. The results are used as upper bounds for the second phase clustering. Image from Bus et al. [6].

The technique clusters the points (pixels) in an octree for which the first levels are subdivided according to the surface normal, and clusters the light sources into a light tree of the same structure as presented in Lightcuts. This lets them cluster together surfaces that are in close proximity and that contain similar surface orientations. An approximate image is then rendered by selecting pairs of point-VPL that satisfy the following criteria: The maximum of the enclosing radii of the individual clusters must be $\frac{1}{10}$ of the distance between the point and the VPL, and the VPL's light cone aperture must be below 20°. Figure 2.16 presents an example of the results of this first phase of their method. They then traverse the light and point trees, using the approximate image as an upper bound. Similarly to the Lightcuts algorithm, they use the light clusters in the light tree to approximate the shading of all the light sources inside it, and add a visibility estimation via a shadow test on the representative cluster. Illuminationcut takes advantage of the spatial coherency of scenes by clustering nearby pixels into dynamic groups. The technique, however, imposes hard constraints on the construction of its upper bound reference image. Since the reference images are built with only the nearby light sources that are nearby and well oriented towards the surface, they do not catch the potential illumination that many far away light sources could sum up to.

CHAPTER 3

LIGHTCUTS GPU ADAPTATION

As presented before, Lightcuts [26] is a many-lights technique that works in two steps. First, in a preprocess, a binary tree is built with the scene's light sources, called the light tree. The light tree is built in a greedy bottom-up fashion, clustering the most similar light sources until a single representative cluster remains. Then, to render a surface point, the light tree is traversed top-down, subdividing nodes that have the largest relative error until the maximum individual relative error of each node in the cut is under a user-defined ratio of the estimated total illumination. At every step of the subdivision, the current set of nodes currently selected in the light tree is referred to as the cut.

We propose moving the second step of the Lightcuts algorithm, i.e., refining the cut, on GPU hardware into the fragment shader stage. GPU has long been used in real-time rendering to shade pixels because it is efficient in executing calculations on vectors and matrices. Most of the GPU efficiency comes from its parallel architecture, which evaluates multiple pixels at the same time. However, it imposes that each parallel process must execute the same code (with potentially different starting values, variables, and some exceptions). Moving the light tree traversal on GPU benefits from the parallelism and speed of the hardware, but imposes that we adapt the algorithm to satisfy special hardware specifications. One such specification is the absence of support for recursive functions in GPU shading languages, such as GLSL (OpenGL Shading Language). Branching statements are another one, since all processes currently running on GPU must execute the same assembly code. Branching statements such as *if-else* and *switch-case* are a known potential slowdown for a GPU shader as, if one GPU process enters a different branch than other parallel processes, both branches will be executed on all processes, with a flag making sure that processes that should not have entered the branch do not modify their final values.

To integrate the algorithm into GPU, we modify some of its mechanics while keeping the advantages of clusters and partial subdivisions of the light tree. In order to reduce

the number of branching statements on GPU, we modify the light tree's structure and the error evaluation necessary to make the cut.

3.1 The Light Tree

The light tree is a binary tree with a total of $2n-1$ nodes for a scene containing n light sources. A cut containing only leaf nodes must execute $n-1$ subdivisions, if starting from the root of the light tree. In a parallel execution on GPU, each process computes a cut for a unique pixel and each light tree subdivision executes a branching statement. Since every completed process must wait until the slowest one is also completed before returning its result, a pixel requiring complete subdivision of the light tree imposes its execution time onto all other processes executing in parallel. Furthermore, since all other processes would have completed refining the cut for their pixel, they cannot profit from the extra execution time available and thus waste computations while waiting unknowingly.

We propose modifying the binary nature of the light tree to a varying number of children. By letting spatial coherency of a city direct the number of children under each node, we reduce the maximum depth of the tree and the number of subdivisions (branchings) needed to reach leaf nodes. By reducing the depth of the tree, we also reduce the variance in the number of subdivisions for all processes. Each subdivision adds substantially more nodes to the cut, adding potentially more details per subdivision than with a binary tree. Although processes that would have required fewer subdivisions may end up with more nodes in their cut, the added nodes give more details and still represent a reduction in wasted time due to less waiting for incompleted processes. Thus, the method is more conservative.

By making each tree node contain a varying number of children, we create a light tree that can better adapt to the city's natural layout. Our clustering method starts with the smallest objects in our scene containing light sources, in our case light emanating from apartments through windows. It moves up the object hierarchy of the scene, clustering all the light sources of children nodes at every step. For instance, the representative cluster

of a building floor would contain the light sources shining through all the windows on the floor, and a city block would contain all the light sources from every building within it. Figure 3.1 presents the conceptual difference between the Lightcuts' clustering method and our revision.

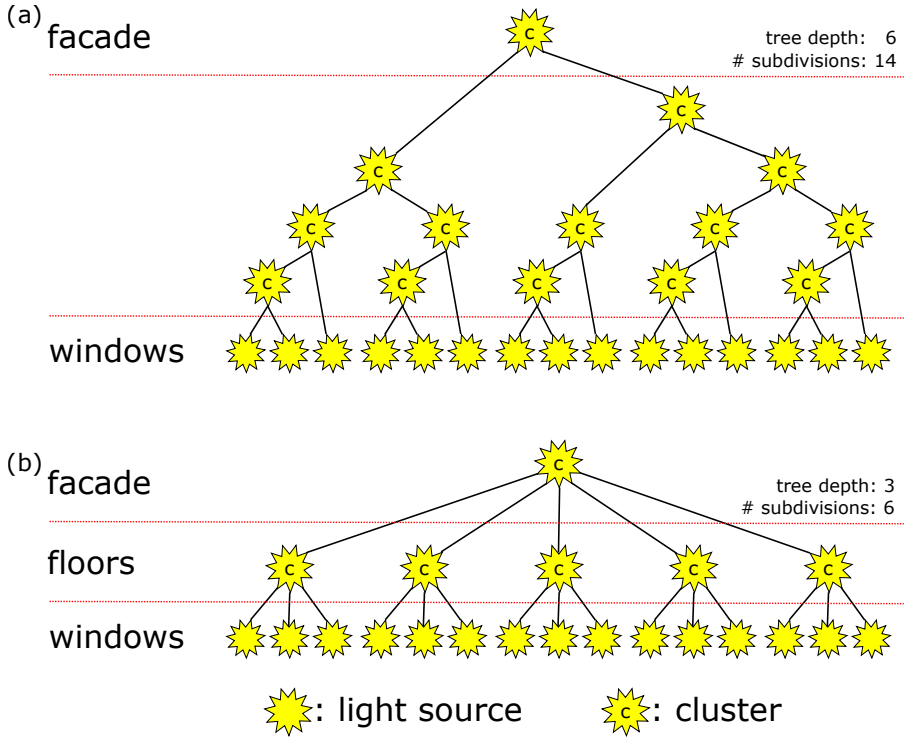


Figure 3.1 – We build the light tree for a building facade containing five floors and three light sources on each floor according to (a) the Lightcuts method, and (b) our revised method. The clustering of the Lightcuts method builds the light tree by clustering the most similar light sources in a binary tree, resulting in a larger depth and a larger number of subdivisions necessary to achieve a cut. Our revised clustering structure clusters light sources together according to the special city-like object hierarchy, resulting in a shallower tree and fewer subdivisions to achieve a cut.

Furthermore, traversing the light tree is a recursive operation that cannot be efficiently implemented as if it would be done on CPU, directly on GPU. To alleviate the problem of recursion we linearize the light tree before sending it to the GPU. Our linearization goes depth first through the tree and saves each node in an array. To be able to skip the sub-trees of clusters, we store, for each node, the number of nodes in their sub-tree. Then, we can skip a node's sub-tree by advancing the array index by its sub-tree

size. Table 3.I shows the structure stored on GPU for each node in our revised light tree.

Data Type	Description
Light	Representative light
↳ 3D vector	↳ Position
↳ 3D vector	↳ Radiance
↳ 3D vector	↳ Direction
↳ Float	↳ Cone opening angle
AABB	Axis aligned bounding box
↳ 3D vector	↳ Minimum position
↳ 3D vector	↳ Maximum position
Integer	Number of nodes in sub-tree

Table 3.I – Node structure of our revised light tree.

3.2 Refining the Cut

At the start of an evaluation, a cut is composed of the node at the root of our light tree. The Lightcuts technique refines the cut by subdividing the node with the largest relative error until the summed relative error of all the nodes in the cut is under a user-defined threshold. The relative error is computed by comparing the estimated radiance and the upper-bound radiance of the current cut for a surface point to shade. The estimates are evaluated with Equation 2.1, with the material, geometric term, and visibility term computed with independent upper bounds.

To reduce redundant evaluations, Lightcuts keeps each node of the cut in a list ordered by its relative error, and reorders the list at every subdivision. Since maintaining an ordered list on GPU requires a large number of branching statements, which, as presented previously, slows down evaluation, we tried replacing the order list with a traversal of the list to choose the node with the largest error. This did not lead to satisfactory performance.

To better integrate the Lightcuts algorithm to GPU, we replace the summed error evaluation with individual tests at each node. By evaluating nodes independently, we lose the guarantee of a maximum relative error threshold per pixel, but we benefit from faster independent node subdivisions. To estimate the relative error of a cluster, the Lightcuts

method evaluates the maximum potential contribution of each node, its radiance upper bound, and compares it against the estimated radiance of the cluster.

We find that this test fails to subdivide clusters when its representative light source is positioned at the best potential position to evaluate its upper bound. Since both the upper bound and the estimate use the same position, there is no difference between the two. Figure 3.2 shows resulting artifacts. This indicates there is no benefit to subdividing the cluster when in fact the cluster is not guaranteed to properly estimate the radiance of its children.

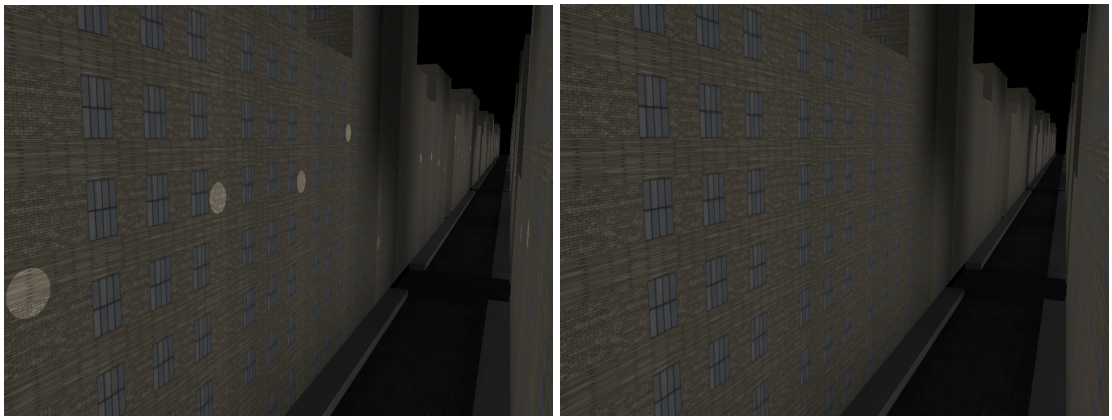


Figure 3.2 – Left: A relative error threshold depending on the comparison of the upper bound radiance of a cluster and the estimated radiance of the same cluster can lead to artifacts, appearing here as brighter disks on the building facades, when both use similar positions to evaluate the geometric term and the material term. Right: Our subdivision does not suffer from such artifacts as we subdivide as long as the error bound has a radiance over the user-defined threshold.

We propose comparing the upper-bound value to a user-defined threshold value, such as $\frac{1}{255}$ in sRGB color space, and to subdivide clusters until their radiance falls under the threshold. Although the sum of all cluster errors for a pixel can exceed the threshold, we find that in practice, the quality is quite close to the fully subdivided result. This modification achieves a significant speedup, which is further detailed in Chapter 5.

3.3 Additional Modifications

Building the light tree is a very expensive operation, as we have to cluster all the scene's light sources according to a similarity metric that evaluates their summed intensity, cone of orientations, and bounding box. Since in a real-time scenario we could change a light's intensity, we have facilitated the construction of the light tree by using the city's natural hierarchy to replace the previously mentioned similarity metric. Using the city's spatial coherency, we cluster light sources together when they are contained in the same object, starting from the leaves and moving up in the hierarchy of the city until we have only one representative cluster.

Furthermore, the Lightcuts technique uses a probabilistic function to choose the position of the representative light source in each cluster. The position is chosen randomly from the two children light sources, with a weight proportional to their respective intensity. This probabilistic method helps the light tree construction to remain unbiased in a Monte Carlo sense, over the whole image. Following the previous changes to error evaluation, we change the probabilistic method to a deterministic one, using the average of the positions of children nodes weighted according to their intensity, similarly to Paquette et al. [22]. We compute the cluster's position $x_{\mathbb{C}}$ using Equation 3.1, in which I represents the intensity of the light source and i one of its child nodes.

$$x_{\mathbb{C}} = \frac{\sum_{i \in \mathbb{C}} x_i I_i}{\sum_{i \in \mathbb{C}} I_i} \tag{3.1}$$

Through the adaptation of the Lightcuts technique to GPU hardware, we make the technique more suitable to a real-time rendering setup. Results in timings and visual accuracy are presented and detailed in Chapter 5.

CHAPTER 4

FACADE-CLUSTER VISIBILITY

Our Lightcuts GPU integration lets us shade a pixel once we have its fragment, all its shading properties, and all the light sources that shine on its visible surface. Although the deferred pass gives us the shading properties, we do not yet have the list of light sources that shine on the geometry. If we were to shade each pixel with the full light tree, the Lightcuts technique would assume complete visibility for all the light sources of the scene as it does not provide visibility. Therefore, we need a prepass that computes visibility of light sources for all visible geometry.

Doing so would be prohibitively expensive, as will be shown later in Chapter 5. Therefore, we exploit image coherency to devise a from-region visibility technique that we name *Facade-Cluster Visibility* (FCV), inspired by Durand et al. [13], to evaluate the light sources visible from a view cell. This way, we compute the visible light sources for a group of pixels at once. Furthermore, we compute the visibility of light sources by traversing our light tree. This traversal inherently gives us a precut of the light tree, which can be interpreted as a visibility cut. To compute the final image, we provide to each visible object its precut, refine the final cut for each pixel on GPU, and shade each pixel with the clusters present in it.

4.1 Building the View Cell

The first step to our technique is the construction of view cells for which we will evaluate visibility. As presented in Chapter 2, CHC++ [19] is a visibility algorithm that uses a bounding volume hierarchy (BVH) over the scene's geometry, and hardware occlusion queries (HOQs) to determine objects visible to the camera. Since we use CHC++ in conjunction with deferred shading, we build the GPU buffers containing our shading properties during visibility evaluation of our scene. By doing so, we benefit from HOQ support, which returns each object's pixel coverage, i.e., the number of pixels

where it falls in front of the current depth buffer. This value is not accurate as it only reflects pixel coverage at the time of the test. Indeed, we update the depth buffer when an object is evaluated as visible. If later in the processing of the ordered list of objects, a new object B considered visible hides a previously visible object A, completely or partially, the pixel coverage of object A is not accurate anymore. Therefore, such pixel coverage can only be used as an upper-bound estimate of how many pixels the object covers in the final image.

When retrieving the results of visibility tests from HOQs, we save the pixel coverage for the tested BVH node. Doing so gives the BVH information on the estimated pixel coverage of each of its clusters. To build view cells, we traverse the BVH, stopping only at clusters with a pixel coverage larger than a user-defined threshold. Every cluster at which we stop constitutes a view cell. This step produces dynamic view cells that have similar pixel coverage but that may have much different sizes in the scene.

For each view cell constructed this way, we traverse its cluster and note for each normal of its children objects, which of the six cardinal directions in 3D space ($\pm X$, $\pm Y$, $\pm Z$) is the normal best represented by. More precisely, we choose the cardinal direction with which the normal makes the largest dot product. Then, we divide the problem of evaluating visibility for the view cell from all directions at the same time into up to six individual visibility evaluations, one for each unique cardinal direction that we noted for the view cell. We later associate the results of each visibility test for the view cell to its cluster and children objects according to the direction of the visibility test and their surface normals.

For instance, if an object A is represented by the front facade of a building and one of its side facades, assuming both are represented by flat surfaces, we would note that two cardinal directions, say $+X$ for the front facade and $-Z$ for the side facade, are representative of the object's normals. Later, we set the light-PVS of A as the union of the divided visibility tests for A's view cell for the cardinal directions $+X$ and $-Z$. If, instead, both facades were separated into objects A and B, then object A would only be associated the light-PVS of direction $+X$ and similarly B only the light-PVS of direction $-Z$.

A similar division of view cell visibility computations is also done by Durand et al. [13] for their extended projections technique, upon which we base our visibility technique.

4.2 Visibility Algorithm

As presented in Chapter 2, Durand et al. [13] (Extended Projections) compute visibility by intersecting the projections of occluders' umbra and the projections of potential occludees' visibility region. Figure 4.1 presents a quick overview of the process. We base our visibility algorithm on their theory and adapt their implementation to better suit specificities of our problem, i.e., by evaluating visibility analytically instead of using rasterization, by integrating the light tree of our Lightcuts implementation, by representing the 2.5D occluders with only their horizon on the projection plane, and by computing the reprojection without using rasterization.

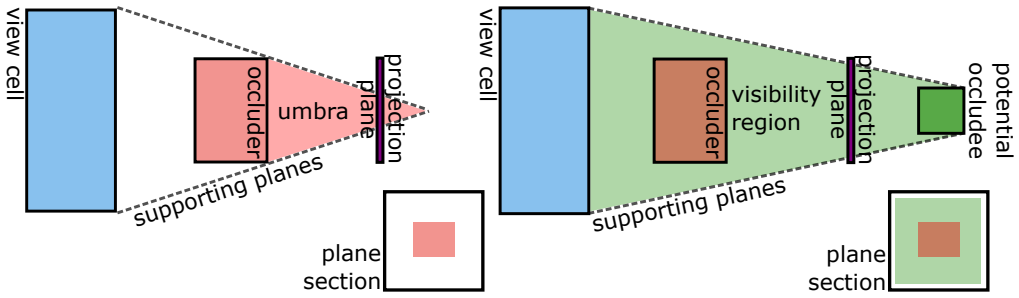


Figure 4.1 – Left: The umbra region of an occluder is defined by the region behind the occluder, in relation to the view cell, and in between its supporting planes. We intersect the umbra region with our projection plane to define the projected occlusion region of the occluder. Right: A visibility region is defined by the region between the view cell and a potential occludee, and in between its supporting planes. We intersect the visibility region with our projection plane, and if the projected visibility region (green) is not completely covered by the projected umbra (red), it is considered at least partially visible.

4.2.1 Analytical Evaluation of Visibility

To achieve real-time performance, the original technique [13] uses rasterization to intersect the umbra of occluders with the projection plane. We propose instead to keep visibility computation on CPU.

4.2.2 Integrating the Light Tree

The original technique, Extended Projections, builds a BVH over the scene's geometry to represent potential occludees and to test visibility of each node, starting at the root. When the visibility test evaluates a node as completely occluded, the node is pruned from the tree, otherwise it is traversed. In our problem, since our only potential occludees are light sources of the scene, and since we have already computed a light tree for the following shading pass using Lightcuts, we replace the BVH with our light tree. With that change, while evaluating visibility, we actually compute a visibility precut of the light tree, which contains every cluster that is fully visible from the point of view of the view cell. We assign this precut to every object inside the view cell associated with the tested direction.

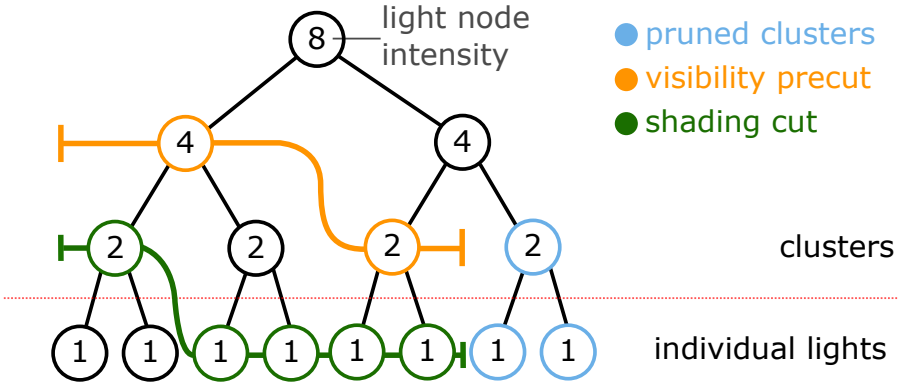


Figure 4.2 – In our visibility algorithm, for each view cell, our light tree is first refined by evaluating a visibility precut. Then, for each pixel, a second cut is computed from the precut, further refining the light tree to acquire details relevant to the surface position and normal.

Then, when shading a pixel, we traverse only the clusters present in its visibility precut to effectively execute a final cut, referred thereafter as a shading cut. As presented in

Figure 4.2, since the shading cut refines the precut, the pruned clusters are not evaluated during the shading cut and therefore are not used to shade the pixels using that precut.

To pass the information of the precut to the GPU, we build an array in GPU memory that references the indices in the light tree array, for each view cell and for each cluster inside the precut of that view cell. To link every object to their view cell precut, we also build a 2D array in GPU memory that references, for each object, the first index of their precut and the number of clusters in it. Finally, every pixel stores its object ID in a texture during the G-buffer pass of the deferred pipeline. During the shading pass, each pixel only needs to look up their object ID inside the *per-object precut indices* array, fetch the clusters for their view cell in the *visibility precut cluster list* array, and execute a shading cut for each cluster. Figure 4.3 presents the two dynamic arrays and how they link their data.

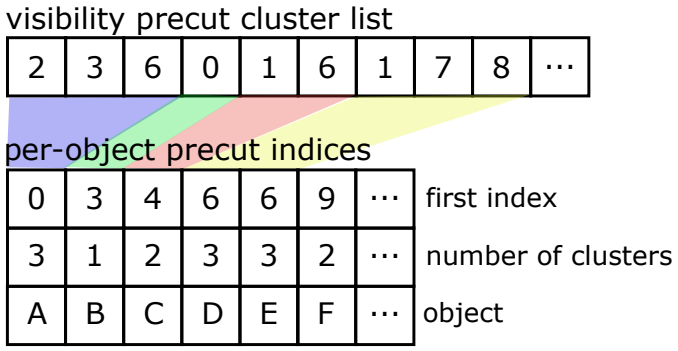


Figure 4.3 – To reference the visibility precut per object on GPU, we build a precut cluster list that contains all the light tree indices for its precut clusters, for each unique view cell (illustrated here with different colors). Each object only needs to reference where its precut starts and the number of clusters in it. For instance, in the array above, objects D and E both have the same precut, since they were computed in the same view cell, starting at index 6 and containing 3 clusters. The two arrays are transferred to GPU memory after visibility evaluation.

4.2.3 Occluder Representation on the Plane

To cull our light sources we must project occluders on the projection plane. While we could use every object in our scene as an occluder, Coorg and Teller [9] observe that large occluders in a city, such as buildings, produce most of the occlusion occurring at every

point of view. Therefore, to reduce the size of our occluder list, we only use buildings as occluders in our cities. We build a BVH over the buildings to group together nearby buildings when projecting them onto the projection plane. Furthermore, we project the bounding box of a BVH node onto the plane to reduce the complexity of the objects projected. Projecting the bounding box of a BVH node implies that we consider it as fully occluding. Since a BVH node could contain two buildings with a street in between them or two buildings of different heights, it is not true that all BVH nodes are fully occluding. To reduce potential errors, we start by grouping contiguous buildings before growing our BVH with more distant buildings, thus reducing the number of partially occlluding BVH nodes.

Using only buildings adds another benefit to our technique. Since we consider buildings as 2.5D objects from the ground updward, instead of projecting onto the projection plane every edge of a bounding box of a set of buildings, we project only the top-most visible edges of the box, i.e., those forming its horizon, thereafter referenced as the occluder horizon. Visibility with 2.5D occluders implies that a point of view can never be lower than the bottom of an occluder, which reduces the precomputation of the horizon edges for boxes to only two height categories: below or above the box's height. Figure 4.4 presents each category along their precomputed horizon edges and a sample point of view.

To project an occluder onto a plane, we must compute the occluder's horizon from our view cell. As the horizon changes depending on where a point of view lies inside the view cell, we build the horizon by selecting every edge that could form a horizon edge for a point of view inside the view cell. Figure 4.5a presents a view cell and the occluder horizon formed by regrouping the edges of underlying points of view.

Although a point of view and a box form either one or two horizon edges, as seen in Figures 4.4 and 4.5a, a view cell with a box can form one to four edges. By forcing the view cells and the bounding boxes in the scene to be axis-aligned, we reduce the maximum number of edges to three. To further reduce the complexity of visibility evaluation, we conservatively reduce the horizon to a single edge, formed by its extremal vertices. Figure 4.5b shows a view cell with the extremal vertices of its three-edge horizon in red

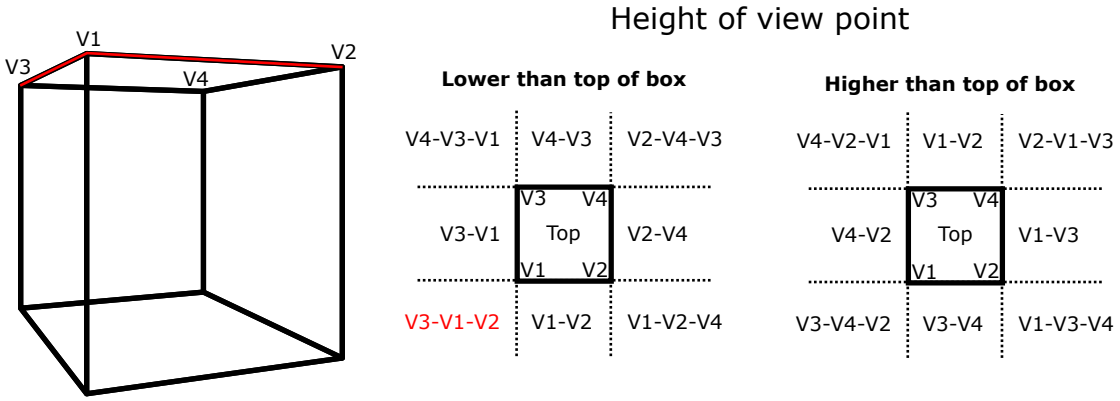


Figure 4.4 – The vertices V that form the horizon for a single point of view can be precomputed. Left: A box, with the edges forming its horizon in red. Middle and Right: Two height categories exist to determine horizon edges for 2.5D boxes. Each category is presented with eight sets of vertices placed respectively in their section around the box, seen from the top. Any point of view inside one section would define the box's horizon with the listed vertices in that particular order, for left-to-right. The set of vertices forming the horizon on the left is highlighted in red for its respective point of view.

and the simplified horizon edge also traced in red. Figure 4.7 overlays our simplified horizons on the city from a point of view.

This simplification has drawbacks when the occluder gets closer, as the lost occlusion area is also getting larger. Fortunately, as we project occluders that are further away, the lost occlusion area reduces too. Furthermore, this simplification conservatively represents the shifting horizon for a view cell. As Figure 4.6 shows, a view cell can contain multiple points of view that see a much different horizon for the same box. While our simplification (edge in red) does not optimally represent the occluder for all these points of view, it is quick and efficient in creating a good approximation that is valid for any point of view inside the view cell. At worst, something that could have been culled will not be, but a secondary pass could test it again.

4.2.4 Visibility Evaluation

We alter Durand et al. [13]'s visibility test of the occluder and occludee coverage on the plane to instead take into account the occluder horizon and the occludee horizon. Since every occluder projected is between the view cell and the projection plane and

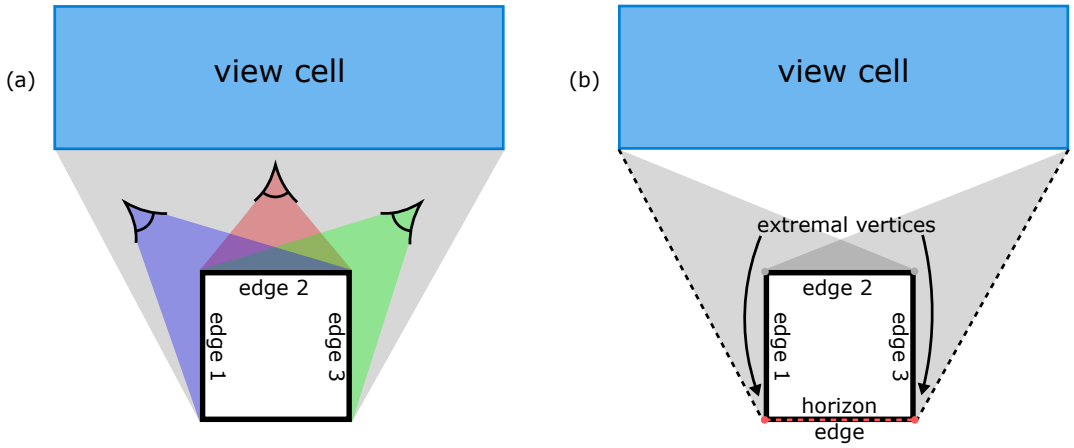


Figure 4.5 – (a) The horizon of an axis-aligned box from a point of view contains either one or two edges. From any view point in an axis-aligned view cell that does not overlap the box, there can be one to three edges forming the horizon. (b) To reduce the complexity of the horizon, especially for view cells, we replace all horizon edges with a single edge, constructed from the two vertices at each end of the standard horizon edges.

since every potential occludee is behind the projection plane, we can assume that if an occludee horizon is above an occluder horizon, than the occluder does not fully occlude the potential occludee, and vice versa. Figure 4.7 presents the concept from a point of view, showing that for fully occluding 2.5D buildings, each horizon of buildings achieves complete coverage of any object behind it with a lower horizon.

Since GPU hardware is not adapted to visibility computations that contain many branching statements, we decided to execute the computations on CPU, and transfer the visibility data to GPU memory for the final shading pass. This modification of the computations also lets us, unlike the original technique, use an infinite plane for projection and removes the need to rasterize occluders onto an image for reprojection. This change of precision removes potential artifacts originating from the image resolution, well known for the family of shadow mapping techniques.

4.2.5 Grouping and Reprojection

We have not yet described how we reproject the planes onto further planes and how we group horizons to achieve occluder fusion. As can be seen in Figure 2.7, an occluder

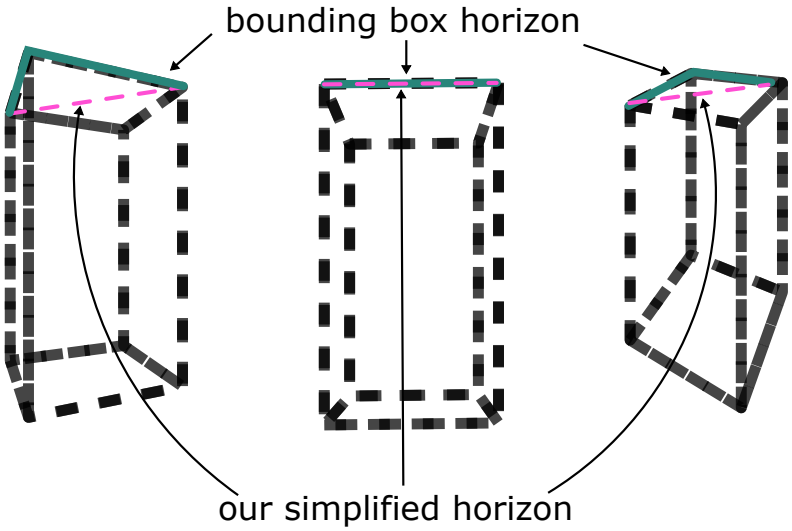


Figure 4.6 – When viewing an occluder within a view cell, shifting the position of a point of view inside the view cell can alter the shape of the occluder. The three boxes above show three sampled points of view of the same box. As can be noticed, although our simplified horizon (in pink) loses some occlusion potential by being conservative, it does so while maintaining much of the occlusion potential no matter the point of view inside the view cell.

smaller than the view cell has a finite occlusion area. The fusion of smaller occluders can combine their occlusion areas to better represent their actual mutual occlusion. Moreover, since our view cell can be much larger that our smaller buildings, we must implement occluder fusion to catch all occlusions from our scene.

To achieve occluder fusion, we group occluder horizons, from now on referenced to as nodes, in multiple tree hierarchies. When two nodes have their respective horizons overlap along the X axis of the plane (the axis perpendicular to height), we group them together into a new node, with a fix-height horizon. The height of the representative node is determined by the minimum height of both underlying nodes. This way, if a representative node's horizon is higher than a tested occludee, we can guarantee that it is occluded by the fusion of its underlying nodes. Figure 4.8 shows a node hierarchy of horizons being grouped and the heights of newly formed groups. Each highest node in the groups is that group's root and is considered a sibling node to every other group root.

To test whether a potential occludee is visible or not, we project it onto the plane

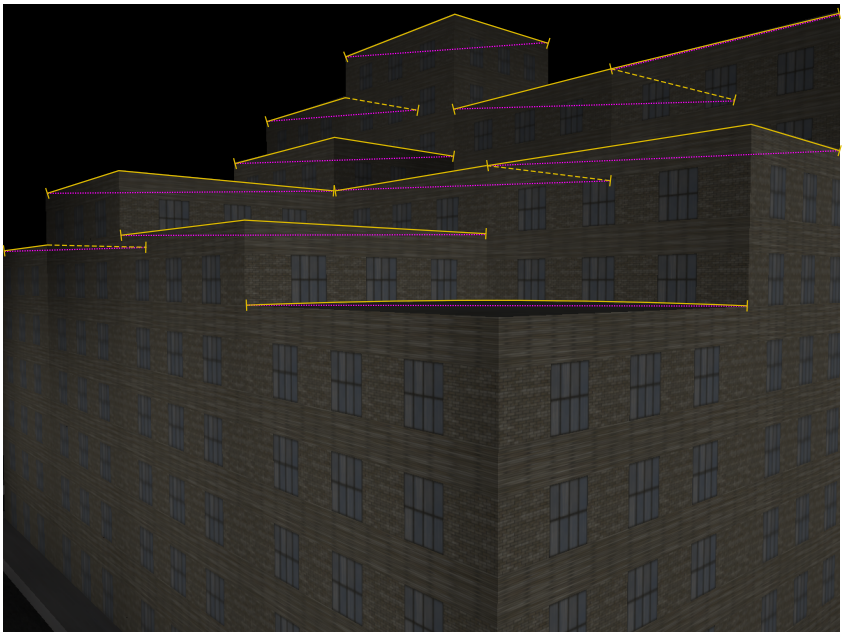


Figure 4.7 – We project only the horizons of our occluders, the buildings, onto the projection plane to test for visibility. Here, a sample of horizons are highlighted in yellow for a point of view, with our simplified horizon traced in pink.

and compare it to every potential horizon group. If the potential occludee overlaps a horizon group in the X axis of the plane and if the occludee's minimum height is lower than the maximum height of the group, the group could hide the occludee, and needs to be tested with it. When testing an occludee with a group, we first test the highest node, i.e., the root, to see if it occludes the potential occludee. If the node occludes it, the test ends with a full occlusion verdict. However, if the group does not fully occlude the potential occludee, we traverse the node and test the potential occludee with the node's two children. This step is repeated until either all nodes of the group are tested against the potential occludee or it is occluded. When we have finished testing with a group, if the potential occludee is not occluded, we repeat the same steps with sibling groups until either all potential groups have been tested or the occludee is evaluated as hidden.

The final step to achieve useful occluder fusion is the reprojection. Introduced by Durand et al. [13], the reprojection step takes all the projections on the current projection plane and reprojects them onto a projection plane further away. This lets us reuse the previous projections and accumulate the projections onto every projection plane. The

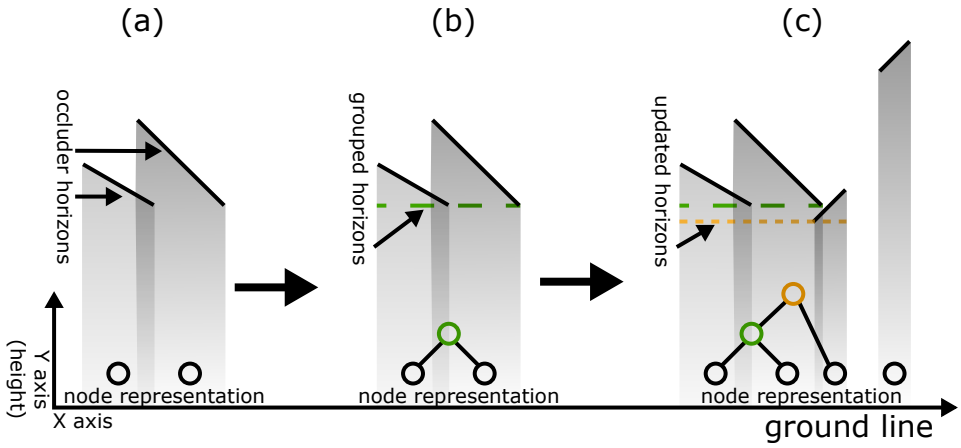


Figure 4.8 – Three steps are presented for grouping occluder horizons. The current nodes and their hierarchy are presented under each horizon. The top nodes of each group are siblings. (a) Occluders are projected as horizons on the plane where they overlap in X. (b) The horizons are grouped in a conservative flat horizon at the lowest point. (c) Two additional occluder horizons are projected on the plane. One horizon overlaps with the previous group and is grouped with it, forming a new node, higher in the hierarchy, but lower in height than its horizon. The other horizon does not overlap and builds its own group containing only itself.

reprojection is a crucial step as we group occluders only once they are projected onto the projection plane. This gives us the possibility to project small occluders on a nearby plane, group them together, and reproject them onto a further plane while benefitting of their combined occlusion.

Since their original method used rasterized images to represent occluders, their reprojection step applied a convolution based on the inverse result of the previous projection plane and the projection of the new occluders. Since we do not use rasterization, we can use a simpler solution. To reproject a projected horizon, we build the supporting lines from the view cell to the horizon vertices, and intersect the lines with the next projection plane. Then, if the node is not collapsed (a node is collapsed if its occlusion area has collapsed), we trace the horizon edge on the new projection plane.

When reprojecting a group, we traverse it, starting from the root, and reproject every node onto the new plane. While some groups may lose the nodes of smaller occluders or subgroups, their hierarchy is kept the same. While traversing the group, if a node is collapsed, we stop the traversal for this node and its children, but do not change the

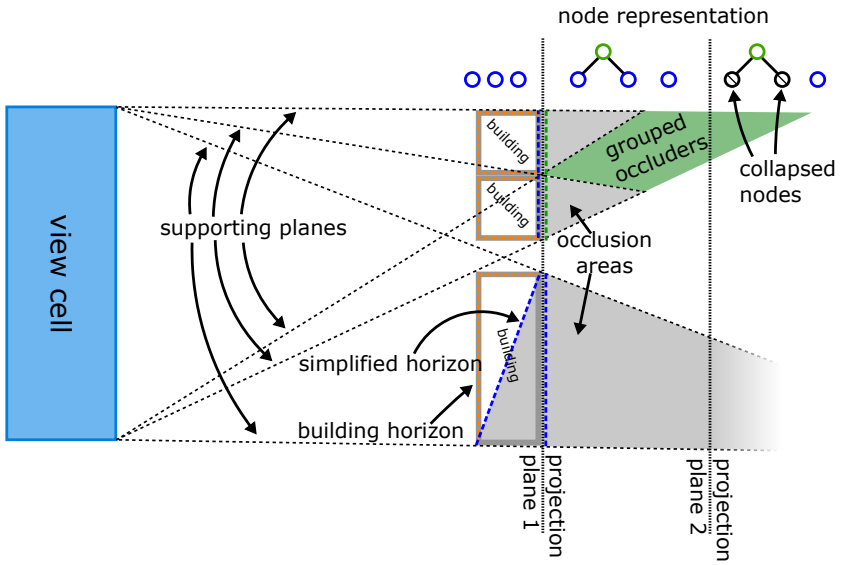


Figure 4.9 – To build the occlusion node structure, we project every occluder's simplified horizon onto projection plane *1* and build their occlusion nodes, which can be seen in the node representation at the top. We then group every overlapping node into a larger representative node. To reproject the occlusion nodes onto projection plane *2*, we use the supporting planes of each node to project them from projection plane *1*. Nodes that have their supporting planes meet before the new plane (collapsed nodes) are pruned from the structure. Grouped occluders are kept even if their children nodes are pruned, as long as they are not considered collapsed themselves.

reprojection of its parent nodes. Doing this, we reproject and keep the details of each node until their independent reprojections are no longer valid. While we potentially lose the details of independent projections, we keep the advantage of occluder fusion at a low cost. However, since we do not have a perfectly defined representation of fusioned occluders (we only compute a simple conservative fusion), we potentially lose occlusion coverage when rejecting the reprojection of collapsed nodes. Figure 4.9 illustrates in 2D how we reproject nodes and how collapsed nodes are pruned.

Overall, we have refined a technique to evaluate visibility for view cells. It optimizes the projection of large occluders onto a common plane and evaluates visibility by testing the horizon of occluders with potential occludees. Furthermore, we have developed a new structure to conservatively fuse occluder horizons on the plane while keeping the details of individual occluders as long as possible.

CHAPTER 5

RESULTS AND IMPLEMENTATION

The goal of our research is the real-time rendering of a city scene at night, including all the shading and shadow details of the light sources found in such a scene. To achieve our goal, we implemented our visibility algorithm introduced in Chapter 4, connected it to the CHC++ [19] algorithm, and rendered images with our revised version of Lightcuts [26].

In this chapter, we introduce the procedural scene generator that we built to analyze our research, we recap how the steps of our algorithm connect with each other, as well as the optimizations that were added to the algorithm, we study the scaling and behavior of our technique under different perspectives, and we discuss results that we achieved.

The results were obtained with an Nvidia Geforce GTX 1070 GPU, and an Intel Core i5-3470 CPU. Rendering times were measured for an image of a 1200×900 resolution.

5.1 Scene

Since our algorithm depends on features of an underlying common structure for a city, we built our own procedural city modeler, which generates blocks of buildings inside a layout defined by our algorithm. The streets are laid out in a grid pattern and provide visibility of many buildings at the same time when looking down along the streets. Furthermore, we can offset the position of block rows by half a block, similar to a regular offset brick pattern, creating street intersections with an H pattern for which a point of view looking along the offset street sees much fewer buildings. Figure 5.1 shows our layout seen from above.

To illuminate our scene, at every window for every building facade, we add a spot light source pointing in the direction normal to the window. All our light sources project light in a cone with a 45°opening from the normal and a linear falloff of intensity towards its edge. We optimize the generation of the city to avoid light sources between adjacent

buildings. One of our tests is composed of 6×6 blocks, each containing 44 buildings. Each building is divided into four facades and a roof, with each facade divided in a procedural number of floors. Each floor is composed of five rectangles (further referenced to as quads) representing walls and three quads representing windows. In total, this city contains 978,652 triangles and 39,202 light sources.

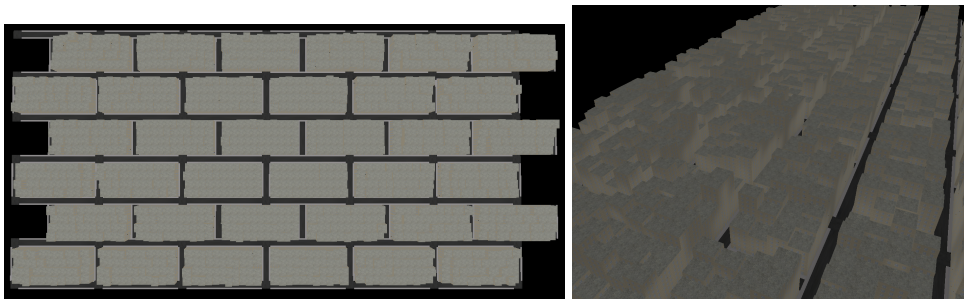


Figure 5.1 – We build our street layout of our city to create multiple different points of view that enable us to test how the algorithm scales with larger or smaller numbers of visible buildings. Each block is composed of multiple buildings of varying heights, and is separated by simple models of streets and sidewalks.

5.1.1 Impact of Different Points of View

When rendering a city, its structure and the camera's point of view have a great impact on the number of visible facades. Here we present multiple points of view, that will be discussed further in the chapter when analyzing the performance and scaling of our technique.

Table 5.I shows statistics for four points of view in the same procedural city. Each point of view is chosen to increase by tenfold the number of triangles in the potentially visible set (PVS). The poor performance in the first pass of our deferred pipeline is due to the fact that our city is defined by a hierarchical structure, for which only the leaf nodes are represented with geometry. In our case, every wall and window is defined by a quad (two triangles forming a rectangle), and every floor on a single facade is composed of five walls and three windows (the wireframe structure of the facade is shown in Figure 5.2). Since every quad listed for render is sent to the graphics hardware one by one, every visible floor of every visible facade generates nine draw calls.

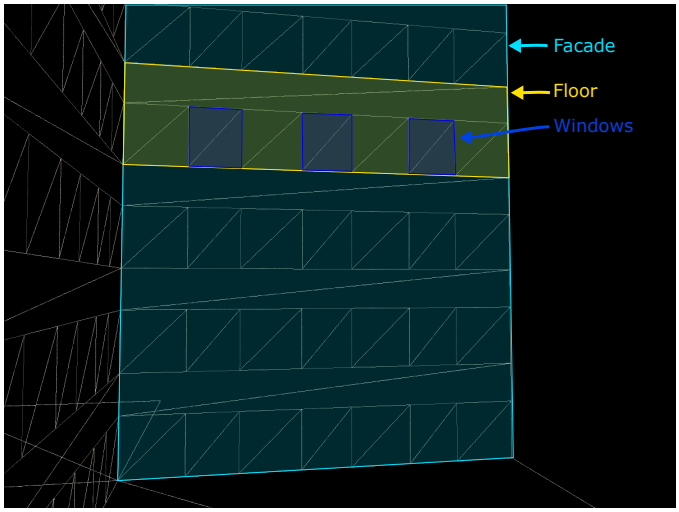
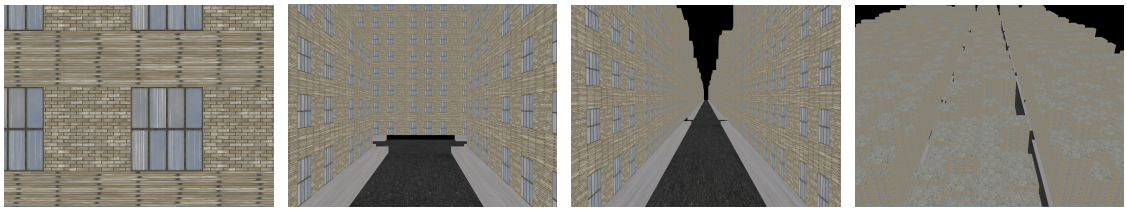


Figure 5.2 – The wireframe structure of a facade in our generated city. A facade is composed of a procedural number of floors. Each floor is composed of three quads representing windows and five quads representing walls.

The implementation of instanced draw calls would help to improve performance, however, this was not done, since the specific focus of the research is on the visibility technique and the implementation would require a lengthy modification of our CHC++ implementation. Figure 5.3 presents each point of view with only ambient illumination on the surfaces.

Point of view	Triangles in PVS	Deferred first pass
Single facade	160	1.0 ms
Close-ended street	1 156	1.7 ms
Long linear street	14 260	9.2 ms
Bird's eye view	100 112	54.3 ms

Table 5.I – Changing the point of view on the same city can drastically impact the ratio of visible geometry. Four points of view are chosen with increasingly more visible buildings; they are listed along with the time required to render them on GPU for the first pass of our deferred pipeline. The scene contains 978,652 triangles.



(a) Single facade (b) Close-ended street (c) Long linear street (d) Bird's eye view

Figure 5.3 – The scene viewed from four different points of view.

5.2 Implementation

We implemented our technique defined in Chapter 4 on CPU using C++ and OpenGL as graphics hardware API. Our CPU pipeline is not implemented using multi-threading and so could be further optimized. This section provides a quick overview of our complete pipeline, and presents a few additional optimizations that were implemented to accelerate the technique.

5.2.1 Pipeline Summary

To render a frame with the final implementation of our visibility algorithm, we perform three major steps.

First, using the CHC++ [19] visibility algorithm on CPU, we render the scene from the camera's point of view. When traversing the BVH structure of the scene, for each node, we save the visibility estimation returned by the HOQ as its maximum pixel coverage. Furthermore, when rendering the visible geometry on GPU, we save the information relevant for each pixel, such as the surface's position, normal and color, in G-buffers (textures of the size of the final image, saved on GPU).

Then, with the surfaces identified as visible by the CHC++ algorithm, we build groups of similar pixel coverage with nearby facades. For each of these groups, we build a view cell and evaluate the visibility of the light tree, i.e., a hierarchy of light clusters over the scene's light sources, against the occlusion created by the city's buildings. This visibility evaluation prunes the nodes in the light tree that are not visible from the view cell. After the pruning, the remaining nodes, called the visibility precut, are

saved in a list that we store on GPU. These nodes are represented as orange nodes in Figure 4.2.

Finally, for every pixel in the final render on GPU, we retrieve the pixel's properties in the G-buffers and the visibility precut associated with it. Each pixel executes a final cut on the light sources previously evaluated as visible for its view cell. The last cut is evaluated using our revised Lightcuts algorithm introduced in Chapter 3. The final pixel color is computed by shading the surface with all the light sources in the final cut.

5.2.2 Optimizations

When implementing our visibility algorithm, we further optimized our technique to accelerate visibility evaluations. The optimizations are detailed in this section and the statistics are shown in Table 5.II.

First, as presented and discussed in Chapters 2 and 4, Coorg and Teller [9] note that most of the occlusion present in an urban scene is generated from large occluders close to the point of view. In a city scene, most of the occlusion comes from buildings, which are great candidates to generate most of the occlusion by themselves. Therefore, we restrict the occluder set used for visibility evaluations to only contain buildings in the scene. As can be seen in the results, a significant speed-up ($3\times$) is attained when only using buildings as potential occluders.

Second, when testing visibility of light clusters, our visibility algorithm compares the projection of their bounding boxes on the current projection plane with the accumulated horizon on the same plane. After testing the light sources, we reproject the horizon on the projection plane defined by the next occluder. Light clusters that are found between the current and next projection planes and that were not evaluated as occluded by the horizon are considered visible. When light clusters' bounding boxes intersect the projection plane, we subdivide the clusters until their children find themselves either fully in front or behind the projection plane. Many subdivisions are done over the course of one frame as every group of facades for which we evaluate visibility starts with its precut containing the root of the scene's light tree. To reduce the number of subdivisions in one frame, we cull clusters whose maximum potential irradiance is lower than a user-

defined threshold. We evaluate the potential, similarly to the Lightcuts technique, by evaluating Equation 2.1 and finding the upper bound of each term independently. Our only difference with the Lightcuts evaluation is that we also apply an upper bound on the diffuse term to 1. Since light sources have a $\frac{1}{distance^2}$ reduction in intensity, setting a low-enough threshold should cull those that are too far away while not removing many lighting details. As most lighting details come from either light sources that are close with a strong intensity or far away light sources that are combined as a cluster with strong intensity, our optimizations neglect mostly small lighting details.

Third, one of the most important optimizations present in the CHC++ [19] technique is the temporal coherence of visibility evaluations. In the parameters of the technique, the user can define the number of frames during which a visibility evaluation remains valid. Keeping a visibility evaluation valid for multiple frames enables the algorithm to spread the cost of multiple visibility evaluations over multiple frames, reducing the number of evaluations executed in a particular one. We use the same concept and apply it to our visibility evaluations, keeping the visibility precut in memory for a user-defined number of frames. By doing so, we similarly benefit from spreading the computations over multiple frames, while keeping memory usage low compared to precomputing the visibility for the whole scene.

Table 5.II shows the impact of each optimization, along with its associated speed-up (in %). The results are achieved by placing the camera at the point of view "Close-Ended Street", described in Section 5.1.1. We also set our minimum potential irradiance threshold at one quarter of one RGB ($\frac{1}{4} \times \frac{1}{255}$). We cull any light source or cluster whose maximum potential irradiance is under our threshold. The temporal coherence keeps the visibility results valid for 50 frames and the average frame time (in milliseconds) is computed over 120 frames.

5.3 Results

We analyze below the performance of our algorithm and each of its steps. The test setup consists of the camera positioned at the "Close-Ended Street" point of view pre-

Optimization	Frame time (ms)	Speed-up (%)
No optimization	613.7	-
Buildings as occluders	205.2	299.1
Culling by irradiance	178.8	114.8
Temporal coherence	47.5	376.4

Table 5.II – Impact of the optimizations on the average frame time of our pipeline. Each optimization is applied in addition to the previous ones (rows above in the table) and the speed-up is computed against the previous average frame time.

sented in Figure 5.3. The shading pass used for analysis varies based on the different steps, and is mentioned for each result.

We first present the culling achieved by the CHC++ algorithm. As the first step in our algorithm, this visibility culling aims to reduce our potentially visible set (PVS) as much as possible. Table 5.III shows, for different culling methods, how many triangles remain in our PVS and the time spent building the G-buffers in the first pass of the deferred pipeline. We use the CHC++ occlusion queries in our main pipeline, which has both the advantage of using GPU hardware for visibility and keeping the PVS information on CPU. The results do not take into account a possible future shading pass. The speed-up (close to 10×) attained by using HOQs confirms the efficiency of the technique. We believe that our choice of visibility technique is well justified by its performance and the information it gives back to the CPU which we build upon, i.e., the estimated pixel coverage of each tested BVH node.

Camera visibility technique	Triangles in PVS	Deferred first pass	Speed-up
No visibility, no ordering	978,652	102.5 ms	-
CHC++, CPU frustrum culling	41,664	11.3 ms	9.07×
CHC++, occlusion queries	2,312	1.1 ms	93.18×

Table 5.III – Impact of the visibility culling on the potentially visible set (PVS).

Then, we evaluate the efficiency of our Lightcuts GPU revision, presented in Chapter 3. As the final step in our algorithm, the Lightcuts implementation receives the light tree, which can be seen as a light-PVS.

Table 5.IV shows the efficiency of our GPU Lightcuts implementation using our

revised light tree against a light tree built according to the original technique [26]. The results highlight how our light tree is shallower compared to the original light tree and how it contains almost 20,000 fewer inner nodes.

Light tree	Node count	Tree depth	Full tree	Culled tree	Speed-up
Original light tree	78 560	16	1370.2 ms	136.6 ms	-
Our light tree	58 944	5	1267.8 ms	67.5 ms	2.02×

Table 5.IV – Comparison of performances between the original and revised light trees.

The column "Full tree" shows the shading time when every pixel is given the root of the light tree as its light-PVS. In this case, the time difference between both tests is rather small ($\approx 10\%$) compared to the total evaluation time, as the optimal structure for smaller cuts of the original light tree does not compensate the additional subdivisions imposed by the structure. Figure 5.4 compares both results and shows how similar the final cut for both trees are.

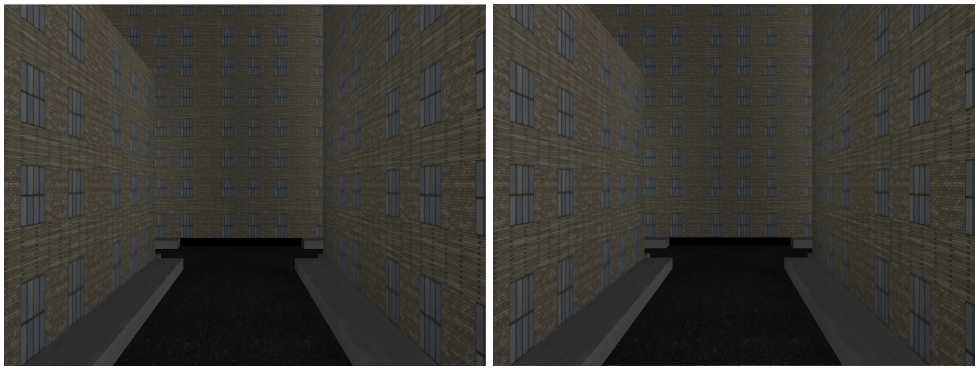
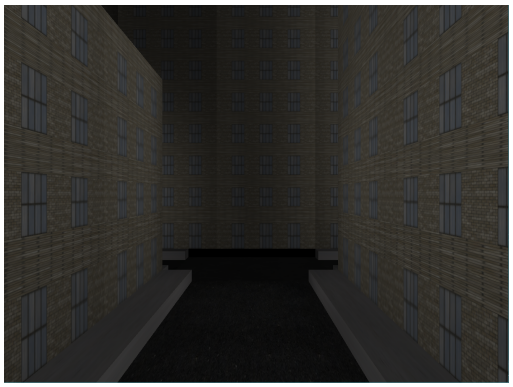


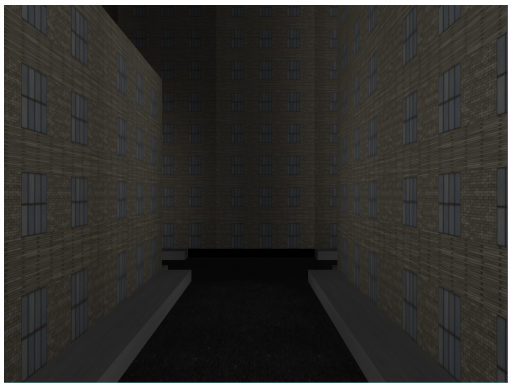
Figure 5.4 – Left: Our revised light tree. Right: The original light tree. As can be seen, both trees converge towards the same image when given the scene's light tree as light-PVS. Here, both images are completely identical.

The "Culled tree" column shows a more realistic scenario where visibility is evaluated with the light tree and only visible clusters for each facade are contained in its light-PVS. In this case, our light tree variation benefits from a significant speed-up (about half the time), mainly due to the reduction of subdivisions necessary to make the cut. The timing is evaluated by setting our error threshold to 3.921e-5, or one percent of one RGB ($\frac{1}{255} \times \frac{1}{100}$).

Figure 5.5 shows the difference between shading by making a cut in our pruned light trees or by using all the light sources in it. The error between the two images is at most 2 RGB, which can be seen as brighter white regions on the street in Figure 5.5d. Such a low error implies that when the error is enhanced for visualization, compression errors due to the image format used in the comparison is increased as well. However, we can still notice shading differences on the building facades, which highlights differences between the choice of clusters in the cut and using only the scene's light sources.

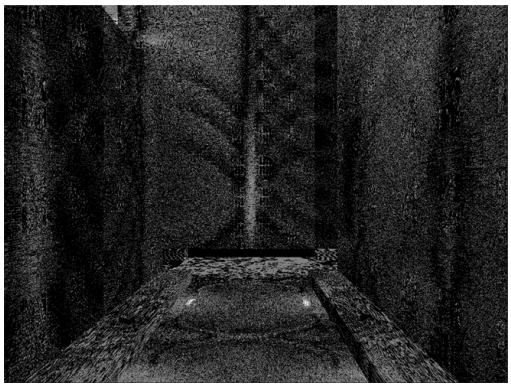


(a) Our revised cut



(b) All light sources

(c) Error image



(d) 128×Error image

Figure 5.5 – The scene rendered with our light tree pruned by visibility. (a) Shading is performed by making a cut in the light-PVS. (b) Shading is performed with every light source in the pruned tree. (c) Difference between the two images. (d) Difference amplified by a factor of 128 to better illustrate where the small errors are distributed in the image.

Finally, we present the performance of each sub-step in FCV, our visibility algorithm.

For this comparison, we keep the same point of view in the scene, but set the error threshold for the Lightcuts to 9.803e-4, or the quarter of one RGB. We also disable temporal coherence in our visibility algorithm, to analyze the full visibility evaluation. This evaluation is normally distributed over multiple frames, and consequently, does not represent time spent in a regular frame.

Algorithm step	Total time	Self time	Number of calls
1 Clusters visibility	130.7 ms	2.9 ms	1
2 Process visibility for element	127.8 ms	2.8 ms	17
3 Collect object indices	0.1 ms	0.1 ms	17
4 Prepare queues	18.5 ms	18.5 ms	53
5 Visibility evaluation	105.4 ms	0.1 ms	53
6 Evaluate clusters	96.8 ms	4.3 ms	379
7 Cull by potential test	19.8 ms	5.2 ms	56 646
8 Cluster upper bound	14.6 ms	14.6 ms	56 646
9 Last cull test	51.6 ms	42.2 ms	10 925
10 Project onto horizon	9.4 ms	9.4 ms	59 323
11 Early cull test	21.1 ms	15.7 ms	32 751
12 Project onto horizon	5.4 ms	5.4 ms	32 751
13 Reprojection	0.4 ms	0.4 ms	328
14 Projection	8.1 ms	8.1 ms	328

Table 5.V – Performance and number of calls for each step in our Facade Cluster Visibility algorithm with no temporal coherency. The column *Total time* shows the total time of the step added to the time of its hierarchical substeps. The column *Self time* presents only the time step. The sum of the time in the *Self time* column is equal to the value in first row of the *Total time* column. The column *Number of calls* presents the number of times the step was executed.

Table 5.V shows each sub-step along its execution time, the execution time of its children sub-steps, and the number of times the sub-steps were called in the frame. As can be seen in Step 2, we compute visibility for 17 BVH clusters, and collect the IDs of every surface in it to later assign it the visibility result. We subdivide these 17 clusters in 53 visibility evaluations, one for each visible face of each evaluated cluster. The average of 3.1 visible faces per cluster comes from the need to evaluate all four cardinal directions for a cluster when one surface within it has a normal pointing up. In Step 4, for each visibility evaluation, we pre-prune the light-PVS and the occluder list with the

facade, culling every light or occluder that is fully behind the plane constructed from the facade position and normal.

Each visibility evaluation then tries culling the light sources by comparing them with the horizon built from the processed occluders. The original horizon is empty and is built in Steps 13 and 14. Steps 6 to 14 are repeated until all light sources are either culled or guaranteed to be visible. Furthermore, the projection plane at which the horizon is built first starts at the facade plane and is moved to the next occluder in the visibility evaluation's ordered occluder list in Step 10. On average, each visibility evaluation executes seven iterations before terminating. It is important to note that the average is for the current point of view and can vary with different points of view. We discuss this variance throughout Section 5.4.

Step 7 tests if every cluster or light source has the potential to illuminate the facade with an intensity of more than the user-defined threshold. Of the 56,646 tests, 12,970 result in a culled light, with a cull efficiency of 22.9%. In Step 9, each light that is located between the current projection plane and the next one undergoes its last potential cull evaluation. In this test, we attempt to cull the light sources against the horizon, and subdivide partially occluded clusters, until they are either completely visible or completely occluded. Of the 10,925 tests, 59,323 light sources are projected onto the horizon and 6,753 are extracted from the sub-tree of the tested nodes to be added to the list of visible light sources. In Step 11, the last culling test, we evaluate all the light sources that are behind the next projection plane by testing them against the current horizon. The light sources are culled only if they are completely occluded, as we do not subdivide light sources at this point. In the 32,751 tests executed for Step 11, 5,418 tests result in a successful cull, with a cull efficiency of 16.5%.

If some light sources remain after attempting to cull them, in Step 13, we move the projection plane to the nearest occluder along the plane normal and reproject the current horizon onto it. Finally, in Step 14, we project every occluder that falls fully or partially before the plane onto it, in order to build the horizon. As can be noticed in Table 5.V, the projection and reprojection steps are executed an average of 6.2 times per evaluation ($\frac{328}{53}$).

The results shown in Table 5.V highlight the most costly operation in our visibility technique, i.e., testing the visibility of light sources. Of the average 1.98 milliseconds necessary to compute the visibility from a view cell in one direction, 1.82 ms are spent for the evaluation of the light clusters. The rest of time is divided by the projection of new occluders onto the projection plane (0.15 ms) and the reprojection of occluders from the previous projection plane (0.01 ms). The cost of the evaluation of clusters is divided between our three different tests to cull clusters. The *Cull by potential test* takes an average of 0.37 ms, which is spent computing the upper bounds of each tested cluster's representative light source. The *Early cull test* and the *Last cull test* respectively take 0.4 ms and 0.97 ms. These last two tests spend most of their time projecting the clusters onto the projection plane and testing coverage with the occluders projected onto it.

Finally, although the results are computed for our generated city, we should expect similar results for varying cities of similar dimensions. We achieve interactive frame rates in our city when using temporal coherency and believe that further research to optimize the technique, presented in Section 6.1, could achieve satisfying real-time results.

5.4 Scaling

When rendering a city, the position of the camera has a great impact on the number of visible surfaces. Furthermore, a larger city contains many more potential occluders. In this section, we present how our technique scales with such changes. We start by going over every step in our pipeline and looking at how they each scale with different points of view. Then, we analyze a single facade cluster visibility evaluation in progressively larger cities, to study how the increase in occluders affects performance.

Table 5.VI compares the performance of our pipeline according to different points of view. The four points of view we use, previously introduced in Table 5.I, contain 160, 1,156, 14,260 and 100,112 triangles in their potentially visible sets, representing a tenfold increase in the visible geometry between each point of view. Notable in the table is the nonlinear progression of time spent in the pipeline. Furthermore, a discrepancy can be noticed in the total time of the pipeline compared to the summed up time of its

steps. This difference comes from the fact that the shading step takes place on GPU and enables us to start earlier the next frame on CPU. Finally, the discrepancy between the shading times comes from the fact that the *Long linear street* point of view contains significantly less screen coverage by buildings while also having a smaller average light-PVS for those buildings compared to the *Close-ended street* point of view.

First, the performance of the CHC++ technique depends on the performance of draw calls on the GPU. Since our implementation does not execute instanced draw calls, the performance of the technique is affected. Second, the technique's performance also depends on the number of visible surfaces at the same time. By changing our point of view we increase by tenfold the number of simultaneously visible buildings, but by using a bounding volume hierarchy over the scene, the CHC++ technique does not scale as quickly.

Pipeline step	Point of view on city (relative to Figure 5.3)			
	(a)	(b)	(c)	(d)
Complete pipeline	22.2 ms	48.7 ms	52.9 ms	142.8 ms
Camera visibility (CHC++)	1.9 ms	4.9 ms	20.3 ms	82.2 ms
Facade cluster visibility (FCV)	1.2 ms	6.4 ms	12.9 ms	73.4 ms
Shading (Lightcuts)	19.1 ms	38.4 ms	29.7 ms	56.1 ms
Number of visible facades	1	20	190	2185
FCV Evaluations per frame	< 1	3	4	20
FCV Graphics memory usage	< 0.1 MB	0.2 MB	0.4 MB	1.8 MB

Table 5.VI – Comparison of our technique's performance across four points of view in the same city.

Since our visibility technique executes on the geometry identified as visible by the CHC++ technique, it also depends on the number of visible surfaces at the same time. Furthermore, since our technique evaluates visibility according to each view cell containing the grouped surfaces, its performance depends on their perspective view on the city. By only changing the camera's point of view, our technique only suffers in scaling from the increase of simultaneously visible buildings. In this regard, the scaling identified in Table 5.VI is very similar to the scaling of the CHC++ technique, as it should be.

To better understand the scaling of our technique when the city itself changes, we generate three cities of different sizes and analyze the changes in performance of our technique in each case. Table 5.VII shows the associated performance of the cities when the camera is positioned at the *Close-ended street* point of view. Each sub-step in our technique is averaged for the visibility evaluation of a single element in the scene. The sizes of the three cities were chosen with a twofold and threefold increase, respectively.

As can be noticed, although the increasing size of the city affects the performance of our algorithm, it does so at a lesser degree. The use of a light tree to cull light sources proves itself effective as the number of tests executed to cull light sources also does not scale linearly with the number of light sources. Finally, we notice that the number of light clusters that fall in the *Last cull test* is consistent for all the cities for the same point of view. This demonstrates that essentially, the same number of light clusters are kept visible until the last moment for each city.

Table 5.VI also highlights the low use of GPU memory by our algorithm, which was one of the main goals in our implementation. We save graphics memory by building a unique light tree for the scene, only saving the indices of the visible light clusters per surface. We also present the average number of FCV evaluations executed in a frame according to our previous setup of spreading evaluations over 50 frames. The FCV performance in the table scales according to the average number of evaluations.

For our Lightcuts implementation, the scaling depends on the resolution of the screen, the size of the cut for each surface, and the number of subdivisions necessary to achieve that cut. Since we do not change the rendering resolution in any of our tests, we know that performance differences between each point of view is a result of the different cuts required to shade the scene. This technique too does not scale linearly, since it uses a light tree to shade surfaces.

The average visibility distance for a point of view in the city depends on multiple factors. By controlling the building heights in our procedural city, we can change the average distance between the light sources and the surfaces they shine on. Furthermore, a city with fewer buildings per area also increases the average view distance. The space in each city where there is no building or occluder corresponds to buildings with no (0)

	City size					
	6×6 blocks		9×9 blocks		15×15 blocks	
Triangles in scene	978 652		2 204 572		6 124 564	
Light sources in scene	39 202		88 278		245 106	
Algorithm step	Average # of calls, Total time (ms)					
1 Clusters visibility	1	8.2	1	11.4	1	18.1
2 Process visibility for element	1	8.1	1	11.0	1	17.2
3 Collect indices	1	0.0	1	0.0	1	0.0
4 Prepare queues	3	1.1	3	1.6	3	3.0
5 Visibility evaluation	3	6.7	3	9.1	3	13.7
6 Evaluate clusters	22	6.2	28	7.9	41	11.4
7 Cull by potential test	3332	1.3	4940	1.8	8719	3.4
8 Cluster upper bound	3332	0.9	4940	1.3	8719	2.5
9 Last cull test	642	3.3	685	3.8	641	3.5
10 Project onto horizon	3484	0.6	3826	0.6	3644	0.6
11 Early cull test	1926	1.5	2936	2.1	5669	3.8
12 Project onto horizon	1926	0.4	2936	0.5	5669	1.0
13 Reprojection	19	0.0	25	0.0	38	0.1
14 Projection	19	0.5	25	1.1	38	2.2

Table 5.VII – Comparison of the performance for a single visibility evaluation with our FCV technique between increasingly larger cities.

floor.

Table 5.VIII shows the average frame time when rendering our city without temporal coherence. Each result is obtained by placing the camera at the *Long linear street* point of view. Our setup with no variance forces the height of all buildings at seven floors, while our setup with variance selects, with equal probability, from one to seven floors for each building, which gives the distribution of heights an average value and a variance of 4. The third setup keeps the same variation of building heights, but reduces the number of FCV evaluations per frame to the same number as our setup without variance. We reduce the number of FCV evaluations per frame by increasing the threshold of pixel coverage required to build the view cells used in our technique.

Figure 5.6 shows the scaling of cities of different sizes when we change the variance in them. We use the same variance as for Table 5.VIII but do not enforce an equal

number of evaluations between both. The results are generated by placing the camera at our *Close-ended street* point of view, and by setting the temporal coherence so that visibility evaluations are kept for 50 frames.

Scene setup	Frame time	Notes
No variance	178.8 ms	All buildings have 7 floors. 100 FCV evaluations per frame.
With variance	365.2 ms	Buildings have 1 to 7 floors with equal probability. 120 FCV evaluations per frame.
Equal evaluations	317.9 ms	Buildings have 1 to 7 floors with equal probability. 100 FCV evaluations per frame.

Table 5.VIII – Increasing the variance in building heights increases the average distance light can travel in our scene, and thus reduces the efficiency of our technique. Both setups are compared with the same number of FCV evaluations per frame and the same pixel coverage threshold to build the view cells.

By culling light sources, our technique achieves a form of shadowing for the surfaces. While it does not compute shadows per pixel, soft shadows can be seen on the building facades. The quality of computed shadows depends on the choice of view cells. We can increase shadow details by computing visibility for smaller view cells, but by doing so, we also increase the number of FCV evaluations per frame.

Figure 5.7 shows two results of shadows on our buildings. The same buildings are subdivided in different view cells for both images. On the left, the facades cannot be subdivided into smaller view cells. On the right, we let facades be subdivided into smaller view cells. We study the building in the center of both images more precisely. In the left image, since our visibility technique is conservative, the light sources seen from the top floor of the building are also considered visible for lower floors. In the right image, since we subdivide the view cell further, we can notice that each floor under the top-most floor gets darker. Since the surrounding buildings offer more occlusion on these lower floors, more light sources are pruned from their light tree. Furthermore, a sharp shadow discontinuity can be noticed at the floor edges. When the view cell of a floor contains a different light-PVS than an adjacent floor, a shading disparity appears at the edges.

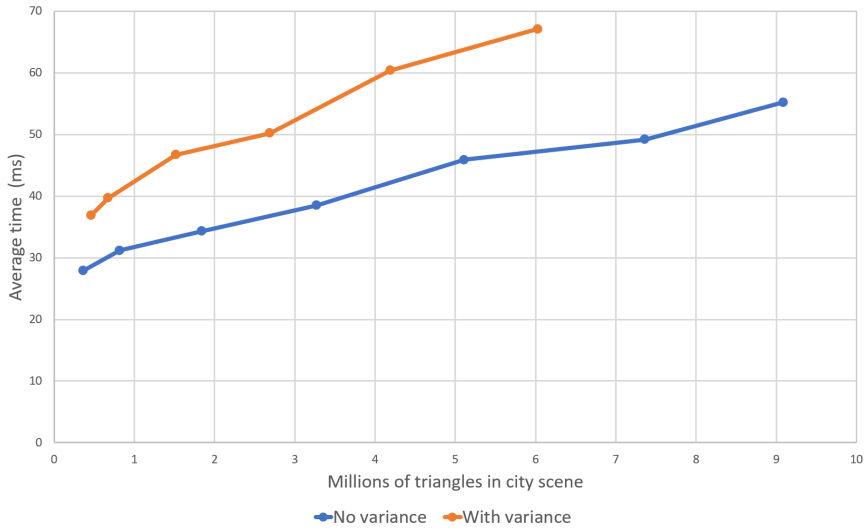


Figure 5.6 – Changing the variance in building heights affects the scaling of our technique. Both results are computed by generating procedural cities of varying sizes while using the same degrees of variance as in Table 5.VIII.

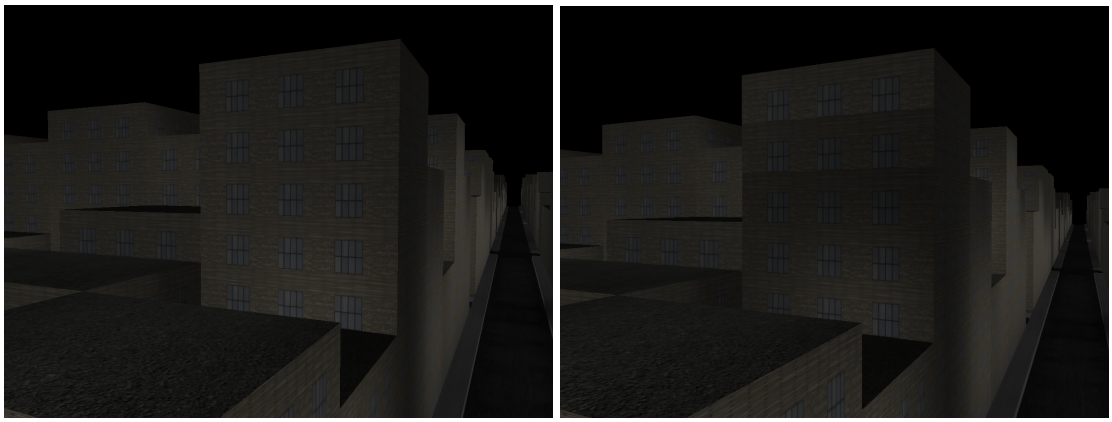


Figure 5.7 – By changing the size of view cells, our technique can refine shadowing details. On the left, we build view cells by forcing facades not to be subdivided into smaller view cells. On the right, we subdivide view cells further. As can be noticed, the subdivision of view cells gives better shadow details.

CHAPTER 6

CONCLUSION

Rendering city scenes at night proves to be a challenge to modern real-time rendering engines, as the time required to shade each pixel is dependent on the number of light sources shining on the surface it contains.

One offline rendering technique to tackle such scenes is to build a hierarchy over the light sources and to traverse it to determine a subset of light clusters and light sources that represent, as closely as possible, the total illuminance that all the light sources would add up to. This process can be done for each pixel in the image.

For real-time rendering, this process is too slow to produce an image. Since those rendering engines normally tend to render 30 to 60 frames per second, each image can take no more than 16.6 to 33.3 milliseconds to generate. Therefore, real-time rendering engines normally limit drastically the number of light sources that can be used simultaneously in a scene.

In this thesis, we presented a new pipeline to render a city while taking into account all its light sources. In a preprocess, we build a bounding volume hierarchy (BVH) of our scene's geometry, as well as a light tree over all the scene's light sources. Our pipeline benefits from the CHC++ [19] implementation of GPU occlusion queries to compute an estimated pixel coverage of every node in our scene's BVH. Then, our facade-cluster visibility (FCV) technique traverses the BVH and groups together visible nodes to satisfy a user-defined threshold of pixel coverage. Each group is then converted into a view cell for which we compute visibility of the scene's light tree, pruning light tree nodes that are not visible from the view cell. Finally, we shade each pixel's surface with our Lightcuts [26] GPU adaptation while starting the cut from the pruned light tree precomputed for the view cell containing the surface.

Our contributions to the field are our new visibility technique, which is tailored to evaluate visibility in dense city scenes with large occluders, and our adaptation of the Lightcuts technique for real-time rendering on GPU. We achieved interactive frame rates

(7-50 frames per second, see Table 5.VI) while rendering our city at night with all light sources found in it.

6.1 Future Work

Many optimizations and future work could be added to the technique to increase its performance and accuracy.

As visibility evaluation for each view cell is independent and can be run simultaneously, implementing multithreading in our visibility technique is guaranteed to improve the speed at which it prunes the light trees. We expect that such an implementation could double the performance of our visibility computation and should be easy enough to implement.

Instancing draw calls in our deferred pipeline could greatly improve performance, but was not implemented because it would not affect our visibility technique. This feature in itself is easy to implement but would require a larger refactoring of our CHC++ implementation.

Our light tree currently only contains static light sources (e.g., windows and street lights) found in the city scene. We could build a second light tree containing the moving light sources (e.g., car headlights), and prune both light trees at the same time. Since this tree would normally be much smaller than our static light tree, we believe we could rebuild or rearrange it at every frame to keep it optimized for our technique.

By executing the visibility pruning of the light tree on CPU, we compute a precise upper bound on the lighting that each surface in the view cell can receive. We believe that the level of detail (LOD) for the geometry within those view cells could depend on the maximum amount of light shining on them. Doing so would let us reduce details in poorly lit alleys. However, since our facade-cluster visibility evaluation is executed after the first pass of our deferred pipeline, this information could only be used for the next frame(s). Furthermore, if a surface is poorly lit during a frame, but is well lit during the next one (e.g., a car turning on its headlights), the lower LOD would be visible for one frame, even though it is well lit. To allow such information to be used for the current

frame would require to change the global pipeline of our technique.

In our facade-cluster visibility technique, since our occluders are 2.5D in nature, we compute the horizon of the occluders viewed from the top of the view cell. Although it guarantees that every light source visible from that view cell will not be culled, we lose much of the lighting variations inside the view cell. While decreasing the size and increasing the number of view cells would increase lighting details visible in the image, it would also very quickly increase the number of visibility evaluations. Another option would be to compute the height at which light sources and clusters are culled on the view cell, and save this height in the view cell precut. By doing so, we could cull clusters during the Lightcuts evaluation, as each pixel could compute its height in the view cell.

Still in our FCV technique, we currently project the light clusters onto the projection plane every time that we test them against the projected occluders. Since the supporting planes of the clusters do not change over the frame, we could build them once and reproject them with the occluders every time we move the projection plane. Such a modification should be quick and slightly reduce the number of projections done per frame. We believe it would approximately increase the performance of the visibility evaluation by 5%.

Finally, our results did not prove efficient enough to be included into a real-time rendering engine, but it achieved interactive frame rates and good scaling behavior for large cities with many light sources. We believe that the visibility technique we presented opens up new research ideas, and could be optimized to be useful in real-life scenarios, such as video game engines or city design software.

BIBLIOGRAPHY

[1] BVH image. https://en.wikipedia.org/wiki/Bounding_volume_hierarchy#/media/File:Example_of_bounding_volume_hierarchy.svg. Accessed: 2018-10-04.

[2] City at night picture. https://www.pexels.com/photo/city-glass-architecture-windows-34136/. Accessed: 2018-11-17.

[3] Mipmap illustration. https://upload.wikimedia.org/wikipedia/commons/e/ed/Mipmap_illustration2.png. Accessed: 2018-10-04.

[4] Painter's algorithm. https://en.wikipedia.org/wiki/Painter's_algorithm. Accessed: 2018-03-08.

[5] Jiri Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 2004. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2004.00793.x.

[6] N. Bus, N. H. Mustafa, and V. Biri. Illuminationcut. *Comput. Graph. Forum*, 34 (2):561–573, May 2015. ISSN 0167-7055. doi: 10.1111/cgf.12584.

[7] Jennifer Chandler, Lei Yang, and Liu Ren. Procedural window lighting effects for real-time city rendering. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, i3D '15, pages 93–99. ACM, 2015. ISBN 978-1-4503-3392-4. doi: 10.1145/2699276.2699290.

[8] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 7 2003. ISSN 1077-2626. doi: 10.1109/TVCG.2003.1207447.

[9] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97. ACM, 1997. ISBN 0-89791-884-3. doi: 10.1145/253284.253312.

[10] Adrian Courrèges. Doom (2016) - graphics study. `http://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/`. Accessed: 2018-03-08.

[11] Tomáš Davidovič, Iliyan Georgiev, and Philipp Slusallek. Progressive lightcuts for gpu. In *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12. ACM, 2012. ISBN 978-1-4503-1683-5. doi: 10.1145/2343045.2343047.

[12] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 21–30. ACM, 1988. ISBN 0-89791-275-6. doi: 10.1145/54852.378468.

[13] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 239–248. ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 1-58113-208-5. doi: 10.1145/344779.344891.

[14] Dorian Gomez, Mathias Paulin, David Vanderhaeghe, and Pierre Poulin. Time and space coherent occlusion culling for tileable extended 3D worlds. In *2013 International Conference on Computer-Aided Design and Computer Graphics, CAD/Graphics 2013*, pages 1–8, 2013. URL `https://doi.org/10.1109/CADGraphics.2013.9`.

[15] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive*

Techniques, SIGGRAPH '93, pages 231–238. ACM, 1993. ISBN 0-89791-601-8. doi: 10.1145/166117.166147.

[16] Takahiro Harada, Jay McKee, and Jason C. Yang. Forward+: Bringing deferred lighting to the next level. In Carlos Andujar and Enrico Puppo, editors, *Eurographics 2012 - Short Papers*. doi: 10.2312/conf/EG2012/short/005-008.

[17] Miloš Hašan, Fabio Pellacini, and Kavita Bala. Matrix row-column sampling for the many-light problem. *ACM Trans. Graph.*, 26(3), July 2007. ISSN 0730-0301. doi: 10.1145/1276377.1276410.

[18] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4): 143–150, August 1986. ISSN 0097-8930. doi: 10.1145/15886.15902.

[19] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. CHC++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Proceedings Eurographics 2008)*, 27(2):221–230, April 2008. ISSN 0167-7055. URL `https://www.cg.tuwien.ac.at/research/publications/2008/mattausch-2008-CHC/`.

[20] Ola Olsson, Markus Billeter, and Ulf Assarsson. Clustered deferred and forward shading. In *High Performance Graphics*, 2012.

[21] Jiawei Ou and Fabio Pellacini. Lightslice: Matrix slice sampling for the many-lights problem. *ACM Trans. Graph.*, 30(6):179:1–179:8, December 2011. ISSN 0730-0301. doi: 10.1145/2070781.2024213.

[22] Eric Paquette, Pierre Poulin, and George Drettakis. A light hierarchy for fast rendering of scenes with many lights. *Computer Graphics Forum*, 17(3):63–74, 1998. ISSN 1467-8659. doi: 10.1111/1467-8659.00254.

[23] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 197–206. ACM, 1990. ISBN 0-89791-344-2. doi: 10.1145/97879.97901.

[24] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*, pages 229–238, New York, NY, USA, 2000. ACM. doi: 10.1145/344779.344886.

[25] Jeremiah van Oosten. Deferred vs forward+ rendering. `https://www.3dgep.com/forward-plus/`. Accessed: 2018-03-08.

[26] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Lightcuts: A scalable approach to illumination. *ACM Trans. Graph.*, 24(3):1098–1107, July 2005. ISSN 0730-0301. doi: 10.1145/1073204.1073318.

[27] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Implementing lightcuts. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05. ACM, 2005. doi: 10.1145/1187112.1187177.

[28] Bruce Walter, Adam Arbree, Kavita Bala, and Donald P. Greenberg. Multidimensional lightcuts. *ACM Trans. Graph.*, 25(3):1081–1088, July 2006. ISSN 0730-0301. doi: 10.1145/1141911.1141997.

[29] Bruce Walter, Pramook Khungurn, and Kavita Bala. Bidirectional lightcuts. *ACM Trans. Graph.*, 31(4):59:1–59:11, July 2012. ISSN 0730-0301. doi: 10.1145/2185520.2185555.