Universitat de Girona

GOOD-VISIBILITY COMPUTATION USING GRAPHICS HARDWARE

Narcís MADERN LEANDRO

ISBN: 978-84-693-9851-7
Dipòsit legal: GI-1365-2010
http://www.tdx.cat/TDX-1026110-114006

Universitat de Girona

PhD thesis:

Good-Visibility Computation using Graphics Hardware

Narcís Madern Leandro

2010

Programa de Doctorat de Software

PhD supervisors:
Dr. Joan Antoni Sellarès Chiva
Dr. Narcís Coll Arnau

Memòria presentada per a optar al títol de Doctor per la Universitat de Girona

Universitat de Girona

Dr. Joan Antoni Sellarès Chiva i Dr. Narcís Coll Arnau, Titulars d'Universitat del Departament d'Informàtica i Matemàtica Aplicada de la Universitat de Girona,

CERTIFIQUEM:

Que aquest treball titulat "Good-Visibility Computation using Graphics Hardware", que presenta en Narcís Madern Leandro per a l'obtenció del títol de Doctor, ha estat realitzat sota la nostra direcció.

Signatura

Prof. Joan Antoni Sellarès Chiva Prof. Narcís Coll Arnau

Girona, 15 de Juliol de 2010

a la mama, al papa i a l'Aida

a la iaia Maria i a l'avi Rufino,

a la Sílvia.

Abstract

In this thesis we design, implement and discuss algorithms that run in the graphics hardware for solving visibility and good-visibility problems. In particular, we compute a discretization of the multi-visibility and good-visibility maps from a set of view objects (points or segments) and a set of obstacles. This computation is carried out for two-dimensional and three-dimensional spaces and even over terrains, which in computational geometry are defined as a 2.5D space.

First, we thoroughly review the graphics hardware capabilities and how the graphics processing units (known as GPUs) work. We also describe the key concepts and the most important computational geometry tools needed by the computation of multi-visibility and good-visibility maps. Afterwards, we study in a detailed manner the visibility problem and we propose new methods to compute visibility and multi-visibility maps from a set of view objects and a set of obstacles in 2D, 2.5D and 3D using the GPU. Moreover, we present some variations of the visibility to be able to deal with more realistic situations. For instance, we add restrictions in the angle or range to the visibility of viewpoints or we deal with objects emitting other kinds of signals which can cross a certain number of obstacles.

Once the multi-visibility computation is explained in detail, we use it together with the depth contours concept to present good-visibility in the two-dimensional case. We propose algorithms running in the GPU to obtain a discretization of the 2D good-visibility map from a set of view objects and a set of obstacles. Related to the view objects, we present two alternatives: viewpoints and view segments. In the case of the obstacles we also expose two variants: a set of segment obstacles or an image where the color of a pixel indicates if it contains an obstacle or not. Then we show how the variations in the visibility change the good-visibility map accordingly. The good-visibility map over a terrain is explained as a variation of the 2D version, since we can first compute it in the plane by using a projection and then re-project again the solution to the faces of the terrain.

We finally propose a method that using the graphics hardware capabilities computes the depth contours in a three-dimensional space in a fast and efficient manner. Afterwards, a set of triangle obstacles is added to the previously mentionned set of (view) points in order to compute a discretization of the good-visibility map in the three-dimensional space.

Resum

Aquesta tesi tracta del disseny, implementació i discussió d'algoritmes per resoldre problemes de visibilitat i bona-visibilitat utilitzant el hardware gràfic de l'ordinador. Concretament, s'obté una discretització dels mapes de multi-visibilitat i bona-visibilitat a partir d'un conjunt d'objectes de visió i un conjunt d'obstacles. Aquests algoritmes són útils tant per fer càlculs en dues dimensions com en tres dimensions. Fins i tot ens permeten calcular-los sobre terrenys.

Primer de tot s'expliquen detalladament les capacitats i funcionament de les unitats de processament gràfic (GPUs) i els conceptes i eines clau de la geometria computacional necessaris per calcular mapes de bona-visibilitat. Tot seguit s'estudia detalladament el problema de la visibilitat i es proposen nous mètodes que funcionen dins la GPU per obtenir mapes de visibilitat i multi-visibilitat a partir d'un conjunt d'objectes de visió i un conjunt d'obstacles, tant en 2D i 3D com sobre terrenys. A més a més es presenten algunes variacions de la visibilitat i així ser capaços de tractar situacions més reals. Per exemple, afegim restriccions en l'angle o el rang de la visibilitat dels punts de visió. Fins i tot es mostren formes de canviar el tipus de senyal que emeten els objectes, que pot atravessar un cert nombre d'obstacles abans de desaparèixer.

Una vegada s'ha explicat amb detall com calcular els mapes de visibilitat, podem utilitzar-ho juntament amb els mapes de profunditat per presentar la bona-visibilitat en el pla. Es proposen alguns algoritmes que s'executen dins el hardware gràfic per obtenir una discretització del mapa de bona-visibilitat en el pla a partir d'un conjunt d'objectes de visió i un conjunt d'obstacles. Pel que fa els objectes de visió, es presenten dues alternatives: punts de visió i segments de visió. En el cas dels obstacles també es proposen dues variants: un simple conjunt de segments o una imatge binària on el color de cada píxel indica si hi ha obstacle o no. Més endavant es mostra com les variacions en la visibilitat canvien també el mapa de bona-visibilitat. Els mapes de bona-visibilitat sobre terrenys s'expliquen com una variació de la versió en el pla, ja que podem calcular-ho en el pla i tot seguit projectar la solució de nou sobre els polígons del terreny.

No es té constància de cap algoritme que calculi mapes de profunditat a l'espai, per tant es proposa un mètode que, utilitzant la GPU, obté mapes de profunditat a l'espai d'una manera ràpida i eficaç. Finalment, s'afegeix un conjunt d'obstacles per poder calcular una discretització dels mapes de bona-visibilitat a l'espai.

Acknowledgements

És poc menys que impossible donar les gràcies, per escrit, a tots els que en algun moment m'han ajudat en el procés d'escriptura de la tesi. Des dels companys i professors que vaig tenir durant la carrera, fins als meus directors de tesi, passant pels companys dels cursos de doctorat, els amics i la família. A tots vosaltres us dono les gràcies per ajudar-me i confiar en la meva feina.

Voldria començar els agraïments donant les gràcies als meus directors de tesi, en Toni i en Narcís. Encara que algunes vegades hem discutit i fins i tot ens hem enfadat, entenc que sempre ha estat per millorar la tesi o intentar publicar els articles a les millors revistes i congressos. A més a més, vosaltres vau ser els que em vau donar l'oportunitat de començar a fer recerca, primer amb un contracte quan encara estava estudiant la carrera i, més endavant, donant-me la possibilitat de gaudir d'una beca per poder fer el doctorat. Així doncs, us agraeixo molt el suport, l'ajuda i la confiança que heu tingut en mi i en que aquesta tesi finalment arribaria a bon port.

També he d'agrair tota l'ajuda que m'han donat els meus companys (i ex-companys) de despatx i en general de tot el grup. En especial, m'agradaria donar les gràcies a la Marta i en Yago pel seu suport matemàtic, a la Tere, la Marité i en Nacho per la seva ajuda en temes algorísmics i d'implementació, i a tots ells a més de tots els que no he mencionat per les xerrades durant els dinars o berenars, per les festes, pels sopars (en especial els del Bar Padules), pels partits de bàsquet o de pàdel, i tants altres bons moments viscuts.

No em vull oblidar tampoc de la gent de l'institut de química computacional que, encara que no per temes de recerca, sí que he passat molts bons moments amb ells. Han estat molts sopars, events esportius diversos, i fins i tot viatges, els que hem compartit.

Evidentment, vull dedicar unes paraules a la meva família, als meus pares, la meva germana, els meus avis, oncles, "tios", ties i cosins, i també a en Miquel i la Marta. Moltes gràcies per recolzar-me sempre i ajudar-me en tot el que estava a les vostres mans, incondicionalment, tan en els bons com en els mals moments.

M'agradaria donar les gràcies als meus amics de "tota la vida", en Marc, en Jaume i l'Elisa, i a d'altres que fa menys temps que conec, però que igualment aprecio molt: en Sergi, la Marta, en Juanma i l'Anna. Gràcies per ser com sou i espero que poguem continuar veient-nos i fent coses junts durant molt de temps.

Finalment, vull dedicar aquesta tesi especialment a la Sílvia. Estic segur que sense

tu no hagués estat possible acabar la tesi sense tornar-me boig. Moltes gràcies per tota l'ajuda que m'has donat sempre, en qualsevol moment i sobre qualsevol aspecte de la feina. Moltes gràcies per les correccions de l'anglès, per llegir-te una "aburrida" tesi de geometria computacional els cops que han fet falta. Moltes gràcies per ser com ets i per estar al meu costat i recolzar-me en tot moment i en qualsevol circumstància.

Published work

In this chapter we expose the articles published during this PhD thesis. The papers are sorted by date of publication and they include journal articles as well as papers published in conference proceedings.

Publications related to this thesis

- N. Coll, N. Madern, J. A. Sellarès Three-dimensional Good-Visibility Maps computation using CUDA (in preparation)
- N. Coll, N. Madern, J. A. Sellarès Parallel computation of 3D Depth Contours using CUDA (submitted to Journal of Computational and Graphical Statistics)
- N. Coll, N. Madern, J. A. Sellarès Good-visibility Maps Visualization The Visual Computer, Vol. 26, Num. 2, pp 109-120, 2010.
- N. Coll, N. Madern, J. A. Sellarès Drawing Good Visibility Maps with Graphics Hardware Computer Graphics International, pp 286-293, 2008.
- N. Coll, N. Madern and J. A. Sellarès Good Visibility Maps on Polyhedral Terrains 24th European Workshop on Computational Geometry, pp 237-240, 2008.
- N. Coll, M. Fort, N. Madern, J.A. Sellarès GPU-based Good Illumination Maps Visualization Actas XII Encuentros de Geometría Computacional, Valladolid, Spain, pp 95-102 , 2007.
- N. Coll, M. Fort, N. Madern and J.A. Sellarès Good Illumination Maps 23th European Workshop on Computational Geometry, pp 65-68, 2007.

List of Figures

List of Algorithms

Contents

Chapter 1

Introduction

In this thesis, we solve visibility and good-visibility problems by using Computational Geometry and Graphics Hardware techniques.

Computational Geometry is a relatively new discipline which aims to investigate efficient algorithms to solve geometric-based problems. Consequently, it is essential to identify and study concepts, properties and techniques to ensure that new algorithms are efficient in terms of time and space. For instance, the complexity of algorithms and the study of geometric data structures are important concepts to consider. Computational Geometry problems can be applied to a wide range of disciplines such as astronomy, geographic information systems, data mining, phisics, chemistry, statistics, etc.

One of the most important and studied research topics in computational geometry is visibility which, from a geometric point of view, is equivalent to illumination. Regarding this illumination or visibility concept, some questions naturally arise, for instance (1) which zones of the space are visible from a set of points taking into account the obstacles? (2) are all these visible zones connected or are all the interior points of a certain object visible from a known point of observation? In addition, one can even ask questions related to the inverse problem, for example how many points are needed to directly view all the zones of a building and where they have to be placed. An increasing number of studies related to visibility in computer graphics and other fields have been published, including its problems and their solutions. Extensive surveys on visibility can be found in [Dur00] and [COCSD03].

In practice, when dealing with an environmental space of viewpoints and obstacles, it is some times not sufficient to have regions simultaneously visible from several viewpoints but it is necessary that these regions are well-visible, i.e. that they are surrounded by

viewpoints. This concept, known as *good-illumination* or *good-visibility*, was described for the first time in the PhD Thesis of S. Canales in 2004 [Can04]. It can be seen as a combination of two well studied problems in computational geometry: visibility and *location depth* [KSP$^+$03, MRR$^+$03]. Let V be a set of points in the plane or space, the location depth of a point p indicates how deep p is with respect to V. The depth map of V shows how deep is every point of the space with respect to V. Intuitively, we can say that the more interior a region is with respect to V, the more depth it has. Thus points inside a more deeper region have more points of V surrounding it (see Figure 1.1).

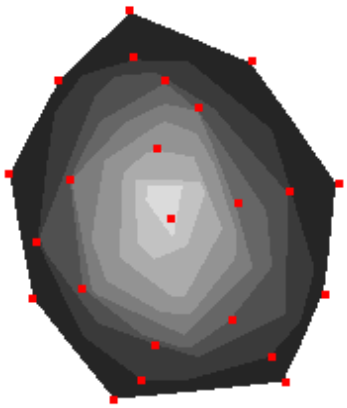
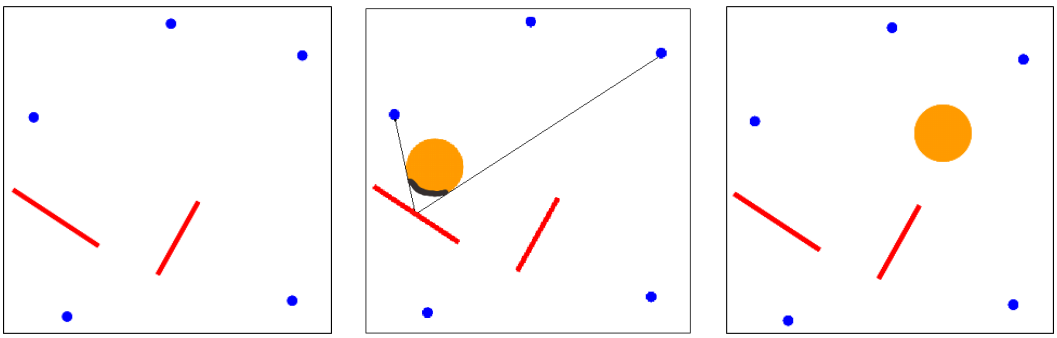


Figure 1.1: *Example of a depth map from a set of points V. The lighter the gray of a region is, the deeper it is situated with respect to V.*

Good-visibility is based on the same intuitive idea of location depth with the addition of obstacles which can block the visibility in certain zones. Taking this into account, good-visibility can be treated as a generalization of the location depth concept by adding visibility information from the set V. A point p is *well visible* if the main part of the viewpoints in V are *well distributed* around p and visible from there. Otherwise it is not well visible if the main part of the viewpoints visible from p are grouped on the same *side* of p. Figure 1.2 contains a scheme showing this intuitive idea. A scene with five viewpoints and two segment obstacles is depicted in (a). It is important to remark that every point in the plane is visible from at least one viewpoint. Nevertheless, when a convex object is placed in the scene it acts as a barrier and it is possible that some points on its boundary become invisible to every viewpoint (b). Of course, this effect can be avoided if the object is placed at a *better* position (c).

Abellanas, Canales and coworkers published some other relevant work about good-visibility, providing its exact calculation in concrete cases [ACH04, ABM07b], and even some variations were also published [ABHM05, ABM07a]. However, to the best of our



(a) (b) (c)

Figure 1.2: *Good-visibility concept illustration.*

knowledge a generic solution to compute good-visibility from any set of viewpoints and any set of obstacles has still not been found.

The *good-visibility map* is defined as the subdivision of the plane in regions with different good-visibility depth. Good-visibility maps and their extensions can have many applications in a wide variety of fields, i.e. in construction and architecture, the position of lights in buildings and wireless points or antenna distribution over a terrain or inside an office. In particular, they are useful in the design of the position, orientation and number of lights inside an art gallery in order to obtain the best *quality of light* to all exposed pieces (see Figure 1.3). Another practical application is for deciding where a set of WiFi access points have to be placed to obtain a good signal in the main part of a building taking also into account the number of walls that their signal can cross [YW08, DC08, AFMFP+09] (see Figure 1.4). They also have many applications in 2.5D (also known as terrains) and 3D cases. For instance, a good-visibility map over a terrain can be useful to place a set of antennas in order to obtain the largest number of zones with good signal, or to ensure that a particular region of the terrain will have a good enough signal.

Problems related to visibility and good-visibility have a high computational complexity in terms of time and space. Thus, computing a discretization of the solution might still bring accurate solutions to the problem with the advantage of having a substantial reduction of the resources needed. Moreover, if a discretized solution is sufficient, it is also possible to compute each part of this discrete solution in a parallel way. it is exactly at this point where graphics hardware comes into play. Graphics Processing Units (GPUs) are specialized processors which use a highly parallel structure that makes them perfect for solving problems that can be partitioned into independent and smaller parts.

GPUs have evolved tremendously in the last years, mainly due to the decreasing prices of the electronic parts and the increasing demand for real-time graphical effects in video-

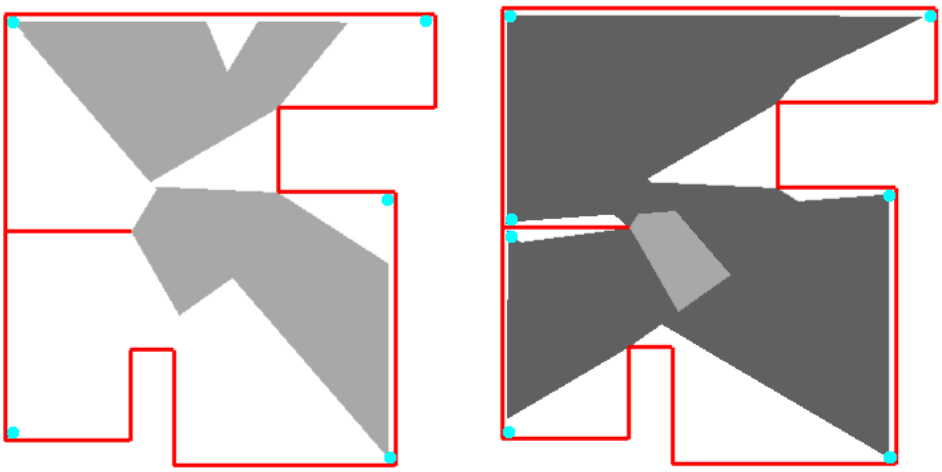


Figure 1.3: *The two images show the good-visibility map from two distinct set of viewpoints. The red lines represent the plant of an art gallery while the blue points represent light focus. The good-visibility map is painted in a gray gradiation: the darker the gray is, the higher the level of that point is.*

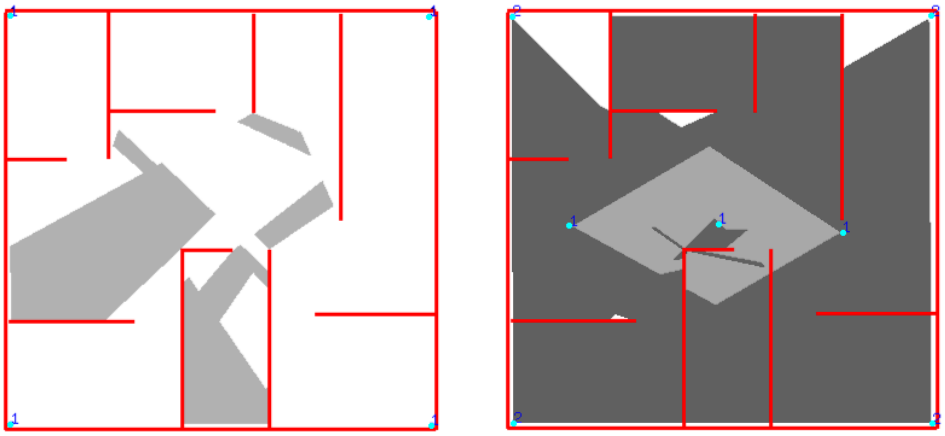


Figure 1.4: *The two images show the good-visibility map from two distinct set of wireless access points. The red lines represent walls and the blue points represent the wireless access points. The number associated to each point indicates its* power of emission *(the number of walls that its signal can cross before disappearing).*

games. In a few years the GPU has evolved from a non-modifiable black box capable of doing fast computations related to computer graphics to flexible and programmable units able to execute algorithms and solve problems belonging to a large variety of fields. Currently, the use of GPUs has now been extended to a wide range of disciplines.

1.1 Objectives

It is important to remark that the first and probably the most important aspect to consider when one wants to use graphics hardware capabilities is based on how the GPU might be programmed. Thus it is also important to know its limitations. This is usually a hard learning process if one tries to solve problems not related to computer graphics by using graphics hardware. This is the case of computational geometry problems which usually use sequential algorithms for solving them.

Since good-visibility computation needs to compute the visibility as part of the whole process, *the first goal of this PhD thesis is to develop and implement an algorithm for computing the discretization of a multi-visibility map from a set of view objects and a set of obstacles using the graphics hardware capabilities.* A visibility map is a subdivision of the space in visibility regions from a viewpoint p, where visible and non-visible regions from p can be identified. A multi-visibility map is a combination of two or more visibility maps that are obtained from two or more different viewpoints. Using the multi-visibility map we can obtain different information about the visibility according to the problem we are dealing with. We present algorithms for computing multi-visibility maps in the plane, in the space and even on a terrain.

Once we know how the multi-visibility map can be obtained, *the second objective of this thesis is the design of an algorithm capable of computing the two-dimensional good-visibility map from a set of viewpoints and a set of segment obstacles running in the GPU and taking advantage of its parallel processing capabilities.* We also present a solution to compute good-visibility maps on a terrain from a set of viewpoints, where its faces take on the role of obstacles.

Of course, improvements to the latter algorithm can also be introduced. Therefore, as *a third goal we want to compute the good-visibility map from another kind of input, for example segments or polygons instead of viewpoints and more complex obstacles, even non-geometric ones like images where the obstacles are represented by the color of their pixels.*

Once the good-visibility map in the two-dimensional space and on terrains is presented, *we want, as a fourth objective, to obtain good-visibility and some of its variations in* $\mathbb{R}^3$. Since no implementations exist for the visualization of the depth contours from a set of viewpoints in a three-dimensional space, we also want to compute it with the help of the GPU.

1.2 Structure of this thesis

With the aim of making this thesis self-contained, all necessary geometric concepts and a detailed explanation of the graphics hardware are thoroughly described in Chapter 2.

Chapter 3 is dedicated to the visibility and multi-visibility map computation using the GPU. First of all we introduce the previous work on this topic and afterwards we present our own algorithms for computing the discretized 2D, 2.5D and 3D multi-visibility map from a set of viewpoints. We also present some variations on the visibility, some of them related to the shape of the view objects (viewpoints or view segments) and others related to the obstacles. Moreover we present restrictions on the visibility as well as view objects and obstacles with attributes affecting the multi-visibility map. Finally a running time analysis for some of the exposed cases is given.

Once the computation of the visibility has been presented, we can focus on the problem of good-visibility. In Chapter 4 we expose our proposed methods to compute the good-visibility map from a set of viewpoints and a set of segment obstacles in the plane. Moreover we describe how good-visibility is affected by the variations applied to the visibility and how they can be obtained. This chapter also contains how the two-dimensional good-visibility computation can be adapted to obtain the good-visibility map over a terrain. Some examples, images and running time analysis to complete the study are also included.

Chapter 5 is focused on the computation of depth contours and good-visibility in a three-dimensional space. First of all a detailed explanation of how the depth contours can be computed from a set of 3D points is given, and in the second part of the chapter an extension of the algorithms is presented in order to deal with scenes containing viewpoints and also obstacles and to be able to finally compute volumetric good-visibility maps.

The last chapter included in this thesis draws the most important conclusions, some final remarks and the related future work.

Let us finally mention that several results from this thesis have been published in journals and conference proceedings [CFMS07a, CFMS07b, CMS08a, CMS08b, CMS10]. Moreover, the article entitled *Parallel computation of 3D Depth Contours using CUDA* has been submitted to Journal of Computational and Graphical Statistics and a paper about the computation of 3D good-visibility maps is in preparation.

Chapter 2

Background

In this chapter a detailed explanation of the existent Graphics Hardware programming paradigms is explained. In addition to this, the most important geometric concepts used in the computation of the good-visibility are thoroughly described.

2.1 The Graphics Hardware

This thesis could not be carried out without the knowledge of the principles and capabilities of current GPUs, since the main idea is to make use of the GPU not only to solve unsolved geometric problems but also to program faster algorithms to the problems which already have a CPU implementation.

Graphics Processing Units (GPU) have long been used to accelerate gaming and 3D graphics applications. In the past, the structure of the GPU programming was always the same: the programmer sent basic primitives like polygons, points or segments as a set of vertices and set the lights position and the perspective desired by using a graphic environment (i.e OpenGL). The graphics card was responsible for rendering all this primitives taking into account the parameters chosen previously. In fact, the GPU could be seen as a black box with some basic controls providing an input for the geometry and an output for its visualization. A few years later, the GPUs incorporated some programmable parts in this still inflexible graphic pipeline. With these slightly modifiable parts, called vertex shaders and pixels shaders, the programmers were able to look into the black box and change a little the path followed by the geometry before it is rendered. At present there are a lot of researchers interested in using the tremendous performance of the GPUs to do computations not necessarily connected with computer graphics topics. There is already

work related to mathematics, physics, chemistry and other disciplines which used GPUs as a parallel general purpose computer. These GPGPU (General-Purpose computation on GPUs) systems produced some impressive results, although there are many limitations and difficulties in doing generic calculations by using programming languages oriented to computer graphics, like OpenGL and Cg. To overcome these kinds of problems, NVIDIA developed the CUDA programming model. In the following sections the basics of these two programming paradigms are explained.

Since the GPUs are in constant evolution, every new graphics card that hits the market is more powerful than previous ones, thus the running time for algorithms using the graphics hardware can be reduced a lot every time a new generation of GPUs appear. Their price and parallel computation possibilities makes them a fantastic tool for improving running times in a lot of fields.

It is important to remark that, probably, the most difficult part in programming the graphics hardware relies on learning the philosophy of the GPU programming: i.e. what are the problems that can be solved using GPU? When is it better to use GPU instead of CPU? Is it always better to use CUDA instead of Cg for implementing GPGPU algorithms? Then in those cases where it is necessary to use Cg, how a geometric problem can be transformed into an image-space based one in order to exploit the power of the GPU?

2.1.1 Graphics Pipeline and Cg language

In this section, a general explanation about graphics pipeline is presented. [FK03] is a good reference to learn about graphics hardware capabilities and how to properly use it.

First of all, the geometry and raster pipelines are described, their input and output items, and the capabilities of each one. The main part of these texts has been obtained from [Den03] and [FK03]. The tutorial included in [Kil99] gives a thorough description of an important part of the raster pipeline: the stencil test and its applications.

By using a graphics API (for example OpenGL) it is possible to define objects using different primitives: points, segments, polygons, polygon strips, etc. This API also allows for the modification of the state variables, which control how the geometry and the fragments inside the geometry and raster pipelines are affected, respectively. All the geometric objects defined by the CPU enter the graphics engine at the geometry pipeline, one at a time. The geometry pipeline is responsible for transforming and cutting the input geometry taking into account the user-defined state variables as projection and clipping planes, and finally subdividing the geometric objects into fragments, which are the input

for the raster pipeline. There, four tests are applied to the fragments and the ones that pass all of them are transformed to pixels, losing their depth information.

Finally, if two fragments coincide into the same pixel, the final color of the pixel can be determined using distinct strategies: the color of the fragments can be blended, they can be logically combined, or simply the last fragment gives its color to the pixel.

The current contents of the color and stencil buffers can be read back into the main memory of CPU.

In Figure 2.1 there is a diagram of the general graphics pipeline.

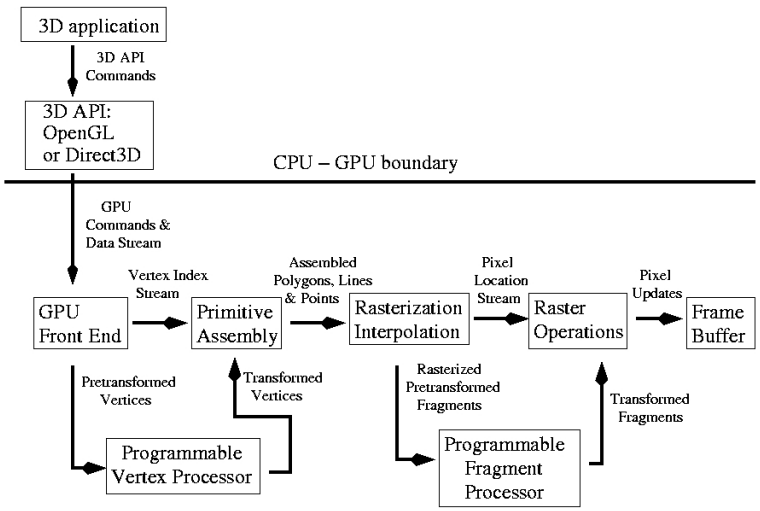


Figure 2.1: *Graphics pipeline. The unique programmable parts of the graphics hardware are the vertex and pixel (also called fragment) shaders. It is only allowed to modify all the other parts by changing some global attributes from the API (OpenGL, Direct3D, etc), for example the depth function to evaluate in the depth test, the activation of any test, etc.*

Geometry pipeline

The geometry pipeline is responsible for applying the projection determined by a state variable to the input geometric objects. Apart from that, the geometry pipeline has the ability to change the input objects by modifying their vertices. Nowadays it is also possible to add or delete vertices to change completely the input geometry.

Then, the clipping planes defined in the CPU cut and discard the parts of the objects that are, typically, outside the field of view determined by the projection.

At the end of the geometry pipeline any remaining portion of the object is discretized in a grid producing a set of fragments. Each of these fragments corresponds to a pixel of

the screen in the xy plane, however they also have depth information.

Raster pipeline

In contrast to the geometry pipeline, the input of the raster pipeline is a set of fragments. The fragments from the input set are pushed through the pipeline, separately and independently of each other. This processing of fragments can be compared to a parallel processor field with a rather simple processor residing on each pixel. According to the x and y values of a fragment, it is attached to the appropriate pixel. Based on the position, depth, and color values, a fragment has to undergo four tests until the buffers of the associated pixel are eventually altered.

When a fragment fails either of the first two tests, it is rejected from the pipeline without any further side effect. The values and parameters of the per fragment tests define the state of the raster pipeline and they are valid for the entire set of fragments. A change of parameters can only be caused by a new set of fragments.

The four tests are:

1. Scissor test

   This is the first test to pass. It is possible to define a rectangular portion of the active window containing the picture. If a fragment resides inside this area, it passes the scissor test. Otherwise, the fragment is rejected without any side effects on the pixel buffers.

2. Alpha test

   The alpha value of a fragment is compared to the value of the corresponding state variable. The allowed comparison functions are *smaller than*, *bigger than*, *equal*, *smaller or equal*, *bigger or equal* and *different*. Additionally it is possible to always accept or reject a fragment. Again, if the fragment does not pass this test, it is discarded from the pipeline without any further side effects.

3. Stencil test

   In contrast to the other tests, this test is applied to the pixel attached to the fragment. The stencil value of the fragment is compared to a reference value, determined by a state variable. Any result of the comparison causes a side effect on the stencil value of the fragment. A negative outcome of the test will erase the fragment. It

is possible to execute a predefined action on the stencil test, depending on whether the stencil and depth tests are passed or not

- The fragment fails the stencil test.
- It passes this test but fails the subsequent depth test.
- It passes both tests.

For each possible result one of the following actions can be executed:

- Keep the current value of the stencil buffer.
- Replace the stencil buffer value with 0.
- Replace the stencil buffer value with a reference value.
- Increment or decrement the stencil buffer by 1.
- Invert the value of the buffer.

4. Depth test

   It is divided in two consecutive depth units. The depth test is declared to be passed if and only if both test units are successfully passed by the fragment. An unsuccessful test will cause the fragment to be vanished. The first depth unit operates on the z-buffer of the fragment, and the second one on the z'-buffer.

   Similar to the stencil test, the allowed compare function might be *smaller than*, *bigger than*, *equal*, *smaller or equal*, *bigger or equal* and *different*. Again it is also possible to always accept or reject a fragment.

   The depth test corresponds to the only test in which data from the fragment is directly compared to data belonging to the pixel. If any incoming fragment passes the depth test, the z-buffer value of the pixel is replaced by the fragment's value.

Any fragment which passed all the per-fragment tests is finally displayed on the screen, which means that red, green, blue and alpha buffers of the corresponding pixel are updated.

The simplest method to accomplish an update is to overwrite the existing values with the incoming ones. Apart from that, there are two other methods:

- The values can be combined using logical operations.
- The fragment data can be blended with pixel data.

Some examples of the available operations are: *Clear buffers* (all 0's), *AND*, *XOR*, *Set buffers* (all 1's).

Programming the GPU: the Cg language

As a result of the technical advancements in graphics cards, some areas of 3D graphics programming have become quite complex. To simplify the process, new features were added to graphics cards, including the ability to modify their rendering pipelines using vertex and pixel shaders.

In the beginning, vertex and pixel shaders were programmed at a very low level with only the assembly language of the graphics processing unit. Although using the assembly language gave the programmer complete control over code and flexibility, it was pretty hard to use. In this context, a portable, higher level language for programming the GPU was needed, thus Cg was created to overcome these problems and make shader development easier.

Some of the benefits of using Cg over assembly are:

- High level code is easier to learn, program, read, and understand than assembly code.
- Cg code is portable to a wide range of hardware and platforms, in contrast with assembly code, which usually depends on hardware and the platforms it is written for.
- The Cg compiler can optimize code and do lower level tasks automatically.

Cg programs are merely vertex and pixel (or fragment) shaders, and they need supporting programs that handle the rest of the rendering process. Cg can be used with different graphical APIs, for instance OpenGL or DirectX. However each one has its own set of Cg functions to communicate with the Cg program.

In addition to being able to compile Cg source to assembly code, the Cg runtime also has the ability to compile shaders during the execution of the supporting program. This allows the shader to be compiled using the latest available optimizations. However this technique also permits the user of the program to access the shader source code, since it needs to be present in order to be compiled, which can be problematic if the author of the code does not want to share it.

Related to this, the concept of profiles was developed to avoid exposing the source code of the shader, and still maintain some of the hardware specific optimizations. Shaders can be compiled to suit different graphics hardware platforms (according to profiles). When the supporting program is executed, the best optimized shader is loaded according to its

profile. For instance there might be a profile for graphics cards that support complex pixel shaders, and another one for those supporting only minimal pixel shaders. By creating a pixel shader for each of these profiles, a supporting program enlarges the number of supported hardware platforms without sacrificing picture quality on powerful systems.

All tools and utilities commented before have been designed for working with graphics. However, a problem can be transformed to an image-based algorithm (if it is possible and if one finds the way) and solved (usually an approximated solution is mostly obtained) using the GPU capabilities. Thanks to the use of Cg we are able to test and program our image-based algorithms in a simple and faster way. Moreover, all these algorithms can be implemented in a transparent way with respect to the GPU hardware and low-level calls.

2.1.2 General-Purpose computation on GPU and CUDA

CUDA is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that execute parallel kernels, which may be simple functions or full programs. The execution in CUDA is structured in blocks. All the blocks of a single execution form a grid and every block is subdivided in threads that are executed in a parallel fashion. The GPU has a finite number of concurrent multiprocessors and every processor inside them is responsible for executing a single thread. Thus we can imagine that all these processors are executing threads at the same time. Normally, each of these threads computes a small portion of the problem, independent to all the other ones. Apart from the parallelism in the execution, we can also access to the GPU memory concurrently in order to increment the efficiency. There are different kinds of memory classified by their access speed and physical distance to the concurrent processors. Independently from the kind of memory used, it is important to remark that the access to stored data in the GPU memory has to be done carefully if we want an optimum implementation. In this section, the CUDA language will be described based on [NBGS08], an introductory document downloadable from the NVIDIA website that explains the basics of the Graphical Processing Units and more specifically the advantages of CUDA. Moreover it has information about how to access efficiently the GPU memory.

Architecture advantages over Cg

An obvious advantage of CUDA over Cg is that it is a GPGPU (General-Purpose computation on GPU) programming language, which implies that the GPU can be used to program general purpose problems not necessarily related to the computer graphics field.

Therefore, one does not have to worry about graphics primitives or how to discretize the problem in pixels and *render* them in the right way in order to obtain a non-graphical solution. However, there are some other important advantages.

In Cg or other GPU languages which use the graphics pipeline, a pixel is not allowed to write any position of the graphics memory. The pixel x, y in screen coordinates can only write at position x, y on a color or depth buffer. This is because the architecture can not handle conflicts between writting operations from different pixels, due to the fact that all pixels are completely independent to each other. They can not communicate with other pixels executed at the same time in any way. Cg only permits a pixel in position x, y to read from any position of a texture and to read or write the position x, y of the color and depth buffers. If it is needed to read and write values in the same memory space, two or more rasterization steps using a *ping-pong* technique are often employed.

CUDA architecture permits a thread (the pixel in Cg can be seen as equivalent to the thread in CUDA) to read or write any position on the graphics memory space. There is an exception with shared memory that can be found in Section 2.1.2. However, the latter exception is the key to another advantage of CUDA over Cg. Threads can communicate with some others using the shared memory, which represents a great advantage in solving a wide range of problems.

It is important to remark that CUDA has a much more flexible computation architecture which substantially reduces the limitations of the graphics oriented paradigms.

The execution model

First of all, some definitions are needed to understand the following paragraphs. When using CUDA, two environments exist: the *host* and the *device*. The host corresponds to the CPU computation part, from where the device functions, called *kernels*, partition the problem in small portions called *threads* which are executed in a parallel way inside the GPU.

All threads executed by a kernel are organized in *blocks* and all these blocks are grouped in a single *grid*. Therefore each grid of threads defined in the host is always executed by a unique kernel inside the device (see Figure 2.2).

Every block is logically divided in *warps*. All warps have the same fixed size depending on the model of GPU. Usually, a warp contains 32 threads belonging to the same block. All threads of a warp always execute exactly the same instructions and have restrictions on the access to shared memory (see Section 2.1.2), therefore this warp size must be taken

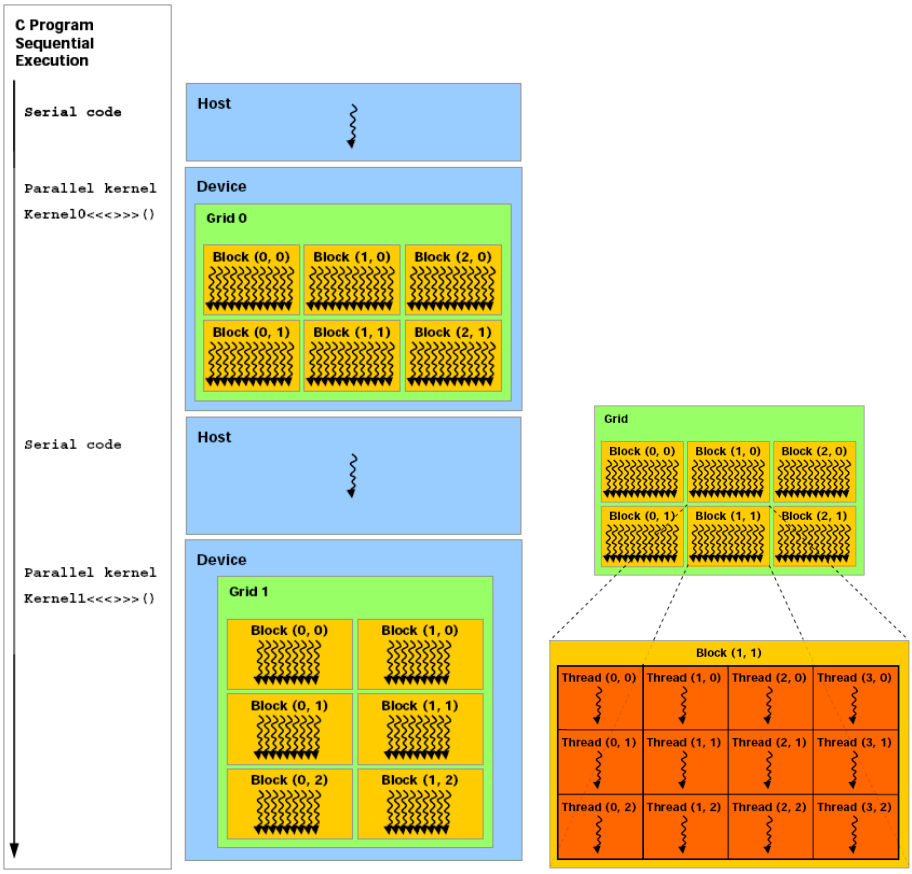


Figure 2.2: *The left image shows the CUDA execution process. The right one shows that every block inside the grid is composed by an arbitrary number of concurrent threads. Image taken from [NBGS08].*

into account especially when current kernel has control flow instructions or uses per block shared memory.

All the threads within the same block can cooperate by sharing data and synchronizing their execution in order to access shared memory efficiently. When a synchronize instruction is put in a point in the code, the threads reaching this point wait until all other threads in their same blocks reach it. Then all the threads within a block continue the parallel execution. Since the shared memory is only available at a block level, threads of different blocks cannot synchronize or share information (examples can be found in the following sections).

There are strong limitations in the number of threads per block, but a kernel can be executed using a grid with a lot of blocks. This gives us the possibility to have a very large number of threads being executed at the same kernel. In addition to this, the thread cooperation is reduced because the main part of them are actually in distinct blocks.

This model allows kernels to run without recompilation on different devices with different hardware capbilities. If the device has very few parallel capabilities it can run all the blocks sequentially. On the other hand, if the device has large parallel capabilities, it will run a lot of them in a parallel fashion. The advantage here is that all these facts are almost totally transparent to the programmer, therefore he can focus his work on improving the algorithms.

Memory model

A thread only has access to the device memory, and this space memory is divided into some different memory spaces (see Figure 2.3). The most important ones are explained as follows.

Global memory occupies the main part of the device memory space. It can can be accessed by all the threads within the grid and is not *cached* (it is the slowest one). The advantage is that it has a lot of space, therefore it might be useful when data is too big to fit in any other kind of memory. All global space memory is accessible from any thread within the grid, thus all the threads of the grid can read or write any position there.

Constant memory is a cached small portion (usually 64Kb) of the GPU memory that cannot be modified during the CUDA kernel processing and it is accessible by all the threads. The constant memory is very useful when we have relatively small data

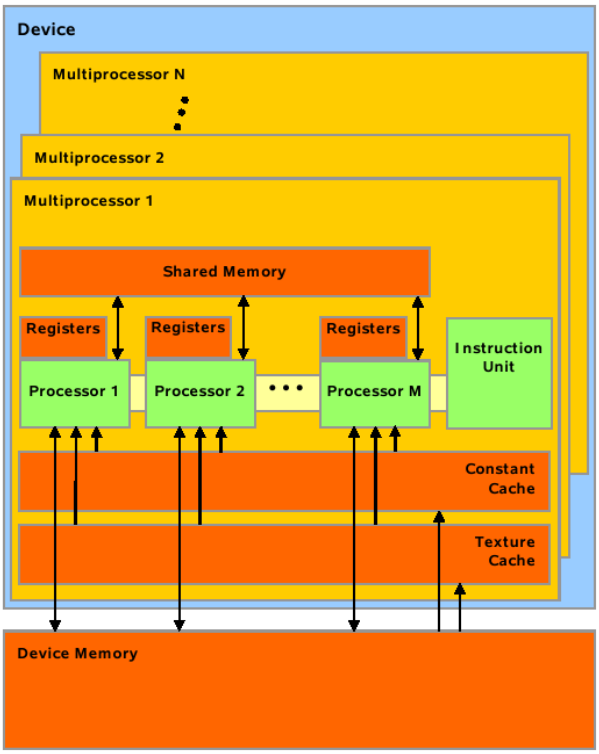


Figure 2.3: *Memory model of CUDA architecture.*

that has to be accessed many times by the main part of the threads, because its cache increases the access speed. As its name indicates, we can only read from this space memory. All threads within the grid have access to all the constant space memory.

Registers are used by the local variables of each thread. It is a small memory space and a variable located here is only accessible by its thread. On the other hand it is the fastest one. It is important to be careful with this kind of memory because when a thread has occupied all its registers, the next declared variables will be located in the *local memory*, a portion of memory inside the global memory (the slowest kind of memory inside the GPU).

Shared memory is, probably, the most important memory type in CUDA. If the data needed for the CUDA computation must be accessed by distinct threads and a lot of times, the use of shared memory is highly recommended. Shared variables are visible for all the threads within a certain block, thus all data has to be structured in a good way to take advantage of its cache. In fact, the shared memory is the fastest memory space in the GPU.

The global and constant spaces can be set by the host before or after the kernel execution inside the device. This mechanism is useful for transfering data to the device before the kernel is launched or to host in order to get the results of a previously executed CUDA kernel.

A multiprocessor takes 4 clock cycles to issue one memory instruction for a warp. When accessing global memory, there are, in addition, 400 to 600 clock cycles of memory latency.

Much of this global memory latency can be hidden by the thread scheduler if there are sufficient independent arithmetic instructions, which can be issued while the global memory access is being complete.

Since global memory is of much higher latency and lower bandwidth than shared memory, global memory accesses should be minimized. A typical programming pattern is to stage data coming from global memory into shared memory; in other words, to have each thread of a block:

- Load data from device memory to shared memory,
- Synchronize with all the other threads of the block so that each thread can safely read shared memory locations that were written by different threads,
- Process the data in shared memory,
- Synchronize again if necessary to make sure that shared memory has been updated with the results,
- Write the results back to device memory.

The CUDA language

The CUDA programming interface provides a relatively simple group of primitives for users familiar with the C programming language to program algorithms that can be executed inside the GPU (device).

It has extensions to the C language that allow the programmer to target portions of the source code for execution on the device. It also supplies a runtime library composed by: (1) a host component that provides functions to control and access one or more compute devices from the host; (2) a device component providing device-specific functions; (3) a common component with built-in vector types and a subset of the C standard library that are supported in both host and device code.

It is important to remark that only the functions provided by the common runtime component from the C standard library are supported by the device.

The following paragraphs describe the basic extensions needed to understand and program CUDA algorithms .

Function type qualifiers are the first extension. There are three of them:

- A function declared using the `__device__` qualifier is always executed on the device and it can only be called from other functions inside it.
- The `__global__` qualifier is used to create a function that acts as a kernel. A kernel is the *main* program of a parallel computation running in CUDA. Therefore it can only be executed inside the device and it is always called by the host.
- If a function has the `__host__` qualifier it is executed on the host and of course, it can only be called from there. It is the default function qualifier. It can be used together with `__device__` in order to compile the function for the device and the host and be carried out from both at the same time.

Variable type qualifiers are used to specify the memory location of a variable on the device. There are three of them:

- The `__device__` qualifier declares a variable that resides on the device. The memory space that a variable belongs to, is defined using this qualifier together with the constant and shared ones (see next paragraphs). If none of them is present, then the variable is located in the global memory space, it has the same lifetime as the application and is accessible from all the threads within the grid (from the device) and from the host through the runtime library.
- `__constant__` qualifier declares a variable that resides in the constant memory space. As with the `__device__` qualifier, it has the lifetime of an application and is accessible from all the threads within the grid and from the host through the runtime library.
- Finally, `__shared__` qualifier declares a variable that resides in the shared memory space of a thread block. It has the lifetime of the block and is only accessible from all the threads within the block. Shared variables are guaranteed to be visible by other threads only after the execution of `__syncthreads()`.

These variable qualifiers are not allowed on struct and union members, on formal parameters and on local variables within a function that is executed on the host.

Variables defined using `__shared__` or `__constant__` qualifier can not have dynamic storage. `__device__` and `__constant__` variables are only allowed at file scope. They can not be defined inside a function. As its name says, a `__constant__` variable cannot be modified from the device, only from the host through host runtime functions. They are actually *constant* to the device.

Generally a variable declared in device code without any of these qualifiers is automatically put in a register. However in some cases the compiler must choose to place it in local memory (much more slower). This is often the case for large structures or arrays that might consume too much register space or when the register space for a thread is already full. This is also the case of arrays for which the compiler cannot determine if they are indexed with constant quantities.

A new directive to specify how a `__global__` function or kernel is executed on the device from the host is needed.

This directive specifies the execution configuration for the kernel and it defines, basically, the dimension of the grid and blocks that will be used to execute the function on the device. It is specified by inserting an expression of the form `<<< Dg, Db >>>` between the function name and the argument list, where:

- Dg is of type dim3 and specifies the dimension and size of the grid, such that $Dg.x * Dg.y$ equals the number of blocks being launched; $Dg.z$ is unused or it is always 1.
- Db is also of type dim3 and specifies the dimension and size of each block, such that $Db.x * Db.y * Db.z$ equals the number of threads per block.

Four built-in variables that specify the grid and block dimensions and the block and thread indices.

- `gridDim` and `blockDim` are of type dim3 and contain the dimensions of the grid and block respectively.
- `blockIdx` and `threadIdx` are of type uint3 and contain the block index within the grid and thread index within the block respectively.
- `warpSize` is a variable of type int and contains the warp size in threads.

It is not allowed to take the address or assign values to any of them.

Each source file containing these extensions must be compiled with the CUDA compiler *nvcc* that will give an error or a warning on some violations of these restrictions. However,

in most cases the errors or warnings cannot be automatically detected, thus we must take special care when programming CUDA algorithms.

2.2 Computational geometry concepts

In this section the geometric concepts and algorithms mainly used in our visibility and good-visibility maps computation are briefly described. One of the most importants topics needed to compute good-visibility maps is the so-called depth contours problem, which treats the problem of *how deep* a region of the plane or space is with respect to a set of points. The depth contours are based on the convexity and its properties, thus the first subsection describes this concept. In order to compute multi-visibility maps, i.e. an overlay of distinct visibility maps, a little explanation of how the overlay of two or more planar subdivisions can be computed is necessary. Finally, some comments related to the definition and creation of terrains is given.

2.2.1 Convexity

Let a region R and two points p and q inside R in $\mathbb{R}^d$, with $d > 0$, R is a *convex region* if all points belonging to the segment $\overline{pq}$ are interior to R. If two points p and q exist inside R generating a segment which is partially outside R, then R is not convex. Let a set of points S in $\mathbb{R}^d$, its *convex hull* $CH(S)$ is defined as the minimal convex region containing every point of S. Basically, the computation of $CH(S)$ consists of finding all the *exterior* points of S. We say that $s \in S$ is not exterior with respect to S if a triangle (p, q, r) exists where $p, q, r \in S$, $p, q, r \neq s$ and s is inside (p, q, r). Another useful property says that s is exterior to S if and only if a line containing s that leaves all the other points in S at one side exists.

If the extremal points of S are known and ordered clockwise, $CH(S)$ can be easily computed as the intersection of the halfplanes defined by the oriented lines constructed from every consecutive pair of points. This is possible because an important property of the halfplanes (hyperplanes when considering any number of dimensions) states that an intersection of any of them always results in a convex region.

From 1972, a lot of algorithms to efficiently compute the convex hull of a set of points has been reported. Graham [Gra72] proposed the first convex hull algorithm running on the plane with a $O(n \log n)$ worst-case running time. Later, Shamos proved in his Ph.D. thesis [Sha78] that the convex hull problem can be reduced to the sorting one, which

has a lower bound of $\Omega(n \log n)$. After the Graham's algorithm was published, a lot of others algorithms for computing the convex hull from a set of points in a two-dimensional space were reported, as H. Bronnimann and coworkers expose in [BIK$^+$04]. There are also algorithms to compute the convex hull of a set of points in three-dimenional spaces (T. Chan briefly describe some of them in [Cha03]) or even in any dimension using the QuickHull algorithm presented by C.B.Barber et al. in [BDH96].

2.2.2 Overlay of planar subdivisions

A planar subdivision, also known as planar map, divides the plane using vertices, edges and faces induced by a set of line segments. Overlaying two planar subdivisions produces a new planar subdivision. Since a planar subdivision is configured by a set of line segments, if we want to obtain the overlay of two planar maps M_1 and M_2 it is necessary to intersect every pair of segments, one for each planar map. It can be useful in a wide range of fields, including for Geographic Information Systems (GIS) or for finding multi-visibility maps from a set of view objects.

The overlay of two planar maps can be seen as a specialized version of the two-dimensional line intersection problem. This problem has been studied for over twenty-five years. Bentley and Ottmann [BO79] developed the first solution to the line intersection problem in 1979, using a plane sweep algorithm with a computational complexity of $O(n \log n + k \log n)$ time, where k is the number of intersections found and n the number of segments. This algorithm is not optimal because the lower bound has been proved to be $\Omega(n \log n + k)$. Later, Chazelle et al. [CE92] presented the first algorithm running in optimal time complexity, however it required $O(n + k)$ space. The first optimal algorithm in both time and space was reported by Balaban in [Bal95], which runs in $O(n \log n + k)$ time and requires $O(n)$ space.

For the case of the map overlay problem, the intersection problem is actually easier than the general case. This is because we assume that a map is a planar subdivision, so every intersection will be between one segment from the first map and one segment from the second map. This problem was solved in $O(n \log n + k)$ time and $O(n)$ space by Mairson and Stolfi [MS87] before the general problem was solved optimally. Moreover, if we assume that planar maps are connected subdivisions, Finke and Hinrichs [FH95] showed in 1995 that the problem can be solved in $O(n + k)$ time.

2.2.3 Depth contours

In this section the depth contour concept is presented. Hereafter its formal definition and some algorithms to compute it on the plane are summarized. Afterwards some implementations using graphics hardware techniques are reported and finally compared with our implementation.

Definition

Let P be a set of n points. The location depth of an arbitrary point q relative to P, denoted by $ld_P(q)$, is the minimum number of points of P lying in any closed halfplane defined by a line through q. The k-th depth region of P, represented by $dr_P(k)$, is the set of all points q with $ld_P(q) = k$. For $k \geq 1$, the external boundary $dc_P(k)$ of $dr_P(k)$ is the k-th depth contour of P. The depth map of P, denoted $dm(P)$, is the set of all depth regions of P (see Figure 2.4) whose complexity is $O(n^2)$. When all points of P are in convex position we achieve the latter complexity, thus this bound is tight.

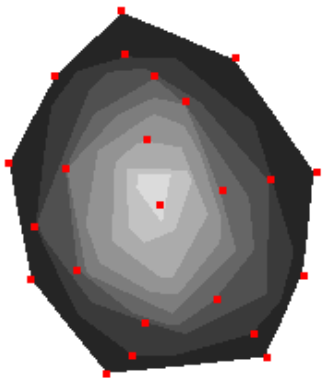


Figure 2.4: *Representation of the depth contours of a set of 23 points.*

Depth Contours computation in the plane

Miller *et al.* [MRR+03] presented an algorithm for computing the depth contours for a set of points that makes an extensive use of duality, and proceeds as follows: given a set P of points, the algorithm maps all points of P to their dual arrangement of lines. Then, a topological sweep is applied to find the planar graph of the arrangement whose vertices are labeled with their respective *levels*, i.e. the number of dual lines above them. The depth of a vertex can be computed using $min(level(v), n - level(v) + 1)$. Finally, for a

given k, $dc_P(k)$ is computed by finding the lower and upper convex hulls of the vertices at depth k. Each of these vertices corresponds to a halfplane in the primal plane. $dc_P(k)$ is the boundary of the intersection of the previously mentionned halfplanes. $dc_P(k)$ does not exist if this intersection is empty. The complexity of the algorithm is $O(n^2)$ in time and space, which has been shown to be optimal.

Since for large n the running time of Miller *et al.* algorithm might be too large, in the following section some algorithms are presented that solve the latter problem by using graphics hardware capabilities (i.e. an image of the depth contours is drawn where the color of a pixel represents its depth value).

Graphics Hardware solutions

Depth contours have a natural characterization in terms of the arrangement in the dual plane induced by the set P of n points.

The mapping from the primal plane Π to the dual plane Π' is denoted by the operator $\mathcal{D}(\cdot)$. $\mathcal{D}(p)$ is the line in plane Π' dual to the point p, similarly $\mathcal{D}(l)$ is the point in plane Π' dual to the line l. We denote the inverse operator by $\mathcal{P}(\cdot)$, i.e. $\mathcal{P}(q)$ is the line in plane Π primal to the point q in the dual plane Π'. Each point $p \in P$ in the primal plane is mapped to a line $l = \mathcal{D}(p)$ in the dual plane (see Figure 2.5). We can also define the dual of a set of points P as $\mathcal{D}(P) = \cup_{p \in P} \mathcal{D}(p)$. The set of lines $\mathcal{D}(P)$ define the dual arrangement of P. In this dual arrangement the *level* of a point x is computed as the number of lines of $\mathcal{D}(P)$ that satisfy the following criteria: they either contain or stricly lie below x (see Figure 2.6). The depth contour of depth k is related to the convex hull of the k and $(n-k)$ levels of the dual arrangement.

Krishnan and coworkers [KMV02] and Fisher *et al.* [FG06] developed algorithms that draw a discretization of the depth contours from a set of points using graphics hardware capabilities. Both algorithms are based on the dual concept. In the first step, the input point set P is converted to a set of lines in the dual plane. The algorithm runs on two bounded duals instead of one, due to the finite size of the dual plane. In that way it is guaranteed that all intersection points between the dual lines lie in one of the two latter dual regions. Since each dual plane is discrete, it is possible to compute the level of each pixel by drawing the region situated above every dual line of P, by incrementing by one the stencil value of the pixels located inside each of these regions. In the second step the two images generated in this fashion are analyzed. For each pixel q located on a dual line, its corresponding primal line $\mathcal{P}(q)$ is rendered with the appropriate depth and color as a graphics primitive using the depth test (z-buffer). The complexity in time of the algorithm

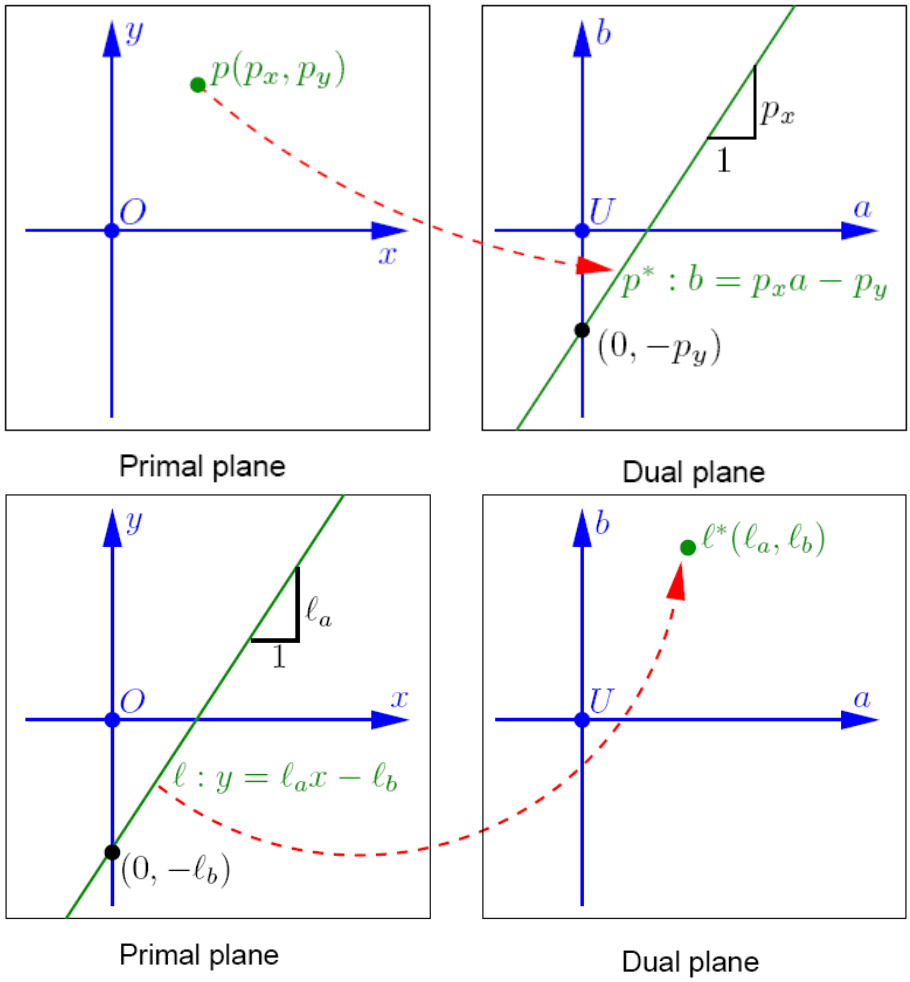


Figure 2.5: *Representation of the dual transformation. The dual transformation for points is depicted in the upper images whereas the dual transformation for lines is shown in the lower ones.*

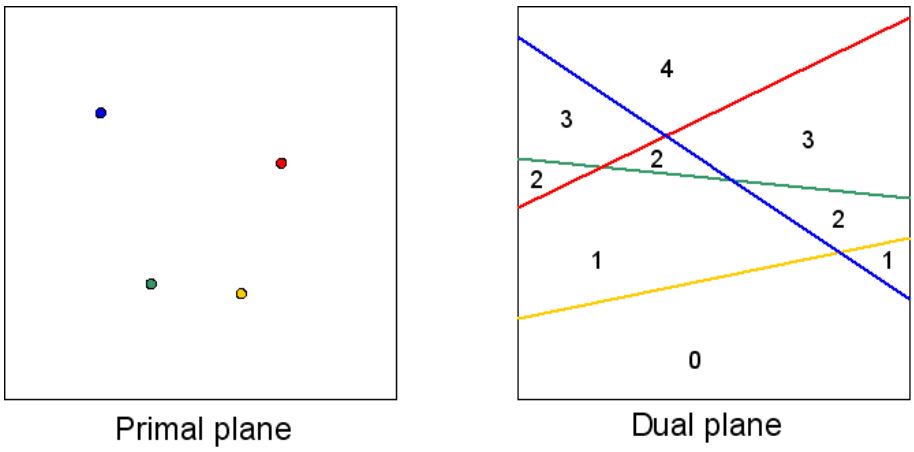


Figure 2.6: *Representation of a set of points in the primal plane (left image) and its arrangement with a level map in the dual plane (right image). The level of each zone is represented with numbers.*

is $O(nP_W + P_{W^3} + A_{W^2} + nCP_{W^2}/512)$, where W denotes the number of pixels of each column and row of the color buffer, C is the cost of a single readback to the CPU, and A_x and P_y correspond to the time needed to access x pixels of a texture and render y pixels, respectively.

Mustafa *et al.* [MKV06] developed an algorithm to compute depth contours that has a time complexity of $O(n^2 A_1 P_{W^2} + nCP_{W^2}/512)$. It improves the running time for relatively small sets of points. In this other case, the algorithm constructs a line ℓ for every pair of points of P, then it uses its dual point ℓ^* to determine its level in the dual plane images. Finally it paints the two halfplanes ℓ^+ and ℓ^- with depths $level(\ell)$ and $n - level(\ell)$ (n is the number of points in P), respectively, using the depth test.

One of the main problems of the latter implementations is to decide the number of points used to discretize each dual line. If few discretization points are employed, the final result might be of very bad quality. However if a large number of discretization points is used, then the same pixel might be processed several times (due to the fact that the rasterization of different lines can lead to the same pixels). In order to avoid this problem, a slightly different algorithm can be implemented, which directly searches for the intersection points between the lines and paints the dual of the segments defined by the previous intersections. The dual transformation of a segment is, in fact, the same region obtained when painting the dual line of every point contained in the segment.

Krishnan et al. [SKV06] have recently developed a new method to compute depth contours. It has a superior algorithmic complexity but with relatively small sets of points

the running time is substantially decreased. As in Mustafa *et al.* algorithm [MKV06], a line ℓ for every two points of V is constructed. Afterwards, for each line ℓ, its dual point ℓ^* is used to know the level of ℓ in the dual planes images. The two halfplanes ℓ^+ and ℓ^- are painted with depth equal to $level(\ell)$ and $n - level(\ell)$, respectively, using the depth test.

Our own implementation of the depth contours (explained in Chapter 4) is based on Mustafa and coworkers approach [MKV06]. Depth contours are necessary to compute the 2D, 2.5D and 3D good-visibility maps. .

2.2.4 Terrains

A terrain is a two dimensional surface in three dimensional space with a special property: every vertical line intersects it only at one point, if it intersects it at all [dBvKvOO97]. As a more formal definition, it is the graph of a function $f : A \subset \mathbb{R} \rightarrow \mathbb{R}$ that assigns a height $f(p)$ to every point p in the domain, A, of a terrain. Measures of height of real terrains is carried out through sampling. Then, our knowledge about the f is restricted to a finite set $P \subset A$ corresponding to a sample points.

There are several models to represent heights, namely, Regular Square Grid (GRID), Contour Line, Triangulated Irregular Network (TIN) and Hierarchical [vKNRW97b]. These models are named Digital Elevation Models (DEM) and they are a finite representation of an elevation model. This terminology arises in the field of Geographic Information System (GIS) where these DEM are mainly used. Sea height is the elevation model which is best known, which is why the term terrain or Digital Terrain Model (DTM) are frequently used.

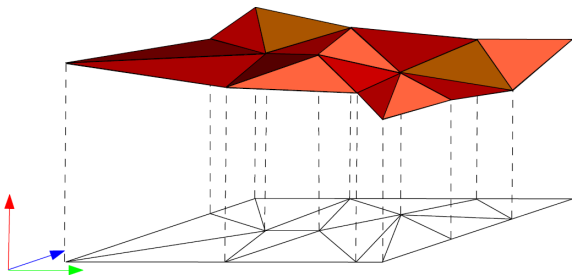


Figure 2.7: *A schema of a terrain.*

Digital Elevation Models

In this section we give a brief explanation of the existing DEMs, and extend, in the following sections, the two main models used in this thesis, GRIDs and TINs.

A GRID is a structure that specifies values in a regular square division of the domain. In a computer it is stored in a bi-dimensional array. For each square or array entry a height value is exactly specified. There are different interpretations of a grid. The first, the stored height can be thought of as the elevation for all points in the square. In this case, the DEM is not a continuous function. The second, the stored elevation might represent the height correspondent to the central point of the square or the average elevation of the square. Here an interpolation method is necessary in order to obtain a DEM that specifies the elevation for each point.

Triangulated Irregular Network (TIN) TIN arises from any set of data points in the plane for which an elevation is given. The set of points which join its elevation are stored, and a planar triangulation is created over these points. Often, the triangulation of choice is the Delaunay triangulation because of its natural properties. The elevation of any intermediate point is given by linear interpolation on the elevations of three original points that form a triangle that contains the intermediate point. The TIN can be stored in a DCEL or in a quadedge. Another possibility is a net structure: for each triangle, edge, and vertex there is a record. The record of a triangle has three pointer fields. These pointers are directed to the record of each of its incident edges. The record of an edge has four pointer fields, two of them addressed to its adjacent triangles, and the other two to the incident vertices. Vertex record contains vertex coordinates x and y, and the elevation. This structure allow us to find the vertex elevation, adjacent triangles to a given triangle and much more, in constant time. The TIN model is attractive because of its simplicity and economy and is a significant alternative to the regular raster of the GRID model.

Now a formal definition which will be useful in posterior chapters. A TIN, $(\mathcal{T}, \mathcal{F})$, is formed by a triangulation $\mathcal{T} = \{T_1, \ldots, T_n\}$ of the domain D (in the xy plane), and by a family $\mathcal{F} = \{f_1, \ldots, f_n\}$ of linear functions such that: a) function $f_i \in \mathcal{F}$ is defined on triangle $T_i, i = 1 \ldots n$; b) for any pair of adjacent triangles T_i and T_j, f_i and f_j coincide in $T_i \bigcap T_j$. For any triangle $T_i \in \mathcal{T}$, $f_i(T_i)$ is a triangle in space called a face of the terrain, and the restriction of f_i to an edge or a vertex of T_i is called an edge or a vertex of the terrain.

Chapter 3

Multi-visibility maps computation using the GPU

In this chapter we first introduce the previous work in visibility and multi-visibility maps computation and afterwards we present a detailed explanation of the algorithms we use to compute them using the GPU. For the two-dimensional and terrain cases the algorithms are implemented using pixel shaders in Cg and OpenGL. However, the three-dimensional case is designed and implemented using CUDA, the new GPGPU language designed by NVIDIA.

The multi-visibility maps will be used in all the next chapters as a fundamental part of the good-visibility map computation.

3.1 Introduction

Visibility gathers combinatory theory, computational geometry and computer science and its results have applications in a wide variety of fields ranging from robotics to path finding, computer vision, graphics, CAD, etc. The basic visibility problem of determining the visible portions of the scene primitives from a viewpoint is now believed to be mostly solved by applying the Z-buffer technique. However, visibility has recently regained attention in numerous applications ranging from planning the placement of communication towers or watchtowers, to planning buildings and roads so that they have a good view, to finding routes on which you can travel while seeing a lot, or without being seen. One related subject of study is the problem called *Art galleries*, presented for the first time by Victor Klee in 1973: the problem is to determine the minimum number of polygon vertices

necessary to view all the rest. Klee made a conjecture that $\lfloor n/3 \rfloor$ was the number of guards always sufficient and sometimes necessary to illuminate a polygon with n vertices. However, it was Chvátal who gave the first proof of this in 1975 [Chv75]. This result is known as the *Art galleries theorem*. More details can be found in [She92]. The Art gallery problems can be applied to a wide range of real worldwide industrial applications related in any way to illumination or visibility: street or building illumination, vigilance of spaces, etc. Of course, in the major part of the practical cases the real physical objects are substantially different from the theoretical points and segments used. Therefore the introduction of more sophisticated view elements and obstacles is needed to describe more realistic situations. Moreover, one idea that one can come up with is the fact that light emitted by a focus loses intensity as the distance to this focus increases. Therefore, when an object is situated sufficiently away from a focus, it can be considered that this object is not illuminated at all by it. This limitation was defined for the first time in Computational Geometry by Ntafos in 1992 [Nta92] who introduced the concept of *range limited visibility*.

Given a viewpoint v, the *visibility map* of v is the partition of the scene into the visible and invisible parts from v. The calculation of visibility maps from a set of viewpoints is a tool needed in a wide range of these kinds of applications, and the combination of these visibility maps in only one structure, which is usually called *multi-visibility map*. From a multi-visibility map a great variety of visibility queries can be efficiently answered: portions of the scene visible from at least one viewpoint, portions of the scene visible from all viewpoints, portions of the scene visible from a specific subset of viewpoints, portions of the scene visible that see a specific number of viewpoints, etc.

Since the exact computation of multi-visibility maps cannot be directly solved except for trivial cases, we propose to approximate them using the GPU considering different kinds of view elements, different kinds of obstacles and different kinds of visibility.

3.2 Definitions and previous work

In a scene defined by a set of objects S, called *obstacles*, two points p and q are *visible* if the line segment from p to q intersects none of the given obstacles. The *visibility region* or *viewshed* $V_v(S)$ of v is the set of points of the scene that are visible from v. The complement of $V_v(S)$ is called the *shadow region* of v. Then, the visibility map of v is the partition of the scene into the visibility and the shadow region.

3.2.1 2D visibility

When the input set of obstacles defines a simple polygon P, we are interested in determining the interior region of P visible from some distinguished interior viewpoint v. El Gindy and Avis [GA81] developed an algorithm for determining the visibility from a point inside a polygon. Their algorithm runs in optimal $\Theta(n)$ time and space, where n is the number of vertices of the given polygon. However, the main disadvantage is that the method does not work for polygons with holes.

Suri *et al.* [SO86] reported a solution to the problem of computing visibility inside a polygon P presenting holes. The resulting algorithm runs in $O(n \log n)$ time, which is proved to be optimal by reduction to the problem of sorting n positive integers (Asano et al. [AAG+85] obtained the same result independently). They consider a more general problem for computing visibility in polygons in the presence of a set of line segments $S = \{s_1, s_2, ..., s_n\}$, where two line segments $s_i, s_j \in S$ do not intersect except at their endpoints. The algorithm performs an angular plane sweep, and can be implemented in $O(n \log n)$ time in a straightforward manner. However, it is proved to be optimal time in the worst case.

The above methods are not easily generalized to the case of having input segments that do not form a simple polygon and, possibly, even might intersect. In this case it is more natural to consider the construction of the visibility region from a point v to be an *upper envelope* problem. In this general framework a collection $F = \{f_1, f_2, ..., f_n\}$ of functions from $\mathbb{R}$ to $\mathbb{R}$ is employed. The *upper envelope* of F is defined as the function

$$f(x) = max_{f_i \in F}\{f_i(x)\}$$

In geometric settings it is quite common for the functions in F to possess a crossing property which is based on the fact that any two such functions cross at most k times. Then it is appropriate to study the question of how a representation of the upper envelope function f can be efficiently constructed, as well as how large the latter representation must be. The most standard way of representing this function is as a list of intervals of $\mathbb{R}$ together with the indices of the functions in F that fulfil the maximum in each interval.

The crossing property previously defined gives rise to the so-called *Davenport-Schinzel* sequences. Such a sequence is defined using two parameters: n and k, where n is the number of characters and k is the maximum number of alternations that can occur between any two characters. Letting $\lambda_k(n)$ denote the maximum length of a Davenport-Schinzel sequence with parameters n and k, it is one of the most interesting results of combinatorics that, for any fixed value of $k \geq 3$, $\lambda_k(n)$ is $O(n \log^* n)$. This parameter is linear for smaller

values of k.

By a simple mergesort-like divide-and-conquer algorithm, we can construct the upper envelope of F in $O(\lambda_k(n)\log n)$ time. For instance, computing the upper envelope of a collection of functions defined by line segments in the plane can be done in $O(n\log n)$ time [Her89]. Equivalently, the region of the plane visible from a point can also be computed in this time.

Visibility can be computed not only from viewpoints, but also from view segments. Related previous studies indicate that two variants of the visibility can be computed when dealing with view segments: the so-called strong and weak visibility (see Figure 3.1). A point p has strong visibility from a view segment v if and only if all the points of v are visible from p. In contrast, p is weakly visible from v if any of the points of v is visible from p.

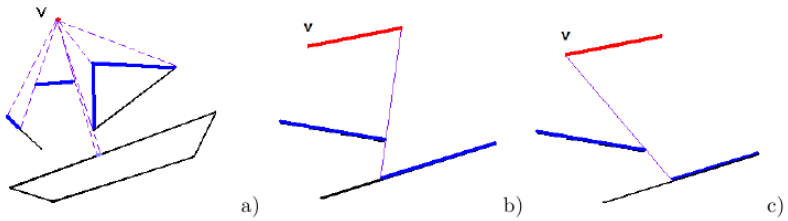


Figure 3.1: *Scenes in $\mathbb{R}^2$ where the visible parts are marked in blue when considering a: a) view point; b) view segment and weak visibility; c) view segment and strong visibility.*

Exact algorithms that provide the visible parts of the scene from a view segment v are sweep algorithms [GS96a, GS96b, NT]. A moving point p is placed on the view segment and is moved along v keeping track of the critical points. The latter are those points where the visible scene in both directions of any of its neighborhoods in s is different. A topological or visibility change occur in a critical point. This moving point approach can be generalized to handle free trajectories of the moving point. The weakly or strongly visible parts of the scene from a view segment can be obtained by placing a linear light source on the view segment [Gha98]. After the critical points are found by using the moving point strategy, the amount of light received by each part of the scene is computed. On the basis of the amount of light received, a part of the scene is classified as weakly or strongly visible, or not visible.

3.2.2 Visibility on terrains

Let T be a 3D triangulation representing a terrain (i.e., there is a height, the z-coordinate, associated to each triangle vertex). A point p on T is visible from v if and only if the line segment $\overline{pv}$ lies above T. The combinatorial complexity of the visibility region of v might be $\Omega(n^2)$, where n is the number of vertices of the triangulation.

The problem of computing the visibility region of a point arises as a subproblem in numerous applications, and, as such, has been studied extensively. For example, the coverage area of an antenna for which the line of sight is required, may be approximated by clipping the region that is visible from its tip with an appropriate disk centered at the antenna.

In [FM94] an algorithm for computing the visibility map from a point on a terrain modelled by a TIN is described (more details about TINs and other terrain representations can be found in references [vKNRW97a, FM94] and in Section 2.2.4). It is based on the determination of the lower envelope of a set of triangles and thus it can be applied to TINs. The worst case time complexity for this algorithm is $O(n^2)$ where n is the number of triangles in the terrain.

It is also desirable to have fast approximation algorithms, i.e. algorithms that compute an approximation of a visibility map. Moreover, a good approximation of the visible region is often sufficient, especially when the triangulation itself is only a rough approximation of the underlying terrain. CPU radial sweep based algorithms are presented in references [BMCK08, HTZ09, FHT09] for computing an approximation of a visibility region. There are other approximations for visibility region computation, for instance in [CFMS06], however, they are based on other types of viewing objects, such as segments.

3.2.3 3D visibility

3D visibility is studied in computer graphics, architecture, computational geometry, computer vision, robotics, telecommunications, and other research areas. Computer graphics aims to synthesize images of virtual scenes by simulating the propagation of light. Visibility is a crucial phenomenon that is an integral part of the interaction of light with the environment. The first visibility algorithms aimed to determine which lines or surfaces are visible in a synthesized image of a 3D scene. These problems are either known as *visible line* and *visible surface determination* or as *hidden line* and *hidden surface removal*. The classical visible line and visible surface algorithms were developed in the early days of computer graphics in the late 60s and the beginning of the 70s (Sutherland et al.,

1974 [SSS74]). These techniques were mostly designed for vector displays. Later, with the increasing availability of raster devices, the traditional techniques were replaced by the z-buffer algorithm (Catmull, 1975 [Cat75]). Nowadays, we can identify two widely spread visibility algorithms: the z-buffer for visible surface determination and ray shooting [PMS+99, WS01, WSBW01, IWW01] for computing visibility along a single ray. The z-buffer and its modifications dominate the area of real-time rendering whereas ray shooting is commonly used in the scope of global illumination methods. Recently, an increasing interest on 3D visibility has been awakened due to the advances in graphics hardware and video games. Therefore, the algorithms implemented in this field are basically designed to run inside graphics hardware [BW03].

We can distinguish three different classes of visibility. The *exact* visibility set: This is the set of all polygons that are partly or completely visible; the *approximate* visibility set: This set includes most of the visible polygons, but also some hidden ones. the *conservative* visibility set: This set includes all visible polygons, and may contain some hidden ones too.

Determining the exact visibility is very intensive in terms of computational power. Furthermore, it is difficult to design a fast algorithm to be *exact* and still robust. Most of the methods described in the literature are conservative, which use simplifications in some areas. Some methods can also be modified to calculate the approximate visibility set, which can lead to enormous speedups. This occurs especially in the case of video games, where frame rate is often more important than accuracy, thus a less detailed visibility computation is admissible.

There are two main different methods to approach visibility problems in computer graphics, depending on the type of geometrical data present: visibility from a region and visibility from a point.

Two different categories of methods exist when it comes to point-visibility. Calculations in image-space and those in object space. Current methods have started to use the best of both worlds.

More detailed information about the state of the art on 3D visibility can be found in many published surveys, as the ones presented by F. Durand and coworkers in [COCSD03] or [Dur00].

3.3 Two-dimensional multi-visibility maps using the GPU

In this section we present our algorithms which compute multi-visibility maps in a two-dimensional environment using the GPU with the help of the OpenGL computer graphics language.

The algorithms, explained in the following sections, basically consist of obtaining a representation of the the multi-visibility map of a set V of view elements in a texture denoted by MVM. Since we assign a bit of the RGBA channel to each viewpoint in V, the number of viewpoints is limited by the number of available bits per RGBA texture pixel (up to 128 with nowadays GPUs). In practice this is not a limitation because this maximum number of viewpoints is sufficient for most (if not all) applications.

First we describe the basic visibility algorithm from viewpoints and some of its variations, and later the version of the algorithm to compute visibility from view segments taking into account both weak and strong visibility. At the end of each section a brief description of the running time of the presented algorithms is given.

3.3.1 Multi-visibility maps from viewpoints

We consider two cases depending on the type of obstacles: segments or general objects. This is because the shadow region of obstacle with respect to a viewpoint can be easily computed only if the obstacle is a segment.

Segment obstacles

The texture MVM encoding the multi-visibility map is created by a simple image based algorithm. The shadow region of a viewpoint v with respect to the segment obstacle s, denoted by $sr(v, s)$, is the zone of the plane not visible from v if s were the only obstacle present. Let s_s and s_f be the endpoints of s. The region $sr(v, s)$ can be expressed as the intersection of three half-planes h^0, h^1 and h^2. h^0 does not contain v and its boundary is the supporting line of s, h^1 contains s_s and its boundary is the supporting line of vs_f, h^2 contains s_f and its boundary is the supporting line of vs_s. Then, for each viewpoint v_i the union $\bigcup_{s \in S} sr(v_i, s)$ is painted with the RGBA color whose bit in position i is the only one equal to 1, where s is a segment belonging to the set of segment obstacles S.

Each union is rendered using the OpenGL logic operation *OR* for combining the colors. By doing this, we ensure that the bits of the RGBA channel of a pixel p represent the viewpoints of V visible from p. A 1 in the bit in position i of a pixel p indicates that the

viewpoint v_i is not visible from p. Otherwise v_i is visible from p. Since n unions of m triangles are created and painted, texture MVM is created in time $O(nm + nP_{W^2})$ where P_{W^2} is the time spent in rendering a texture of W^2 pixels.

Figure 3.2 shows the scene from which the multi-visibility map is computed, the texture MVM with the combination of all shadow regions and finally the obtained visibility is also represented. In this and all subsequent figures, the multi-visibility map indicates from how many viewpoints a pixel is visible using a black to white gradation (black means that a pixel is not visible from any viewpoint). On the other hand, the shadow regions are painted using red, green and blue for every viewpoint, respectively (for the examples we only use three viewpoints to make the visualization easier), and the combinations.

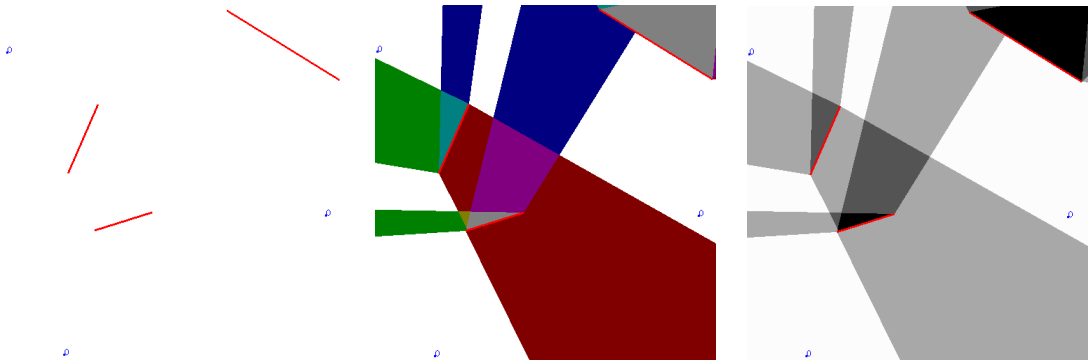


Figure 3.2: *Visibility map (center image) and visibility (right image) computed using the GPU from the scene in the left image.*

Generic obstacles

The set S of generic obstacles is represented by a binary image placed in a texture OBS. A black pixel corresponds to a part of an obstacle whereas a white one to the free space. The screen coordinates of the viewpoints of V are stored in the texture called $VPos$.

Figure 3.3 shows a scheme of the algorithm explained in the following paragraphs and the result of applying it to every pixel of the screen in a parallel fashion using the GPU.

The texture MVM encoding the multi-visibility map is created by rendering to the texture a screen-aligned quad using a pixel shader that requires the textures $VPos$ and OBS as parameters. The shader computes the visibility of a pixel p from each viewpoint v_i and stores this information into the i bit of the RGBA channel of $MVM[p]$. The process runs as follows. If $OBS[p]$ is black then p is not visible from v_i because p is a part of an obstacle. Otherwise, the pixels t intersecting the segment $\overline{pv_i}$ are tested in an incremental

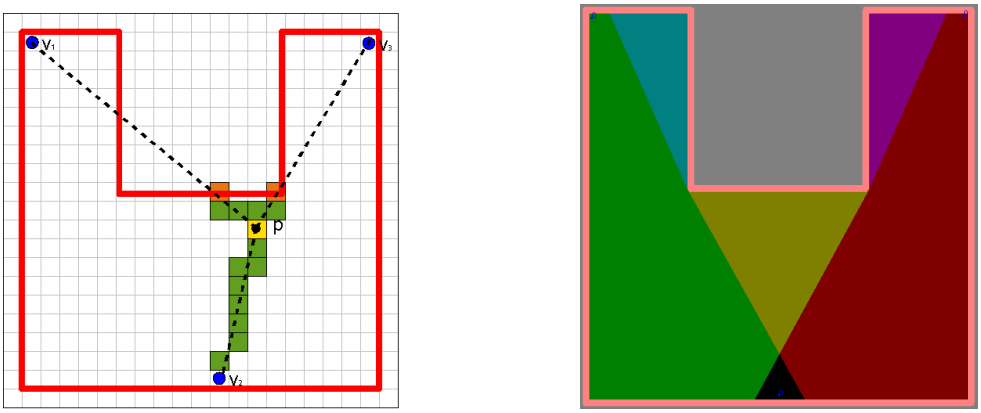


Figure 3.3: *The left image shows how the algorithm works. For every pixel p (yellow) the algorithm performs a double loop looking for obstacles between p and all viewpoints v_i. If the segment connecting p and v_i contains an obstacle pixel, then v_i is not visible from p and this information is stored in the position of p in the color buffer (right image).*

way. If a pixel t satisfying that $OBS[t]$ is black is found, then p is not visible from v_i ($\overline{pv_i}$ intersects an obstacle stored in OBS) and therefore the process is stopped. Otherwise p is visible from v_i. Since the per-pixel cost is $O(nA_W)$, the texture MVM is created in $O(nA_W P_{W^2})$ time, A_W being the time spent to access W pixels of OBS and P_n the time needed to render n pixels.

Figure 3.4 shows an example of scene from which the visibility is computed, the texture MVM with the combination of all shadow regions and finally the obtained visibility.

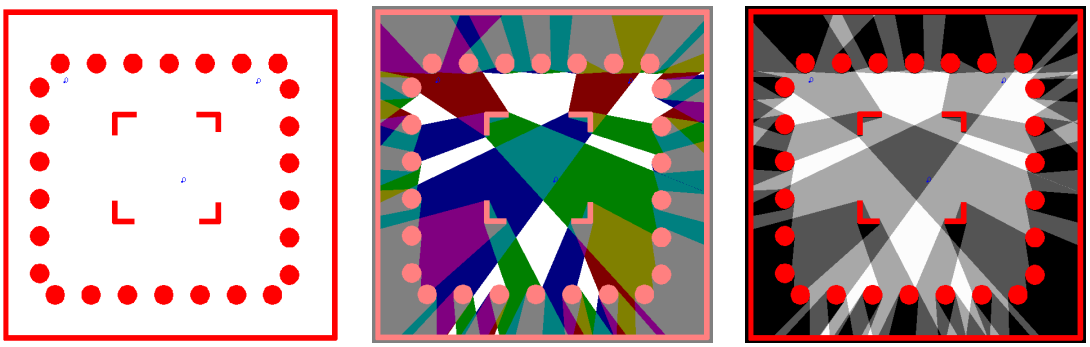


Figure 3.4: *The left image shows the scene before the computation while the center and right images show the shadow regions and the multi-visibility map, respectively.*

Running time analysis

Figure 3.5 shows three plots with the running time when the number of viewpoints is incremented for a screen size of 200x200, 500x500 and 650x650 pixels, respectively. Each of these plots contains six data series, five of them correspond to five distinct number of segment obstacles and the last one represents the time spent by the generic obstacles version of the algorithm. We can observer that, obviously, running time is incremented when the number of segment obstacles increases and also when the screen size becomes larger. Notice that the computation of the visibility using generic obstacles is much less dependent on the size of screen than using segment obstacles. Moreover, it is always faster than using 5000, 10000 or more segment obstacles. This indicates that when S has more than 1000 segments, it is probably a good idea to change the set of segment obstacles by an image of them and use them as generic obstacles. We can also observe that the larger the screen size, the more profit we get if the set of segment obstacles is transformed to generic obstacles.

3.3.2 Multi-visibility map of viewpoints with restricted visibility

A point v has restricted visibility when its visibility region is constrained within an angular region or/and with limited range [ABHM05]. As in the unrestricted case, we want to obtain the texture MVM. However, we must consider now the restriction in distance and angle for each viewpoint v_i.

Again we can consider two different possibilities. The restricted visibility can be studied considering either segment obstacles or generic obstacles stored in a binary image.

Segment obstacles

For every viewpoint v we must paint the union of the shadow regions related to v using its associated bit of color. Once this has been achieved, we must also paint, using the same color, the exterior part of the restriction visibility zone of v in order to finally obtain the visibility map for v.

The algorithm proceeds as in the unrestricted case (see previous section), thus doing this process for every v_i we obtain the texture MVM.

Figure 3.6 contains an example of restricted visibility computation using the GPU.

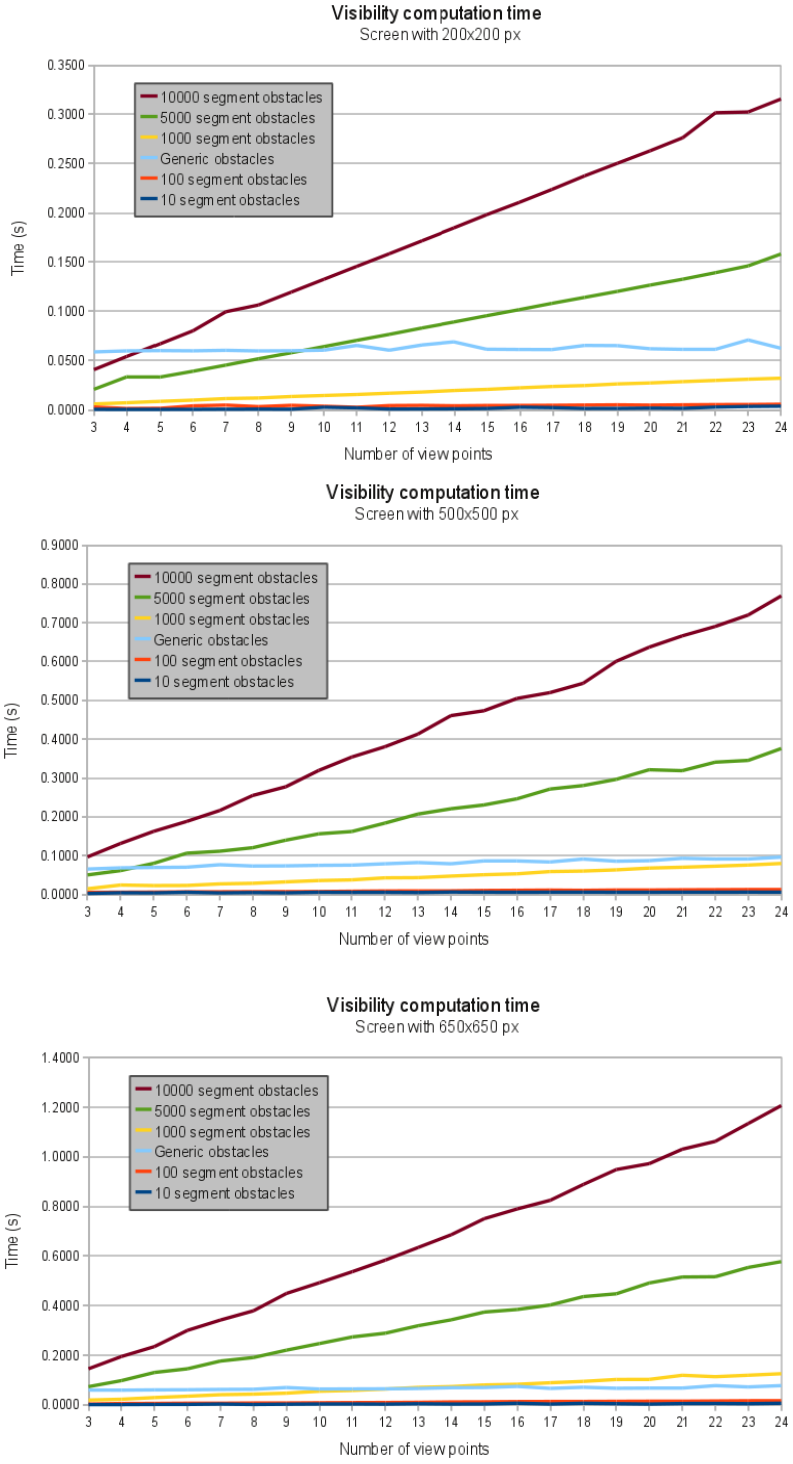


Figure 3.5: *Plots from top-left to bottom-right show the running time for the visibility computation with screen size of 200x200, 500x500 and 650x650 pixels, respectively, when viewpoints are considered.*

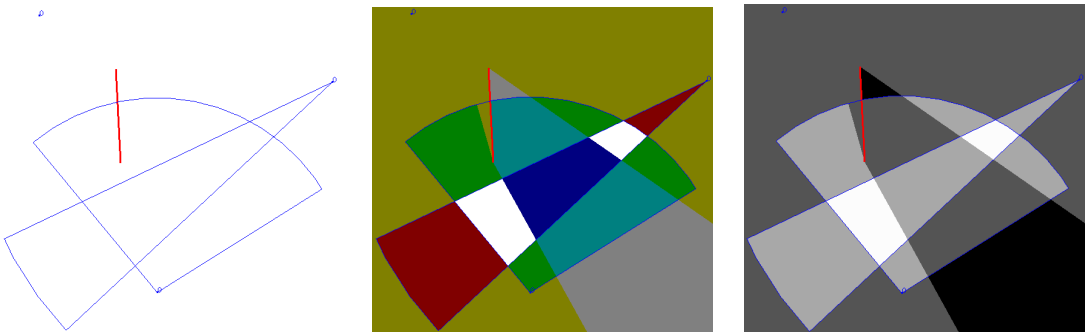


Figure 3.6: *The left image shows the scene before the computation while the center and right images show the shadow regions and the multi-visibility map, respectively. The empty blue regions indicate the restriction of the viewpoints.*

Generic obstacles

In this case, the pixel shader receives two additional parameters related to current viewpoint v_i: its range in pixels $range_{v_i}$ and two points r_1 and r_2 in image coordinates. v_i has visibility in the angular region defined by the rays $\overline{v_i r_1}$, $\overline{v_i r_2}$ and at a maximum distance of $range_{v_i}$.

Before running the algorithm explained in Section 3.3.1 we first test if the current pixel p is inside the visible range of v_i. If not, p is not visible from v_i and the process stops.

See an example in Figure 3.7.

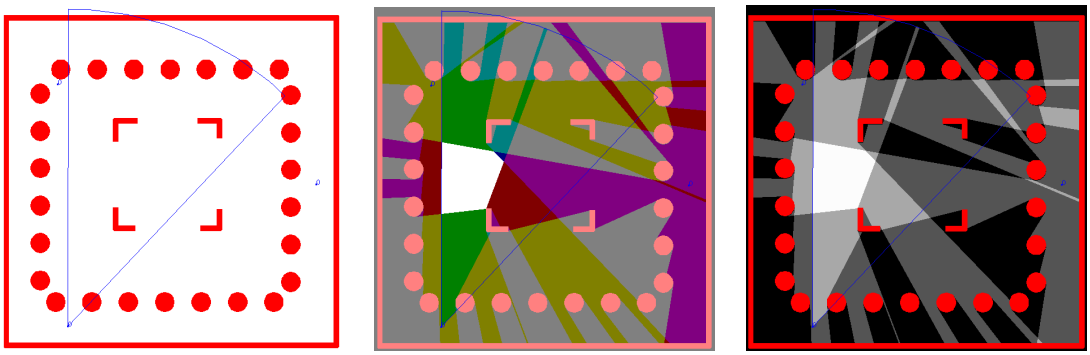


Figure 3.7: *The left image shows the scene before the computation while the center and right images show the shadow regions and the multi-visibility map, respectively. The empty blue regions indicate the restriction of the viewpoints.*

3.3.3 Multi-visibility map of viewpoints with power of emission

In some applications, for example WiFi routers placement, the signal of a viewpoint v can cross a certain number P_v of segment obstacles . This number P_v represent the *power of emission* of v. In what follows, we explain how we compute the visibility map for the two cases of obstacles.

Segment obstacles

If the signal of a point v can cross at most P_v segment obstacles, its shadow regions have to be computed by taking into account this information. We use the stencil buffer. To compute the shadow regions of v, we construct the shadow region for each segment obstacle s_i $sr(v, s_i)$ and the stencil buffer is incremented by one. We only paint the pixels having a stencil value greater than P_v in order to obtain a shadow only in points where the signal emitted by v has crossed at least $P_v + 1$ obstacles.

In Figure 3.8 we can see how the visibility map changes when the power of emission of the viewpoints is incremented.

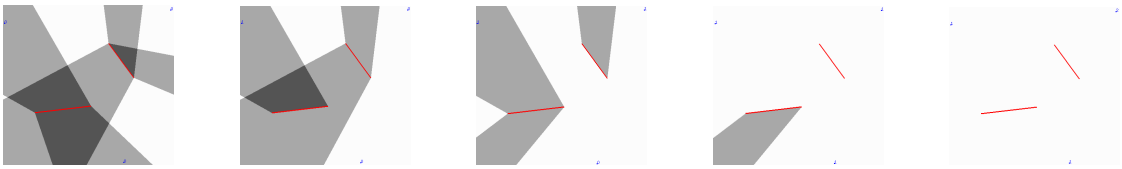


Figure 3.8: *From left to right, the power of emission of the viewpoints is incremented gradually, and the visibility map changes accordingly.*

Generic obstacles

In this case, the pixel shader receives the power of emission P_{v_i} as a number of pixel units for the current viewpoint v_i as an additional integer parameter.

Some tests have to be included in the algorithm explained in Section 3.3.1 to check if the current pixel p is visible from v_i. Another variable called *obstacle_pixels* is used (it is initialized to 0 before the loop begins). When a pixel t of the segment is over an obstacle pixel, *obstacle_pixels* is incremented by one and when *obstacle_pixels* $> P_{v_i}$, p is not visible from v_i.

In Figure 3.9 we can see how the visibility map changes when the power of emission

of the viewpoints is incremented.

Figure 3.9: *From left to right, the power of emission of the viewpoints is incremented in a gradual manner, and the visibility map changes accordingly.*

Adding *opacity* to generic obstacles

Instead of incrementing the variable *obstacle_pixels* by one every time an obstacle pixel is reached by the segment $\overline{pv_i}$, it could be interesting to increment it by a factor determined by the actual opacity or resistance of the current obstacle pixel.

Following this idea, the opacity of each obstacle pixel is saved in the red and green channels of the color buffer. By using this new approach, every time a pixel p is tested, *obstacle_pixels* is incremented by its opacity Then, as in the case of obstacles without *opacity* value, when *obstacle_pixels* $> P_{v_i}$ the signal emitted by v_i is not visible from the current pixel p.

Figure 3.10 shows an example containing an obstacle image with color gradation. An obstacle pixel is less *transparent* to the signal emitted by the viewpoints when it presents a higher content of red component on its color. A red pixel has the maximum permitted opacity.

3.3.4 Multi-visibility map from view segments

In this section, the computation of multi-visibility maps from view segments is described for the two variants of visibility: the so-called strong and weak. As in the case of viewpoints, we can have two kinds of obstacles: a set S of segment obstacles or an image representing more generic obstacles. The visibility computation taking into account view segments together with generic obstacles at the same time will be studied and implemented as future work.

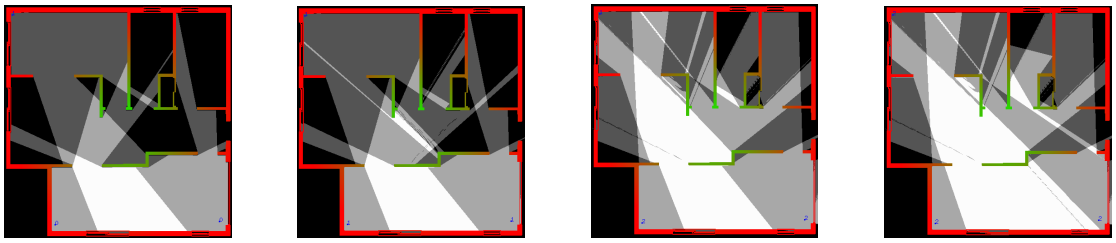


Figure 3.10: *From left to right, the power of emission of the viewpoints are incremented gradually, and the visibility map changes accordingly. The color of the obstacle indicate its opacity with respect to the signal emitted by viewpoints: green is used for totally* transparent *objects while red indicates the maximum opacity.*

Let V be a set of n view segments and S be the set of m segment obstacles. A view segment v_i and an obstacle segment s_j are composed by two endpoints v_{i_s}, v_{i_f} and s_{j_s}, s_{j_f}, respectively. There are four distinct lines ℓ connecting one endpoint of v with one endpoint of s. For any of the lines ℓ we define ℓ^O as one of the two halfplanes defined by ℓ containing the object O and $\ell^{\sim O}$ the one not containing O. The line containing the obstacle s is defined as ℓ_s.

In order to compute the shadow region defined by the view segment v and the segment obstacle s taking into account strong visibility one can do the following. The four distinct lines connecting v and s are computed. We only use two of the lines ℓ which contain one endpoint not belonging to ℓ at each side. Once the two useful lines ℓ_1 and ℓ_2 are known, we obtain the expected shadow region by making an intersection between the three halfplanes ℓ_1^s, ℓ_2^s and $\ell_s^{\sim v}$ (see Figure 3.11).

Otherwise, if we want to compute the shadow region from v and s by using weak visibility, the useful lines ℓ_1 and ℓ_2 are the two not selected for the strong visibility, and the shadow region is computed as the intersection between the halfplanes ℓ_1^s, ℓ_2^s and $\ell_s^{\sim v}$. In the special case of s and v forming a triangle instead of a quadrilateral being s inside this triangle, all points of the plane are weakly visible from v. (see Figure 3.12).

The only restriction in both cases is that v can not touch or intersect s, and v and s can not be aligned.

Therefore, in order to compute the visibility map from a view segment v and a set of segment obstacles S we only have to construct the union of all shadow regions obtained before, taking into account that the two useful lines ℓ are selected depending on the type of visibility wanted.

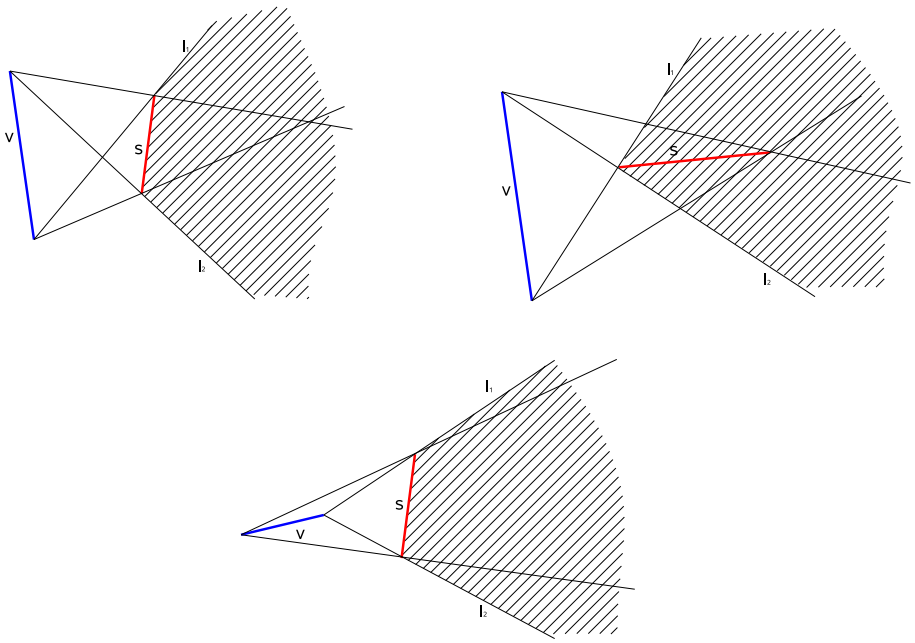


Figure 3.11: *The filled regions represent the not visible points from v when strong visibility is considered. Points outside the dark region see the whole segment v. The algorithm works properly without bothering about the relative position of v with respect to s.*

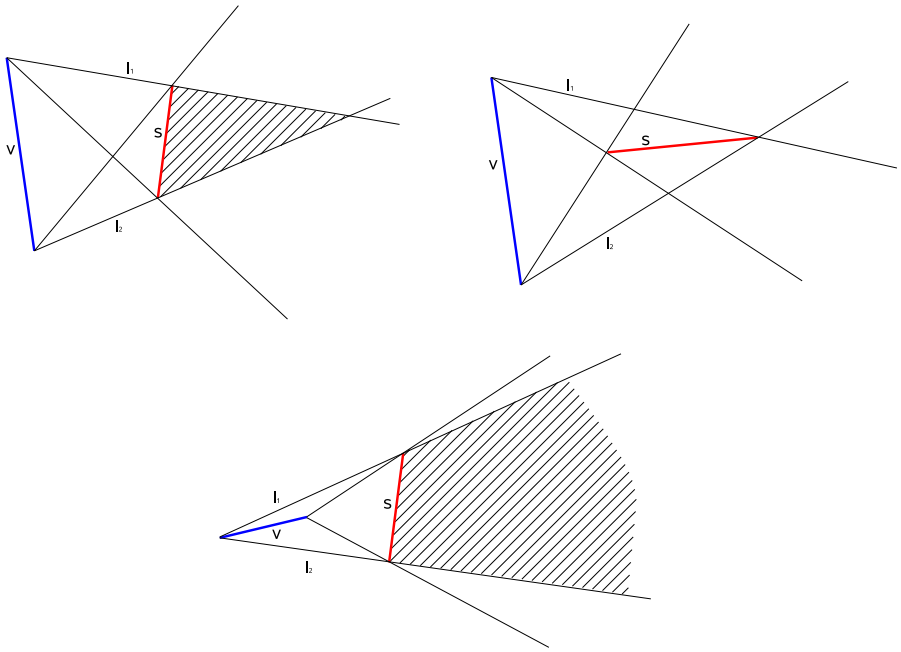


Figure 3.12: *The filled regions represent the not visible points from v when weak visibility is considered. The points outside the dark region see at least one point of segment v. The algorithm works properly without bothering about the relative position of v with respect to s.*

Implementation

The algorithm, implemented using the Cg language on the GPU together with the OpenGL API, is very similar to the one explained for the case of viewpoints and segment obstacles in Section 3.3.1.

The texture MVM encoding the multi-visibility map is created using a simple image based algorithm too. For each view segment v_i the union $\bigcup_{s \in S} sr(v_i, s)$ is painted with the RGBA color whose bit in the position i is the only one equal to 1, where s is a segment belonging to the set of segment obstacles S and $sr(v_i, s)$ is the zone of the plane not visible from v_i (its shadow region). The discretized shadow region $sr(v_i, s)$ is computed by intersecting the three halfplanes mentioned before (therefore there is a previous step to obtain the two useful lines ℓ). This intersection is reached by using the stencil buffer. When we paint each of the halfplanes, all pixels belonging to the current halfplane increment their stencil value by one. The only pixels actually painted are those whose stencil value is three after the three halfplanes are painted. This process is repeated for every segment obstacle and their shadow regions created by v_i are joined simply by painting them on the same color buffer.

All the previously described unions are rendered using the OpenGL logic operation *OR* for combining their colors. This ensures that the different activated bits of the RGBA channel of a pixel p represent the distinct view segments of V non-visible from p. Each shadow region can draw up to three times the whole screen due to the intersection of the halfplanes and there are $n \times m$ shadow regions, thus texture MVM is created in time $O(3nmP_{W^2})$ where P_{W^2} is the time spent in rendering a texture of W^2 pixels.

Figure 3.13 contains some examples of weak and strong multi-visibility maps obtained with our implementation. As we can see in Figure 3.14 the implementation also allows us to mix viewpoints and view segments in the same set V. It is even possible to compute the visibility from a set of view segments, some of them using weak visibility and others using strong visibility (see Figure 3.15).

Running time analysis

Figure 3.16 shows three plots with the running time when the number of view segments is incremented for a screen size of 200x200, 500x500 and 650x650 pixels, respectively. Each plot depicts three data series for every kind of considered visibility: strong and weak. These data series are used to show the running time for scenes with 1000, 5000 and 10000 segment obstacles, respectively.

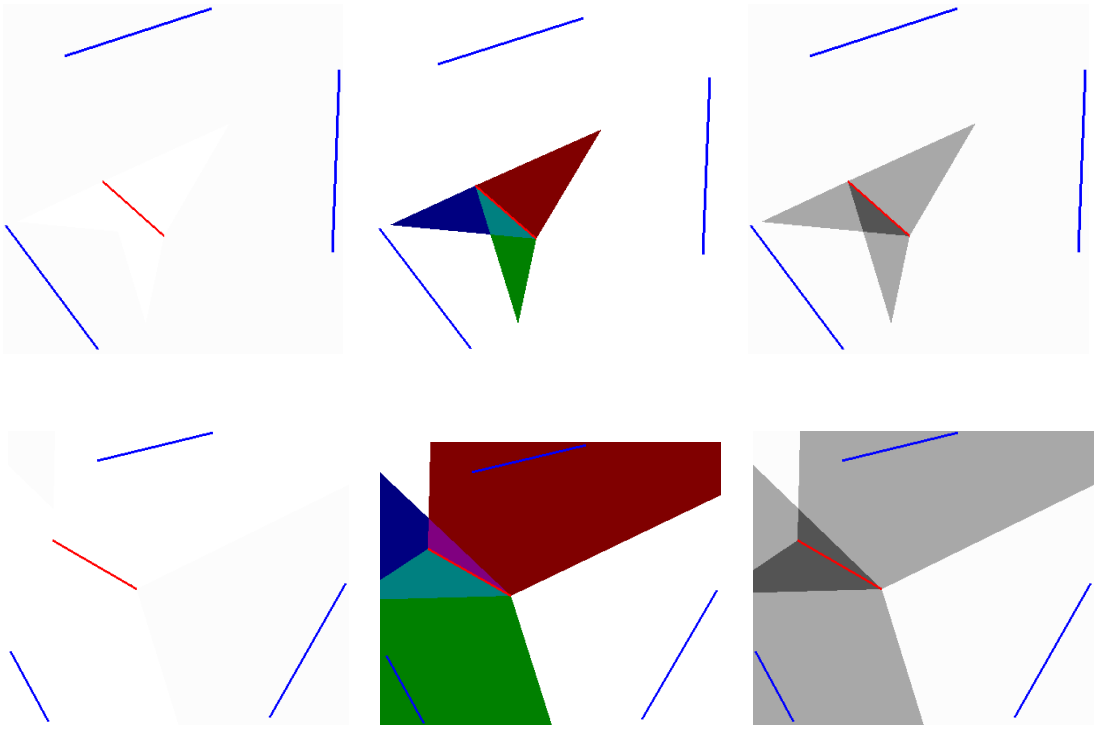


Figure 3.13: *From left to right, the images show the scene, the shadow regions and the visibility computation, respectively. The upper images are computed by using weak visibility while the lower ones use strong visibility.*

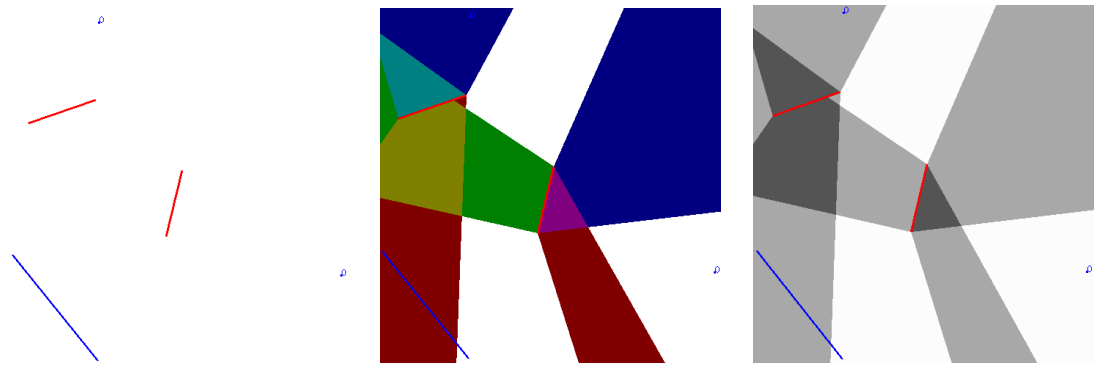


Figure 3.14: *From left to right, the images show the scene, the shadow regions and the visibility computation, respectively. The set V is now a mix of points and segments.*

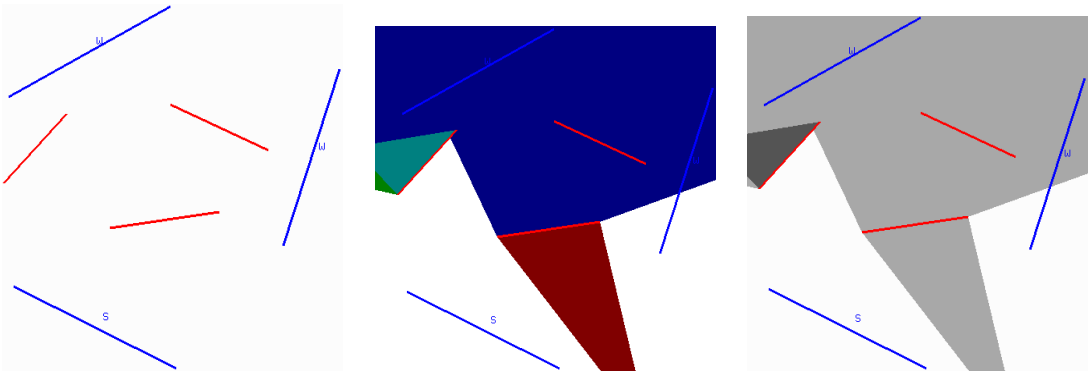


Figure 3.15: *From left to right, the images show the scene, the shadow regions and the visibility computation, respectively. The set V is now a mix of segments with strong (S) and weak (W) visibility.*

3.4 Multi-visibility on terrains using the GPU

We consider that the projection $\mathcal{T}^*$ of the terrain $\mathcal{T}$ is approximated by a rectangular grid of size $W \times W$, the grid coordinates of the viewpoint projections are stored in a texture $VPos$ of $1 \times n$ pixels, and the heights of the viewpoints are stored in a texture VH of $1 \times n$ pixels. In this way we can solve the problem by using a similar approach to the two-dimensional case explained in Section 3.3.1.

First, the algorithm, by applying two different pixel shaders, renders the terrain twice using an orthographic projection and a bird's-eye view. The first pixel shader obtains a texture H of size $W \times W$ where the RGBA value of each pixel p^* stores the height of its corresponding point p on $\mathcal{T}$. The second one obtains a texture F of the same size, where the RGBA value of each p^* stores the index of its corresponding face on $\mathcal{T}$.

Next, a quad of size $W \times W$ and aligned with the rectangular grid approximating the terrain is rendered to the texture MVM by using another pixel shader. The shader requires the textures H, F, $VPos$ and VH as parameters. For each pixel p^*, the pixel shader determines if its corresponding point p on $\mathcal{T}$ is visible from each viewpoint v_i and stores this information into the bit i of the RGBA channel of the texture MVM. The process runs as follows. The pixels t^* intersecting the segment of endpoints p^* and $VPOS[i]$ are visited in an incremental way. Given a pixel t^*, the height $h'(t^*)$ of its corresponding point on $\overline{pv_i}$ is computed by a linear interpolation of the heights $H[p^*]$ and $VH[i]$. When a pixel t^* satisfying $F[t^*] \neq F[p^*]$ (a face can not be occluded by itself) and $H[t^*] > h'_{t^*}$ is found, it means that p is not visible from v_i ($\overline{pv_i}$ intersects $\mathcal{T}$) and the process is stopped. Otherwise, p is visible from v. In this way the texture MVM is created

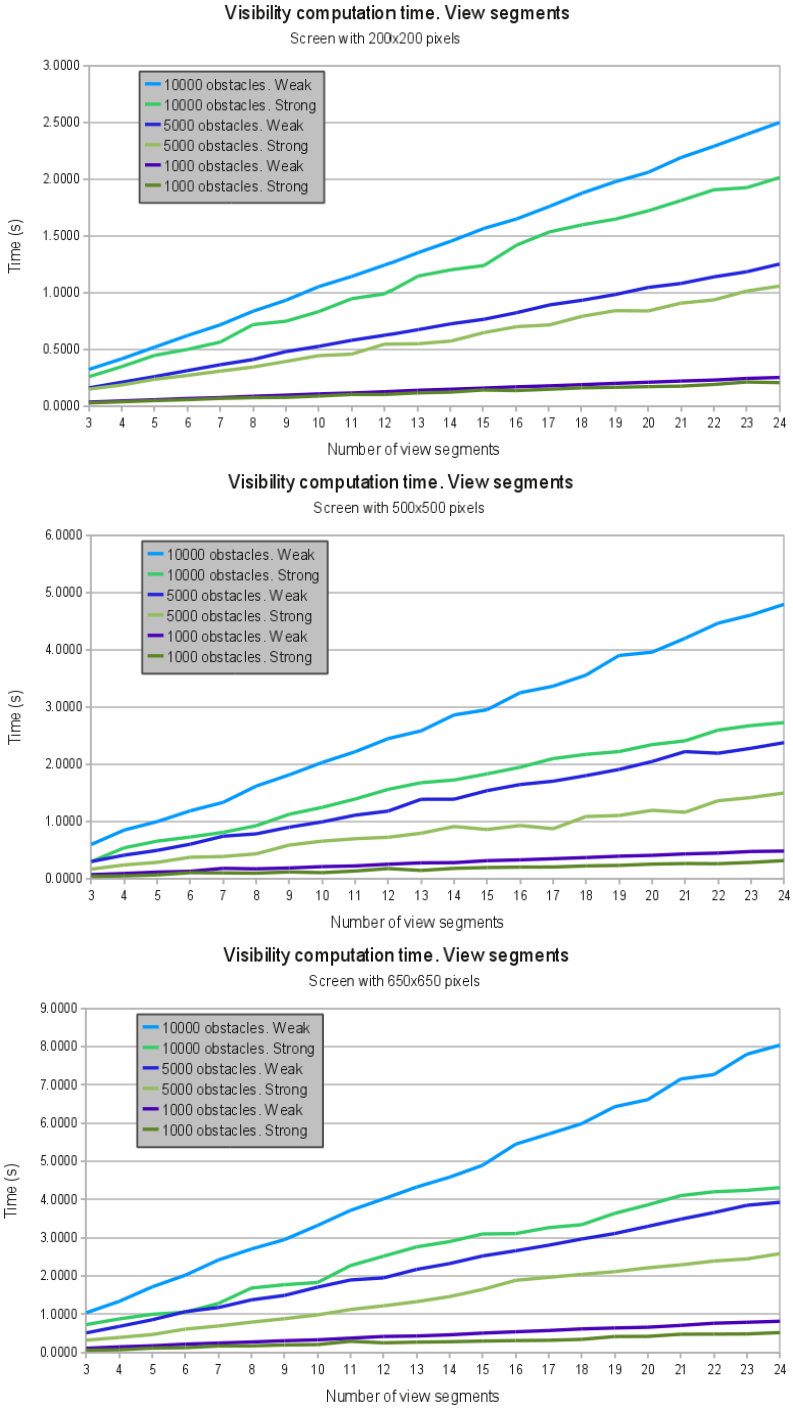


Figure 3.16: *Plots from top-left to bottom-right show the running time for the visibility computation with screen size of 200x200, 500x500 and 650x650 pixels, respectively, when view segments and segment obstacles are considered. Both strong and weak visibility are used.*

in $O(nA_W P_{W^2})$ time. Consequently, the overall complexity of computing a multi-visibility map on a terrain is the same as in the planar case with generic obstacles.

We can obtain the multi-visibility map on the terrain $\mathcal{T}$ by projecting the texture MVM to $\mathcal{T}$ using texture mapping (see Figure 3.17).

As in the planar case, we can also consider viewpoints with restricted visibility range (a sphere of a determined radius with center at v and/or angle (visibility cone with vertex v). We only need to modify the visibility criterion applied in the algorithm explained in the previous section to take into account whether point p on $\mathcal{T}$ is inside the visibility range of v.

3.5 Three-dimensional multi-visibility maps using CUDA

Our purpose is to develop a method which computes the visibility information from a set of viewpoints and a set of triangle obstacles placed on a $\mathbb{R}^3$ space. Our proposed algorithms have been implemented using CUDA, the new GPGPU language designed by NVIDIA. The main reason for the use of CUDA is that Cg deals always with image-based solutions that can be hard to apply to a three-dimensional environment involving volumes. In contrast, CUDA has direct support to easily manage volumetric spaces and is a more generic platform useful for computing solutions without the need of using a computer graphics API.

3.5.1 CUDA implementation

As in the two-dimensional approach, our goal is not to obtain an exact solution for the visibility on the space, but only a discretization of the solution. For this purpose we subdivide the wanted portion of the space into a uniform grid of voxels of side W, and the visibility of every one of these voxels is computed by a thread of our CUDA kernel in a parallel fashion. Obviously, if we want to obtain a more precise approximation, W has to be larger. The larger that W is, the more threads the kernel has to process and the more time the whole process needs.

To the best of our knowledge, no algorithm for computing the discretization of the multi-visibility map from a set of viewpoints exists. However, our work is closely related to ray tracing algorithms using the GPU [PBMH02, STK08, ZH10], from which we have borrowed some techniques.

The pseudocode shown on Algorithm 1 is the kernel responsible for computing the

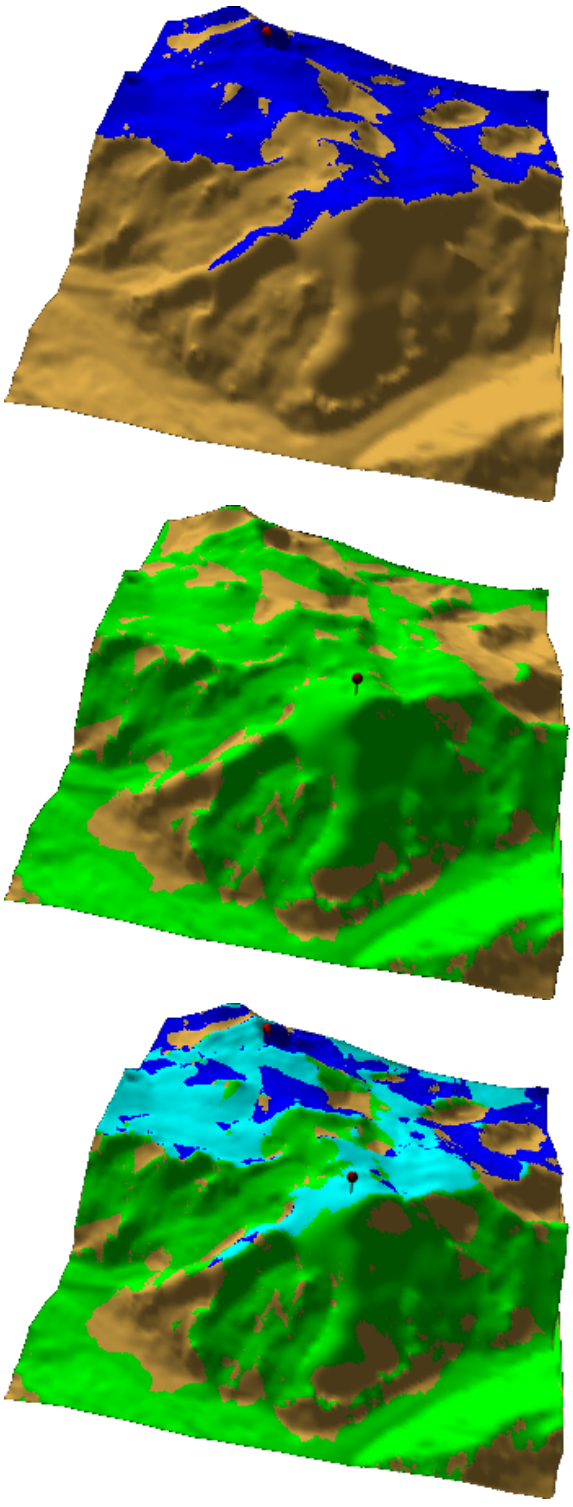


Figure 3.17: *The two first images show the visibility map from two different viewpoints. The right one shows the multi-visibility map.*

multi-visibility of every voxel. The visibility of a single voxel is computed by a single CUDA thread and all such threads are executed in a parallel fashion. The result is stored in the array MVM containing as many elements as the number of threads and voxels of the discretization. Every element encodes the visible points of V from its corresponding voxel, taking into account the set of triangle obstacles.

In a brute force algorithm we would need to compute the intersection between n segments with m triangles, for each of the voxels present in the discretization, where n is the number of viewpoints and m is the number of triangle obstacles. Therefore $n \times m$ intersection tests would be performed inside every thread of the kernel, although most of these intersection tests are going to be negative.

In order to avoid this huge number of unnecessary intersections, a voxelization of the triangles can be used. There are many possible acceleration data structures to choose from: bounding volume hierarchies [WBS07, Mah05], bsp-trees [HKBv97, SS92, CF89], kd-trees [PGSS07, HSHH07], octrees [SVNB99], uniform grids [NO97, AW87, KS09], adaptive grids [KS97], etc. We chose uniform grids for two reasons. First, many experiments have been performed using different acceleration data structures on different scenes (for an excellent recent study see [ZH10]). From these studies no single acceleration data structure appears to be the most efficient; they all appear to be within a factor less than two of each other. Second, uniform grids are particularly simple for hardware implementations since accesses to grid data structures require constant time, and the code for implementing the grid traversal algorithm in a uniform grid is straightforward.

Our method is similar to the one proposed in [KS09] and it tries to obtain an efficient (both in time and space) data structure which stores the triangles that intersect every voxel of the uniform grid. Basically, we need to create an array which consequtively stores the triangles intersecting the voxels (actually, it contains references to the triangles, not the triangles themself). The list of triangles intersecting the voxel in position i are put exactly after the triangles intersecting the voxel in position $i-1$ (see Figure 3.18). Thus this array will contain the *triangles voxelization* and is denoted as TV.

Since CUDA is not capable of managing dynamic memory, we first have to count the number of triangle references that we will need to store in order to allocate the exact quantity of memory needed by TV. Moreover, without the possibility of using dynamic memory, the list of triangles of each voxel would have to be of the same size. However, a much better option for saving space is to have another array where each element represents a voxel *vox* and contains one index adressing where the list of triangles intersecting *vox* start in the array TV. Therefore, this new array contains the *triangles voxelization indices*

and it is represented by TVI. Using these two arrays together, we can determine if the voxel vox in the array position i intersects any of the triangles of the scene. If $TVI[i-1] < TVI[i]$ then there are triangles intersecting vox (in the special case of $i = 0$, the test is $0 < TVI[0]$). The number of triangles intersecting the voxel in position i is $TVI[i]-TVI[i-1]$ and their references start at position $TVI[i-1]$ in the array TV.

The proposed algorithm needs an additional input parameter, H, which is the side of the voxelization used to store the triangles of T (not necessarily $H = W$). It is explained in detail in the next paragraphs, and is executed only once before Algorithm 1 begins, as a pre-process step.

First of all, the list of triangles is uploaded to the array *triangles* in the CUDA global memory.

The first CUDA kernel is responsible for counting the number of triangles that intersect every voxel vox and it stores this information in the array TVC of size $H \times H \times H$. Every distinct thread computes the voxels that intersect a different triangle t, thus the triangles are treated in parallel. When all the voxels that intersects t are found, the kernel calculates their indices and increments by one the array TVC at their positions.

The second kernel implements a parallel prefix sum algorithm for obtaining the array TVI of size $H \times H \times H$ where the element i contains the sum $\sum_{j=0}^{i} TVC[j]$. Using the array TVI we can determine how many triangle references we will have ($TVI[m-1]$) and the number of triangle references we have to store for every voxel. Before the third kernel is executed, the array TV of size $TVI[m-1]$ is allocated in the CUDA global memory.

The last CUDA kernel implements the method that fills the array TV where the element i, representing the voxel vox, will contain the list of references to the triangles that intersect vox. The kernel receives the arrays *triangles* and TVI as input parameters and each thread (representing a triangle t) is responsible for finding (again) all the voxels that intersect t and store its references in the correct position of TV.

Figure 3.18 shows an example of how all these arrays and kernels work. The example uses a two-dimensional space for simplicity purpose. The translation to the three-dimensional version is straightforward.

We would save time if we could store the arrays *triangles*, TVC, TVI and TV in the constant or shared memory, however this is not possible because the size of a normal scene

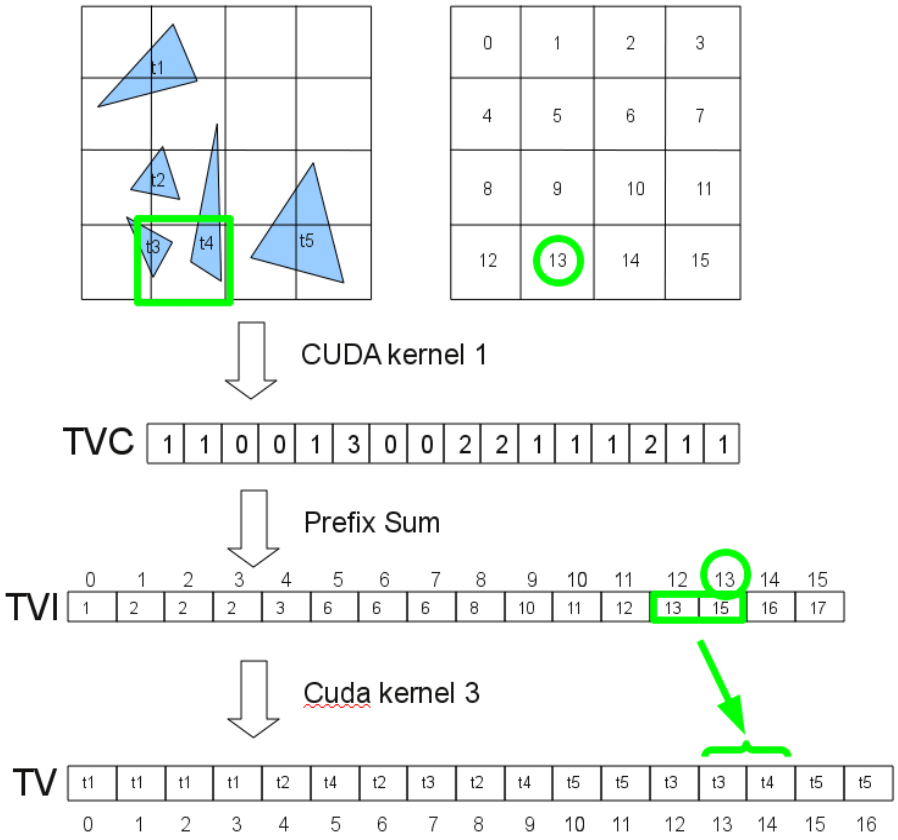


Figure 3.18: *The scheme of the voxelization of the triangles. The upper images show the obstacles, the voxels forming the uniform grid and the index of every voxel. The green squares indicate the voxel we are interested in and where we can find the triangles it intersects. The green circles indicate the index of the voxel of interest.*

exceeds the (small) space dedicated to constant memory. Moreover, neighbor threads do not necessarily access neighbor elements of TVI or TV, which enormously difficult to put the results in the shared memory. However, we can upload the triangles to the CUDA global memory using a *float3* array instead of a *float* array. This saves a significant amount of time when accessing the triangles list and later, when the information stored in TV and TVI is used to compute the visibility.

Although the running time of our method is longer than the algorithm proposed by J. Kalojanov and P. Slusallek in [KS09], the implementation is easier and the running times keep on beeing very reasonable taking into account that this is only a pre-process step, as we can see in Figure 3.19. The number of triangles of the used scenes goes from 1000 of our columns scene to 10 millions of the Thai Statue. Observe that the time is not linear dependent with the number of triangle obstacles. Moreover, it depends on how the triangles are located and distributed inside the scene.

An important part of the total time is spent in performing two atomic operations that are totally necessary to compute the voxelization of the obstacles in this way. These atomic increments avoid writing conflicts when two or more threads try to update the same position in the result arrays. This fact makes the code easier to understand and, moreover, it avoids the necessity to implement a parallel sorting algorithm in CUDA, as in [KS09].

For a fast computation of the voxel-triangle intersection test, the algorithm proposed by Akenine-Moller in [AM01] can be used. Eventually we realized that a complete triangle-box intersection test is not really necessary, as J. Kalojanov and P. Slusallek suggest in [KS09]. Therefore, a *simplified* and *conservative* intersection is used. By doing this, more triangles than the ones actually intersecting the voxel can be found. However, the time spent in storing and accessing these *useless* triangles in the visibility computation is less than the time wasted in computing an exact intersection test.

Once the voxelization of the triangles has been obtained, the algorithm for computing the multi-visibility map from the set of triangles and the set of viewpoints can be executed (see Algorithm 1).

The method works as follows. For the current voxel vox (with its central point p), it constructs a segment $\overline{pv_i}$ for every $v_i \in V$. Each of these segments traverse the voxelization represented by the array TVI in an incremental way using the fast voxel traversal algorithm presented by J. Amanatides and A. Woo in [AW87]. If the segment $\overline{pv_i}$ reaches a voxel of TVI which contains triangles, then an exact intersection test with all present triangles must be done. As soon as any one of these intersection tests is positive, then we

Algorithm 1: 3DPointVisibility CUDA kernel

Input: Set of viewpoints V, Number of points n, set of triangles T, Number of triangles m, array TVI, array TV, grid side H

Output: Multi-visibility map MVM.

```
index ← voxelIndex(threadPosition,BlockSize,GridSize) ;
p ← voxelPosition;
visiblePoints ← 0;
for i = 0 to n do
    v ← V[i];
    vp ← segment(v, p);
    TVI_v ← voxelPosition(H, v);
    visible ← true;
    while visible and voxelInSegment(TVI_v, vp) do
        triangleList ← references from TV[TVI_v − 1] to TV[TVI_v];
        if nonEmpty(triangleList) then
            for every j in triangleList do
                triangle ← T[j];
                if intersection(vp,triangle) then
                    visible ← false;
                end
            end
        end
        TVI_v ← nextVoxel(H, vp);
    end
    if visible then
        visiblePoints ← visiblePoints + (1 << i);
    end
end
MVM[index] ← visiblePoints;
return MVM;
```

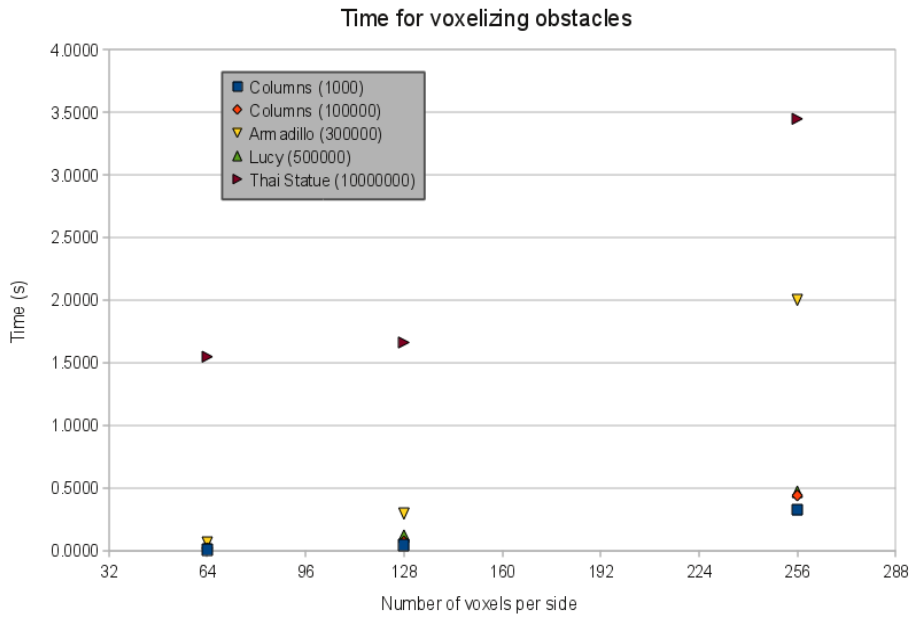


Figure 3.19: *The image shows the running time spent in the voxelization of the obstacles for some distinct scenes. The number of triangles of each scene is indicated next to its name.*

can conclude that v_i is not visible from p and the traversal of the segment $\overline{pv_i}$ is finished. If the process continues and the segment does not intersect any triangle on a non-empty voxel of TVI or it does not reach any non-empty voxel, then v_i is visible from p. Figure 3.20 contains a 2D example of the algorithm. The extension to the three-dimensional case is straightforward. The visibility information is stored in the array MVM using the single bit located in the position i of the *visiblePoints* variable. We choosed *visiblePoints* as an *unsigned integer* variable in order to make the modification of its single bits easier. Unsigned integers variables are normally composed by 32 bits, thus the algorithm is restricted to compute the multi-visibility of at most 32 viewpoints. This can be avoided by using a number of unsigned integers equal to $ceil(n/32)$ and modifying the algorithm accordingly. For every element of the array MVM, we would have to allocate a number of unsigned integers equal to $ceil(n/32)$ and carefully control which integer has to be modified in every step of the loop.

3.5.2 Restricted visibility

In order to take into account range restriction in the visibility of the viewpoints we have to slightly change the previously presented algorithm. Now, apart from set V, we also

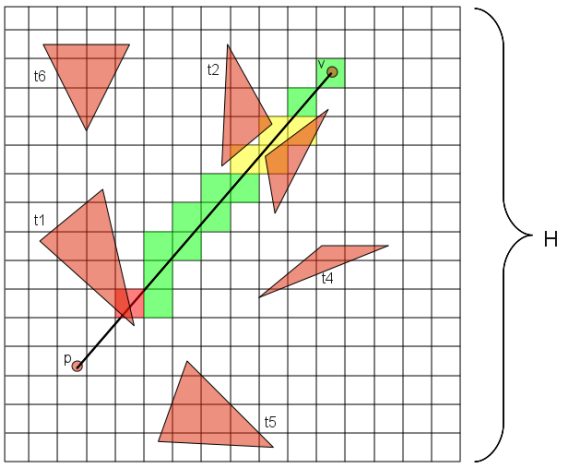


Figure 3.20: *The two-dimensional version of the incremental algorithm for finding the visibility of p from v. The voxels painted are the visited ones by the incremental algorithm. Green is used to represent voxels which do not contain intersections, while yellow denotes those voxels containing triangles that have to be tested. Finally, red voxels contain one or more triangles for which the test is positive (p is not visible from v). Notice that triangles t4, t5 and t6 are never tested since they do not intersect any voxel in the segment* $\overline{pv}$.

need an array containing the range restriction for the visibility of every viewpoint of V. We will define this array as VR, and $VR[i]$ will contain the distance from which the viewpoint v_i is not visible. The only special test that we have to include in the algorithm is to check, before testing the intersection between $\overline{pv_i}$ and any of the triangles of T, if $distance(p, v_i) < VR[i]$. If this is the case, the algorithm continues normally, otherwise p is not visible from v_i and the process can be stopped immediately.

3.5.3 Results

Finally, we present some images showing the computation of the visibility from a few viewpoints using two different scenes. Moreover, some figures showing the running times of the algorithm with distinct grid resolution and number of triangle obstacles are given. The tests are done on a computer equipped with a Intel Pentium 4 CPU at 3.20 GHz and a graphics card using a GeForce GTX 280 GPU. The results of every test are the mean running time of ten executions. For every one of these executions the viewpoints are randomly generated.

Figure 3.21 show an example of the visibility computed from one viewpoint and a scene with approximately 450 triangles. The scene used in Figure 3.22 has over 2000 triangles

and the visibility is computed from two different viewpoints. The right image on this figure shows the shadow regions from both viewpoints.

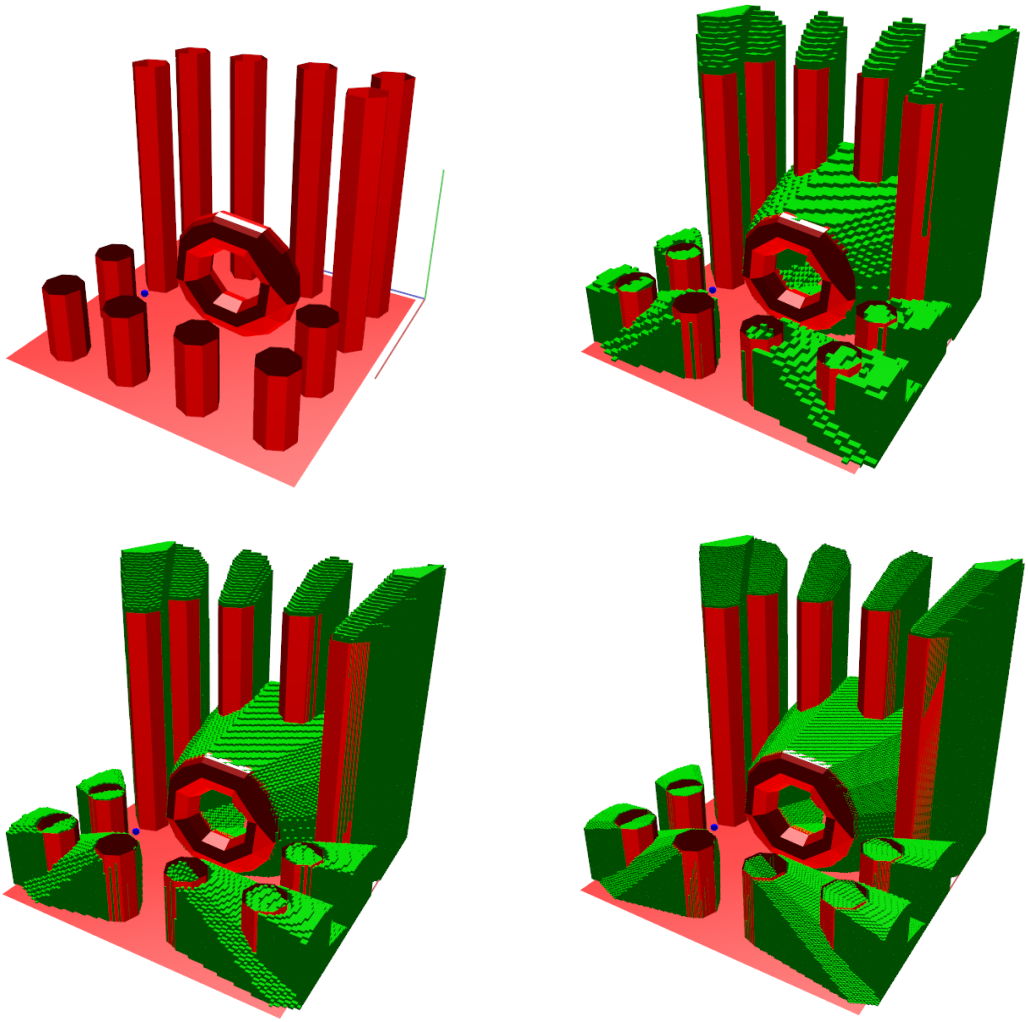


Figure 3.21: *The top left image shows only the scene (composed by 450 triangles) and the only present viewpoint (blue sphere) while the other ones show, from top left to bottom right, the visibility map every time with more detail.*

In Figure 3.23 we can see two different images containing the running times using 200 and 1000 triangle obstacles, respectively. Notice that by doubling the side of the voxelization, the time is increased by eight times approximately, which is the expected behaviour because the number of voxels is incremented by eight times too.

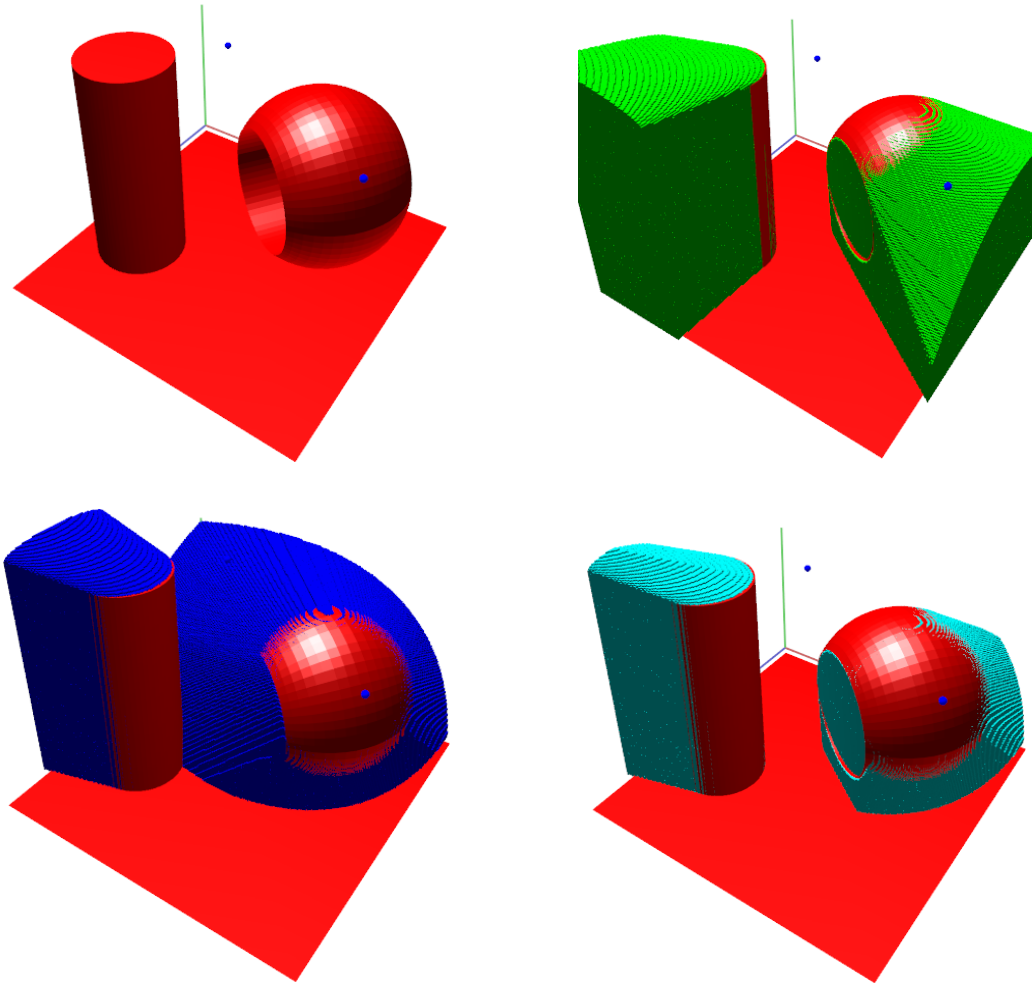


Figure 3.22: *From top left to bottom right, the images show the scene, the shadow regions from viewpoint 1 and viewpoint 2, and the shadow regions from both viewpoints, respectively.*

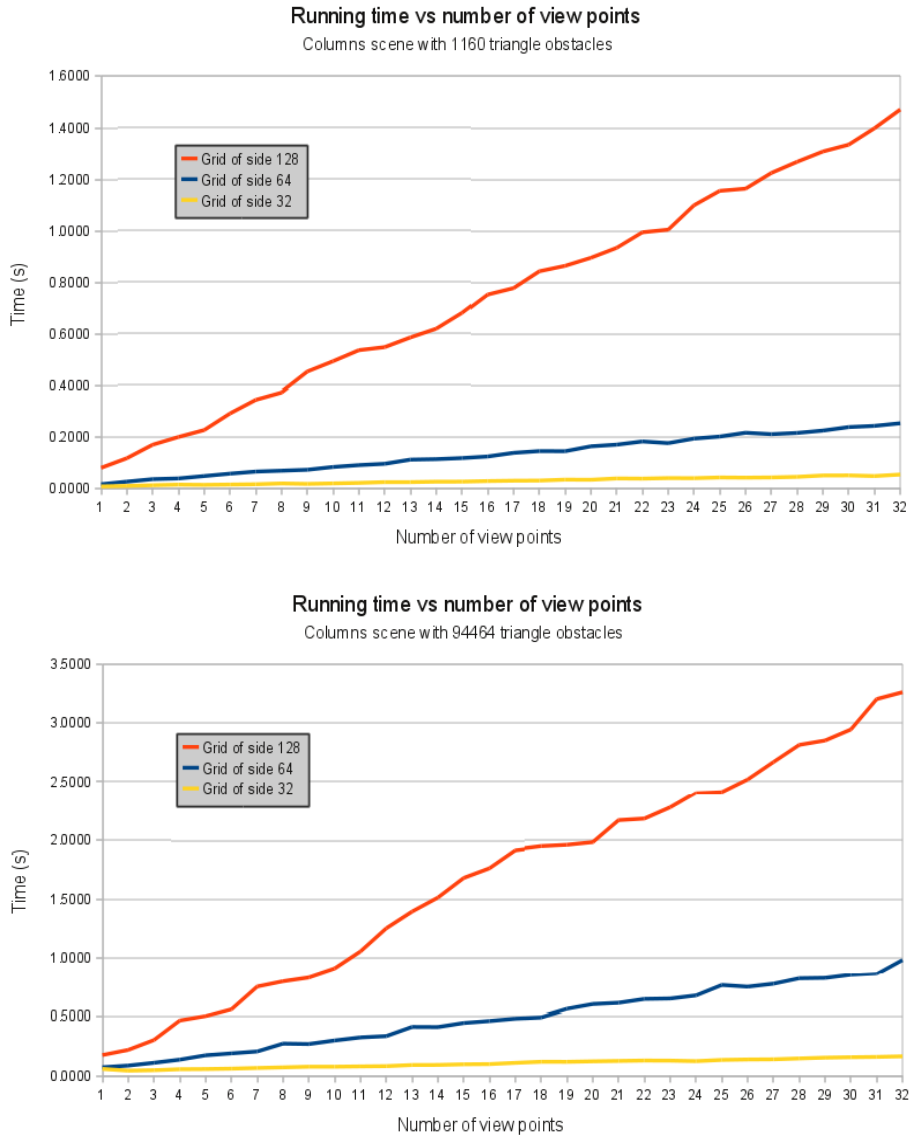


Figure 3.23: *The images show the running time for the 3D visibility computation using the explained CUDA method. Times on the top image are taken from a scene with 1000 triangles while times on the bottom image are based on the same scene with 100000 triangles.*

Chapter 4

2D and 2.5D good-visibility maps computation using the GPU

This chapter is structured as follows. First, our exact algorithm to compute good-visibility maps and its theoretical complexity are proved. Then a new algorithm for computing the depth map from a set of points using the GPU is described. Afterwards the design and implementation of our method to calculate a discretization of the good-visibility map from a set of viewpoints and a set of obstacles are presented. Finally, we expose variations on the visibility applied to the good-visibility map as well as some representative examples and a detailed running time analysis.

4.1 Introduction

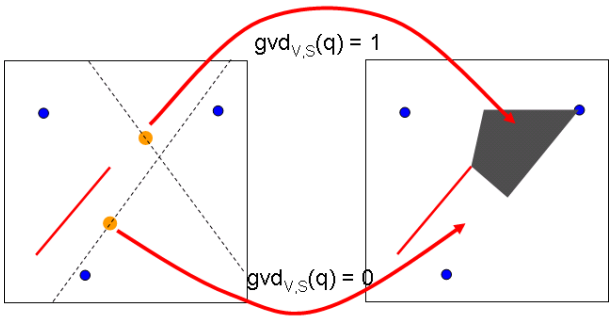
A variation of the illumination or visibility problem is the so-called *good-illumination*, described for the first time in the PhD Thesis of S. Canales in 2004 [Can04]. The basic idea is that a point p is *well visible* if all the viewpoints are *well distributed* around p. Therefore it is not well visible if most of the viewpoints visible from p are grouped in the same *side* of p. Abellanas, Canales and coworkers published some other relevant work about good-visibility, providing its exact calculation in some concrete cases [ACH04, ABM07b]. They also published some variations on the good-visibility [ABHM05, ABM07a], for example for taking into account the restriction in the visibility or illumination of the viewpoints. In contrast to Abellanas, Canales and the coworkers research, we focus our work on the computation of a generic and discrete solution for good-visibility using the graphics hardware, while they try to find an exact solution to the problem.

For the computation of the good-visibility map we use two geometric concepts: the visibility and the depth contours. Both concepts will be used to compute 2D and 2.5D good-visibility maps.

4.2 Exact algorithm

Let V be a set of n viewpoints and S a set of m obstacles in the plane. We assume that no point in V is interior to an obstacle in S. The location depth of an arbitrary point q relative to V, denoted by $ld_V(q)$, is the minimum number of points of V lying in any closed halfplane defined by a line through q. The k-th depth region of V, represented by $dr_V(k)$, is the set of all points q with $ld_V(q) = k$.

The free space F_S relative to S is the complement of S. Given two points $q \in F_S$ and $v \in V$, we say that a viewpoint v is visible from q if the interior of the segment with endpoints v and q remains completely inside F_S. A point q is t-well-visible in relation to V and S if and only if every closed halfplane defined by a line through q contains at least t viewpoints of V visible from q (see Figure 4.1). The good-visibility depth of q relative to V and S, denoted by $gvd_{V,S}(q)$, is the maximum t such that q is t-well-visible in relation to V and S.



(a) (b)

Figure 4.1: *(a) To obtain $gvd_{V,S}(q)$, we consider the set of lines through q and we choose the line that leaves less visible viewpoints on any side. (b) By doing this for every point we obtain the good-visibility map of V and S.*

Let H_q be the set of halfplanes whose boundary line contains q and denote $NV(q,h)$ the number of viewpoints of V visible from q contained in any halfplane h. Since by definition:

$$gvd_{V,S}(q) \;=\; \min_{h \in H_q} NV(q,h)\,,$$

it directly follows Lemma 1, which expresses the relationship between the good-visibility

depth and the location depth of a point.

Lemma 1 *If V_q denotes the subset of points of V visible from q, then $gvd_{V,S}(q) = ld_{V_q}(q)$.*

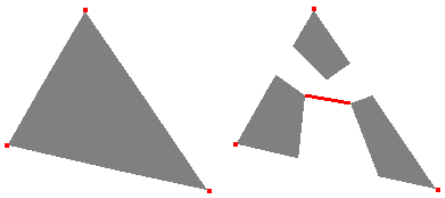
The k-th good-visibility region relative to V and S, denoted by $gvr_{V,S}(k)$, is the set of all points q with $gvd_{V,S}(q) = k$. Observe that $gvr_{V,S}(k)$ can be non connected (see Figure 4.2).

Lemma 2 *If S is empty or is external to the convex hull of V, $CH(V)$, then $gvr_{V,S}(k) = dr_V(k)$.*

Proof. We will highlight two cases depending on the position of a point q relative to $CH(V)$. If $q \notin CH(V)$, a halfplane $h \in H_q$ exists that does not contain any viewpoint of V and then $gvd_{V,S}(q) = ld_V(q) = 0$. If $q \in CH(V)$, it holds $V_q = V$ because S is external to $CH(V)$ and, by using Lemma 1, we have $gvd_{V,S}(q) = ld_{V_q}(q) = ld_V(q)$. Thus, for all q we have $gvd_{V,S}(q) = ld_V(q)$ and consequently $gvr_{V,S}(k) = dr_V(k)$.

□

We call the set of all good-visibility regions relative to V and S the good-visibility map of V and S and denote it with $gvm(V, S)$. Also we denote with $gvm_r(V, S)$ the restriction of $gvm(V, S)$ to region r.



(a) (b)

Figure 4.2: *(a) Good-visibility map of a set of three points. (b) When an obstacle is added the* 1 *good-visibility region becomes smaller and disconnected in some polygons.*

From now on we will focus on the non trivial case in which we have $n \geq 3$ viewpoints and $m \geq 1$ segment obstacles intersecting $CH(V)$.

Lemma 1 induces a way to compute $gvm(V, S)$. First we decompose the free space F_S into visibility regions so that all points in a single connected region are visible exactly

from the same points in V. Then, in each visibility region we compute the depth map of its visible points.

Given a point $v \in V$ and a segment $s \in S$, the shadow region of s with respect to v, denoted by $sr(v, s)$, is the set of points that are invisible to v when we consider the segment s as an obstacle. Denoted by s_0, s_1 the endpoints of s. If $v \notin s$, $sr(v, s)$ is the region delimited by the segment s, the ray of origin s_0 and direction $\overrightarrow{vs_0}$ and the ray of origin s_1 and direction $\overrightarrow{vs_1}$. When v is an endpoint of s, for example s_0, the shadow region $sr(v, s)$ is the ray of origin v and direction $\overrightarrow{vs_1}$. From the collection of all the shadow regions $sr(v, s)$, $s \in S$, we can determine the visibility map $M_v(S)$ of v, the subdivision of the plane into visible and invisible maximal connected components with respect to v and S.

Let $\mathcal{O}$ be the overlay determined by the family of all the visibility maps $M_v(S)$ for $v \in V$, or equivalently of all the shadow regions $sr(v, s)$ for $v \in V$ and $s \in S$, interior to $CH(V)$. All cells in $\mathcal{O}$ are convex and all points in a cell c of $\mathcal{O}$ are seen from exactly the same subset V_c of points of V. Observe that two cells $c \neq c'$ that are seen from the same subset of points of V may exist, it is to say with $V_c = V_{c'}$.

Theorem 3 *The overlay $\mathcal{O}$ consist of $O(n^2m^2)$ cells and each cell has $O(n)$ visible points.*

Proof. Since each shadow region is bounded at most by two rays and one segment, and the convex hull $CH(V)$ has $O(n)$ edges, there are $O(nm)$ elements (segments and rays) defining $\mathcal{O}$. Consequently, $\mathcal{O}$ has $O((nm)^2)$ cells. Figure 4.2 shows that this upper bound is tight. In a) we can see a segment placed in the diameter of a circle and $n/2$ viewpoints v_i placed on the circle and above the segment. The point v_i is placed in a way that one ray of its shadow region intersects $i-1$ rays of all other shadow regions inside the circle and the free space. Then, the number of cells of the line overlay is $\Omega(\sum_{i=1}^{n/2}(i-1)) = \Omega(n^2)$. In b) the segment is split in m segments. Since we have the same properties of a) for each one of the m segments, the new line overlay has $\Omega((nm)^2)$ cells. In c) we have placed $n/2$ light points on the circle and under the segments. This placement ensures that there are $\Omega((nm)^2)$ cells interior to $CH(V)$ that see a minimum of $n/2$ viewpoints. Consequently $O(n^2m^2)$ is a tight upper bound of $\mathcal{O}$, and $O(n)$ a tight upper bound of the points visible from each cell.

□

For each cell c of $\mathcal{O}$ with the visible set of viewpoints V_c, Lemma 1 states that $gvm_c(V, S)$ can be computed as $dm_c(V_c)$, the depth map of the set V_c restricted to c.

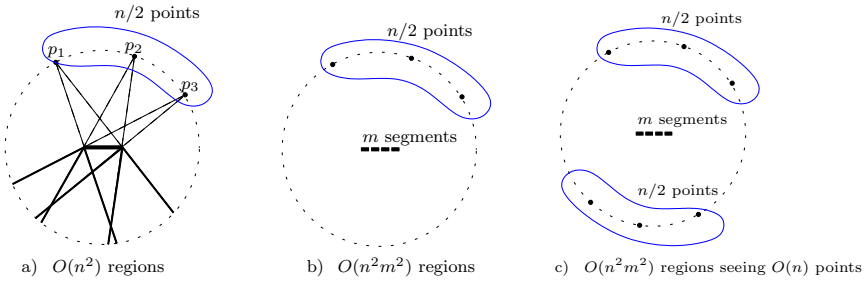


Figure 4.3: *Example showing a worst case $\mathcal{O}$ configuration.*

Then, we have:

$$gvm(V,S)) = \bigcup_{c\in\mathcal{O}} dm_c(V_c)\,.$$

Theorem 4 *The good-visibility map of V and S can be computed in $O(n^4m^2)$ time.*

Proof. Given a set of $\overline{n}$ segments, there is an optimal algorithm (see [Bal95]) that finds the $k' \in O(\overline{n}^2)$ intersections between the $\overline{n}$ segments in $O(\overline{n}\log\overline{n}+k')$ time and $O(\overline{n})$ space ($O(\overline{n}+k')$ space if the intersections are stored). Consequently the cells of the overlay $\mathcal{O}$ can be computed in $O(n^2m^2)$ time and space. By traversing $\mathcal{O}$, the collection of sets V_c associated to the cells c of $\mathcal{O}$ can be computed in $O(n^3m^2)$ time. Next, for each cell c we compute $dm_c(V_c)$ by intersecting the convex cell c with the depth contours determined by $dm(V_c)$. This spends $O(n^2)$ time per cell (see section 2.2.3). Thus, the time needed to compute $gvm(V,S)$ is $O((nm)^2n^2) = O(n^4m^2)$. □

Observe that any good-visibility region $gvr_{V,S}(k)$ can be composed by one or more convex or non-convex polygons.

4.3 Our algorithm for computing depth maps

As can be seen in Section 2.2.3, all known algorithms to compute depth maps using the GPU have to create and make use of dual planes in order to compute the depth of the points in the plane. Due to the way the latter dual planes are dealt with, some discretization errors are introduced which could be avoided if the process runs without looking for those points situated inside the dual planes using image-based algorithms. That is why we have designed and implemented a new algorithm to compute Depth Maps using the GPU which avoids the use of dual planes. This algorithm is implemented using the Cg language

together with the OpenGL primitives like all the other explained algorithms included in this chapter.

The algorithm is based on the more recent Krishnan et al. contribution, commented on at the end of Section 2.2.3. The main difference relies on the computation of the level of each line ℓ. Instead of using the dual planes, we use a method that computes in the primal plane the side of ℓ where each point of V is placed. We use a pixel shader that receives a texture PT containing the viewpoints coordinates, and a texture LT containing the indices of the two points for every ℓ. Then we send $n(n-1) \times n$ pixels through this pixel shader (number of lines multiplied by number of points that we have to test for each one). Finally, inside the pixel shader, for each pixel we use its position to determine the line ℓ which we have to access in LT and the point p of PT we have to test. The position of LT contains the two points p_1, p_2 of PT that determines ℓ. Now we only have to test, using a simple computation, if p is at the left or at the right of $\overline{p_2 - p_1}$. When we know the level of each line ℓ, the algorithm proceeds in the same way.

The running time of this algorithm is approximately the same as the Krishnan one, but it has two important advantages. The levels of the lines are computed in a parallel way. Moreover, it is a more accurate method because it does not use a pixel based algorithm, but an exact algorithm taking into account the real coordinates of the points. The second advantage is that it avoids the use of dual planes which simplifies the whole process a lot and minimizes the errors due to the dual discretization.

4.4 Visualizing good-visibility maps

Lemma 1 induces a relatively easy way to compute the good-visibility map. The method, based on that idea, proceeds in two steps.

First step. We start drawing $CH(P)$ on a black screen and we store the result in a texture. Next we rasterize in white the boundary, interior to $CH(P)$, of all shadow regions $sr(p, s), p \in P, s \in S$ and we transfer the *frame buffer* to an array in the CPU so that each element represents a pixel. Then we find all the cells of $\mathcal{A}(P, S)$ using a CPU based growing method as follows. We take any black pixel of the array and we choose an unused color. Then we visit its four surrounding pixels and we paint each pixel with the current color. If the visited pixel is white (belonging to the boundary) we store it in a *waiting list* and we continue visiting and painting pixels until we have visited an entire cell. While there are pixels in the waiting list we take the first waiting pixel and we repeat the process from this position. In this way we

paint each cell with a different color. During the process we store an interior pixel of each cell and its color. Finally, for each cell c we determine the set P_c of points of P illuminating c. To this end, we take the interior pixel of c and we draw in white, on a black screen, the shadow regions defined by the pixel and the m segments of S. By doing this, a point p illuminates the cell c if its corresponding pixel is black. We use the *readPixels* function to obtain the set P_c by checking if the corresponding pixel of each of the n points of P is colored in black. Moreover, we assign a distinct color to each different subset P_c so that all cells illuminated by P_c will have the same color. In this way we ensure that we paint the same depth map at most once in the second step.

Second step. For each cell $c \in \mathcal{A}(P, S)$ we draw $dm_c(P_c)$ using the algorithm described in Section 2.1 that draws depth contours. In order to paint only the pixels inside c we use a fragment shader. Its input parameters are a texture containing the arrangement $\mathcal{A}(P, S)$ and the color assigned to P_c. The fragment shader only paints a pixel (x, y) if the color in the position (x, y) of the texture representing $\mathcal{A}(P, S)$ is equal to the color of c, since in this case the pixel is inside cell c.

This algorithm was published in the proceedings of the 23th European Workshop on Computational Geometry [CMS08a].

4.5 A better solution

Theorem 4 highlights the fact that computing good-visibility maps in that way is expensive even in real situations, in which the number n of viewpoints is low but the number m of obstacles is high. This motivates us to explore an alternative GPU-based algorithm for visualizing them faster.

The main drawback of the previously described algorithm, based on the fact that $gvm(V, S)) = \bigcup_{c \in \mathcal{O}} dm_c(V_c)$, is the huge quantity of depth maps, one for each of the $O(n^2m^2)$ cells of $\mathcal{O}$, that need to be computed. In order to overcome this drawback we proposed a new more efficient algorithm. The first step of the algorithm consists of obtaining a representation of the overlay $\mathcal{O}$ of all the visibility maps $M_v(S)$, $v \in V$. We use a texture, denoted by MVM, to represent this overlay. The second step efficiently approximates $gvm(V, S)$. In the following we consider that the screen where $gvm(V, S)$ has to be visualized has a size of $W \times W$ pixels, $n < W$, and the screen coordinates of the viewpoints are stored in a texture $VPos$ of $1 \times n$ pixels.

4.5.1 Obtaining the texture MVM

In the first part of the algorithm we have to compute the multi-visibility map representing the overlay $\mathcal{O}$ of the shadow regions. Chapter 3 contains all the algorithms created for computing this overlay, stored in the texture MVM, for this case and all the variations of the visibility proposed in next sections. Section 3.3.1 contains the explanation of the two algorithms for computing the visibility map for this simple case depending on if we consider segment or generic obstacles.

4.5.2 Approximating $gvm(V, S)$

Once the texture MVM containing the visibility information is computed, we can focus on the computation of the good-visibility map. We have designed an efficient algorithm for computing the good-visibility map that needs to calculate a depth map only once during the whole process (see Algorithm 2).

First, for each oriented line ℓ_{ij} passing through two different viewpoints v_i and v_j the side of ℓ_{ij} where each viewpoint is placed is computed using a pixel shader. The shader renders a quad of $n \times n$ pixels to the texture $VP\ell$ where the pixel in position (i, j) represents the oriented line ℓ_{ij}. The k bit of the RGBA channel of $VP\ell[i, j]$ stores 1 if viewpoint v_k is placed at the left of the oriented line passing through the viewpoints v_i and v_j. This pixel shader needs the texture $VPos$ as parameter.

Second, for each oriented line ℓ_{ij} the half-plane ℓ_{ij}^+ is rendered on-screen using another pixel shader (Algorithm 3). The shader that renders ℓ_{ij}^+ receives MVM and $VP\ell$ textures, and n as global parameters, and computes for a pixel p the number nvp of bits equal to 0 in $MVM[p]$, i.e. the number of viewpoints visible from p, and the number $nvpl^+$ of bits equal to 0 in $MVM[p]$ and 1 in $VP\ell[i, j]$ (the number of viewpoints contained in ℓ_{ij}^+ visible from p). The depth of p returned by the pixel shader is $\min\{nvpl^+, nvp - nvpl^+\}$ and the grey-scale color of p is chosen proportionally to its depth.

After doing this process for every oriented line and having the depth test activated with the *LESS* function we obtain the good-visibility map $gvm(V, S)$.

Since the cost per pixel is $O(n)$ time, we get a cost per line of $O(nP_{n^2} + nP_{W^2})$ time, where P_{n^2} is the time for rendering n^2 pixels, and the cost of approximating $gvm(V, S)$ is $O(n^3 P_{W^2})$ time. Consequently, the overall complexity of visualizing $gvm(V, S)$ is $O(nm + n^3 P_{W^2})$ or $O(nA_W P_{W^2} + n^3 P_{W^2})$ time depending on the kind of the obstacles (A_W represents the time needed for accessing W pixels of a texture).

Algorithm 2: 2D good-visibility map

Input: Set of viewpoints V, Texture MVM.

Output: $gvm(V, S)$.

Activate Depth Test using LESS function;

for $i = 0$ *to* n **do**

  for $j = i + 1$ *to* n **do**

    $\ell \leftarrow \overline{v_i v_j}$;

    $h_V \leftarrow pointsInHalfplane(\ell^+, V)$;

    $h_n \leftarrow numberOfBitsActivated(h_V)$;

    activate $pixelShaderGVM(MVM, h_V, h_n)$;

    $paint(\ell^+)$;

    $h_V \leftarrow !h_V$; `// Bit to bit negation`

    $h_n \leftarrow numberOfBitsActivated(h_V)$;

    activate $pixelShaderGVM(MVM, h_V, h_n)$;

    $paint(\ell^-)$;

  end

end

Algorithm 3: pixelShaderGVM

Input: Pixel position x, y, Texture MVM, points in halfplane h_V, number of points in halfplane h_n.

Output: *color* and *depth* of current pixel.

$visibility\_value \leftarrow MVM[x, y]$;

$coincidences \leftarrow 0$;

for $i = 0$ *to* $numberOfBits(h_V)$ **do**

  if $visibility\_value.bit(i) == 0$ *AND* $h_V.bit(i) == 1$ **then**

    $coincidences++$;

  end

end

$level \leftarrow min(coincidences, h_n)$;

$color, depth \leftarrow level$;

$return(color, depth)$;

Comparing the $O(n^4m^2)$ complexity of the exact algorithm with the complexity of the GPU-based algorithm, one can observe that the running time is drastically reduced. Figure 4.4 shows the comparison between the running times of both algorithms, the first method proposed and the improved algorithm. In the top image the number of segments is fixed to 3 and the number of viewpoints is increased. In contrast, in the bottom picture the number of viewpoints is fixed to 24 and we increase the number of segments. We can observe that the running times of our new algorithm are always much smaller, which is a direct consequence of the reduction of the theoretical complexity.

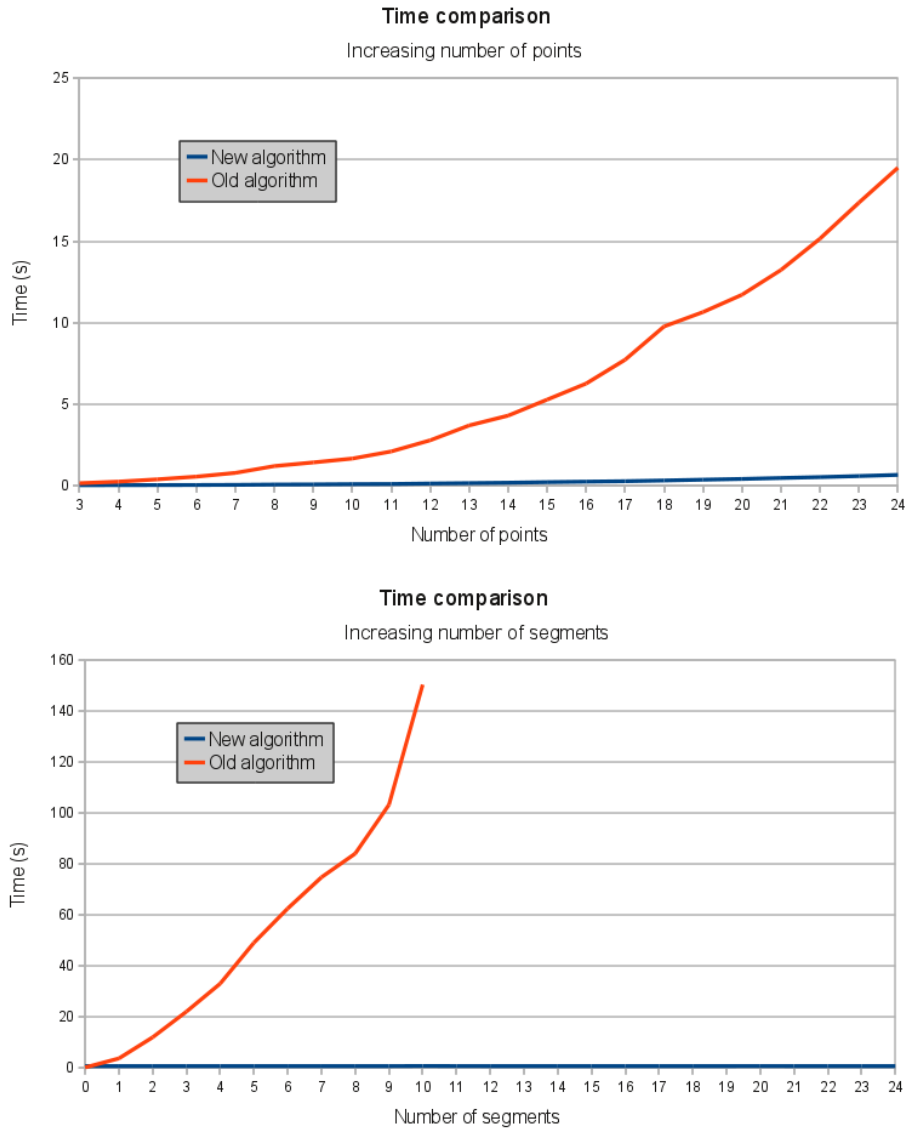


Figure 4.4: *Running time comparison between the two proposed algorithms to compute gvm. In the top image the number of segments is fixed to 3 and in the bottom one the number of viewpoints is fixed to 24.*

Figure 4.5 shows the differences between the multi-visibility map computed in the first step of the algorithm and the good-visibility map. The huge differences between these images show that good-visibility maps provide a totally new way to understand the visibility concept and how they can be useful for taking into account the relative position between the viewpoints and the rest of the plane.

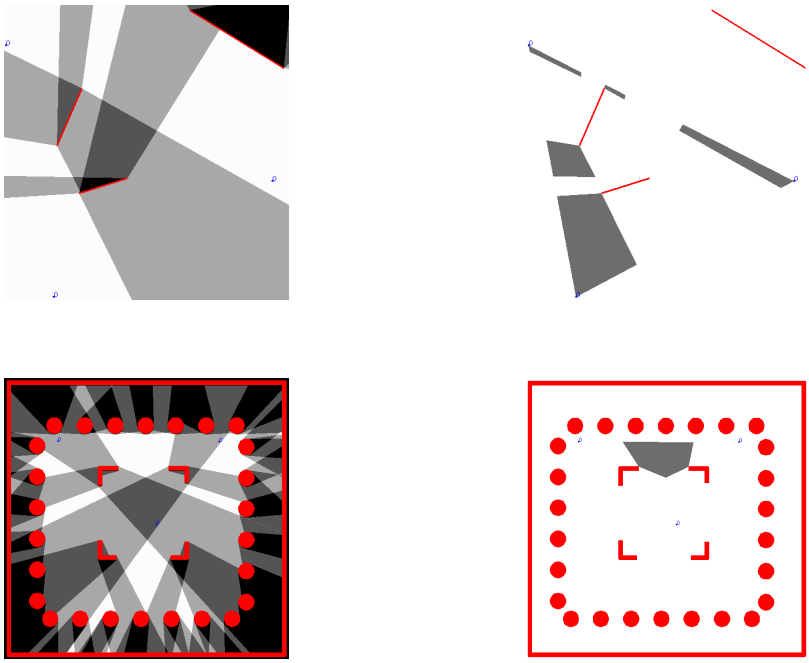


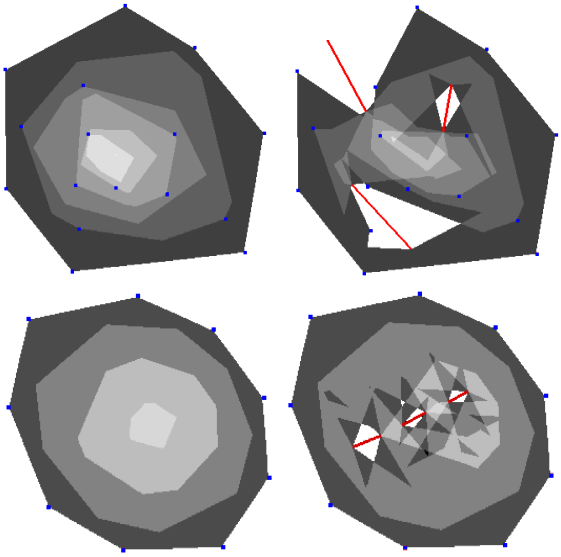
Figure 4.5: *The left images show two examples of visibility map and the right ones have the good-visibility map computed from the same scenes. In the visibility map, black means invisible to all viewpoints while in the good-visibility maps, white represents this.*

4.6 Results

We have implemented the proposed methods using C++ and OpenGL for the main application and Cg language for the pixel shaders executed inside the GPU. All tests and images have been carried out on a laptop equipped with an Intel Core 2 Duo P8400 at 2.26GHz, 4GB of RAM and a GeForce 9600M GT graphics card supporting texels of 128 RGBA bits and using a screen resolution of 500x500 pixels for the GPU-based computations (in Figure 4.10 we also used other screen resolutions to compute the times).

4.6.1 Good-visibility maps in the plane

Figure 4.6(a) shows two depth maps corresponding to two different sets of viewpoints, while Figure 4.6(b) shows the good-visibility maps of the same sets of viewpoints mixed with segment obstacles. All maps have been obtained with our implementation, and their good-visibility regions are colored in a grey gradation according to their depth (black corresponds to level one) except the level zero region which is colored in pure white.



(a) (b)

Figure 4.6: *(a) Depth maps of the points. (b) Good-visibility maps of the points and the segments.*

Figure 4.7 shows a detailed example of a good-visibility map (last image) when generic obstacles represented by a binary image are considered. The first five images correspond to the regions of level greater or equal to k, $k = 1 \ldots 5$.

In order to corroborate the predicted computational cost of our algorithm, we have tested it with several configurations of viewpoints placed randomly in a square box mixed with segment obstacles or generic obstacles represented by a binary image.

Figure 4.8 shows the cubic relation between the running time and the increment of viewpoints. Observe that when the number of viewpoints and segments obstacles is high enough, for a fixed number of viewpoints the running time corresponding to segment obstacles is greater than the running time corresponding to generic obstacles. This is because the computational cost for generic obstacles only depends on the number of viewpoints and the screen size.

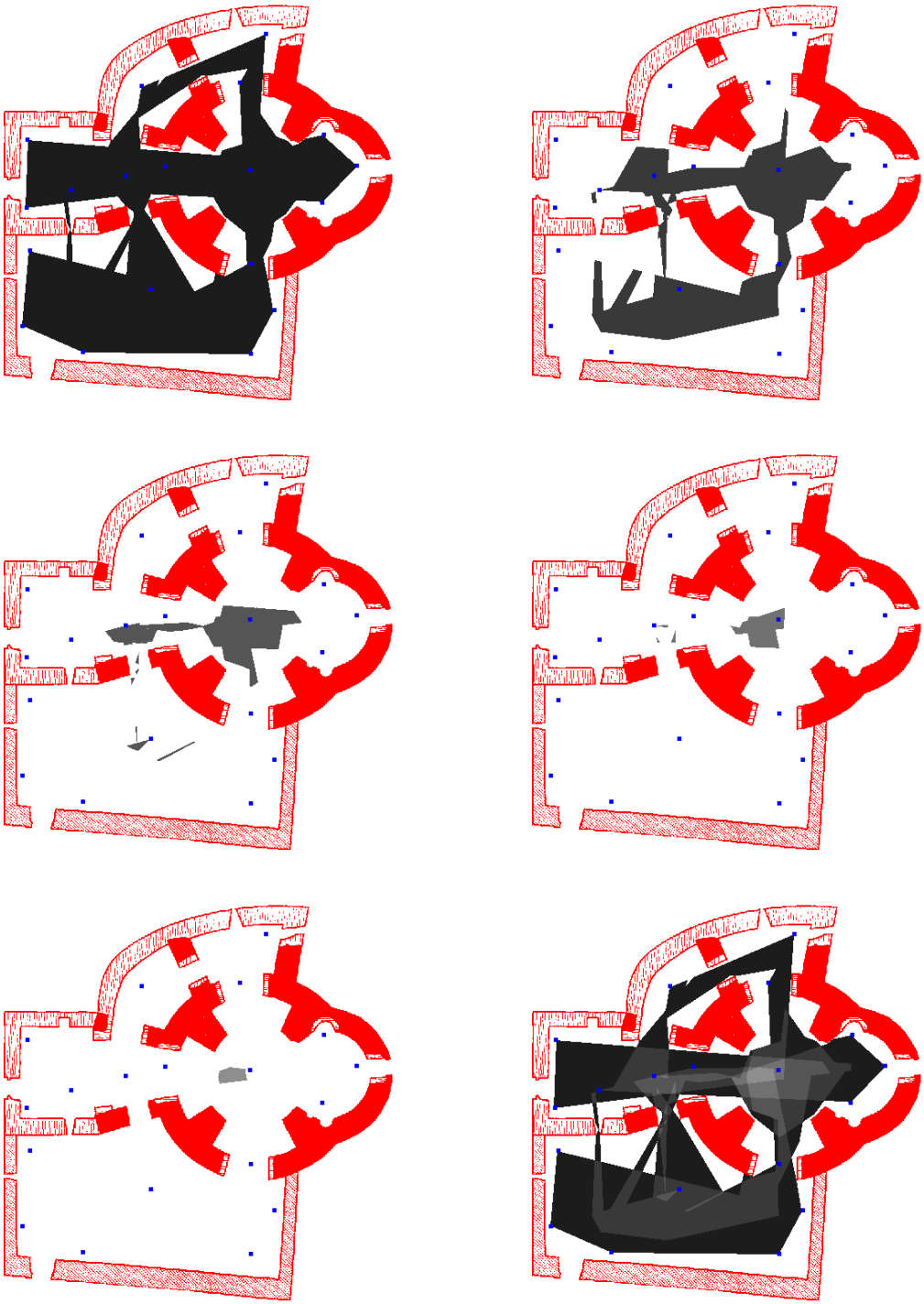


Figure 4.7: *The walls of a church are represented by a binary image. The first five images show the five good-visibility regions of the good-visibility map. The last image shows the good-visibility map.*

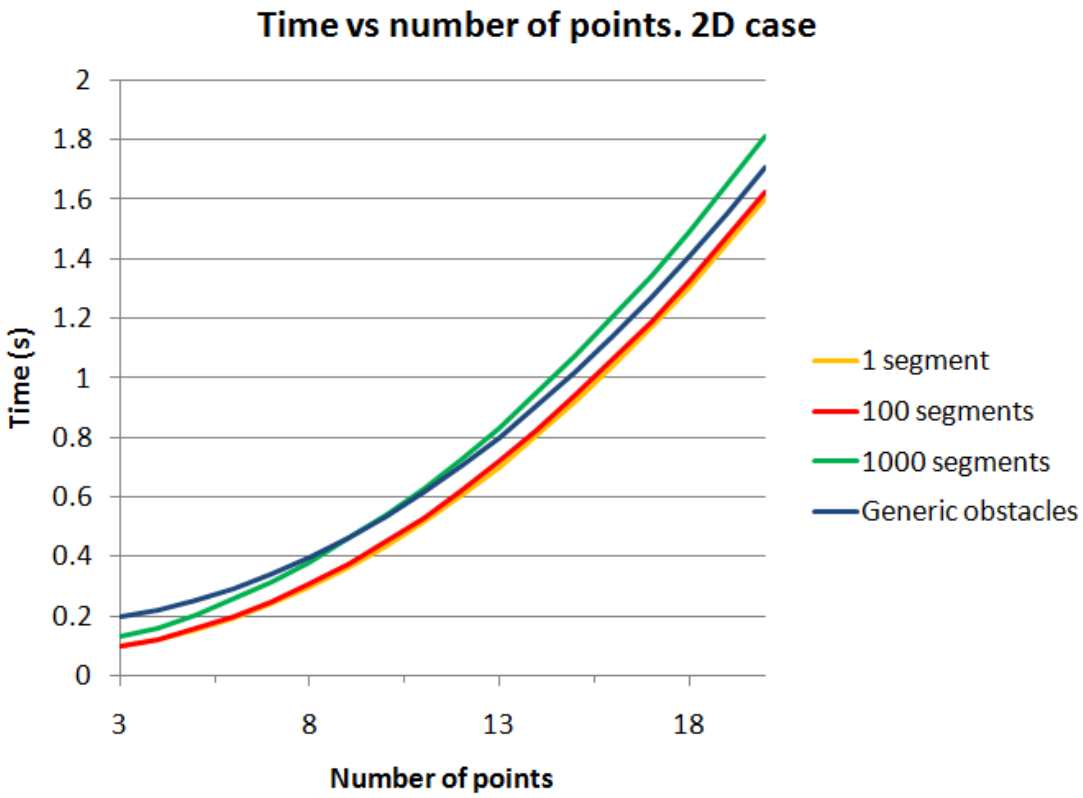


Figure 4.8: *Relation between the running time and the number of viewpoints for different number of segments and generic obstacles.*

Figure 4.9 indicates the linear relation between the running time and the number of segment obstacles.

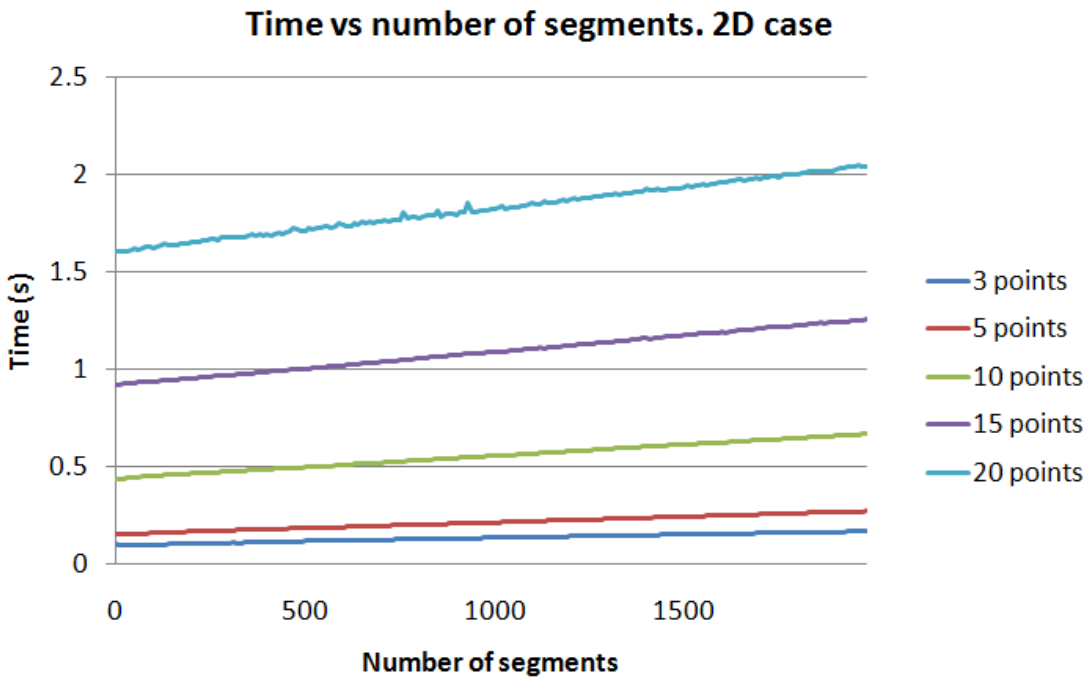


Figure 4.9: *Running time depending on the number of viewpoints and the number of segment obstacles.*

Figure 4.10 shows how the running time increases depending on the number of viewpoints and the screen size used to render the good-visibility map for a fixed number of segment obstacles (10) or generic obstacles.

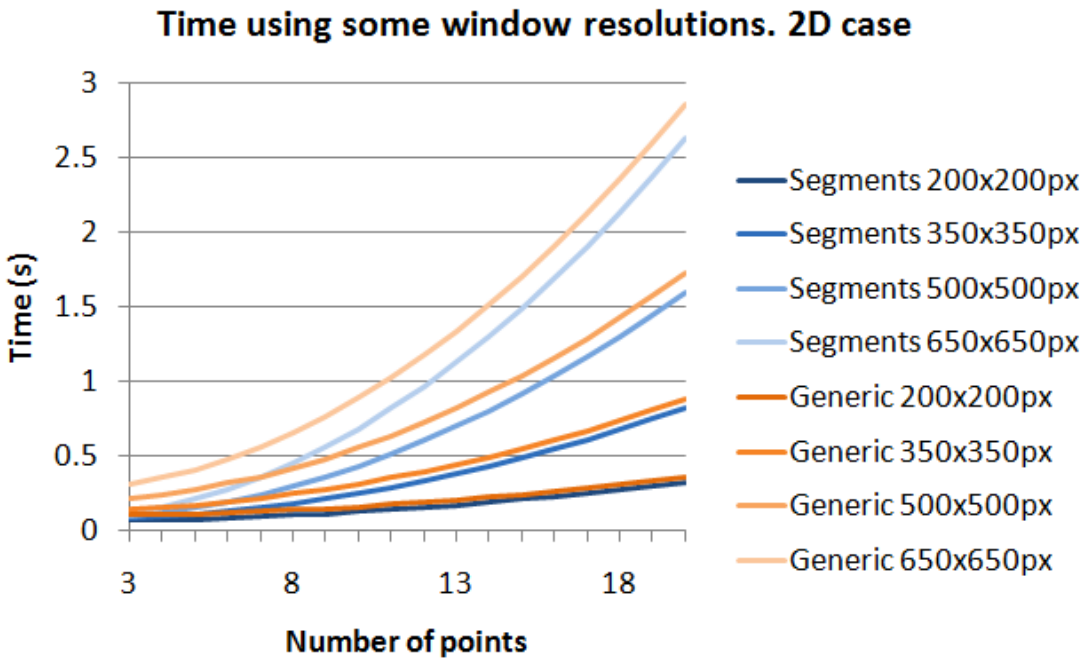


Figure 4.10: *Running time depending on number of viewpoints and screen size.*

4.7 Variations

A lot of variations to the problem can be included to solve different types of problems or to model more realistic scenes with a variety of real situations. Some of these variations are about the properties of the visibility of viewpoints (restricted visibility in range or angle or *power of emission*) or even the shape of the *view objects* (we can consider view segments instead of viewpoints) as in Figure 4.11.

All the implemented variations consist of changing the computation of the MVM texture that stores the multi-visibility map. Since all the visibility algorithms implemented are explained in Chapter 3, here we only show images of the results and the comparison between the visibility map and the good-visibility map for the same scene.

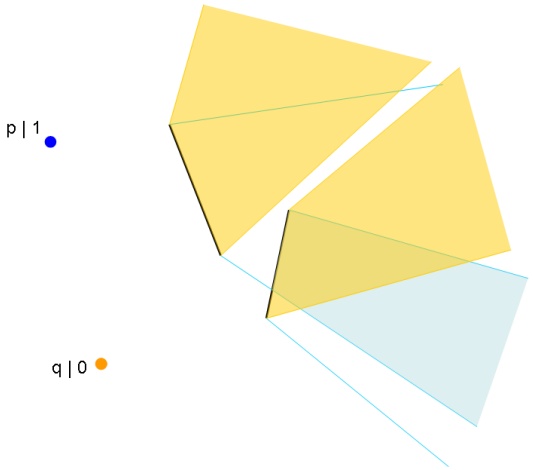


Figure 4.11: *Shadow regions with signal crossing segment obstacles. Point p has sufficient power to cross one segment obstacle.*

4.7.1 Viewpoints with Restricted Visibility

As Abellanas et. al state in [ABHM05], a point v has restricted visibility when it is restricted within an angular region or/and with limited range. It is necessary to distinguish two cases due to the different implementation for the calculation of the MVM texture for segment obstacles and generic obstacles (see Section 3.3.2 for a more detailed explanation).

Once the modified texture MVM is obtained, the algorithm can continue in the same way as described in previous sections in order to draw good-visibility maps with restricted visibility points. Figure 4.12 compares the visibility map and the good-visibility map for a simple scene.

In Figure 4.13 we can see two good-visibility maps computed from a set V containing some viewpoints with restricted visibility and a set S of segment obstacles. Moreover, Figure 4.14 shows an example of a good-visibility map using viewpoints with restricted visibility and a binary image representing generic obstacles.

4.7.2 Viewpoints with *power of emission*

The viewpoints are now treated as points emitting another kind of signal. In this extension we have added the *emission power* to the points and it is represented by the number of segment obstacles or pixels that the signal emitted by a viewpoint can cross.

As in the case of restricted visibility, we can implement the emission power by changing the computation of the texture MVM. Figure 4.15 shows the comparison between visibil-

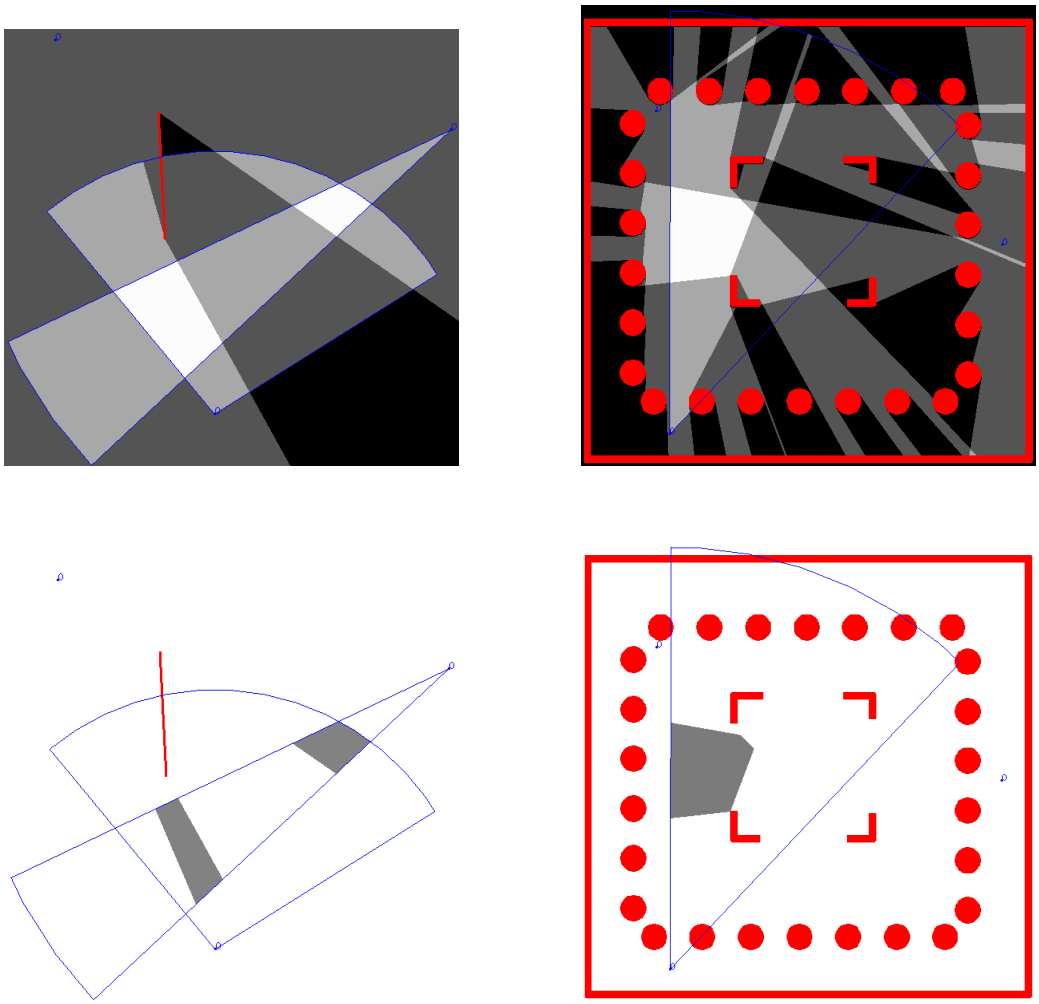


Figure 4.12: *Visibility (upper images) and good-visibility (lower images) for a simple scene using segment and generic obstacles and restricted visibility.*

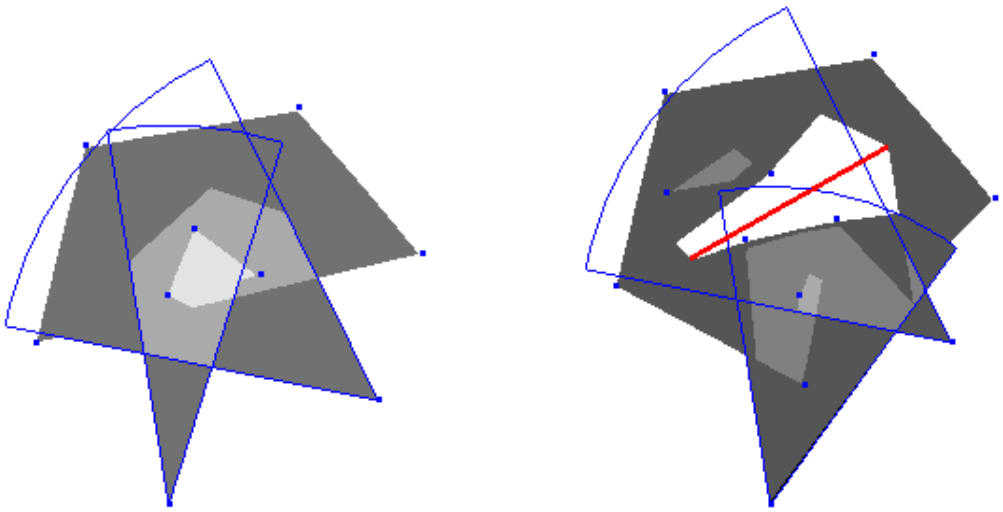


Figure 4.13: *Good-visibility maps for restricted viewpoints.*

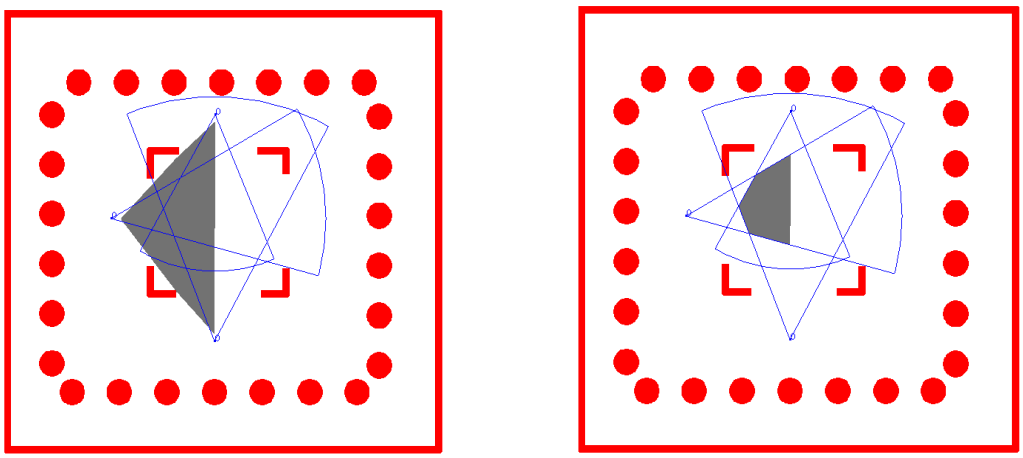


Figure 4.14: *The red pixels represent the obstacles and blue regions indicate the restriction in the visibility of the viewpoints. The left image does not take into account the visibility restriction while the good-visibility map on the right one is computed with it.*

ity and good-visibility in the three cases implemented, when the obstacles are segments, when they are represented by a binary image, and when this image uses a color gradiation to codify the *opacity* or resistance to the emitted signal of each pixel belonging to the obstacles.

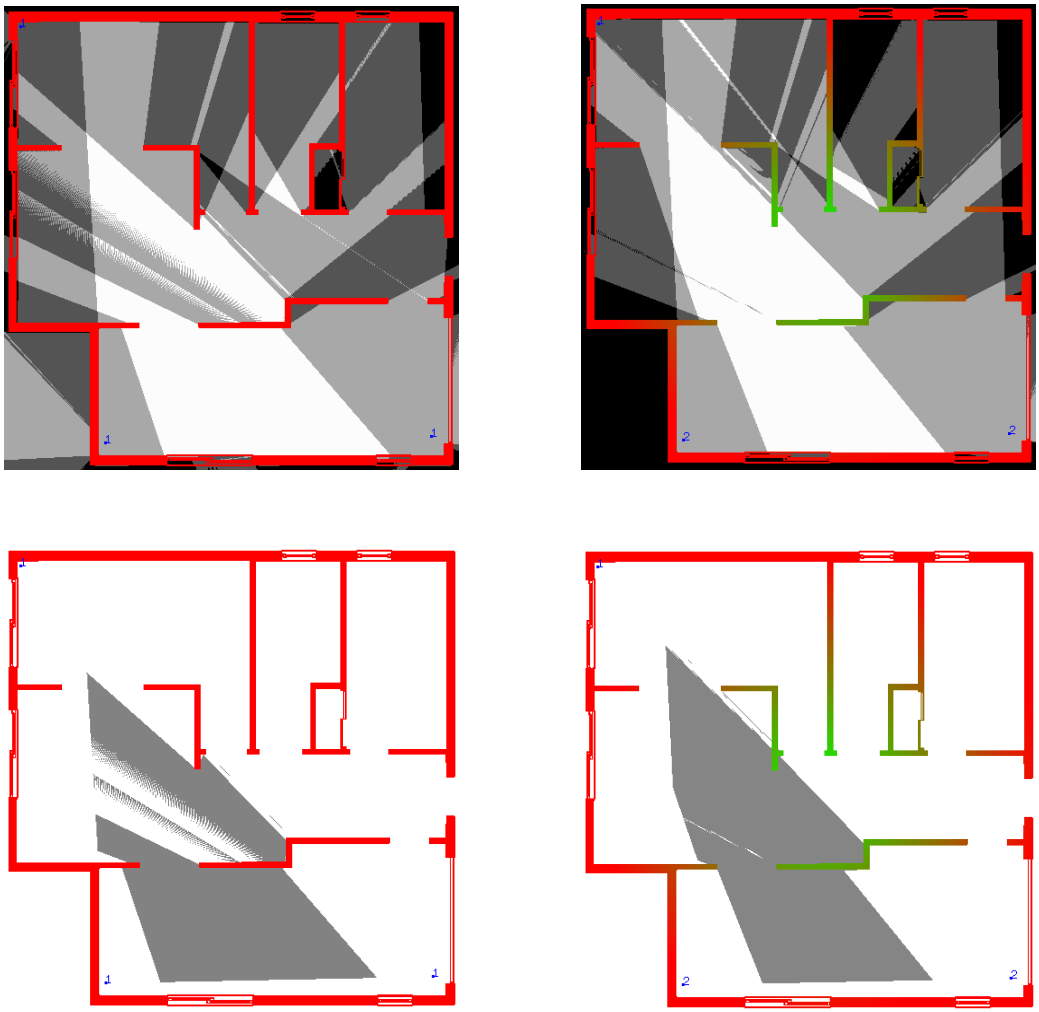


Figure 4.15: *Visibility (upper images) and good-visibility (lower images) for a simple scene using segment and generic obstacles and power of emission.*

In Figure 4.16 there is an example of three viewpoints with variable power of emission in a scene with some segment obstacles.

In Figure 4.17 we can see an example scene with five viewpoints, where the number on each v_i indicates its power of emission P_{v_i}. In these images (especially in the second one) some artifacts on the good-visibility map can be observed due to the power of emission and obstacles discretization.

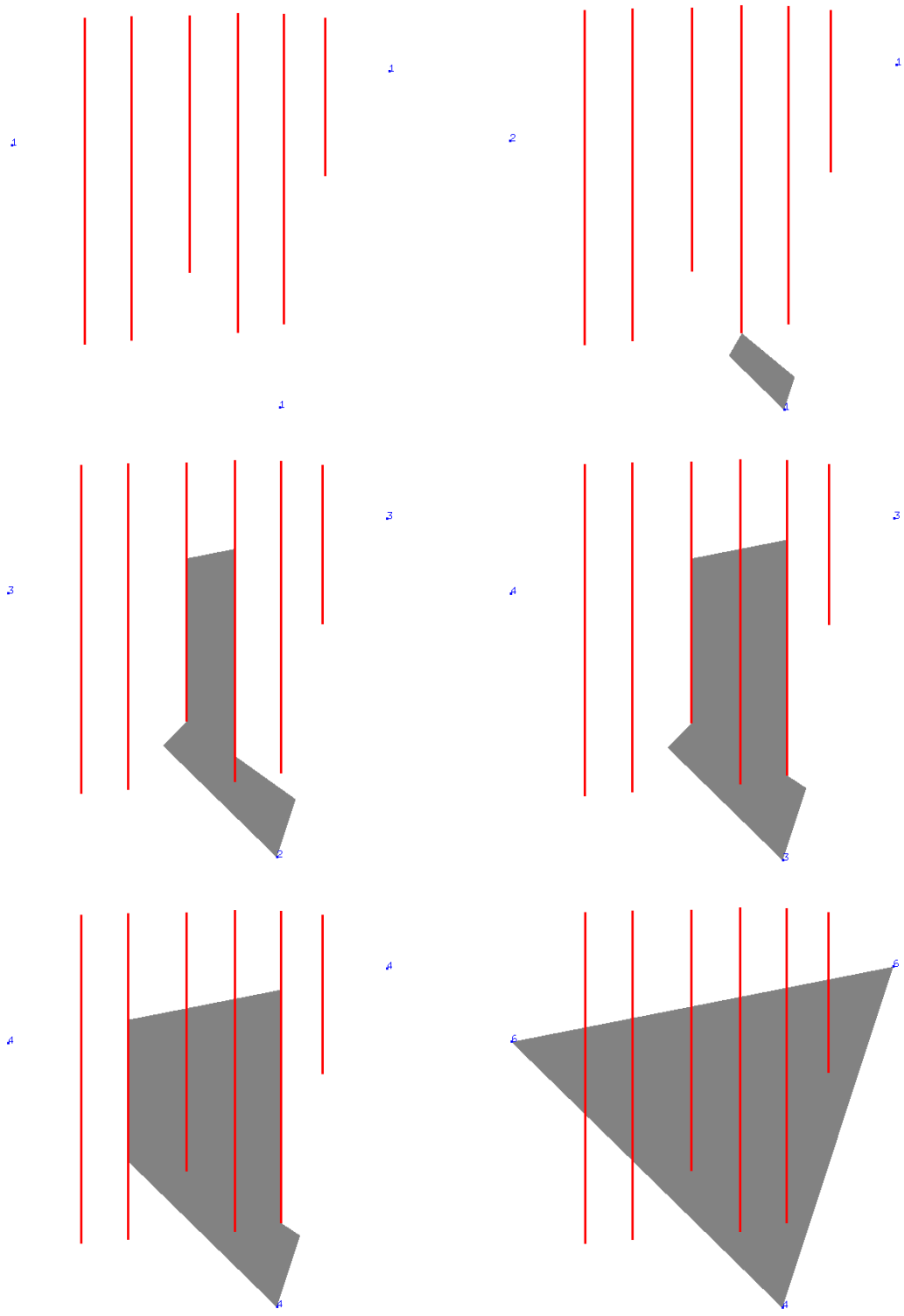


Figure 4.16: *Scene composed by some segment obstacles and three viewpoints where the number indicates their power, which is incremented gradually from top left to bottom right and the good-visibility map grows consequently.*

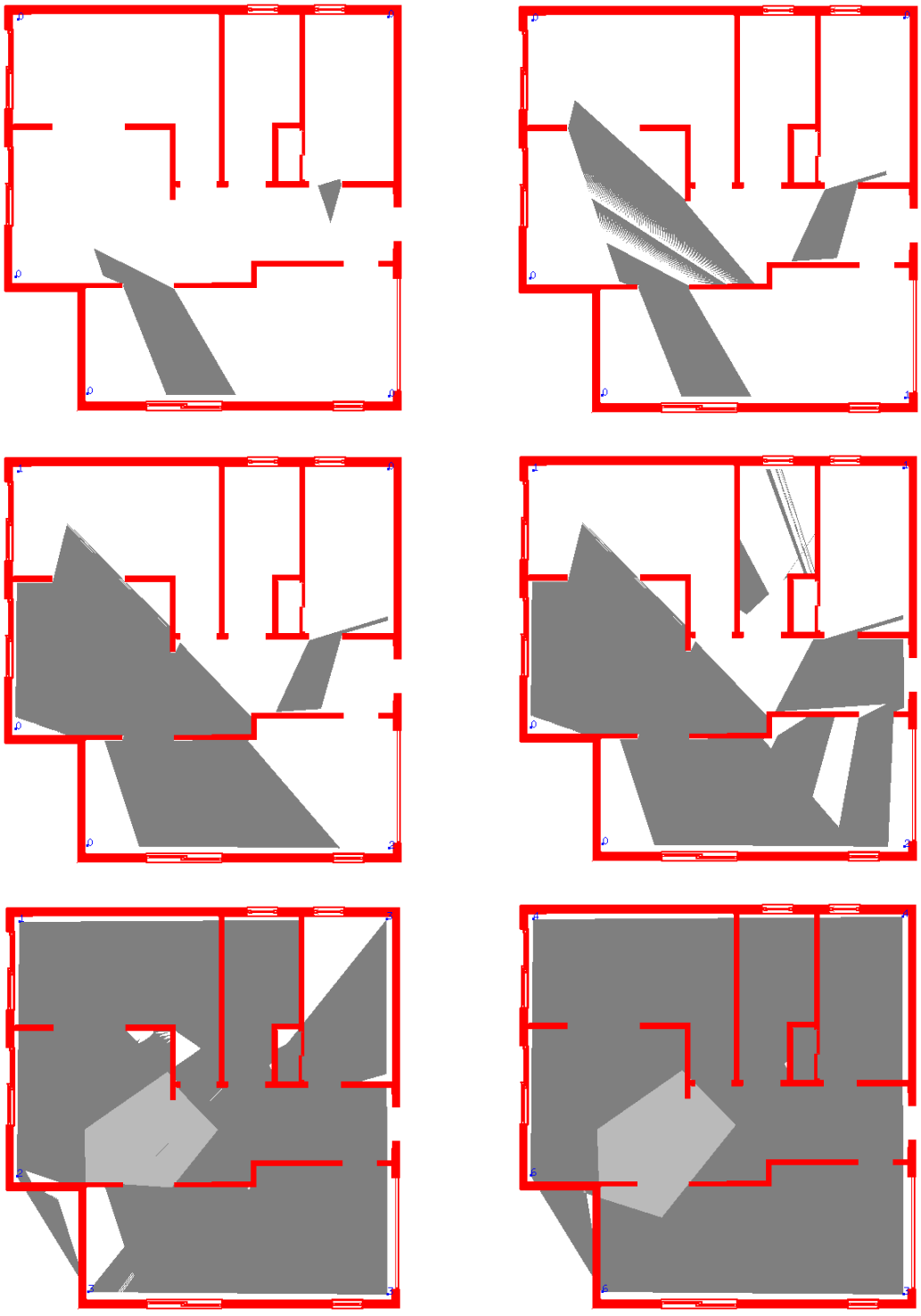


Figure 4.17: *Red pixels represent the obstacles and blue numbers indicate the power of emission of the viewpoints. From top left to bottom right, the power of emission of the distinct viewpoints is incremented gradually, thus the good-visibility regions becomes larger in every image of the sequence.*

Figure 4.18 shows some examples of adding *Opacity* in generic obstacles using the color of its pixels. In the upper images there is a portion of the obstacle bar which is totally green. This means that this part of the obstacle has no resistance, thus 100% of the signal crosses as it might do if there were no obstacles present in the scene.

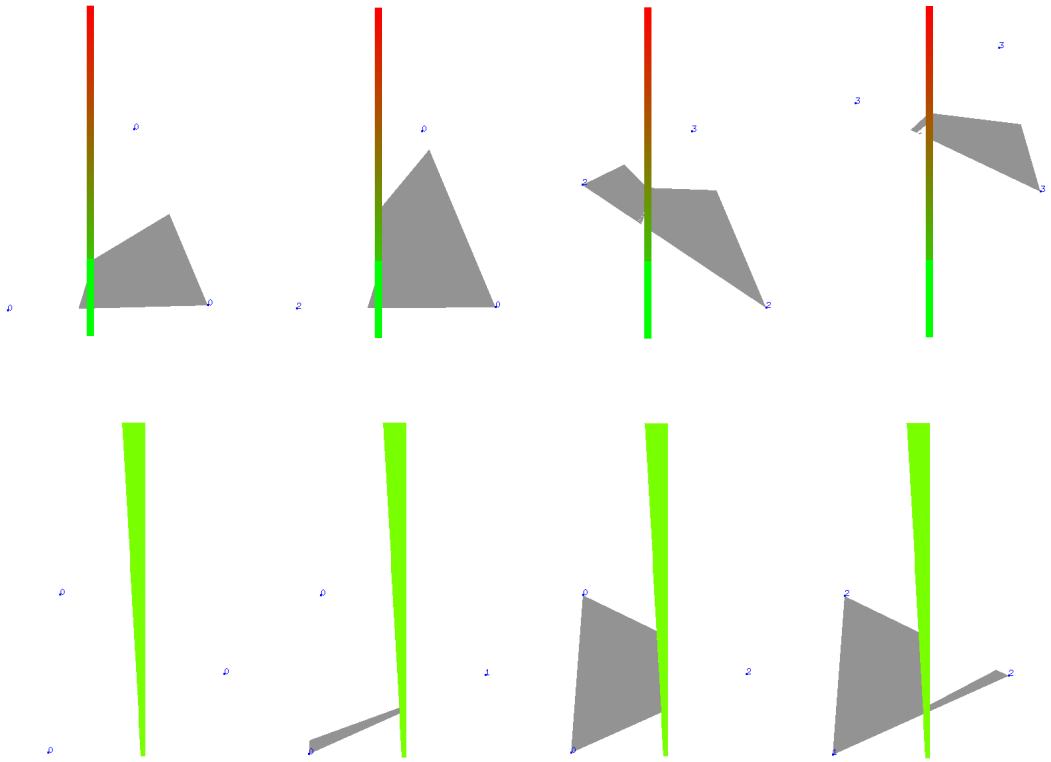


Figure 4.18: *Blue numbers indicate the power of emission of viewpoints. Upper images show how the change in the* opacity *(color of the obstacle) modify the good-visibility. In the lower images we can see how the good-visibility changes with variable obstacle thickness.*

4.7.3 View segments instead of viewpoints

Now we want to have a set of view segments instead of viewpoints or even the possibility of having a set of view objects, including both points and segments.

The fact that visibility from a segment can have two different interpretations (see Section 3.3.4) means that we also have to change the concept of location depth to take into account two distinct types of good-visibility when dealing with view segments: weak and strong.

The location depth of a point p with respect to a set of viewpoints V and a set of segment obstacles S is defined as the minimum number of viewpoints visible from p that

the halfplane defined by any line passing through p contains.

It seems logical to think that for *strong* good-visibility, if some of the elements of V are view segments, they are counted for this minimum only if the whole segment is included in the halfplane. It can be easily computed taking into account only the two end points of the segment. On the other hand, considering *weak* good-visibility, a segment is counted if any of its interior points belong to the considered halfplane. This is true when one or both of its end points are inside the halfplane (see Figure 4.19 for a conceptual scheme and Figure 4.20 for a practical example).

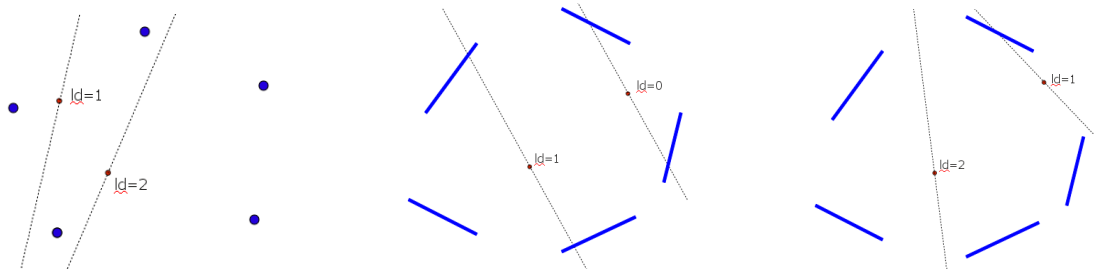


Figure 4.19: *Left image shows how the location depth works when V is a set of viewpoints. In the middle and right images, strong and weak location depth can be seen respectively, when V is a set of view segments.*

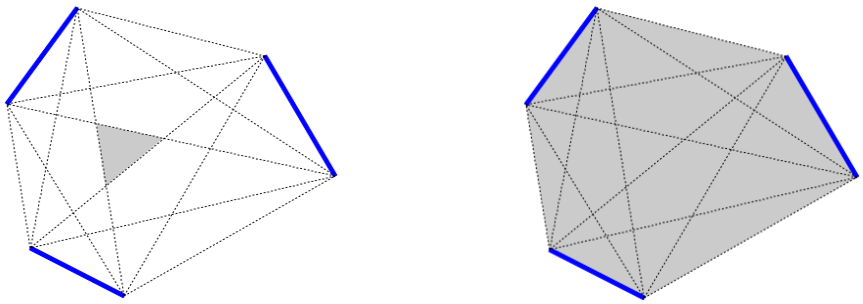


Figure 4.20: *Left and right images show the depth contours from a set of three view segments with strong and weak visibility, respectively.*

Thereafter, the depth map and the good-visibility map implementations using the GPU will be described.

Computing the depth map using the GPU

First of all we give a solution for the problem without obstacles. In order to compute depth contours from a set of view segments with strong or weak visiblity the algorithm

(explained at the beginning of this same Chapter 4) has to be slightly updated. The first part is equal for both strong and weak location depth and it works as follows. The two end points v_{i_s} and v_{i_f} of every view segment v_i are connected to form oriented lines with both end points of all the other view segments, in the same way each viewpoint is connected to each other in the viewpoints version. In this process, a level must be assigned to every one of these oriented lines l.

If we consider the case of strong visibility, the level of ℓ is computed as the number of full view segments at the left of ℓ, without taking into account the end points of the segments generating ℓ. On the other hand, the level of a line ℓ generated from v_{i_s} and v_{j_s} for weak visibility is computed by counting all the fully or partially view segments placed to the left of ℓ. A segment is considered partially inside a halfplane if one of its end points is inside it (we do not take into account the points v_{i_f} and v_{j_f}).

Finally, for every line ℓ, its left halfplane is painted with depth and color equal to the level found before and its right halfplane is painted with depth and color equal to $n - level - 2$, where n is the number of view segments in V. If this process is done with the depth test activated and using the LESS function, as in the case of viewpoints, the depth contours of V are correctly computed.

In Figure 4.21 some examples of the computation of the depth contours from a set of view segments are shown.

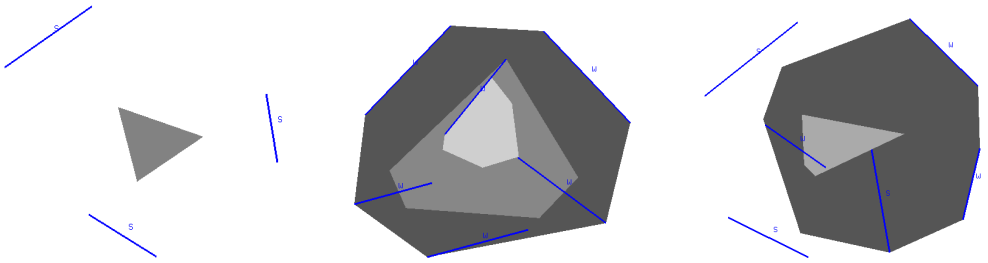


Figure 4.21: *From left to right images show the computation of the depth map from a set of view segments using strong visibility (S), weak visibility (W) and a mix of them, respectively.*

Good-visibility map computation

Once we know how to compute the multi-visibility map from a set of view segments and a set of segment obstacles (explained in Section 3.3.4) and we know how the location depth of a point p is affected by the strong or weak visibility, we are ready to compute

the good-visibility map from view segments or even from a mix of viewpoints and view segments, any of them having strong or weak visibility.

Since the texture MVM containing the visibility information has been already obtained (see Section 3.3.4), now we can compute the good-visibility map. The algorithm is very similar to the one used for computing the good-visibility map from a set of viewpoints, described in Section 4.5.

We need to construct a line ℓ for every pair of endpoints that generates the set of view segments V and count how many view segments are located on the left and on the right of ℓ. The way a segment is counted or not is determined by the kind of visibility it has associated (see previous section). When considering viewpoints, the set V has n points, however in the case of view segments V is composed by $2 \times n$ endpoints. The line ℓ_{ij} connects the endpoints i and j. For every oriented line ℓ_{ij}, the number of view segments situated at its right or at its left is counted using a pixel shader. The shader renders a quad of $2n \times 2n$ pixels to the texture $VP\ell$ where the pixel in position (i, j) represents the oriented line ℓ_{ij}. The k bit of the RGBA channel of $VP\ell[i, j]$ stores 1 if the view segment v_k is placed at the left of ℓ_{ij}. This pixel shader uses an input texture, denoted by $VPos$, which contains the position of every endpoint of the view segments (see Algorithm 4).

Second, for each oriented line ℓ_{ij} the half-plane ℓ_{ij}^{+} is rendered on-screen using another pixel shader (Algorithm 5). The shader that renders ℓ_{ij}^{+} receives MVM, $VP\ell$ and n as global parameters. As in the viewpoints version, explained in Section 4.5.2, it computes for a pixel p the number nvp of bits equal to 1 in $MVM[p]$ and the number $nvpl^{+}$ of bits equal to 1 simultaneously in $MVM[p]$ and $VP\ell[i, j]$. The depth of p returned by the pixel shader is $\min\{nvpl^{+}, nvp - nvpl^{+}\}$ and the gray-scale color of p is chosen proportionally to its depth.

After doing this process for every oriented line with the depth test activated using the *LESS* function, the good-visibility map $gvm(V, S)$ is obtained.

This algorithm can also be used to compute the good-visibility map from a set V of mixed viewpoints and view segments. It is necessary to slightly modify the previously detailed code in order to take into account when a point v_i is a viewpoint or a endpoint of a view segment.

Results and running time Figures 4.22 and 4.23 contain examples of good-visibility maps computed using the latter algorithm. Some images show good-visibility maps from sets V containing only view segments with a fixed type of visibility while others depict a good-visibility map from a set of view segments with mixed visibility or even from a set

Algorithm 4: 2D GVM viewSegments

Input: Set of view segments V, Texture MVM.

Output: $gvm(V, S)$.

Activate Depth Test using LESS function;

for $i = 0$ *to* $2n$ **do**

  for $j = i + 1$ *to* $2n$ **do**

    $\ell \leftarrow \overline{v_i v_j}$;

    $h_V \leftarrow segmentsInHalfplane(\ell^+, V)$;

    $h_n \leftarrow numberOfBitsActivated(h_V)$;

    activate $pixelShaderGVM_viewSegments(MVM, h_V, h_n)$;

    $paint(\ell^+)$;

    $h_V \leftarrow !h_V$; `// Bit to bit negation`

    $h_n \leftarrow numberOfBitsActivated(h_V)$;

    activate $pixelShaderGVM_viewSegments(MVM, h_V, h_n)$;

    $paint(\ell^-)$;

  end

end

Algorithm 5: pixelShaderGVM viewSegments

Input: Pixel position x, y, Texture MVM, segments in halfplane h_V, number of segments in halfplane h_n.

Output: *color* and *depth* of current pixel.

$visibility\_value \leftarrow MVM[x, y]$;

$coincidences \leftarrow 0$;

for $i = 0$ *to* $numberOfBits(h_V)$ **do**

  if $visibility\_value.bit(i) == 1$ *AND* $h_V.bit(i) == 1$ **then**

    $coincidences++$;

  end

end

$level \leftarrow min(coincidences, h_n)$;

$color, depth \leftarrow level$;

$return(color, depth)$;

V containing both viewpoints and view segments.

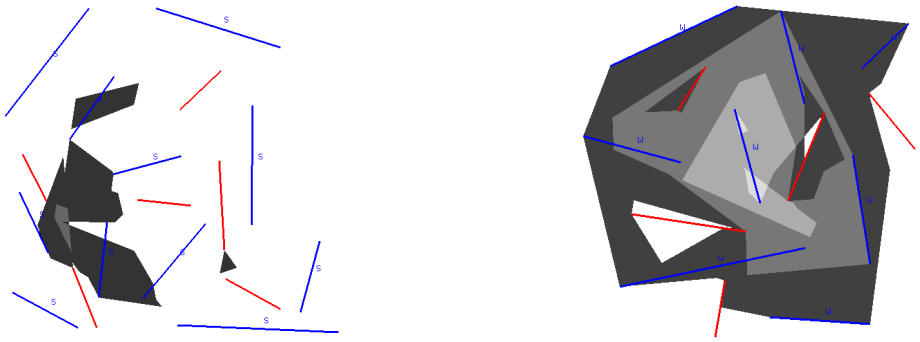


Figure 4.22: *The images show the good-visibility map from a set of 11 view segments with strong visibility and from a set of 8 view segments with weak visibility, respectively.*

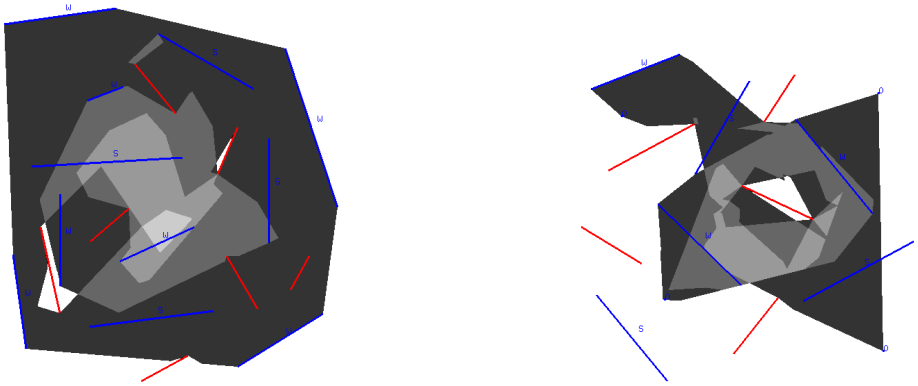


Figure 4.23: *The image on the left show the good-visibility map computed from a set of view segments with mixed visibility while the right image contains the good-visibility map calculated from a set of mixed viewpoints and view segments.*

4.8 Good-visibility on a terrain

An adaptation of the two-dimensional good-visibility computation can be introduced considering that it is computed on a terrain, where its faces play the role of obstacles.

Let $\mathcal{T}$ be a polyhedral terrain composed of m triangular faces and V a set of n viewpoints on or over $\mathcal{T}$. A point q on $\mathcal{T}$ is t-well-visible in relation to V if and only if every closed halfspace defined by a vertical plane through q contains at least t points of V visible from q (see Figure 4.24). The good-visibility depth of q relative to V, denoted by $gvd_V(q)$, is the maximum t such that q is t-well-visible in relation to V. The k-th good-visibility region relative to V, denoted by $gvr_V(k)$, is the set of all points q on $\mathcal{T}$ such that $gvd_V(q) = k$. Observe that $gvr_V(k)$ can be formed by several connected components of $\mathcal{T}$. The good-visibility map of $\mathcal{T}$ relative to V, denoted by $gvm(V)$, is the subdivision of $\mathcal{T}$ determined by the set of all k-th good-visibility regions $gvr_V(k)$.

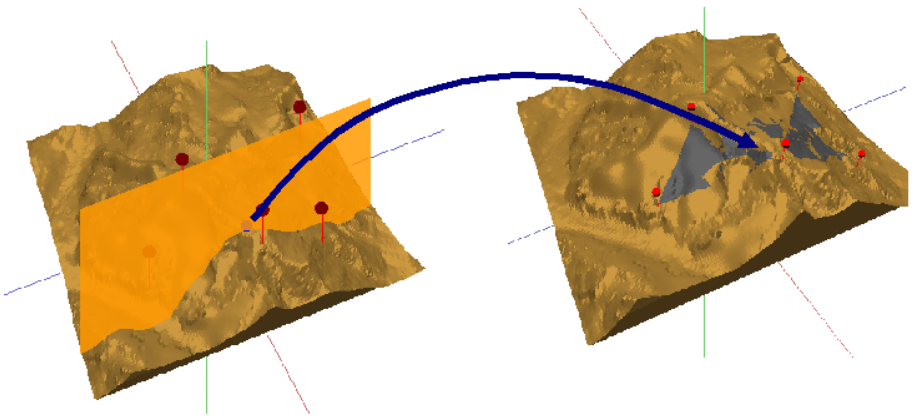


Figure 4.24: *Intuitive idea of good-visibility over a terrain.*

Let V_q be the subset of points of V that are visible from a point q on $\mathcal{T}$. We denote by q^*, V^* and V_q^* the orthogonal projection of q, V and V_q onto the domain $\mathcal{T}^*$ of $\mathcal{T}$, respectively. We also denote by $gvr_V(k)^*$ the orthogonal projection of the k-th good-visibility region $gvr_V(k)$ onto $\mathcal{T}^*$. Finally, we denote by $gvm(V)^*$ the subdivision of $\mathcal{T}^*$ determined by the set of all regions $gvr_V(k)^*$, and $gvm_r(V)^*$ the restriction of $gvm(V)^*$ to a region r of $\mathcal{T}^*$.

We will first compute $gvm(V)^*$ on the domain $\mathcal{T}^*$ of $\mathcal{T}$ and next we will lift up $gvm(V)^*$ to $\mathcal{T}$ to obtain $gvm(V)$. Our algorithm to compute $gvm(V)^*$ is based on Lemma 5, similar to the Lemma 1 for the two-dimensional case.

Lemma 5 *For any $q \in \mathcal{T}$, $gvd_V(q) = ld_{V_q^*}(q^*)$.*

The algorithm starts with the computation of the visible region T_v of $\mathcal{T}$ for each of the n points $v \in V$ and its orthogonal projection T_v^* onto the domain $\mathcal{T}^*$. This can be done, by using the algorithm of Katz et al. [KOS92], in $O(n((m\alpha(m)+k)\log m))$ time, where $k \in O(m^2)$ is the maximal combinatorial complexity among the n visibility maps T_v. The overall combinatorial complexity of the n visibility maps T_v, and consequently of their n projections T_v^*, is $O(nm^2)$.

Next, the algorithm computes the overlay $\mathcal{O}$ of the n planar subdivisions T_v^*. All points in a cell c of $\mathcal{O}$ are seen from exactly the same subset V_c of points of V. Observe that two cells $c \neq c'$ of $\mathcal{O}$ may exist, so that $V_c = V_{c'}$, that is the same as saying that the points in c and c' are seen from the same subset of points of V. The overlay $\mathcal{O}$ can be computed by using an algorithm to find segments intersections like the one in [Bal95], in $O(n^2m^4)$ time and space.

Finally, for each cell c of $\mathcal{O}$ whose points are seen from the subset V_c of V, Lemma 5 states that $gvm_c(V_c^*)$ can be computed as $dm_c(V_c^*)$, the depth map of the set V_c^* restricted to c. Then, we have:

$$gvm(V)^* = \bigcup_{c \in \mathcal{O}} dm_c(V_c^*)\,.$$

We compute $dm_c(V_c^*)$ by intersecting cell c with the depth contours determined by $dm(V_c^*)$. Assuming that on average the complexity of the cell c is constant (however, the complexity of a single cell can be superlinear in the worst case) and since $|V_c^*| \in O(n)$, this can be done in $O(n^2)$ time.

To summarize, we conclude with the following theorem.

Theorem 6 *The set $gvm(V)^*$, and consequently the good-visibility map $gvm(V)$, can be computed in $O(n^4m^4)$ time.*

4.8.1 Visualizing good-visibility maps on Terrains

Lemma 5 states that the visibility depth of a terrain point q is the location depth of its projection q^* relative to the projections of the visible viewpoints of q. Consequently, a good-visibility map on a terrain $\mathcal{T}$ can be approximated by applying the algorithm described in Section 4.5.2 to the domain $\mathcal{T}^*$, storing the result in a texture and reprojecting this texture to $\mathcal{T}$ using texture mapping. However, we have to redesign the process of computing the texture MVM representing the overlay $\mathcal{O}$ of the collection of planar

subdivisions T_v^*, $v \in V$. In order to obtain the texture MVM containing the multi-visibility map the algorithm explained in Section 3.4 in the visibility chapter is used. There can also be found the little changes that have to be done in the computation of MVM to take into account restricted visibility, as in the two-dimensional case.

4.8.2 Results

In this section, some examples and running time tests generated using our implementation are presented. All tests and images have been carried out on a laptop equipped with an Intel Core 2 Duo P8400 at 2.26GHz, 4GB of RAM and a GeForce 9600M GT.

We have tested our implementation with different sets of view points on two different terrains, the Mount Kilimanjaro modelled with 100.000 faces and the Mont Blanc mountain modelled with 40.000 faces.

Figure 4.25 shows a good-visibility map on the Mount Kilimanjaro. The good-visibility regions are colored in a grey gradation according to their depth.

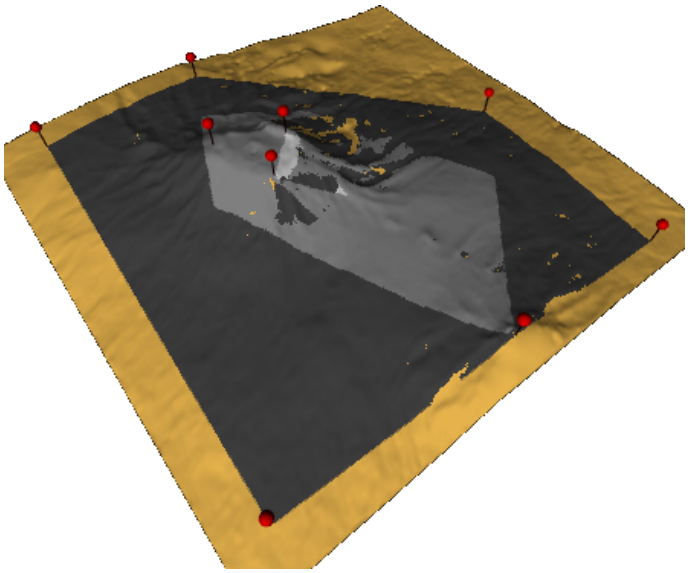


Figure 4.25: *Good-visibility map on the Kilimanjaro Mount.*

Figure 4.26 shows the good-visibility map (last image) of the Mont Blanc mountain for 17 viewpoints. The first six images correspond to the regions of level greater or equal to k, $k = 1 \dots 6$.

Figure 4.27 shows an example where some viewpoints have restricted visibility range.

Figure 4.28 shows how the running time increases depending on the number of view-

Figure 4.26: *The six first images show each of the six good-visibility levels present in the good-visibility map. The last image shows the good-visibility map on the terrain.*

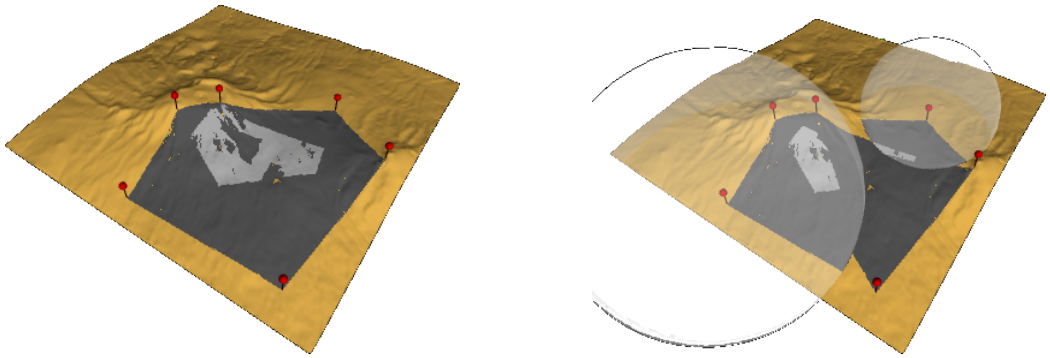


Figure 4.27: *The left image shows the good-visibility map obtained without restricted viewpoint while in the right one we can see the good-visibility map obtained when two viewpoints have restricted visibility.*

points. Observe that its behavior is the same as in the 2D case for generic obstacles.

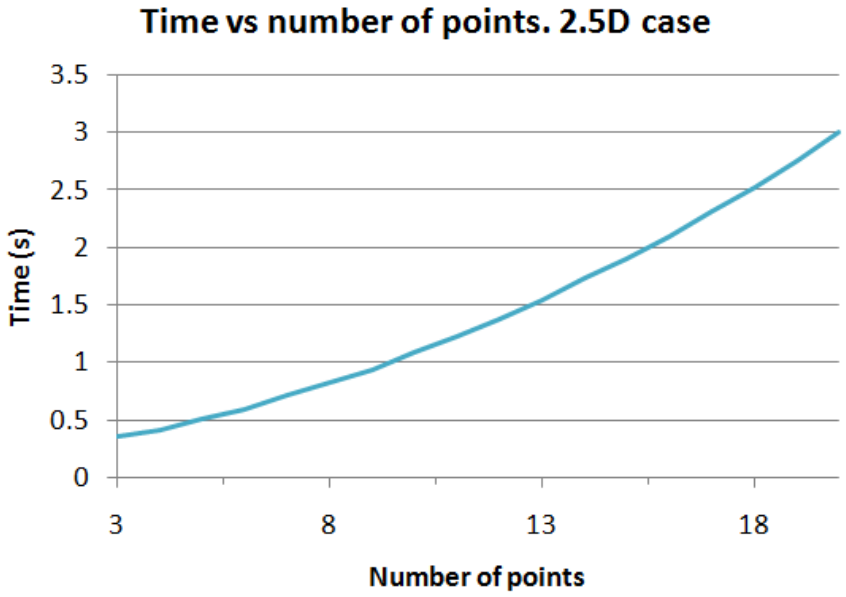


Figure 4.28: *Running time when the number of viewpoints increases.*

Chapter 5

3D good-visibility map computation using CUDA

In this chapter we present a new algorithm that runs in the graphics hardware for computing the depth contours. Moreover, an algorithm for computing three-dimensional good-visibility maps using CUDA is presented, as well as the results obtained, which are thoroughly described.

5.1 Introduction

After developing new algorithms for computing good-visibility maps from 2D scenes and on terrains using the GPU, the following and final step of our work is based on the three-dimensional space. 3D good-visibility maps might have potential applications in diverse areas such as architecture, building illumination or vigilance of spaces.

To the best of our knowledge, there is no algorithm capable of computing and visualizing depth contours from a set of points in a three-dimensional space, thus an algorithm for computing them is presented. The algorithm runs in the GPU and is implemented using the CUDA language provided by NVIDIA. The QHull program is then used to visualize one or more depth contours.

Afterwards a scene composed by triangle obstacles is added to the latter algorithm and it is accordingly modified to be able to compute 3D good-visibility maps using a voxelization of the space.

5.2 Computation of 3D Depth Contours

As far as we know, there is no efficient and implementable algorithm for computing Depth Contours [RS04] in three dimensions. In this Section, by working towards practical solutions, we describe a CUDA implementation for computing the Depth-k Contours of a set V of points in $\mathbb{R}^3$. We also present an algorithm for computing the bagplot. Finally, we provide experimental results obtained with the implementation, using real and synthetic data sets that show the effectiveness and efficiency of our approach.

5.2.1 Half-space Depth, Depth Regions and Depth Contours

Depth Contours and Depth Regions in $\mathbb{R}^d$ are defined similarly as in the two-dimensional case.

Given a set V of points in $\mathbb{R}^d$, the Half-space Depth of a point $p \in \mathbb{R}^d$ relative to V, denoted by $d_V(p)$, is the minimum number of points of V lying in any closed half-space whose bounding hyperplane passes through p. The more centrally located is p, the higher its Half-space Depth. For a set V of $n \geq d+1$ points, the Half-space Depth $d_V(p)$ is an integer in the range $0, .., n$. It is 0 when p lies outside the Convex Hull $CH(V)$ of V and n when all points of V coincide with p. For sets V in general position, meaning that no $d+1$ points lie in a common hyperplane, the Half-space Depth $d_V(p)$ is bounded above by $\lfloor n/2 \rfloor$. The latter occurs when V is symmetric about p.

The Depth-k Region of V, represented by $D_k(V)$, is the set of all points $p \in \mathbb{R}^d$ with $d_V(p) \geq k$. If $k \leq \lceil n/(d+1) \rceil$, the region $D_k(V)$ is nonempty [Rad46]. Points of depth at least $\lceil n/(d+1) \rceil$ are called center points of V, and the nonempty region $D_{\lceil n/(d+1) \rceil}(V)$, denoted by $C(V)$, is the Center of V. For any set V of $n \geq d+1$ points in general position and any positive integer $j \leq n$, we have

$$D_k(V) = \bigcap_{h \in \mathcal{H}_{\geq n-k+1}(V)} h \,,$$

where $\mathcal{H}_{\geq x}(V)$ is the set of all closed half-spaces that contain x points of V and have d points of V in its boundary [ASW08]. The Depth Regions form a nested sequence: $D_{k+1}(V) \subseteq D_k(V)$. The outermost Depth Region $D_1(V)$ is the Convex Hull of S: $D_1(V) = CH(V)$. For any k, the Depth Region $D_k(V)$ is a convex polytope (not necessarily of full dimension), because it is an intersection of a finite number of half-spaces bounded by $CH(V)$.

The Depth-k Contour of V, denoted by $C_k(V)$, is the boundary of $D_k(V)$. Note that the vertices of $C_k(V)$ are either points from the original set V or new points from the

intersection of hyperplanes through d points of V. The collection of all $C_k(V)$ determines the subdivision of $\mathbb{R}^d$ in regions whose points have the same Half-space Depth relative to V and it provides a comprehensive view of some shape characteristics of V (see Figure 5.1).

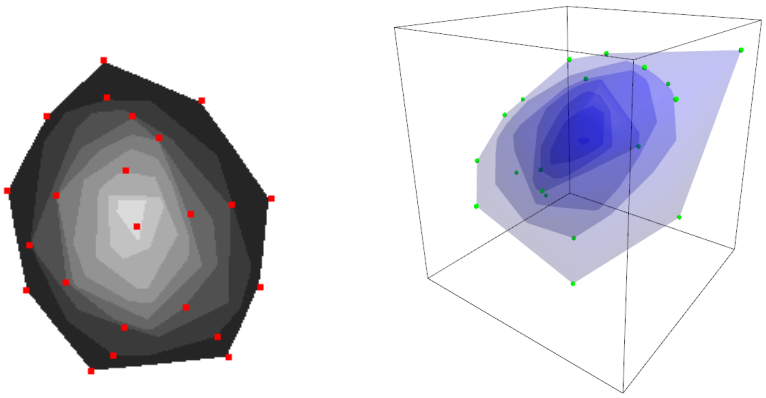


Figure 5.1: *Examples of Depth Contours for sets of points in* $\mathbb{R}^2$ *and* $\mathbb{R}^3$.

The Half-space Median, T^*, of the set V is the center of gravity of the deepest Depth-k Contour. The bag is the region $\mathcal{B}$ surrounding T^* containing the $n/2$ points of V with largest Half-space Depth. The fence $\mathcal{F}$ is obtained by inflating $\mathcal{B}$, relative to T^*, by a factor of 3.

From [ASW08] we know that in $\mathbb{R}^3$, the Depth-k Contour $C_k(V)$ has $O(n^2)$ complexity and it can be obtained in $O(n^{2+\epsilon})$ time, for any $\epsilon > 0$, computing the upper convex hull of the $(k-1)$-level and the lower convex hull of the $(n-k)$-level of the arrangement dual to V of n planes. Consequently, all the Depth Contours of V can be computed in $O(n^{3+\epsilon})$ time and $O(n^3)$ space. The Half-space Median and the bag can be also computed in the same time.

5.2.2 Computing Depth Contours using CUDA

The algorithm described in [KMV02, FG06] to compute Depth Contours in $\mathbb{R}^2$ using a GPU is difficult to export to $\mathbb{R}^3$ because in this case the dual spaces are more difficult to construct and their treatment is not straightforward. Moreover, instead of analyzing a set of pixels in $\mathbb{R}^2$ we need to analyze and maintain a set of voxels in $\mathbb{R}^3$, which is even more difficult. Because of all of these reasons we have designed a more basic algorithm that is affordable by the todays GPUs hardware.

Our $\mathbb{R}^3$ approach is based on the $\mathbb{R}^2$ algorithm proposed by Mustafa et al. in [SKV06] and on our algorithm explained in Section 4.5.

The definition of the problem states that the depth map of V can be obtained by computing the Half-space Depth of each point in the space inside the convex hull of V. Since we have an infinite number of such points, we can discretize the latter space into voxels forming a d-dimensional grid of side W and computing the Half-space Depth for each of these voxels.

Thus in a first implementation, the algorithm was designed in two steps as follows.

1. For each plane h determined by every three different points of V, the number of points of V that are located on its left and right, denoted $h_l(V)$ and $h_r(V)$ respectively, are counted. Then, the *level* of h is computed as $\min(h_l(V), h_r(V))$.

2. The space is discretized into voxels and every one of these voxels v is tested to determine its Half-space Depth with respect to V. With the information computed in the last step, for every plane h generated by V we test if v is on its left or on its right and the new level of v, denoted $d_V(v)$, is computed as $min(d_V(v), h_l(V))$ or $min(d_V(v), h_r(V))$, respectively.

The following sections are dedicated to explaining in more detail these two steps.

Computing the level of the planes

Since the generation of a plane through three different points of V and the computation of its level is independent from the others, the process of generating all such planes and computing their level can be parallelized by using a single thread for every plane.

Let n be the number of points of V. The simplest way to have all the planes generated by V would be to create a three-dimensional array H where the tuple of the three indices (i_1, i_2, i_3) of each element would codify the three points $V_{i_1}, V_{i_2}, V_{i_3}$ generating a plane. Then, each CUDA thread of the kernel would be responsible for computing the level of a single plane.

However, we are not interested in all the possible tuples of three points of V, but only in the *distinct* planes ($\binom{n}{3}$ elements instead of n^3). By using this first approach we might lose computation power and waste memory space. For this reason we use a one-dimensional array H of size $\binom{n}{3}$ in which each element corresponds to a unique plane generated by three points of V. By doing this we execute the CUDA kernel with the exact number of useful threads.

We use the function *IndicesComputation* (see Algorithm 6) inside the CUDA kernel

to compute the indices of the three points generating a plane from the current thread *index*. *IndicesComputation* is an injective function that for each index $0 \leq index < \binom{n}{3}$ of H returns three indices i_1, i_2 and i_3 satisfying $0 \leq i_1 < i_2 < i_3 < n$. This function takes $O(n)$ time for every plane computation. Consequently, the time complexity of the algorithm is not affected by this linear process and the three indices can be obtained inside the parallel computation.

Algorithm 6: IndicesComputation

Input: thread index *index*, Number of points n.

Output: i_1, i_2, i_3.

$i_1 \leftarrow -1$;

$c \leftarrow 0$;

$n_x \leftarrow n$;

repeat

  $i_1 \leftarrow i_1 + 1$;

  $n_x \leftarrow n_x - 1$;

  $c \leftarrow c + \frac{n_x*(n_x-1)}{2}$;

until $c \leq index$;

$c \leftarrow c - \frac{n_x*(n_x-1)}{2}$;

$n_x \leftarrow n_x + 1$;

$i_2 \leftarrow i_1$;

repeat

  $i_2 \leftarrow i_2 + 1$;

  $n_x \leftarrow n_x - 1$;

  $c \leftarrow c + (n_x - 1)$;

until $c \leq index$;

$c \leftarrow c - (n_x - 1)$;

$i_3 \leftarrow (index - c) + i2 + 1$;

$return(i_1, i_2, i_3)$;

In order to compute the plane levels, we assign a CUDA thread of the kernel to each index of the final array H. The three indices i_1, i_2, i_3 are obtained from *index* by using the *IndicesComputation* function. They determine the points $v_{i_1}, v_{i_2}, v_{i_3}$ which generate the current plane. All these threads are treated in a parallel way by CUDA and are executed in blocks (see Algorithm 7).

Since we have a loop visiting all the points used by each thread it is recommended to use the shared memory to store V. In order to do this we can use the 32 first threads of each block to copy one point per thread from the input array of points to a shared array.

Algorithm 7: PlanesLevel CUDA kernel

Input: Set of points V, Number of points n.

Output: unsorted H.

$index \leftarrow$ threadPosition.x ;

$i_1, i_2, i_3 \leftarrow IndicesComputation(index)$;

$h(i_1, i_2, i_3) \leftarrow$ plane determined by the points $V[i_1], V[i_2]$ and $V[i_3]$;

__shared__ float3 *points=new float3[n] ;

$left \leftarrow 0$;

for $part = 0$ *to* $n/32$ **do**

  if $index < 32$ **then**

    $points[index] \leftarrow S[part * 32 + index]$;

  end

  __syncthreads();

  for $r = 0$ *to* 32 **do**

    $read \leftarrow part * 32 + r$;

    if $read \neq i_1$ *AND* $read \neq i_2$ *AND* $read \neq i_3$ **then**

      $p \leftarrow points[r]$;

      if p *is at left of* $h(i_1, i_2, i_3)$ **then**

        $left \leftarrow left + 1$;

      end

    end

  end

end

$level \leftarrow left$;

$H[index] \leftarrow (level, i_1, i_2, i_3)$;

return H;

Afterwards, we need to synchronize all the threads to avoid accessing a shared point which has not been stored yet. When all the threads have been synchronized, it can be tested to see if the 32 previously stored points are on the left or on the right of the current plane. By repeating this procedure until all the points of V are tested we ensure that H will contain the *level* of every plane Π and the indices of points of V generating Π. For every plane, its relative position to every point of V has to be tested, thus the algorithm takes $O(n^4/MP)$ time, where MP is the number of total parallel processors that the GPU has.

Computing Depth Contours

Once H has been calculated, then the level for every voxel of the space dicretization can be computed. The kernel shown in the Algorithm 8 is responsible for computing it. Every voxel of the discretization is represented by a thread, and all such threads are executed by this kernel in a parallel fashion. The kernel receives H, V, the number of points n and the number of total planes in H, defined as pn, as input parameters and it stores the result in the output array DM.

The kernel first initializes to infinite the output of the current voxel, denoted by $DM[index]$, and then performs a loop in order to check the level of all the planes stored in H. Inside this loop it is tested to see if the current voxel v is on the left or on the right of h, and depending on the side it is located, $min(DM[index], h)$ or $min(DM[index], n-h-3)$ is assigned to the ouput $DM[index]$, respectively.

Visualization

Using the information of the level of every discretized voxel of the space, it is easy to draw all the voxels inside a certain level Depth-k Contour $C_k(V)$. Obviously, the more voxels the space is discretized in, the more precise will be the depth contours visualization (Figure 5.2 contains some examples computed using this approach).

5.2.3 A better approach

At some point we realized that the latter presented algorithm is redundant because we do not actually need to discretize the space in voxels and give a discretized solution. When we compute the level of each distinct plane in the first step of the algorithm, in fact we already have all the information needed to compute a certain Depth-k Contour $C_k(V)$. At this point we only have to do an intersection between the halfspaces determined by all

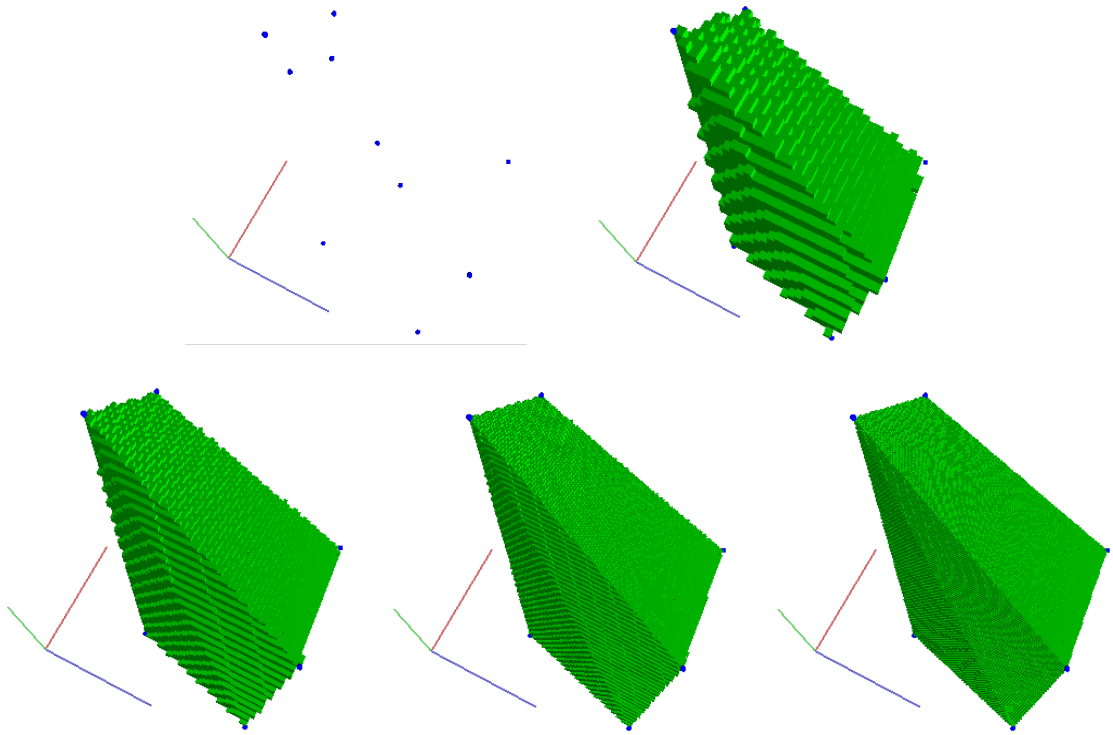


Figure 5.2: *The top-left image shows how the 10 points in the space are placed. The other pictures show the $C_1(V)$ discretized in a space of side 32, 64, 128 and 256 voxels, respectively.*

Algorithm 8: 3DDepthMap CUDA kernel

Input: Set of points V, Number of points n, Array H, Number of planes np

Output: DM.

$index \leftarrow$ voxelIndex(threadPosition,BlockSize,GridSize) ;

$v \leftarrow voxelPosition$;

$DM[index] \leftarrow \infty$;

for $i = 0$ *to* np **do**

  $level \leftarrow H[i * 4]$;

  $i_1, i_2, i_3 \leftarrow H[i * 4 + 1], H[i * 4 + 2], H[i * 4 + 3]$;

  $h \leftarrow$ plane determined by the points $V[i_1], V[i_2]$ and $V[i_3]$;

  if v *is at left of* h **then**

    $DM[index] \leftarrow min(DM[index], level)$;

  end

  else

    $DM[index] \leftarrow min(DM[index], n - level - 3)$;

  end

end

return DM;

the planes with level k.

Therefore this new and better algorithm runs in two steps, where the first one is exactly the same as before except for the sorting we apply to the array H containing the level of all planes.

The two steps, that are explained in detail in the following Sections, are:

1. Computing the level of the planes: for each plane h determined by every three different points of V, the number of points of V that are located on its left and right, denoted $h_l(V)$ and $h_r(V)$ respectively, are counted. Then, the *level* of h is computed as $\min(h_l(V), h_r(V))$ and the set of all planes is sorted according to this level.

2. For a given k, the Depth-k Contour $C_k(V)$ is computed as the boundary of the intersection between the halfspaces defined by all the planes of level k. Repeating this process for different k values, distinct Depth-k Contours can be obtained.

The direct finding of the intersection between all the planes with level k gives us two important advantages: the computational time is reduced by avoiding some costly computations (even performed in a parallel fashion) and the visualization of the solution

is easier and much better because now we have to paint a well defined surface instead of a volume composed by voxels. Figure 5.3 shows a comparison between the final result of both methods.

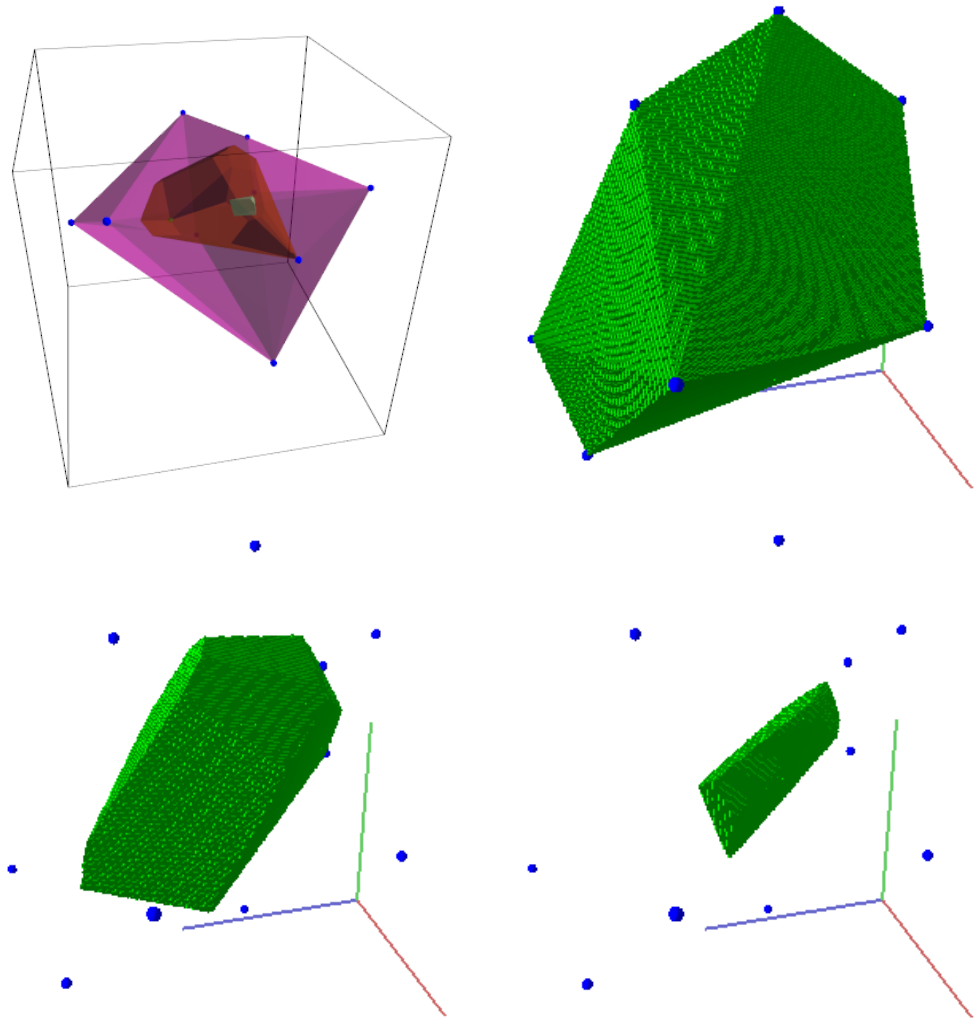


Figure 5.3: *The top-left image shows $C_1(V)$, $C_2(V)$ and $C_3(V)$ at the same time using the new approach while the three other images show only one distinct $C_k(V)$ each using our first proposed algorithm.*

Computing the level of the planes

Since the output for this step is the same as for the initially proposed algorithm, the first part of the method for this new approach is exactly the same as before. However, taking into account the second step in the following section, we need to sort the planes by their level as we need to access as fast as we can all the planes with a particular level. For

this reason, we use a sorting CUDA kernel to sort the one-dimensional array H. This kernel is responsible for sorting the elements of H (where each element has three point indices and a level) from lower to higher level (see [SA08] and [CT08] for sorting algorithms implemented in CUDA). This process is executed only once and it takes $O(\frac{n^3 \log n}{MP})$ time. It simplifies the search and the access time of planes of a certain level, which is executed every time a new Depth-k Contour is wanted. Thus, the whole algorithm to compute H takes $O(\frac{n^4+n^3 \log n}{MP})$ time.

By doing this procedure, the array H contains the list of planes ordered by its level, therefore we can obtain the first plane of level k in $O(\log n)$ time. Moreover, we can access all planes of level k in $O(n^2)$ time, instead of the required $O(n^3)$ time if the unsorted H is used.

Computing Depth Contours

This part of the algorithm that finds the intersection between all halfspaces of level k is more difficult to solve with a parallel algorithm because for every distinct k we can have many planes involved. A possible way to solve the problem could be to assign one CUDA thread to compute one single Depth-k Contour. However, it is not a good solution because of the huge amount of planes with level k that would be processed in a sequential way by every parallel thread. For this reason we decided to use the QHull [Web] software for computing the Depth-k Contour as the intersection of all halfspaces of level k. QHull provides robust and fast computation of the convex hull of a set of points and the intersection of a set of halfspaces in $\mathbb{R}^3$.

From the output provided by QHull we can easily visualize the Depth-k Contour. Taking into account that there are $O(n^2)$ planes with level k and that they are accessed in $O(n^2)$ time, the Depth-k Contour can be computed in $O(n^2 \log n)$ time.

The huge amount of memory needed to compute and store all distinct planes generated from the set V might be a problem. There are $\binom{n}{3}$ distinct planes stored in H and every one of these planes h is stored using 4 integer values, three of them for indexing the 3 points of S that generates it and the last one to store its level. Thus the space needed is $\binom{n}{3} * 4 * sizeof(integer)$. In order to reduce this amount of memory and to be able to compute Depth-k Regions from larger sets of points, we could compute the Depth-k Contours one by one and store only the planes of level k. Obviously, this solution takes more time than the proposed approach if more than one Depth Contour is computed.

5.2.4 The Bagplot

The bagplot depicts the set of points V, the Half-space Median T^*, the bag $\mathcal{B}$ and the fence $\mathcal{F}$.

The Half-space Median T^* is the center of gravity of the deepest non-empty Depth-k Contour. In order to compute T^* we do a binary search over all possible Depth-k Contours ($0 \leqslant k \leqslant n/2$) to test if a certain Contour is empty or not. Deciding whether or not an intersection of halfspaces is empty can be solved in linear time in the number of halfspaces by linear programming [Meg83]. This search is stopped when we find a k^* such that $C_{k^*+1}(V)$ is empty and $C_{k^*}(V)$ is non-empty.

The bag $\mathcal{B}$ can be constructed using the information stored in H together with T^*. For every k ($0 \leq k \leq k^*$), let d_k be the number of points of V inside the Depth-k Contour $C_k(V)$, that is to say, the number of points inside the intersection between all the halfspaces of level k. In order to compute which points of V are contained in $C_k(V)$ we use another CUDA kernel. The input of the kernel is the set V and the array H. The kernel tests in a parallel way if each point of V is on the same side of all the planes of level k with respect Half-space Median T^*. If the point v_i is on the same side for all the planes then v_i is interior to $C_k(V)$. The output of this kernel is a list of boolean values, one for each one of the points v_i in V, indicating if any point of V is inside or outside of $C_k(V)$. The sum of the elements of this list outputs d_k. Since there are, at the most, $O(n^2)$ planes of level k generating $C_k(V)$, the algorithm has a time cost of $O(n^3/MP)$. Then, we use a binary search algorithm to determine the value of k satisfying $d_k \leq \lfloor n/2 \rfloor \leq d_{k-1}$. Finally we have to linearly interpolate between $C_{k-1}(V)$ and $C_k(V)$, relative to the Half-space Median T^*, to obtain $\mathcal{B}$. The bagplot $\mathcal{B}$ is obtained by QHull by computing the convex hull of the interpolated points found before. Since we have $O(n)$ Depth-k Contours and the time needed to compute the points of V inside $D_k(V)$ is $O(n^3/MP)$, the binary search is done in $O(\frac{n^3 \log n}{MP})$ time.

The fence $\mathcal{F}$ of V is also a convex polyhedron and it can be computed by inflating 3 times the bag $\mathcal{B}$ (see Figure 5.5) relative to the Half-space Median T^*. The points of V external to the fence are considered outliers.

Observe that by using a similar process we can find a convex polyhedron containing any desired percentage of points of V, not only 50% as in the case of the bag.

5.2.5 Results

All images and tests have been carried out on a PC equipped with an Intel Core 2 Quad Q9550 at 2.83GHz, 4GB of RAM and a graphics card NVIDIA GeForce GTX 280 which has 240 parallel cores or processors, thus in this case we have $MP = 240$. The version of the CUDA compilator used is the release 2.1.

In our implementation, for every one of $\binom{n}{3}$ planes we need 8 bytes to store 3 short integers which are the indices to the three points generating it and 1 short storing its level. Consequently, we need approximately $\binom{n}{3} * 8$ bytes of memory to store them. A present day non-professional GPU with $1GB$ of graphics memory would permit us to compute Depth Contours for a set of about $n = 900$ points, which is sufficient in most of the cases. In our experiments we have used random data points in $\mathbb{R}^3$ following a uniform distribution as well as points from real data. For the running time calculations we have used the average of 10 repetitions for each single value shown in the images, every one with a new set V.

Figure 5.4 shows a set of 100 random data points from which some nested Depth Contours are visualized. In Figure 5.5 we can see a real data set of 100 points, each one represented by three variables: the height and width of the head and the neck perimeter of a person. The figure also shows the bag and fence of the set visualized from different points of view. The bagplot helps to detect a significant correlation between the three variables, since the thinner the bag is, the more correlation the data set has. Figure 5.6 shows a set of 300 random data points and polyhedra containing distinct percentages of the points.

All running times present in the next images are based on a set V of points following a uniform random distribution. In Figure 5.7 we can see a plot that relates the number of points of a random set V and the time needed for the computation of the level of all the planes determined by three points of V. The increment of the total number of planes when n increases is also shown. We can observe that the running time for the parallel computation of H using CUDA is almost linearly dependent on the total number of planes generated by V. This computation is a pre-process step and, consequently, it is computed only once. The time taken by the computation of different Depth Contours and the bag for different sets of random points, when H has been already obtained, can be seen in Figure 5.8. For example, visualizing the bag of 900 points following a uniform distribution takes aproximately 9 seconds. Also observe that the running time to compute the Depth Contour for a fixed level k increases when n becomes larger. The reason is that a bigger n implies also a bigger number of planes at level k (see Figure 5.9). Moreover, computing deeper contours of a set takes more time due to a similar reason: they are determined by

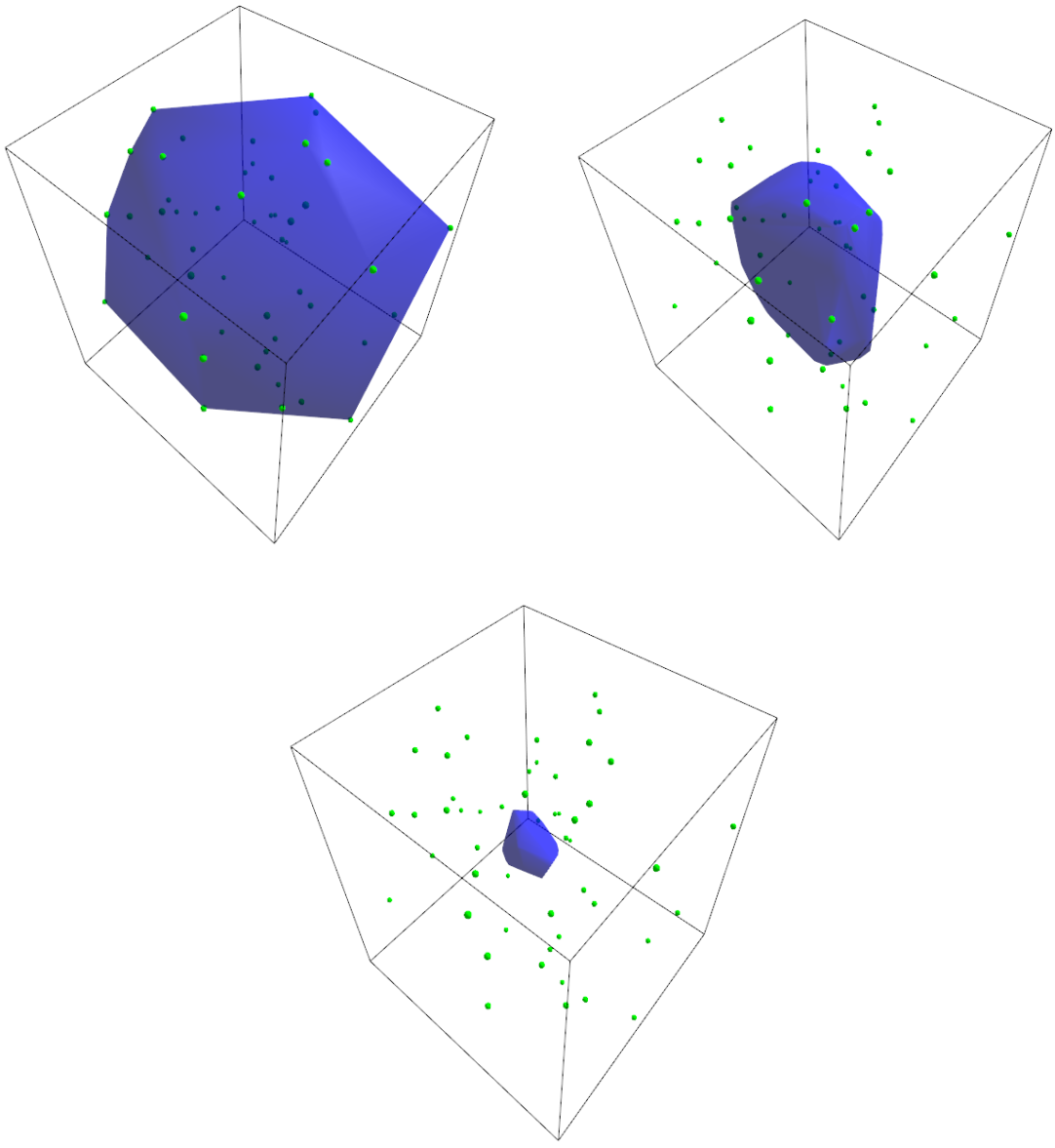


Figure 5.4: *Depth Contours* $C_0(V)$,$C_5(V)$ *and* $C_{15}(V)$ *of a set* V *with 50 points.*

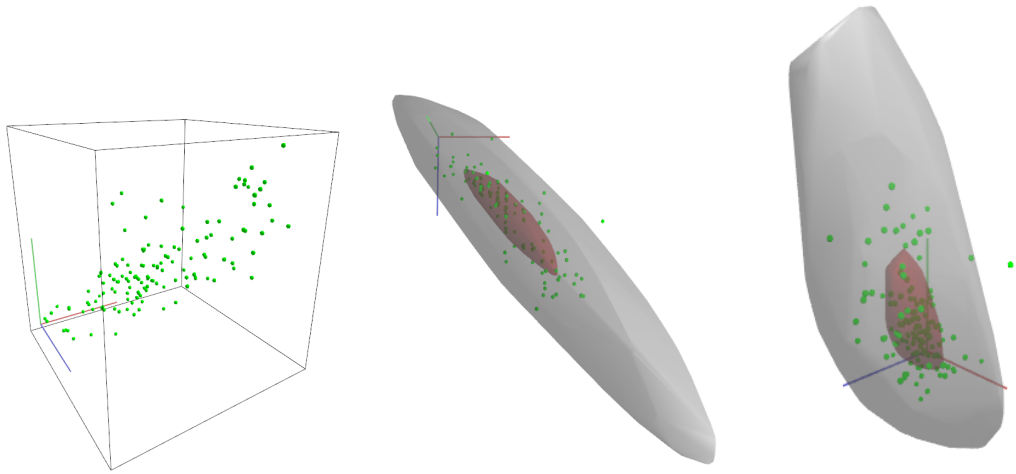


Figure 5.5: *Data set and its bag and fence visualized from different points of view. We can observe one outlier point.*

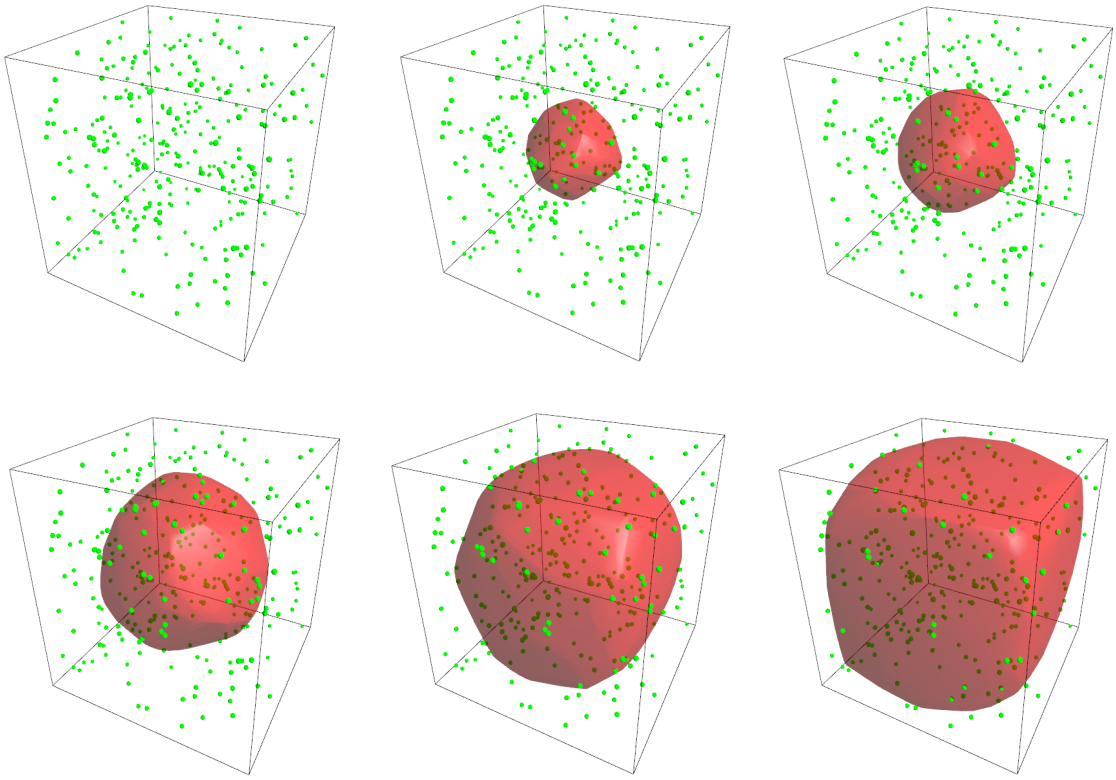


Figure 5.6: *Set of points and polyhedra containing 5%, 10%, 25%, 50% (bag) and 75% of the points.*

a much larger number of planes.

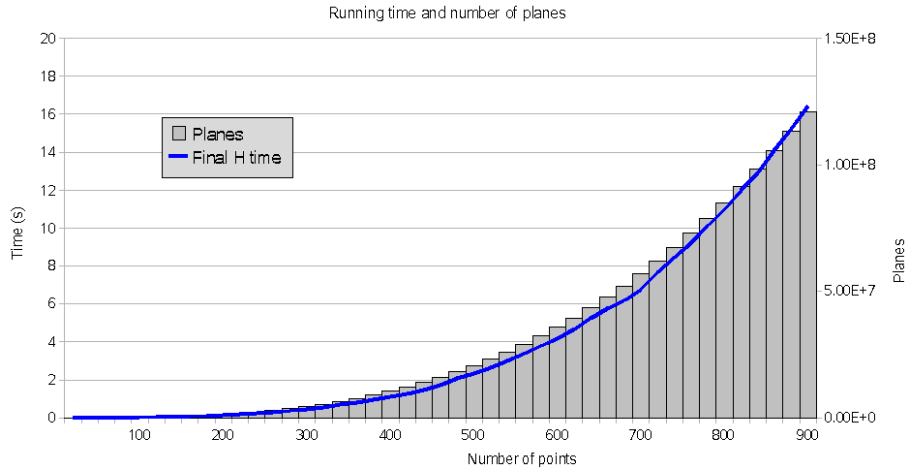


Figure 5.7: *Line and bar charts showing the time needed to compute H and the total number of planes generated when n increases.*

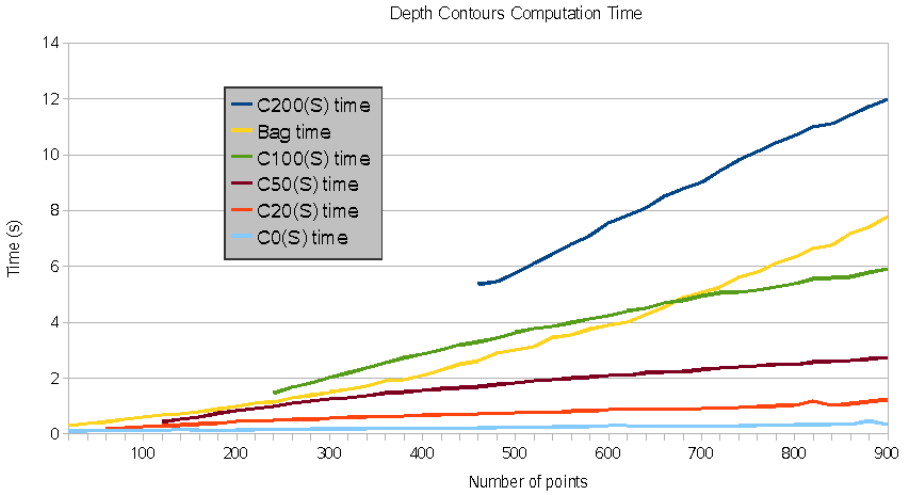


Figure 5.8: *Running time for the computation of some Depth-k Contours and Bag taking the data from H and using the QHull software when n increases.*

The running time for computing a certain level k is linearly dependent with n due to the increment of the number of planes of level k which is also linear with n, as can be observed in Figure 5.9.

Figure 5.10 shows the number of planes when the number of viewpoints increases.

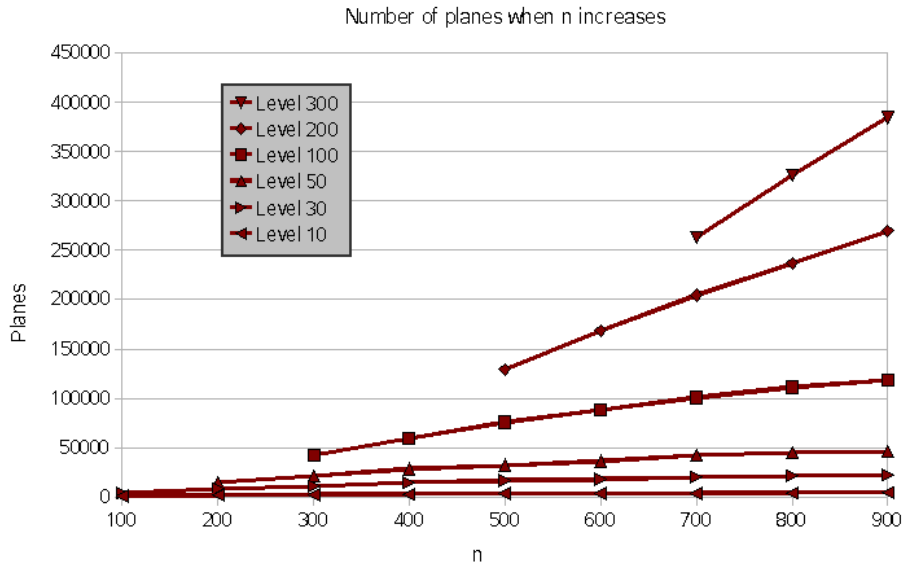


Figure 5.9: *The number of planes when n increases for some distinct levels is represented.*

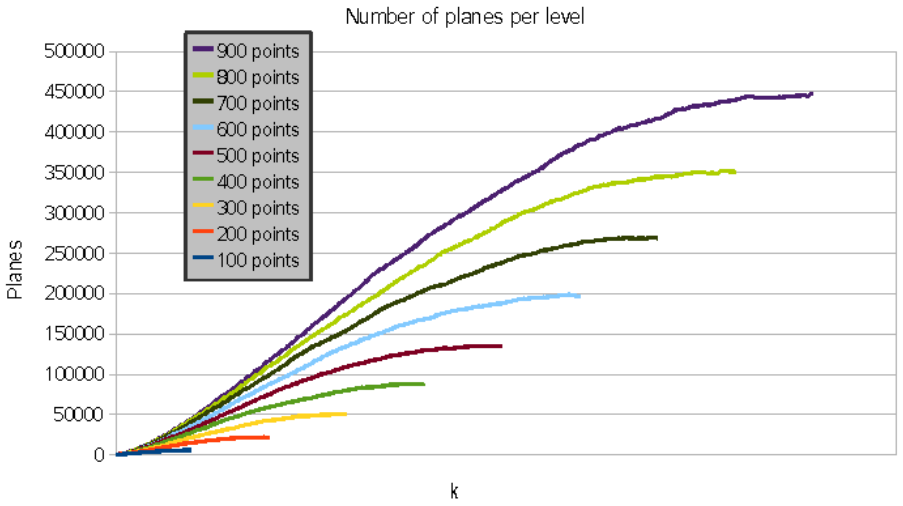


Figure 5.10: *Number of planes when k increases for some distinct n values.*

5.3 3D Good-visibility maps

In order to compute good-visibility maps in a three-dimensional space we need to add obstacles to the previously studied 3D depth contours. The simplest obstacles to deal with in $\mathbb{R}^3$ are triangles, thus our scenes will be composed by a set of viewpoints V and a set of triangles S playing the role of obstacles.

5.3.1 Definitions

Similarly to the 2D definition in Section 4.2 and extending the idea of the three-dimensional depth contours explained in the latter section, we can define the problem as follows.

Let V be a set of n viewpoints and S a set of m triangle obstacles in the space. We assume that no point of V is interior to an obstacle in S. The location depth of an arbitrary point q relative to V, denoted by $ld_V(q)$, is the minimum number of points of V lying in any closed halfspace defined by a plane through q. The k-th depth region of V, represented by $dr_V(k)$, is the set of all points q with $ld_V(q) = k$.

The free space F_S relative to S is the complement of S. Given two points $q \in F_S$ and $v \in V$, we say that a viewpoint v is visible from q if the interior of the segment with endpoints v and q remains completely inside F_S. A point q is t-well visible in relation to V and S if and only if every closed halfspace defined by a plane through q contains at least t viewpoints of V visible from q (Figure 4.1 shows the idea on the plane, which is equivalent). The good-visibility depth of q relative to V and S, denoted by $gvd_{V,S}(q)$, is the maximum t such that q is t-well visible in relation to V and S.

Let H_q be the set of halfspaces whose boundary plane contains q and $NS(q,h)$ the number of viewpoints of V visible from q contained in any halfspace h. Since by definition:

$$gvd_{V,S}(q) \;=\; \min_{h \in H_q} NS(q,h)\,,$$

it directly follows Lemma 7 which expresses the relationship between the good-visibility depth and the location depth of a point.

Lemma 7 *If V_q denotes the subset of points of V visible from q, then $gvd_{V,S}(q) = ld_{V_q}(q)$.*

The k-th good-visibility region relative to V and S, denoted by $gvr_{V,S}(k)$, is the set of all points q with $gvd_{V,S}(q) = k$. Observe that $gvr_{V,S}(k)$ can be composed by some non connected components (see Figure 5.11).

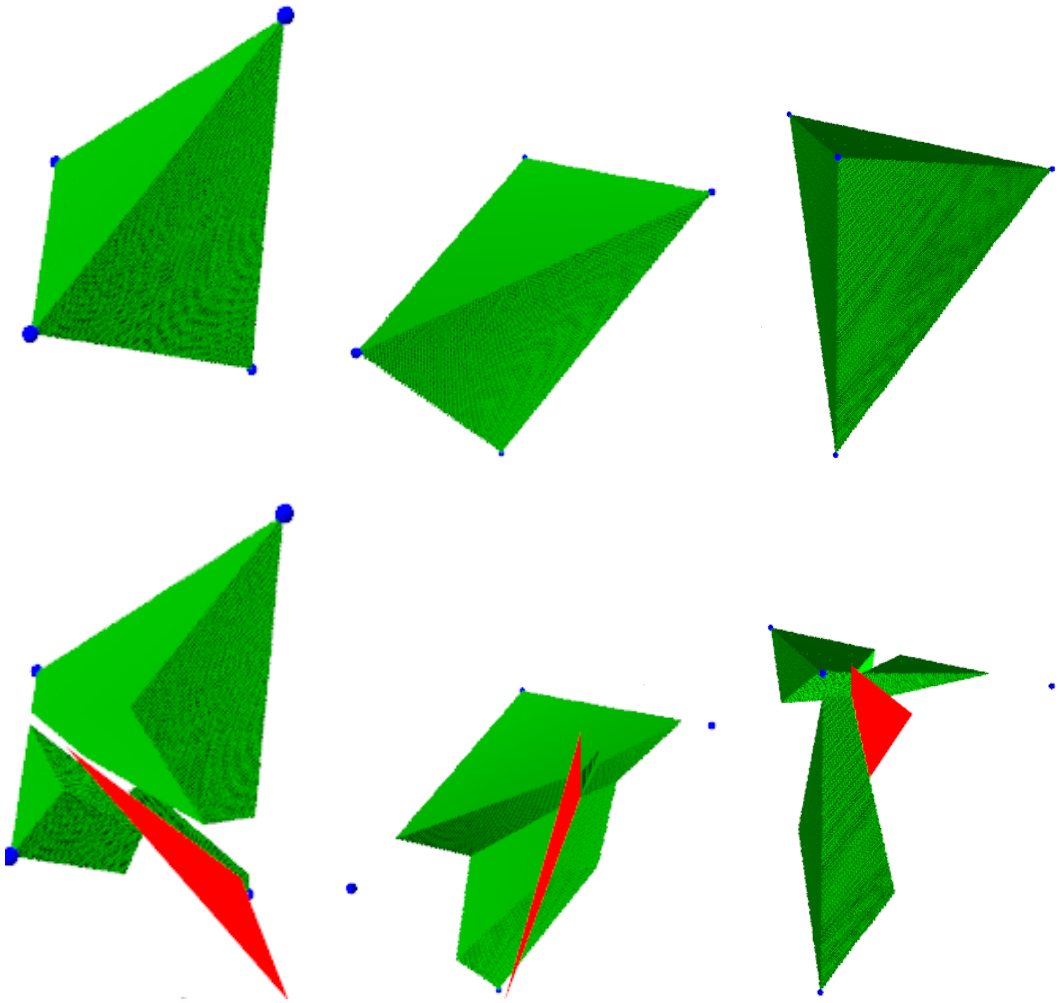


Figure 5.11: *Upper images show some views of the good-visibility map of a set of four points (equivalent to the Depth Contours because there are no obstacles). In the lower ones a triangle obstacle is added and the* 1 *good-visibility region becomes smaller and disconnected in some convex polyhedra.*

Lemma 8 *If S is empty or is external to the convex hull of V, $CH(V)$, then $gvr_{V,S}(k) = dr_V(k)$.*

The proof for the latter lemma is the same as the one used in the explanation of the exact algorithm for computing good-visibility maps in the plane (see Section 4.2).

We call the set of all good-visibility regions relative to V and S the good-visibility map of V and S and denote it with $gvm(V, S)$.

From now on we will focus on the non trivial case in which we have $n \geq 4$ viewpoints and $m \geq 1$ triangle obstacles intersecting $CH(V)$.

5.3.2 Computing good-visibility maps with CUDA

In this section we explain how our CUDA algorithm for computing good-visibility maps in a three-dimensional space works and its results are discused. Moreover, we present some considerations or improvements that can be dealt with as future work. The algorithm is based on the 2D approach detailed in Section 4.5.

We have presented two different versions of the 3D depth contours CUDA implementation. In this case we must base this new algorithm on the first (and worst) one because, unfortunately, when adding obstacles to the good-visibility map, its good-visibility regions can be composed by more than one polyhedra, thus a region of the good-visibility map can be non-convex or even composed by some convex or non-convex portions. For this reason we cannot use the QHull software in the case of visualizing the good-visibility regions. Therefore we must use the version which discretizes the space into voxels in order to compute the good-visibility map. The process works as follows.

1. Computing the information of the planes: for each plane h determined by every three different points of V, the number of points of V that are located on its left, denoted $h_l(V)$, are counted and it is stored as the level of h. Apart from counting the number of points lying on its left, we also store *which* are these points.

2. The space is discretized into voxels. For every one of these voxels v the set of points of V visible from v is computed, taking into account the set of triangles S. Then the algorithm performs a loop over all the planes generated by V. Each step of this loop tests if v is situated on the left or on the right of a plane h. If v is on its left, then the level assigned to v is computed as the number of points of V visible from v which are, at the same time, located on the left of h (which is computed in the previous step). Otherwise the level of v is the number of points visible from v and located on the right of h. The final level of v, denoted $gvd_{VS}(v)$, is the minimum level found in the latter loop.

Using the information of the level of every discretized voxel of the space, it is easy to draw all the voxels inside a certain good-visibility region $gvr_{V,S}(k)$. Obviously, the more

voxels the space is discretized in, the more precision the good-visibility map computation will have.

In the next sections, the details of this CUDA algorithm for computing good-visibility maps are presented.

Computing the information of the planes

This step is very similar to the one that computes the level of the planes in the 3D depth contours computation, in Section 5.2.2.

Every thread of the CUDA Grid represents a distinct plane generated by V and all of them reach the CUDA kernel in a parallel fashion.

In addition to counting the number of points of V on the left of each distinct plane h, it is also needed to store which are these points. In order to reach this goal, we can use one single bit to store the information about every point v_i in V. The bit in position i will contain a *one* when v_i is on the left of h, and a *zero* otherwise. Obviously, for every plane h, we need to store n bits, where n is the number of points in V. By using an *unsigned integer* we can store information for 32 points, but this limit can be easily incremented by using more unsigned integers to store this information.

Algorithm 9 shows how this first step works. We can observe that the algorithm is an extension of the one used in the three-dimensional depth contours computation.

Note that in order to activate a bit we use the shift operator. Moreover, this algorithm only works if $pointsAtLeft$ and $PH[index]$ are variables of a data type composed by, at least, n bits. If they are declared as unsigned integers, V will contain 32 points at most. However this limit can be easily avoided by using more memory space.

The time complexity of the algorithm is the same as in the case of computing depth maps. The algorithm takes $O(n)$ time for every distinct plane generated by V. The number of such planes is $\binom{n}{3}$, thus the the time to compute H and PH is $O(n^4/MP)$. The space needed is greater since we must also store the array PH. Both arrays contain $\frac{n}{3}$ elements, and one element in H needs 4 short integers while each element of PH occupies one unsigned integer for every pack of 32 points of V. Therefore the total space required is $\binom{n}{3} * 4 * 2bytes$ and $\binom{n}{3} * ceil(n/32) * 4bytes$ for storing H and PH, respectively. This $O(n^4)$ space complexity indicates that n cannot be incremented as in the depth contours computation.

Algorithm 9: PointsAtLeft CUDA kernel

Input: Set of points V, Number of points n.

Output: level of planes H, Points at the left of every plane PH.

$index \leftarrow$ threadPosition.x ;

$i_1, i_2, i_3 \leftarrow IndicesComputation(index)$;

$h(i_1, i_2, i_3) \leftarrow$ plane determined by the points $V[i_1], V[i_2]$ and $V[i_3]$;

__shared__ float3 *points=new float3[n] ;

$left \leftarrow 0$;

$pointsAtLeft \leftarrow 0$;

for $part = 0$ *to* $n/32$ **do**

  if $index < 32$ **then**

    $points[index] \leftarrow S[part * 32 + index]$;

  end

  __syncthreads();

  for $r = 0$ *to* 32 **do**

    $read \leftarrow part * 32 + r$;

    if $read \neq i_1$ *AND* $read \neq i_2$ *AND* $read \neq i_3$ **then**

      $p \leftarrow points[r]$;

      if p *is at left of* $h(i_1, i_2, i_3)$ **then**

        $left \leftarrow left + 1$;

        $pointsAtLeft \leftarrow pointsAtLeft + (1 << read)$;

      end

    end

  end

end

$H[index] \leftarrow (left, i_1, i_2, i_3)$;

$PH[index] \leftarrow pointsAtLeft$;

return H, PH;

Computing the good-visibility map

Once H and PH have been computed, then the level for every voxel of the space dicretization can be computed. The CUDA kernel shown in the Algorithm 11 is responsible for doing this. Similarly to the computation of the depth maps in the previous section, every voxel is represented by a thread of the CUDA execution. The kernel receives the array PH, the list of triangle obstacles S, the number of obstacles, the two arrays TV and TVI, and W_{TVI}, apart from all other input parameters explained before. The arrays TV and TVI are computed as a pre-process step and represent a voxelization of the space where each of these voxels contains the list of triangles of S intersecting it. W_{TVI} is simply the side, in voxels, of the obstacles voxelization. In Section 3.5.1 a more detailed explanation of these two arrays and how they are computed can be found. Using this structure we can compute the visibility of every voxel in a fast way (see Algorithm 10 for more details). The ouput is also an array GVM where each element contains the good-visibility level of a voxel of the discretization.

Every thread representing a voxel v executes the kernel once and the first step consists of testing which points of V are visible from v, taking into account the obstacles in S. See Section 3.5 for a detailed explanation of the visibility algorithm implemented also in CUDA (Algorithm 1). The visibility information for every v_i in V is stored using the bit i of an unsigned integer called $visiblePoints$. If the bit in the position i of $visiblePoints$ contains a 1, then v_i is visible from v, otherwise it is not visible.

Then the algorithm performs a loop testing all the planes. For every plane h, the level of v is updated. If v is located to the left of h, then a bit-to-bit AND operation is performed between $PH[index]$ and $visiblePoints$, and the result contains which points are visible from v and, at the same time, are to the left of h. The level of v is computed by counting the number of activated bits on this result. On the other hand, if v is located to the right of h, we obtain the bit-to-bit complement of $PH[index]$ and moreover we assign a zero on its bits i, j, and k, corresponding to the points v_i, v_j and v_k generating h. Then the same process between this new $PH[index]$ and $visiblePoints$ is done.

At the end of a loop step, the level of v, denoted by $GVM[index]$, is computed as $min(GVM[index], level)$. This process ensures that, at the end of the loop, the voxel v will obtain its actual good-visibility level.

This step is performed by all the voxels the space is discretized on. This means that the time complexity is dependent on the number of points n, the number of triangle obstacles m and also on the number of voxels used to discretize and compute the good-visibility

Algorithm 10: 3DPointVisibility

Input: Point p, Set of viewpoints V, Number of points n, set of triangles S, Number of triangles m, array TVI, array TV, grid side H

Output: *visiblePoints*.

```
visiblePoints ← 0;
for i = 0 to n do
    v ← V[i];
    vp ← segment(v, p);
    TVI_v ← voxelPosition(H, v);
    visible ← true;
    while visible and voxelInSegment(TVI_v, vp) do
        triangleList ← references from TV[TVI_v − 1] to TV[TVI_v];
        if nonEmpty(triangleList) then
            for every j in triangleList do
                triangle ← T[j];
                if intersection(vp,triangle) then
                    visible ← false;
                end
            end
        end
        TVI_v ← nextVoxel(H, vp);
    end
    if visible then
        visiblePoints ← visiblePoints + (1 << i);
    end
end
return visiblePoints;
```

Algorithm 11: 3DGVM CUDA kernel

Input: Set of points V, Number of points n, set of triangles S, Number of triangles m, Array H, Array PH, Number of planes np, array TVI, array TV, grid side W_{TVI}

Output: GVM.

$index \leftarrow$ voxelIndex(threadPosition,BlockSize,GridSize) ;

$v \leftarrow$ voxelPosition;

$visiblePoints \leftarrow 3DPointVisibility(v, V, n, S, m, TVI, TV, W_{TVI})$;

$GVM[index] \leftarrow \infty$;

for $i = 0$ *to* np **do**

  $pointsAtLeft \leftarrow PH[i]$;

  $i_1, i_2, i_3 \leftarrow H[i * 4 + 1], H[i * 4 + 2], H[i * 4 + 3]$;

  $h \leftarrow$ plane determined by the points $V[i_1], V[i_2]$ and $V[i_3]$;

  if v *is at left of* h **then**

    $resultPoints \leftarrow pointsAtLeft \& visiblePoints$;

  end

  else

    $max \leftarrow 0$;

    for $j = 0$ *to* n **do**

      $max \leftarrow max + (1 << j)$;

    end

    $pointsAtLeft \leftarrow pointsAtLeft - max - (1 << i_1) - (1 << i_2) - (1 << i_3)$;

    $resultPoints \leftarrow pointsAtLeft \& visiblePoints$;

  end

  $level \leftarrow numberOfBitsActivated(resultPoints)$;

  $GVM[index] \leftarrow min(GVM[index], level)$;

end

return GVM;

map. If we suppose that the grid of voxels is a cube with the three sides of the same size W, the time complexity is $O(\frac{W^3*(nm+\binom{n}{3})}{MP})$.

Therefore the whole algorithm (taking into account the two steps) runs in a total time of $O(\frac{n^4+W^3*(nm+\binom{n}{3})}{MP})$.

Once the second kernel is executed and the array GVM is obtained, H, PH, TVI and TV can be destroyed, thus the space occupied at the end of the whole process is $W^3 * 2bytes$, which is the space needed to store the good-visibility depth or level of every voxel.

5.3.3 Visualization of good-visibility regions

Since the final result obtained by our algorithm is a discretization of the solution, where each voxel contains the *level* corresponding to its central point, we can visualize them by applying different techniques often used in the volume rendering field. An easy-to-program visualization is the one used in Figures 5.12 and 5.13, where each voxel is represented as a cube of side $1/W$. However, a marching cubes technique can also be used. In the latter Figure a slightly transparent effect is added to the voxels.

5.3.4 Results

Figure 5.14 shows the running times for computing the good-visibility map from the scene shown in Figure 5.12 (using 1000 and 100000 triangles to represent it, respectively) when the number of viewpoints is incremented. We can observe that, obviously, the running time is larger when the discretization of the grid is larger too. These running times have been carried out on a computer equipped with an Intel Pentium 4 at 3.2GHz, 4GB of RAM and a graphics card NVIDIA GeForce GTX 280, the same as in the case of three-dimensional depth contours, explained in Section 5.2.5.

Restricted range visibility

In order to calculate good-visibility maps taking into account restricted visibility, we must only change the way the visibility is computed. In Section 3.5.2 there is an explanation of how this process is affected.

Figure 5.15 shows a good-visibility map computed from a set V composed by some viewpoints with unrestricted visibility, one point using restricted visibilty, and a scene composed by a set S of triangles.

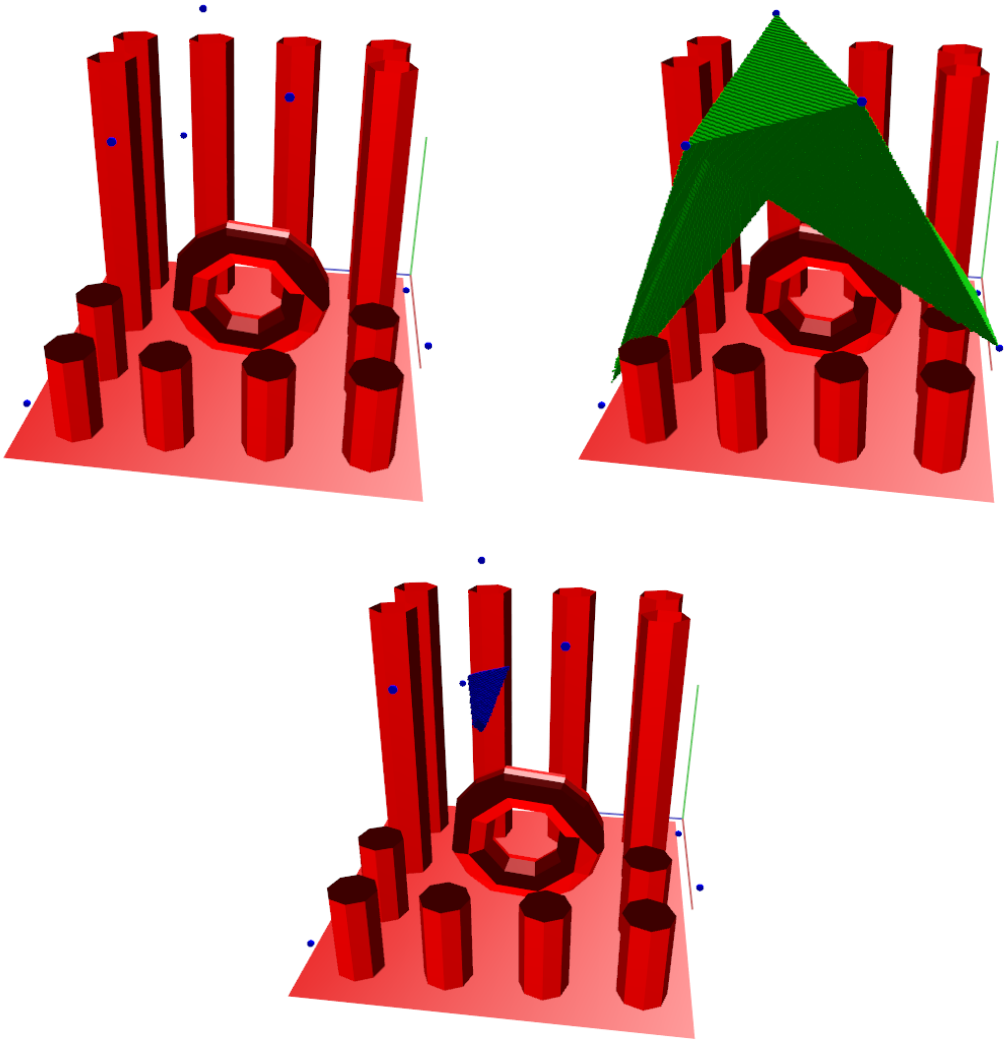


Figure 5.12: *3D good-visibility map computed from a set of 7 viewpoints and the scene represented by the red triangles. Top-left image shows only the scene and the set of viewpoints while the other pictures show* $gvr_{V,S}(1)$ *and* $gvr_{V,S}(2)$*, respectively.*

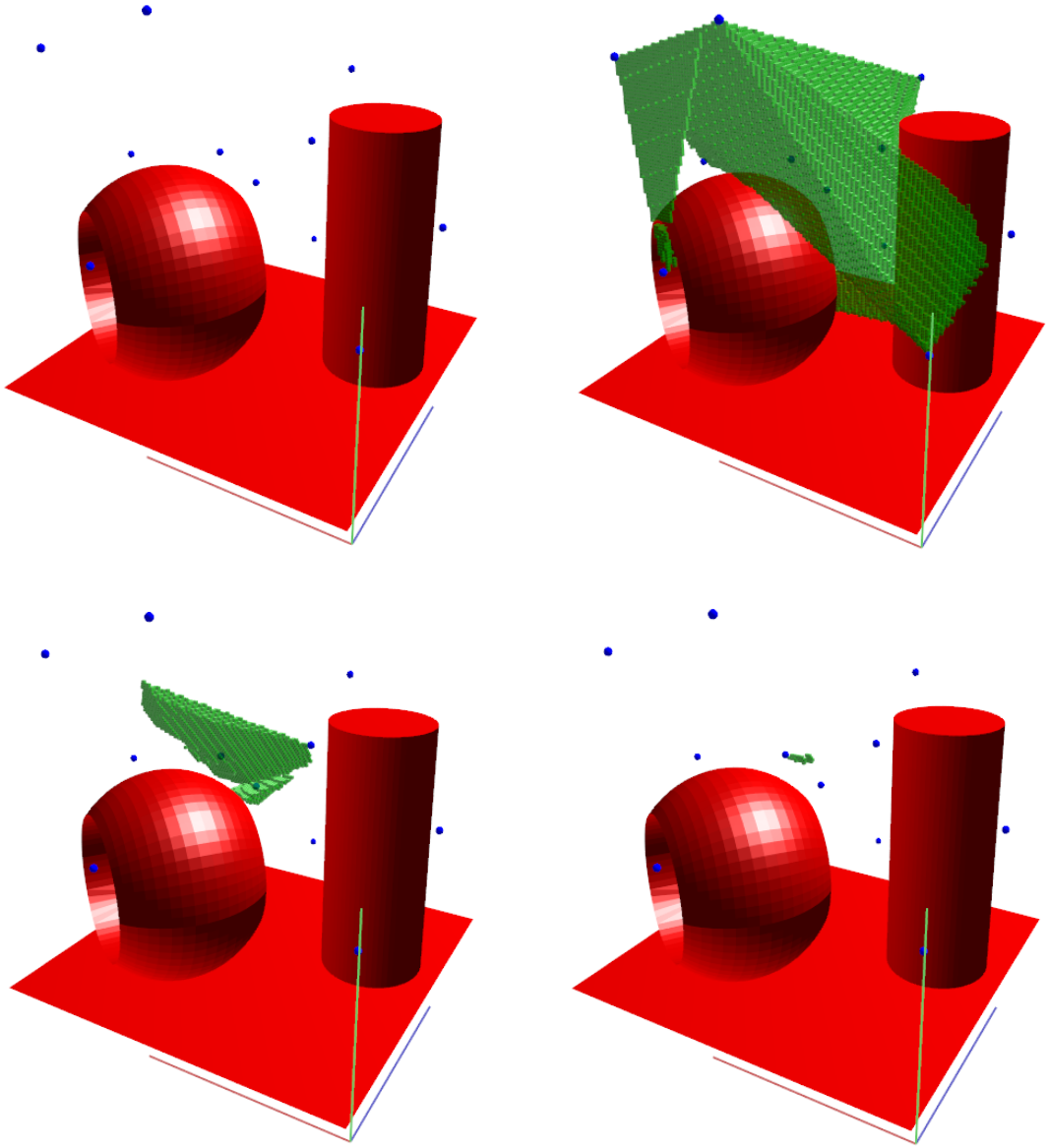


Figure 5.13: *The top-left image contains the scene from which the good-visibility map is computed. The other three images show, one by one the good-visibility regions* $gvr_{V,S}(1)$*,* $gvr_{V,S}(2)$ *and* $gvr_{V,S}(3)$*, respectively. Notice that a slightly transparent effect is applied to the visualization.*

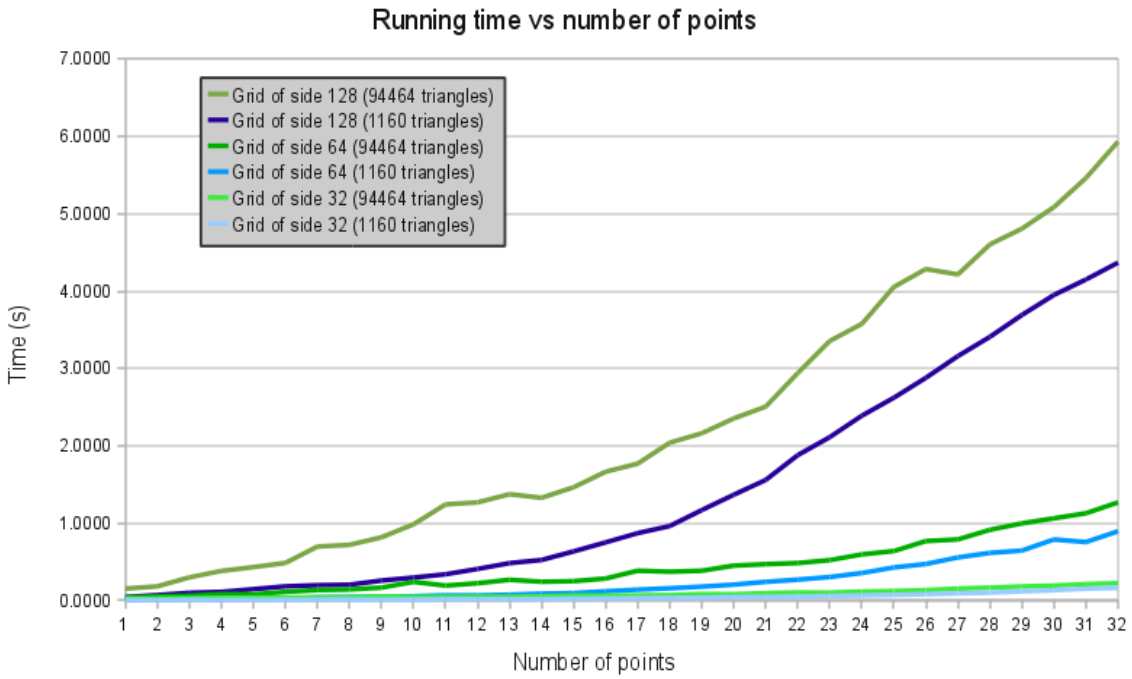


Figure 5.14: *Running times for the computation of the 3D good-visibility map when the number of viewpoints and grid size increase. These plots compare the times for the same scene using 1000 and 100000 triangles to represent it, respectively.*

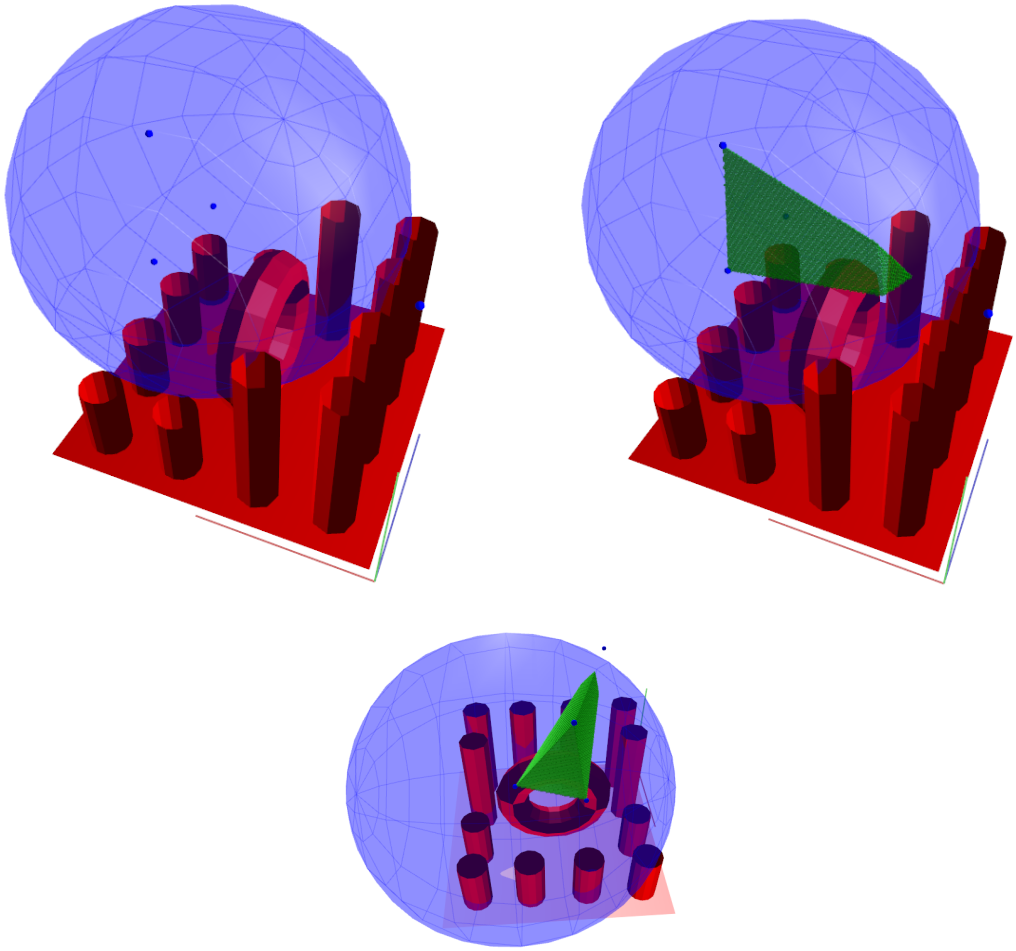


Figure 5.15: *The top-left image shows the scene from which the good-visibility map is computed. Notice that one of the viewpoints uses restricted visibility. The result is shown using different points of view in the other images.*

Chapter 6

Conclusions and final remarks

In this thesis we solved multi-visibility and good-visibility problems using the graphics hardware.

We first presented the design and algorithms needed to compute visibility and multi-visibility maps in the plane, on a terrain and even in the space. For the three-dimensional visibility we used the CUDA language provided by NVIDIA as a general purpose programming language for the GPU. In contrast, for the two-dimensional case and for computing multi-visibility maps on a terrain we proposed methods running also in the GPU, but we used Cg language and a computer graphics API to run the calculations. The multi-visibility map computation is presented for some different cases using visiblity variations: from a simple set of view points and a set of segment obstacles in the plane, to view points using restricted visibility on a terrain. We also thoroughly described how to compute the multi-visibility map from a set of view segments in the plane instead of view points or how the multi-visibility map is affected by using a binary image instead of a set of obstacles.

Multi-visibility computation is an essential step for computing the good-visibility map from a set of view objects and a set of obstacles. The good-visibility map can be obtained by combining the multi-visibility map together with the location depth and depth map concepts. We first provided an exact algorithm for computing the good-visibility map from a set of view objects and a set of segment obstacles in the plane. In addition, we presented a theoretical study where we give proof of its computational cost in time. Afterwards, we presented an algorithm for calculating a discretization of the good-visibility map which runs in the GPU and has the same complexity as the exact algorithm. By adding some variations to the latter method we obtained another algorithm, which also runs in the GPU, that significantly reduces the time complexity and provides a great reduction of the actual running time. The good-visibility map computation on a terrain is based on the

previously mentioned method. In this particular case, the terrain faces play the role of obstacles. The algorithm for obtaining good-visibility maps on terrains is very similar to the two-dimensional version. It is necessary to first compute the multi-visibility map in the terrain space and store it in a texture where each pixel codifies the visibility information of the corresponding projected point on the terrain. Afterwards, the view points (located over the terrain) must be projected to a two-dimensional plane by removing their height information. It is in this two-dimensional plane where the good-visibility map is computed taking into account the multi-visibility information obtained before. Finally the solution is re-projected to the faces of the terrain to visualize the good-visibility map.

Since there do not exist any algorithms for computing and visualizing depth contours from a set of points in a three-dimensional space, we provided a method that computes and visualizes them in an efficient way using a parallel algorithm implemented using CUDA that runs in the GPU. Basically, the method computes, as a pre-process step, the level of every plane generated by all the points in the set of view points. Then the depth contour of level k can be visualized by intersecting the halfspaces generated by the planes with level k. In order to compute and visualize a depth contour in a robust and fast way, we use the free tool QHull, which computes the intersection of a set of halfspaces in any dimension using the quick-hull method. We also showed how to use the latter algorithm to compute and visualize the bagplot from a set of points.

Finally, we presented an algorithm for computing the good-visibility map in a three-dimensional space from a set of view points and a set of triangle obstacles. As in the previously explained method for obtaining the depth contours, it first executes a pre-process step to compute the level of every plane generated by the set of view points. Afterwards, the space where we want to compute the good-visibility map is discretized into small portions, and a CUDA-based algorithm is responsible for computing the good-visibility depth of all such portions in a parallel fashion. Inside this parallel process, the multi-visibility information is also computed in order to finally obtain the good-visibility map. In contrast to the depth contours solution, an intersection of halfspaces to visualize a good-visibility region cannot be used. This is mainly due to the fact that good-visibility regions can be composed by non-convex polyhedra. In order to avoid this problem, we used the voxelization of the space for visualizing a good-visibility region as the boundary of a volume.

6.1 Final remarks

Our experimental results demonstrate that, although all the presented algorithms have a high time complexity, the fact of using the GPU to accelerate the calculations reduces the time in a constant value MP, depending on the number of multiprocessors and processors of the used GPU. Since this constant reduction is significant enough we have taken it into consideration in the time complexity formulas we presented. Depending on the number of concurrent threads involved in the computation with respect to the size of the whole problem, it is possible that this time reduction has a linear behaviour.

As a final remark, we have to say that since the GPUs are in constant evolution, every new graphics card that hits the market is more powerful than previous ones, thus the running time is reduced a lot every time a new generation of GPUs appears. Their price and parallel computation possibilities makes them a fantastic tool for improving running times for a lot of algorithms, including the ones presented in this thesis.

Bibliography

[AAG+85] T. Asano, T. Asano, L.J. Guibas, J. Hershberger, and H. Imai. Visibility-polygon search and euclidean shortest paths. In *FOCS*, pages 155–164, 1985.

[ABHM05] M. Abellanas, A. Bajuelos, G. Hernández, and I. Matos. Good illumination with limited visibility. In Wiley-VCH Verlag, editor, *Proc. International Conference of Numerical Analysis and Applied Mathematics*, pages 35–38, 2005.

[ABM07a] M. Abellanas, A. L. Bajuelos, and I. Matos. Good θ-illumination of points. In *Proc. 23rd European Workshop on Computational Geometry*, pages 61–64, 2007.

[ABM07b] M. Abellanas, A.L. Bajuelos, and I. Matos. Some problems related to good illumination. In Springer Berlag, editor, *Proc. of 7th Annual International Workshop on Computational Geometry and Applications (CGA'07), ICCSA 2007, Lecture Notes in Computer Science 4705*, pages 1–14, 2007.

[ACH04] M. Abellanas, S. Canales, and G. Hernández. Buena iluminación. In *Actas de las IV Jornadas de Matemática Discreta y Algorítmica*, pages 239–246, 2004.

[AFMFP+09] O. Aichholzer, R. Fabila-Monroy, D. Flores-Peñaloza, T. Hackl, C. Huemer, J. Urrutia, and B. Vogtenhuber. Modem illumination of monotone polygons. In *Proc.* 25^{th} *European Workshop on Computational Geometry EuroCG '09*, pages 167–170, 2009.

[AM01] T. Akenine-Moller. Fast 3d triangle-box overlap testing. *Journal of Graphics Tools*, 6:29–33, 2001.

[ASW08] P.K. Agarwal, M. Sharir, and E. Welzl. Algorithms for center and tverberg points. *ACM Trans. Algorithms*, 5(1):1–20, 2008.

[AW87] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987.

[Bal95] I.J. Balaban. An optimal algorithm for finding segments intersections. In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 211–219, New York, NY, USA, 1995. ACM.

[BDH96] C.B. Barber, D.P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE*, 22(4):469–483, 1996.

[BIK+04] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint. Space-efficient planar convex hull algorithms. *Theor. Comput. Sci.*, 321(1):25–40, 2004.

[BMCK08] B. Ben-Moshe, P. Carmi, and M.J. Katz. Approximating the visible region of a point on a terrain. *GeoInformatica*, 12(1):21–36, 2008.

[BO79] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, 28(9):643–647, 1979.

[BW03] J. Bittner and P. Wonka. Visibility in computer graphics. *Environment and Planning B: Planning and Design*, 30(5):729–756, September 2003.

[Can04] S. Canales. *Métodos heurísticos en problemas geométricos, Visibilidad, iluminación y vigilancia*. PhD thesis, Universidad Politécnica de Madrid, 2004.

[Cat75] E.E. Catmull. Computer display of curved surfaces. In *IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pages 11–17, 1975.

[CE92] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, 1992.

[CF89] N. Chin and S. Feiner. Near real-time shadow generation using bsp trees. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 99–106, New York, NY, USA, 1989. ACM.

[CFMS06] N. Coll, M. Fort, N. Madern, and J.A. Sellarès. Computing terrain multivisibility maps for a set of view segments using graphics hardware. In *ICCSA (1)*, pages 81–90, 2006.

[CFMS07a] N. Coll, M. Fort, N. Madern, and J.A. Sellarès. Good illumination maps. In *23th European Workshop on Computational Geometry*, pages 65–68, 2007.

[CFMS07b] N. Coll, M. Fort, N. Madern, and J.A. Sellarès. Gpu-based good illumination maps visualization. In *Actas XII Encuentros de Geometría Computacional*, pages 95–102, 2007.

[Cha03] T.M. Chan. A minimalist's implementation of the 3-d divide-and-conquer convex hull algorithm. Technical report, 2003.

[Chv75] V. Chvatal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory Series B*, 18:39–41, 1975.

[CMS08a] N. Coll, N. Madern, and J.A. Sellarès. Drawing good-visibility maps with graphics hardware. In *Proc. of Computer Graphics International Conference (CGI)*, pages 286–293, 2008.

[CMS08b] N. Coll, N. Madern, and J.A. Sellarès. Good visibility maps on polyhedral terrains. In *24th European Workshop on Computational Geometry*, pages 237–240, 2008.

[CMS10] N. Coll, N. Madern, and J.A. Sellarès. Good-visibility maps visualization. *Visual Computer*, 26(2):109–120, 2010.

[COCSD03] D. Cohen-Or, Y. Chrysanthou, C.T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Trans. Vis. Comput. Graph.*, 9(3):412–431, 2003.

[CT08] D. Cederman and P. Tsigas. A practical quicksort algorithm for graphics processors. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, pages 246–258, Berlin, Heidelberg, 2008. Springer-Verlag.

[dBvKvOO97] M. de Berg, M.J. van Kreveld, R. van Oostrum, and M. Overmars. *Computational Geometry: algorithms and applications*. Springer, 1997.

[DC08] Y. Okamoto T. Uno D. Christ, M. Hoffmann. Improved bounds for wireless localization. In *SWAT '08: Proc. 11th Scandinavian workshop on Algorithm Theory*, pages 77–89. Springer-Verlag, 2008.

[Den03] M.O. Denny. Algorithmic geometry via graphics hardware, 2003.

[Dur00] F. Durand. A multidisciplinary survey of visibility, 2000.

[FG06] I. Fisher and C. Gotsman. Drawing depth contours with graphics hardware. In *Proc. Canadian Conference Computational Geometry*, Kingston, Ontario, 2006.

[FH95] U. Finke and K.H. Hinrichs. Overlaying simply connected planar subdivisions in linear time. In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 119–126, New York, NY, USA, 1995. ACM.

[FHT09] J. Fishman, H. Haverkort, and L. Toma. Improved visibility computation on massive grid terrains. In *GIS '09: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 121–130, New York, NY, USA, 2009. ACM.

[FK03] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional; Pap/Cdr edition, 2003.

[FM94] L. De Floriani and P. Magillo. Visibility algorithms on triangulated digital terrain models. *International Journal of Geographical Information Systems*, 8(1):13–41, 1994.

[GA81] H. El Gindy and D. Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, (2):186–197, 1981.

[Gha98] S. Ghali. Computation and maintenance of visibility and shadows in the plane. In *Proc. 6th Int. Conf. Computer Graphics & Visualization*, pages 117–124, Feb 1998.

[Gra72] R.L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.

[GS96a] S. Ghali and A.J. Stewart. Incremental update of the visibility map as seen by a moving viewpoint in two dimensions. In *Proceedings of the Eurographics workshop on Computer animation and simulation '96*, pages 3–13, New York, NY, USA, 1996. Springer-Verlag New York, Inc.

[GS96b] S. Ghali and A.J. Stewart. Maintenance of the set of segments visible from a moving viewpoint in two dimensions. In *SCG '96: Proceedings of the twelfth annual symposium on Computational geometry*, pages 503–504, New York, NY, USA, 1996. ACM Press.

[Her89] J. Hershberger. Finding the upper envelope of n line segments in o(n log n) time. *Inf. Process. Lett.*, 33(4):169–174, 1989.

[HKBv97] V. Havran, T. Kopal, J. Bittner, and J. Žára. Fast robust bsp tree traversal algorithm for ray tracing. *J. Graph. Tools*, 2(4):15–23, 1997.

[HSHH07] D.R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[HTZ09] H. Haverkort, L. Toma, and Y. Zhuang. Computing visibility on terrains in external memory. *J. Exp. Algorithmics*, 13:1.5–1.23, 2009.

[IWW01] C. Benthin I. Wald, P. Slusallek and M. Wagner. Interactive rendering with coherent raytracing. In *Computer Graphics Forum (Proceedings of Eurographics)*, volume 3, pages 153–164, 2001.

[Kil99] M.J. Kilgard. Improving shadows and reflections via the stencil buffer. Technical report, NVIDIA Corporation, 1999.

[KMV02] S. Krishnan, N.H. Mustafa, and S. Venkatasubramanian. Hardware-assisted computation of depth contours. In *SODA '02: Proc. thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 558–567, 2002.

[KOS92] M.J. Katz, M.H. Overmars, and M. Sharir. Efficient hidden surface removal for objects with small union size. *Comput. Geom. Theory Appl.*, (2):223–234, 1992.

[KS97] K.S. Klimaszewski and T.W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, 17:42–51, 1997.

[KS09] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 23–28, New York, NY, USA, 2009. ACM.

[KSP+03] Miller K., Ramaswami S., Rousseeuw P., Sellarès J.A., Souvaine D., Streinu I., and Struyf A. Efficient computation of location depth contours by methods of computational geometry. *Journal of Statistics and Computing*, (13):153–162, 2003.

[Mah05] J.A. Mahovsky. *Ray tracing with reduced-precision bounding volume hierarchies.* PhD thesis, Calgary, Alta., Canada, Canada, 2005.

[Meg83] N. Megiddo. Linear-time algorithms for linear programming in r^3 and related problems. *SIAM Journal on Computing*, 12(4):759–776, 1983.

[MKV06] N. Mustafa, S. Krishnan, and S. Venkatasubramanian. Statistical data depth and the graphics hardware. In Regina Liu, Robert Serfling, and Diane Souvaine, editors, *Data Depth: Robust Multivariate Analysis, Computational Geometry and Applications*, volume 72, pages 223–246. AMS, 2006.

[MRR+03] K. Miller, S. Ramaswami, P. Rousseeuw, J.A. Sellarès, D. Souvaine, I. Streinu, and A. Struyf. Fast implementation of depth contours using topological sweep. In *Statistics and Computing*, pages 153–162, 2003.

[MS87] H. Mairson and J. Stolfi. Reporting and counting line segment intersections. In *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *Proceedings of the NATO Advanced Science Institute, Series F*, pages 305–325. Springer, July 1987.

[NBGS08] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[NO97] K. Nakamaru and Y. Ohno. Breadth-first ray tracing utilizing uniform spatial subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 3:316–328, 1997.

[NT] K. Nechvíle and P. Tobola. Local approach to dynamic visibility in the plane. In *Proc. 7th Int. Conf. Computer Graphics, Visualization and Interactive Digital Media (WSCG '99).*

[Nta92] S. Ntafos. Watchman routes under limited visibility. *Comput. Geom. Theory Appl.*, 1(3):149–170, 1992.

[PBMH02] T.J. Purcell, I. Buck, W.R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, 2002.

[PGSS07] S. Popov, J. Günther, H. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

[PMS+99] S. Parker, W. Martin, P.J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 119–126, New York, NY, USA, 1999. ACM.

[Rad46] R. Rado. A theorem on general measure. *J. London Math. Soc.*, (21):291–300, 1946.

[RS04] P.J. Rousseeuw and A. Struyf. Computation of robust statistics: depth, median, and related measures. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 57, pages 1279–1292. CRC Press LLC, 2004.

[SA08] E. Sintorn and U. Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.

[Sha78] M.I. Shamos. *Computational geometry.* PhD thesis, Yale University, 1978.

[She92] T. Shermer. Recent results in art galleries. In *Proceedings of the IEEE*, 1992.

[SKV06] N.H. Mustafa S. Krishnan and S. Venkatasubramanian. Statistical data depth and the graphics hardware. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 223–246. American Mathematical Society, 2006.

[SO86] S. Suri and J. O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 14–23, New York, NY, USA, 1986. ACM Press.

[SS92] K. Sung and P. Shirley. Ray tracing with the bsp tree. pages 271–274, 1992.

[SSS74] I.E. Sutherland, R.F. Sproull, and R.A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, (6):1–55, 1974.

[STK08] A. Schmitz, M. Tavenrath, and L. Kobbelt. Interactive global illumination for deformable geometry in cuda. *Comput. Graph. Forum*, 27(7):1979–1986, 2008.

[SVNB99] C. Saona-Vázquez, I. Navazo, and P. Brunet. The visibility octree. a data structure for 3d navigation. *Computers & Graphics*, 23:635–643, 1999.

[vKNRW97a] M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer. Digital elevation models and tin algorithms. *Lecture Notes in Computer Science (tutorials)*, (1340):37–78, 1997.

[vKNRW97b] M.J. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors. *Algorithmic Foundations of Geographic Information Systems, this book originated from the CISM Advanced School on the Algorithmic Foundations of Geographic Information Systems, Udine, Italy, September 16-20, 1996*, volume 1340 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[WBS07] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007.

[Web] The QHull Webpage. http://www.qhull.org.

[WS01] I. Wald and P. Slusallek. State of the art in interactive ray tracing, 2001.

[WSBW01] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 277–288, London, UK, 2001. Springer-Verlag.

[YW08] Y. Tseng Y. Wang, C. Hu. Efficient placement and dispatch of sensors in a wireless sensor network. *IEEE Transactions on Mobile Computing*, 7(2):262–274, 2008.

[ZH10] M. Zlatuska and V. Havran. Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms. In *Proceedings of WSCG'2010, communication papers*, pages 69–76, Feb 2010.