Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Čestmír Houška

Efficient visibility calculation for light transport simulation in participating media

Department of Software and Computer Science Education

Supervisor of the master thesis: doc. Ing. Jaroslav Křivánek, Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague 2013

I would like to thank my supervisor, Jaroslav Křivánek, for his invaluable feedback and advice.

I would also like to thank Mr. Wenzel Jakob for creating and maintaining an easy-to-use and open-source renderer Mitsuba.[20]

Název práce: Efektivní výpočet viditelnosti pro simulaci přenosu světla v opticky aktivních médiích

Autor: Čestmír Houška

Katedra / Ústav: Kabinet software a výuky informatiky

Vedoucí diplomové práce: doc. Ing. Jaroslav Křivánek, Ph.D.

Abstrakt: Tato práce zkoumá použití metod pro urychlení dotazů na viditelnost v algoritmech pro výpočet přenosu světla, přičemž je kladen důraz na konzervativnost a nízkou režii urychleného dotazu. Je zde prezentováno několik publikovaných nesměrových i směrových metod, využívajících pole vzdáleností (distance field), spolu s popisem jejich vlastností. Dvě z těchto metod jsou pak implementovány a důkladně otestovány v existujícím vykreslovacím systému na integrátoru, sledujícím světelné cesty (path tracer), i na vlastní implementaci krokujícího (ray marching) integrátoru pro výpočet jednou rozptýleného světla. Je zde také navržena, implementována a otestována metoda, která dále urychluje původní metody, založené na polích vzdáleností, tím, že předem uloží výsledky některých dotazů. Dále je nastíněno několik možných rozšíření této metody.

Klíčová slova: počítačová grafika, vykreslování, opticky aktivní média, viditelnost

Title: Efficient visibility calculation for light transport simulation in participating media

Author: Čestmír Houška

Department / Institute: Department of Software and Computer Science Education

Supervisor of the master thesis: doc. Ing. Jaroslav Křivánek, Ph.D.

Abstract: This thesis investigates the use of acceleration methods for the testing of visibility in light transport calculation algorithms with the emphasis on conservativeness and low accelerated query overhead. Several published non-directional and directional distance field methods are presented with the description of their characteristic properties. Two of these methods are then implemented and thoroughly tested in an existing rendering framework on a path tracing volumetric integrator as well as on an own implementation of a ray marching single scattering integrator. A method that further accelerates the original distance field methods by pre-caching results of some of the queries is also proposed, implemented and tested. Furthermore, several possible extensions to this method are outlined.

Keywords: computer graphics, rendering, participating media, visibility

Contents

1. Introduction

Image rendering algorithms nowadays can be divided into two groups. First group consists of real-time rendering techniques that strive to render the image in time frames of fractions of a second, which is enabled by the use of a variety of tricks and simplifications. The second group of algorithms, on the other hand, tries to simulate the transport of light between objects in the scene as realistically as possible.

A prominent example of algorithms from the second group are path tracing algorithms that construct various possible paths of the light's particles from light emitters to the image sensor. When the scene or a part of it are submerged in a medium that interacts with the light's particles, the light path vertices can be located in the medium as well as on an object's surface.

The construction of the light paths requires a lot of visibility queries between two points in space and the execution of these queries takes a large portion of the rendering time, especially for scenes with complex and detailed geometry, which is often the case when trying to render as realistic images as possible. The goal of this thesis is to lower the rendering times by accelerating these visibility queries.

The fundamental idea behind this thesis is the fact that in order to accelerate a visibility query from point `A` to point `B`, we don't have to know the exact shape of the scene's geometry. A knowledge of the closest point on the scene's geometry for both points will be sufficient. If the distance between `A` and `B` is smaller than the closest point on the geometry for either of the points, we know that the segment between these two points is not occluded by any part of the geometry.

The data structures that contain this closest point information are called distance functions or distance fields. I studied several methods for implementing the distance fields, both undirected and directed (where the distance to the closest point varies with the direction of the query) and considered their applicability to my problem in the light of two main criteria. First, I needed the methods to be conservative so that no ray intersection would be skipped by the acceleration. Second, I needed the acceleration to be as fast as possible, to be able to compete with a well-written and optimized k-d tree structure that is traditionally used to test visibility and find ray intersections.

After initial research of several distance field methods, I chose some of these methods and implemented them in an existing renderer in order to test their usability for visibility query acceleration.

1.1 Structure of the Text

The rest of this text is organized as follows.

The first chapter, which you are currently reading, is an introduction to the thesis.

Follows the second chapter, which contains the necessary theoretical background to the thesis, describing the light transport equations and renderers that use them to estimate the incoming light intensity. This chapter also defines a visibility query and discusses, which parts of the described renderers contain visibility queries and which of these queries can be accelerated. A discussion on the acceleration potential of general visibility query acceleration algorithms concludes the chapter.

The third chapter defines and describes distance fields and distance functions and the notions related to them. The details of non-directional distance field methods[†], like metrics and initialization, are also described here.

The fourth chapter builds upon the third chapter by analyzing three directional distance field methods that could be used for visibility query acceleration. My own method is then introduced that improves the acceleration by caching the result of the distance field visibility queries for the most frequently queried voxels. An extension to the method is then proposed that could accelerate a higher percentage of visibility queries than it does by caching distance field visibility queries. A possible method for accelerating visibility queries to point lights using this method is then outlined.

The implementation details are discussed in the fifth chapter. In that chapter, I also write about the technology that was used to test the distance field acceleration methods, talk about the debugging features that I created for the implementation and then provide some details regarding the compilation and running of the Mitsuba renderer with my changes.

The sixth chapter then describes the testing conditions and contains tables with the results from testing of the implemented distance field methods. These results are then discussed and interpreted.

The final chapter then concludes the text with a summary of the whole thesis and a proposal of possible areas of further research.

[†]although many of them are used in the directional methods in the same manner.

2. Theoretical Background

Generating images from scenes created with the help of computers is a complex task that was originally solved only by empirical shading or ray tracing algorithms. Although these algorithms might have been perceived as diverse in the techniques they employed, all of them strived to simulate the same physical phenomenon – perception or detection of light particles that arrive to the observer's eye or camera after being emitted by a light source and possibly after several interactions with the scene's geometry and/or participating media.

The theoretical construct that united all these algorithms was James T. Kajiya's rendering equation [10], which is an integral equation that mathematically describes the light transport equilibrium in a scene. Apart from the rendering equation, Kajiya also developed a Monte Carlo technique for estimating the intensity of light traveling through a certain point in space in a certain direction[†]. This method was named *path tracing* and for its simplicity and properties[‡] continues to be used in many rendering frameworks.

2.1 Volumetric Rendering Equation

The rendering equation was later generalized to account for interaction of light with participating media. This generalization was described for example by James Arvo in [2]. Aside from taking into consideration the light bouncing off of scene's surfaces, the generalization mathematically describes absorption and scattering of photons on particles of the participating medium as well as light emission from the medium's particles.

Let us look at the quantities used in the (slightly re-written) equation:

$$L(x,\vec{\omega}) = Tr(x' \leftrightarrow x)L_{surf}(x',\vec{\omega}) + \int_0^{|x-x'|} Tr(x \leftrightarrow (x-\vec{\omega}l))L_{in}(x-\vec{\omega}l,\vec{\omega})\mathrm{d}l \tag{2.1}$$

In the above equation:

x'	is the first surface that is visible from x in the direction $-\vec{\omega}$.
$L(x,\vec{\omega})$	is the radiance passing through point x in the direction $\vec{\omega}$.
$Tr(x' \leftrightarrow x)$	is the so-called *transmittance function*, whose values range in $[0,1]$ and which accounts for diminishing of the light's intensity due to out-scattering and attenuation in the medium.
$L_{surf}(x,\vec{\omega})$	is radiance coming from a surface at x in the direction $\vec{\omega}$.
$L_{in}(x,\vec{\omega})$	is radiance that gets scattered via interaction with the medium to direction $\vec{\omega}$ at x from all directions.

[†]This physical quantity is called luminosity, or radiosity, depending on whether the reaction of human visual system to the radiation is taken into account or not.

[‡]For example, it is an unbiased and consistent estimator of the radiance passing through a given point in space in a given direction.

Equation 2.1 is an integral equation, because both L_{surf} and L_{in} contain L:

$$L_{surf}(x,\vec{\omega}) = L_e(x,\vec{\omega}) + L_r(x,\vec{\omega}) \tag{2.2}$$

$$L_{in}(x,\vec{\omega}) = \sigma_s(x) \cdot \int_{\vec{\omega}_i \in \mathcal{S}} p_x(\vec{\omega},\vec{\omega}_i) \cdot L(x,-\vec{\omega}_i) \cdot \mathrm{d}\omega_i \tag{2.3}$$

$$L_r(x,\vec{\omega}) = \int_{\vec{\omega}_i \in \mathcal{H}} f_x(loc(\vec{\omega},x),\vec{\omega}_i) \cdot L(x,-\vec{\omega}_i) \cdot (\vec{\omega}_i \cdot \vec{n}(x)) \cdot \mathrm{d}\vec{\omega}_i \tag{2.4}$$

In the previous equations:

$L_e(x,\vec{\omega})$	is the light emitted by the surface at point x in direction $\vec{\omega}$.
$L_r(x,\vec{\omega})$	is the light reflected by the surface from other directions into direction $\vec{\omega}$ at point x.
$\sigma_s(x)$	is the scattering coefficient of the medium in point x. This coefficient tells us the proportion of light that gets scattered per unit of length.
$\mathcal{S}$	is a sphere that contains all unit vectors.
$p_x(\vec{\omega}_o,\vec{\omega}_i)$	is the scattering phase function[†]in point x.
$\mathcal{H}$	is a positive hemisphere (i.e. it contains all unit vectors that would be above a surface in that surface's local tangent coordinate system).
$f_x(\vec{\omega}_o,\vec{\omega}_i)$	is the bi-directional reflection distribution function[‡]in point x.
$loc(\vec{\omega},x)$	is an operator that converts the vector $\vec{\omega}$ into the local tangent coordinate system in point x (see Figure 2.1).
$\vec{n}(x)$	is the normal vector to the surface in point x.
$\vec{\omega}_i \cdot \vec{n}(x)$	is actually the cosine of the elevation angle of $loc(\vec{\omega}_i,x)$. This factor accounts for the reduction in radiance due to Lambert's cosine law.

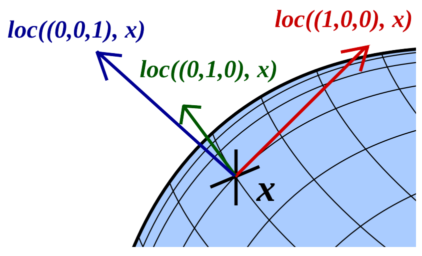

Figure 2.1 – Local tangent coordinate system in point x on the scene's geometry.

As can be seen from the Equation 2.1, in order to render physically correct images, one just has to evaluate the integral in the equation for a ray (or several

[‡]This function gives us the probability density of the event that photon bouncing of the surface at x from the direction $\vec{\omega}_i$ will be reflected in the direction $\vec{\omega}_o$. More on this function can be found in Matt Pharr's and Greg Humphreys' book on physically-based rendering [12, pg. 583]

[‡]This function returns the probability density of the event that a photon traveling from the direction $\vec{\omega}_i$ and interacting with the medium in x will be scattered into the direction $\vec{\omega}_o$. Again, the function is described more in-depth in [12, Pharr, Humphreys, pg. 294].

rays) that is cast through every pixel in the image plane of the camera. Unfortunately, the evaluation is complicated by the fact that the function L appears in an integral on the right-hand side of the equation. This makes the Equation 2.1 an integral equation, which is generally not analytically solvable. However, numerical methods such as Monte Carlo integration exist to estimate the value of L. What every physically grounded renderer then tries to achieve is essentially to calculate such an estimate.

Single and Multiple Scattering

In media with high scattering coefficient $\sigma_s(x)$, the probability of light interacting with the medium and scattering away from the original ray direction gets higher. Often, the light particles bounce in the medium multiple times before arriving to the observer. This effect, known as *multiple scattering*, is very difficult and expensive in terms of computation time and many renderers choose not to simulate it.

Single scattering, on the other hand, is the situation, when the light changes its path in the medium only once. Often, when only single scattering is computed, the light that is allowed to scatter is limited to the emitted light, which is much more simple to evaluate. In that case, the in-scattered radiance is calculated by $L^*(x,\vec{\omega})$, which is a simplification of the Equation 2.1 that takes into consideration only the emitted radiance and ignores the in-scattered light:

$$L^*(x,\vec{\omega}) = Tr(x' \leftrightarrow x)L_e(x,\vec{\omega})$$

Substituting $L^*(x,\vec{\omega})$ into the Equation 2.3 yields L^*_{in} for computation of single-scattered radiance:

$$L^*_{in}(x,\vec{\omega}) = \sigma_s(x) \cdot \int_{\vec{\omega}_i \in \mathcal{S}} p_x(\vec{\omega},\vec{\omega}_i) \cdot Tr(x' \leftrightarrow x)L_e(x,-\vec{\omega}_i) \cdot \mathrm{d}\omega_i$$

2.2 Light Emitters

For the needs of the subsequent discussion, we will need to describe the various light emitters that can appear in the scene. This is neither a complete list of all emitter types that can be simulated on a computer, nor a complete list of all emitter types implemented in Mitsuba. The purpose of this section is to get the reader acquainted with all the light emitter types that are relevant to my work.

Point Lights

As the name suggests, a point light is a light emitter, which emits radiance from a single point in space evenly into all directions. The emitter is not associated with any geometry, so any ray casts will miss it. In order to sample light from this light source, the renderer has to explicitly connect light paths to the light's position.

The fact that the emitter does not have to originate from a scene's surface is important for us, because the renderers that I used in this work query visibility between the connected point on a light path and the sampled position on the light source (in this case the point, where the point light source is positioned). If the light was positioned on a surface, the visibility query would not be accelerable using a distance field. This is the main reason why I use point lights almost exclusively in the testing scenes.

Area Lights

An area light is a light source that emits light from a patch of surface geometry. Usually, the radiance is uniform across the patch and across the possible emission directions. It can be intersected by ray casting and it can also be sampled for a point on the light's geometry[†].

The visibility tests between an arbitrary point in space and a sampled point located on the light's geometry cannot be accelerated by the distance field methods that I currently use. Nevertheless, a method for accelerating visibility queries on area lights is proposed in the Section 4.4.1.

Environment Maps

An environment map is a light emitter that can be used to simulate light coming from the distant surroundings of the scene. Usually, the light source of the environment light is regarded to be distant and the light emitted by an environment map is thus parametrized only by the outgoing direction of the ray that is used to sample the light.

However, in Mitsuba, an environment map is implemented as a sphere, which encompasses the whole scene and acts as an area emitter, with the difference that its geometry cannot be intersected. Instead, the intersection is calculated only when a ray is cast out of the scene. In addition to this, the environment map emitter can be sampled for a point in the same manner as an area light.

The fact that an environment light has its own geometry might convince us that visibility queries due to sampling of the environment lighting cannot be accelerated because the sampled point will be located on the scene's geometry. Fortunately, the geometry is only virtual and will never be registered in the distance field. We only have to ensure that the distance field is large enough to contain not only the scene, but the environment map as well.

2.3 Volumetric Path Tracing

The volumetric path tracing algorithm, a generalization of Kajiya's path tracing, is a Monte Carlo solver of the volumetric rendering equation. A good introduction into the theory behind Monte Carlo methods can be found in lecture materials by Stefan Weinzierl [19].

In this section, I will describe the volumetric path tracer as it appears in the Mitsuba renderer that I used for my work. An other volumetric path tracer implementation might differ in details, but its general structure will be the same.

[†]Because of this, a special care has to be taken so as not to count this emitter's radiance twice.

In the original path tracer from Mitsuba, the ray casting function, which finds the nearest intersection along a ray, is called several times, but in some cases, a simple information on whether the ray intersected the scene's geometry is needed. These cases are in fact visibility tests, because their purpose is to determine the visibility between two points in the scene. These parts of the code are of the greatest interest to this work, because our goal is to accelerate these queries.

Volumetric Path Tracer in Mitsuba

Note: In all of the pseudo-code descriptions in this thesis, the names of functions, classes, variables, etc. might be different than those in the original source code of Mitsuba. Also, the inner structure of the path tracer such as functions, recursion and other control structures, might be changed. This is for clarity reasons.

Excursion: Pseudo-code conventions

The pseudo-code used in this thesis is a combination of Python, C++ and Pascal. Keywords are typeset in a **bold** font, so that the structure of the code stands out better.

If a functions needs to access variables or data that are not its parameters or local variables, these variables are labeled by the keyword **global**.

Assignment operator is ':=' instead of simple '=', so as to avoid confusion with comparison operator '=='.

Sometimes, a function needs to modify one of its parameters. Such parameters are passed by reference and are marked as such by the ampersand before the parameter name in the function definition as well as the call:

function `test(param, &modifiableParam)`

    // Comments are preceded by two slashes and are typeset in grey color.

Every radiance integrator in Mitsuba gets as a parameter the ray and position, along which it should calculate the radiance. After the integrator finishes, a spectral value is returned, which represents the radiance along the ray that was passed as the parameter. Nevertheless, for the sake of simplicity of the pseudo-code, I will regard all radiance values as simple floating-point numbers.

Follows a detailed pseudo-code for the volumetric path tracer from Mitsuba[†].

```
// Returns the radiance in a given point along the given direction
function Li(point, dir):
  global scene

  // Try to find intersection point by ray casting:
  surfPoint := scene.intersect(point, dir)
  return LiRecursive(point, surfPoint, dir)
end function

function LiRecursive(point, surfPoint, dir):
```

[†]This is actually the simple version which does not use multiple importance sampling to reduce variance, but that is only a slight implementation detail.

```
global scene
retval = 0.0
pdfSuccess := 0.0
pdfFailure := 1.0
medPoint := NULL
isInMedium := inMedium(point, dir)

if (surfPoint == NULL) return retval
// Try to sample point in the medium and save the probability density
if (isInMedium) then
  medPoint := samplePointInMedium(point, dir, &pdfSuccess)
  pdfFailure := 1.0 - pdfSuccess
endif
// If the sampled point was before surfPoint, do medium interaction
if (isInMedium and |point - medPoint| < |point - surfPoint|) then
  tr1 := transmittance(point, medPoint)
  s := sigmaS(medPoint)

  // Direct illumination sampling
  value := pickAndSampleEmitter(medPoint, &sampledPoint, &pdfSample)
  if (visible(medPoint, sampledPoint)) then
    tr2 := transmittance(medPoint, sampledPoint)
    phase := phaseFunc(medPoint, -dir, norm(sampledPoint - medPoint))
    retval := retval +
      s * value * phase * tr1 * tr2 / (pdfSample * pdfSuccess)
  endif

  // Phase function sampling
  sampledDir1 := samplePhaseFunc(medPoint, &phase)
  surfPoint := scene.intersect(medPoint, sampledDir1)
  shouldContinue := russianRoulette(&rrProb)
  li := 0.0
  if (shouldContinue) then
    li := LiRecursive(medPoint, surfPoint, sampledDir1)
  endif
  return retval + s * li * phase * tr1 / (pdfSuccess * rrProb)
// If the sampled distance was further than the intersection
// or there is no medium, generate surface interaction
else
  // Evaluate transmittance
  tr3 := 1.0
  if (isInMedium) then
    tr3 := transmittance(point, surfPoint)
  endif

  // Direct illumination sampling
  value := pickAndSampleEmitter(surfPoint, &sampledPoint2, &pdfSample)
  sampledDir2 := norm(sampledPoint2 - surfPoint)
  if (visible(surfPoint, sampledPoint2)) then
    tr4 := 1.0
```

```
      if (inMedium(surfPoint, sampledDir2)) then
        tr4 := transmittance(surfPoint, sampledPoint2)
      endif
      bsdfVal := evalBsdf(surfPoint, -dir, sampledDir2)
      retval := retval +
        value * bsdfVal * tr3 * tr4 / (pdfSample * pdfFailure)
    endif

    // BSDF sampling
    bsdfVal := sampleBsdf(surfPoint, &sampledDir3)
    point := surfPoint
    surfPoint := scene.intersect(point, sampledDir3)
    shouldContinue := russianRoulette(&rrProb)
    li := 0.0
    if (shouldContinue) then
      li := LiRecursive(point, surfPoint, sampledDir3)
    endif
    return retval + li * bsdfVal * tr3 / (pdfFailure * rrProb)
  endif
end function
```

As we can see, the integrator first casts a ray from `point` in direction `dir` in order to find the next intersection with the scene – point `surfPoint`. Then, if the segment from `point` to `surfPoint` is in a participating medium (this happens when the ray from the starting point points into the medium, because a medium boundary is always regarded as a surface and as such would be detected and intersected by the ray-cast), a distance is sampled in the medium. This distance can be shorter than the distance from `point` to `surfPoint`, in which case L_{in} part of the Equation 2.1 is sampled. If the surface interaction is closer than the medium interaction, L_{surf} is sampled.

In both cases, the direct illumination is sampled by sampling the light emitters from `medPoint` and `surfPoint` and then recursion is applied to sample longer paths. Russian Roulette is used to ensure path termination while keeping the integral estimation consistent.

The described integrator is capable of evaluating contribution of light paths with an arbitrary length and consisting of arbitrary interaction types (surface or medium) along the way[†].

2.4 Visibility Test Classification

Definition 1. *Let $a \in \mathbb{R}^3$ and $b \in \mathbb{R}^3$ be two points and let $\Sigma \subseteq \mathbb{R}^3$ be a set of a scene's surface points. A **visibility test** (or query) on the line between two*

[†]Actually, as it was described here, the path tracer never calculates the contribution of paths from the eye/camera directly to a light source, but this is another implementation detail that I chose to omit for the sake of simplicity.

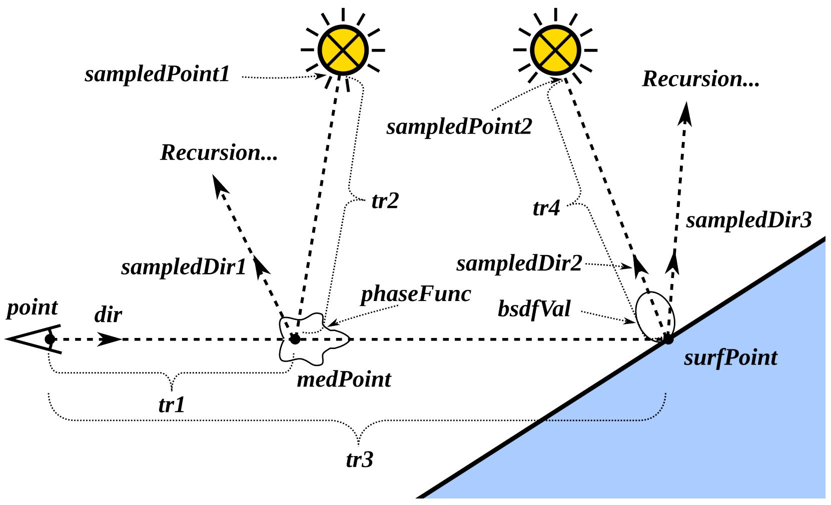

Figure 2.2 – Illustration of the variables used in the pseudo-code description of the volumetric path tracer.

points a and b is a boolean predicate defined in the following way:

$$vis(a,b) = \begin{cases} 1 & : \quad \{(1-t)a + tb : t \in [0,1]\} \cap \Sigma = \emptyset \\ 0 & : \quad \{(1-t)a + tb : t \in [0,1]\} \cap \Sigma \neq \emptyset \end{cases}$$

There are several visibility tests in the pseudo-code that was described above. Let us now look at them and discuss, which of them bear some acceleration potential.

One visibility test is on line 31, where the direct illumination is sampled in the medium. Here, visibility is tested between points `medPoint` and `sampledPoint1`. Another visibility test is on line 59 in direct illumination sampling on surfaces, testing visibility between `surfPoint` and `sampledPoint2`.

Possibly another visibility test would be after each medium sampling (line 21), if the ray-cast was postponed after the sampling. That way, the integrator could first sample the medium for the medium point, test visibility between that sampled point and the starting point and only after the test it would possibly cast a ray. Whether or not the ray would be cast would depend on the visibility test result. If the line between the two points was certainly unoccluded, the integrator would not have to proceed with the ray casting, because the intersection (if any) would certainly lie behind the sampled point in the medium.

The aforementioned acceleration would, however, need a refactoring of the volumetric path tracer and I chose to test viability of the acceleration techniques first before optimizing the original integrators in Mitsuba.

Definition 2. *If $a \in \mathbb{R}^3$ and $b \in \mathbb{R}^3$ are two points and $\Sigma \subseteq \mathbb{R}^3$ is a set of a scene's surface points, a visibility test $vis(a,b)$ is called* ***spatial****, if and only if $a \notin \Sigma \wedge b \notin \Sigma$. I will call non-spatial visibility tests* ***superficial****.*

If we want to use distance fields to accelerate the visibility queries, only spatial visibility queries are candidates for acceleration. Points on a surface would fall

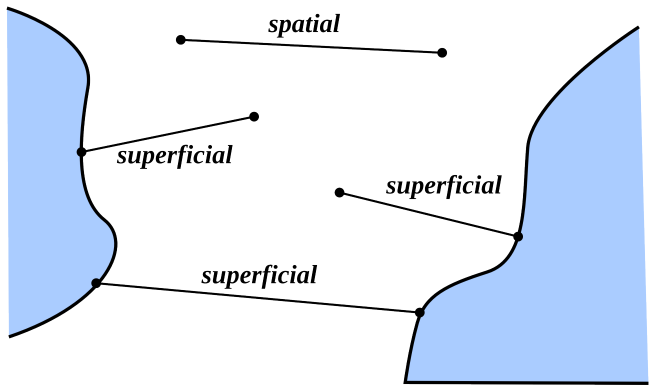

Figure 2.3 – The two types of visibility tests.

into a voxel from the set Σ' (see the Section 3.3) and the visibility query between them and other points would not be accelerated.

Looking back at the pseudo-code of the volumetric path tracer, we can now identify the visibility test categories in the code. Visibility test on the line 31 can be spatial or superficial, depending on the type of light emitter. Unfortunately, visibility test on the line 59 has one point on a surface of the scene, so it will always be superficial. If we re-factored the source code of the integrator and first sampled the distance in the medium before casting the intersection ray, we would possibly create another spatial visibility test, but only after the light path had an interaction with the medium (otherwise, the test would be superficial).

2.5 Volumetric Ray Marching

In order to increase the number of spatial visibility tests, but also to simulate conditions for which the visibility test acceleration was primarily intended, I implemented a single-scattering volumetric integrator that uses ray-marching in participating media to estimate the in-scattered radiation.

The integrator was created by copying the source code of the volumetric path tracer and changing the sampling strategy in the medium so that instead of choosing between sampling the radiance from a distant intersected surface and sampling the medium in-scattered radiance, both the surface radiance and the in-scattered medium radiance are estimated for every ray and summed together.

```
// Returns the radiance in a given point along the given direction
function Li(point, dir):
  global scene

  // Try to find intersection point by ray-casting:
  surfPoint := scene.intersect(point, dir)
  return LiRecursive(point, surfPoint, dir)
end function

function LiRecursive(point, surfPoint, dir):
  global scene
  retval := 0.0
  isInMedium := inMedium(point, dir)

  if (isInMedium) then
    // Calculate sigma_t of the medium as if it always were homogeneous
    transmittance := scene.evalTr(point, surfPoint)
    sigma_t := -log(transmittance) / |surfPoint - point|

    // Sample the single scattering according to transmittance
    intervalWidth = (1.0 - transmittance) / sampleNum
    sampleMin := 1.0
    sampleMax := 1.0
    repeat sampleNum times
      // Modify the current sampling interval
      sampleMax := sampleMin
      sampleMin := sampleMax - intervalWidth

      // Generate sample and calculate its ray parameter and pdf
      sample := uniform(sampleMin, sampleMax)
      t := -log(sample) / sigma_t
      pdf := sigma_t * exp(-sigma_t * t) / intervalWidth

      // Sample an emitter from the generated point
      medPoint := point + t * dir
      value := pickAndSampleEmitter(medPoint, &sampledPoint, &pdfSample)

      if (visible(medPoint, sampledPoint)) then
        // Calculate transmittance and retrieve the scattering coeff.
        // and the value of the phase function
        tr := transmittance(point, medPoint)
        s := sigmaS(medPoint)
        phase := phaseFunc(medPoint, -dir, norm(sampledPoint - medPoint))

        // Add the result to the in-scattered radiance estimate
        retval := retval + tr * s * phase * value / (pdf * pdfSample)
      endif
    end repeat
  endif

  // Only the scattered radiance will be returned when no surface
  // was intersected
  if (surfPoint == NULL) return retval

  // This part of code is omitted, because it calculates the radiance
  // reflected by a surface in the same way as the path tracer does
  // The code would copy path tracer lines 50 - 77
  return retval + li * bsdfVal * tr3 / rrProb
end function
```

The ray marcher begins in the same manner as the volumetric path tracer – it casts a ray from `point` in direction `dir` to see, whether a point on the geometry will be intersected. Regardless of the outcome of the ray-cast, if the current segment is in medium, the in-scattered radiance L^*_{in} along the segment is estimated using ray-marching. The surface radiance L_{surf} is then estimated in the same way as in the volumetric path tracer and added to the in-scattered radiance.

Ray marching divides the whole segment into separate parts, casts a sampling ray from a randomly generated point in each of the parts and sums the resulting radiances to estimate the radiance along the whole segment. This technique of subdivision of the domain of the estimated integral is called *stratified sampling* [19, S. Weinzierl, pg. 13] and serves to reduce the variance of the result. The number of the subdivisions is given as a parameter `sampleNum`.

To further reduce the variance, I use *importance sampling* [19, S. Weinzierl, pg. 14] to generate the ray marching samples. If the segment was divided into subsegments of the same length, the first segment would (most probably) contribute to the estimated radiance much more than the last, because of the transmittance term `tr` that would be almost zero for the sample in the first segment and large for the furthest samples. In fact, if the in-scattered radiance at all points were uniform, the contribution of all of the ray marching samples would be proportional to the transmittance.

It is therefore advantageous to sample the ray not uniformly, but taking the transmittance into account, so that the cumulative distribution function is proportional to the transmittance – roughly half of the samples should be positioned before a point on the ray, where the transmittance reaches a value of 0.5, three quarters before a point with transmittance of 0.25, etc…

Visibility Tests in Ray Marcher

The number of spatial visibility tests in the ray marcher can be potentially much higher than in the volumetric path tracer. This is due to large number of light samples placed in the medium. For example, if the whole scene was submerged in the participating medium, each level of recursion of the sampling ray would mean `sampleNum` spatial visibility tests due to light sampling in the medium and one superficial visibility test from the surface to a light source. With the increasing number of the ray marching samples, the percentage of accelerable visibility tests will approach 100%.

2.6 Acceleration in Theory

In the following chapters, I will thoroughly describe the methods that were examined for the acceleration of visibility queries. In order to evaluate or improve the methods, one needs to understand, which properties of these methods and the used rendering algorithms influence the resulting acceleration.

I will now assume the following acceleration scheme: for each spatial visibility query (as was mentioned in Section 2.4, only spatial visibility tests can be accelerated), first query an acceleration data structure that can speed the query up by quickly proving that the two points in the query are unoccluded. If, however, the answer is not conclusive, the standard test proceeds as it would without the acceleration structure.

In the following calculations, the probability of a given visibility query being spatial is denoted p_{sp}, whereas the probability that a certain spatial visibility test will be accelerated is denoted $p_{acc|sp}$. The resulting probability of a given visibility test being accelerated is denoted as $p_{acc} = p_{sp} \cdot p_{acc|sp}$ [†]. With the increasing number of rays shot into the scene, the measured ratios of the visibility tests types will converge to these probabilities.

We will now split the total time needed to render a scene without a visibility acceleration method T_{render} into the time needed for all N visibility tests ($T_{query} = N \cdot t_{query}$) and the rest of the render time T_{rest}. The time to render with an acceleration method is denoted by T^{acc}_{render} and can be split into time to build the acceleration structure T_{build}, time needed for visibility tests in those cases that are not accelerated (that happens $(1 - p_{acc}) \cdot N$ times out of N), time spent for querying the acceleration structure for spatial tests (this is only done for $p_{sp} \cdot N$ tests) and the remainder T_{rest}:

$$T_{render} = T_{query} + T_{rest} = N \cdot t_{query} + T_{rest} \tag{2.5}$$

$$\begin{aligned} T^{acc}_{render} &= T_{build} + T^{acc}_{query} + T_{rest} = \\ &= T_{build} + (1 - p_{acc}) \cdot N \cdot t_{query} + p_{sp} \cdot N \cdot t_{acc} + T_{rest} \end{aligned} \tag{2.6}$$

If the visibility queries are to be accelerated at all, then $T_{render} > T^{acc}_{render}$ must hold. By modifying this inequality, we get:

$$\begin{aligned} T_{render} &> T^{acc}_{render} && / - T_{rest} \\ N \cdot t_{query} &> T_{build} + (1 - p_{acc}) \cdot N \cdot t_{query} + p_{sp} \cdot N \cdot t_{acc} && / : N \\ t_{query} &> \frac{T_{build}}{N} + (1 - p_{acc}) \cdot t_{query} + p_{sp} \cdot t_{acc} && / - (1 - p_{acc}) \cdot t_{query} \\ p_{acc} \cdot t_{query} &> \frac{T_{build}}{N} + p_{sp} \cdot t_{acc} && / : p_{sp} \\ p_{acc|sp} \cdot t_{query} &> \frac{T_{build}}{N \cdot p_{sp}} + t_{acc} && \Big/ \frac{T_{build}}{N \cdot p_{sp}} \overset{def}{=} t_{build} \\ p_{acc|sp} \cdot t_{query} &> t_{build} + t_{acc} \end{aligned} \tag{2.7}$$

Here, I introduce t_{build} as the *amortized build time*. It is defined as $\frac{T_{build}}{N \cdot p_{sp}}$, which means that the total build time is divided between the spatial visibility tests.

In the Inequality 2.7 on the left side we have the time that is saved per a spatial visibility query, i.e. the time it would take to perform the query multiplied by the probability that the query will not have to be performed. On the right hand side of the inequality, we have the time that is added per each spatial visibility query, i.e. time to build the acceleration data structure amortized between the

[†]Note that $p_{acc|sp}$ is a conditional probability $P(\textit{test accelerated} \mid \textit{test spatial})$.

queries plus the time it takes to query the acceleration data structure. The saved time then has to be larger than the added overhead.

The conditions for acceleration can now be discussed. We can strive to diminish the $t_{build} = \frac{T_{build}}{N \cdot p_{sp}}$ summand, which can be achieved by shooting a larger number of rays into the scene, making the number of visibility queries N larger. We could also make a larger proportion of visibility tests spatial (higher p_{sp}) or reduce the total time T_{build} needed to build the data structure.

We can also directly optimize the time it takes to query the acceleration data structure t_{acc}.

Looking at the left hand side of the equation, a larger percentage of accelerated spatial visibility tests $p_{acc|sp}$ increases the time saved. Also, if the original visibility query takes a lot of time t_{query}, for example when the scene contains a huge amount of geometry primitives, the opportunity for acceleration gets much higher.

If we neglect the build time (which we theoretically can, for large N), from 2.7 we get

$$p_{acc|sp} > \frac{t_{acc}}{t_{query}} \tag{2.8}$$

That said, if we express t_{acc} as a percentage of t_{query}, a larger percentage of spatial visibility tests have to be accelerated in order for the acceleration to be of any use.

We can also look at the total acceleration time, which is $T_{render} - T^{acc}_{render}$:

$$\begin{aligned} & T_{render} - T^{acc}_{render} && = \\ & = N \cdot t_{query} - T_{build} - (1 - p_{acc}) \cdot N \cdot t_{query} - p_{sp} \cdot N \cdot t_{acc} && = \\ & = p_{acc} \cdot N \cdot t_{query} - p_{sp} \cdot N \cdot t_{acc} - T_{build} \end{aligned} \tag{2.9}$$

For large N, we can neglect T_{build} and we get:

$$\begin{aligned} & p_{acc} \cdot N \cdot t_{query} - p_{sp} \cdot N \cdot t_{acc} - T_{build} && \approx \\ & \approx p_{acc} \cdot N \cdot t_{query} - p_{sp} \cdot N \cdot t_{acc} && = \\ & = N \cdot p_{sp} \cdot (p_{acc|sp} \cdot t_{query} - t_{acc}) \end{aligned}$$

Factor N is not surprising in the above result, because after neglecting the time to create the acceleration structure, the total speed up should be linear in number of visibility queries. We already know that by dividing the T_{build}, the factor p_{sp} helps amortize the acceleration structure build time. But we see here that it also improves the total acceleration time.

From the Equation 2.9, we can calculate the upper limit for the time saved by the acceleration. We would reach this limit if we were able to accelerate 100% of the visibility tests and all of them would be spatial, if the time it would take us to query the acceleration data structure would be almost zero and if the build time of the data structure was negligible.

In that case, because $p_{acc} = 1$, $t_{acc} = 0$ and $T_{build} = 0$, we would get:

$$\begin{aligned} T_{render} - T^{acc}_{render} &= 1 \cdot N \cdot t_{query} - p_{sp} \cdot N \cdot 0 - 0 \\ &= T_{query} \end{aligned}$$

Thus, we cannot possibly save more time by the acceleration than it takes to execute all of the visibility queries.

3. Distance Fields

The rationale behind studying distance fields for the purpose of visibility query acceleration is that to accelerate some of these queries, we do not need to know the exact shape of the scene's geometry. It is sufficient to know the distance of the query from the geometry. If the geometry is far enough, we do not need to perform any tree traversal and we immediately know that the query is unoccluded.

To give a more specific example, if we want to know, whether a line between points A and B is unoccluded, and we happen to know the distance to the closest point on the scene's geometry for both points and this distance is for either of the points larger than distance from A to B, we can safely conclude that the line does not intersect the scene's geometry (see Figure 3.1).

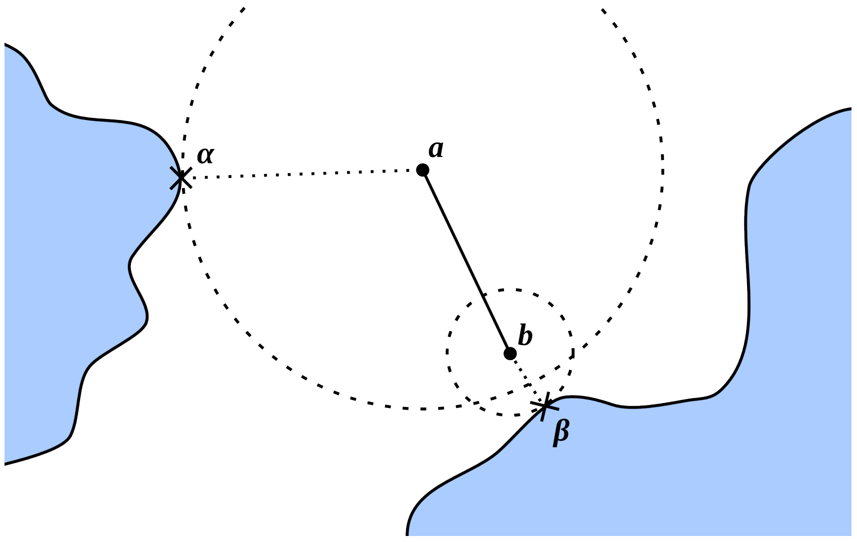

Figure 3.1 – Test on visibility between points a and b. Closest points on the scene's geometry are marked by a cross for both points. Point b lies closer to a than a to α, which means that we can say that line $a \leftrightarrow b$ is unoccluded, only from the knowledge of distances from a to α and a to b.

In this chapter, I introduce the concept of distance functions and distance fields more formally and describe how I used them in my work.

3.1 Distance Functions

My definition of a distance function is analogous to the definition used by Jones et al. in [9], the only difference being in explicitly using the metric function so as to hint on the relation between the definition of distance and the resulting distance field.

Definition 3. *Given a metric space* $\mathbb{M}$ *with metric* d *and a subset of the space* $\Sigma \subseteq \mathbb{M}$*, let us define an* ***unsigned distance function*** dist_Σ *with respect to* Σ

in a following manner:

$$\mathrm{dist}_\Sigma(p) = \inf_{x\in\Sigma} \mathrm{d}(x, p)$$

That means that for every point p in the space $\mathbb{M}$, the function dist_Σ returns the distance to such point from the set Σ that is closest to p in terms of the metric d. In this thesis, $\mathbb{M}$ will always be $\mathbb{R}^2$ or $\mathbb{R}^3$. Furthermore, the subset Σ will always correspond to the set of all surface points of the scene geometry, so I will omit the subscript.

Reworded in the context of rendering and accelerating visibility queries, the function $\mathrm{dist}(p)$ will return for every point $p \in \mathbb{R}^3$ the smallest distance to the scene's geometry.

There are also *signed distance functions* which return signed distance depending on whether the point is inside or outside of the geometry defined by Σ, but we do not need them for our purposes, so I will omit the term "unsigned".

Definition 4. *A* ***distance field*** *is a distance function such that its underlying metric space* $\mathbb{M}$ *is discrete.*

Note that the terms *distance function* and *distance field* are used somewhat interchangeably in the literature, but I will always refer to a discrete distance function as a distance field.

3.2 Choosing Grids over Hierarchies

When choosing the correct visibility query acceleration data structure, it is important to have in mind the initial requirements, which were to minimize the query time and most importantly ensure that the acceleration is conservative in the sense that the data structure does not classify an occluded visibility query as unoccluded.

As was outlined at the beginning of this chapter, to be able to accelerate visibility queries, we need to know the distance function value for every point in the scene. To calculate this value, we have several options.

The most obvious one is a brute force method, which calculates the closest distance for every geometrical primitive in the scene and keeps track of the smallest value. Of course, if our goal was to make the visibility queries as quick as possible, this approach would not be useful.

In chapter 3.1 of a paper by Jones & al. [9], we can find a summary of a variety of hierarchical methods to compute the value of the distance function and sometimes also the closest point, but after some consideration, I decided to not use these methods because their time overhead is still not negligible.

The only approach that is quick enough for our purposes is use of a data structure which stores the distance function value in a grid. That way, a distance function value query consists of a mere affine transformation of scene coordinates, rounding the transformed result to yield grid coordinates and finally of the retrieval of the stored value.

The grid methods for distance fields usually but not necessarily use a regular rectilinear grid.[†] In my work, I decided to use Cartesian grid for the sake of simplicity of the used algorithms, but more importantly also because using an irregular grid would mean for each of the coordinates some sort of tree data structure that would convert the scene coordinates into grid coordinates, which would add unnecessary time complexity to the visibility queries.

3.3 Voxel Grids

For the purposes of further definitions and discussion, let us now define voxel grids and voxels more formally.

Definition 5. *A* ***voxel grid*** $\mathbb{V}$ *with* ***resolution*** (X, Y, Z)*,* ***origin*** (o_x, o_y, o_z) *and* ***voxel size*** s *is defined as a Cartesian product of sets of certain intervals on* $\mathbb{R}$*:*

$$\begin{aligned}\mathbb{V} = \{[o_x, o_x+s], [o_x+s, o_x+2s], \dots, [o_x+(X-1)\cdot s, o_x+Xs]\} &\quad \times\\ \times \{[o_y, o_y+s], [o_y+s, o_y+2s], \dots, [o_y+(Y-1)\cdot s, o_y+Ys]\} &\quad \times\\ \times \{[o_z, o_z+s], [o_z+s, o_z+2s], \dots, [o_z+(Z-1)\cdot s, o_z+Zs]\} &\end{aligned}$$

Definition 6. *A member of a voxel grid is called a* ***voxel****.*

I define ***coordinates*** *of the voxel* $v \in \mathbb{V}$ *to be the unique triple*

$$coords(v) = (a, b, c) \in \mathbb{N}$$

for which the following equation holds:

$$v = ([o_x+as, o_x+(a+1)\cdot s], [o_x+bs, o_x+(b+1)\cdot s], [o_x+cs, o_x+(c+1)\cdot s])$$

Each voxel can also be assigned unique linear coordinates which are useful for indexing a linear array. If a voxel v *has coordinates* (a, b, c)*, I define* ***linear coordinates*** *of the voxel to be equal to:*

$$lincoords(v) = a + b * X + c * X * Y$$

where (X, Y, Z) *is grid resolution of the corresponding voxel grid.*

3.3.1 Σ on a Voxel Grid

If we have a set of surface points $\Sigma \subset \mathbb{R}^3$ and a voxel grid $\mathbb{V}$ on which we want to create a distance field, we can set

$$\Sigma' = \{v \in \mathbb{V} \mid v \cap \Sigma \neq \emptyset\}$$

This ensures that Σ' is conservative in the sense that every point from Σ is contained in a voxel from Σ'. The distance can then be created with respect to Σ'.

A voxel from the set Σ' is called an **occupied** or non-free voxel. The other voxels are called **free**.

[†]For example Miloš Šrámek and Arie Kaufman extend their *Chessboard Distance Traversal algorithm* to irregular rectilinear grids in [17].

3.4 Metrics

When defining the distance functions and distance fields, various underlying metrics can be used. The three basic and most frequently used metrics are Euclidean metric, Manhattan metric and chessboard metric.

In the following descriptions of the metrics, a and b denote points from $\mathbb{R}^3$ and V and U denote voxels from a voxel grid. For better clarity, coordinates of the points or voxels are designated using letter subscripts instead of number subscripts – a_x, b_y, V_z, etc. Only three-dimensional versions of the metric functions are described here – their two-dimensional counterparts can be obtained easily.

Definition 7. *In a metric space* $\mathbb{M}$ *with metric* d*, an (open)*[†] ***ball*** *with radius* r *around a point* $\mathrm{p} \in \mathbb{M}$ *is defined as a locus of all points closer to* p *than* r*:*

$$\mathrm{ball}(p, r) = \{q \in \mathbb{M} \mid \mathrm{d}(p, q) < r\}$$

With each metric, I describe what an open ball in that metric looks like, because the notion of a ball corresponds with a later notion of a safe zone.

3.4.1 Euclidean Metric

In Euclidean metric, differences between the coordinates are squared, added together and a square root is taken of the result. The locus of points with the same distance to a given point is a circle (in 2D) or a sphere (in 3D) centered around that point. It is important to note that when distance transforms are used to compute values in a distance field, the result is only an approximation of the Euclidean metric.

$$\mathrm{d}_{\mathrm{Eucl}}(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$
$$\mathrm{d}_{\mathrm{Eucl}}(V, U) = \sqrt{(V_x - U_x)^2 + (V_y - U_y)^2 + (V_z - U_z)^2}$$

3.4.2 Manhattan Metric

The Manhattan metric defines distance between two points or voxels as a sum of differences between the individual coordinates. The locus of points which have the same distance to a given point is an octahedron centered around the point.

$$\mathrm{d}_{\mathrm{Man}}(a, b) = |a_x - b_x| + |a_y - b_y| + |a_z - b_z|$$
$$\mathrm{d}_{\mathrm{Man}}(V, U) = |V_x - U_x| + |V_y - U_y| + |V_z - U_z|$$

3.4.3 Chessboard Metric

The chessboard metric calculates the distance of two points or voxels by taking the maximum of differences of all their coordinates. The locus of points equally distant from a given point is a cube centered around the point.

[†]There are also closed balls, but I will not need them in this thesis.

$$\mathrm{d}_{\mathrm{CD}}(a,b) = \max(|a_x - b_x|, |a_y - b_y|, |a_z - b_z|)$$
$$\mathrm{d}_{\mathrm{CD}}(V,U) = \max(|V_x - U_x|, |V_y - U_y|, |V_z - U_z|)$$

3.4.4 Choosing a Metric

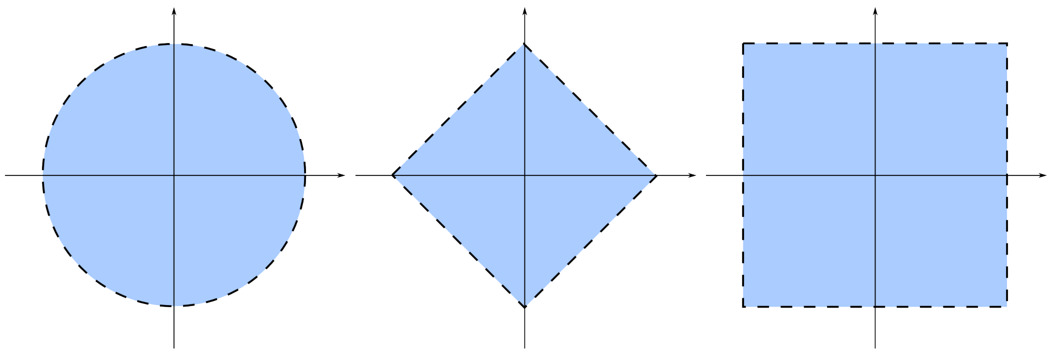

Figure 3.2 – Comparison of open balls in the Euclidean, Manhattan and chessboard metrics.

The first experiments were tried with a distance field based on the chessboard distance and using the Euclidean metric for calculating the length of the line between a and b. This choice was made because both the chessboard metric distance field and Euclidean distance between two points were easy to compute. However, there are two major problems with this choice.

To understand the first problem, let us introduce the concept of a safe zone.

Definition 8. *In an unsigned distance function* dist*, given a point* p *from the domain of that function (i.e. a metric space* $\mathbb{M}$ *with a metric* d*), we can define a **safe zone*** safe(p) *around that point in the following manner:*

$$\mathrm{safe}(p) = \{q \in \mathbb{M} \mid \mathrm{d}(p,q) < \mathrm{dist}(p)\}$$

Safe zone for a voxel is defined analogously. In a distance field Dist*, given a voxel* v *from the voxel grid* $\mathbb{V}$ *with metric* d*, the safe zone around* v *is then defined as:*

$$\mathrm{safe}(v) = \{u \in \mathbb{V} \mid \mathrm{d}(v,u) < \mathrm{Dist}(v)\}$$

We can see that the safe zone around a point p from a metric space $\mathbb{M}$ is actually a ball with radius $\mathrm{dist}(p)$. The meaning of a safe zone is quite obvious – it defines a set of points around p that does not intersect Σ. If the safe zone did contain some point $\pi \in \Sigma$, it would by definition of a ball (and a safe zone) mean that $\mathrm{d}(\pi, p) < \mathrm{dist}(p)$. But that would contradict the definition of the distance function.

With the definition of a safe zone and with the knowledge of the shape of balls under the three described metrics comes another argument for the use of the chessboard metric for the distance field. Scene geometries more often contain rectangular shapes, rooms and corridors rather than spherical or octahedral empty spaces.

To return back to visibility acceleration, if we knew that one point is in the safe zone of the other point, we could be certain that the line between those two points is not occluded. If we were able to find out the exact value of the distance function in a point, we could use the following algorithm to accelerate visibility queries.

```
 1  // Tells whether the line between points a and b is unoccluded
 2  function isUnoccluded(a, b):
 3    distA := distanceAtPoint(a)
 4    distB := distanceAtPoint(b)
 5    if (distA > d(a, b) or
 6        distB > d(a, b)) then
 7      return true
 8    endif
 9    return standardOcclusionTest(a, b)
10  end function
```

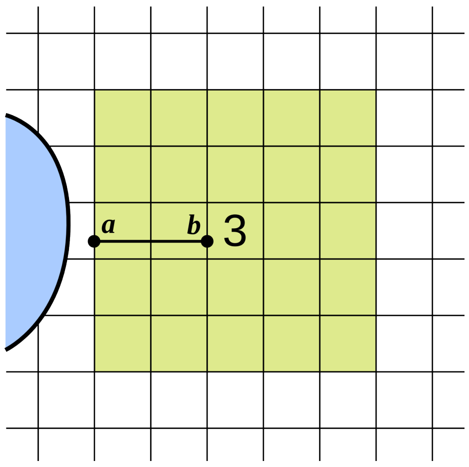

Figure 3.3 – Radius of a safe zone and the shortest possible line going from the center voxel out of the safe zone.

With voxel grids, however, the situation is slightly more complicated. As was discussed above, the value of a voxel in a distance field defines a safe zone of a corresponding radius. As can be seen in the Figure 3.3, the safe zone around a voxel with the value 3 has an actual radius of 2, because the voxels one step further are already too far – their distance of 3 equals to the value in the central voxel, which, by the definition of a safe zone, is not allowed (remember that the safe zone is an open ball, which does not contain the boundary).

This means that the shortest line that could possibly span from the central voxel out of the safe zone would in this case be only 2 units long. In order to still ensure correctness when accelerating visibility queries, we have to change lines 5 and 6 in the previous function to

```
5    if (distA > d(a, b) + 1 or
6        distB > d(a, b) + 1) then
```

This, however creates a problem. In voxels with a distance field value of one, no acceleration can be achieved, since by adding one to any distance, we get at least one, which will always be equal to or larger than the value from the distance field. This is the case of the line $c \leftrightarrow d$ from the Figure 3.4. A visibility query between points c and d would not be accelerated, although the line between them does not intersect any voxels with the value zero and although both points lie inside the safe-zone of the voxel where the other point is located.

The second problem with the approach is obviously the usage of different metrics in the distance field and for the calculation of the length of the line between the two points. It would make much more sense and more queries would be accelerated if the chessboard metric was used here as well[†]. This might not seem as such a big problem, but the ratio of the volumes of a cube and an inscribed ball is approximately 1.9, which means that about 48% of the points that would be classified to be inside of the safe zone by the chessboard metric are classified to be outside using the Euclidean metric.

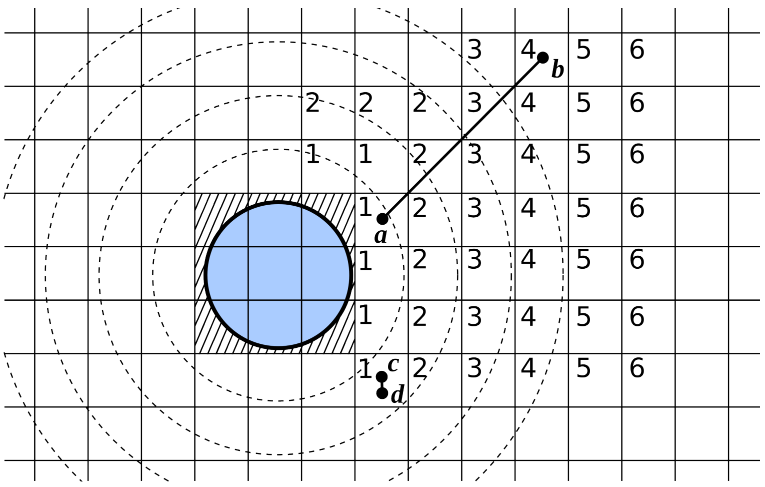

Figure 3.4 – Illustration of the problem with usage of the Euclidean metric for line lengths but chessboard metric for generating the distance field.

I briefly tried to use the chessboard metric to calculate the length of the line, but the problem with queries that stay in a single voxel, of course, persisted. I then realized that instead of measuring the length of the line between the two points, I could find the respective voxels for both points and check, whether one of the voxels is in the safe zone of the other one. This would happen when the distance between the two voxels (in the same metric that was used to construct the distance field) is smaller than either of the two distance field values. That way, the problem with short queries is completely eliminated.

The two lines in the accelerated visibility query function would then be

```
5  if (distA > dV(voxel(a), voxel(b)) or
6      distB > dV(voxel(a), voxel(b))) then
```

where `dV` is a metric defined on the voxels of the distance field and the `voxel` function returns for a given point the voxel in which the point is included.

[†]Although, the Euclidean ball is completely contained in the chessboard ball, which means that by using the Euclidean metric, no points are classified as unoccluded that would not be classified equally using the chessboard metric to calculate the line length.

3.5 Distance Transforms

In order to create a grid-based distance field, a variant of a method called the *distance transform* is usually applied. From a high-level point of view, this transform is just a function, which converts a 2D or 3D binary grid into a representation, where each pixel or voxel maps to a number which represents an approximation of the distance function in that pixel or voxel.

3.5.1 Initialization

Before the distance transform methods are applied, the grid first needs to be initialized. This is done by setting the value of every grid voxel that is intersected by the scene's geometry to zero and the value of other voxels to infinity (or any value that is larger than any possible distance)

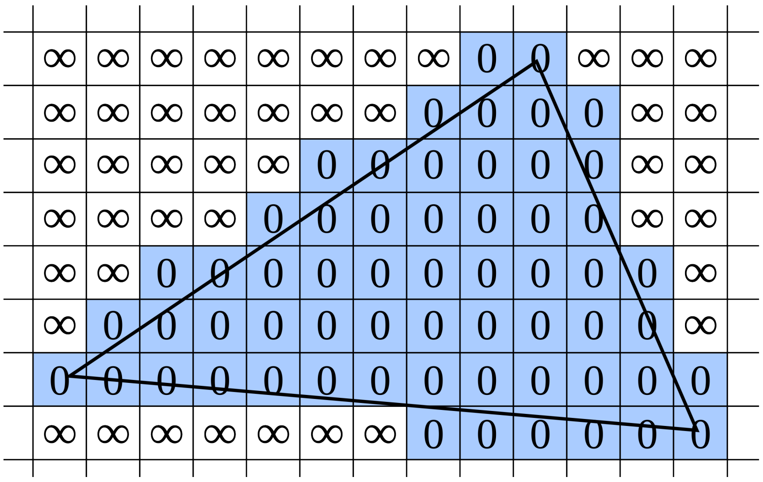

Figure 3.5 – Initialization of the voxel grid – the voxels intersected by the triangle are set to zero, the others to infinity.

Old Method

Initialization of the grid is usually done by a scan-line conversion of the scene or other input data. I first tried a slightly different approach – loop over all the scene's triangles and tessellate them so that each of the sub-triangles fits into a sphere with a radius smaller than half of the voxel grid. Then for each vertex of the sub-triangles, I marked the voxel, in which the vertex lied, as a voxel that belonged to the set Σ'. Using this method, I could guarantee that for each input triangle, the resulting marked set of voxels was 26-connected.[†] Unfortunately, as I found out later, there were situations in which an intersected voxel was left unmarked, which was unacceptable because one of our initial requirements was for the acceleration to be conservative.

[†]A set of pixels in 2D is said to be 8-connected, when for every pair of pixels in the set there exists a path consisting only of pixels in the set such that from each pixel the path continues to one of the 8 possible neighbors – vertically, horizontally or diagonally – hence 8-connectivity. In 3D, the situation is analogous, but there are 26 possibilities.

Improved Method

Although scan-line conversion or rasterization is an often encountered problem that can be solved using a dedicated graphics processing hardware, a conservative rasterization† is more difficult and until several years ago had rarely been described in literature or implemented in the dedicated hardware (as stated in the abstract of a paper by Tomas Akenine-Möller and Timo Aila [1]). I implemented the method described by Zhang et al. in [21].

This method is a three-dimensional conservative rasterization technique that can be applied to planar polygons. Each of the polygons first has to be projected into two dimensions and a two-dimensional conservative rasterization method applied to it. The minimal and maximal depth value (value of the last dimension) of the intersecting triangle for each of the intersected pixels are then found and converted to depth coordinates of three-dimensional voxels in which the extrema were found. It is then safe to assume that all voxels in between the minimal and maximal voxel are also intersected by the plane.

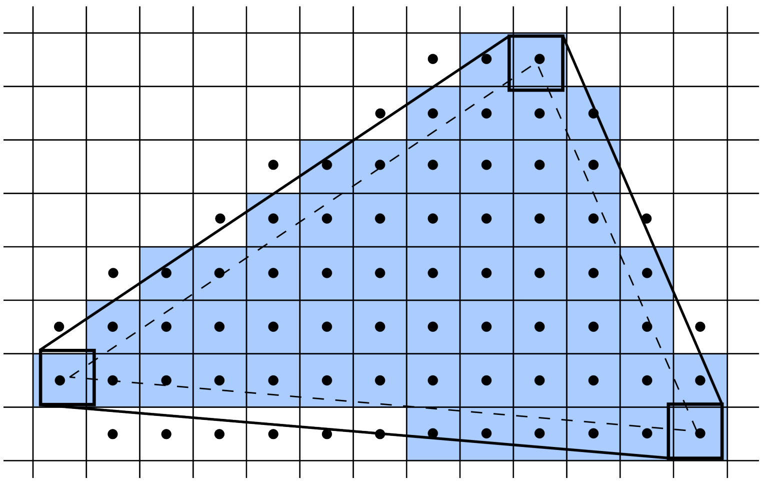
Figure 3.6 – Conservative rasterization of a triangle. The triangle is converted into an offset polygon, which is then rasterized using the standard method – select those pixels, whose centers lie in the polygon.

The two-dimensional rasterization algorithm that I used was a method described by Hasselgren et al. in one of the second edition GPU Gems [7]. Their method creates an offset polygon encompassing the rasterized triangle by centering a pixel-sized box at each of the triangle's vertices and then creating a convex hull of these three boxes. This can be done surprisingly easily by assigning to each of the triangle's edges the quadrant of the Cartesian plane that this edge's normal lies in and creating vertices of the offset polygon according to the relative positions of the quadrants of the neighboring edges.

Thus created polygon is then rasterized into a set of pixels by selecting those pixels, whose sample point – their center – lies in the polygon (see Figure 3.6). This type of rasterization by sampling the central point is mentioned in [1, Akenine-Möller, Aila].

†This means either selecting all intersected pixels/voxels or an another, but similar, problem – finding all pixels/voxels completely encased in the given primitive.

Because there is a potential of a large number of point-in-polygon tests, I convert the convex polygon[†] into a set of parallel trapezoids using the following algorithm (see Figure 3.7):

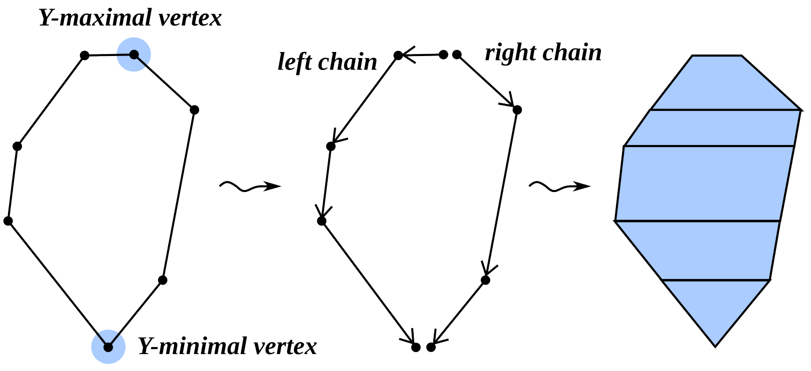

Figure 3.7 – Illustration of the algorithm that converts a convex polygon into a set of parallel trapezoids.

First, I find the two vertices of the polygon that have the minimal and maximal y coordinate. The other vertices of the polygon are added into two chains that both start with the maximal vertex and end with the minimal one, but one of the chains consists of vertices left of the polygon and the other one of vertices that are to the right of it.

The two chains are then traversed in parallel in order to find a way to decompose the polygon into a set of parallel trapezoids. In this representation, it is much quicker to determine, which of a set of points lie in the polygon, especially if this set contains subsets of points with an equal y coordinate. This is because a single test on a point belonging into the polygon is done by finding a trapezoid whose upper and lower bounds contain the y coordinate of the point and then testing whether the point lies left of the right trapezoid edge and right of the left one. When testing a set of points that share the same y coordinate, the trapezoid can be selected once for the whole set for an even quicker subsequent querying.

3.5.2 Chamfer Distance Transform

There are a variety of possible methods to implement a distance transform and they are well summarized in [9, Jones & al.]. For our purposes, a simple chamfer distance transform with a wavefront scheme is suitable.

Chamfer distance transforms update the value in a voxel by taking the values of the neighboring voxels, adding values from a pre-defined distance template to them and then taking the minimal of the resulting values as a new value for the current voxel.[‡] This is not done if the current value of the voxel is smaller than the new value. The distance template essentially contains distances from a voxel to neighboring voxels and during update these distances are added to the

[†]The convex property is important for the following algorithm to work.

[‡]The name comes from the Chamfer metric, which is an approximation of the Euclidean metric and emerges when the distance templates contain Euclidean distances between the neighboring voxels.

distances already stored in the neighboring voxels in order to see, whether the distance in the current voxel should be updated.

The following pseudo-code clarifies the update of a voxel. For clarity reasons, the code is shown in the 2D version. Extension to three dimensions would be straightforward.

```
// The distance template
template := [[ 1.4, 1, 1.4 ],
            [ 1 , 0, 1 ],
            [ 1.4, 1, 1.4 ]]

// The update function receives voxel coordinates as arguments
function update(x, y):
  global distfield
  smallestValue := distfield[x, y]
  for i from -1 to 1 do
    for j from -1 to 1 do
      if (validCoords(x+i, y+j) and
          distfield[x+i, y+j] < smallestValue) then
        smallestValue := distfield[x+i, y+j]
      endif
    end
  end
  distfield[x, y] := smallestValue
end function
```

3.5.3 Propagation Schemes

When updating a value of a voxel, one has to bear in mind that the neighboring voxels might not contain correct distance field values yet, so the distance might not propagate correctly. To counter this problem, sweeping schemes have been developed (as early as in 1966 by Azriel Rosenfeld and John L. Pfaltz [14]), in which the distance field is swept over several times, each time with a different template and in a different order of voxels, which ensure that the distance will be propagated correctly.

I chose another approach (mentioned in [9, Jones & al.]), which is to process the voxels in one pass. This method starts with a queue that is filled with voxels neighboring the marked voxels. Then, one by one, the voxels are taken from the queue and updated. After each update, the voxel's neighbors that have not yet been added to the queue are sorted into the queue. This continues while there are voxels in the queue. In my case, I used a distance template, which contained only ones and zeros. This way, I didn't have to order the new voxels into the queue, but could append them to the end of the queue instead. This also ensured that all the distances were integers.

The latter approach is called a *wavefront scheme*, because a visualization of the voxels in the queue resembles a wavefront moving away from the scene's objects.

3.6 Performance of Non-Directional Methods

As can be seen from the results presented in Chapter 6, the non-directional distance field methods perform well in terms of low query time t_{acc}, but the percentage of accelerated spatial visibility queries $p_{acc|sp}$ is very low due to the safe zones' size being hindered by geometry in irrelevant directions. This can be clearly seen in Figure 3.8, which visualizes the sets of voxels, from which the visibility queries towards the point light can be accelerated in the scene **cbox_vol6**. First, this set is displayed for a non-directional distance field, then for the Anisotropic Macro-Regions, one of the methods that will be described in the next chapter.

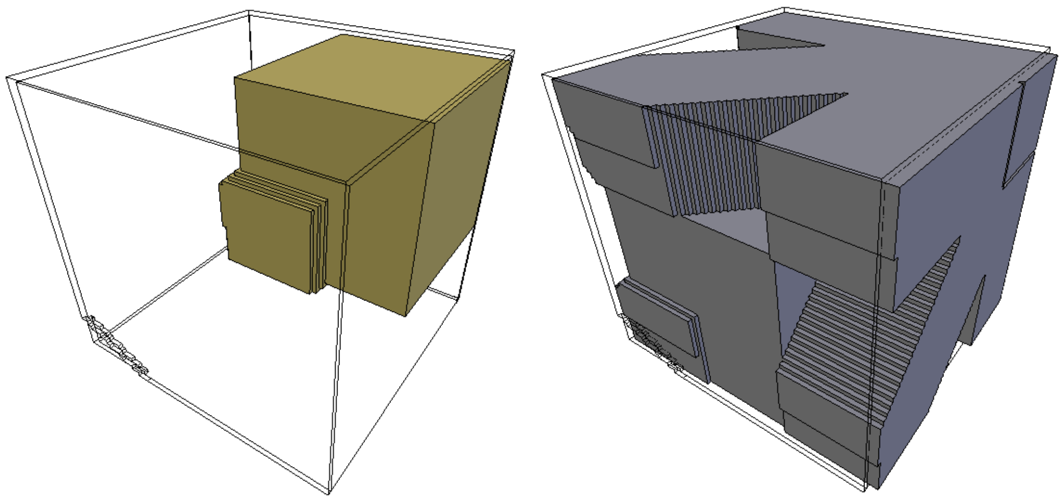

Figure 3.8 – The visibility queries between the visualized sets of voxels and the point light can be accelerated. To the left are such voxels for the non-directional distance field and to the right for AMR. The displayed scene is **cbox_vol6** and the colored meshes are the safe neighborhoods (see Section 4.4) of its point light source, which is located somewhere inside the meshes.

For this reason, I decided to research directional distance fields, which are described in the next chapter.

4. Directional Distance Fields

In order to find a better method for acceleration of visibility queries, I looked into several directional distance field methods. They redefine the notion of a safe zone so that it no longer depends on the given pixel/voxel, but also on a direction. Some of the methods extend the classical distance field information with directional data (e.g. the Dual Extent Method described by Sudhanshu K. Semwal and Håkan Kvarnström[15]), whereas the others divide the voxel into several parts with each of these parts containing safe zone data for a different direction (e.g. Anisotropic Macro-Regions by [17, Šrámek, Kaufman]).

4.1 Anisotropic Macro-Regions

This method was proposed in [17, Šrámek, Kaufman] as an improvement to the authors' Chessboard Distance (CD) Traversal algorithm, which used distance fields for accelerating ray-tracing queries. In a ray-tracing query, a point and an associated vector are given (construing a ray) and the query returns the closest intersection with the scene geometry.

The CD Traversal algorithm registers the individual objects from the scene to all voxels of a voxel grid that are intersected by each object. Ray-tracing could be then implemented by stepping through all voxels intersected by the given ray and testing intersection with all objects registered in each voxel. The CD Traversal algorithm exploits the observation that empty voxels can be safely skipped and that such voxels tend to aggregate into larger areas of empty space. A distance field is then used to skip more than one voxel, according to the value in the distance field. The distance field in this case tells us the minimal distance to a non-empty voxel using a chessboard distance metric.

In order to simplify the following definitions, I define the relation "$A\xrightarrow[\pm\pm\pm]{}B$"[†] for points or voxels in $\mathbb{R}^3$, where each "$\pm$" can be either "$+$" or "$-$" and these signs then denote relation of the individual coordinates of A and B, that is, whether we have to add or subtract a positive value from the respective coordinates in order to get from A to B. For example:

$$A\xrightarrow[-+-]{}B \quad \Leftrightarrow \quad A_x \geq B_x \;\wedge\; A_y \leq B_y \;\wedge\; A_z \geq B_z$$

This notation can also be interpreted as saying that B lies in the $(-+-)$ octant of a new coordinate system that has its origin in A.

The Anisotropic Macro-Regions (AMR) improve upon the ideas in CD Traversal Algorithm by modifying the distance field to contain eight values in each voxel. I will denote these values by appending the distance function name with a superscript of three sign marks (one for each dimension):

[†]there would be just two $\pm$ signs in the similar 2D definition

Definition 9. *In a voxel grid $\mathbb{V}$ over $\mathbb{R}^3$ and for a given $\Sigma' \subseteq \mathbb{V}$, the* ***anisotropic distance field*** *in the direction $- + -$ is defined as:*

$$\mathrm{dist}^{-+-}(V) = \inf_{U \in \Sigma' \,\wedge\, V \xrightarrow[-+-]{} U} \mathrm{d_{CD}}(U, V)$$

That is, $\mathrm{dist}^{-+-}(V)$ returns the distance to the closest non-empty voxel U in the $(- + -)$ octant of V such that $U \in \Sigma'$. (see Figure 4.1).

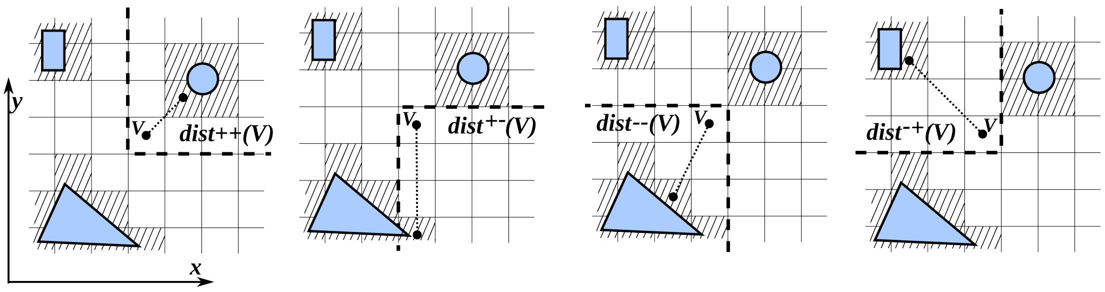

Figure 4.1 – Illustration of $\mathrm{dist}^{\cdot\cdot}(V)$ in 2D case

Similarly to the definition in [17, Šrámek, Kaufman], anisotropic macro-regions can be defined for each voxel, parametrized by the voxel itself, maximal distance of the voxels in the macro-region from the origin voxel and by sign marks denoting the direction of the macro-region†:

$$\mathcal{O}^{-+-(n)}(V) = \{U \in \mathbb{R}^3 \mid V \xrightarrow[-+-]{} U \;\wedge\; \mathrm{d_{CD}}(U, V) \leq n\}$$

Each of the eight $\mathrm{dist}^{-+-}(V)$ values now describes a size of the maximal empty macro-region in the corresponding direction and can be used to accelerate ray-tracing in the same way values from a classical distance field are used. Only now we can choose the value that corresponds to the direction vector of the ray. That way, the traversal can be much quicker in situations when the ray passes a geometry which would normally reduce the size of the safe-zone, because the classical distance field value is equal to minimum of all the $\mathrm{dist}^{\cdot\cdot}(V)$ values in the given vertex V. This situation is illustrated in Figure 4.2.

4.1.1 Construction

To construct a directional distance field that uses AMR, a sweeping scheme distance transform can be used, analogous to the one that is used to construct an ordinary distance field.

In the first phase of the construction algorithm, all eight values in the voxels that intersect the scene's geometry are set to zero. Values in the remaining voxels are set to infinity.

The second phase then has eight sub-phases, each corresponding to one of the eight parts of the voxels and consequently to one of the eight diagonal directions. In each of these sub-phases, the values are propagated only in the respective parts of the voxels using a correct distance template and voxel evaluation order (see Figure 4.3). The value from the current voxel is added to the values in the

†The meaning of the sign marks is the same as in the definition of $\mathrm{dist}^{\cdot\cdot}(V)$

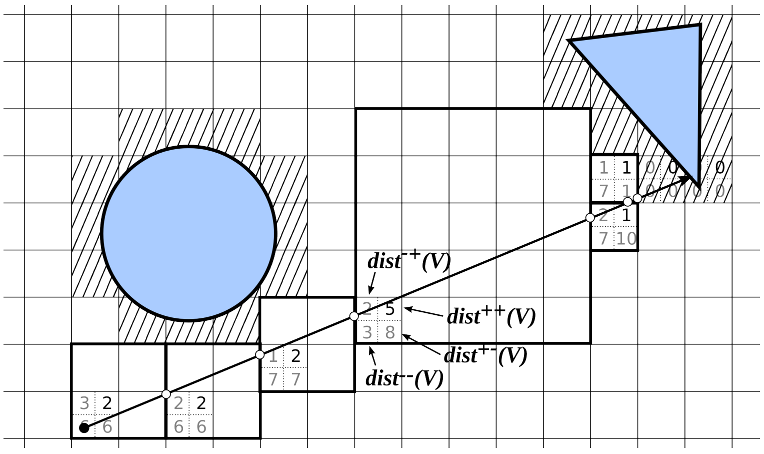

Figure 4.2 – Illustration of ray-tracing acceleration using AMR in 2D. Distance field values in each visited voxel are shown together with the associated macro-regions.

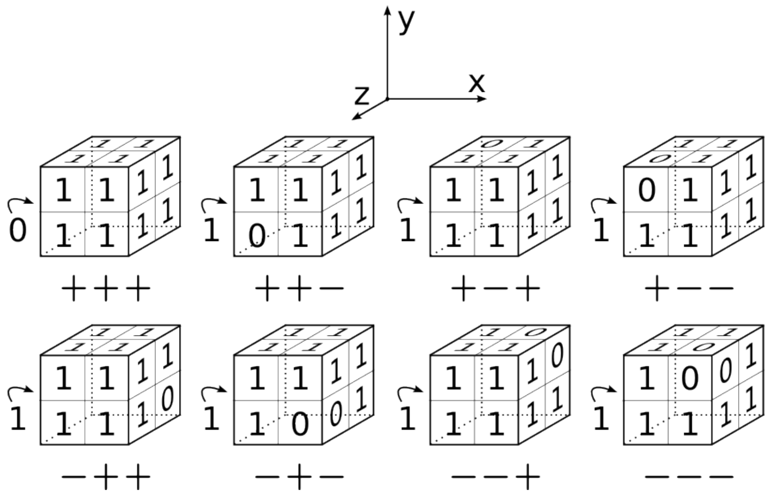

Figure 4.3 – Distance templates for the eight sub-phases of the construction algorithm for AMR-based directional distance field. The current voxel is always the one with a 0 value in the template

voxel template and the results are compared to the values in the corresponding neighbors of the current voxel. If the value in a neighbor is larger, it is updated with the newly calculated value.

4.1.2 AMR Safe Zones

The notion of safe zone can be extended to Anisotropic Macro-Regions by introducing function Dist_{AMR}, whose value does not depend on a single voxel, but on two voxels. The function first finds out the octant of the second voxel, in which the second voxel lies. The value of the corresponding function $\text{dist}^{\cdot\cdot}$ is then returned:

$$\mathrm{Dist}_{AMR}(U,V) = \begin{cases} \mathrm{dist}^{+++}(U) : U \underset{+++}{\rightrightarrows} V \\ \mathrm{dist}^{++-}(U) : U \underset{++-}{\rightrightarrows} V \\ \mathrm{dist}^{+-+}(U) : U \underset{+-+}{\rightrightarrows} V \\ \mathrm{dist}^{+--}(U) : U \underset{+--}{\rightrightarrows} V \\ \mathrm{dist}^{-++}(U) : U \underset{-++}{\rightrightarrows} V \\ \mathrm{dist}^{-+-}(U) : U \underset{-+-}{\rightrightarrows} V \\ \mathrm{dist}^{--+}(U) : U \underset{--+}{\rightrightarrows} V \\ \mathrm{dist}^{---}(U) : U \underset{---}{\rightrightarrows} V \end{cases} \tag{4.1}$$

Note that several of the $\underset{\pm\pm\pm}{\rightrightarrows}$ relations can be satisfied for a single pair of voxels U, V [†]. In that case, the $\mathrm{dist}^{\pm\pm\pm}$ for the first satisfied relation is returned.

The Equation 4.1 can now be used to define a safe zone in the Anisotropic Macro-Region distance field:

Definition 10. *Given an anisotropic distance field* $\mathrm{dist}^{\pm\pm\pm}$ *defined for all directions* $\pm\pm\pm$ *on a voxel grid* $\mathbb{V}$ *over* $\mathbb{R}^3$*, and a voxel* $V \in \mathbb{V}$ *the* ***anisotropic safe zone*** *is defined as:*

$$\mathrm{safe}_{AMR}(V) = \{U \in \mathbb{V} \mid \mathrm{d}(V,U) < \mathrm{Dist}_{AMR}(V,U)\}$$

4.1.3 AMR Acceleration of Visibility Queries

In order to find out whether two points a and b in $\mathbb{R}^3$ are visible from each other, an AMR-based distance field can now be used in a following manner:

First, one has to find out two grid voxels A and B that correspond to these points. Without loss of generality, let's presume that B lies in the (+++) octant of A. Now we just have to check, whether either A lies in the anisotropic macro-region of B defined by $\mathrm{dist}^{---}(B)$ or B lies in the anisotropic macro-region of A defined by $\mathrm{dist}^{+++}(A)$. If either of these conditions is met, the line $a \leftrightarrow b$ is surely unoccluded by the scene's geometry. Conversely, if both of these tests fail, standard visibility testing has to proceed.

The query can also be described in terms of safe zones. We can presume the line $a \leftrightarrow b$ to be unoccluded if either A is contained in $\mathrm{safe}_{AMR}(B)$ or B is contained in $\mathrm{safe}_{AMR}(A)$.

4.2 Directed Safe Zones

This is a method described by Sudhanshu K. Semwal and Håkan Kvarnström in [15]. Similar to the paper that described the Anisotropic Macro-Regions, the authors propose this method for acceleration of ray tracing queries. And similar to the AMR method, the Directed Safe Zones (DSZ) modify the distance field information by changing it to directional information. However, the directional

[†]That happens when at least one of their coordinates is equal

information in DSZ consists of six values instead of eight. Each value describes a safe zone for the ray that exits the voxel through one of the six faces of the voxel.

The notable difference of the Directed Safe Zones from the Anisotropic Macro-Regions is the use of Manhattan metric instead of Chessboard metric.

Similarly to the definitions for Anisotropic Macro-Regions, let us define distance field for the directed safe zones.

Definition 11. *In a voxel grid $\mathbb{V}$ over $\mathbb{R}^3$ and for a given $\Sigma' \subseteq \mathbb{V}$, we define the* ***directed safe zone size*** *in the positive direction of the X coordinate axis as:*

$$\mathrm{dist}^{x+}(V) = \inf_{U \in \Sigma' \,\wedge\, V_x < U_x} \mathrm{d_{Man}}(U, V)$$

The directed safe zone distance fields dist^{x-}, dist^{y+}, dist^{y-}, dist^{z+} and dist^{z-} are defined in a similar fashion. Note the strict inequality in the previous definition, because it implies that the voxel will not be included in its own safe zone.

Using the directed safe zone distance fields, we can define the directed safe zones themselves.

Definition 12. *Given a voxel grid $\mathbb{V}$ over $\mathbb{R}^3$ and a subset of the voxel grid $\Sigma' \subseteq \mathbb{V}$, a* ***directed safe zone*** *can be defined in positive or negative direction of each of the coordinate axes as:*

$$\mathrm{safe}^{x+}(V) = \{U \in \mathbb{V} \mid \mathrm{d_{Man}}(U, V) < \mathrm{dist}^{x+}(V) \ \wedge\ V_x < U_x\}$$

Again, the definitions for safe^{x-}, safe^{y+}, safe^{y-}, safe^{z+} and safe^{z-} are similar.

4.2.1 Construction

The DSZ distance field is created by a sweeping scheme propagation method, in a similar manner to the AMR distance field.

The first phase of the algorithm is initialization of all six values in every voxel to infinity, except for the voxels that are intersected by the scene's geometry. Values of these voxels are set to zero.

The second phase has again eight sub-phases that correspond to one of the eight diagonal directions. However, slightly different propagation templates are used this time. These templates ensure that the values are propagated only to those neighbors of the currently processed voxel that share a face with it and lie in the correct direction (see Figure 4.4).

When propagating a value from a voxel, we first take the minimum of the three values in those positions that are being updated in this sub-phase[†] and then for each neighbor that is marked by the number one in the template we add one to the value from the current voxel and propagate it to the neighbor into the correct position. The position has to correspond to the direction that we have to take in order to get from the neighbor to the current voxel. Again, as with the AMR distance fields, the value is only updated in the neighbor, if the old value was larger.

[†]These positions are exactly opposite than the propagation directions. For example, if the propagation direction is +-+, that is in the positive direction of the X axis, negative Y and positive Z, the values of dist^{x-}, dist^{y+} and dist^{z-} are used.

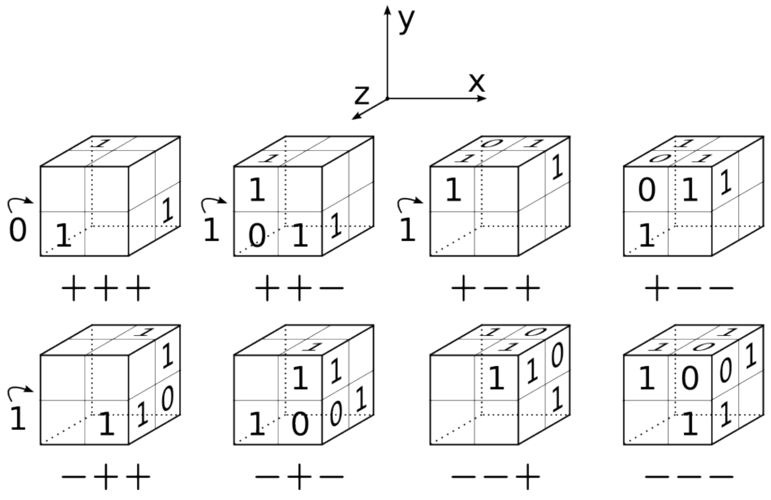

Figure 4.4 – Distance templates for the eight sub-phases of the construction algorithm for DSZ-based directional distance field. The current voxel is denoted by a zero in the template. The neighbors in the empty voxels are ignored.

4.2.2 DSZ Acceleration of Visibility Queries

The general pattern in the acceleration of visibility queries with Directed Safe Zones is the same as with the Anisotropic Macro-Regions.

For the two points a and b, we find the corresponding voxels A and B and check whether A lies in the directed safe zone of B or B in the directed safe zone of A.

We only have to select the correct safe zone, which is done using the points a and b. In voxel A, we select the safe zone according to the first face of the voxel that is intersected by the ray $a \rightarrow b$ and in the voxel B the safe zone according to the first face intersected by the ray $b \rightarrow a$.

4.3 Dual Extents

This method is described in [15, Semwal, Kvarnström]. It does not re-define the distance field information. Rather, it adds six directional values to the original non-directional distance field value[†]. Each directional value tells us, how far the safe zone from the non-directional field can be extended in the given direction without intersecting an occupied voxel. This corresponds to the number of voxels that we can traverse in the given direction without encountering a voxel, whose non-directional distance field value is smaller than the value of the original voxel. Using this knowledge, we could easily construct the Dual Extent distance field, however the authors of the original paper propose a smarter solution - see [15, Semwal, Kvarnström] for details.

To simplify the following definition, I introduce the function vox, which returns the voxel of a voxel grid on the given coordinates:

$\text{vox}(\mathbb{V}, x, y, z) = v \Leftrightarrow v \in \mathbb{V} \ \wedge \ coords(v) = (x, y, z)$

[†]The non-directional distance field used in this method uses the Manhattan distance, similar to the DSZ method.

Definition 13. *Let $\mathbb{V}$ be a voxel grid over $\mathbb{R}^3$, $\Sigma' \subseteq \mathbb{V}$ a subset of the voxel grid and* Dist *be a distance field on $\mathbb{V}$ with respect to Σ'. The* ***dual extent*** *in the positive direction of the X axis is defined as:*

$$\begin{aligned}\mathrm{dual}^{x+}(V) = \{U \in \mathbb{V} \mid |U_y - V_y| < \mathrm{Dist}(V)\ \wedge \\ \wedge\ |U_z - V_z| < \mathrm{Dist}(V)\ \wedge \\ \wedge\ \mathrm{d_{Man}}(U, V) < D^{x+}_{de}(V) + \mathrm{Dist}(V)\}\end{aligned}$$

where $D^{x+}_{de}(V)$ is the ***dual extent distance*** *in the positive direction of the X axis, defined as:*

$$D^{x+}_{de}(V) = min(\{n \in \mathbb{N} \mid \mathrm{Dist}(\mathrm{vox}(\mathbb{V}, V_x + n, V_y, V_z)) < \mathrm{Dist}(V)\}) - 1$$

The definitions for the remaining axes and directions are defined analogously.

4.3.1 Dual Extent Acceleration of Visibility Queries

In a visibility query between points a and b, we find the corresponding voxels A and B again. Similarly to the previous two methods, we proclaim the line $a \leftrightarrow b$ as unoccluded if either A lies in the correct dual extent of B or if B lies in the correct dual extent of A. The question is, how to choose the "correct" dual extent.

If we are trying to find out, whether A lies in a dual extent of B, we count the number of coordinate axes, for which the difference $|A_x - B_x|$, $|A_y - B_y|$ or $|A_z - B_z|$, is larger than or equal to the value $\mathrm{Dist}(B)$. If there is one such coordinate, we use the corresponding dual extent in such direction that it corresponds to the direction of ray $b \to a$. If there is no such coordinate, we choose from the three dual extents that are in the correct direction the one with the largest value. If there are more than one such coordinates, we cannot accelerate this query using dual extents.

4.4 Safe Neighborhoods

The distance field methods, either directional or non-directional, can be useful if the visibility queries are incoherent in the sense that both points of the queries are distributed over the whole scene. If, on the other hand, one of the points is always located in a single position in space or if there is a small number of such positions (which certainly is the case of visibility queries that sample point lights in path tracing renderers), we can accelerate the distance field queries even more by exploiting this coherence. I achieve this acceleration using my own technique that I call Safe Neighborhoods. This technique can be used in conjunction with any distance field acceleration method to further accelerate queries with one of the voxels from a selected small set of input voxels.

In order to understand the Safe Neighborhood method, we have to first define the safe neighborhood:

Definition 14. *Let* safe *be a safe zone function, defined by a directional or a non-directional distance field (using any method of choice) on a voxel grid $\mathbb{V}$ and* *let $V \in \mathbb{V}$ be a voxel. The* ***safe neighborhood*** *of V is defined as:*

$$\text{neigh}(V) = \{U \in \mathbb{V} \mid U \in \text{safe}(V, U) \vee V \in \text{safe}(U, V)\}$$

if a directional distance field is used. Otherwise, it is defined as:

$$\text{neigh}(V) = \{U \in \mathbb{V} \mid U \in \text{safe}(V) \vee V \in \text{safe}(U)\}$$

The safe neighborhood of a voxel V thus contains all voxels, which are contained in its safe zone or which contain V in their own safe zone. The safe neighborhood of a voxel V is a set of voxels which would, together with the voxel V, form visibility queries that the underlying distance field would classify as unoccluded.

The Safe Neighborhoods method saves the safe neighborhoods for a set of voxels in binary arrays. These arrays contain one, if the corresponding voxel is contained in the safe neighborhood and zero otherwise. This can be viewed as precomputing the visibility query results for a small subset of voxels. The advantage of using this method over the distance field itself stems from the fact that we can skip all distance calculations between the two points and we also ignore one of the points completely once we identify the safe neighborhood array associated with it. The query is then just a retrieval of a boolean value from the array. If none of the points of the query has a precomputed safe neighborhood, we proceed as we would without this method. In my implementation of the Safe Neighborhoods method, I can guarantee that the light sample point will be the second point of the query, which further accelerates the query.

4.4.1 Extending the Safe Neighborhoods

From the definition of the safe neighborhood and safe zones for the AMR method follows an interesting property:

The Free Cuboid Property: In the Anisotropic Macro-Regions method, if a voxel $U \in \mathbb{V}$ is in the safe neighborhood of a voxel $V \in \mathbb{V}$, then the minimal axis aligned bounding voxel cuboid $\mathbb{C}$ that contains these two voxels, defined as

$$\begin{aligned} \mathbb{C} = \{W \in \mathbb{V} \mid \quad & min(U_x, V_x) \leq W_x \leq max(U_x, V_x) \wedge \\ & \wedge\, min(U_y, V_y) \leq W_y \leq max(U_y, V_y) \wedge \\ & \wedge\, min(U_z, V_z) \leq W_z \leq max(U_z, V_z)\} \end{aligned}$$

will contain only free voxels.

Proof: By definition of safe neighborhoods, either $U \in \text{safe}(V)$ or $V \in \text{safe}(U)$. Without loss of generality, suppose that U is in the safe zone of V. The coordinates of an arbitrary voxel W from the cuboid $\mathbb{C}$ will by the definition of $\mathbb{C}$ lie between the respective coordinates of U and V and $\text{d}_{\text{CD}}(W, V)$ will thus be smaller or equal to $\text{d}_{\text{CD}}(U, V)$. This means that voxel W will also lie in the safe zone of V, which means that it must be free. ⊠

This property is valid not only in AMR, but also in non-directional distance fields and in the Dual Extent distance fields. This is because in all these methods, for a given voxel U in the safe zone of a voxel V, the minimal axis-aligned bounding voxel cuboid of U and V is contained in the safe zone of V.

As will be described in Chapter 5.3, I created a way to visualize voxel sets in order to be able to debug my implementation of the distance field methods. This method enabled me to visualize the safe neighborhoods as well. The visualization revealed a common feature of the safe neighborhoods derived from an AMR distance field – cascade edges (see the green region in Figure 4.5 or the jagged safe neighborhood in Figure 5.1).

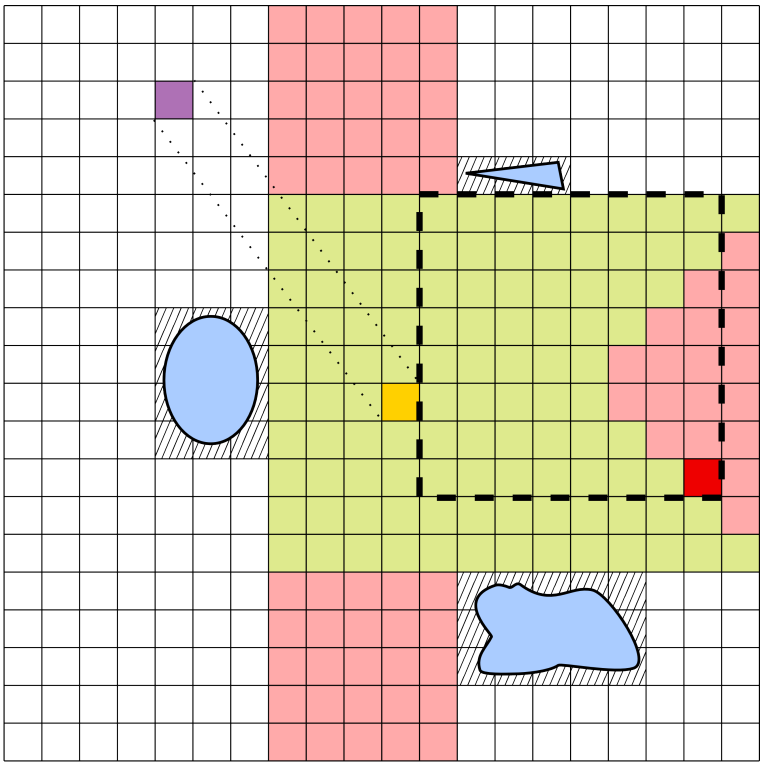

Figure 4.5 – Visualization of a Safe Neighborhood derived from AMR distance field in 2D. The safe neighborhood of the yellow voxel is depicted as the green region, however it could be extended to contain the pink voxels. An anisotropic macro-region of the red voxel is depicted to show, why that voxel was not included in the safe neighborhood. The violet voxel is an example of a voxel that could also be considered safe but is much harder to calculate than the pink voxels.

These cascade edges completely ignore the free cuboid property. The reason for the cascade edges is geometry that limits the free zone size, although the tested voxel lies in a different direction from the original voxel. This is illustrated in the Figure 4.5 by the red voxel and its safe zone marked by the slashed line.

But as can also be seen in the Figure 4.5, there are even more voxels that create a free bounding cuboid together with the original voxel. These voxels are contained in the union of the green and pink regions of the illustration. To find these voxels, I propose the following algorithm, although its implementation is out of scope of this work. The algorithm has to select a coordinate axis as its main axis. Here, the Y axis was chosen. Note that for the sake of clarity, the 2D version of the algorithm is described.

```
min_y // The y coordinate of the last free voxel when going from V in
      // the direction of the -Y axis
max_y // ditto, but in direction of +Y
min_x // ditto, but x coordinate and in the direction of -X
max_x // ditto, but in direction of +X

// Phase 1
min := min_x
max := max_x
for y from V_y to min_y do
  for x from V_x to min do
    if voxelIsFree(x, y) then
      markVoxel(x, y)
    else
      min := x + 1
      break
    endif
  done
  for x from V_x to max do
    if voxelIsFree(x, y) then
      markAsSafe(x, y)
    else
      max := x - 1
      break
    endif
  done
end

// Phase 2 is the same as phase 1, only in the other direction:
min := min_x
max := max_x
for y from V_y to max_y do
  // --||--
end
```

This algorithm scans the voxel lane $min_x \leftrightarrow max_x$ from V_y to min_x and then to max_y, updating the border voxels of the lane as it encounters occupied voxels. The extension to three dimensions would entail scanning a voxel plane instead of a lane and instead of updating two border points of the lane, there would be several border points along a chosen minor axis in the plane. The schematic of the function of the algorithm can be seen in the Figure 4.6.

Yet another step in the extension of the safe neighborhoods would be considering voxels similar to the violet voxel in the Figure 4.5. If an arbitrary point from the violet voxel is connected with any point from the yellow voxel, the line that connects them would surely be unoccluded. However, finding such voxels would be a non-trivial task.

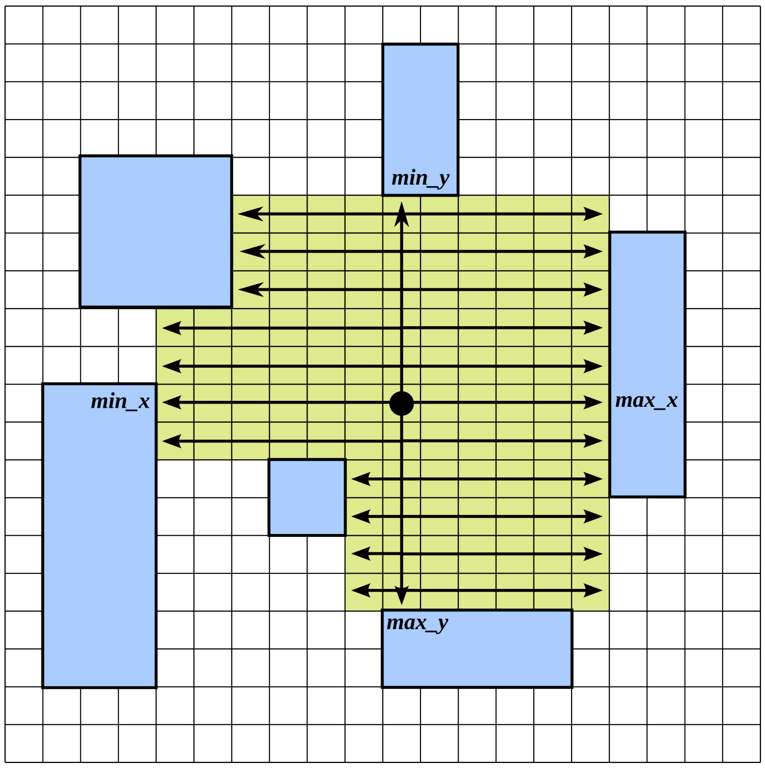

Figure 4.6 – Illustration of the algorithm for creation of the extended safe neighborhoods. The arrows depict the direction of exploration of free voxels by the algorithm. The resulting safe neighborhood is marked green.

Limitation of Safe Neighborhoods

One might rightfully ask, why we bothered with the distance fields in the first place, when the Safe Neighborhoods method together with the proposed extension accelerates a much larger proportion of visibility queries. However, safe neighborhoods contain visibility information only from a point of view of a single voxel in the voxel grid, whereas the distance fields allow us to accelerate visibility queries between arbitrary voxels.

If the visibility from a single voxel or a set of voxels (see the next section) is all that is needed, safe neighborhoods are a better choice than distance fields. Safe neighborhoods constitute an interesting area for future research, as well as methods for making them more effective, calculating them faster, finding a way to compress them or exploring their similarity to the well-known shadow maps algorithm.

Accelerating Area Light Queries

The algorithm for creating the extended safe neighborhoods can be slightly modified to allow area light acceleration. I will describe the modification very briefly here and without pseudo-code, as it is out of scope of this work. However, thorough description and exploration of the possibilities of the algorithm would be another area of possible future endeavors. Again, the described version here is only a 2D variant.

First, the area light would have to be rasterized using a conservative rasterization method. Then, the resulting set of voxels would be projected onto the two hyperplanes perpendicular to the axes X and Y and these projections would be swept along the axes in the direction of the normal vector of the area light (in our example, the area light's normal is pointing in the increasing direction of the X axis as well as the increasing direction of the Y axis. The intersection points of the projections with the geometry will mark the maximal extent of the safe neighborhood. One of the axes would then be chosen as the algorithm's main axis and again, we would find lanes of free voxels, while modifying the minimal and maximal coordinate of the lane.

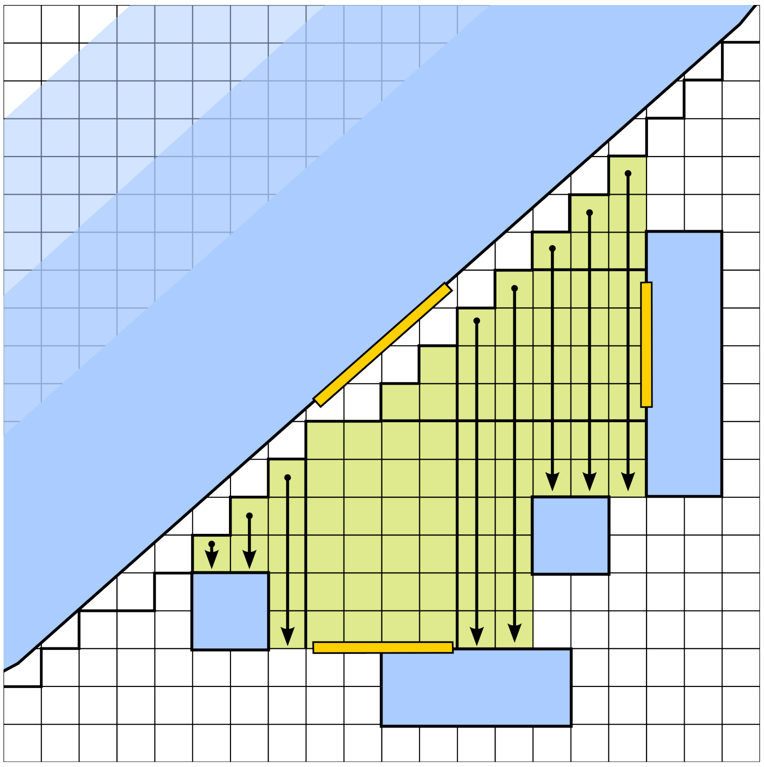

Figure 4.7 – Illustration of the algorithm for creation of safe neighborhoods for an area light. The area light and its projections are the yellow lines. The lanes of free voxels are scanned to the left and to the right of the area light and the visited voxels are added to the safe neighborhood (marked in green).

4.5 Implemented Directional Methods

From among the described directional distance field methods, I first implemented the Anisotropic Macro-Regions and later added the safe neighborhood extension.

The Anisotropic Macro-Regions were chosen because they use the chessboard metric instead of the Manhattan metric used by the Directed Safe Zones and Dual Extents. The chessboard metric is better suited for the rectangular features that are often found in architectural scenes. Another reason for choosing AMR over the other two methods was that its safe zones divide the space around the voxel into eight almost disjoint regions, which ensures that the distance field values are more independent from the geometry in irrelevant directions. In contrast, the Directed

Safe Zones divide the space around each voxel into six regions. These regions can be grouped into three pairs, such that the regions in each pair together form the whole space around the voxel. This implies a lot of unnecessary overlap of the regions. The shape of the Dual Extent safe zones is a bit more complicated to describe, but their directionality is probably better than in the AMRs. However, the safe zones in Dual Extents also overlap and their width is reduced by nearby geometry, even if it is located in a direction irrelevant to the query.

5. Implementation

5.1 Used Technology

The distance field methods were tested on the Mitsuba physically based renderer by Wenzel Jacob [20]. Mitsuba is written in C++ [18] and uses Mercurial [11] as its revision control system, so I had to use them as well if I wanted to modify the source code of the renderer.

I also used the Python programming language [13] for helper and debugging scripts. The open-source 3D modeling application Blender 3D [3] was used for scene editing as well as for debugging and distance field visualization purposes.

5.2 Implementation Details

Of the methods described in the Chapters 3 and 4, I implemented the non-directional distance field with the chessboard metric for distance field generation and chessboard metric on voxels for measuring the distance between the query points. Another implemented method are the directional Anisotropic Macro-Regions as well as their extension with the Safe Neighborhoods method.

The distance fields are implemented as C++ classes that are derived from a generic distance field base class, which provides the basic functionality that is common to all distance fields. However, polymorphism is not used, because virtual calls to critical distance field functions would slow down the execution of these functions. Instead, the used distance field class is chosen by C preprocessor directives at compile time.

The distance field is contained in the scene class and its acceleration methods are called in the `rayIntersect` method defined in the header file `scene.h`.

I could have associated a distance field with each of the participating media in the scene, but that would be unnecessarily complicated. Instead, I use a single distance field for the whole scene and thus have to make it large enough to contain all of the geometry. If the scene contains an environment emitter, the distance field has to contain its virtual geometry (that is not part of the scene) as well.

5.3 Debugging Features

In order to be able to debug the distance fields, I needed a way to visualize them. Visualizing the whole distance field as a three dimensional function would be difficult, so I chose another approach. After the distance field has been generated, I iterate over all of the voxels in the distance field and divide the voxels into two sets – free voxels and non-free voxels – each of these sets is converted into a list of linear coordinates of the voxels that it contains and this list is output into a file. Also, another file is created for each of the point lights in the scene. This file contains a list of linear coordinates of voxels from the safe neighborhood of the given point light source.

All of these files are created in the current working directory, but note that they are only created on the condition that the corresponding C preprocessor directive was uncommented in `config.py`, as will be described in the following section.

A python script was then created that converts these lists of linear indices into Wavefront `.obj` files[†].

This script first reconstructs the indices into a 3D array of voxels and then iterates over the voxels and creates their faces if their corresponding neighbors are not in the set. The resulting `.obj` files can then be imported and viewed in Blender 3D.

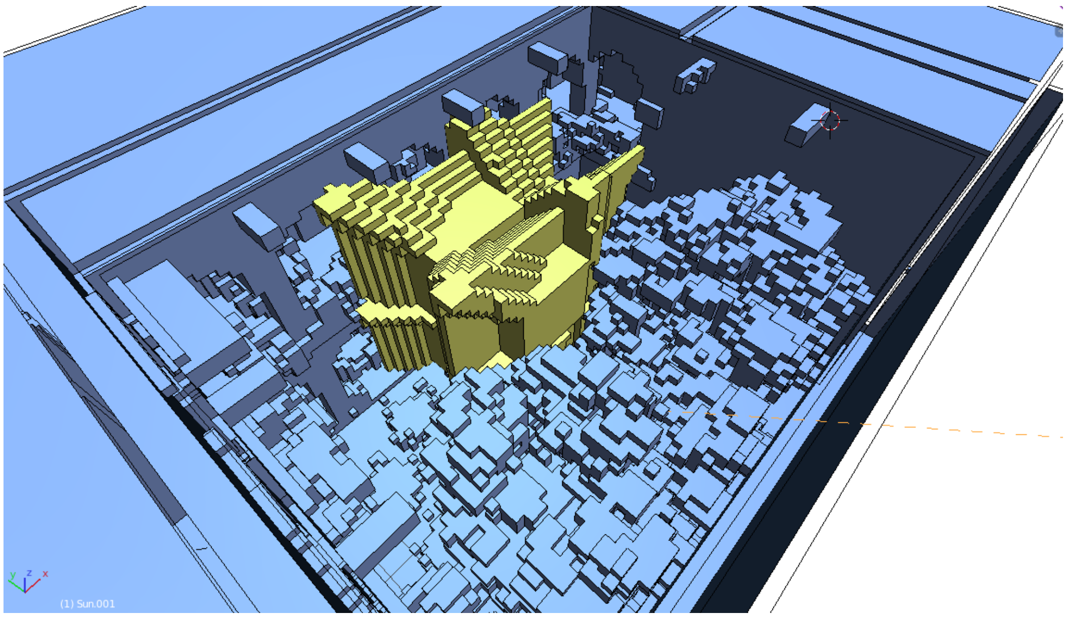

Figure 5.1 – An example of visualization of voxel sets in Blender 3D. The blue voxels are occupied voxels (note that some of them were removed in Blender so that the court in this scene would be visible), while the green voxel set is the safe neighborhood of the scene's point light.

5.4 Compilation and Running

The code of the Mitsuba renderer with my modifications can be found in the `mitsuba` directory.

For compilation, the SCons build system [16] was used. The original compilation configuration files for the Mitsuba renderer are contained in the directory `build`

I used the release configuration for 64-bit GNU/Linux operating system, renamed it as `config.py` and added several preprocessor directives that control

[†]This is a common open and widely supported format for 3D models.

which distance field methods and which debugging features will be compiled. The file contains comments that clarify the purpose of each of the directives.

The compilation itself is executed using the command `scons` After the compilation has finished, the executables can be found in the two directories named `build/release/mitsuba` and `build/release/mtsgui`.

The command-line interface can be run using the command `mitsuba` whereas the graphical interface is executed using the command `mtsgui`. For the thesis, I used the GUI, because it allows the rendering settings to be changed more easily.

On my testing system, I have an Intel embedded graphics chip and an NVidia external graphics card, whose cooperation is managed by the NVidia Optimus technology. Although the graphics cards were not used in my code, they were used by Mitsuba to visualize the scenes before the main rendering was started. This caused Mitsuba to crash on my system. My solution was to use programs from the Bumblebee Project [4] whose goal is to enable the use of the NVidia's proprietary Optimus technology on GNU/Linux. After installing the Bumblebee Project, Mitsuba can be run by typing `optirun mtsgui` and the visualization is then calculated using the NVidia card.

In `mtsgui`, the scene can be loaded by selecting `File` → `Open` from the main menu. The selected scene is then loaded into the memory and a k-d tree is built over the scene's geometry to enable logarithmic-time ray casting and visibility queries.

5.4.1 Scene Settings

After the scene has been loaded and preprocessed, the rendering parameters can be changed in the Render Settings dialog. This dialog is invoked by clicking the gear wheel icon in the top panel.

The most important settings are the length of the edge of a voxel or, in other words, voxel size, sample number per pixel and sample number per path segment.

The voxel size setting can be found in the section "Scene" in the list in the middle of the dialog. It is labeled "Distance field voxel size". By setting this parameter to zero, the distance field can be disabled entirely.

Sample number, labeled "Samples per pixel", can be found in the section that corresponds to the used sampler. In the testing scenes, this section will be "Independent sampler". This parameter sets the number of samples that are done for each pixel of the rendered image.

If the "Volumetric ray-marching path tracer" is selected as the integrator for the rendering, the number of samples per each path segment can be changed in the integrator's section in the parameter list.

If the visibility test time is measured, it is important to render with one thread only, because the timer used for the measurement will not perform correctly if manipulated by several threads. The number of worker threads can be set in the settings dialog, invoked from the main menu by selecting `Tools` → `Settings`.

5.4.2 Statistics and Measurements

During the render, the implemented distance field methods show the progress of building the distance field in Mitsuba's progress bar. The process of building the distance field is divided into two parts – initializing the distance field's voxels and propagating the values over the whole distance field – for the purposes of progress reporting.

After the render, there are three statistics that are shown by default: resolution of the voxel grid, size of the distance field in bytes and the distance field build time T_{build}. Also, Mitsuba always displays the render time T_{render}. These statistics show in the output to the log file, which can be displayed in Mitsuba's GUI by clicking the rightmost icon in the top panel of the window.

If the `DISTFIELD_STATS` directive has been defined in the compilation configuration file `config.py`, optional distance field statistics (together with other Mitsuba's statistics) can be displayed by clicking the pie chart icon in the top panel of the log file window. The statistics that are shown for the distance fields are percentage of free voxels in the distance field, percentage of spacial visibility tests p_{sp} and percentage of spatial tests that were accelerated $p_{acc|sp}$. If the `DISTFIELD_SNS` directive has been defined in the `config.py` file, the point light safe neighborhood statistics are shown as well – the number of the point light's safe neighborhood voxels and the ratio of these voxels to the number of free voxels in the scene.

The ray-cast timer can be turned on in `config.py` by defining `RAYCAST_TIMER`. If this is done, the statistics that were displayed by clicking the pie chart icon will now contain one more measurement – the spent ray-cast time. This is the time spent on visibility queries T_{query}.

6. Results

This chapter contains the results from measuring the performance of the distance field methods on the testing scenes.

6.1 Testing Scenes

To be able to test the implemented methods in a variety of environments, I created several testing scenes which were all encompassed in a participating medium. Most of them are just boxes with or without simple interior geometry. This choice was made because for testing and debugging purposes, simple geometry is much more suitable. However, I also included two scenes that are more complex, so that the methods could be tested on data that more closely resembles the real situations that the algorithms would be used in.

6.1.1 Cornell Box Scenes

The first four scenes (**cbox_vol2**, **cbox_vol3**, **cbox_vol4** and **cbox_vol6** – see Figures 6.1 through 6.4) are based on the existing Cornell Box scene that can be found in the Mitsuba renderer repository.

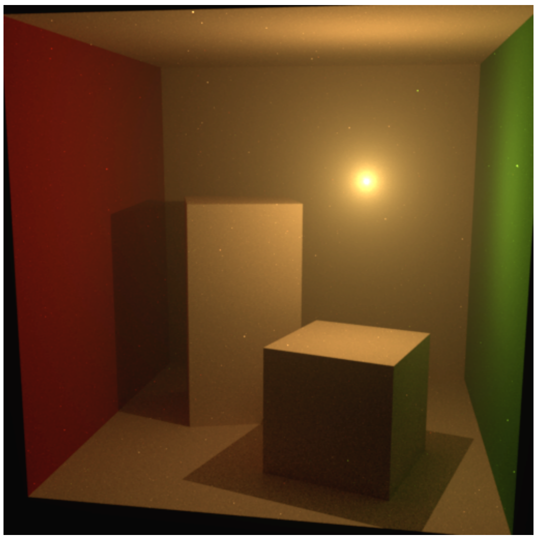

Figure 6.1 – Scene cbox_vol2

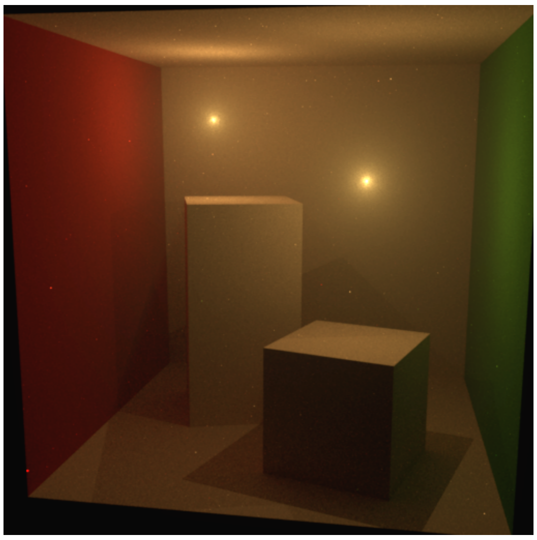

Figure 6.2 – Scene cbox_vol3

Scene **cbox_vol2** can be thought of as a base scene. It contains two cuboids in a box without its front face. The viewpoint is outside of the box, looking at the cuboids. The whole scene is submerged in a not very dense participating medium. There is a point light inside the box above one of the cuboids.

Scene **cbox_vol3** is a modification of the **cbox_vol2** scene. It contains two point lights instead of one.

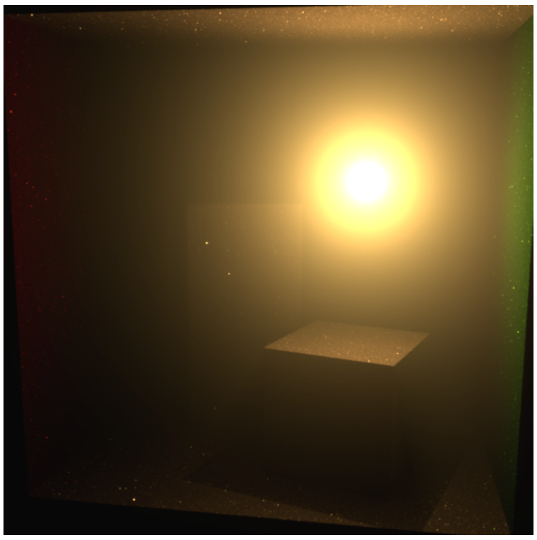
Figure 6.3 – Scene cbox_vol4

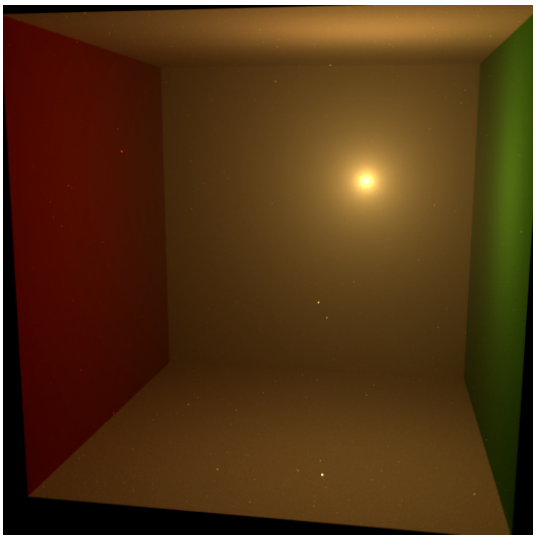
Figure 6.4 – Scene cbox_vol6

Scene **cbox_vol4** modifies **cbox_vol2** by making the medium ten times as dense. This improves p_{sp} (see Section 2.6) with the volumetric path tracer.

Scene **cbox_vol6** is the same as **cbox_vol2**, except that the box does not contain any geometry. This is useful for testing the distance field methods. Also, $p_{acc|sp}$ should be highest of all scenes.

6.1.2 San Miguel Scene

The **san-miguel** scene (the mesh was obtained from [6, McGuire Graphics Data]) contains a large number of triangles and the average time needed for a visibility test t_{query} is high. On the other hand, the scene is relatively rugged, so the percentage of accelerated visibility tests p_{acc} will be smaller than in the other scenes.

The scene is an oblong court with two stories of arcades around it. The court contains a lot of objects, most notably a large tree that occupies the center of the court. A point light was positioned inside the scene between the tree and the arcades, so as to ensure that the light was surrounded by some free space. See Figure 6.5 for a rendering of the scene.

6.1.3 Landscape Scene

The **scape** scene contains a moderate number of triangles, but a large empty space, ensuring high percentage of accelerated visibility tests p_{acc}. It consists of a simple landscape that I modeled in Blender 3D [3].

The light in this scene is provided by an environment emitter. Due to the Mitsuba's implementation of the environment emitters, the emitter is represented as a hemisphere surrounding the whole scene. However, the emitter's geometry is not registered in the distance field †, allowing to accelerate the visibility queries

†It is safe to do so, because the emitter's geometry cannot occlude any other geometry behind it – there is none.

even when one of the points of the query lies on the geometry of the emitter. Normally, such tests would be superficial and could not be accelerated, but I regard them as spatial.

The medium in this scene encompasses the landscape as well as the virtual geometry of the emitter. A rendering of the scene is pictured in Figure 6.6

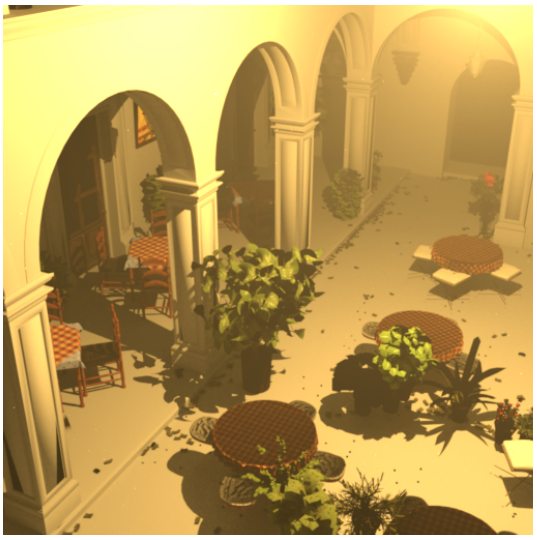
Figure 6.5 – Scene san-miguel

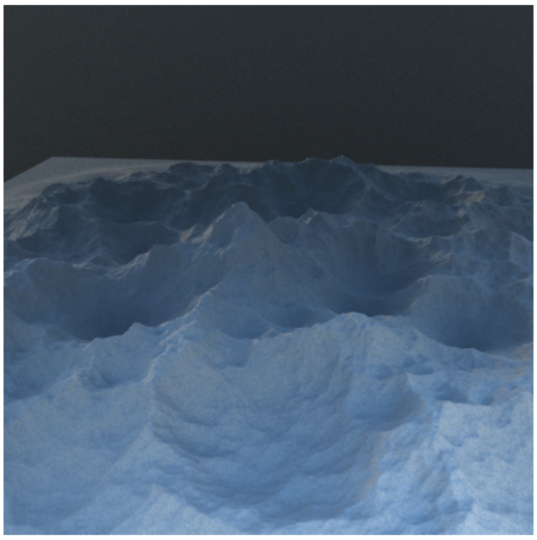
Figure 6.6 – Scene scape

6.2 Test Conditions

All of the results were rendered on a laptop PC with a GNU/Linux operating system, an Intel Core i7-3632QM quad core processor at 2.20GHz clock rate and with 8GB of the random access memory. The graphics processing unit was never used for the rendering.

When running the renderer to obtain the test results, all other unnecessary user processes were terminated in order to consistently provide the renderer with as much resources as possible and avoid context swaps and the influence of the system scheduler on the results. The renderer was also restricted to use only one thread in order for the distance field build times that were not implemented as multi-threaded to be comparable to the render times.

I experimented with giving the rendering process higher priority for the operation system scheduler with the `renice` command, but this did not make any difference in the execution times.

6.2.1 Variance of the Measured Times

In order to estimate, how much the measured times can differ from their average, several scenes with different settings were run multiple times and the relative standard deviation (calculation details in a course handout by Mark Iannone [8]) was estimated from the resulting data sets [†]. This statistical quantity is calculated

[†]The estimate was not a rigid statistical test, it was done in order to find out, whether the measurements are reliable.

Table 6.1 – Test results for consistency of measured times

cbox_vol2, $V_{size}/sps/spp : 4/4/4$
Raymarching vol. pathtracer

	T_{build}	T^{acc}_{render}	T^{acc}_{query}
	1.89	18.86	4.24
	1.89	18.70	4.19
	1.88	18.68	4.15
	1.87	18.64	4.15
	1.86	18.67	4.14
	1.88	18.79	4.20
	1.86	18.69	4.19
	1.89	18.70	4.18
RSD	0.68%	0.39%	0.79%

cbox_vol4, $V_{size}/spp : 4/16$
Simple volumetric pathtracer

	T_{build}	T^{acc}_{render}	T^{acc}_{query}
	1.89	47.25	7.53
	1.88	47.03	7.44
	1.88	47.10	7.45
	1.88	47.00	7.44
	1.88	47.00	7.46
	1.88	46.97	7.40
	1.88	46.94	7.47
	1.89	46.94	7.38
RSD	0.27%	0.22%	0.61%

Legend:

V_{size}	Voxel size (voxel edge length)
sps	Samples per path segment
spp	Samples per pixel
T_{build}	Time in seconds to build the distance field
T^{acc}_{render}	Render time with a distfield
T^{acc}_{query}	Time spent on visibility queries
RSD	Relative standard deviation

scape, $V_{size}/sps/spp : 0.25/8/16$
Raymarching vol. pathtracer

	T_{build}	T^{acc}_{render}	T^{acc}_{query}
	1.53	115.2	29.57
	1.67	114.6	29.23
	1.53	115.2	29.36
	1.54	114.6	29.19
	1.52	114.0	29.13
	1.67	114.6	29.08
	1.53	115.2	29.40
	1.53	114.6	29.28
RSD	4.16%	0.37%	0.54%

by dividing the standard deviation by the average of the data. The reason for using relative standard deviation instead of standard deviation or variance is that the relative standard deviation is a dimensionless quantity and can be easily compared between different data sets.

The relative standard deviation was calculated with the use of the GNU Octave programming language[5].

All the results were tested with the distance field statistics and time measurement turned on. The distance field used was an ordinary non-directional distance field. The T^{acc}_{query} time in the Table 6.1 is the time needed for all of the visibility queries. It is equal to $T^{acc}_{render} - T_{build} - T_{rest}$ (see Equation 2.6 in Section 2.6).

As can be seen from the results in the Table 6.1, the relative standard deviation does not, in most of the cases, exceed 1%. The only exception – build times in the scape scene – could have been an anomaly, because if we take a closer look at the data, most of the values oscillate very closely around 1.53s, except for two times, when it took 1.67s to build the distance field.

Because of the very low deviation of the time values from their mean, I decided

Table 6.2 – Test results for measurement influence

cbox_vol3, $V_{size}/sps/spp$: 8/8/8
Ray marching vol. path tracer

	T_{build}	T^{acc}_{render}	T^{acc}_{query}
w/ M.	0.24	55.96	14.61
w/o M.	0.26	52.63	——
Diff	+0.02	-3.33	——
RDiff	+8.3%	-6.0%	——

cbox_vol6, $V_{size}/sps/spp$: 8/2/16
Ray marching vol. path tracer

	T_{build}	T^{acc}_{render}	T^{acc}_{query}
w/ M.	0.24	39.28	5.86
w/o M.	0.24	37.02	——
Diff	0	-2.26	——
RDiff	0%	-5.8%	——

Legend:

w/ M.	With measurement
w/o M.	Without measurement
Diff	Difference
RDiff	Relative difference

Also see legend for Table 6.1

san-miguel, V_{size}/spp : 1/32
Simple volumetric path tracer

	T_{build}	T^{acc}_{render}	T^{acc}_{query}
w/ M.	3.83	128.4	33.08
w/o M.	3.82	124.2	——
Diff	-0.1	-4.2	——
RDiff	-0.26%	-3.3%	——

that for our purposes, it would suffice to make each time measurement only once. When interpreting the results, we have to bear in mind that the real average might differ slightly (but not as much as to completely skew the results).

6.2.2 Influence of Measurement on the Measured Times

The visibility query time measurement is done using a timer class from Mitsuba. A question is, whether the time measurement itself cannot significantly influence the measured times and in case it can, whether this influence could not be predicted and accounted for. In order to find out, I rendered several testing scenes with different parameters with and without the time measurement turned on and then compared the results.

From the results in the Table 6.2, it is obvious that the times indeed are influenced by turning the timer on. Furthermore, we cannot simply subtract a fixed number or a fixed percentage from the resulting measured times, because this would be incorrect. This means that in order to calculate the overhead of the timer, we have to run the rendering both with and without the timer and then take the difference of T_{render} as the timer overhead. Note that even with the visibility query timer turned off, T_{render} and T_{build} are still measured. This does not impose large time overhead on the algorithm, as it only requires starting a timer and stopping it twice (as opposed to N times when measuring visibility query times).

The problem is that due to the implementation of the timer, it does not start (or stop) counting time immediately after the function call, but there is a slight delay, which grows to noticeable proportions due to repetition. Because of this, we cannot simply subtract the total overhead (which we are now able to calculate, as was described in the previous paragraph) from the total measured time, since part of this overhead will not have been counted into the measured time. To solve this problem, I looked at the implementation of timers in Mitsuba.

The methods that start and stop the timer have a very similar structure and at their beginning they both call the same method, which returns the current system time. We now don't have to care at which point the latter method samples the system time, as long as this sampling event occurs at a consistent point in time of the method's execution. This is a safe assumption, especially in average over a lot of calls.

The previous discussion implies a way to calculate the real T_{query} from the measured T_{query} – in order to calculate the real value, we just have to subtract half of the overhead from the measured value, because the other half was not counted into the measurement.

6.3 Measured Times

6.3.1 Single Query Time

Let us now calculate the time needed for a single visibility query t_{query} for each of the scenes and then compare it to the acceleration structure query time t_{acc} for each of the distance fields to see, whether any acceleration is possible at all.

Recall from the Equation 2.5 in Section 2.6, that $T_{query} = N \cdot t_{query}$. As we already discussed, we cannot measure T_{query} accurately, but we can get T_{query} by subtracting half of the time measurement overhead from the measured T_{query}. In the Table 6.3, we can see the measured render times for four testing scenes with visibility query time measurement turned on and off. The parameters $V_{size}/sps/spp$ were chosen so that there would be a large number of the visibility queries, which ensures that we have enough data to average.

Table 6.3 – T_{render} times with visibility time measurement and without

Visibility time measurement ON

Scene	$V_{size}/sps/spp$	T_{render}	T_{query}
cbox_vol6	0/49/4	2.12 m	18.72 s
cbox_vol4	0/49/4	2.16 m	28.57 s
san-miguel	0/49/4	18.26 m	16.43 m
scape	0/49/4	1.68 m	18.87 s

Visibility time measurement OFF

Scene	$V_{size}/sps/spp$	T_{render}	T_{query}
cbox_vol6	0/49/4	1.92 m	——
cbox_vol4	0/49/4	1.98 m	——
san-miguel	0/49/4	17.67 m	——
scape	0/49/4	1.62 m	——

The average single visibility query time t_{query} can now be calculated for each of the scenes. The number of the visibility queries N does not depend on the query acceleration method and the values from Table 6.11 can thus be used. First we need to calculate the measuring overheads for all scenes:

$$\begin{aligned}
O_{cbox6} &= 2.12m - 1.92m = 12s \\
O_{cbox4} &= 2.16m - 1.98m = 10.8s \\
O_{smig} &= 18.26m - 17.67m = 35.4s \\
O_{scape} &= 1.68m - 1.62m = 3.6s
\end{aligned}$$

The real T_{query} is calculated by subtracting half of the overhead from the measured times.[†]

$$\begin{aligned}
\textbf{cbox\_vol6:}\quad & T_{query} = 18.72s - 6s = 12.72s \\
\textbf{cbox\_vol4:}\quad & T_{query} = 28.57s - 5.4s = 23.17s \\
\textbf{san-miguel:}\quad & T_{query} = 16.43m - 17.7s = 968.1s \\
\textbf{scape:}\quad & T_{query} = 18.87s - 1.8s = 17.07s
\end{aligned}$$

The final step is to divide the T_{query} times by the number of visibility queries.

$$\begin{aligned}
\textbf{cbox\_vol6:}\quad & t_{query} = \frac{12.72s}{170.53 \cdot 10^6} = 75ns \\
\textbf{cbox\_vol4:}\quad & t_{query} = \frac{23.17s}{165.05 \cdot 10^6} = 140ns \\
\textbf{san-miguel:}\quad & t_{query} = \frac{968.1s}{177.80 \cdot 10^6} = 5.4\mu s \\
\textbf{scape:}\quad & t_{query} = \frac{17.07s}{58.22 \cdot 10^6} = 293ns
\end{aligned}$$

We can see that the calculated average time of a single visibility query corresponds with the scene complexity, i.e. with the number of the scene's triangles – scene **san-miguel** contains by far the largest number of triangles and the visibility queries in this scene are also by far the most time consuming ones.

6.3.2 Acceleration Time

Let us now calculate the accelerated query time t_{acc}. As was already mentioned in Section 2.6 in the Equation 2.8, the ratio of times of t_{acc} and t_{query} for the given

[†]Formally the equations contradict themselves, but I chose to omit scene names from T_{query} for the sake of clarity. Otherwise, I should have written $T_{query,cbox6}$, $T_{query,cbox4}$, etc...

scene and distance field method defines the theoretical minimal percentage of the accelerated spatial queries $p_{acc|sp}$ that is needed for the acceleration to work, in the limit of N.

To calculate t_{acc}, we will use the Equation 2.6 and express T^{acc}_{query} from it:

$$T^{acc}_{query} = (1 - p_{acc}) \cdot N \cdot t_{query} + p_{sp} \cdot N \cdot t_{acc}$$

After a few modifications, we get an equation which can be used to calculate the accelerated query time t_{acc}:

$$t_{acc} = \frac{T^{acc}_{query} - (1 - p_{acc}) \cdot N \cdot t_{query}}{p_{sp} \cdot N}$$

Table 6.4 – Calculated t_{acc} times

Scene	Method	V_{size}	t_{acc}	t_{acc}/t_{query}
cbox_vol6	Non-directional DF	4	67 ns	89 %
		8	45 ns	60 %
		16	45 ns	60 %
	Anisotropic Macro-Regions	4	86 ns	115 %
		8	68 ns	91 %
		16	44 ns	59 %
	AMR with Safe Neighborhoods	4	16 ns	21 %
		8	14 ns	19 %
		16	14 ns	19 %
cbox_vol4	Non-directional DF	4	75 ns	54 %
		8	67 ns	48 %
		16	76 ns	54 %
	Anisotropic Macro-Regions	4	85 ns	61 %
		8	69 ns	49 %
		16	55 ns	39 %
	AMR with Safe Neighborhoods	4	29 ns	21 %
		8	25 ns	18 %
		16	23 ns	16 %
scape	Non-directional DF	0.25	189 ns	65 %
		0.5	177 ns	60 %
		1	84 ns	29 %
	Anisotropic Macro-Regions	0.25	227 ns	78 %
		0.5	182 ns	62 %
		1	152 ns	52 %

To calculate t_{acc} for various testing scenes, I used the data from the tests with the ray marcher with 49 samples per path segment from the Tables 6.5 – 6.10. The time needed for the accelerated visibility queries T^{acc}_{query} was calculated by subtracting half of the measurement timer overhead from the measured T^{acc}_{query} (as was described in Section 6.2.2). The probability p_{acc} equals to $p_{sp} \cdot p_{acc|sp}$, N can be found in the Table 6.11 and the values of t_{query} are known from the previous calculations.

The Table 6.4 contains the resulting accelerated times for three of the testing scenes. The times for **san-miguel** scene could not be calculated due to inaccuracies in the measurements.

We can see that the Anisotropic Macro-Regions with the Safe Neighborhood modification perform much better than the other two methods. That was expected since the Safe Neighborhoods only retrieve a boolean value from an array, whereas the other two methods calculate distances and compare them to the values in the distance field.

Another interesting observation is that the acceleration query time decreases with the increasing voxel size of the distance field (which decreases the total voxel number). This is probably due to the various caches in the CPU – the larger voxel size means a smaller distance field and less frequent cache misses. Figure 6.7 shows this dependency in a chart.

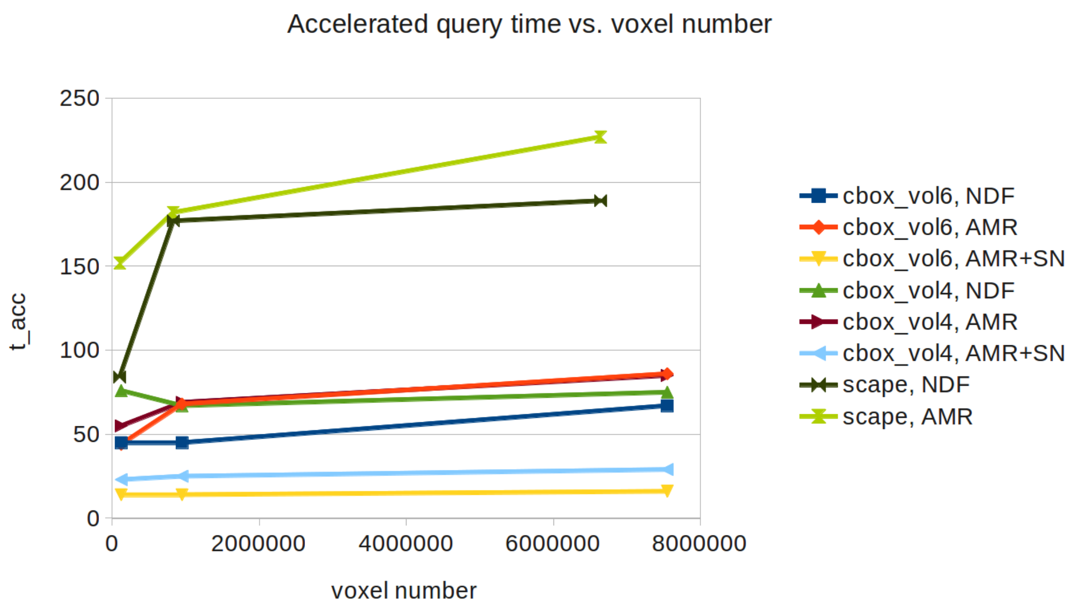

Figure 6.7 – Accelerated query time vs. voxel number

The larger query times in the scene **scape** are also interesting because the distance field dimensions are comparable to the dimensions in the other scenes. This can be a consequence of the incoherence of the visibility queries. In all of the other scenes, one of the points is often (if not every time) the position of the scene's point light source. The **scape** scene, on the other hand, contains an environment map emitter, whose emitted light needs to be sampled in various points, totally destroying the query coherence and making the CPU's caching strategies less efficient.

According to the Criterion 2.8, it will be very difficult to accelerate the visibility queries with non-directional distance fields or the Anisotropic Macro-Regions. However, we must bear in mind that the percentages in the Table 6.4 reflect the fact that even the original queries were quite fast. Adding more triangles would lower the ratios and create more opportunity for acceleration.

6.3.3 Acceleration Results

This section contains the complete visibility acceleration test measurements. First, the testing scenes were rendered without any acceleration for comparison. Then, each of the distance field methods was tested with three different voxel size V_{size} settings and four different settings for the number of samples per path segment sps. The setting "VPT" of the parameter sps means that the original volumetric path tracer was used and this parameter does not make sense.

The probabilities p_{sp} were measured only once per setting of the sps parameter, because they don't depend on any other parameter. Likewise, the distance field build time T_{build}, the distance field dimensions and the probabilities $p_{acc|sp}$ didn't have to be calculated for every setting, because their values are identical for several parameter settings.

The remaining quantities, T_{render} and T_{query} had to be measured separately for every combination of the parameters. Note that the T_{render} column actually contains T^{acc}_{render} where an acceleration method is used. Similarly, the T_{query} column contains T^{acc}_{query} values if a distance field is used.

Table 6.5 – Acceleration results 1/6

Non-directional DF, cbox_vol6

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	7.43 s	828 ms	———	54.34 %	———	———
4/1/4	9.59 s	1.10 s	1.87 s	54.34 %	$153 \times 146 \times 338$	7.42 %
8/1/4	7.85 s	1.02 s	0.24 s	54.34 %	$77 \times 73 \times 169$	6.87 %
16/1/4	7.63 s	982 ms	41 ms	54.34 %	$39 \times 37 \times 85$	5.13 %
0/7/4	22.68 s	3.16 s	———	89.28 %	———	———
4/7/4	26.25 s	4.84 s	1.87 s	89.28 %	$153 \times 146 \times 338$	7.42 %
8/7/4	24.04 s	4.27 s	0.24 s	89.28 %	$77 \times 73 \times 169$	6.87 %
16/7/4	23.58 s	4.12 s	41 ms	89.28 %	$39 \times 37 \times 85$	5.13 %
0/49/4	2.12 m	18.72 s	———	98.31 %	———	———
4/49/4	2.33 m	28.99 s	1.87 s	98.31 %	$153 \times 146 \times 338$	7.42 %
8/49/4	2.24 m	25.48 s	0.24 s	98.31 %	$77 \times 73 \times 169$	6.87 %
16/49/4	2.28 m	25.62 s	41 ms	98.31 %	$39 \times 37 \times 85$	5.13 %
0/VPT/4	5.75 s	487 ms	———	31.18 %	———	———
4/VPT/4	7.71 s	599 ms	1.87 s	31.18 %	$153 \times 146 \times 338$	11.84 %
8/VPT/4	6.04 s	556 ms	0.24 s	31.18 %	$77 \times 73 \times 169$	11.04 %
16/VPT/4	5.84 s	534 ms	41 ms	31.18 %	$39 \times 37 \times 85$	8.38 %

Anisotropic Macro-Regions, cbox_vol6

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	7.43 s	828 ms	———	54.34 %	———	———
4/1/4	7.30 s	922 ms	2.95 s	54.34 %	$153 \times 146 \times 338$	73.77 %
8/1/4	7.62 s	858 ms	0.41 s	54.34 %	$77 \times 73 \times 169$	68.11 %
16/1/4	7.22 s	783 ms	59 ms	54.34 %	$39 \times 37 \times 85$	59.37 %
0/7/4	22.68 s	3.16 s	———	89.28 %	———	———
4/7/4	26.34 s	4.08 s	2.95 s	89.28 %	$153 \times 146 \times 338$	73.77 %
8/7/4	23.42 s	3.65 s	0.41 s	89.28 %	$77 \times 73 \times 169$	68.11 %
16/7/4	22.52 s	3.25 s	59 ms	89.28 %	$39 \times 37 \times 85$	59.37 %
0/49/4	2.12 m	18.72 s	———	98.31 %	———	———
4/49/4	2.28 m	23.84 s	2.95 s	98.31 %	$153 \times 146 \times 338$	73.77 %
8/49/4	2.23 m	21.62 s	0.41 s	98.31 %	$77 \times 73 \times 169$	68.11 %
16/49/4	2.11 m	18.64 s	59 ms	98.31 %	$39 \times 37 \times 85$	59.37 %
0/VPT/4	5.75 s	487 ms	———	31.18 %	———	———
4/VPT/4	8.70 s	525 ms	2.95 s	31.18 %	$153 \times 146 \times 338$	81.42 %
8/VPT/4	6.06 s	501 ms	0.41 s	31.18 %	$77 \times 73 \times 169$	78.34 %
16/VPT/4	5.68 s	475 ms	59 ms	31.18 %	$39 \times 37 \times 85$	72.18 %

Table 6.6 – Acceleration results 2/6

Anisotropic Macro-Regions with Safe Neighborhoods, cbox_vol6

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	7.43 s	828 ms	——	54.34 %	——	——
4/1/4	9.89 s	585 ms	3.09 s	54.34 %	153 × 146 × 338	73.75 %
8/1/4	7.23 s	579 ms	0.41 s	54.34 %	77 × 73 × 169	68.18 %
16/1/4	6.96 s	608 ms	60 ms	54.34 %	39 × 37 × 85	59.44 %
0/7/4	22.68 s	3.16 s	——	89.28 %	——	——
4/7/4	23.98 s	2.15 s	3.09 s	89.28 %	153 × 146 × 338	73.75 %
8/7/4	21.45 s	2.19 s	0.41 s	89.28 %	77 × 73 × 169	68.18 %
16/7/4	21.20 s	2.33 s	60 ms	89.28 %	39 × 37 × 85	59.44 %
0/49/4	2.12 m	18.72 s	——	98.31 %	——	——
4/49/4	2.01 m	12.22 s	3.09 s	98.31 %	153 × 146 × 338	73.75 %
8/49/4	1.97 m	12.59 s	0.41 s	98.31 %	77 × 73 × 169	68.18 %
16/49/4	1.98 m	13.60 s	60 ms	98.31 %	39 × 37 × 85	59.44 %
0/VPT/4	5.75 s	487 ms	——	31.18 %	——	——
4/VPT/4	8.54 s	395 ms	3.09 s	31.18 %	153 × 146 × 338	81.40 %
8/VPT/4	5.88 s	383 ms	0.41 s	31.18 %	77 × 73 × 169	78.41 %
16/VPT/4	5.52 s	387 ms	60 ms	31.18 %	39 × 37 × 85	72.25 %

Non-directional DF, cbox_vol4

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	8.05 s	1.18 s	——	60.51 %	——	——
4/1/4	10.21 s	1.45 s	1.95 s	60.51 %	153 × 146 × 338	2.14 %
8/1/4	8.53 s	1.38 s	0.25 s	60.51 %	77 × 73 × 169	1.91 %
16/1/4	8.41 s	1.39 s	41 ms	60.51 %	39 × 37 × 85	1.51 %
0/7/4	23.59 s	4.80 s	——	91.47 %	——	——
4/7/4	27.17 s	6.44 s	1.95 s	91.47 %	153 × 146 × 338	2.14 %
8/7/4	25.17 s	6.10 s	0.25 s	91.47 %	77 × 73 × 169	1.91 %
16/7/4	24.93 s	6.06 s	41 ms	91.47 %	39 × 37 × 85	1.51 %
0/49/4	2.16 m	28.57 s	——	98.69 %	——	——
4/49/4	2.46 m	40.22 s	1.95 s	98.69 %	153 × 146 × 338	2.14 %
8/49/4	2.44 m	39.02 s	0.25 s	98.69 %	77 × 73 × 169	1.91 %
16/49/4	2.54 m	40.67 s	41 ms	98.69 %	39 × 37 × 85	1.51 %
0/VPT/4	11.18 s	1.39 s	——	81.71 %	——	——
4/VPT/4	13.45 s	1.86 s	1.95 s	81.71 %	153 × 146 × 338	6.97 %
8/VPT/4	11.69 s	1.71 s	0.25 s	81.71 %	77 × 73 × 169	6.32 %
16/VPT/4	11.42 s	1.66 s	41 ms	81.71 %	39 × 37 × 85	5.13 %

Table 6.7 – Acceleration results 3/6

Anisotropic Macro-Regions, cbox_vol4

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	8.05 s	1.18 s	——	60.51 %	——	——
4/1/4	12.96 s	1.65 s	3.12 s	60.51 %	$153 \times 146 \times 338$	20.65 %
8/1/4	8.43 s	1.28 s	0.41 s	60.51 %	$77 \times 73 \times 169$	19.31 %
16/1/4	8.02 s	1.23 s	58 ms	60.51 %	$39 \times 37 \times 85$	15.37 %
0/7/4	23.59 s	4.80 s	——	91.47 %	——	——
4/7/4	28.23 s	6.38 s	3.12 s	91.47 %	$153 \times 146 \times 338$	20.65 %
8/7/4	25.18 s	5.89 s	0.41 s	91.47 %	$77 \times 73 \times 169$	19.31 %
16/7/4	24.43 s	5.63 s	58 ms	91.47 %	$39 \times 37 \times 85$	15.37 %
0/49/4	2.16 m	28.57 s	——	98.69 %	——	——
4/49/4	2.37 m	37.65 s	3.12 s	98.69 %	$153 \times 146 \times 338$	20.65 %
8/49/4	2.28 m	35.09 s	0.41 s	98.69 %	$77 \times 73 \times 169$	19.31 %
16/49/4	2.26 m	34.00 s	58 ms	98.69 %	$39 \times 37 \times 85$	15.37 %
0/VPT/4	11.18 s	1.39 s	——	81.71 %	——	——
4/VPT/4	14.50 s	1.83 s	3.12 s	81.71 %	$153 \times 146 \times 338$	35.48 %
8/VPT/4	11.75 s	1.67 s	0.41 s	81.71 %	$77 \times 73 \times 169$	34.70 %
16/VPT/4	11.27 s	1.54 s	58 ms	81.71 %	$39 \times 37 \times 85$	30.46 %

Anisotropic Macro-Regions with Safe Neighborhoods, cbox_vol4

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	8.05 s	1.18 s	——	60.51 %	——	——
4/1/4	10.67 s	1.07 s	3.17 s	60.51 %	$153 \times 146 \times 338$	20.64 %
8/1/4	8.01 s	1.05 s	0.41 s	60.51 %	$77 \times 73 \times 169$	19.31 %
16/1/4	7.67 s	1.07 s	64 ms	60.51 %	$39 \times 37 \times 85$	15.37 %
0/7/4	23.59 s	4.80 s	——	91.47 %	——	——
4/7/4	25.90 s	4.70 s	3.17 s	91.47 %	$153 \times 146 \times 338$	20.64 %
8/7/4	23.34 s	4.63 s	0.41 s	91.47 %	$77 \times 73 \times 169$	19.31 %
16/7/4	22.85 s	4.70 s	64 ms	91.47 %	$39 \times 37 \times 85$	15.37 %
0/49/4	2.16 m	28.57 s	——	98.69 %	——	——
4/49/4	2.16 m	28.51 s	3.17 s	98.69 %	$153 \times 146 \times 338$	20.64 %
8/49/4	2.11 m	28.23 s	0.41 s	98.69 %	$77 \times 73 \times 169$	19.31 %
16/49/4	2.11 m	28.82 s	64 ms	98.69 %	$39 \times 37 \times 85$	15.37 %
0/VPT/4	11.18 s	1.39 s	——	81.71 %	——	——
4/VPT/4	13.67 s	1.26 s	3.17 s	81.71 %	$153 \times 146 \times 338$	35.48 %
8/VPT/4	10.99 s	1.24 s	0.41 s	81.71 %	$77 \times 73 \times 169$	34.70 %
16/VPT/4	10.61 s	1.26 s	64 ms	81.71 %	$39 \times 37 \times 85$	30.45 %

Table 6.8 – Acceleration results 4/6

Non-directional DF, san-miguel

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	45.99 s	29.37 s	——	72.80 %	——	——
0.25/1/4	1.58 m	29.02 s	49.96 s	72.80 %	277 × 63 × 109	0.75 %
0.5/1/4	1.07 m	32.55 s	13.01 s	72.80 %	139 × 32 × 55	0.47 %
1/1/4	51.52 s	30.40 s	3.87 s	72.80 %	70 × 16 × 28	0.12 %
0/7/4	3.15 m	2.64 m	——	94.93 %	——	——
0.25/7/4	3.83 m	2.54 m	49.96 s	94.93 %	277 × 63 × 109	0.75 %
0.5/7/4	3.29 m	2.59 m	13.01 s	94.93 %	139 × 32 × 55	0.47 %
1/7/4	3.27 m	2.70 m	3.87 s	94.93 %	70 × 16 × 28	0.12 %
0/49/4	18.26 m	16.43 m	——	99.24 %	——	——
0.25/49/4	18.92 m	16.26 m	49.96 s	99.24 %	277 × 63 × 109	0.75 %
0.5/49/4	18.41 m	16.25 m	13.01 s	99.24 %	139 × 32 × 55	0.47 %
1/49/4	18.17 m	16.25 m	3.87 s	99.24 %	70 × 16 × 28	0.12 %
0/VPT/4	16.42 s	4.37 m	——	6.71 %	——	——
0.25/VPT/4	1.09 m	4.31 m	48.75 s	6.71 %	277 × 63 × 109	1.60 %
0.5/VPT/4	31.59 s	4.65 m	13.10 s	6.71 %	139 × 32 × 55	1.01 %
1/VPT/4	20.60 s	4.45 m	3.76 s	6.71 %	70 × 16 × 28	0.24 %

Anisotropic Macro-Regions, san-miguel

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	45.99 s	29.37 s	——	72.80 %	——	——
0.25/1/4	55.33 s	28.98 s	10.36 s	72.80 %	277 × 63 × 109	5.95 %
0.5/1/4	54.46 s	28.98 s	9.50 s	72.80 %	139 × 32 × 55	4.96 %
1/1/4	54.22 s	28.85 s	9.38 s	72.80 %	70 × 16 × 28	4.25 %
0/7/4	3.15 m	2.64 m	——	94.93 %	——	——
0.25/7/4	3.23 m	2.59 m	10.36 s	94.93 %	277 × 63 × 109	5.95 %
0.5/7/4	3.22 m	2.59 m	9.50 s	94.93 %	139 × 32 × 55	4.96 %
1/7/4	3.21 m	2.58 m	9.38 s	94.93 %	70 × 16 × 28	4.25 %
0/49/4	18.26 m	16.43 m	——	99.24 %	——	——
0.25/49/4	18.73 m	16.70 m	10.36 s	99.24 %	277 × 63 × 109	5.95 %
0.5/49/4	18.67 m	16.65 m	9.50 s	99.24 %	139 × 32 × 55	4.96 %
1/49/4	18.62 m	16.62 m	9.38 s	99.24 %	70 × 16 × 28	4.25 %
0/VPT/4	16.42 s	4.37 s	——	6.71 %	——	——
0.25/VPT/4	26.23 s	4.26 s	10.28 s	6.71 %	277 × 63 × 109	11.27 %
0.5/VPT/4	26.08 s	4.34 s	9.53 s	6.71 %	139 × 32 × 55	9.46 %
1/VPT/4	25.92 s	4.32 s	9.36 s	6.71 %	70 × 16 × 28	8.32 %

Table 6.9 – Acceleration results 5/6

Anisotropic Macro-Regions with Safe Neighborhoods, san-miguel

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	45.99 s	29.37 s	——	72.80 %	——	——
0.25/1/4	55.57 s	29.35 s	10.48 s	72.80 %	277 × 63 × 109	5.97 %
0.5/1/4	55.03 s	29.18 s	9.40 s	72.80 %	139 × 32 × 55	4.98 %
1/1/4	54.50 s	29.16 s	9.52 s	72.80 %	70 × 16 × 28	4.23 %
0/7/4	3.15 m	2.64 m	——	94.93 %	——	——
0.25/7/4	3.22 m	2.58 m	10.48 s	94.93 %	277 × 63 × 109	5.97 %
0.5/7/4	3.18 m	2.56 m	9.40 s	94.93 %	139 × 32 × 55	4.98 %
1/7/4	3.18 m	2.56 m	9.52 s	94.93 %	70 × 16 × 28	4.23 %
0/49/4	18.26 m	16.43 m	——	99.24 %	——	——
0.25/49/4	18.90 m	16.87 m	10.48 s	99.24 %	277 × 63 × 109	5.97 %
0.5/49/4	18.39 m	16.39 m	9.40 s	99.24 %	139 × 32 × 55	4.98 %
1/49/4	18.39 m	16.40 m	9.50 s	99.24 %	70 × 16 × 28	4.23 %
0/VPT/4	16.42 s	4.37 s	——	6.71 %	——	——
0.25/VPT/4	26.15 s	4.25 s	10.60 s	6.71 %	277 × 63 × 109	11.30 %
0.5/VPT/4	25.27 s	4.23 s	9.72 s	6.71 %	139 × 32 × 55	9.48 %
1/VPT/4	25.27 s	4.23 s	9.56 s	6.71 %	70 × 16 × 28	8.28 %

Non-directional DF, scape

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	7.88 s	1.17 s	——	52.65 %	——	——
0.25/1/4	10.41 s	1.38 s	1.91 s	52.65 %	188 × 188 × 188	6.12 %
0.5/1/4	8.92 s	1.35 s	0.40 s	52.65 %	94 × 94 × 94	5.48 %
1/1/4	8.77 s	1.32 s	0.22 s	52.65 %	47 × 47 × 47	3.71 %
0/7/4	19.86 s	3.49 s	——	88.63 %	——	——
0.25/7/4	23.73 s	4.60 s	1.91 s	88.63 %	188 × 188 × 188	6.12 %
0.5/7/4	21.87 s	4.40 s	0.40 s	88.63 %	94 × 94 × 94	5.48 %
1/7/4	21.49 s	4.16 s	0.22 s	88.63 %	47 × 47 × 47	3.71 %
0/49/4	1.68 m	18.87 s	——	98.20 %	——	——
0.25/49/4	2.08 m	28.66 s	1.91 s	98.20 %	188 × 188 × 188	6.12 %
0.5/49/4	2.07 m	28.06 s	0.40 s	98.20 %	94 × 94 × 94	5.48 %
1/49/4	1.79 m	23.07 s	0.22 s	98.20 %	47 × 47 × 47	3.71 %
0/VPT/4	7.47 s	1.56 s	——	63.58 %	——	——
0.25/VPT/4	9.88 s	1.79 s	1.89 s	63.58 %	188 × 188 × 188	12.10 %
0.5/VPT/4	8.38 s	1.76 s	0.40 s	63.58 %	94 × 94 × 94	11.51 %
1/VPT/4	8.16 s	1.73 s	0.22 s	63.58 %	47 × 47 × 47	9.88 %

Table 6.10 – Acceleration results 6/6

Anisotropic Macro-Regions, scape

$V_{size}/sps/spp$	T_{render}	T_{query}	T_{build}	p_{sp}	Dimensions	$p_{acc\|sp}$
0/1/4	7.88 s	1.17 s	——	52.65 %	——	——
0.25/1/4	10.11 s	505 ms	4.98 s	52.65 %	$188 \times 188 \times 188$	19.30 %
0.5/1/4	7.76 s	506 ms	2.72 s	52.65 %	$94 \times 94 \times 94$	18.97 %
1/1/4	7.43 s	505 ms	2.40 s	52.65 %	$47 \times 47 \times 47$	17.89 %
0/7/4	19.86 s	3.49 s	——	88.63 %	——	——
0.25/7/4	22.76 s	4.42 s	4.98 s	88.63 %	$188 \times 188 \times 188$	19.30 %
0.5/7/4	21.16 s	4.25 s	2.72 s	88.63 %	$94 \times 94 \times 94$	18.97 %
1/7/4	20.72 s	3.99 s	2.40 s	88.63 %	$47 \times 47 \times 47$	17.89 %
0/49/4	1.68 m	18.87 s	——	98.20 %	——	——
0.25/49/4	1.97 m	28.59 s	4.98 s	98.20 %	$188 \times 188 \times 188$	19.30 %
0.5/49/4	1.88 m	26.09 s	2.72 s	98.20 %	$94 \times 94 \times 94$	18.97 %
1/49/4	1.85 m	24.53 s	2.40 s	98.20 %	$47 \times 47 \times 47$	17.89 %
0/VPT/4	7.47 s	1.56 s	——	63.58 %	——	——
0.25/VPT/4	12.99 s	1.86 s	5.00 s	63.58 %	$188 \times 188 \times 188$	23.79 %
0.5/VPT/4	10.68 s	1.81 s	2.72 s	63.58 %	$94 \times 94 \times 94$	23.43 %
1/VPT/4	10.34 s	1.77 s	2.42 s	63.58 %	$47 \times 47 \times 47$	22.29 %

Table 6.11 – Visibility query statistics

Scene	$V_{size}/sps/spp$	N	p_{sp}
cbox_vol6	8/1/4	6,300,000	54.34 %
	8/7/4	26,820,000	89.28 %
	8/49/4	170,530,000	98.31 %
	8/VPT/4	3,670,000	31.18 %
cbox_vol4	8/1/4	5,490,000	60.51 %
	8/7/4	25,440,000	91.47 %
	8/49/4	165,050,000	98.69 %
	8/VPT/4	6,970,000	81.71 %
san-miguel	8/1/4	4,940,000	72.80 %
	8/7/4	26,560,000	94.93 %
	8/49/4	177,800,000	99.24 %
	8/VPT/4	3,530,000	6.71 %
scape	8/1/4	2,220,000	52.65 %
	8/7/4	9,220,000	88.63 %
	8/49/4	58,220,000	98.20 %
	8/VPT/4	1,920,000	63.58 %

6.3.4 Discussion

As can be seen from the results of the tests in the Tables 6.5 – 6.10, the probability of a given visibility query being spatial p_{sp} increases with the number of samples per light path segment, which should not be surprising. Also, the probability of a spatial test being accelerated $p_{acc|sp}$ increases slightly with the voxel grid resolution. This is because a finer grid approximates the continuous distance function more precisely.

As for the visibility query times, their dependence on the voxel grid resolution can be described by three major influences. First, the distance field build time increases with the increasing resolution, up to by a factor of 2^3, because that many more voxels have to be visited during the propagation of the distance values. The time of a single query also increases with the increase of the voxel grid size, as was already discussed in Section 6.3.2.

A factor that works against the build times and query times is the decreasing of the probability $p_{acc|sp}$ with the increasing voxel size. In most of the scenes, the first two factors dominate the query time, but for example, in the scene **cbox_vol6** with Anisotropic Macro-Regions with the Safe Neighborhoods, the differences in the probabilities $p_{acc|sp}$ are large enough for the visibility query time to actually increase with the decreasing resolution.

If the distance field methods were to be used in a real setting, it would be necessary to estimate the grid size with respect to the factors that were just described, so as to ensure that the visibility query time would be as small as possible.

In most of the scenes, the render times with the distance field are larger than without it, with the notable exception of the Anisotropic Macro-Regions with the Safe Neighborhoods in scenes **cbox_vol6** and **cbox_vol4**. However, as I showed in Section 6.3.2, in most cases the time spent for the accelerated query is smaller than the time for a standard query, even for very simple scenes like **cbox_vol6** and **cbox_vol4**. Therefore, the distance field methods can be expected to work much better with complex scenes. The problem with my testing complex scene **san-miguel** is that the free space in the scene is divided by columns, furniture and vegetation to the extent that the probability $p_{acc|sp}$ drops to around 5%.

The scenes that will see the largest benefit from the distance field acceleration methods will thus be scenes with a very large number of triangles and large regions of free space. Because of the Safe Neighborhoods method, scenes with point light sources will also benefit from the distance field acceleration much more than other scenes.

7. Conclusion

In this thesis, I explored a number of existing methods for creation of distance functions or distance fields, both with and without directional information. The objective was to investigate, whether any of them could be used for acceleration of visibility queries in participating media. I decided that if the acceleration were to be as fast as possible, the only choice was to use the discrete grid-based distance field methods.

Initially, I chose two methods, a non-directional distance field with the chessboard voxel metric and directional Anisotropic Macro-Regions. The methods were implemented into the Mitsuba open-source renderer and tested with the existing integrators, mainly the volumetric path tracer. I also implemented a ray marching single scattering volumetric path tracer, so that I could test the methods on an integrator that had a higher percentage of spatial visibility queries.

A debugging feature was also implemented that enabled me to visualize sets of voxels from the distance field methods in three dimensions. The study of the visualizations led to the creation of the Safe Neighborhoods method that I also implemented and tested together with the previously implemented methods.

7.1 Future Work

Due to the large difference in the performance of the directional Anisotropic Macro-Regions and a non-directional distance field, I am forced to conclude that if any method is to succeed in acceleration of visibility queries, it has to make use of the directional information. I thus suggest that the eventual future research concentrates on directional methods.

The results have shown a high dependency of the acceleration potential of the implemented methods on the geometry of the scene. For example, the addition of two boxes to the scene **cbox_vol4** in comparison to the empty **cbox_vol6** resulted in the decrease of accelerated spatial visibility tests from around 70% to slightly above 20%. Also, the **san_miguel** scene, which has a very complex geometry, has only around 5% of accelerated spatial queries. Also, the acceleration depends on a variety of other factors such as the number of geometrical primitives in the scene, the used integrator, etc. It would be interesting to quantify this dependency so as to be able to decide, in which cases to use the distance field.

The most promising method for visibility query acceleration are the Safe Neighborhoods, both because of their speed and percentage of accelerated queries. As was shown in the Chapter 4, with some modifications, they could be even used to accelerate visibility queries to area lights and it would be interesting to explore these modifications more thoroughly and implement and test them.

Future research might also focus on finding a compete set of voxels that are unoccluded when viewed from any point of a given voxel. Similarly, one could strive to find completely occluded voxels, which would further accelerate the queries by being able to provide an immediate answer not only for certainly unoccluded visibility queries, but now also for the certainly occluded ones.

References

[1] Akenine-Möller T., Aila T.:
Conservative and Tiled Rasterization Using a Modified Triangle Setup
Journal of Graphics Tools,
January 2005, Vol. 10, No. 3, pp. 1–8

[2] Arvo J.: *Transfer Equations in Global Illumination*
Global Illumination, SIGGRAPH'93 Course Notes

[3] Blender Foundation: *Blender 3D modeler*
cit. July 22, 2013, http://www.blender.org

[4] *Bumblebee Project*
cit. July 28, 2013, http://bumblebee-project.org

[5] Eaton J. W. & community: *GNU Octave*
cit. July 23, 2013, http://www.gnu.org/software/octave

[6] McGuire M.: *McGuire Graphics Data*
cit. July 22, 2013, http://graphics.cs.williams.edu/data

[7] Hasselgren J., Akenine-Möller T., Ohlsson L.: *Conservative Rasterization*
GPU Gems II 42,
Second printing, April 2005, pp. 677–690

[8] Ianonne M.: *Relative Standard Deviation*
Handout for the General Chemistry course at the Millersville University,
cit. July 22, 2013,
http://mustang.millersville.edu/∼iannone/handouts/ERAD.pdf

[9] Jones M. W., Bærentzen J. A., Sramek M.:
3D Distance Fields: A Survey of Techniques and Applications
IEEE Transactions on Visualization and Computer Graphics,
July-August 2006, Vol. 12, No. 4, pp. 581–599

[10] Kajiya J. T.: *The Rendering Equation*
Computer Graphics,
August 1986, Vol. 20, No. 4, pp. 143–150

[11] *Mercurial SCM*
cit. July 28, 2013, http://mercurial.selenic.com

[12] Pharr M., Humphreys G.:
Physically based Rendering - From theory to implementation
Morgan Kaufmann, 2nd ed., 2010

[13] Python Software Foundation: *Python Programming Language*
cit. July 28, 2013, http://www.python.org

[14] Rosenfeld A., Pfaltz J. L.:
Sequential Operations in Digital Picture Processing
Journal of the Association for Computing Machinery,
October 1966, Vol. 13, No. 4, pp. 471–494

[15] Semwal S. K., Kvarnström H.: *Directed Safe Zones and the Dual Extent Algorithms for Efficient Grid Traversal during Ray Tracing*
Graphics Interface,
May 1997, pp. 76–87

[16] SCons Foundation: *SCons: A software construction tool*
cit. July 28, 2013, http://www.scons.org

[17] Sramek M., Kaufman A.: *Fast Ray-Tracing of Rectilinear Volume Data*
IEEE Transactions on Visualization and Computer Graphics,
July-September 2000, Vol. 6, No. 3, pp. 236–252

[18] Stroustrup B.: *The C++ Programming Language*
cit. July 28, 2013, http://www.stroustrup.com/C++.html

[19] Weinzierl S.: *Introduction to Monte Carlo methods*
Topical lectures given at the Research School Subatomic Physics
Amsterdam, June 2000

[20] Wenzel J.: *Mitsuba Renderer*
cit. July 22, 2013, http://www.mitsuba-renderer.org

[21] Zhang L., Chen W., Ebert D. S., Peng Q.: *Conservative Voxelization*
The Visual Computer: International Journal of Computer Graphics
August 2007, Vol. 23, No. 9, pp. 783–792

Contents of the CD

The enclosed compact disc contains the digital version of this text and the source code of the Mitsuba renderer with my modifications. Here, the directory structure of the cd is briefly described.

mitsuba This directory contains the modified source code of the Mitsuba renderer as well as the testing scenes.

- **scenes** The testing scenes can be found here.
- **scripts** The voxelization debugging script is located in this directory
  - ***voxelator.py*** The voxelization script
  - ***voxelixe.sh*** Shell script for easier execution of the above python script.
- ***include/mitsuba/render/genericdistfield.h***
- ***src/librender/genericdistfield.cpp*** Base class for the distance fields.
- ***include/mitsuba/render/distfield.h***
- ***src/librender/distfield.cpp*** The non-directional distance field implementation.
- ***include/mitsuba/render/distfield_amr.h***
- ***src/librender/distfield_amr.cpp*** Implementation of the AMR distance field.
- ***src/integrators/raymarcher/volraymarcher.cpp*** The volumetric ray marching integrator.
- ***include/mitsuba/render/scene.h*** The distance field acceleration is in the `rayIntersect` function, which is defined here.
- ***config.py*** The example compilation configuration file that contains the directives needed to change the distance field compilation settings. Note that the configuration file is for the GNU/Linux operating system only.

tex The source codes and images for this text are found here.

- **pics** This directory contains pictures and diagrams used in the text.
- **previews** The screenshots and previews for the thesis text are contained in this directory.

dp-houska.pdf The PDF version of the text.

List of Figures

List of Tables

List of Abbreviations

AMR	...	Anisotropic Macro-Regions
CD	...	Chessboard distance
CPU	...	Central processing unit
DSZ	...	Directed Safe Zones
GPU	...	Graphics processing unit
GUI	...	Graphical user interface
NDF	...	Non-directional distance field
RSD	...	Relative standard deviation
VPT	...	Volumetric path tracer