Université de Montréal

Semi-Transparent Textures Based on Opaque and Transparent Texels Augmented with a Thickness

par

Mathieu David-Babin

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

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique, Option Imagerie

8 septembre 2022

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Semi-Transparent Textures Based on Opaque and Transparent Texels Augmented with a Thickness

présenté par

Mathieu David-Babin

a été évalué par un jury composé des personnes suivantes :

Mikhail Bessmeltsev

(président-rapporteur)

Pierre Poulin

(directeur de recherche)

Michalis Famelis

(membre du jury)

Résumé

Le rendu en temps réel repose sur des compromis entre la performance et le réalisme. Un de ces compromis est de représenter des matériaux plus minces tels que les tissus comme étant infiniment minces pour économiser mémoire et temps de rendu. Par contre, cette perte de dimension prive la surface de propriétés essentielles à certains effets visuels. Dans ce mémoire, nous présentons une méthode pour simuler les effets de l'épaisseur sur des surfaces semi-transparentes en utilisant des textures composées de texels opaques et transparents. Nous analysons les trous formés par les texels transparents et nous conservons de l'information sur les contours des trous dans une structure hiérarchique compatible avec la méthode de filtrage de textures par *MIP* map. Nous dérivons des équations représentant la proportion de lumière passant dans un trou avec des murs intérieurs en fonction de l'angle incident des rayons de lumière. Nous combinons ces équations avec l'information conservée pour calculer un terme de transparence à différents niveaux de détail en temps réel.

Mots clés : semi-transparence, *MIP* map, épaisseur, texture

Abstract

Real-time rendering is built upon compromises between performance and realism. One such compromise is to represent thinner materials like textile as infinitely thin in order to save on memory and rendering time. However, this loss of dimension robs the surface of properties key to some visual effects. In this thesis, we present a method to simulate the effects of thickness on semi-transparent surfaces using textures consisting of opaque and transparent texels. We analyze holes formed by transparent texels and store information about the contours of the holes in a hierarchical structure compatible with the filtering method of *MIP* mapping. We derive equations representing the proportion of light passing through a hole as a function of the incident angle of light. The proportions of texel top, texel side wall, and hole are computed accurately. We combine these equations with the information stored to compute a transparency term at different levels of detail in real time.

Keywords: semi-transparency, *MIP* map, thickness, texture

Contents

List of Tables

List of Figures

List of Acronyms, Abbreviations and Notations

LOD	Level of detail
MIP	Latin phrase "*multum in parvo*" meaning "many things in a small place". MIP mapping is a common hierarchical method to filter textures [**Wil83**].
VSync	Vertical Synchronization
$\mathbf{A}$	A matrix or a plane
$\mathbf{v} = (v_x,v_y,v_z)$	A vector or a point
s	A scalar

Remerciements

Je tiens à exprimer ma plus profonde gratitude à mon directeur de recherche Pierre Poulin pour m'avoir apporté le soutien technique et financier nécessaire à la réalisation de ce beau projet que ce soit depuis son bureau ou d'outre-Atlantique. J'aimerais aussi remercier le CRSNG pour leur soutien financier par l'entremise du financement CRSNG Découverte de Pierre.

Je suis également reconnaissant envers ma famille et ma copine pour leur soutien moral constant durant ces années marquées par une pandémie mondiale. Je tiens aussi à remercier mon chat pour avoir autorisé sa photo utilisée pour agrémenter ce document.

Acknowledgments

I would like to express my deepest gratitude to my research director Pierre Poulin for giving me the technical and financial support needed to achieve this great project may it have been from his office or from across the Atlantic. Additionally, I would like to thank NSERC for their financial support via Pierre's NSERC Discovery grant.

I am also grateful to my family and my partner for their constant moral support during these years marked by a global pandemic. I would also like to thank my cat for allowing its picture to be used to embellish this document.

Chapter 1

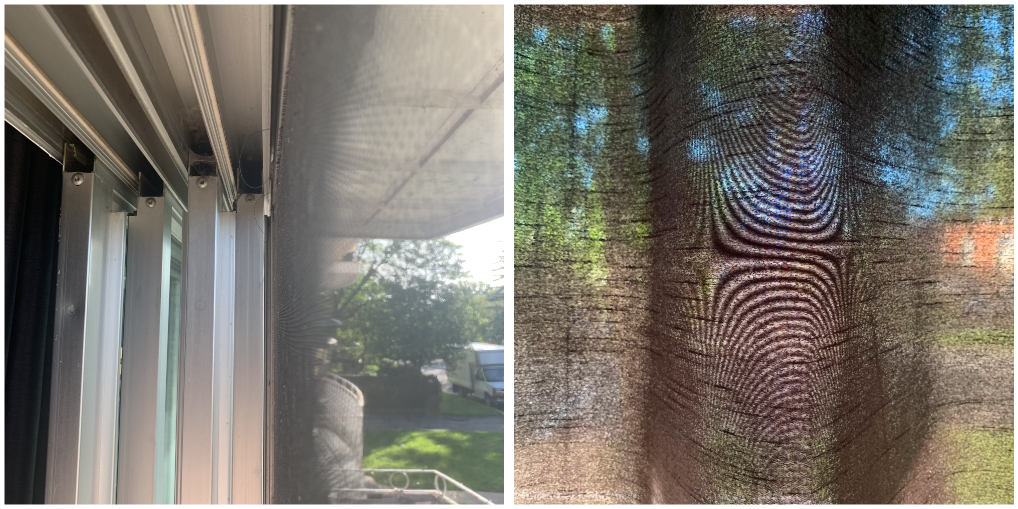
Introduction

The evolution of computer graphics has produced many different techniques to increase realism and details of simple surfaces. They have resulted in richer experiences for users. Such techniques include the mapping of textures onto surfaces, the effects of light on surface details, the simulation of reliefs or wrinkles on surfaces, etc. With the mapping of textures also came the concept of transparency, which can transform a simple supporting shape with few edges into something with much more sophisticated contours, different thicknesses, and different appearances due to interactions with light. Transparency can enhance the representation of all kinds of objects such as silhouettes of 2D avatars from a photograph, contours and veins of leaves, wire fences, etc. The representation of surfaces containing holes at different scales (like fences, patio screen doors, or embroidery curtains) is especially important because those surfaces can require a lot of fine three-dimensional details if they are modeled with meshes.

Semi-transparency allows light to go through a surface; light can be blocked, change its color, or traverse the surface without being affected. Thin textiles such as silk or some curtains are good examples of opaque objects that have a semi-transparent property because of small gaps (holes) in their material. Up close, the different opaque threads forming the textile and the spaces in-between them are visible, but from a distance the surface as a whole appears as a single semi-transparent object.

One way to encode semi-transparency is through a texture. Texture mapping, being prone to aliasing problems, has also motivated the introduction of *MIP* maps, a data structure allowing for the simple and compact representation of textures at multiple resolutions. The multiple resolutions enable more intricate filtering than what is efficiently possible with a single resolution. MIP maps empower rendering engines to save time and to reduce aliasing artifacts caused by the discrete nature of an image-based texture.

Representing surfaces with holes using textures and transparency instead of fine geometry produces more accurate filtered results for little additional computation time. Unfortunately, with the elimination of the extensive geometry, a loss of realism may also be observed, exacerbated as the represented surface loses the effects due to its thickness. When looking at such surfaces at an angle, the visual impact due to their thickness becomes more and more apparent as the angle becomes steeper (see Figure 1.1). The loss of transparency at an angle is currently not represented with semi-transparent textures in modern rendering engines. At close distance from the surface, the infinitesimal thickness of the surface becomes also apparent even with state-of-the-art shading methods. A solution to this loss of details would be to associate a thickness to the surface and use a transparency function affected by the thickness and the viewing angle. However, with transparency dependent on the viewing angle, the linear compound of transparency, required to correctly build appropriate MIP maps, would no longer be respected.



(a) (b)

Figure 1.1 – Photos of a patio door screen observed from a steeper angle and of a wavy curtain observed from a perpendicular angle.[1]

1. Unless stated otherwise, all photos and figures come from the author of this thesis.

In this thesis, we present a method to simulate thickness for a semi-transparent surface. We investigate the effect due to the view angle on the perceived transparency of a surface containing holes by calculating the proportion of light making contact with the top (opaque and semi-transparent texture elements, also called texels, analogous to pixels for images) and the interior walls (walls on the sides of the holes) for that surface. We derive analytic equations for the proportions. We introduce a new representation for the thickness and the interior walls of a texture that is suitable with a MIP map structure, thus removing the problem of nonlinearity caused by transparency being dependent of an angle. We design an algorithm to compute colors with the appropriate transparency at multiple levels of detail in the MIP map structure. Our method can render images in real time.

This thesis is organized in the following way. In Chapter 2, we present the computer graphics background necessary to the understanding of our research. We also review previous work most related to our research. This is not intended as an in-depth survey,but rather as an introduction to the context of our textures with a thickness. In Chapter 3, we present the calculations leading to our algorithm as well as the necessary structure needed to comply with MIP mapping. In Chapter 4, we show and analyze our results of semi-transparency using scene renderings, discuss performance metrics, before concluding and giving directions for future improvements in Chapter 5.

Chapter 2

Background and Previous Work

2.1. Textures and Texture Mapping

A texture map is a data structure containing texture information to enhance the details of a given computer generated surface. The types of data that can be contained in a texture map are quite diverse and could be any combination of RGB colors, surface normals, displacement values [**Bli78**], reflection parameter values, transparencies, and others. The structure of a texture map is most often a 2D array of texels forming an image. A texel is the basic unit of a texture map.

A geometric model is commonly composed of vertices forming triangles, quadrilaterals (quads), or more complex polygons. The use of triangles is far more common than the use of quads and complex polygons, the latter being much less common. That structure is called a mesh. Without texture maps, a vertex can be assigned a color by user input or by a function based on an available variable such as position or time. When using a texture map, a vertex is instead assigned a relative 2D position in the map (image) as coordinates (u,v). Any point on the mesh, thus located between vertices (i.e., on a triangle or a quad) will not have coordinates assigned to it. Instead, the (u,v) coordinates of the surrounding vertices will be interpolated to provide a relation between those points and the texture map. Other types of geometry exist, such as parametric surfaces, implicit surfaces, volumetric datasets, etc. In the context of this thesis, we assume they would be converted into meshes for real-time rendering.

Texture maps that store the color of a surface are also called albedo maps or diffuse maps. Both maps store the color of the surface, but colors of diffuse maps are pre-shaded to give a more realistic appearance without the need for shading computations with a lighting model. Color is most often represented as a triplet of values (each called a channel). The three channels are the red, green, and blue channels, forming the RGB triplet. Each channel stores a value contained in an interval (the [0,255] interval is often used to represent 8-bit colors, otherwise encoded as a floating-point number in [0,1]). That value encodes the intensity of its channel. The higher the value is, the higher the intensity of the channel. It is frequent for transparency to be encoded alongside the triplet as a fourth channel to form an RGBα quadruplet, 8 bits per channel, as a 32-bit word.

Texture maps that store surface normals or displacement values are often called bump maps [**Bli78**], normal maps, and displacement maps. These maps are used as a way to simulate a meso-structure on a surface (see Section 2.5). With normals, each element of the map stores a normalized normal vector $\mathbf{n} = (n_x,n_y,n_z)$. The normal $\mathbf{n}$ is used in a shading model instead of the original normal. The new normal can be defined by a function representing a more complex shape than the underlying one, in order to add an illusion of depth, roughness, or complexity. When storing displacement values (or normal vector perturbations), each element of the map stores a scalar h. That scalar represents a variation of the height of the surface at a point along the normal. A new normal can be calculated from the new position of the point and/or from its neighboring texels. That normal can then also be used for shading computations.

2.2. Sampling and Texture Sampling

When rendering a three-dimensional scene, the continuous world of the scene is discretized into an array of pixels. Two common ways of rendering are by rasterization and by ray tracing. In rasterization, each vertex in the scene is projected onto a two-dimensional screen by multiplying its position by a projection matrix. The screen is then discretized into an array of pixels, and a color is assigned to each pixel. In ray tracing, multiple rays are shot from the camera into the scene by passing through a screen. The screen is also discretized into an array of pixels, and each ray passing through one pixel assigns to it some portion of its computed color. In both cases, the transfer of colors to pixels is done by evaluating the color of the scene at a single point (x,y) on the screen (generally at the center of the pixel). This process is called sampling or point sampling. Multiple points per pixel could be evaluated (either randomly or not) and then averaged if a box filter is used over the pixel. This is called super-sampling or multi-sampling.

The color assigned to the pixel comes from a given point on the object. If the object is associated with a texture, the color will come fully or partially from the texture. The color is obtained with the given point on the object and the texture mapping function. That operation is called texture sampling.

2.3. Filtering and Texture Filtering

A big problem can arise when a continuous signal is sampled at regular intervals. If the sampled signal (scene or texture) contains frequencies that are higher than the sampling frequency, then aliasing appears as staircases or Moirés [**Sha49**], or as noise. Even if a texture (or a scene) is composed of low-enough frequency components, the context in which the texture is sampled can create a problem. Figure 2.1 shows in 2D the impact that moving away a textured object can have on the sampling frequency ω_s while keeping the texture frequency ω_b constant. An object moved far enough could then produce more aliasing when much less aliasing is present when placed closer to the view point. A chess board pattern is a common example of a texture with high frequencies. Figure 2.2(b) shows the impact of observing a textured object at from a steep angle. Even if the texture's frequency stays constant in object space, a steeper angle of view causes a variation of the resulting sampling frequency on the object's surface. In screen space, the samples on the screen are at a constant frequency, but the textured object projected on the screen shows irregularity in frequency.

To reduce aliasing, the rendering process needs a way to compensate for the textures containing high frequencies. A simple way of doing so would be to increase the sampling frequency, and therefore reduce the difference between ω_s and ω_b. That solution works well but has a few caveats. First of all, it increases computations and thus rendering time. It also does not guarantee the complete removal of aliasing. If the sampling is increased for one object in the scene, others might still exhibit textures with higher frequencies.

Another solution found for anti-aliasing is prefiltering [**Hec89**]. In signal processing, filtering is the action of convolving a signal function f with a filter function g (denoted as

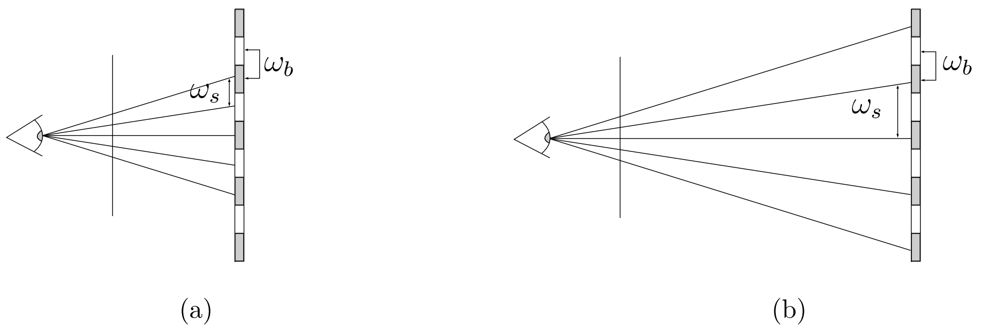


Figure 2.1 – Sampling frequency and texture frequency of an object at different viewing distances. The high frequency of the texture causes the sampled colors to vary.

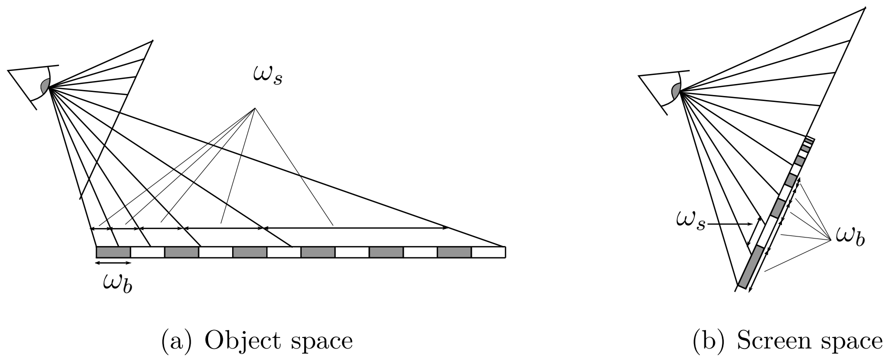


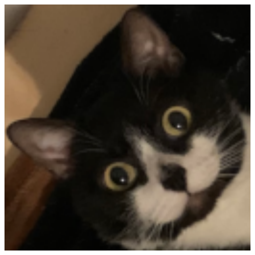
Figure 2.2 – Sampling frequency and texture frequency of an object at a steep viewing angle in object space (a) and in screen space (b). Even if the samples are processed at regular intervals, the viewing angle causes the distance interval of the samples on the object to be irregular.

$f \star g$). The filter g is often nonzero for only a small portion of the domain. We can define a convolution on a two-dimensional signal (continuous and discrete) as

$$(f \star g)(s,t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y)g(s-x,t-y)dxdy \tag{2.3.1}$$

$$(f \star g)(i,j) = \sum_{k=-\infty}^{\infty} \sum_{p=-\infty}^{\infty} f(k,p)g(i-k,j-p), \qquad \text{for } i,j \in \mathbb{Z}. \tag{2.3.2}$$

Texture filtering is simply a convolution on a texture treated as a discrete function f with a discrete filter g. The filter is represented as an $a \times b$ matrix and is applied on each element of f. More often than not, a and b are odd to give the matrix a central element in



$$\star \quad \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} =$$

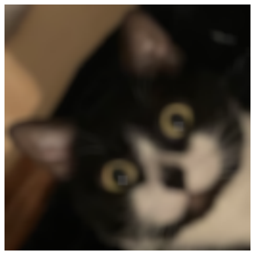


Figure 2.3 – A 128×128 image (left) convolved with a 5×5 blur filter (center) produces a blurry image (right).

each dimension. For the filtering of a texel (i,j) in f, the filtered texel t' is defined as

$$t' = (f \star g)(i,j) = \sum_{k=0}^{a-1}\sum_{p=0}^{b-1} f\left(i - \left\lfloor \frac{a}{2} \right\rfloor + k, \quad j - \left\lfloor \frac{b}{2} \right\rfloor + p\right) g(k,p).$$

Doing the same operation for all texels of f results in a filtered texture. For example, a blur filter operation, also called box filter, is shown in Figure 2.3. For coordinates evaluated by f that would be outside of the texture, arbitrary constant values can be used or f can be defined to look at coordinates at the opposite sides of the texture (replicated, mirrored, etc.). Filters come in different shapes and sizes as shown in Table 2.1.

Prefiltering is the application of a filter on a texture prior to its sampling. The goal of prefitering is to reduce the high frequencies in a texture in order to reduce the aliasing from the sampling. Prefiltering is therefore an anti-aliasing technique.

2.4. *MIP* Mapping

Even when using a prefiltered texture, aliasing effects could reappear as the viewing distance to the object increases. The filtered frequencies of the texture could still be too high at further distances or steeper viewing angles. However, simply increasing the size of the filter used to further dampen the frequencies would cause a greater loss of detail. A solution is to have multiple versions of the same texture, each filtered with varying sizes of filters. Unfortunately, this considerably increases the memory required. Williams [**Wil83**] introduces MIP mapping as a solution to this problem.

MIP mapping is the combination of a hierarchical structure (a pyramid) [**Cla76**] and texture filtering that serves for both anti-aliasing and hardware acceleration [**Wil83**]. The idea is to compute several lower resolutions of a texture, and store them together for rapid access. Defining the original texture as level 0 of the hierarchy, a MIP map level l is half

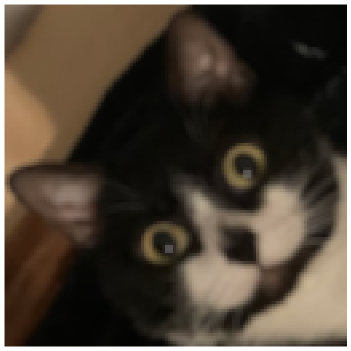
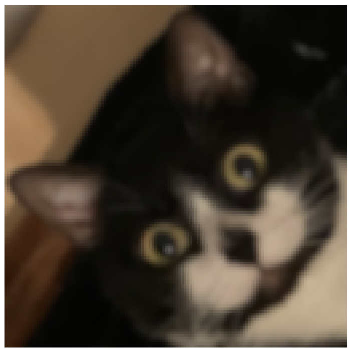
Shape/Size	Representation	Example
Box (3×3)	$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian (5×5)	$\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	

Table 2.1 – Different types of filters, the matrix of their weights, and their resulting images.

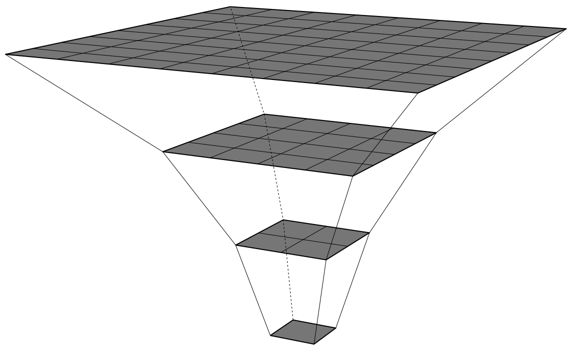


Figure 2.4 – Pyramidal structure of the MIP map texture.

the resolution of level $l-1$ in each dimension. It is a filtered version of the previous level. Most often, a 2×2 box filter is used on consecutive 2×2 patches of texels on level $l-1$ to generate all the texels of level l. The textures form an inverse pyramid going from the highest resolution (the original) to the lowest (a single texel) (see Figures 2.4 and 2.5).

A MIP map is precomputed to save on execution time. Even though the pyramid requires more memory to store the additional levels, a MIP map including the full resolution texture

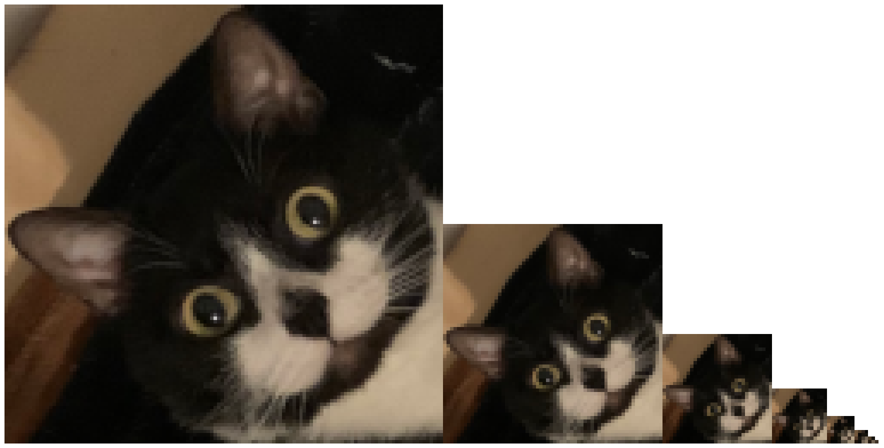


Figure 2.5 – Levels of the MIP map of a texture.

will only be at most $\frac{4}{3}$ of the original nonfiltered texture. When a texture's resolution is halved, its number of texels is divided by four. Therefore,

$$Area * \sum_{n=0}^{\infty} \frac{1}{4^n} = \frac{4 * Area}{3}.$$

During rendering, two pieces of information are necessary for sampling a MIP mapped texture: the considered level of detail (LOD) and an interpolation method. The LOD is a floating-point non-negative number that represents how many texels of the original texture can fit in a pixel on a logarithmic scale. If λ is the number of texels per pixel then

$$LOD = \log_2 \lambda. \tag{2.4.1}$$

The LOD can be used to select one or two adjacent levels from the MIP map. The number of levels selected depends on the interpolation method chosen. Four methods are used most often :

(1) Nearest neighbor (one level); no interpolation.

(2) Bilinear interpolation (one level).

(3) Linear interpolation (two levels) from two nearest neighbors.

(4) Trilinear interpolation (two levels); linear interpolation from two bilinear interpolations.

Interpolation is often used as a smoothing mechanism of abrupt changes in color or resolution. When point sampling a texture, the relative point in (u,v) coordinates will not

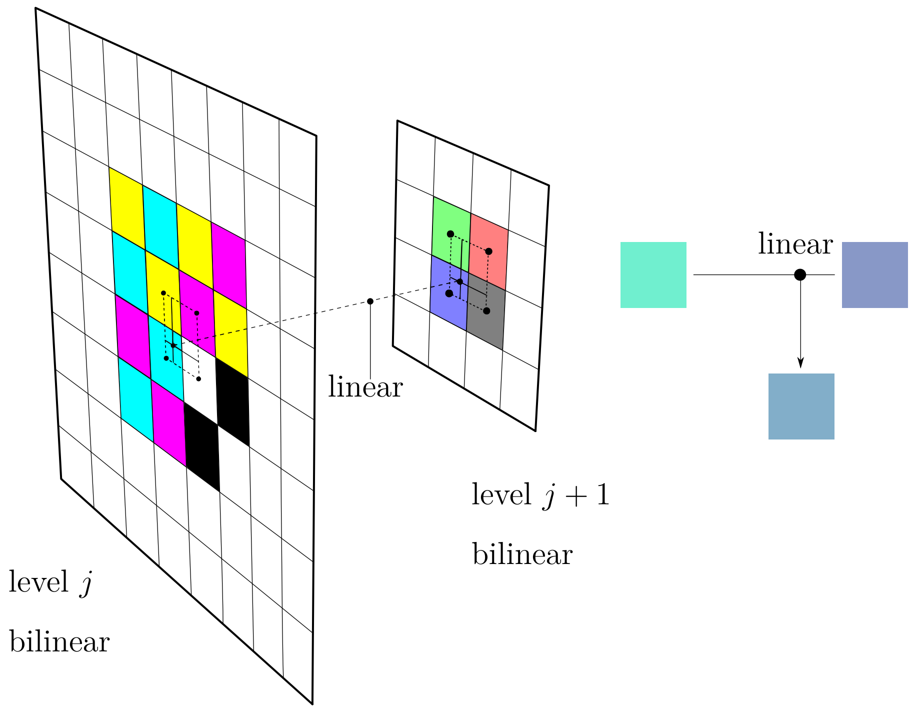


Figure 2.6 – Trilinear interpolation in a MIP map as a linear interpolation of two bilinear interpolations.

necessarily correspond to the center of texture's texel. Since the texture is stored as an array of discrete texels, there are two ways of sampling a color from the array. The first way is to convert the (u,v) coordinates into texture coordinates (i,j) and then to round the result to the "nearest" texel (integer point). The color from that texel is used. The second way is to convert the (u,v) coordinates into texture coordinates (i,j), and then to bilinearly interpolate the colors from the four texels (i,j; $i+1$,j; i,$j+1$; $i+1$,$j+1$) surrounding (u,v) (see Figure 2.6).

The interpolation methods enumerated above provide different levels of smoothing. With the nearest neighbor approach, it is not really an interpolation as the computed LOD is rounded to the nearest level, and the nearest texel on the corresponding level is sampled. It results in blocky textures with a visible change of resolution at the limit between levels. With linear interpolation a color is sampled at the nearest texel from the two levels surrounding the value of the LOD, and then the two colors are linearly interpolated. With bilinear interpolation the nearest MIP level to the LOD is picked, and then the colors of the four texels nearest to the sample point are bilinearly interpolated according to their distance to the sample point. Trilinear interpolation is a combination of the last two methods. Two colors are sampled by bilinear interpolation from the two levels surrounding the LOD value, and then the resulting colors are linearly interpolated (see Figures 2.6 and 4.6).

2.5. Geometries at Multiple Scales

Real-life objects can be perceived as having multiple scales of details. Kajiya [**Kaj85**] proposes to model the computer representation of those objects as having three scales (LODs) visible at one given LOD. Section 2.4 defines the LOD concept for texture mapping, but the same concept can also be used for geometric models by having multiple versions of a mesh with decreasing numbers of faces [**Wil83**], as it is often done in the industry. The three proposed scales are the geometry, the meso-scale (texture), and the micro-scale (a reflection model).

2.6. Transparency and Semi-Transparency

Generally speaking, transparency is the property allowing light to pass through an object so that a viewer may see through that object. In computer graphics, transparency can be associated to an object directly or to a texture mapped on the object. Transparency can also represent an object with a complex contour or an object filled with holes, both displayed on a simple flat surface. Contours of leaves and wired fences are good examples of such uses of transparency.

Transparency is most often stored in textures alongside color values. While an object (or part of it) can be fully opaque or fully transparent, storing the information in a texture allows for a finer control of opacity and transparency, i.e., semi-transparency over different parts of a surface. Transparency is stored as a fourth channel along the RGB triplet to form an RGBα quadruplet. If each channel is represented by one byte, then a texel is four bytes (or 32 bits). The fourth channel is called the alpha (written α) channel. The larger the value of the alpha channel, the more opaque the texel. If the value is 0, then the texel is fully transparent. Therefore, any value between 1 and 254 inclusively will show semi-transparency (if the channel is encoded in one byte).

2.7. Bump Mapping

As described in Section 2.1, texture maps can store more than simple color data. Texture maps can also store surface displacement normal values. Bump mapping uses such textures [**Bli78**]. The displacement normals are used to modify the normals of the surface of a geometric model. When those updated normals are used alongside a shading model, such as the Blinn-Phong reflection model [**Bli77**], simple surfaces produce much finer details, thanks to the shading. However, a problem arises when filtering methods are needed to reduce aliasing and are used on the bump maps. The function combining normals on the texture should not be linear, unlike the one used for colors in an albedo map. If the normals are averaged like colors are, then surface details are generally smoothed out. Solutions with good results to this problem have been published [**Fou92, DHI^{+}13, OB10**]. Generally speaking, averaging distributions of normals rather than normals themselves produces better results. It works well for Gaussian-like distributions with a single peak.

2.8. Relief Mapping

Oliveira et al. [**OBM00**] introduce relief mapping, a method utilizing texture mapping to add volumetric details to geometric models. Relief mapping relies on a relief texture, which is a standard albedo or diffuse texture augmented with orthogonal displacements per texel. The effects produced by the relief map can vary with the viewing distance to a surface (see Figure 2.7). When looking at a far enough surface, the texture can be rendered normally. When getting closer, the texture can be warped, using the orthogonal displacement, to offer a parallax effect. At very close distance to the surface, the texture can be used to render a mesh of micro-polygons. The use of textures to generate volumetric details to an otherwise flat surface in relief mapping motivates our goal of providing thickness details to a surface using information stored in a texture.

2.9. LEAN Mapping and LEADR Mapping

Olano and Baker [**OB10**] (LEAN mapping) and Dupuy et al. [**DHI^{+}13**] (LEADR mapping) both address the problem of MIP mapped bump maps. LEAN mapping offers an efficient technique for filtering specular highlights in bump and normal maps. The method

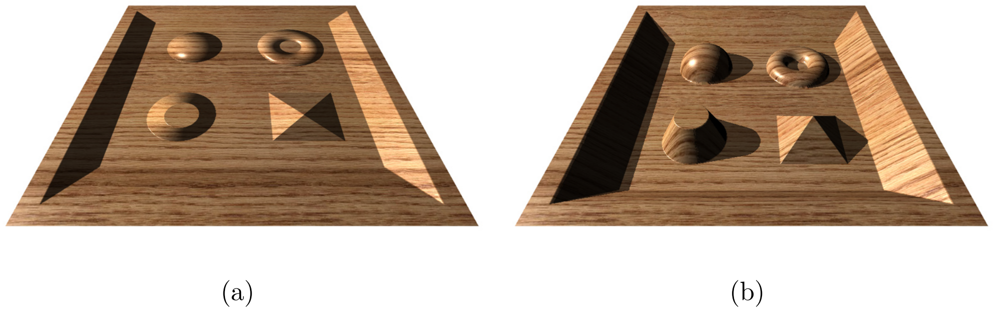


Figure 2.7 – A single rectangle rendered from the same view point using two different techniques: (a) Bump mapping, and (b) Relief mapping with self-shadowing. A 2D wooden texture was mapped to the surface. Image from [**POC05**].

stores a Beckmann distribution of normals. A distribution is encoded with the main bump direction projected onto the tangent plane of the surface and the second moments of the covariance matrix used in the Beckmann distribution to control the spread of the distribution. These variables are enough to reconstruct all that is needed for the Ward shading model [**OB10**], which is based on Beckmann distributions. Both controls of Beckmann distributions can be combined linearly. Being linearly-filterable, the encoded elements can be used alongside the standard MIP mapping mechanisms. The authors base their method on a modified Ward shading model, but present a mapping to make use of the standard and more popular Blinn-Phong model [**OB10**].

LEADR mapping adapts LEAN mapping to a physically based reflectance model. Doing so for displacement values rather than normals enables LEADR mapping to support masking and shadowing affecting surface roughness, and under environment lighting. The physically based model also supports point and directional lighting that were already supported in LEAN mapping.

LEAN and LEADR mappings motivate our goal of developing a method that is compatible with MIP mapping as they demonstrate its importance and its impact. An adaptation of their work for our context could be to interpret a hole as a distribution of direction vectors that let light pass through the surface augmented with a thickness. Instead, we chose to study and store information about the simulated geometry for multiple directions. This path puts us conceptually closer to Horizon mapping [**Max88**].

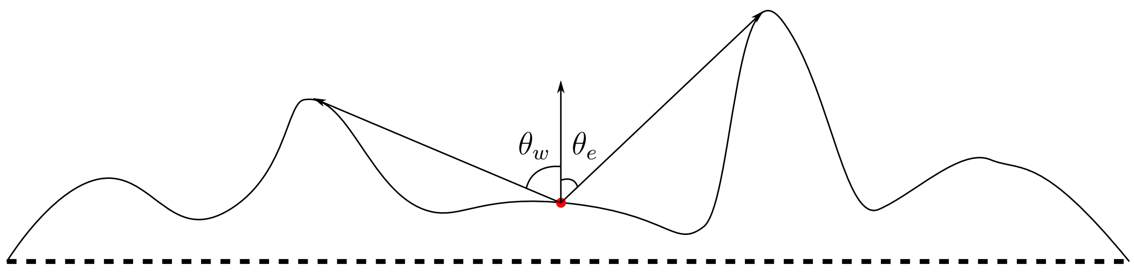


Figure 2.8 – Horizon angles in west and east direction.

2.10. Horizon Mapping

Max [**Max88**] introduces Horizon mapping, a method to produce shadows on a surface from the bumps of a normal map. In Horizon mapping, shadows are rendered using precomputed tables containing information about the horizon elevation angles (see Figure 2.8) in multiple sampled directions tangent to the texture base. During rendering, the presence of a shadow for a point is established using the light direction, the light angle to the normal, and the horizon angles stored. If the light angle is greater than the one stored for that point at that direction, then it is in shadow. For intermediate directions to the ones stored, angles can be interpolated. Horizon mapping has similarities with our final method, as it stores directional data in a table for points on a surface. Unfortunately, it lacks the support for MIP map, which we implement in our final method.

Chapter 3

Semi-Transparent Textures with a Thickness

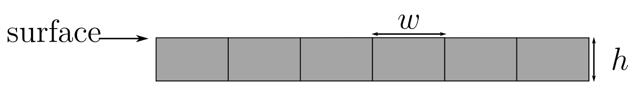
In this chapter we present the development of our sampling and filtering algorithm for a semi-transparent texture augmented with a thickness. Our method complies with the structure and linear quality of MIP maps, and can be used to render scenes in real time.

3.1. Thickness in 2D

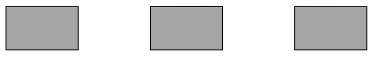
We first observe the behavior of light when illuminating a hole in a simplified 2D scene consisting of a planar surface, with a texture defining holes, and an opaque material underneath the surface of opaque texels. Our 2D scene is composed of several identical, evenly-spaced, rectangular boxes. The space between boxes defines what we call holes in this context. Figure 3.1 is a depiction of our scene.

When sampling a scene with ray tracing, we compute the color of the first object intersected by each ray. For a given pixel area, multiple rays are traced and their colors are averaged uniformly or with weights. We study the effect that adding holes to a surface would have on the final color of the sampling. If our scene is flat and continuous (i.e., without holes) as in Figure 3.1 (a), then the sampling corresponds to the surface's filtered (e.g., average) color. The sampling is different in presence of holes. Some rays intersect the top, some intersect the inside edge of the hole, and some go through without intersecting any part of surface. Each type of intersection could return a different color. We focus our observation on the rays entering one hole in particular.

First of all, we consider all rays hitting the surface to be parallel in order to simplify our calculations. This reduces the problem to parallel projection instead of perspective



(a) Continuous and fully opaque.



(b) With regular holes.

Figure 3.1 – Visibility of a planar surface of opaque or transparent texels of width w and a certain thickness h.

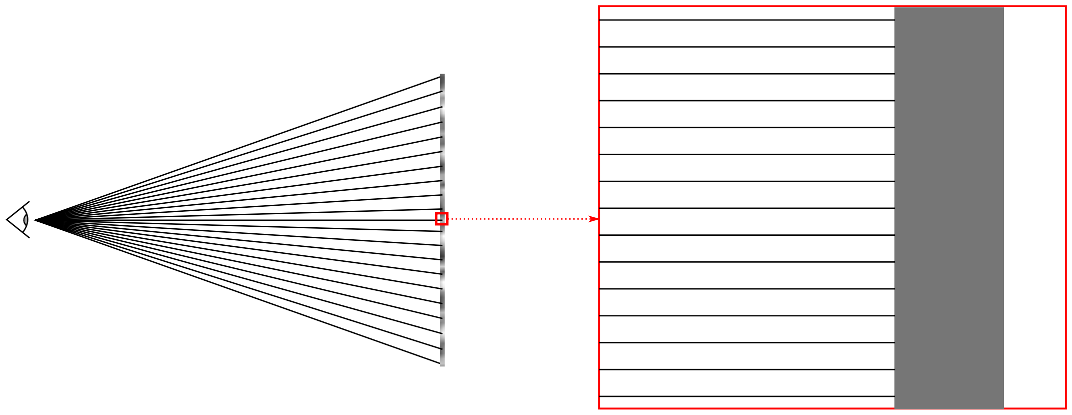


Figure 3.2 – Rays intersecting a very small region of a surface can be considered parallel to each other.

projection. This is assumption is common when a pixel is considered small compared with respect to the surface (see Figure 3.2).

Each box has a height h, a width w, and each neighboring hole has a width w. If we look at the ray intersecting the bottom corner of the box, there is a clear separation of all rays intersecting the box and all rays intersecting nothing. Those two types of rays output one color each (we assume the rays intersecting nothing return an arbitrary background color). To get the combined color of the two sets of rays, we need to establish the proportion of each type. One of the factors affecting proportions is the direction of the incident rays, i.e., the angle θ formed between the surface's normal and the vector along the rays going away from the surface (see Figure 3.3).

When $\theta = 0$, all the rays go through the entire hole and non intersect the side wall. We define θ_{max} as the maximum angle at which some rays still go through part of the hole, meaning that for all angles superior or equal to θ_{max}, all the rays intersect the side wall. At

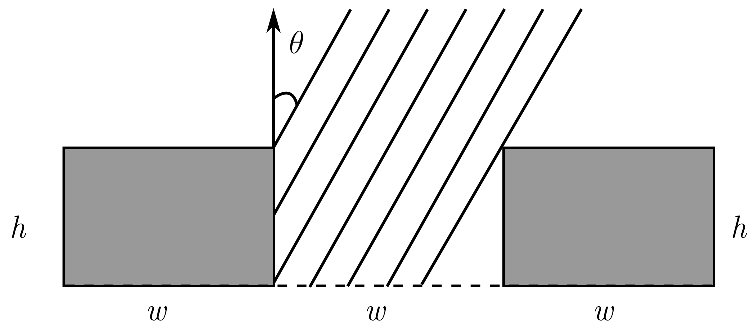


Figure 3.3 – 2D scene sampled by parallel rays.

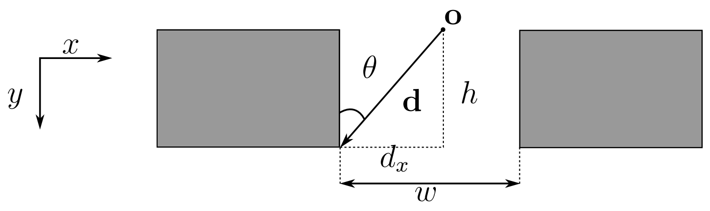


Figure 3.4 – 2D scene sampled by a single ray of direction **d** (and angle θ) passing through the surface at point **o** and intersecting the bottom of the side edge.

$\theta = \theta_{max}$, a single ray intersects exactly the bottom corner of the box and the top corner of the adjacent full box. With this, we can express θ_{max} in terms of the known dimensions of the box

$$\tan\theta_{max} = \frac{w}{h}.$$
$$\theta_{max} = \arctan\left(\frac{w}{h}\right).$$

We use a vector **d** to form the angle θ. We can set its components to fit h and w ($\mathbf{d} = (w,h)$) and θ will not change. We can change θ by increasing or decreasing one or both components of **d**, from θ_{max} to 0.

For a view angle $\theta \in [0, \theta_{max}]$, we define the ray R passing through the surface at a point $\mathbf{o} = (x_0, y_0)$ and intersecting the bottom of a side edge. We define the direction of R to $\mathbf{d} = (d_x, h)$. Figure 3.4 depicts this situation. At this angle, any other ray that intersects the side edge would go through the surface at a point to the left of **o** in x. Any ray that would go through the surface to the right of **o** would not intersect the side edge.

The type of rays then depends on the distance from the top of the side edge to the point where the ray goes through the surface. Figure 3.4 illustrates well that distance. Any ray going through the surface at a distance smaller or equal to d_x will intersect the wall. This

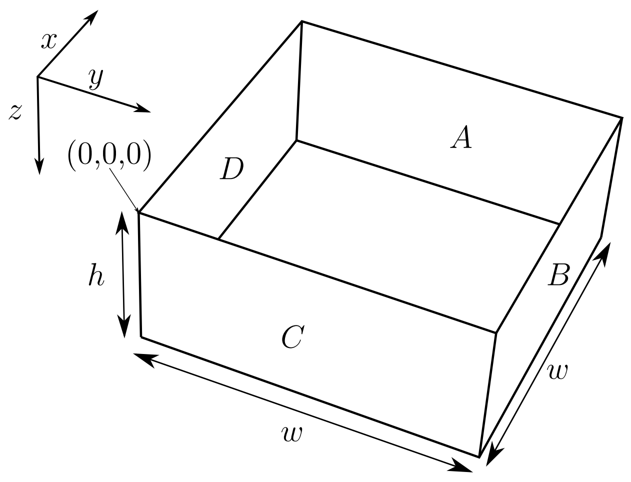


Figure 3.5 – Square shaped hole.

observation allows us to compute the proportion of rays hitting the wall. That proportion O is simply $\frac{d_x}{w}$, and the proportion α of rays not hitting is equal to $1 - O$.

3.2. Thickness in 3D

We now observe the behavior of the rays in a similar three-dimensional setting. We cast rays at a square shaped hole, i.e., surrounded by fully opaque boxes. The hole could be rectangular, but we use a square to simplify the exposition of our calculations. We want to obtain the ratio of rays that are occluded to the rays that are cast. For a ray to be occluded in this situation, it needs to intersect one of the side walls inside the hole. We work in the local frame represented in Figure 3.5. The box has sides of length w and of height h. We cast rays at the top surface of the hole ($z = 0$) from the origin to (w,w,h). The hole is composed of four walls that can be represented by the bounded planes

$$\mathbf{A}: \quad x - w = 0 \quad y \in [0,w], \quad z \in [0,h] \tag{3.2.1}$$

$$\mathbf{B}: \quad x = 0 \quad y \in [0,w], \quad z \in [0,h] \tag{3.2.2}$$

$$\mathbf{C}: \quad y - w = 0 \quad x \in [0,w], \quad z \in [0,h] \tag{3.2.3}$$

$$\mathbf{D}: \quad y = 0 \quad x \in [0,w], \quad z \in [0,h]. \tag{3.2.4}$$

Let the direction vector **r** of a ray and the origin point **o** be

$$\mathbf{r} = (r_{xo}, r_{yo}, r_{zo}),$$
$$\mathbf{o} = (x_0, y_0, z_0).$$

To simplify our calculations we scale r_{zo} by $\frac{h}{r_{zo}}$ to get a vector that starts at the top of the hole and ends at the bottom (similarly to Section 3.1). We now have

$$\mathbf{r} = \left(\frac{r_{xo}h}{r_{zo}}, \frac{r_{yo}h}{r_{zo}}, h\right)$$

that we rewrite as

$$\mathbf{r} = (r_x, r_y, h) \tag{3.2.5}$$

to lighten the notation. We consider the components of **r** to be non-negative and we will generalize our findings later on. An intersection point **p** between a ray and a surface can be written as a composition of parametric equations

$$\mathbf{p} = \begin{cases} x &= x_0 + tr_{xo} \\ y &= y_0 + tr_{yo} \\ z &= z_0 + tr_{zo} \end{cases} \qquad t \in \mathbb{R}$$

where t is the signed distance in direction **r** between **p** and the origin of the ray. For the intersection to be considered, t must be non-negative, as a negative value would indicate an intersection in the opposite direction of the ray. In our current configuration, the only walls that will be intersected by the rays are plane **A** (Eq. 3.2.1) and plane **B** (Eq. 3.2.3). We will compute the occlusions on plane **A**, generalize to plane **B**, and then add them up.

With the conditions we have laid out in Equation 3.2.5, we change **p** to

$$\mathbf{p} = \begin{cases} w &= x_0 + tr_x \\ y &= y_0 + tr_y \\ z &= th \end{cases} \qquad t \geq 0, \quad y \in [0,w], \quad z \in [0,h]. \tag{3.2.6}$$

For $z = th$ and $z \in [0,h]$ to be true, we need $t \in [0,1]$. With this condition we deduce $x \in [w - r_x, w]$ by evaluating **p** at $t = 0$ and $t = 1$.

Looking at Equation 3.2.6, we see that x_0 needs to be in the interval $[w - rx, w]$ for t to be in its defined interval, while y_0 is in $[0,w]$. This gives us the interval for x_0. When taking a closer look at y_0, specifically at the point $(w - r_x, w - r_y)$ where $t = 1$, we have

$$\begin{aligned} & y = w - r_y + tr_y && \text{from Eq. 3.2.6 with } y_0 = w - r_y \\ \Rightarrow & y = w && \\ \Rightarrow & w = y_0 + tr_y. && \end{aligned} \tag{3.2.7}$$

If we increase the value of y_0 in Equation 3.2.7 without changing x_0,

$$\begin{aligned} y_0 + tr_y &> w \\ y &> w && \text{from Eq. 3.2.6.} \end{aligned}$$

This breaks the condition put on y in Equation 3.2.6. When $y_0 > w - r_y$, we need to decrease the value of t for the condition to be respected. Looking at the partial derivative of t in function of x_0 in Equation 3.2.6,

$$\begin{aligned} w &= x_0 + tr_x \\ t &= \frac{w - x_0}{r_x} \\ \frac{\partial t}{\partial x_0} &= \frac{-1}{r_x}. \end{aligned}$$

Therefore, t decreases when we increase x_0. At $y_0 = w$, t must to be 0 to keep $y \in [0,w]$. t becomes 0 when x_0 is increased to w. So, we want to define a function of y_0 expressed in

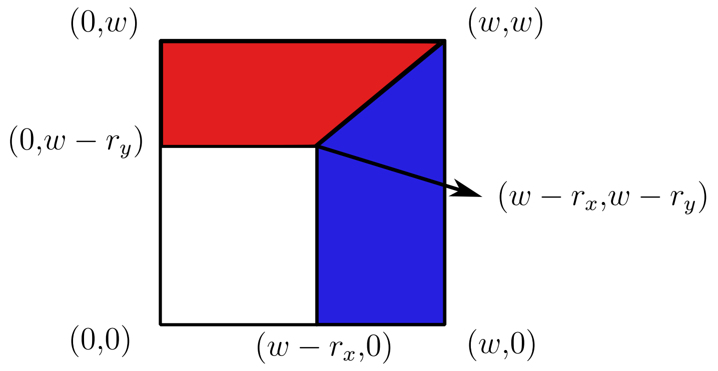


Figure 3.6 – Areas of occlusion of the box in red and blue.

x_0.

$$\begin{aligned} w &= x_0 + tr_x && \text{from Eq. (3.2.6)} \\ t &= \frac{w-y_0}{ry} \\ x_0 &= w - r_x t \\ x_0 &= w - r_x\left(\frac{w-y_0}{r_y}\right) \\ x_0 &= w - \frac{r_x w}{r_y} + \frac{r_x y_0}{r_y} \\ x_0 - w + \frac{r_x w}{r_y} &= \frac{r_x y_0}{r_y} \\ x_0\frac{r_y}{r_x} - w\frac{r_y}{r_x} + w &= y_0. \end{aligned} \tag{3.2.8}$$

When integrating Equation 3.2.8 over the intersection interval of x_0 we get

$$\begin{aligned} &\int_{w-r_x}^{w}\left(x_0\frac{r_y}{r_x} - w\frac{r_y}{r_x} + w\right)dx_0 \\ &= wr_x - \frac{r_x r_y}{2}. \end{aligned} \tag{3.2.9}$$

Equation 3.2.9 gives the area on the surface of the hole containing the rays intersecting plane **A**. We can confirm this by drawing the area on a top view of the hole as in Figure 3.6.

We can now use the same reasoning for intersections with plane **B**. This gives us

$$\begin{aligned} &\int_{w-r_y}^{w}(y_0\frac{r_x}{r_y} - w\frac{r_x}{r_y} + w)dy_0 \\ &= wr_y - \frac{r_x r_y}{2}. \end{aligned} \tag{3.2.10}$$

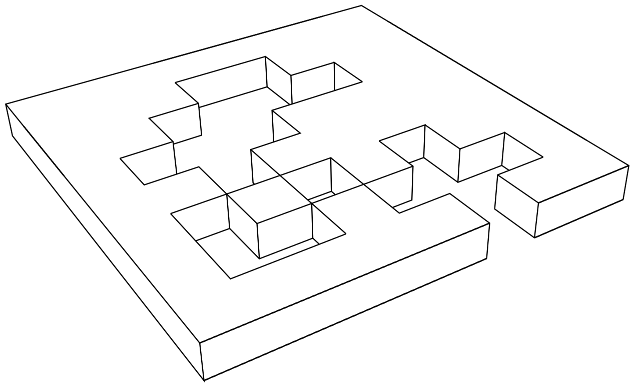


Figure 3.7 – Representation of a surface with a thickness with texels as columns of the same height.

By adding up Equations 3.2.9 and 3.2.10, we get the proportion of rays occluded by the walls in the form of an area. To get the desired ratios of occlusion and transparency, we divide the sum by the surface of the hole (w^2).

$$\rho_{occluded} = \frac{wr_y + wr_x - r_x r_y}{w^2} \tag{3.2.11}$$

$$\rho_{transparent} = 1 - \rho_{occluded}. \tag{3.2.12}$$

These proportions will be used to compute colors for the side walls and surfaces visible through the hole, as described in Section 3.4.

3.3. Representation of Thickness

One of the main contributions of our research is to simulate the effects due to an arbitrary thickness on a texture strictly composed of fully opaque and fully transparent texels. Our representation of the thickness is simply to consider the opaque texels to be columns of thickness h (see Figure 3.7). Another way to represent a thickness would be to have multiple layers of voxels. Since our model and calculations assume inner walls that are aligned with the texels and perpendicular to the surface, the use of voxels in this specific configuration would add complexity to the problem without adding much detail to the final result.

3.4. Sampling and Filtering

The equations shown in Section 3.2 were derived in a context where only one texel from the original texture is sampled. However, if we render a scene and only sample the original texture at the intersection, there will be some aliasing. A proven way of reducing such aliasing is to filter the texture before sampling. Modern rendering engines commonly use MIP mapping as a method of prefiltering. When we sample a texel from a MIP map level, we essentially sample a filtered texel consisting of multiple texels from a finer level. Therefore, we need to adapt our method to work with a pyramidal structure. This adaptation includes a way to derive and store useful data in the structure and a way to use it in our computations.

The current way to compute transparency in a MIP map is the same as it is with colors. Four texels from a level are filtered (usually with a simple box filter, so they are averaged) to form a combined texel. The alpha channels of the texels are simply averaged and the result is used as an indication of the level of transparency. This is not an exact calculation, as applying transparency one texel at a time over a different background color does not correspond to applying an averaged transparency over averaged background or even over different background colors. But it is often considered as a satisfying approximation. One way of adapting our result would be to simply take the averaged alpha channel and use it to approximate the number of holes. For example, a sample with an alpha channel value of 0.75 would have the following color

$$color = 0.75 * c_1 + 0.25 * (\rho_{occluded} * c_2 + \rho_{transparent} * c_\alpha)$$

where c_1 is the color from the sample, c_2 is the color of the side walls, and c_α is the color of the object behind the sampled surface (or background).

However, with this approximation we assume that all the holes are surrounded by four walls. This assumption would make it so that the filtered texture would always correspond to a simple checkered texture or a mostly opaque texture with one-texel-wide holes. This regular representation strongly limits the effects we desire to achieve. To represent more than the one-texel-wide hole scheme, we need to get information about the texels' surroundings. With this information, we should improve our approximation.

In our previous calculations, we focused on the intersection of a ray with a side wall. Therefore, we should do the same for the filtered data. Equations 3.2.9 and 3.2.10 represent areas on the surface of a hole. These areas are actually the projections of the walls onto the surface. The equations can be viewed as the difference between a rectangle and a triangle,

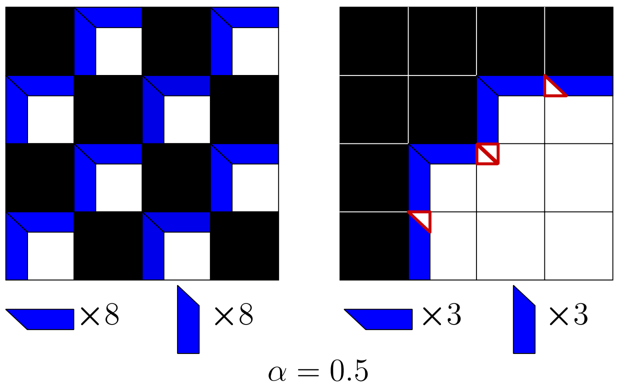


Figure 3.8 – Comparison of different texel patches with the same opaque to transparent ratio (eight texels full and eight texels transparent). Note how different the numbers of side walls are. In red wireframe, the missing triangles from our original calculations.

that can also be rewritten as trapezes.

$$wr_x - \frac{r_x r_y}{2} = \frac{((w - r_y) + w)r_x}{2} \tag{3.4.1}$$

$$wr_y - \frac{r_x r_y}{2} = \frac{((w - r_x) + w)r_y}{2}. \tag{3.4.2}$$

As both areas are calculated separately, this allows us to compute the correct transparency for a texel adjacent to a single side wall. This result inspires a new way of adapting our method to MIP maps. For each texel in the original texture, we analyze its surroundings and store the presence of walls adjacent to it for each cardinal direction (up, down, left, right). Then, when filtering four texels into one texel for the coarser MIP map level, we simply sum the number of side walls. Figure 3.8 shows how this better represents the desired effect as compared to using the alpha channel.

However, Figure 3.8 also shows that we would be missing some information (in red). Along the borders, some discontinuities appear at adjacent walls and at corners. The discontinuities take the form of triangles. They are the same triangles that are subtracted from the rectangles in Equations 3.2.9 and 3.2.10. These triangles appear at a texel that has a diagonal neighbor that is not obstructed by one or two other texels.

Our final method combines the previously mentioned number of side walls with the number of triangles. Each texel in a level l has nine integers associated with it. Each integer corresponds to the number of walls (projected for a direction) from the finer level, the number of triangles, and the number of opaque texels. To properly calculate $\rho_{occluded}$ and $\rho_{transparent}$

after sampling a texel from a texture's MIP map level, we need additional data. We need the number of original texels T that were combined, computed from the current MIP map level l.

$$T = 4^l.$$

With all these data we can now compute the desired ratios following Algorithm 3.1. With the proper ratios for any MIP map level, we can calculate a color for a sample

$$color = \rho_{opaque} * c_1 + \rho_{occluded} * c_2 + \rho_{transparent} * c_\alpha. \tag{3.4.3}$$

Knowing that our method is compatible with MIP maps because the sum of integers is linear, the filtering techniques mentioned in Section 2.4 can be utilized. For trilinear interpolation, instead of simply bilinearily interpolating four sampled colors at two adjacent MIP map levels, we use four colors computed with the proportions returned by Algorithm 3.1 and Equation 3.4.3. As Algorithm 3.1 simply represent the computation of equations and their variables without any loop, its time complexity is trivially $O(1)$.

Algorithm 3.1. Computation of visibility ratios for a texel t.

```
Let $\mathbf{r} := (r_x, r_y, r_z)$ be the scaled ray direction;
Let $l$ be the MIP map level;
Let $t$ be the sampled texel:
$t.walls$ :={top, left, bottom, right}
$t.triangles$ :={topRight, topLeft, bottomLeft, bottomRight}
$t.opaques$ :=opaqueTexels
Let $T := 4^l$
Let $w$ be the width of a texel in texture space
if $r_x > 0$, then $H := t.walls[3]$
  else if $r_x < 0$ , then $H := t.walls[1]$
  else $H := 0$
if $r_y > 0$, then $V := t.walls[0]$
  else if $r_y < 0$ , then $V := t[2]$
  else $V := 0$
if $(r_x > 0 \land r_y > 0)$, then $D := t.triangles[0]$
  else if $(r_x > 0 \land r_y < 0)$, then $D := t.triangles[3]$
  else if $(r_x < 0 \land r_y > 0)$, then $D := t.triangles[1]$
  else if $(r_x < 0 \land r_y < 0)$, then $D := t.triangles[2]$
  else $D := 0$
$O := H * wr_x - \frac{r_x r_y}{2} + V * wr_y - \frac{r_x r_y}{2} + D * \frac{r_x r_y}{2}$
$\rho_{occluded} := O/(Tw^2)$
$\rho_{opaque} := (t.opaques * w^2)/(Tw^2)$
$\rho_{transparent} := 1 - \rho_{occluded} - \rho_{opaque}$
```

3.5. OpenGL Implementation

To render transparent textures with thickness in a MIP map fashion, our method is implemented in the OpenGL 4.2 pipeline. First, the required geometric information is precomputed. The numbers of walls, triangles, and opaque texels are stored in three separate textures. The wall and triangle textures have four channels (one channel per direction) and the opaque texels texture has one channel. Each texel has therefore nine numbers associated with it.

MIP map structures are pre-generated and then passed to the pipeline. In the vertex shader, the tangent space coordinates for the fragment position and the camera position are computed and passed to the fragment shader. The texture coordinates are also passed to the fragment shader. In the fragment shader, with the tangent space camera position and fragment position, the view direction is computed and then scaled so the z component is equal to the thickness value that was passed to the shader as a constant. Algorithm 3.1 is used on eight different texels from two adjacent LODs of the MIP map to compute the colors required for the trilinear interpolation. The LOD value is obtained with built-in shader functions and split in two for its floor and ceiling values. The texel coordinates are computed from the texture coordinates. The values of c_1 and c_2 can be arbitrary or the original texture's color can be used for one or both. The value of c_α should be a fully transparent color (a color with a null α channel). Rendering transparent or semi-transparent objects in OpenGL requires careful preparations to make sure that all the objects are rendered in the right order. In our implementation we assume that all opaque objects are rendered first, and the semi-transparent objects are rendered last. Multiple layers of semi-transparent objects are not treated specially, so they exhibit the standard problems requiring ordered rendering.

Chapter 4

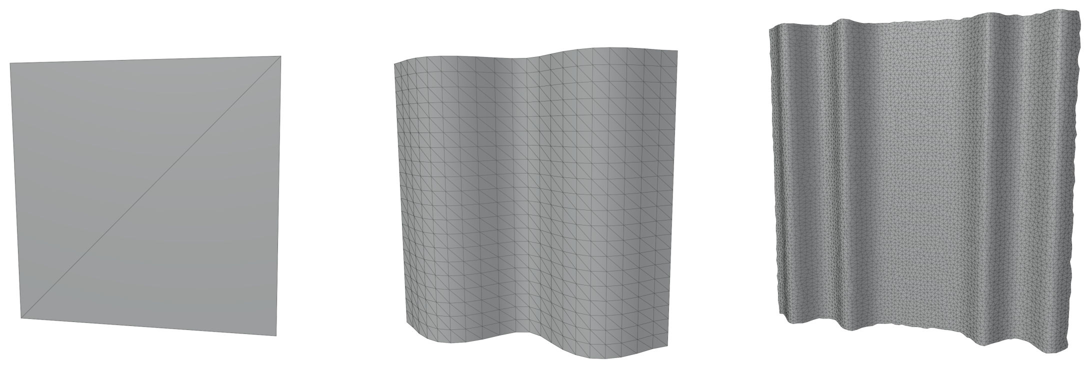
Results and Discussion

In this chapter, we present and discuss the results obtained with our method. The different scenes were rendered using our own code in C++ and OpenGL 4.2. The MIP maps for the color data and the side wall data were pre-generated. Trilinear filtering was computed in the fragment shader. The meshes were created in Blender [**Ble**], and the textures were drawn using GIMP [**GIM**]. In Section 4.1 we present images of simple scenes rendered with our method with different settings. In Section 4.2 we explain some limitations of our method. In Section 4.3 we discuss the performance of our method.

4.1. Different Effects Rendered with our Method

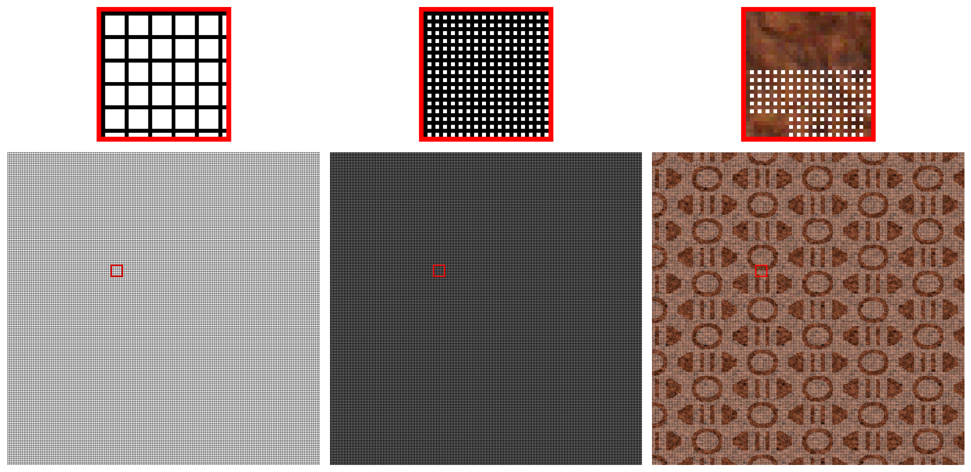
We first present results from a simple scene at different viewpoints. The scene is composed of a foreground mesh and a background mesh (that can be removed). We apply our method on the foreground mesh's texture and use the background mesh as a visual indicator to observe changes of color and transparency. We switch the foreground mesh between three meshes (see Figure 4.1). Three textures are used to test different sizes and distributions of holes (see Figure 4.2). The thickness of the surface is expressed in texel units, i.e., a surface with a thickness 1.0 has a thickness equal to the width/height of one texel, assuming a square texel.

We use the aforementioned scene configurations to see if our method achieves the goals we set for it. We investigate the impact of an added thickness to a semi-transparent surface. The effects should include a decrease in transparency as the viewing angle to the surface



(a) `Square (2 triangles)` (b) `Wave1 (800 triangles)` (c) `Wave2 (12800 triangles)`

Figure 4.1 – Foreground meshes used in our renderings.

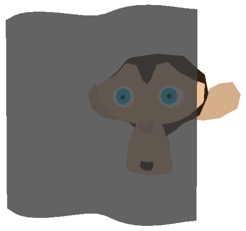


(a) Mosquito net (b) Textile (c) Curtain

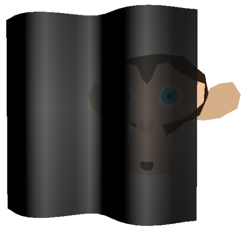
Figure 4.2 – Textures used in our renderings. White indicates a fully transparent hole. Any other color indicates a fully opaque texel of the corresponding color. In the actual texture, transparency is encoded in the α channel.

increases and a decrease in transparency as the thickness of the surface increases. These effects should not affect the smoothness of the texture if filtering is used, as our method was implemented to comply with a MIP map structure.

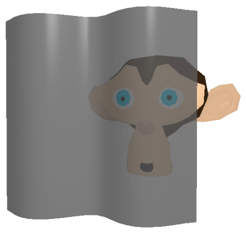
Figure 4.3 shows effects that our method has on a semi-transparent surface with a thickness as compared to a standard infinitely thin semi-transparent surface rendered with minimal shaders in OpenGL. The standard surface (a) is uniform between its silhouette and thus it appears flat. Its transparency α is computed as a distribution of fully opaque and fully



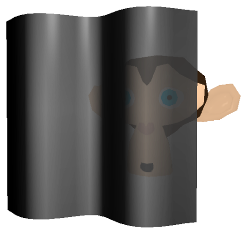
(a) Standard, non lit



(b) Thicker, non lit



(c) Standard, lit



(d) Thicker, lit

Figure 4.3 – Comparison of the surface rendering with the textile texture and transparency with standard α blending ($\alpha \approx 0.3$) (left) and using our thickness augmentation ($h = 1$) (right). Renderings include lighting (bottom) and exclude it (top).

transparent texels at the finest level of the texture. However, the surface with a thickness (b) shows a varying transparency depending on the viewing angle. Less light goes through the holes when the surface makes a stronger angle with the viewpoint. Adding lighting to the shaders does give some information on surface orientation to the standard surface (c), but the level of transparency stays constant. This figure illustrates well the visual effects that holes in a surface with a thickness brings up.

Figure 4.4 shows the impact of the value given to the thickness h. As h increases, the angle at which transparency is high enough to perceive the background mesh becomes narrower. Transparency is clearly nonlinear in function of the thickness. No shading is used on the semi-transparent surface, to better illustrate the visibility/occlusion variations due to the wavy nature of the curtain.

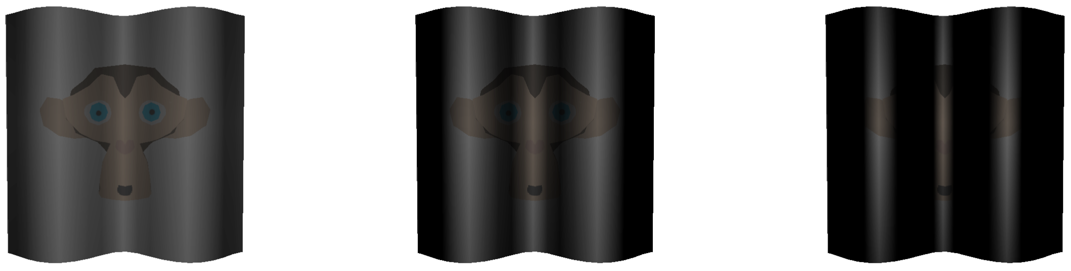

(a) $h = 0.5$ (b) $h = 1$ (c) $h = 2$

Figure 4.4 – Comparison of different thickness values (h) on the `Wave1` mesh with the textile texture.

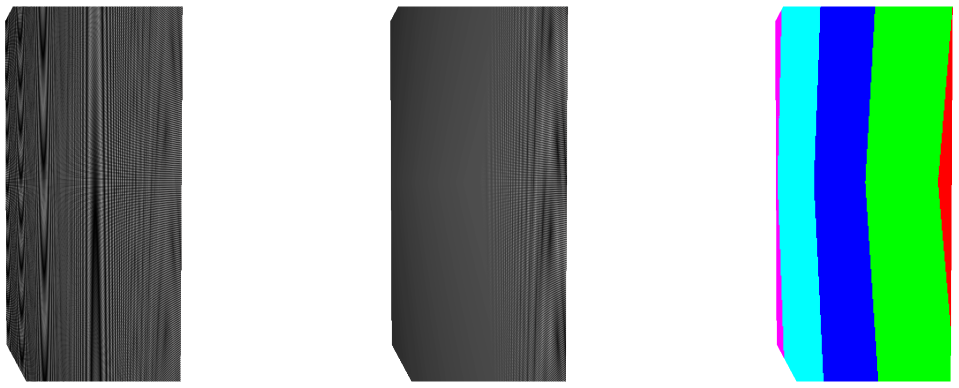
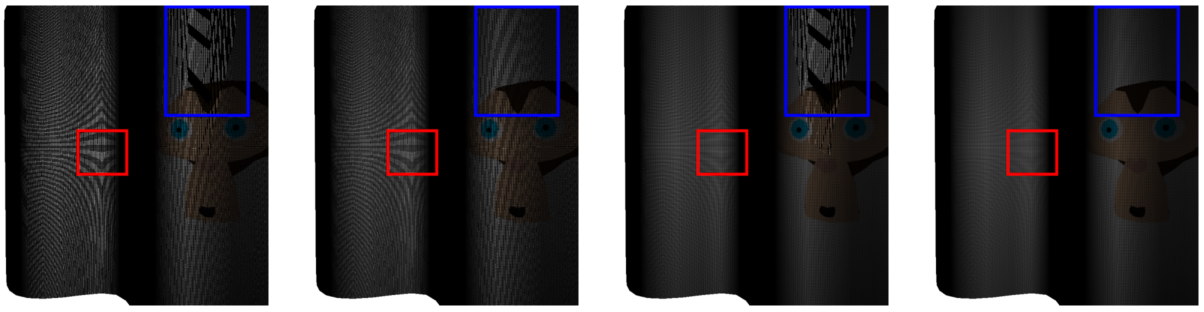

(a) No filtering (b) MIP map, trilinear filter (c) LODs

Figure 4.5 – Demonstration of the beneficial effects of MIP mapping for filtering when using our method with the `Square` mesh and the textile texture with a thickness of 0.1. Figure (c) shows the multiple color-coded LODs visible at that angle.

Figure 4.5 shows the importance of developing a method compatible with the MIP map structure. The aliasing caused by high frequencies in the texture are clearly visible in (a). Even in presence of such strong aliasing, the overall change in transparency of holes at steeper angles is still observable. Zones of the different LOD levels are color-coded in (c).

Figure 4.6 shows results from the different interpolations described in Section 2.4. The trilinear interpolation is more expensive, but its effects are very beneficial. The linear and



(a) Nearest (b) Linear (c) Bilinear (d) Trilinear

Figure 4.6 – Comparison of the four interpolation methods of MIP mapping described in Section 2.4, applied to our method.

bilinear interpolations both have smoother appearances when compared to the nearest neighbor (no interpolation). The linear interpolation (b) does not have a discontinuity (blue rectangle) when the LOD changes, but it still shows strong texel aliasing (red square). The bilinear interpolation (c) smooths out some texel aliasing (red square), but still exhibits visible changes of LOD (blue rectangle). The trilinear interpolation (d) combines the advantages of linear and bilinear interpolations. Note that aliasing is still visible because of the very high frequency in the texture used, but it is much improved.

We are able to use colored textures with more irregular holes to add more details without disturbing the effects due to the thickness (see Figure 4.7). The color pattern is not disturbed by the changes of transparency and the background stays visible.

Our method is compatible with backfacing surfaces and with order independent transparency algorithms. Very often, engines will default to culling the faces of a mesh that are not facing the camera in order to save on rendering time. This is common in modeling sheet-like surfaces that can be viewed from either side. However, if this culling is disabled, our method works as expected. Moreover, as our method outputs a color and a transparency level (α) in the fragment shader, order independent transparency can be used without changes to our implementation.

4.2. Limitations

As our method is an approximation with a number of assumptions, it comes with some compromises and shortcomings in accuracy and realism. The width of the holes has an

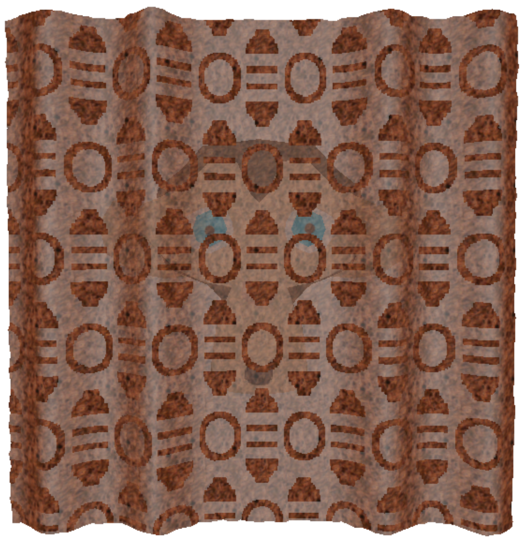


Figure 4.7 – `Wave2` mesh rendered with the curtain texture and a thickness of 0.5.

impact on accuracy in relation with the LOD, as the number of walls for a texel is computed only with regards its direct neighbors. Therefore, a transparent texel without any direct opaque neighbor in some direction will not contribute to the increase in opacity that is expected from a certain view angle (see Figure 4.8). Figure 4.9 demonstrates this effect by comparing a rendered image of the `Wave1` mesh to a ground truth image. The ground truth image was generated with ray tracing with a sample rate of 1024 rays per pixel. The 2D version of a voxel traversal algorithm [**AW87**] was used as a texel traversal algorithm to check if a ray would intersect an inside wall (see Figure 4.10). Using the projection of the scaled ray direction vector on the surface, the traversal algorithm checks if any texel in that direction is opaque. The length vector is used to stop the traversal. If any opaque texel is encountered, the ray intersects an inside wall.

Such an algorithm could be used at very steep viewing angles in OpenGL when the texture possesses larger holes. The algorithm can also be used when the texture needs to be magnified when observed from close distance (i.e., when the number of texels per pixel is smaller than one) in OpenGL. Performance would be affected by doing so. The inaccuracies caused by this limitation depend on the width of the holes and on the thickness of the

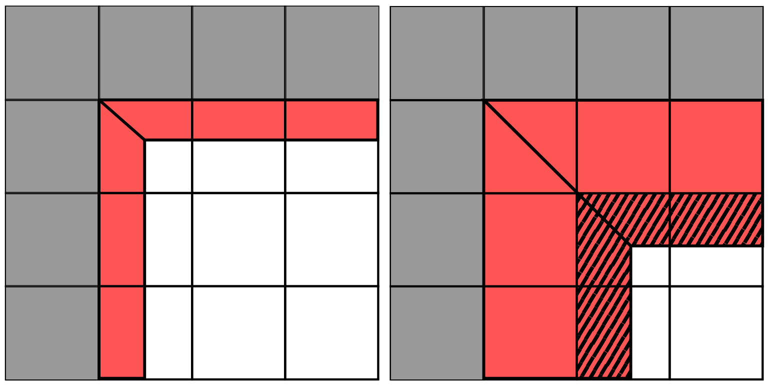


Figure 4.8 – Demonstrating the limitation caused by larger holes in a structure based on directly adjacent texels. Projection of the walls on the surface at a mild viewing angle (a). Projection of the walls on the surface at a much steeper viewing angle (b). The hatched area indicates the expected opacity that is not represented by our method, because it is visible through transparent texels further away from the opaque texels.

surface. As the holes grow in depth, the inaccuracy increases. For smaller holes (one or two texel wide) the thickness does not affect accuracy as much as with larger holes. Larger holes over a very small thickness would give accurate results though, as it would take very steep angles to notice the lack of opacity. A bigger thickness will show inaccuracies much smaller view angles.

Our method is built upon a number of assumptions, some of which limit the use of our method in certain situations. We assume the walls inside the holes are perpendicular or parallel to other adjacent walls and perpendicular to the surface itself. That assumption limits our ability to properly apply a shading model if we want the walls to be highly specular whilst having varying roughness or orientations.

4.3. Performance

We measured the performance of our method to justify its use in a real-time context. All results and performance tests were computed on a Windows 10 system equipped with an Intel® Core™ i7-10750H processor clocked at 2.60 GHz, 16 GB of memory, and an Nvidia GeForce 1650 Ti graphics card clocked at 1035 MHz with 4 GB of video memory. The vertical synchronization (VSync) was turned off in the OpenGL context and in the Nvidia

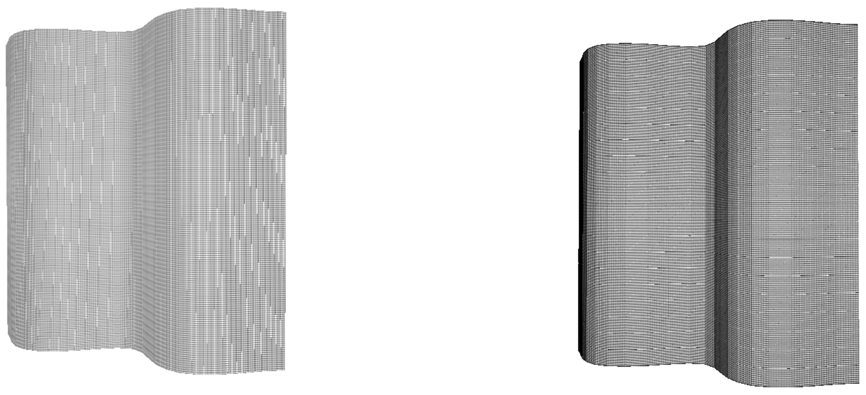

(a) Render (b) Ground Truth

Figure 4.9 – Inaccuracy in the opacity proportion with larger holes. The texture used is the mosquito net from Figure 4.2(a). The holes are 5 × 5 empty texels surrounded by a row/column of one opaque texel. Note that even the ground truth image suffers from aliasing.

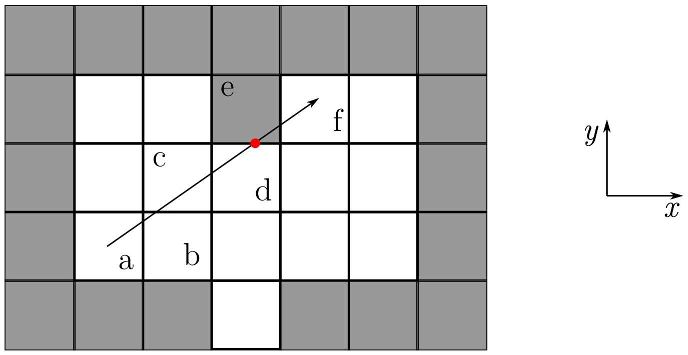


Figure 4.10 – Texel traversal visualization. The ray traverses the texture and visits texels a,b,c, and d before encountering texel e and stopping.

control panel to ignore the refresh rate of the computer screen. When VSync is activated, the rendering loop waits a few frames when the execution is too fast for the screen's refresh rate.

Table 4.1 and Figure 4.11 show the average drawtime per frame of the scene `Wave1` (without background) with and without the use of our method. We queried and averaged the time elapsed during the execution of the drawing command over 280 frames. The measured increase in drawtime is within our expectations. First, the vertex shader requires additional computations to pass tangent space information to the fragment shader. Also, the trilinear interpolation in the fragment shader for a standard rendering of one pixel requires 8 texture fetches per fragment pass. Our method requires 32 fetches because 4 textures are used (albedo, walls, corners, opaques). For these 24 extra fetches, additional computations are also done every three fetches to compute semi-transparent colors. We believe that these are acceptable rendering times in order to render thicker semi-transparent surfaces that would require costly fine geometry instead, and without pre-filtering capacity. Careful optimization of the fragment shader operations could reduce the drawtime as most of the operations are done there. Figure 4.11 also suggests that the resolution of the mesh does not affect performance more than we already expect it to.

Changing the thickness value of a texture has a visible impact on the result (see Figure 4.4). However, since the thickness is simply represented by a coefficient used to scale a vector, its value will not affect rendering performance.

Table 4.1 – Average drawtime per frame over 280 frames of different mesh resolutions of `Wave1`, with and without the use of our method.

Triangles	Standard (ms/f)	Thick (ms/f)
100	0.012	0.105
399	0.013	0.109
800	0.014	0.116
3200	0.021	0.148
9618	0.045	0.185
12800	0.055	0.214

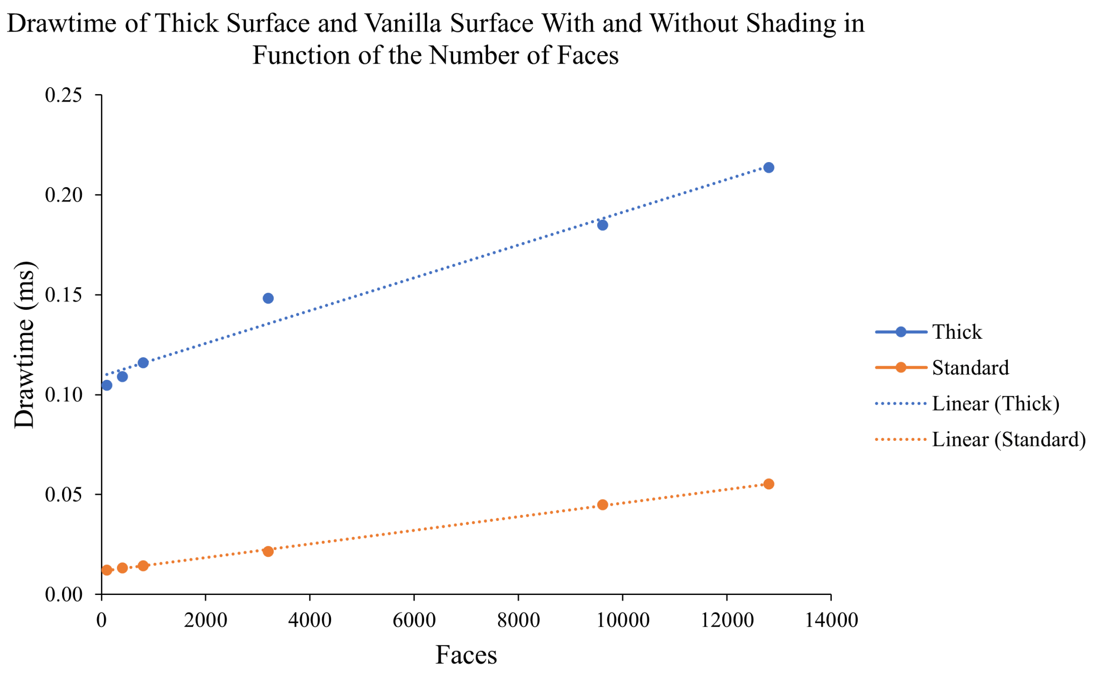


Figure 4.11 – Average drawtime per frame over 280 frames of different mesh resolutions of `Wave1`, with and without the use of our method.

Chapter 5

Conclusion and Future Work

In this thesis, we presented a method to simulate thickness for a semi-transparent surface using textures composed of opaque and fully transparent texels. We simulate the effects of thickness on transparency by approximating the interaction that light would have with the inside walls of a surface covered with small holes. The method works well on surfaces even if they are viewed from underneath if `GL_CULL_FACE` is disabled in OpenGL (or similar feature in other APIs or rendering engines).

Our method successfully simulates the changes in opacity on a surface as the viewing angle changes. Order preserving α blending rules in OpenGL are the same for our method, and the transparency output of our method can be used with order independent transparency algorithms.

We use a texture-like representation that is supported by a MIP map structure and therefore can be adapted to modern real-time rendering engines. This compatibility allows our method to gain the benefits of the MIP map filtering processes. The performance of our implementation achieves mostly real time, and offers quality visual effects related to surfaces with a certain level of thickness. Some simple future improvements include having rectangular texels rather that square, and handling semi-transparent texels rather than fully opaque texels.

The accuracy of our approximation can be affected by the size of the holes in the surface. As the holes get larger, the opacity is lower than what we compute. An improvement would be to store the distance (or pointer) of a transparent texel to the closest opaque texel for each direction. That information could be fetched when dealing with thicker materials, steep viewing angles, or a combination of both. The results would still be approximate though.

A possible extension to our method would be to consider the inside walls to be curved vertically and/or horizontally, instead of axis aligned. Adding such attributes to the walls could produce more accurate specular shading effects. A representation based on Signed Distance Fields (SDF) [**LB05**] could prove efficient and also useful for the distance in larger holes discussed in the previous paragraph. However, rendering complexity could increase significantly.

Ambient occlusion having become very important to render more realistic scenes, the inclusion of an ambient occlusion term to our transparency equation could improve realism of shading results, as less and less light can reach deeper narrow holes. Global illumination applied locally within holes could also improve the appearance of the observed surface, especially if the inside of the holes are defined as reflective materials.

We built our method with the assumption that a diffuse texture and a hole texture combined would have the same resolution. It would be interesting to study the effect of our method on a low resolution hole texture that would effectively represent larger holes in a high resolution context. We believe changing the *uv* mapping and experimenting with texture tiling could give other interesting results.

Considering the higher memory usage of our method, it would be interesting to see if generating only a part of the MIP map pyramid could prove a good trade-off in terms of memory and appearance. Also, it is possible to pre-compute a semi-transparency function based on our derived equations that would serve only as a way of achieving varying semi-transparency. Such a function would only represent one hole pattern and the surface would only display its standard color without the details of the holes.

References

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. *Proceedings of EuroGraphics*, 87(3):3–10, August 1987.

[Ble] Blender Online Community . Blender - a 3D modelling and rendering package. http://www.blender.org.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, August 1978.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, October 1976.

[DHI+13] Jonathan Dupuy, Eric Heitz, Jean-Claude Iehl, Pierre Poulin, Fabrice Neyret, and Victor Ostromoukhov. Linear efficient antialiased displacement and reflectance mapping. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2013)*, 32(6), November 2013.

[Fou92] Alain Fournier. Filtering normal maps and creating multiple surfaces. Technical Report TR-92-41, Department of Computer Science, University of British Columbia, Vancouver, BC, Canada, 1992.

[GIM] GIMP Development Team. Gimp. https://www.gimp.org.

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Technical Report UCB/CSD-89-516, EECS Department, University of California, Berkeley, June 1989.

[Kaj85] James T. Kajiya. Anisotropic reflection models. *SIGGRAPH Comput. Graph.*, 19(3):15–21, July 1985.

[LB05] Charles Loop and James Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph.*, 24:1000–1009, January 2005.

[Max88] Nelson L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4:109–117, 1988.

[OB10] Marc Olano and Dan Baker. Lean mapping. In *Proceedings of I3D 2010: The 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 181–188, February 2010.

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 359–368, 2000.

[POC05] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. *ACM Transactions on Graphics*, 24(3):935, July 2005.

[Sha49] C.E. Shannon. Communication in the presence of noise. *Proceedings of the Institute of Radio Engineers*, 37(1):10–21, 1949.

[Wil83] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, July 1983.