SHADOW HEIGHT MAPS FOR TERRAIN SHADOWING

by

Ufuk Can Biçici

B.S., Computer Science Engineering, Marmara University, 2009

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2012

SHADOW HEIGHT MAPS FOR TERRAIN SHADOWING

APPROVED BY:

Prof. Lale Akarun (Thesis Supervisor)	.....
Prof. Bekir Tevfik Akgün	.....
Prof. Can Özturan	.....

DATE OF APPROVAL: 04.06.2012

ACKNOWLEDGEMENTS

I would like to first thank my thesis advisor Prof. Lale Akarun for her guidance, assistance and encouragement throughout this thesis. Each time when I was technically or motivationally challenged, she was always there to help me with her vast knowledge and unequalled kindness. It was a honor and pleasure for me to work with her.

I would like to thank Prof. Bekir Tevfik Akgün and Prof. Can Özturan for kindly accepting to become committee members. I would also like to thank Cem Keskin for his theoretical and practical advises, Eray Berger for tolerating my six months long absence and Gökhan Acar for our entertaining conversations on the way home.

I would like to thank my family for their support and tolerance throughout this thesis.

At last but certainly not the least, I wish to express my deepest gratitude to Burcu Büyükeren, for her unending support, understanding and patience. She was always by my side, supporting me through all the difficult times without ever losing her faith and patience for me. I deem myself very lucky to have met her in my life.

This work is dedicated to my previous advisor Dr. Ali Vahit Şahiner who passed away unexpectedly at an early age. He will always have a special place in our hearts.

I would like to thank TÜBİTAK for its financial support of this thesis. The author of this thesis is financially supported by TÜBİTAK BİDEB 2210 National Master Scholarship Program.

ABSTRACT

SHADOW HEIGHT MAPS FOR TERRAIN SHADOWING

Shadowing terrains is a challenging problem. Mainstream shadowing algorithms like shadow volumes and shadow maps, do not work very well with height map geometry. Shadow volumes need silhouette extraction which is a very expensive operation for a height map tessellated into thousands of triangles. Shadow mapping is a better alternative, but it suffers from aliasing and self-shadowing artifacts. Additionally, shadow mapping needs two rendering passes to function, this means rendering the height map twice in each frame. In this thesis, we propose a new specialized terrain shadowing algorithm to generate shadows caused by the directional light of the Sun. We propose the use of shadow height map (SHM), which is a discretized version of the shadow volume of the height map. A pixel in the SHM texture represents the height of the shadow on the current height map location. It is a hybrid method which borrows elements from shadow mapping, shadow volumes and ray casting. For each sunlight direction we generate a new SHM, by casting parallel sunlight rays onto the terrain. We rotate the SHM texture, such that each ray corresponds to a single column of the image and we apply a simple, iterative 2D shadow volume extraction algorithm on each column, storing the current height of the shadow to the current pixel in the column, as the algorithm progresses. A pixel of an arbitrary object or to the terrain compares its world height with the sampled shadow height from the SHM and marks itself in shadow if it is below the shadow surface. We define two models to simulate realistic soft shadows, first, by using a constant penumbra and second, by using an adaptively varying penumbra height. According to our tests, our algorithm performed better than the state-of-the-art shadow mapping variant, Parallel Split Shadow Mapping (PSSM), both in the quality and performance aspects.

ÖZET

ARAZİ GÖLGELEME PROBLEMİ İÇİN GÖLGE YÜKSEKLİK HARİTALARI'NIN KULLANIMI

Arazilerin gölgelenmesi zor bir problemdir. Gölge hacimleri ve gölge haritaları gibi genel algoritmalar yükseklik haritası geometrisi ile çok iyi çalışmamaktadır. Gölge hacimleri binlerce üçgene bölünmüş olan yükseklik haritaları için çok pahalı bir operasyon olan silüet çıkarımını gerektirir. Gölge haritaları daha iyi bir alternatif olmakla beraber kendini gölgeleme ve kırpıklanma hataları ortaya çıkarır. İlaveten gölge haritaları çalışmak için iki çizim geçişine ihtiyaç duyar; bu da yükselik haritasının her sahnede iki kez çizilmesi anlamına gelir. Biz bu tezde güneşin doğrusal ışığından kaynaklanan arazi gölgelerini oluşturmak için yeni bir algoritma öneriyoruz. Bunun için yükselik haritasının gölge hacminin ayrık bir versiyonu olan gölge yükseklik haritasının (GYH) kullanımını öneriyoruz. GYH'deki bir piksel gölgenin mevcut yükseklik haritası lokasyonu üzerindeki yüksekliğini temsil eder. Bu gölge haritalarından, gölge hacimlerinden ve ışın izleme yöntemlerinin parçalarından oluşan hibrit bir metoddur. Her güneş ışığı yönü için arazi üzerine parallel güneş ışığı ışınları fırlatarak yeni bir GYH oluşturuyoruz Araziye veya herhangi bir yüzeye ait bir piksel kendi yüksekliğini GYH'den örneklediği gölge yüksekliği ile karşılaştırır ve gölge yüzeyinin altında ise kendisini gölgede olarak işaretler. Gerçekçi yumuşak gölgeleri simüle edebilmek için iki model tanımlıyoruz. Bunlardan ilki sabit yarıgölge yükseliği, ikincisi de adaptif olarak değişebilen yarıgölge yüksekliği kullanmaktadır. Testlerimize göre algoritmamız modern bir gölge haritası türevi olan Paralel Bölünmüş Gölge Haritaları (PBGH) yönteminden hem kalite hem de performans bakımından daha üstün çalışmaktadır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES ix
LIST OF TABLES xix
LIST OF SYMBOLS xx
LIST OF ACRONYMS/ABBREVIATIONS xxi
1. INTRODUCTION 1
2. TERRAIN RENDERING 7
2.1. The Basics of Terrain Rendering 7
2.2. Level of Detail: Definition and Related Work 11
2.3. Implementing GeoMipmapping 13
2.3.1. Error Metric for LOD Transitions 13
2.3.2. Our Index Buffer Based Method for Stitching Edge Cracks . . . 15
2.3.3. GeoMorphing Implementation 18
2.4. Terrain Texturing 21
2.5. Terrain Shading 22
3. SHADOWS IN COMPUTER GRAPHICS 24
3.1. Terminology and Mathematics for Shadows and Definition of the Problem 25
3.2. Projection Shadows 35
3.2.1. Hard Shadows 35
3.2.2. Soft Shadows 38
3.2.3. Projective Shadows for Curved Surfaces 41
3.3. Shadow Volumes 42
3.3.1. The Basic Algorithm and Heidmann's Z-Pass Method 43
3.3.2. The Z-Fail method or "Carmack's Reverse" 44
3.3.3. ZP+ Shadow Volumes 46
3.3.4. Speed Improvement Strategies for Shadow Volumes 46
3.3.5. Summary 51

3.4. Shadow Mapping 51
3.4.1. The Basic Algorithm and Errors 52
3.4.1.1. Depth Bias 53
3.4.1.2. Aliasing 55
3.4.2. Hard Shadow Mapping Methods and Reduction of Sampling Error 59
3.4.2.1. Focusing (or Fitting) the Shadow Map 59
3.4.2.2. Warping Based Methods 60
3.4.2.3. Global Partitioning: Parallel Split Shadow Maps . . . 62
3.4.2.4. Other Strategies For Sampling Error Reduction 65
3.4.3. Filtering Hard Shadows and Shadow Map Reconstruction Techniques 66
3.5. Physically Based Soft Shadows 68
3.6. Methods Specialized for Terrain and Height Map Shadowing 71
4. SHADOW HEIGHT MAPS FOR TERRAIN SHADOWING 81
4.1. Basic Idea of Shadow Height Maps and Summary of Our Method . . . 81
4.2. Literature Survey: Shadow Height Maps in Previous Works 85
4.3. The Rendering Pipeline and The GPGPU Technology 91
4.3.1. The Rendering Pipeline and Shader Technology 92
4.3.2. The GPGPU Technology 94
4.4. The Terrain Shadowing Algorithm: The Basic Case for a Static Directional Light 97
4.4.1. Algorithm Initialization 98
4.4.2. The Transformation Step and CUDA Side 1 101
4.4.3. CUDA Side 2: The 2D Shadow Volume Extraction Algorithm . 108
4.4.4. SHM in the Rendering Pipeline: Vertex and Pixel Shaders . . . 113
4.5. Visibility Determination and Soft Shadow Simulation 113
4.5.1. The Constant Penumbra Height Approach 113
4.5.2. The Varying Penumbra Height Approach 115
4.6. Bilinear Interpolation as the Reconstruction Method 119
4.7. Moving Light Source and Temporal Aliasing Problem 121
4.7.1. The Reason 121

4.7.2. Method 1: Interpolation between Two SHMs 123
4.7.3. Method 2: Temporal Low Pass Filter 125
4.7.4. Method 3: Jittering 128
4.7.5. Method 4: Defining a Threshold for Penumbrae 130
4.8. Other Minor Problems of the Algorithm and Our Solutions 131
4.8.1. Shadow "Sides" Problem 131
4.8.2. Reducing Memory Consumption 133
4.9. Comparison of Our Method with Shadow Mapping 135
5. EXPERIMENTS 143
5.1. The Methodology 143
5.2. Quality Comparisons 146
5.2.1. Results 146
5.2.2. Comments 159
5.3. Performance Comparisons 162
5.3.1. Results 162
5.3.2. Comments 170
6. CONCLUSION AND FUTURE WORK 172
REFERENCES 175

LIST OF FIGURES

Figure 2.1. Different terrain representation techniques. Image taken from [1]. . 8

Figure 2.2. Quadtree space partitioning and frustum culling. 10

Figure 2.3. On the left, two successive block resolutions for a GeoMipmapping block. The image on the right shows the definition of the height displacement error. Images taken from [2]. 15

Figure 2.4. Edge regions and the inner region of a terrain block. 16

Figure 2.5. A dense block with the dimension 17^2 adapts its edge according to a coarser block of the dimension 5^2. Note the triangle pattern of the stitching edge. 17

Figure 2.6. The vertex v slowly travels to its projection as the GeoMorphing progresses. Note that at the next level, there won't be a vertex at the location indicated by the red dot. 19

Figure 2.7. The world space normal map for a 1024^2 height map. 23

Figure 3.1. Depiction of the shadow terminology with umbra, penumbra, light source, occluder and receiver. Image taken from [3]. 25

Figure 3.2. How different light source types create different types of shadows. 34

Figure 3.3. An axis aligned planar receiver (left) and an arbitrarily oriented one (right). Image based on [3]. 36

Figure 3.4. How the method of Heckbert and Herf transforms a pyramid into a parallelepiped which effectively culls out objects behind the light source and shadow receiver. Image taken from [3]. 39

Figure 3.5. Gooch *et al.*'s method moves the receiver plane up and down(right) instead of sampling the area light source at different points(left). Image based on [3]. 41

Figure 3.6. The shadow volume of a triangle, generated by a point light source. 42

Figure 3.7. Heidmann's Z-Pass Algorithm. 44

Figure 3.8. The Shadow Volumes algorithm in work: We increment the counter as we intersect front-facing shadow polygons and decrement when we intersect back-facing ones. Image based on [4]. 45

Figure 3.9. The Z-Fail Algorithm. 45

Figure 3.10. Shadow caster culling as proposed in [5]. Casters which are completely in shadow or casting shadows which are not visible to eye is eliminated. Image taken from [5]. 47

Figure 3.11. Adaptive utilization of Z-Fail and Z-Pass approaches can lower stencil buffer updates. Image based on [6]. 49

Figure 3.12. On the left, the caster and receiver AABB's are shown projected on the light image plane. On the right, the depth intervals of the actual casters are shown. Image based on [5]. 49

Figure 3.13. On the left, the shadow volume corresponding the current slice is empty and omitted from rendering. On the left, it intersects with a receiver, meaning the shadow volume interval for this slice must be rendered. Image taken from [5]. 50

Figure 3.14. The Standard Shadow Mapping Algorithm. 53

Figure 3.15. The basic shadow map algorithm, visualized. Image taken from [7]. 53

Figure 3.16. The z-fighting and incorrect self shadowing error caused by shadow map undersampling and imprecision. Image taken from [8]. 55

Figure 3.17. Simplified sampling error analysis for shadow mapping aliasing. Image taken from [8]. 57

Figure 3.18. Projection aliasing (left) and perspective aliasing (right). Image based on [8]. 59

Figure 3.19. Shadow map focusing under point light (left) and directional light (right). Image based on [6]. 60

Figure 3.20. On the left, the uniform shadow map sampling is shown. On the right, the reparametrization of shadow map samples is shown, after the perspective warp. Image based on [8]. 62

Figure 3.21. Parallel splitting of the view frustum. On the right, logartihmic and uniform splittings are mixed and the practical splitting scheme is obtained. Image taken from [7]. 64

Figure 3.22. Percentage closer filtering. Directly filtering depth values does not produce any blur, since the result will again be binary. Filtering shadow test values produce a soft shadow, instead. The depth values are courtesy of [9]. 67

Figure 3.23. The penumbra region of an area light source changes with the distances between the source-blocker and blocker-receiver. Image based on [10]. 70

Figure 3.24. Horizon angle in a given direction is the smallest angle at which a point p on the terrain can see the sky unblocked. Image taken from [11]. 74

Figure 3.25. The circular visibility aperture on the unit hemisphere(left) and the projection of the light source on it(right). The intersection area between the circles gives amount of illumination. Image based on [12]. 76

Figure 3.26. The quad-directional visibility cones are interpolated into a single cone for light visibility test. Image based on [13]. 77

Figure 3.27. The linear horizon extraction algorithm outputs the horizon angle of a new sample in constant time, using the horizon information from older samples (left). A sweep direction on the height map can be seen on the right. Images taken from [14]. 78

Figure 3.28. Height samples are taken along the light direction to calculate occlusion contributions of each sample in Tatarchuk's method. Image based on [15]. 80

Figure 4.1. The basic Shadow Height Map visibility test algorithm. 82

Figure 4.2. Shadow height map of a height map based terrain. Both the terrain and the tree receive shadows from the shadow height map, without distinction. 83

Figure 4.3. On the left, the discretized shadow volume, i.e. the shadow height map is shown. The original height map is shown on the right. Image taken from [16]. 87

Figure 4.4. Type 1 shadows which are exactly aligned with x or z axes (a) and Type 2 shadows which have an arbitrary direction (b). Image taken from [17]. 89

Figure 4.5. The Direct3D 9 rendering pipeline. Image taken from [18]. 92

Figure 4.6. CPU architecture compared with GPU architecture. Note the immense amount of ALU circuitry in the GPU. Image taken from [19]. 96

Figure 4.7. Different types of memories in CUDA and how they correspond to the thread hiearchy. Image taken from [19]. 96

Figure 4.8. The flow diagram of our algorithm, when used with a static light source. 98

Figure 4.9. On the left, the sun light's projection on xz-plane makes the angle α with negative z-axis. On the right, the terrain is rotated, such that the sun light exactly coincides with negative z-axis. 100

Figure 4.10. The SHM texture with the size of M^2 and the height map placed in the center of it, with the size of R^2N^2. Each SHM column represents a sun ray. The red dots indicate locations of shadow height samples. 104

Figure 4.11. Initial values of the columns $[C_i, C_{i+4}]$. C_{i+2} enters the terrain at (x_i, z_i). 108

Figure 4.12. In this sample figure, 20 CUDA threads for 20 SHM columns have been launched. Assuming an imaginary system where the maximum thread/block number is 5, 4 different CUDA thread blocks are used in the kernel launch. 110

Figure 4.13. Shadow volume extraction on a 2D height map slice. Notice how the current shadow height jumps to assume the heights of the silhouette points. 111

Figure 4.14. Visibility determination with constant penumbra height p. y_0 is completely lit since $y_0 > SHM_0$. y_1 is completely in shadow since $SHM_1 - y_1 > p$. y_2 is partially shadowed because $p > SHM_2 - y_2 > 0$.115

Figure 4.15. Calculation of the varying penumbra height. 117

Figure 4.16. Reconstructing shadow heights from the SHM with bilinear interpolation. 119

Figure 4.17. Two SHMs of successive frames for a rotating light direction. Notice the changing sampling locations. 122

Figure 4.18. Difference between silhouette extraction (ground truth) and SHM ray casting. Notice how the changing ray locations cause the shape of the shadow to differ between successive frames in SHM method. 123

Figure 4.19. Shadow interpolation illustrated: We interpolate over an angle of $\Delta\alpha = N\theta$, as the shadow of the $Frame\ i$ disappears, the shadow of the $Frame\ i+1$ will start to appear. 125

Figure 4.20. Three successive SHMs for a temporal filter of size $K = 1$. The current pixel's location in each SHM is shown by the yellow dot. . 128

Figure 4.21. Aligning the left most and top most corners of the rotated height map with SHM edges provides a natural source of jittering. 129

Figure 4.22. A part of the colored SHM texture. Red pixels show samples shadowed, white pixels show unshadowed sampling locations. Cyan pixels show potential "shadow side" artifact locations. At these pixels we add a bias to the shadow height. 132

Figure 4.23. Actual images from our program showing the "shadow side" error. On the left image, we can see how the shadow is cut off abruptly. The image on the right shows the same location with our shared memory fix. 132

Figure 5.1. (a) On the left, our high frequency height map. (b) On the right, our low pass filtered height map. 145

Figure 5.2. Actual rendered version of the high frequency height map of Figure 5.1a. 146

Figure 5.3. Actual rendered version of the low frequency height map of Figure 5.1b. 146

Figure 5.4. SHM Distance Shot. Const. Pen. SHM Falloff:0.55 Penumbra Height:1.5 Bias:30 FPS:772 147

Figure 5.5. PSSM Distance Shot. PSSM Split Count:3 Shadow Map Resolution:1024^2 PCF Kernel:3^2 FPS:183 147

Figure 5.6. PSSM Split Count:1 Shadow Map Resolution:1024^2 PCF Kernel:2^2 FPS:384 148

Figure 5.7. PSSM Split Count:1 Shadow Map Resolution:1024^2 PCF Kernel:5^2 FPS:278 148

Figure 5.8. PSSM Split Count:1 Shadow Map Resolution:2048^2 PCF Kernel:3^2 FPS:248 149

Figure 5.9. PSSM Split Count:3 Shadow Map Resolution:2048^2 PCF Kernel:3^2 FPS:115 149

Figure 5.10. PSSM Split Count:2 Shadow Map Resolution:4096^2 PCF Kernel:2^2 FPS:60 150

Figure 5.11. PSSM Split Count:4 Shadow Map Resolution:4096^2 PCF Kernel:5^2 FPS:29 150

Figure 5.12. Constant Pen. 32-bit SHM Falloff:0.55 Penumbra Height:1.5 Bias:30 FPS:839 151

Figure 5.13. Constant Pen. 16-bit SHM Falloff:0.55 Penumbra Height:1.5 Bias:30 FPS:861 151

Figure 5.14. Varying Pen. 64-bit SHM Falloff:0.2 Penumbra Height:0.2 Bias:0 FPS:867 152

Figure 5.15. Varying Pen. 32-bit SHM Falloff:0.2 Penumbra Height:0.2 Bias:0 FPS:863 152

Figure 5.16. SHM Distance Shot. Const. Pen. SHM Falloff:0.55 Penumbra Height:1.2 Bias:30 FPS:616 153

Figure 5.17. PSSM Distance Shot. PSSM Split Count:3 Shadow Map Resolution:1024^2 PCF Kernel:3^2 FPS:150 153

Figure 5.18. PSSM Split Count:1 Shadow Map Resolution:1024^2 PCF Kernel:2^2 FPS:374 154

Figure 5.19. PSSM Split Count:3 Shadow Map Resolution:1024^2 PCF Kernel:5^2 FPS:133 154

Figure 5.20. PSSM Split Count:1 Shadow Map Resolution:2048^2 PCF Kernel:2^2 FPS:281 155

Figure 5.21. PSSM Split Count:3 Shadow Map Resolution:2048^2 PCF Kernel:5^2 FPS:97 155

Figure 5.22. PSSM Split Count:1 Shadow Map Resolution:4096^2 PCF Kernel:2^2 FPS:134 156

Figure 5.23. PSSM Split Count:4 Shadow Map Resolution:4096^2 PCF Kernel:5^2 FPS:32 156

Figure 5.24. Constant Pen. 32-bit SHM Falloff:0.55 Penumbra Height:1.0 Bias:30 FPS:632 157

Figure 5.25. Constant Pen. 16-bit SHM Falloff:0.55 Penumbra Height:1.0 Bias:30 FPS:645 157

Figure 5.26. Varying Pen. 64-bit SHM Falloff:0.55 Penumbra Height:0.2 Bias:30 FPS:647 158

Figure 5.27. Varying Pen. 32-bit SHM Falloff:0.55 Penumbra Height:0.2 Bias:30 FPS:632 158

Figure 5.28. Results for 1 split PSSM vs. SHM on the low frequency terrain. . 162

Figure 5.29. Results for 2 split PSSM vs. SHM on the low frequency terrain. . 163

Figure 5.30. Results for 3 split PSSM vs. SHM on the low frequency terrain. . 164

Figure 5.31. Results for 4 split PSSM vs. SHM on the low frequency terrain. . 165

Figure 5.32. Results for 1 split PSSM vs. SHM on the high frequency terrain. . 166

Figure 5.33. Results for 2 split PSSM vs. SHM on the high frequency terrain. . 167

Figure 5.34. Results for 3 split PSSM vs. SHM on the high frequency terrain. . 168

Figure 5.35. Results for 4 split PSSM vs. SHM on the high frequency terrain. . 169

LIST OF TABLES

Table 2.1. GeoMorphing rules table. 20

Table 5.1. Results for 1 split PSSM vs. SHM on the low frequency terrain. . . 162

Table 5.2. Results for 2 split PSSM vs. SHM on the low frequency terrain. . . 163

Table 5.3. Results for 3 split PSSM vs. SHM on the low frequency terrain. . . 164

Table 5.4. Results for 4 split PSSM vs. SHM on the low frequency terrain. . . 165

Table 5.5. Results for 1 split PSSM vs. SHM on the high frequency terrain. . 166

Table 5.6. Results for 2 split PSSM vs. SHM on the high frequency terrain. . 167

Table 5.7. Results for 3 split PSSM vs. SHM on the high frequency terrain. . 168

Table 5.8. Results for 4 split PSSM vs. SHM on the high frequency terrain. . 169

LIST OF SYMBOLS

B	Maximum thread-per-block count
D_{world}	Height map cell length in world space
H_i^c	Current terrain height in column c
K	Temporal filter length
L	Current level of detail
M	Highest resolution leaf node level of detail
N	Number of frames in a shadow interpolation interval
P_i^c	Current penumbra height in column c
R	Ray count/height map resolution ratio
R_i^c	Current shadow height in column c
T	Total ray count
z_i	Distance of the ith block from the viewpoint
α	Sunlight angle measured from starting direction
δ	Vertex height error
$\Delta\alpha$	Sunlight angular displacement in an interpolation interval
τ	Screen space vertex height error threshold

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing
HM	Height Map
LOD	Level of Detail
PCF	Percentage Closer Filtering
PSSM	Parallel Split Shadow Mapping
SHM	Shadow Height Map
SM	Shadow Map

1. INTRODUCTION

Realistic terrain rendering is an important application which has various areas of usage. Geographical information systems, city planning, architectural applications and military simulations mostly include terrain-landscape visualization as a core part of their functions. Additionally, it is a common need for video games: Most of the strategy games and flight simulators involve visualization of a large landscape.

Being such a widely used application, terrain visualization has attracted the interest of many different studies. The biggest challenge of terrain visualization is the real-time rendering of the landscape geometry. Terrain geometry is modeled mainly with two approaches: Triangulated Irregular Networks (TINs) and height maps (also called height fields). TINs model the terrain surface as an irregular collection of triangles; they can be optimized to produce the lowest number of triangles. Height maps, on the other hand, model the terrain as a 2D function of the form $h(x, z) = y$, on a regular 2D sampling grid. They tend to produce more triangles compared to TINs and they cannot model landscape features like caves or overhangs. Nevertheless, height maps are the most common way of terrain representation since they are very simple to use and manipulate: A single array of elevation values is often enough to contain the whole surface data. Moreover, their natural 2.5D structure allows implementation of very effective space partitioning techniques, like Quadtrees [1]. Terrain rendering is a difficult task because both of the terrain visualization models involve rendering of many thousands of triangles; such a large geometry load is a large burden, even for the latest graphics hardware on the market. Therefore, most of the research on terrain visualization is focused on so-called Level of Detail (LOD) algorithms, which aim to simplify the terrain geometry in the most visually plausible way while trying to reduce the amount of geometry as possible. Such algorithms are mainly categorized under the names of Continuous LOD (CLOD) and Discrete LOD (DLOD). CLOD algorithms simplify or refine terrain geometry by a frame-to-frame basis and try to obtain the optimal number of triangles according to a visual error metric. DLOD or Semi-DLOD terrain rendering approaches work on pre-simplified meshes produced from the full resolution

geometry. Again they use a visual error metric but instead of refining the geometry each frame, they switch between pre-simplified meshes of the terrain geometry on a sequence. CLOD algorithms suffer from heavy CPU load due to complex simplification operations and they mostly proposed at dates when GPU processing power was very weak compared to CPUs. DLOD and Semi-DLOD algorithms use a lightweight simplification scheme using pre-simplified meshes and therefore they are more optimized for the powerful GPUs of today which can process many triangles much more easily, while relieving the CPU of complex calculations. The algorithms of these classes suffer from vertex popping artifacts resulting from the sudden simplification or refinement of the geometry. Additionally, when used for terrain rendering, neighboring terrain blocks with different LOD levels cause inconsistencies at their shared edges, which is an artifact commonly called as cracking.

Shadowing is another important challenge for computer graphics applications. It is an indispensable visual effect because it provides invaluable clues about the spatial relationship, shapes and positions of the 3D objects and light sources in a scene. Rendering physically based, realistic soft shadows is nearly a trivial feature of ray tracing based applications, where secondary shadow rays shot from the object surface to the light sources are an integral part of the rendering process. Unfortunately, such ray tracing applications often work offline or at interactive rates, at their best; real time ray tracing is still an area which is at the process of development by using the newest GPGPU technologies, like OptiX of NVIDIA [20].

Shadowing for real time, rasterization based graphics algorithms is still an important problem. Rasterization based applications apply local imitations for most of the global illumination effects, for example local material shading using the Phong Model [21]. Shadowing, by its nature, cannot be imitated by such approximations since it is a global effect. In a large scene, shadows can be cast on an arbitrary surface from all possible objects and light sources; by all of them in the most extreme case. To determine the shape and strength of the cast shadows, the whole scene geometry must be considered for calculations. While this is an integral part of the ray tracing algorithms, it is a big and open problem for rasterization based methods. Physical based,

realistic soft shadows still cannot be achieved in real time for rasterization based applications and pixel-correct hard shadows are only at the verge of becoming applicable in real time [6].

For rasterization applications, there exist three main branches for real time, hard and approximated soft shadowing solutions: Projection shadows, shadow volumes and shadow mapping. Projection shadows based on the geometric projection of the object vertices on a planar surface; therefore it is limited with planar receivers. Shadow volumes are able to attain pixel-correct hard shadows for closed meshes. They are based on building the geometry of an object shadow, by extracting its silhouette and extruding it with actual polygons. Silhouette extaction is an expensive operation, moreover since shadow polygons are often large, they are fill rate limited. Shadow mapping is an image based solution for shadowing problem, based on the idea of building a depth map of the visible scene from the viewpoint of the light. It is considered the most general and fastest shadow solution but due its image based nature it suffers from serious self-shadowing and aliasing artifacts. Therefore a vast number of improvement techniques have been developed for repairing shadow maps.

Terrain rendering and shadowing are two difficult problems, as mentioned. Rendering shadows cast by the height map geometry is an especially difficult problem. The best alternative, shadow mapping, runs into serious self shadowing and aliasing problems when used for height map shadowing. The main light source for landscapes, the Sun, is a directional light source which worsens the aliasing problem. Specialized height map shadowing algorithms have been developed to this end, like horizon mapping. Nearly all of these algorithms are only limited with height map self shadowing: They cannot cast shadows from height map features onto arbitrary objects, like shadow mapping. Therefore, terrain shadowing is still an open problem.

In this thesis, we concentrated on both of these problems, especially with a focus on the height map shadowing. Our first contribution is a new algorithm for terrain shadowing caused by the directional sunlight, which uses the concept of the shadow height maps (SHM). Shadow height maps are an enhanced version of regular height

maps, they are the discrete version of the height map shadow volume. In a SHM image, the height of the height map shadow is stored at the corresponding sampling locations. This can be mathematically defined with the function $s(x, z) = S$ where S is the shadow height over the corresponding xz-plane coordinates. We generate this SHM data by using a specialized ray casting algorithm, which uses each SHM column as a sun ray and extracts the 2D shadow volume along its destination, storing the current height in the each pixel of its corresponding column, in an iterative way. We implement this ray casting scheme with CUDA, NVIDIA's GPGPU framework, in parallel, in which each sunray (SHM column) is processed as a separate thread. Our method supports static and movable sunlight environments, which we completely implemented in our program. The SHM method does not suffer from aliasing errors as shadow mapping does, since we generate a discrete approximation of the height map shadow volume and reconstruct it optimally with texture bilinear filtering. Our method suffers from temporal aliasing problem with movable sunlight due to the changing sampling locations for each frame. This causes the shadow volume approximation to change slightly every frame and causes flickering errors. We have reduced that artifact to acceptable levels by defining a shadow interpolation interval and temporal low pass filter. This interpolation interval has allowed us to cache SHM results over a large number of frames such that the CUDA SHM calculation kernels are invoked once at the start of each interpolation interval. This has caused our algorithm to perform better than our competitor algorithm, the Parallel Split Shadow Mapping (PSSM), which is the recommended solution for outdoor directional light shadowing [22]. Additionally, we are able to simulate approximated soft shadows with two approaches. The first one is to define a constant penumbra height. The second one is a geometrically adapting, variable penumbra height, which is closer to the ground truth. Our method allows shadows cast from the height map over arbitrary objects, a feature which most of the specialized height map shadowing algorithms lack. Since we do not have an expensive visibility precalculation step like other height map shadowing methods, our method can be easily expanded to support dynamic height maps.

Our second contribution is the implementation of a 3D terrain rendering system, using a popular terrain LOD algorithm, GeoMipmapping [2], with the Direct3D API.

Our main contribution here is improving the original algorithm by implementing an efficient stitching support, which solves the cracking and T-junctions at the edges of height map blocks with different LOD levels, a common problem to the tile based LOD algorithms. Our solution removes all resolution difference constraints between terrain blocks. Moreover, we implemented a GeoMorphing mechanism, completely in the vertex shader, which removes the popping artifact resulting from sudden LOD level switches, another common problem of DLOD algorithms. This implementation works completely in conjunction with our edge stitching system. In general, we implemented a completely parametric terrain rendering system, which can generate height maps both using the well-known Perlin Noise algorithm [23, 24] or by reading elevation data from image files fed to the program from outside, with a source code over 20000 lines, consisting of a mixture of C++, HLSL and CUDA C languages.

Our final contribution is a through survey of the hard and approximate soft shadowing methods in the computer graphics literature. We listed and explained the most prominent algorithms, their advantages, disadvantages and differences. This survey could constitute a good review of the previous work on this field for interested readers.

The thesis is organized as follows: In Chapter 2 we give brief information on terrain rendering. We describe the LOD concept and mention about prominent algorithms in this area. We will concentrate on the GeoMipmapping algorithm and describe our improvements to the original algorithm as given in [2]. In Chapter 3 we gave a detailed literature survey on shadows in computer graphics. The survey focuses on hard and approximated soft (filtered hard) shadowing techniques. Additionally a survey of specialized height map shadowing methods is given as well. Chapter 4 begins with the explanation of the SHM concept and its place in the literature. Later, we detail our SHM based shadowing method; describing our ideas, implementation, encountered problems and our solutions, step by step. A conceptual comparison of our algorithm with the shadow mapping is given in the end of chapter. Chapter 5 includes our experiments; we compared our shadowing method with the state-of-the-art shadowing solution PSSM (Parallel Split Shadow Mapping). Our comparisons include both the quality and performance aspects. Finally, in Chapter 6, we give our recommendations

for future work and end the thesis.

2. TERRAIN RENDERING

In the core of our algorithm lays the rendering of a terrain. Terrain rendering is a popular area of research since it is used by many different applications. There is a huge literature on the topic of terrain rendering, especially on accelerating the rendering process by the use of so called *Level of Detail (LOD)* algorithms. We implemented a popular LOD scheme in our work and in a unique way with our modifications, but since terrain rendering is not our main focus in thesis, we will give a brief literature survey and shortly describe our work in this area.

2.1. The Basics of Terrain Rendering

Terrain rendering is an important application area in computer graphics; which is used widely by geographical information systems, architectural or military applications, flight simulators and video games. Having such a wide area of use, terrain rendering is an intensively studied problem. The biggest challenge of terrain rendering is to determine how to store and render the terrain surface data in an acceptable period of time. The most common way to represent terrain in 3D is to make use of a *height map*. A height map is defined as a 2D square array of size $2^N + 1$ where $N \geq 1$. Each array elements holds the height of the terrain surface at the 2D position. Such organization of the height data is sometimes called as a *2.5D environment*. A height map can be mathematically represented as a function of the form $h(x, z) = y$ which returns the terrain height above the xz-plane. The height map can be filled in various ways. The data can come from the actual elevation data of the earth, which is called as *Digital Elevation Model (DEM)*. For video games and similar applications, another very popular method is to procedurally generate height data which simulates realistic terrain surfaces in a plausible way. A widely used method for this end is the Perlin Noise algorithm [23]. Another method for storing terrain data is to use so called *Triangulated Irregular Networks (TINs)*. In this method, the terrain geometry is represented by an irregular triangular mesh which does not have the restriction of being defined by a function of the form $h(x, z) = y$. With TINs, caves and overhangs can be modeled,

a feature which height maps do not have. TINs can be optimized to contains less polygons compared to height maps such that they occupy a smaller amount of memory. Nevertheless, height maps are generally a more preferable approach, since their organization is much more intuitive and they naturally support quad-tree or binary-tree based organization of the terrain surface, leading to easier implementation of view frustum culling and collision detection [1]. Our terrain rendering implementation is based on the use of regular grid height maps and in the rest of this text; we always build our algorithms on this consideration.

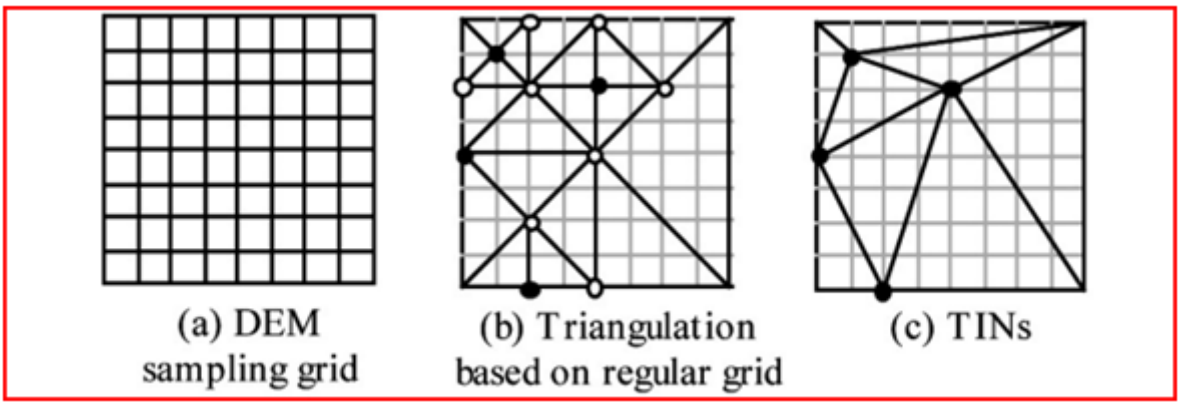


Figure 2.1. Different terrain representation techniques. Image taken from [1].

Independent of the representation model, rendering terrain data is a difficult task since the terrain surface is usually tessellated into thousands of triangles. Considering a 1024^2 regular grid height map based terrain, the terrain mesh will contain $2 * 1024^2 = 2097152$ triangles in total, which is a huge amount of geometry to be rendered every frame. Due to the heavy geometry burden of terrain rendering, most of the research on this topic is dedicated to finding ways to reduce the geometry load. The simplest approaches to reduce geometry load is the use of a space partitioning method and view frustum culling. Space partitioning involves spatial division of the terrain geometry into smaller parts. This provides the opportunity to render the smaller parts of the terrain and only when needed, instead of sending the whole geometry to the graphics pipeline as a whole. There are different space partitioning approaches like Quadtrees, Octrees, Kd Trees and Binary Space Partitioning (BSP). Quadtree is the most simple

and effective way to divide a height map geometry since it is naturally compatible wih 2.5D geometry by its definition [1]. It works as its name implies: We start with a root node, which usually contains the whole terrain geometry. After that we recursively divide the terrain into four children such that each contains $\frac{1}{4}$ of the original geometry. This division process continues until a predefined quadtree depth or polygon-leaf node limit has been reached. Most quadtree implementations for height maps assign actual geometry to leaf nodes, which is our approach in this thesis, also. Since they contain actual geometry and most relevant to our main purpose, quadtree leaf nodes can be simply called only as nodes, as well. In our work, we adjust the quadtree leaf nodes to contain 65^2 vertices.

Space partitioning is not useful without defining a culling mechanism, which will only send terrain parts which are relevant for the final rendered scene (parts which are visible to the camera) and eliminate ones which are not. The common way to achieve this aim is to implement a view frustum culling mechanism. In this technique, the camera viewing volume is represented by a frustum (a pyramid whose tip is chopped off), whose dimensions are adjusted according to the current perspective projection (near and far plane distances, field of view angle, width-height aspect ratio, etc.) This geometrical representation of the camera is translated and rotated in synchronization with the actual camera movement and each frame, its geometry is checked for intersections against the quadtree hierarchy. The culling process is implemented recursively, if a quadtree node's bounding box does not intersect with the viewing frustum, then it is guaranteed that the ones of its children neither will and the sub tree under the current quadtree node can be culled away. If the current node intersects with the frustum, then the bounding boxes of its four children will be checked. The recursion ends in leaf nodes. If a leaf node intersects with the viewing frustum, the geometry it contains is flagged to be drawn and usually added to a rendering queue. The whole process is demonstrated in Figure 2.2.

Space partitioning provides an important reduction in the geometry load just by eliminating invisible terrain parts. But it is usually not sufficient to use it by itself: The geometry which passes the visibility test may still contain a huge geometry load.

An example for such a situation is the case of an overhead camera, which sees a large portion of the terrain. In this particular situation, frustum culling will not provide an important speed up since most of, maybe the whole of the terrain geometry will intersect with the viewing frustum. A very important opportunity of acceleration is sourced from the perspective projection in this case: The terrain parts in the distant will be projected only to a small area on the near plane (screen) and their geometric details will become indistinguishable, all due to the foreshortening effect of the perspective projection. The idea is to reduce the geometry detail for terrain parts which are in distant, as the less detailed geometry will not be distinguished from the full-detail one on the screen after a certain amount of distance has been taken from the camera. This basic idea is the main source of motivation of the various *Level of Detail (LOD)* algorithms and it is a popular area of research for terrain rendering.

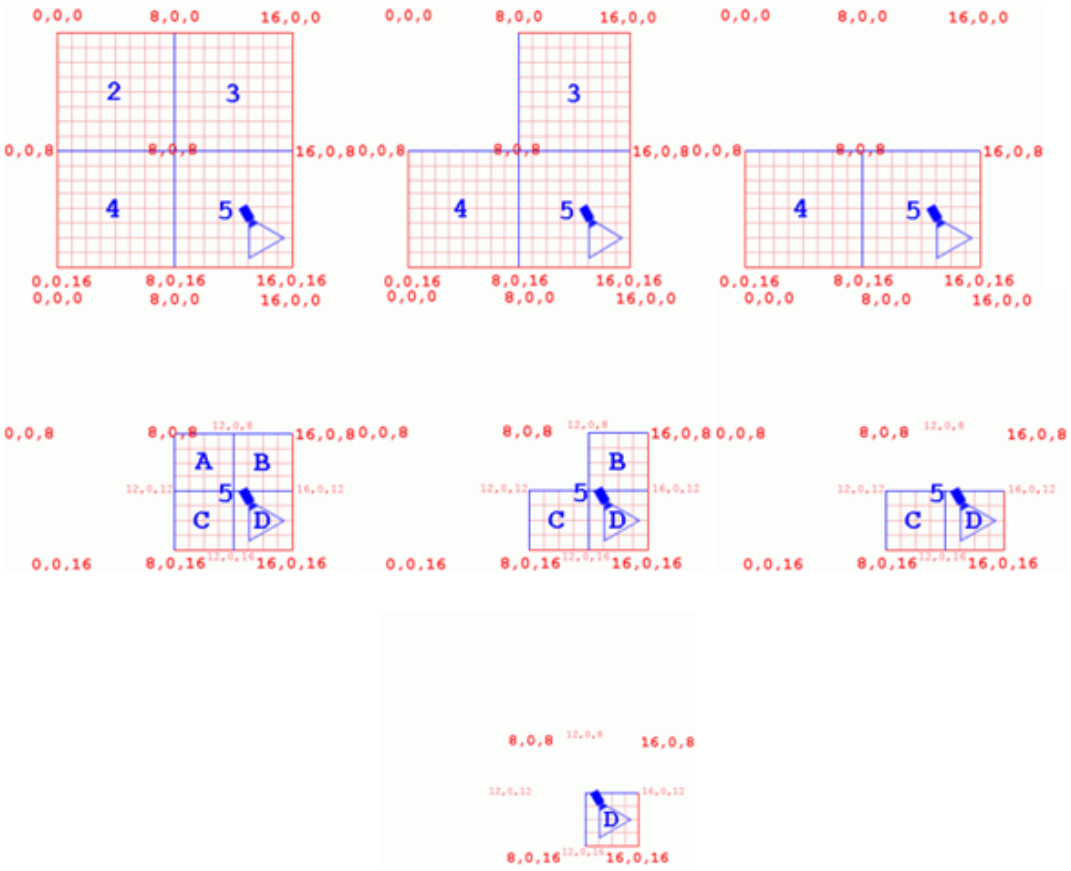


Figure 2.2. Quadtree space partitioning and frustum culling.

2.2. Level of Detail: Definition and Related Work

LOD algorithms try to reduce the complexity of meshes as their distance increases to the viewpoint. The aim is to achieve an acceptable visual quality while keeping the geometry at a reasonable amount in order to accelerate overall rendering speed. [1] defines two types of LOD algorithms: Discrete LOD (DLOD) algorithms and Continuous LOD (CLOD) algorithms. In DLOD algorithms, differently tessellated versions of the same object are generated and the most appropriate version is selected according to an error metric, which is generally a function of the object-viewpoint distance. The major drawback of DLOD approaches is that the switches between LOD levels is visible, as new vertices are introduced to or removed from the surface of the object. This effect is called as the *vertex popping* and it is a visually disturbing artifact. The solution for this effect is to implement some kind of morphing between the LOD levels such that vertices slowly change their positions in order to converge to the next or previous LOD levels as the distance from the viewpoint changes. The implementation of this technique for terrain height maps is called as *GeoMorphing*. The advantage of DLOD approach is that it brings almost no extra CPU load since all LOD levels of the target mesh is preprocessed. No runtime refinement or reduction of the geometry is needed.

CLOD algorithms, on the other hand, reduce or refine the number of polygons at runtime, according to camera position and direction. The mesh is regenerated every frame and this guarantees that in any frame the mesh will contain the optimal number of triangles. Such algorithms provide infinite level of detail in contrast to DLOD approaches but it brings a tremendous processing burden on the CPU since the mesh refinement takes place every frame. Such algorithms include the block based, continuous approach of Lindstorm *et al.* [25], the *ROAM* algorithm of Duchaineau *et al.* [26], which utilizes a binary triangle tree structure, called as a *bin-tree*, which consists of a large triangle in its root and child triangles in higher levels which are generated by bisecting their parents along their base edges and Röttger *et al.*'s method which uses a quadtree hierarchy implemented as a boolean matrix, and a top down simplification method, starting with the whole height map as the root and recursively simplifying child nodes [27]. Most of these CLOD algorithms were proposed quite a while ago, at

a time at which the GPUs were pretty weak processors compared to CPUs. Therefore the aim was to shift the processing burden to the CPU side as possible. Today, GPUs contain much larger amount of raw processing power compared to CPUs [19] and spending precious CPU time on complex mesh refinement techniques like [25–27] is not a good practice anymore. Since modern GPU architectures are able to process thousands of vertices in a single frame without difficulties, lightweight simplification methods which provide non-optimal mesh reduction in a very short CPU time and leave the rest of the processing to powerful GPUs are preferable for today's systems [2]. To this end, some newer methods, which is classified by [1] as *Semi-DLOD* approaches, were proposed. The term Semi-DLOD sources from the fact that these algorithms show the characteristics of both DLOD and CLOD approaches. (DLOD because there are distinct, precalculated LOD levels and CLOD because the simplification-refinement process using the distinct DLOD levels is done in each frame.) One of those algorithms is the *GeoMipmapping* proposed by De Boer [2]. GeoMipmapping uses a $2^N + 1$ sized terrain which is partitioned using a quadtree hierarchy. The leaf nodes of the hierarchy constitute the basic blocks on which the algorithm applies its simplification-refinement process. For each leaf node a set of LOD levels is generated similar to the mipmapping technique for texture filtering. For example, if the highest detail leaf node LOD_0 has the dimension of $2^M + 1$, the next LOD level, LOD_1, will have the dimension of $2^{M-1} + 1$. The simplification process is just the simple power-of-two decimation of the vertices; if the current LOD level is L, the vertices with the 2D indices of $(2^{L-1}(2i + 1), 2^{L-1}(2j + 1))$ will be removed: The number of triangles will be the $\frac{1}{4}$ of a higher level. *Geoclipmapping*, proposed Losasso and Hoppe is another similar approach [28]. It uses a similar simplification scheme with [2]; it generates lower LOD levels by sampling the full detail height map at successive power-of-two locations. As a difference from [2], this method uses nested grids centered about the viewer location, instead of a quadtree hierarchy. Finally, Ulrich's method [29] uses a quadtree hierarchy like [2]. Each child node is exactly one quarter of its parent and each child has a higher level of detail compared to its parent. The LOD levels change according to a predefined screen space error and the meshes of different detail levels are preprocessed by using this error threshold. A child node with higher level of detail will exactly have the half

of the error threshold of its parent.

The algorithms mentioned above are only a few of the numerous LOD methods using DLOD, CLOD or hybrid approaches. They have advantages and disadvantages compared to each other and each of them can work the best with a specific situation. A complete survey of all such algorithms is beyond of the scope of this thesis and we refer the reader to [30] for a very detailed review of LOD methods. Among all of these algorithms, we selected the GeoMipmapping as our choice of LOD method due to its ease of implementation and its full support potential for GPU based GeoMorphing while its CPU side overhead is negligible. We improved the original algorithm as presented in [2] by adding effective edge stitching and GPU based GeoMorphing, in unique ways. In the next section we will give new definitions regarding to the LOD concept and detail our implementation by using those.

2.3. Implementing GeoMipmapping

2.3.1. Error Metric for LOD Transitions

In the previous section we mentioned that all LOD algorithms refine geometry complexity by using some kind of error metric. There can be small details which may vary between different algorithms, but at its core, nearly all LOD methods use a screen space error, $\rho(\delta)$, to determine when to refine or simplify the geometry at hand. δ is the vertex based height change error. Its exact definition is, again, differing among different approaches but for our case with GeoMipmapping it is defined as the geometric height change, which occurs when we remove a vertex from the current geometry. In other words, it is the distance between the current vertex, v, to be removed, and its projection on the line, which will connect two other vertices when v is removed from the geometry. In GeoMipmapping we have unit blocks of the size $2^M + 1$ and LOD changes occur block wise. Therefore, we need to determine a δ_L for the whole block, for which $|\delta_L|$ assumes the largest value among all vertices to be removed at the LOD switch from the level L to level $L+1$ and vice versa. Determination of these values is done with a simple preprocess at the beginning of the application. If $\rho(\delta_L)$ becomes smaller

than a predefined error threshold measured in screen pixels, τ, then it is assumed that the switch from level L to $L+1$, or vice versa, will not cause a disturbing popping effect and it can be done immediately. Knowing δ_L for the current block and assuming y-axis as our up vector, the worst case for $\rho(\delta_L)$ is always a setting in which the camera direction vector is horizontal and it is defined as:

$$\rho(\delta_L) = \frac{h_n}{v_{res}} \frac{z}{n} \delta_L$$

Where h_n is the height of the near frustum plane, z is the distance of the current block's center from the viewpoint, n is the near plane distance, all in world coordinates and v_{res} is the vertical screen resolution, in pixels. De Boer observed by always considering the worst case of a horizontal camera direction vector, we can precalculate the values of z_L, namely the distances of the blocks from the viewpoint, at which they can switch their LOD level. z_L can be computed as the following:

$$\tau = \frac{h_n}{v_{res}} \frac{z_L}{n} \delta_L$$

$$z_L = \tau \frac{v_{res}}{h_n} \frac{n}{\delta_L}$$

In our implementation, we use five LOD levels and for each block, we precalculate and store each z_L with $0 \leq L \leq 3$. At each frame we calculate the 2D distance between the current terrain block center and the camera location on the xz-plane and by using the precalculated transition distances, we determine the LOD level of the block for that frame. Notice that in this scheme, the rotation of the camera will not affect LOD

levels of the terrain blocks, since its x and z coordinates remain constant. Only camera translations are meaningful for LOD switches.

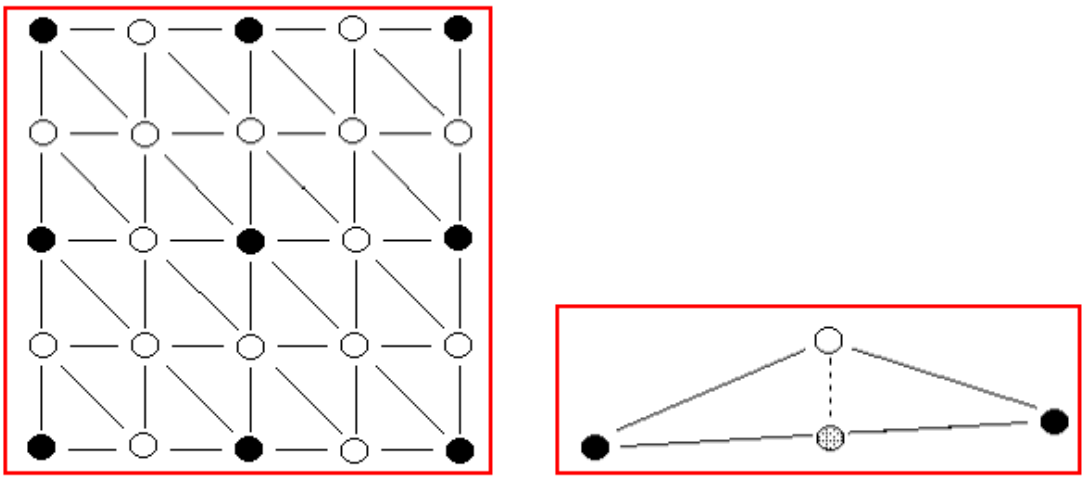


Figure 2.3. On the left, two successive block resolutions for a GeoMipmapping block. The image on the right shows the definition of the height displacement error. Images taken from [2].

2.3.2. Our Index Buffer Based Method for Stitching Edge Cracks

A common problem of all tile based LOD algorithms is the cracks or T-junctions, which appear at the edges of the mesh blocks having different resolutions. The most common approaches to avoid such errors are either limiting the resolution difference between the tiles to one or two, and skipping vertices at the edge neighboring with the coarser resolution as needed, thereby generating a crack free triangulation called as *conforming* [30] or drawing a vertical plane exactly at the location of the crack in order to hide it [29]. Limiting resolution change could cause blocks to assume denser LOD levels where they may be rendered with lower detail, increasing the number of triangles we send to the graphics hardware.

De Boer proposed use of triangle fans at block edges to skip higher resolution vertices but his method is also limited with the resolution difference of one. We im-

plemented an index buffer based solution which removes any resolution difference constraints between blocks of different LOD levels. Our implementation divides a highest level terrain node, LOD_0, with the size of $2^M + 1$, into five distinct regions. Referring to Figure 2.4, where $M = 4$, the red region with the size of $2^M - 1$ is the inner region. The green, blue, yellow and cyan regions are the edge regions. When rendering each terrain block, five "DrawIndexedPrimitive" calls are made to the Direct3D API, for each of these regions, using different index buffers and vertex shader parameters.

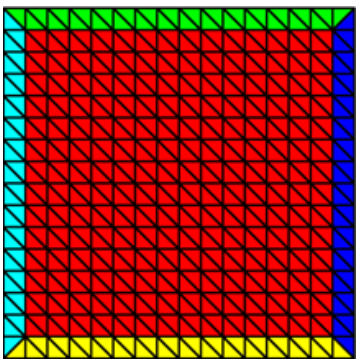


Figure 2.4. Edge regions and the inner region of a terrain block.

We provide connectivity with neighboring blocks via these edge regions. We define the following rule: Assuming the current terrain block has the LOD level of L_i and the neighboring block L_j, we use the index buffer for the according edge, for the level L_i, if $L_i \geq L_j$. We adapt the edge strip to the coarser LOD level otherwise. Assuming the LOD levels are from the interval $[0, N]$, we build for each edge a 2D list,(one list for top edge, one list for the right edge, etc.) $edgeIndices[i][j]$, which contains procedurally generated index buffers, such that we will use the index buffer $edgeIndices[L_i][L_j]$ in the case of $L_i < L_j$. Notice that for $L_i \geq L_j$ we will use $edgeIndices[L_i][L_i]$, since it is the neighboring node which must adapt its edge in this case, in other words $edgeIndices[L_i][L_j]$ is not defined for $L_i \geq L_j$.

The procedure for generating an edge index buffer of the case $L_i < L_j$ is defined as follows: The terrain block of the LOD level L_j has $2^{M-L_j}+1$ vertices and 2^{M-L_j} intervals on its edge. For each interval between the edge vertices of the coarser resolution block,

$\frac{2^{M-L_i}}{2^{M-L_j}} = 2^{L_j-L_i}$ vertex intervals correspond for the denser resolution block. So, we need $2^{L_j-L_i}+1$ triangles in total to cracklessly stitch a single gap between the edges, covered by these intervals, which makes a total of $2^{M-L_i} + 2^{M-L_j} - 2$ triangles to completely stitch the whole gap caused by the missing edge vertices. (Minus two comes from the two triangles at the both ends of the edge strip, which are owned by the neighboring edge regions.) Filling the index buffer is done by stitching the corresponding gap for each coarser resolution edge interval. For each interval on the coarser side, a large triangle is built by connecting coarse side vertices with the middle vertex of the denser side. At the both sides of this large triangle $2^{L_j-L_i-1}$ triangles are built with both of the coarse side vertices, respectively, in a symmetrical way. This process is illustrated by Figure 2.5.

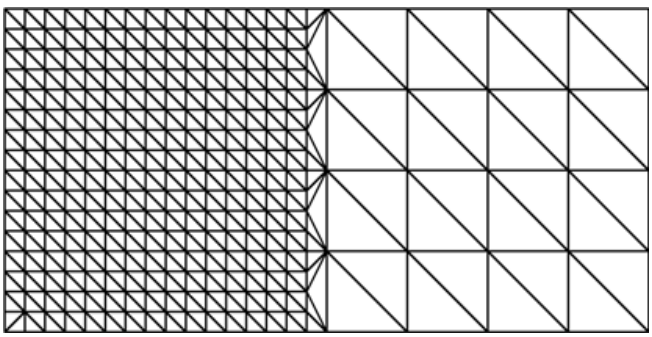


Figure 2.5. A dense block with the dimension 17^2 adapts its edge according to a coarser block of the dimension 5^2. Note the triangle pattern of the stitching edge.

This stitching pattern can cover all neighboring resolutions, without any limitations. Moreover, we only generate the according index buffers only once at the application start and we only hold a single copy of each $edgeIndices[L_i][L_j]$ buffer. We can reuse the same index buffer by adding appropriate offsets for each terrain block. In general, we provided a simple and efficient solution to the stitching problem of the GeoMipmapping algorithm. Notice that this stitching pattern can be generalized for all tile-based LOD algorithms.

2.3.3. GeoMorphing Implementation

We mentioned that the DLOD based algorithms suffer from the vertex popping artifact, which is caused by the introduction or removal of new vertices at LOD transitions. The most common approach to avoid this artifact is GeoMorphing. The idea of the GeoMorphing is similar to the texture bilinear filtering, which linearly interpolates between mipmap images of successive resolutions: The vertices, which will be removed in the next LOD level, $L+1$, are slowly translated on the y-axis such that they gradually assume their new positions in the next or previous LOD level. Notice that such vertices become coplanar with the coarser LOD level block surface just at the moment of transition and therefore removing them will not cause a sudden geometry change. In the similar manner, they will assume their original positions in the denser LOD level block, if the transitions is back to the previous level of $L-1$. In this case, new vertices for the level $L-1$ will be introduced coplanar with the block surface, again avoiding a popping effect. The morphing is controlled with an interpolation coefficient, t. t is calculated with the following formula, for the terrain block i:

$$t_i = \frac{|\overrightarrow{C_{xz}} - \overrightarrow{M_i}| - z_L^i}{z_{L+1}^i - z_L^i}$$

Where $\overrightarrow{C}$ is the camera position, $\overrightarrow{M_i}$ is the center of the current block and z_L^i, z_{L+1}^i are the transition distances for that block from LOD levels $L-1$ to L and L to $L+1$, respectively. For an interpolation value of t_i for the current block, a vertex' current y-coordinate is calculated as the following, if it is going to be removed in the next LOD level:

$$v'_y = t_i \delta_v + v_y \tag{2.1}$$

Where δ_v is the height change for the current vertex. The process illustrated in Figure 2.6.

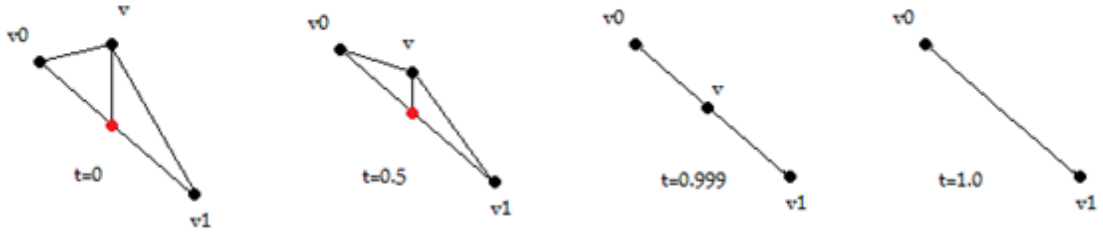


Figure 2.6. The vertex v slowly travels to its projection as the GeoMorphing progresses. Note that at the next level, there won't be a vertex at the location indicated by the red dot.

We implemented this logic in the vertex shader. We first need to determine vertices which are going to be removed at each LOD transition. For a 2D row major grid of the height map vertices, the vertices which are going to be removed at transition from level L to $L+1$ can be addressed as $v_{(2^{L-1}(2i+1),2^{L-1}(2j+1))}$ where i and j are integers with $i \geq 0$ and $j \geq 0$. Since the vertex shader runs on each vertex independently, we need to store per vertex information of vertex height change, δ_v, in the current vertex format, along with the world coordinates. Moreover, a vertex must only morph if it is going to removed at the current LOD level transition. Therefore, we need to store the "LOD level", L_{vertex} of each vertex into the vertex format, as well. As we mentioned, we do the actual rendering block wise. For each block, we pass the block's current LOD level L_{block} and the interpolation coefficient, t, as constants for the vertex shader. If the currently processed vertex has the LOD level, such that $L_{vertex} = L_{block}$, then we morph its y-coordinate as in Equation 2.1.

A problematic case occurs for the morphing of the edge regions. If it is $L_{block} < L_{neighbor}$ for the current edge region, then the edge is adapted to the neighboring block, as mentioned in the previous section. In such a situation, the vertices shared by the

neighboring block will be morphed in their according edge strip, where they are not, in the current block. This would violate our stitching mechanism from the previous section and "morphing" cracks would appear gradually as the camera moves. To make the GeoMorphing compatible with our stitching mechanism, we additionally pass the neighboring block's LOD level, $L_{neighbor}$ and its morphing coefficient, $t_{neighbor}$, as vertex shader constants before calling "DrawIndexedPrimitive" function for the four edge regions. We add another per vertex information to the vertex format, a binary flag, v_{edge} which indicates whether the current vertex lies at the edge of the block. If this flag is true for the current vertex, we morph the vertex if it is $L_{block} < L_{neighbor}$ and $L_{vertex} = L_{neighbor}$, by using the morphing coefficient of the neighbor, $t_{neighbor}$. For the case of $L_{block} > L_{neighbor}$, we morph the vertex as usual if its $L_{vertex} = L_{block}$, by using the current block's coefficient. The case of $L_{block} = L_{neighbor}$ needs special care; because in this case both blocks use same resolution edge strips and they render the same vertices at their shared edges, twice. In order to morph both neighbors in a crack free manner, we use the morphing coefficient $t_{max} = max(t, t_{neighbor})$ such that both neighbors agree on the same amount of morphing. This whole scheme can be summarized in the following rule table:

Table 2.1. GeoMorphing rules table.

Rule	**Morphing**
$v_{edge} = false AND L_{vertex} = L_{block}$	$v'_y = t\delta_v + v_y$
$v_{edge} = true AND L_{block} > L_{neighbor} AND L_{vertex} = L_{block}$	$v'_y = t\delta_v + v_y$
$v_{edge} = true AND L_{block} < L_{neighbor} AND L_{vertex} = L_{neighbor}$	$v'_y = t_{neighbor}\delta_v + v_y$
$v_{edge} = true AND L_{block} = L_{neighbor} AND L_{vertex} = L_{block}$	$v'_y = max(t, t_{neighbor})\delta_v + v_y$

A rare case can occur if for a terrain block, it is $z^i_L > z^i_{L+1}, z^i_L > z^i_{L+2}, \ldots, z^i_L > z^i_{L+N-1}$. In such cases, the LOD level switches from L to $L+N$, skipping the levels between. This can cause vertex popping, since our scheme morphs the vertices according to a transition to level $L+1$. This can be easily fixed by morphing the vertices for all LOD levels between L and $L+N$ according to a transition from L to $L+N$, by

calculating appropriate δ_v values for each vertex. Since we encountered this condition rarely in our experiments, we deem it as negligible and leave our GeoMorphing scheme as described above.

2.4. Terrain Texturing

The most common approach for texturing terrain surface is using a two layered system: A layer for a large ground texture which is stretched over the whole surface and a layer for a detail texture, which is repeated over the terrain surface. In the pixel shader, these two layers are blended together to create the final appearance. A more detailed approach is called as the *texture splatting*, proposed by Bloom [31]. The method consists of combining different textures. It works by using a blend map; this is a texture containing different channels and each channel is used as an alpha map for the corresponding texture layer. According to the alpha coefficient of each layer, a mixture of different textures is obtained as the result. The different textures can be of different surface material types, like stone, grass, dirt, snow, etc. Usually, the alpha component for a specific surface type is modulated according to the elevation and slope of the corresponding height map vertex [32]. In our implementation, we generate four different 8-bit grayscale alpha maps for each channel of a 32-bit RGBA texture, which we are using as our blend map for four different surface layers. We generate the alpha maps using the Perlin Noise algorithm in order to generate a completely random surface look.

Since overstretched surface textures highly lose their details due to undersampling, we use a repeating pattern for each texture layer. The random mixing of the texture splatting approach hides the repeating patterns successfully most of the time for a moderate distance from the terrain surface. But, despite of this, this pattern becomes visible for overhead looks. The solution would be the decrease of the number of repeat patterns. This time, the surface becomes too blurry if the camera approaches closely to the surface. We solved this artifact by interpolating between a dense and coarse repeating pattern of texture layers in a parametrical interval of distances. With this method, we get a densely repeating ground texture pattern close to the camera

and a coarse pattern for distances. The transition between these patterns is smooth due to the interpolation.

2.5. Terrain Shading

In order to shade the terrain, we use the Phong illumination model [21]. We define a parametrical ambient term, and calculate a diffuse term, which is the result of the dot product, $\overrightarrow{n}.\overrightarrow{L}$, where $\overrightarrow{n}$ is the vertex normal and $\overrightarrow{L}$ is the sunlight direction. We do not use a specular component since the terrain surface usually does not consist of reflective materials [33].

We generate the vertex normals by summing and averaging all plane normals which share the current vertex. Averaging vertex normals like this is a common practice in rasterization based graphics applications [33]. But instead of embedding vertex normals into the vertex format structure directly, we use a *normal map*, which stores in its each pixel the normal of a corresponding height map vertex. The normal data is converted to a RGB color representation to be able to be stored in an unsigned integer image. Our purpose in using a normal map is due to fact that the underlying LOD algorithm will cause vertices to be removed and introduced as the camera moves and this will cause a sudden change of the surface shading. This is another form of the geometry popping artifact we mentioned in Section 2.3. By storing normals as they are in the world coordinate space, we avoid such a shading popping artifact since the 2D texture sampling locations on the xz-plane do not change between LOD levels. The normal map is sampled and the obtained RGB data is converted to a surface normal again, in the pixel shader stage. We use 32-bit full precision RGBA textures to store normal map data, but it is also possible to use lossy compression formats, like DXT-1, with a trade-off in the shading quality [34].

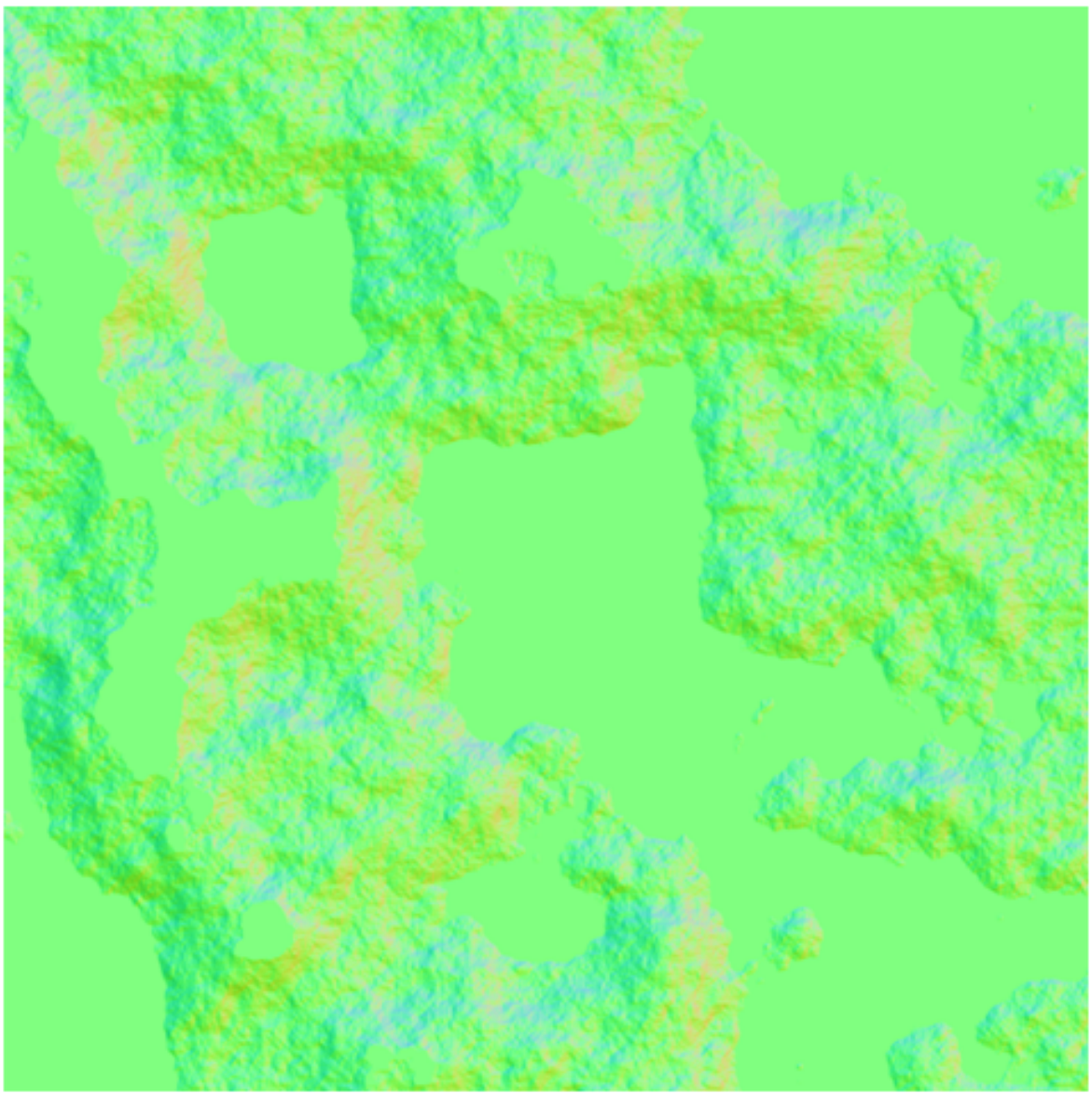


Figure 2.7. The world space normal map for a 1024^2 height map.

3. SHADOWS IN COMPUTER GRAPHICS

Shadows are one of the most important elements for creating realistic looking images in computer graphics and image processing. They give very important visual cues about the spatial and geometric relationship between objects and highly contribute to the realism of a synthetic 3D scene. A shadow can help to understand the position, size, orientation and geometry of its caster and its receiver; additionally it can give clues about the distance of the light source. Therefore, generation of shadows is a popular topic in the area of computer graphics. In [4], a shadow is defined as a region of relative darkness within an illuminated region, which occurs when an object totally or partially occludes the incoming light. Shadows are the result of lack of light in an area and therefore they can be realistically calculated by using the physics of light, but they are mostly treated as separate elements of rendering because calculating them is a difficult task due to their global nature, which means they can be sourced from the whole scene and world geometry [35]. A point is in shadow, if it cannot be seen when looked from the direction of the light source. The object, which blocks the point from the light rays and makes it invisible from the point of view of the light source, is called as the *shadow caster*, *occluder* or *blocker*. The object, which has the point which is blocked from the light by the shadow caster is called as the *shadow receiver*.

In this chapter, we are going to examine the shadow generation problem in computer graphics in detail. We will give the basic physics and terminology of the shadow phenomena in the first section and we will present a wide literature survey on this topic in the computer graphics area, in the coming sections. We will list and show the mostly used, state of the art, mainstream shadow generation algorithms and explain their workings, advantages and disadvantages in detail. After that, we will concentrate on shadow algorithms, which are specialized for rendering of height maps and terrains.

3.1. Terminology and Mathematics for Shadows and Definition of the Problem

The simplest case of shadow generation occurs when an object casts a shadow on a planar object or simply on a plane [3]. When the light source is a point or directional light source, the generated shadows are said to be *hard shadows*, which only contain fully shadowed regions, which are affected only from the ambient element of the lighting model. If the light source is an area light source, then the obtained shadows are said to be *soft shadows*. These types of shadows consist of two different areas; a fully shadowed region, which is called as the *umbra* and a partially shadowed region, which is called as the *penumbra*. Figure 3.1 shows this terminology in more detail. Figure 3.2 shows the differences between point, directional and area light sources and how they create different types of shadows.

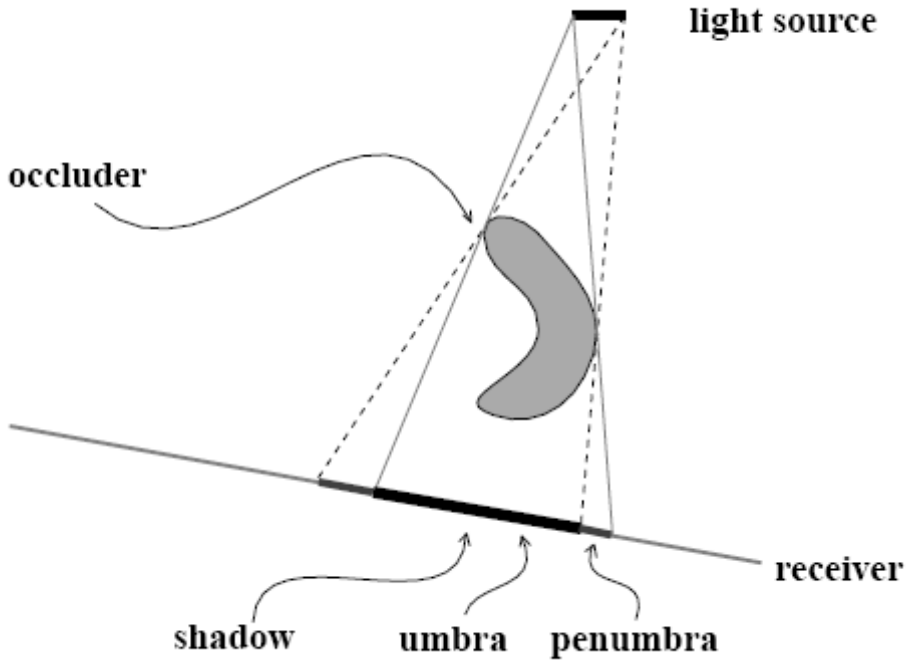


Figure 3.1. Depiction of the shadow terminology with umbra, penumbra, light source, occluder and receiver. Image taken from [3].

Since there are no perfect point and directional light sources nature, soft shadows are preferable over hard shadows since they provide a more realistic effect. Hard edged shadows can be sometimes mistaken for actual geometric features of other objects [3]. A work conducted by Wanger [36], reached interesting results about the relationship between the perception of an object's size, shape and position and its shadow's sharpness and shape. Wanger made three psychophysical experiments on real people to measure the effect of shadow sharpness and shape on the perception of spatial relationships in the computer generated imagery. His first experiment tested the effect of shadow sharpness on the perception of the caster object's size and position. His second experiment measured the effect of the shadow shape on the perception of the caster object's size and position. His third and final experiment measured the effect of shadow sharpness on the perception of the caster object's shape. In each of these experiments, the accuracy with which the experiment subjects were able to perform spatial estimation tasks was measured. He summarized the results of the experiments as follows:

- The existence of shadows greatly improves the accuracy of size and position estimations, but the sharpness of the shadows does not have a statistically significant effect on the measured accuracy.
- The size, depth and height of the object tested are all important factors for accuracy of the tests but the shape of the shadow does not make a significant contribution to the measured accuracy.
- The sharpness of the shadow has a significant effect on the measured accuracy about the shape of the shadow caster object. Wanger found out that soft shadows have a significant negative effect on the estimation of object size.

The most important result of this work is to conclude that in many cases cheaper hard shadow generation techniques are adequate and in fact may be more beneficial compared to more expensive soft shadow generation techniques [36]. We can interpret these results that hard or just simply edge softened; physically incorrect soft shadow techniques are good enough for most of computer graphics applications. It is better

to have physically inaccurate shadows than having none at all. Shadow computations may be most of the time just crude approximations of the real physical phenomena, since our eyes are fairly forgiving about the shape and sharpness of the shadows [3].

How shadows can be fitted in the general rendering process is shown in great detail by Eisemann *et al.* in [6]. They use the well-known rendering equation introduced in [37] so as to place shadows in the rendering process in a mathematically correct way. The rendering equation as interpreted by Eisemann *et al.* is shown in the following way:

$$L_o(p,\omega) = L_e(p,\omega) + \int_{\Omega} f_r(p,\omega,\widehat{\omega}) L_i(p,\widehat{\omega})(\widehat{\omega}.n_p)\,\mathrm{d}\widehat{\omega}$$

Where L_o is the function of outgoing radiance from point p in the direction of ω. L_e is defined as the emitted light at point p. In other words, it is the amount of light energy produced at point p. L_i is the incoming radiance to the point from all directions of the whole hemisphere Ω around the normal, n_p. $\omega.n_p$ is the dot product of the direction of the incoming light and point normal. It represents the amount of attenuation of the incoming light energy due to the orientation of the receiving surface. f_r is the so-called BRDF; the Bidirectional Reflectance Distribution Function. It simply describes the ratio of the outgoing light energy to the incoming light energy. This function can be very complex or can be a simple constant for a perfectly diffuse, Lambertian surface [6].

The rendering equation is derived from the fundamental physics law of conservation of energy. It is impossible analytically to solve the equation except for a few very simple cases and therefore, the basic aim of all computer graphics applications is to approximate the equation in a way or another. Eisemann *et al.* make a series of approximations and simplifications in order to simplify the equation. First of all they

declare that they are only interested in direct illumination; this removes the dependency of the equation on itself by removing the indirect illumination which is defined as the incoming light from non-source objects in the scene through reflections. With this approximation, the integral for the incoming radiance evaluates to zero for all non-light caster points in the scene, limiting it with light sources [6]. Additionally, they leave out the emitted light energy L_e only to add it later to the evaluated integral. Another approximation is to consider only a single light source and integrating over the points which together constitute the light source. If p is the current point for which the integral is being evaluated and q is a single point on the light source and if p and q are mutually visible (no occluding object on the direct line between them) then the outgoing illumination from q to p is equal to the incoming illumination to p from q and vice versa. Another important simplification is to consider the surface on which the point p resides as a perfectly diffuse Lambertian surface. This leads to the simplification of the BRDF which could be a very complex function, to a simple constant. Additionally this makes the outgoing radiance from the point p independent of the looking direction, ω. With all these simplifications, the outgoing radiance formula for the point p becomes:

$$L_o(p) = \frac{\rho(p)}{\pi} \int_L L_e(q, q \to p) G(p, q) V(p, q) \, \mathrm{d}q \tag{3.1}$$

Where $\frac{\rho(p)}{\pi}$ is the Lambertian reflectance constant of the surface where the point p resides. It should be noted that the BRDF has been simplified to this constant and then it is taken out of the integral. $q \to p$ is the notation with which Eisemann *et al.* represents the normalized vector $\frac{p-q}{|p-q|}$ in [6]. L is the surface of the light source. We consider a single light source for mathematical representation of the soft shadows and the hemisphere over the point p is simplified to the surface of the light source as the result. $G(p, q)$ is the geometrical function for the light attenuation between the points p and q. It is defined as $G(p, q) = (p \to q.n_p).(q \to p.n_p)$. $V(p, q)$ is the binary visibility function for the points p and q. If these two points are mutually visible to

each other without any obstruction, then the function evaluates to 1. Otherwise the function evaluates to 0 and by multiplication with other terms in the integral it zeroes out the contribution of lighting from the point q.

The simplified rendering integral in Equation 3.1 handles both the shading and shadowing of the point p together. As stated in [35], it is often desirable to evaluate shadowing as a separate rendering process which is handled independently from the shading part. It should be noted in a ray tracing application physically correct shadows are trivial because shooting secondary rays from points to light sources are an integral part of the rendering process, which includes processing all the world geometry. But in real time raster graphics, shading is done usually at a local level by local illumination techniques such as the popular *Phong Shading* method [21], which only consider the local geometry of the currently processed object most of the time in the form of surface and vertex normals, the light source as approximated simpler models (point, directional and spot lights, as stated in [33]) and the current surface as a perfectly diffuse Lambertian material, which simplifies the BRDF into a single constant. These models do not convey and process any global information about the scene. Therefore, we cannot integrate shadowing naturally in raster graphics, which is a global effect and needs the whole world geometry to be included in its calculations. For example, we cannot know which object or objects cast shadow on the current pixel we are processing just by looking its local geometry information. We need additional methods which check the world geometry and calculate occlusion information for the desired point or pixel. This limitation of the raster graphics requires us to further simplify Equation 3.1, such that the shading and shadowing are separable parts of the equation. To this end, the overall change of the geometrical function, $G(p, q)$, on the light source for each q is considered as negligible for the shadow calculation; this is physically only true if the distance of the light source is relatively large compared to the object's solid angle, which is defined as "the angle an object subtends in three-dimensional space for a given point" in [6]. We will use this fact aggressively in our shadowing techniques for terrain rendering, since our light source will be the Sun, which gracefully meets this simplification requirement. With this consideration, the integral of Equation 3.1 can be separated into two parts, one which handles the shading and one which handles the

shadowing. The resulting equation, as in [6], is the following:

$$L_o(p) = \frac{\rho(p)}{\pi} \int_L G(p,q)\,\mathrm{d}q \int_L L_e(q, q \to p) V(p,q)\,\mathrm{d}q$$

In this new equation, the first integral with the geometrical attenuation function, $G(p,q)$, is the shading integral and the second one is the visibility sampling integral for the light source L, which calculates the contribution of shadow to the final outgoing radiance. If we further assume that the light source has a homogeneous distribution of the outgoing radiation over its surface, L_e transforms into a simple constant light direction, L_c and then can be taken out of the shadowing integral. The remaining integral is simply a visibility integral and modulates the result of the local illumination shading according to the amount of shadow the current point receives. It is expressed as the following:

$$L_c \int_L V(p,q)\,\mathrm{d}q \tag{3.2}$$

Although this visibility sampling integral correctly produces soft shadows, it is not physically correct and its results can vary greatly from the equation of Equation 3.1 [6]. The reason is that the integral only calculates the amount of visibility of the light source from the point p and does not consider which parts of the source are blocked. Due to the geometric term $G(p,q)$, different parts of the light source will have different amounts of influence on the calculated final shadow. But as we have said before, it provides good enough results if the distance to light source is relatively large to source's solid angle or if the light source is considered infinitely distant and large and shooting directional and parallel light beams; which is the case of the Sun [22].

Anyhow, the integral of Equation 3.2 is used when it is aimed in real time graphics applications to produce physically accurate looking soft shadows [6].

The calculation of the overly simplified shadow integral of Equation 3.2 for realistic looking soft shadows is still problematic. The first problem which needs to be solved is to handle different occluders separately in an environment. Even if we obtain physically accurate but discrete occlusion contributions for each occluder, which is a difficult issue by its own, it is generally not possible to obtain a successful estimate to the visibility integral of Equation 3.2; those discrete values only provide lower and upper bounds for the visibility calculation [6]. Eisemann *et al.* defined the following relationship between the true visibility integral for a given point p and separate occlusion values, where B_i represents the discrete occlusion contribution for each object O_i, which blocks the light beams reaching to the point p:

$$max_i(B_i) \leq \int_L V(p,q)\,\mathrm{d}q \leq \sum_i B_i$$

The lower bound; taking the maximum of the blocking contributions is physically correct when all blockers fall in the same part in the hemisphere over the point p, in other words, if they totally overlap and block the same part of the light source. The upper bound; taking the sum of the all blocking contributions is physically correct if all occluders block different parts of the light source, meaning they don't overlap each other in the hemisphere over the point p. When different occluders overlap each other in varying proportions, none of these two extremities are correct; taking the multiplication or the average of the contributions are alternative solutions but they are not valid in all cases either. In short, this *occluder fusion* problem, as defined by Eisemann *et al.* in [6], is a challenging issue when dealing with soft shadows and there is not an ultimate solution for it; one should select an approximation method which fits the best to the situation at hand.

Evaluation of Equation 3.2 for soft shadows needs much attention; but a robust solution would be simply to sample the light source at discrete samples; this is equivalent to treat the area light source L as a collection of many point light sources. By sampling the light source at different points, l_i, which are uniformly distributed over the source, we actually provide a discrete approximation to the integral of Equation 3.2:

$$L_c \int_L V(p,q)\,\mathrm{d}q \approx \frac{1}{n+1}\sum_{i=0}^{n} V(p,l_i) \tag{3.3}$$

This provides a direct link between hard and soft shadows; a single point light source would create exact hard shadows; but sampling an area or volumetric light source with uniformly distributed point light sources and averaging their light energy contributions leads to soft shadows. Note that the visibility function V evaluates to zero if the currently sampled point light source is blocked to send light beams to the point p and one otherwise. Unfortunately, obtaining convincing soft shadows, where even exact hard shadows for a single light source are still problematic, requires many samples. Eisemann *et al.* state that 256 to 1024 samples are needed for each pixel for medium sized light sources; this number would be even higher for larger light sources [6]. Additionally, for each pixel which is being processed we need to check the whole scene geometry, which makes the computation even more complicated. As an important result, Eisemann *et al.* report the following in their up-to-date (2009) survey of real time shadowing methods:

> *"Accurate soft shadows cannot be obtained in sufficient quality in real time and we are only on the verge of achieving pixel-accurate hard shadows for geometrically complex scenes at acceptable speed."*

In this section, we have discussed how the shadows come into being, their phys-

ical and mathematical background and how they are mapped into the practical area of computer graphics. Although being a hot topic in computer graphics, detailed and up-to-date surveys investigating and summarizing all practical shadow generation techniques are fairly rare. The first survey on this topic is the famous one of Woo *et al.* [4], published in 1990, which is significantly outdated and most of the methods it discusses are obsolete. Another review can be found in Watt [38]. A more recent survey is that of Möller *et al.*, in *Real-Time Rendering* [3]. The most detailed, up-to-date and practical review of all shadow methods and their mathematical background we were able to find is presented by Eisemann *et al.* in [6]. Following sections treat shadowing techniques. The first one we will examine is the projection shadows technique. Although it is very limited in its area of usage, it is an important contribution to shadowing problem and therefore it is worth investigating in more detail. The next technique will be the *shadow volumes*, which is introduced by Crow in [39]. After that, we will examine the famous *shadow mapping* technique, which is firstly introduced by Williams in [40]. Shadow volumes and shadow mapping are the mostly used, state-of-the-art methods for handling the shadowing problem in computer graphics. There are countless improvements to both of these standard algorithms in the literature, in terms of performance and quality. Some other improvements extend these algorithms to cover the soft shadowing problem, because both basic algorithms are designed to create hard shadows in their first published versions. While we are covering the most prominent ones among those improvement methods and algorithms in detail in the coming sections, we will only discuss some of them in brief, because of the huge amount of literature on this subject. Soft shadowing is a very challenging topic and therefore it deserves a broad investigation but we will give a brief summary of the problem and explanation of a famous method, since physically correct soft shadowing is not exactly our area in this thesis. Finally, the shadowing methods which are specialized to cover height maps and terrains will be investigated. We are going to explain why the standard methods of shadow volumes and shadow maps can be inadequate when they are used alone for height map, terrain shadowing problem. We are going to show the short comings of both methods when it comes to shadow large meshes like height maps and examine the alternative techniques in the literature in detail and close the chapter.

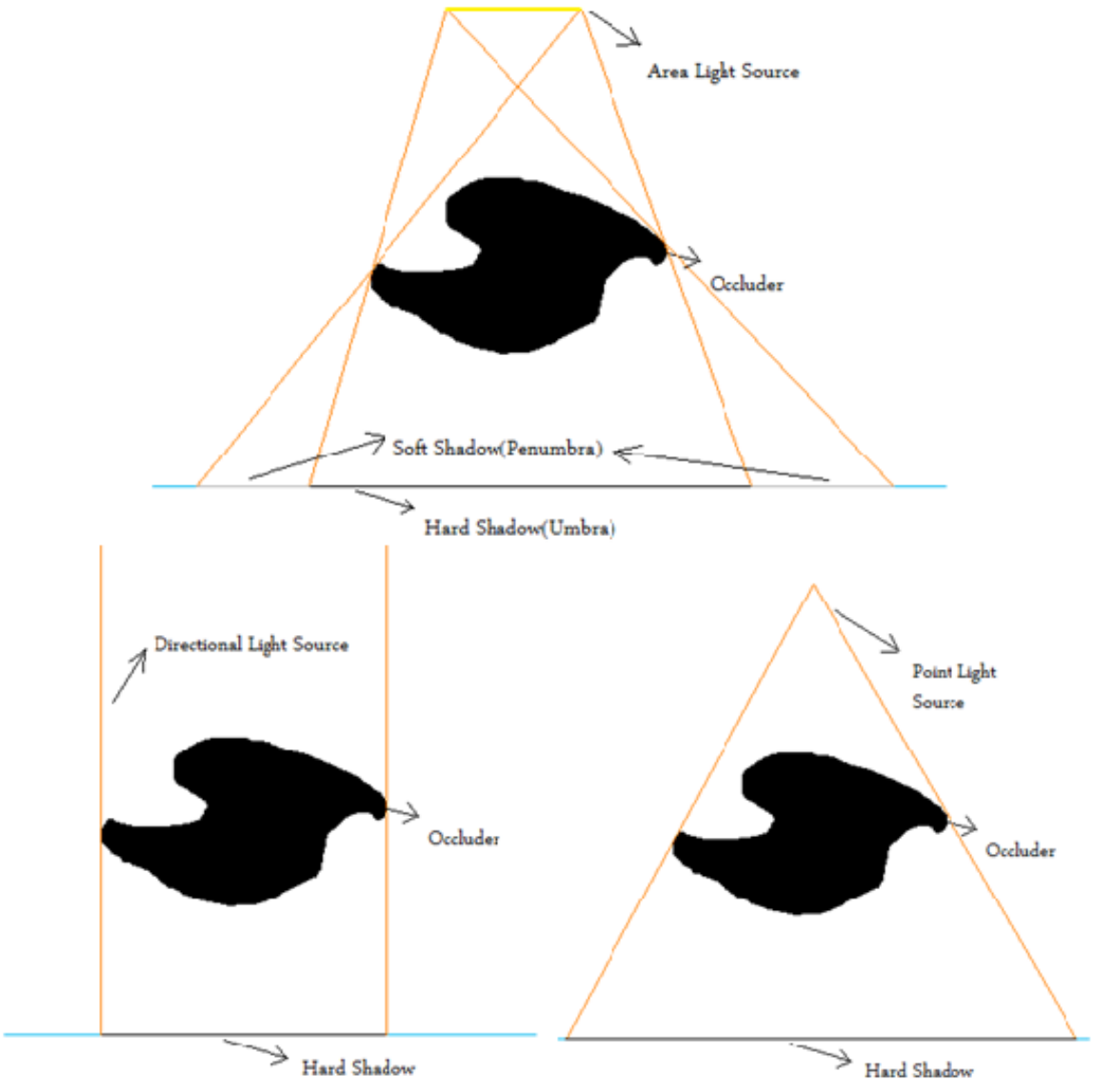


Figure 3.2. How different light source types create different types of shadows.

3.2. Projection Shadows

In projection shadows technique, a three dimensional object is rendered the second time in such a way, that its vertices are projected onto a plane and create its shadow as seen from the light source. These methods actually do not create the real shadow of the object; they create a planar projection of it and redraw that second projected object onto the receiver plane with a shading model according to the lighting properties of shadows in the current scene [41, 42]. As they simulate shadows by redrawing the shadow caster object a second time, they are called as "fake" shadowing methods [4, 41]. These methods are unfortunately limited with flat-planar shadow receivers. Additionally, there are some problems related to rendering the second, projected "fake" object; like depth biasing and clipping the projection shadow when it falls outside of the receiver plane.

3.2.1. Hard Shadows

When the light source is a point or directional one and the shadow receiver is a flat, planar surface, then the basic hard shadow projection method of [41, 42] can be used. Figure 3.3 shows two cases of planar shadow projection; the one in the left is the special case when the receiving plane is at $y = 0$ and the right one is the general case of receiving plane with the equation $\pi : \overrightarrow{n}.x + d = 0$. A projection matrix for the special case of $y = 0$ can be derived by using similar triangles. It is for the x coordinate:

$$\frac{p_x - l_x}{v_x - l_x} = \frac{l_y}{l_y - v_y}$$

When arranged for p_x, the following equation is obtained (with the equivalent one for p_z):

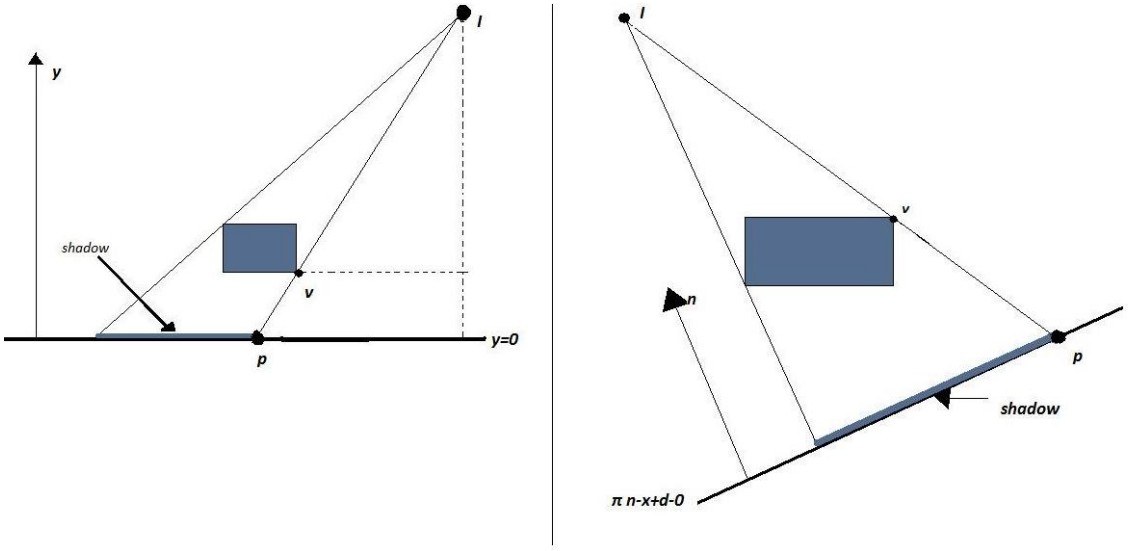


Figure 3.3. An axis aligned planar receiver (left) and an arbitrarily oriented one (right). Image based on [3].

$$p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y} \qquad\qquad p_z = \frac{l_y v_z - l_z v_y}{l_y - v_y}$$

Since the y-coordinate is zero, it is left out in the projection calculation. We need to build a projection matrix which will give $Mv = p$. The matrix M can be expressed as:

$$M = \begin{bmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{bmatrix}$$

A generalized projection matrix for the general receiver plane equation $\pi : \overrightarrow{n}.x + d = 0$ can be produced, too. We will only give the matrix as it is and refer the reader for its derivation to [3, 41, 42]. The matrix will be this time:

$$M = \begin{bmatrix} n.l + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & n.l + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & n.l + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & n.l \end{bmatrix}$$

In practice what needs to be done is to apply the matrix above to an object and to render it on the receiving surface with all its polygons black or only shaded with the ambient component of the lighting model [3]. One thing which needs attention is to avoid z-fight with the receiving surface; if the projected "fake" shadow's vertices have exactly the same z-value as the receiving plane, the shadow and the receiver plane can be erroneously drawn mixed together, due to the low floating point precision of the depth buffer. This is a famous artifact in the area of 3D rendering; called as the *z-fight* [33]. The solution is to give the rendered shadow vertices a little bias in the negative direction of the light beams. Getting this bias correctly needs caution as too much bias will separate the "fake" shadow from the receiving plane and it would appear as "flying" over the plane. Too low bias will lead to z-fight, on the other hand. Another good solution is to render the receiving plane first and then to render the shadow with z-buffer depth test disabled. Then for the other geometry, the buffer will be enabled again. Another problem with this method is to avoid shadows being drawn outside of their receiving planes; since by deriving the projection matrix we assumed the plane as an infinite surface. This is solved by the stencil buffer test: Firstly the ground plane is drawn, both to the frame buffer and to the stencil buffer. Secondly, shadows are drawn, with z-buffer test disabled. We set the graphics API in such a way that only the shadow pixels which correspond to filled stencil buffer pixels are drawn. This is a fast hardware clipping test and is considered a cheap operation. Another

disadvantage of the operation is we need to render the "fake" projection every frame. This doubles the geometry we are processing automatically. Solutions can be using low detail models for rendering the projection shadow or rendering the projection into a texture as a preprocessing step and mapping it onto the receiver plane in real time. Finally, *anti-shadows* and *false shadows* are artifacts which need to be addressed using projection shadows. An anti-shadow appears when the light source is below the top-most point on the object and closer to the receiving plane. Since the object vertices are projected through the light source, a projection of the object still appears on the receiving plane, where in reality no shadow must come into being. False shadows occur when the shadow caster is behind the receiving plane. In other words, the caster object is farther away to the light source than the receiving plane. Similar to anti-shadows, the object is still projected onto the plane and generates shadow where actually it must not. This can be solved by placing a clipping plane at the shadow receiver and cull away all geometry which falls behind this clipping plane. Note that while we used point light sources to derive the projection matrices, they can be generated by using directional light sources, too.

3.2.2. Soft Shadows

The projection methods of the previous subsection support only hard shadows generated by point and directional light sources. They can be extended to support soft shadows. To generate soft shadows with the projection shadows technique, the sampling approximation of Equation 3.3 is used. An area light source is sampled uniformly at different locations and every sample is treated as a separate point light source. For each sampled point light source, a hard shadow image is generated and it is written into an accumulation buffer. Finally, all hard shadows from different point light samples are averaged, which effectively generates a soft shadow image with penumbrae. In [43] and [44], the hard shadow projection technique is extended in that direction to support soft shadows. In [43] and [44], Heckbert and Herf generate a separate pyramid for each point light sample on the area light source. Those pyramids are generated by mapping each point sample a as the tip of the pyramid and the receiver parallelogram

as the base of the pyramid (Figure 3.4). After forming the pyramid, they create a transformation matrix M, which transforms the pyramid into a parallelepiped. The parallelepiped lies in the unit space; this means in (x, y) from $(0, 0)$ to $(1, 1)$ and the z-coordinates are transformed such that $z = 1$ is at the shadow receiver and $z = \infty$ is at the light source. To correctly cull away all objects which fall outside the range between the light source and receiver and in order to avoid anti-shadows and false shadows, one should multiply the matrix M with an orthographic projection matrix P_0 whose clipping planes lie $n = 1$ at near and $f = \infty$ at far. After creating the final projection matrix $P = P_0M$, for each sampled point light source, the resulting matrix can be applied when rendering the scene for that specific light source as the projection matrix.

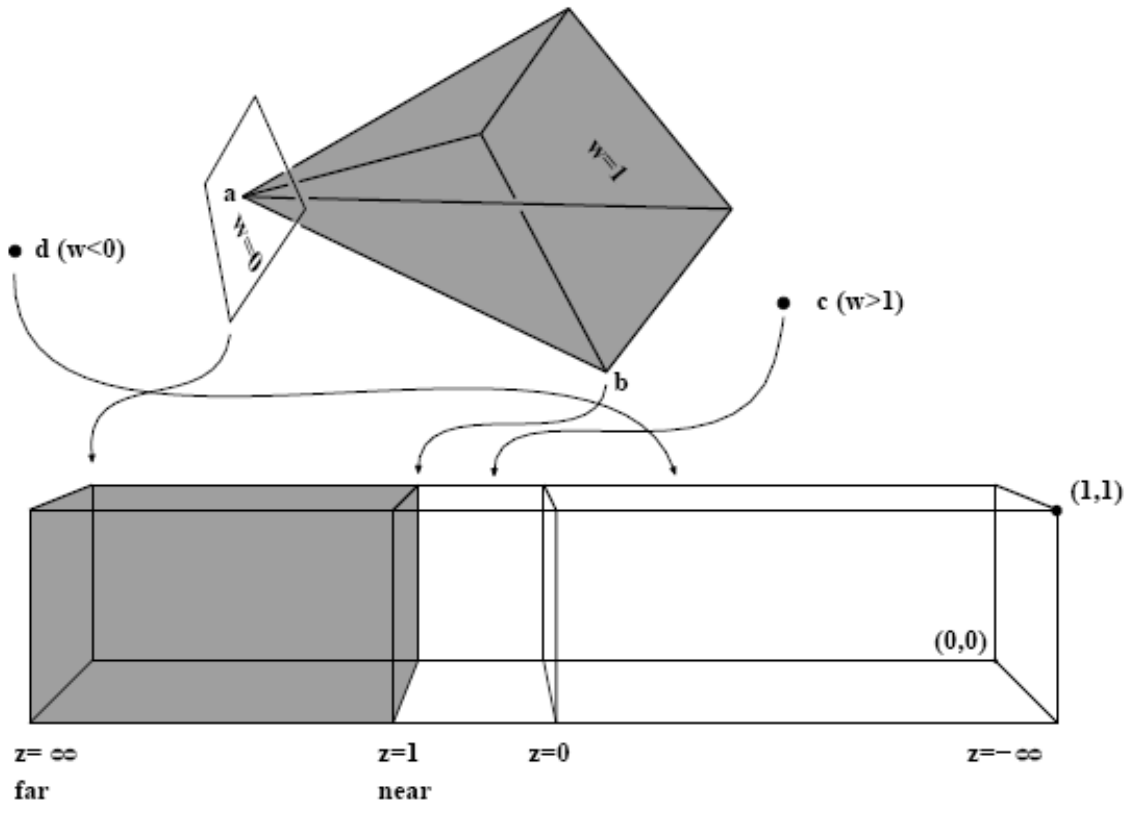


Figure 3.4. How the method of Heckbert and Herf transforms a pyramid into a parallelepiped which effectively culls out objects behind the light source and shadow receiver. Image taken from [3].

P is designed such that all objects which lie between the light source and the shadow receiver ($0 \leq w \leq 1$) are mapped between ($1 \leq z \leq \infty$) in the final unit parallelepiped. Those objects are valid shadow casters and will not get culled. All objects behind the light source ($w < 0$) are mapped between ($-\infty \leq z \leq 0$) and the objects behind the shadow receiver ($1 < w$) are mapped to ($0 \leq z \leq 1$) such that they are culled away by the graphics API after applying the projection matrix P. As told before, all images for different point light sources are additively hold in an accumulation buffer and then averaged for the final soft shadowed result, which is then applied as a regular texture map to the shadow receiver parallelogram [3].

Gooch *et al.* [45] proposes another projective soft shadowing technique, which does not require so many number of samples, as Heckbert and Herf's method does. In their scheme, instead of sampling the light source for multiple point light sources, they move the shadow receiving plane up and down for a single light source, such that different sized hard shadow projections on the receiver plane can be averaged for the final soft shadowed result (Figure 3.5). A single hard shadowed projection can be scaled up and down in order to represent the shadow receiver plane at different heights and this will re-render the whole scene for different samples. The obtained series of hard shadowed textures are later averaged into a final soft shadowed texture. The biggest drawback of this method is that the umbra region is always overestimated, since the projections of the object onto the receiver plane are always bigger than the object itself. Compared to reality, this is not always true since if the light source is much larger compared to the shadow caster, then the shadow may have no umbra region at all [3]. Another problem occurs when the object is very close to the receiver plane or actually touches it. This causes the upwards moved receiver planes to slice the object through. To prevent that, the distance between the object's nearest point to the receiver plane can be tested. If the upwards shifting of the receiver plane causes a collision with the object, we will shift the plane downwards instead. According to [3], this will lead to unrealistic looking shadows.

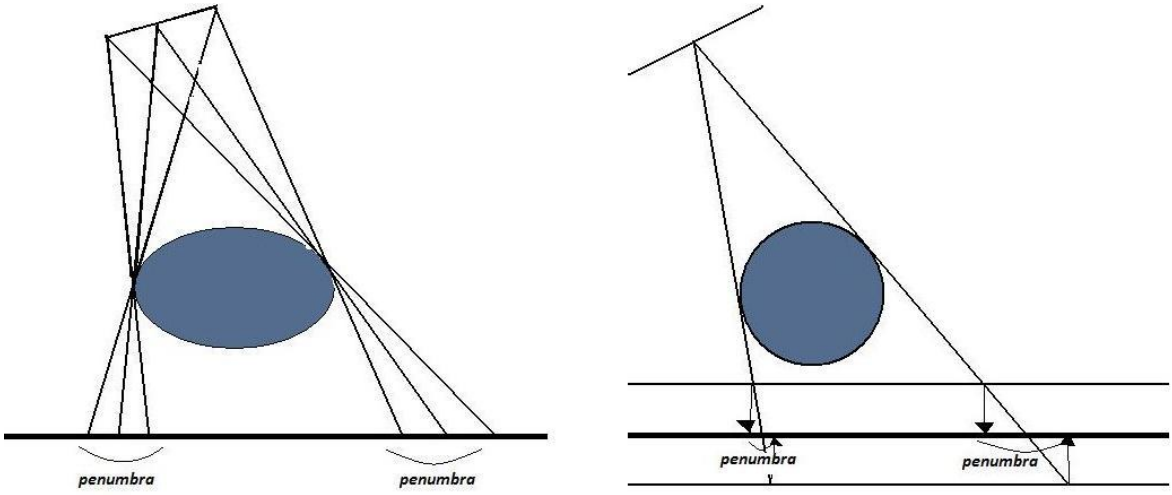


Figure 3.5. Gooch *et al.*'s method moves the receiver plane up and down(right) instead of sampling the area light source at different points(left). Image based on [3].

3.2.3. Projective Shadows for Curved Surfaces

The projection techniques of the previous two subsections are limited to planar receivers. To extend the projection technique to arbitrary surfaces, methods like [46–48] are developed. Such methods are very similar to shadow mapping with z-buffer technique of Williams, [40]. We render the scene from the point of the light source; everything visible by the light is not in shadow and the objects, which are not visible are occluded; meaning they are in shadow. A projection image of the scene is created in that way; all visible surfaces are drawn into a texture. When rendering the scene normally, the vertices of the receiver objects are projected onto this texture, by multiplying (x, y, z) coordinates of the vertices with an appropriate projection matrix and obtaining the (u, v) texture coordinates for them. The receiver objects are drawn with this shadow texture mapped on them, effectively creating the shadow effect. While going beyond planar receivers, this method has two disadvantages. Firstly, the generated shadow texture does not carry any information except the light space projections of the objects in the scene. This means we cannot automatically determine the receiver objects from the occluding ones by simply looking at the projection texture. In every scene, the shadow casters and receivers need to be separated manually; the shadow

texture should only be applied to the receiver objects. Secondly, with this method, occluders cannot cast shadows over themselves; in other words, this method does not support self-shadowing.

3.3. Shadow Volumes

Shadow volumes, which was first introduced by Crow in [39] is one of the oldest shadowing algorithms. It can cast shadows on arbitrary objects, meaning it has no limitation of planar receiver surfaces as the projection shadows do. The algorithm uses the observation of the geometry which shadows create with their casters. If we think of a point light source and a triangle as the shadow caster, the light beams starting from the light source and directed to the corners of the triangle will create an enclosed volume as they go to infinity. This is actually the *shadow volume* of the triangle; the geometric representation of the shadow which the triangle is casting. The shadow volumes method is based on the following observation: When a point p is inside the shadow volume or volumes of a caster or many casters, then it receives no light; meaning it is in shadow. If it is not encapsulated by any shadow volume, then it receives light, meaning it is not in shadow.

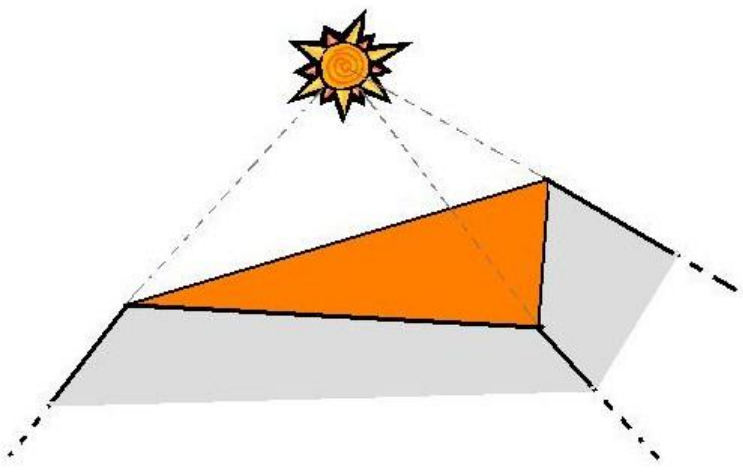


Figure 3.6. The shadow volume of a triangle, generated by a point light source.

3.3.1. The Basic Algorithm and Heidmann's Z-Pass Method

The basic version of the algorithm works with closed caster objects; the object from which the shadow volume is going to be generated, should not contain any cracks exposing its inside. For such closed objects, the algorithm needs the *silhouette edges* to be extracted from the object's mesh. Silhouette edges are the ones, which separates two face triangles of the object; one oriented towards the light source and one oriented backwards. Mathematically, this is determined by taking the dot product of the face normals and the light direction vector. After those silhouette edges are found, they are extruded in the direction of light. This generates the shadow volume of the object according to the light source. In [4], the cost of this operation is defined as *O(E)*, where E is the number of shadow polygons. This means the cost of silhouette extraction increases linearly with the complexity of geometry in the scene. Determining whether a point p lies in shadow is done with a containment test. A ray from outside of the shadow volume is shot from an arbitrary viewpoint to the point p. As the ray travels in the scene to p, for each front-facing shadow volume polygon it intersects (meaning the ray is entering into a shadow volume), an intersection counter is incremented. For each back-facing shadow volume it intersects (meaning the ray is leaving a shadow volume), the counter is decremented. The front or back facing states of the shadow polygons are determined according to the shot ray's direction. When the ray reaches to p if the counter is zero, then the amount of front-facing and back-facing shadow polygons the ray has intersected is equal, meaning p is not in shadow. If the counter is greater than zero, then p lies in shadow. The basic algorithm needs actual, expensive geometry tests for each ray shot from the viewpoint to needed points. This involves many ray-triangle intersection tests, slowing down the algorithm. Heidmann first adapted the basic algorithm for effective usage with graphics hardware in [49]. He proposed the usage of the stencil buffer instead of expensive ray-triangle intersections. The stencil buffer is used as a buffer for holding intersection counters for each pixel in the frame buffer. The pixels of the shadow volume polygons are rendered only to the stencil buffer, which increments and decrements the according entries in the stencil buffer depending on the orientation of the polygons (front or back facing according to eye position) they belong to. The stencil buffer rendering is done with z-test enabled. With

this, a point p will prevent rendering to stencil buffer, if it is closer to the viewer than its corresponding shadow volume pixels. By using the stencil and depth buffer in this way, after the shadow volume rendering is done, a stencil buffer pixel correctly holds the number of intersections from the eye to an arbitrary scene point p. Heidmann's *Z-Pass* algorithm can be stated as the following:

1.Pass Render the scene only with ambient lighting enabled.
2.Pass Turn off the writes to Z-Buffer and Frame Buffer. Enable depth (Z-Buffer) test. First render the front facing shadow polygons by incrementing the stencil buffer counters and then render the back facing shadow polygons by decrementing the stencil buffer counters.
3.Pass Render the diffuse and specular components of the lighting model only at pixels where the stencil buffer is zero; meaning the pixel is not shadowed.

Figure 3.7. Heidmann's Z-Pass Algorithm.

Heidmann's Z-Pass algorithm has greatly increased the performance of shadow volumes and today most graphics APIs provide functions to increment and decrement values in the stencil buffer while rendering; enabling single pass implementations of the algorithm. But the algorithm has some serious flaws, which are addressed in different approaches to the Z-Pass concept. Figure 3.8 shows the z-pass geometry.

3.3.2. The Z-Fail method or "Carmack's Reverse"

In the basic Z-Pass method, the algorithm will not work correctly if the view point is in a shadow volume itself. This can be corrected by initializing the stencil buffer to the number of shadow volumes the view point is contained within [6]. But this brings extra burden to the basic algorithm; it leads to expensive view point – shadow geometry tests which are executed on CPU. An alternative approach is developed in [50] by inverting the z-test for shadow volume intersections. This method is also called as *"Carmack's Reverse"*. The same idea is proposed by Bilodeau and Songy independently from Carmack [51]. In the Z-Fail approach, we count the shadows behind the current pixel, instead of the front ones. The stencil buffer is changed only when there is a

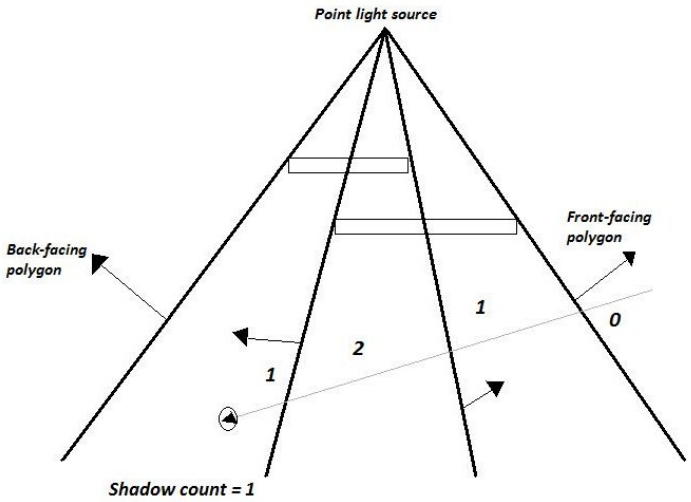


Figure 3.8. The Shadow Volumes algorithm in work: We increment the counter as we intersect front-facing shadow polygons and decrement when we intersect back-facing ones. Image based on [4].

depth test fail at the current pixel location. This effectively makes the shadow volume algorithm work with all camera positions. The algorithm is stated as in Figure 3.9:

1.Pass Render the scene only with ambient lighting enabled.

2.Pass Turn off the writes to Z-Buffer and Frame Buffer. Enable depth (Z-Buffer) test. First render the back facing shadow polygons by incrementing the stencil buffer counters only when there is a depth test failure and then render the front facing shadow polygons by decrementing the stencil buffer counters only when there is a depth test failure.

3.Pass Render the diffuse and specular components of the lighting model only at pixels where the stencil buffer is zero; meaning the pixel is not shadowed.

Figure 3.9. The Z-Fail Algorithm.

The Z-Fail algorithm needs the shadow volumes to be closed, since this time we are counting shadows from the pixel to infinity. Therefore, the shadow volumes need to be capped. This can be done by scaling and translating the light back-facing triangles of the shadow casting object to infinity, to the open end of the shadow volume.

3.3.3. ZP+ Shadow Volumes

ZP+ shadow volumes, proposed by Hornus *et al.* in [52] attempt to remove the limitation of the Z-Pass method that the camera must not be contained in the shadow volume. ZP+ shadows try to improve the Z-Pass method in terms of speed, by projecting the scene from the light position into the near plane of the camera. The scene, projected into the camera near plane, is drawn into the stencil buffer, with depth test open and only front faces of the objects are drawn. For every succeeding depth test, the stencil buffer is incremented. After this initial projection step, the rest of the scene is rendered with the standard Z-Pass method. What the projection to the camera near plane achieves is to fill the stencil buffer with correct values in order to run the standard Z-Pass method correctly, making the camera position irrelevant of the algorithm. The algorithm saves significant amount of speed by replacing expensive geometry tests for the camera position with a fast stencil buffer pre-rendering of the scene, but it suffers from numerical precision issues in the projection step, leading to cracks in the stencil buffer. This might lead to improper shadowing and lighting artifacts in the scene.

3.3.4. Speed Improvement Strategies for Shadow Volumes

Shadow volumes provide pixel correct hard shadows, but detecting the silhouette edges and drawing the shadow volumes into the stencil buffer are considered expensive operations. Silhouette detecting involves linearly growing geometry processing as the scene becomes more complex. Stencil buffer rendering introduces fill rate cost; the graphics hardware spends significant amount of time while filling the stencil buffer with correct values every time a shadow polygon is rendered. This cost can be quite high since quite often, polygons belonging to shadow volumes may cover a big portion of the whole screen. In order to address these two bottlenecks, different methods to reduce the geometry processing and to improve the fill rate have been proposed.

To reduce the geometry processing, the culling technique, introduced by Lloyd *et al.* can be used [5]. Lloyd *et al.* describe the purpose of culling as to eliminate shadow

casters that are in shadow themselves or the ones that cast no shadows contributing to the final image. In this scheme, a depth map is used along with shadow buffering. They firstly render the scene into a depth map; with depth and stencil buffer writing enabled. Depth buffer writing works as usual. Writing to stencil buffer only occurs when there is a depth test failure. By utilizing the stencil buffer in this way, only the shadowed regions of the receiver objects are marked. After these steps, the bounding boxes of the caster objects are drawn with occlusion queries, which is a feature supported by graphics hardware and APIs. For an object to pass the occlusion query, at least one of its pixels is needed to pass the pipeline and drawn into the frame buffer, by succeeding in all predefined tests. In this case, the tests are depth and stencil tests; passing the depth test indicates that the current pixel is not occluded by another shadow caster and passing the stencil test means that the current pixel is being written in a location where the stencil buffer is set before; in other words it is actually shadowing another object. The depth test eliminates casters occluded totally by another shadow casters and the stencil test eliminates casters which do not shadow eye-visible locations. Figure 3.10 illustrates a sample situation for caster culling.

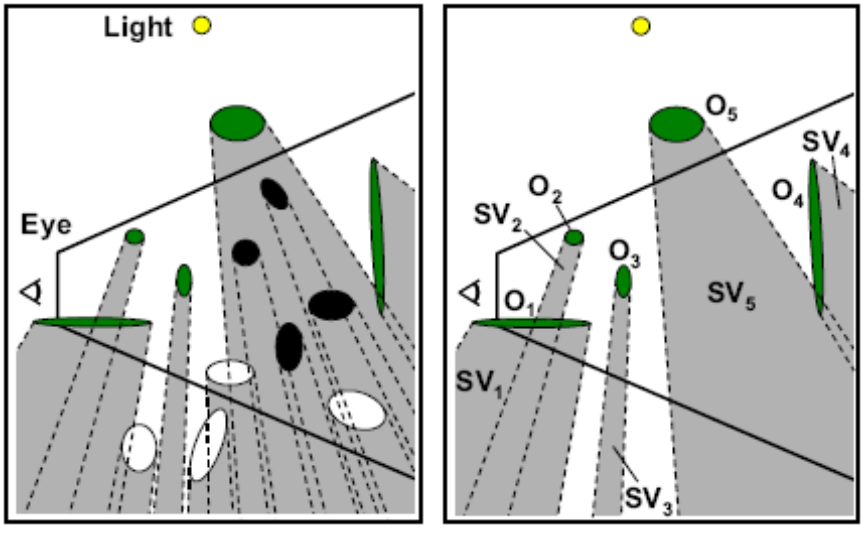


Figure 3.10. Shadow caster culling as proposed in [5]. Casters which are completely in shadow or casting shadows which are not visible to eye is eliminated. Image taken from [5].

There are hybrid approaches of shadow volumes and shadow maps to reduce the amount of geometry processing. In [53], McCool suggests to render the scene into a depth map, from the view point of the light source. In the depth map, discrete z-values of all visible objects in the scene will reside. After this depth map creation, McCool uses edge detection to find the silhouette edges. Then from the edge pixels found by the algorithm, vertices in 3D world space are created, which are then connected together and extruded into infinity, practically creating the cumulative shadow volume of all objects in the scene. This method works well in complex scenes, but as it is an image based method, it is prone to aliasing error generation.

To reduce the fill rate, one needs to reduce the amount of updates to stencil buffer. The fewer the graphics hardware updates the stencil buffer; the more time is gained to process other parts of the rendering pipeline. To this end, the *Split-Plane Shadow Volumes* algorithm is proposed by Laine [54]. The observation is the following: For each pixel, the Z-Pass or Z-Fail methods should be selected adaptively. If there is a point p which lies totally behind a shadow volume, with the Z-Fail approach, there won't be any updates to the stencil buffer as the shadow volume passes depth tests against p. In the same manner, for a shadow volume which lies completely behind the point p, the Z-Pass approach will not update the stencil buffer. It should be noted that changing the method from pixel to pixel will not cause any errors, since both methods are returning the same values for same pixels. To use this observation for actual performance gain, Laine approximately splits a shadow volume with a plane-quad, which has the depth z_{split}. If the point p has a smaller z value than z_{split} then it is likely in the front of the shadow volume; so Z-Pass method is used for it. The opposite is true for the Z-Fail method. Figure 3.11 shows the corresponding sketch.

Another effective idea to reduce fill rate is to clamp shadow volumes, such that they are not rendered where it is not necessary to do it. Two different but orthogonal methods to do it are proposed in [5]. The first method is called *Continuous Clamping*. In this method, for each shadow volume in the scene, the AABBs of each potential shadow caster projected on the light image plane, along with the shadow caster's AABB. Then overlap tests are done between the projected receiver AABBs and the

caster AABB, in order to identify the objects which actually receive shadow from the current caster. Finally, the intervals along the current shadow volume are computed by merging the extents of the shadow receivers it contains [5]. These two steps are shown in Figure 3.12.

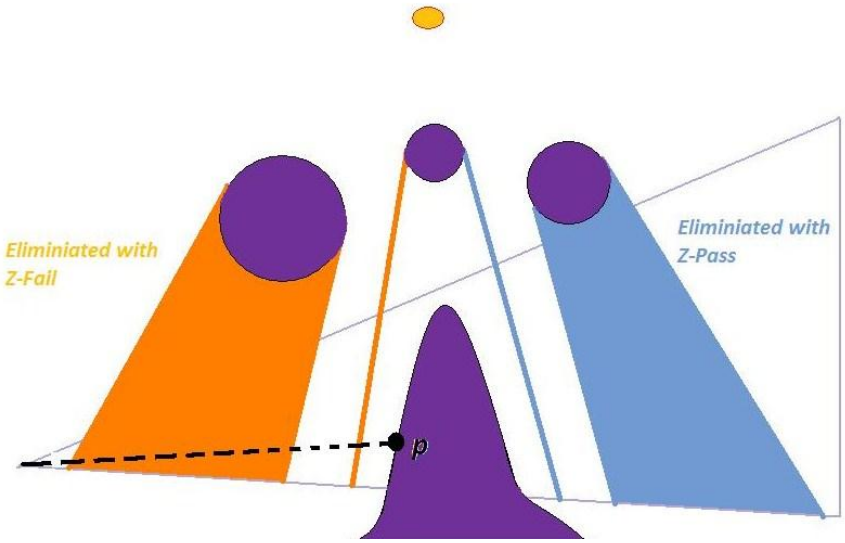


Figure 3.11. Adaptive utilization of Z-Fail and Z-Pass approaches can lower stencil buffer updates. Image based on [6].

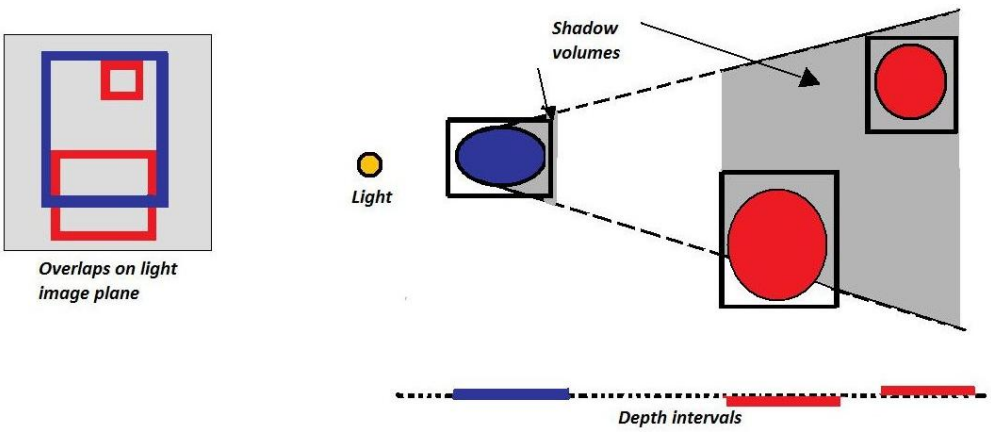


Figure 3.12. On the left, the caster and receiver AABB's are shown projected on the light image plane. On the right, the depth intervals of the actual casters are shown. Image based on [5].

In Figure 3.12 on the right, it can be seen how the shadow volume can be safely clamped to contain only the two actual shadow receivers, thereby reducing the fill rate by not rasterizing its empty regions. The other clamping method proposed is *Discrete Clamping*. Since entire extents of the receiver objects are used to generate clamping intervals in the Continuous Clamping method, they can result in poorly fit clamped shadow volumes if only a small part of an object resides in the shadow volume [5]. Discrete Clamping is designed to complete Continuous clamping in this aspect. In this method, the viewing frustum is partitioned with similarly-oriented planes. The discrete intervals for the shadow volumes are determined by intersection of the volumes with these frustum slices. The rest of the calculation is done in light's view. The slices are processed from back-to-front. For each slice, the front and back planes of the current slice are used as clipping planes and its contents are rendered. Then the current shadow caster is projected on the back slice. The depth test is set to let only pixels with greater z-values than the ones in the depth buffer to pass it and an occlusion query is executed. If the query indicates that no pixels have passed the depth test, this means the shadow volume of the current caster, corresponding to this slice, is empty, meaning it can be clamped. If there are pixels which pass the test, then this means there are receivers in the current slice and the corresponding shadow volume interval must be rendered. The process is illustrated in Figure 3.13.

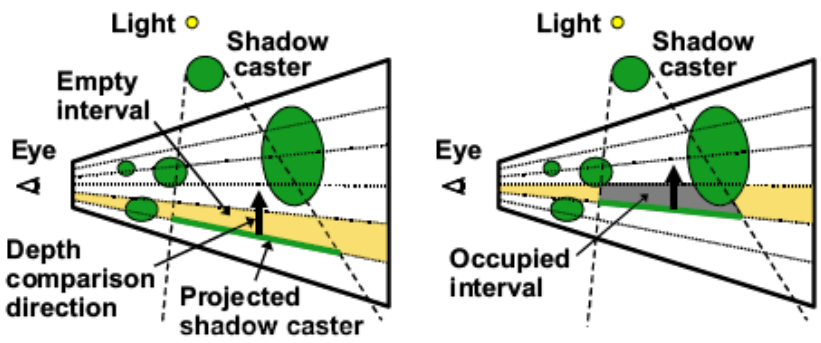


Figure 3.13. On the left, the shadow volume corresponding the current slice is empty and omitted from rendering. On the left, it intersects with a receiver, meaning the shadow volume interval for this slice must be rendered. Image taken from [5].

3.3.5. Summary

Shadow volumes technique is a good solution for the shadowing question in computer graphics. They provide alias free, pixel correct hard shadows. However there are performance problems. Although there are methods to improve them, the silhouette detection and fill rate consumption are still bottlenecks for shadow volume using applications. Moreover, the improvement strategies are often hard to implement. In general, shadow mapping techniques are faster than shadow volumes [6].

3.4. Shadow Mapping

Shadow mapping, first proposed by Williams in [40] is an image based method, as opposed to shadow volumes, which is a geometry based method [39]. It is perhaps the most popular and widely used shadow generation algorithm in the area of computer graphics. It owes this popularity to its generality, ease of use and high performance. It is general, in the sense that it does not need planar receivers like projection shadows or closed meshes like shadow volumes do. It can work with any rasterizable geometry [6]. It is easy to implement, because only an extra additional rendering pass and a single extra texture lookup are all that are needed to implement the basic algorithm. It is beneficial in terms of speed because it does not need any extra geometry processing and it does not suffer from high fill rate consumption issues like shadow volumes do. Therefore, it seems the ideal and ultimate shadowing solution in the first look.

Despite all its advantages, the basic shadow mapping algorithm has many serious drawbacks which need to be carefully analyzed and addressed. Firstly, since it is an image based method, it suffers from aliasing errors like all discrete sampling based methods do. These aliasing errors are grouped as *undersampling*, *oversampling*, *reconstruction* and *temporal incoherence* (or *temporal aliasing*) errors by the survey of Scherzer *et al.* in [8]. Secondly, there are so-called *depth bias* errors caused by undersampling and imprecisions in the depth information stored in the shadow map image [8]. There are countless improvement techniques in the literature to address these problems. To obtain a robust shadow mapping solution, one or more of these

improvements, most of the time in combination, must be used [55].

In this section we will analyze the shadow mapping method in detail. First, we will describe the basic idea and then describe why it is not sufficient only by itself and needs additional improvement techniques. Then, we will investigate the errors caused by the basic algorithm in detail and then explain the various improvement and optimization techniques in the literature.

3.4.1. The Basic Algorithm and Errors

The shadow mapping algorithm does not handle occluders separately. To extract global occlusion information and to test the visibility of light, the algorithm renders the scene from the viewpoint of the light into an image, with depth testing enabled, and stores the z-coordinates of every pixel to the final texture. After this rendering is done, the finally obtained depth image holds the minimum distances between every sampled occluder point, corresponding to an image pixel's center in the world space, and the light's position (point light sources) or plane (directional light sources). With this operation we obtain a pre-computed light visibility test for the whole visible scene [7]. Now, if we project the world coordinates of any point in the visible portion of the scene onto this depth image and compare the z-coordinate of the projected point with the sampled z-distance from the corresponding location on the image, we will make a binary light visibility test: If the sampled z-distance is larger than the projected point's z-coordinate, this means that the point is closer to the light source and it is not shadowed. On the contrary, if the sampled z-distance is smaller, there is a blocker between the current world point and the light source, meaning that the point is shadowed. The whole shadow map algorithm and all improvements that stem from it depend on this basic idea of sampled light-visibility. The algorithm can be stated as in Figure 3.14.

While the insight of the basic shadow mapping algorithm is very simple, there are two big problems related to it, which are needed to be addressed to obtain a robust solution. In [3, 6, 8], these problems are called as *depth bias* and *aliasing*.

1. Render the scene from the viewpoint of the light source into a depth image. For each pixel on the image, the closest Z-coordinate to the light source is stored.

2. Render the scene normally. Build a projection matrix from eye to light space and convert eye space coordinates to light space coordinates for each pixel.

3. If $\langle p_x, p_y, p_z \rangle$ are the light space coordinates of the current pixel, execute a visibility test by sampling the shadow map like: $v = sgn(shadow_map(p_x, p_y) - p_z)$

4. If $v > 0$, then the current pixel receives light. If the test fails, then the pixel is behind some blocker, meaning it is in shadow.

Figure 3.14. The Standard Shadow Mapping Algorithm.

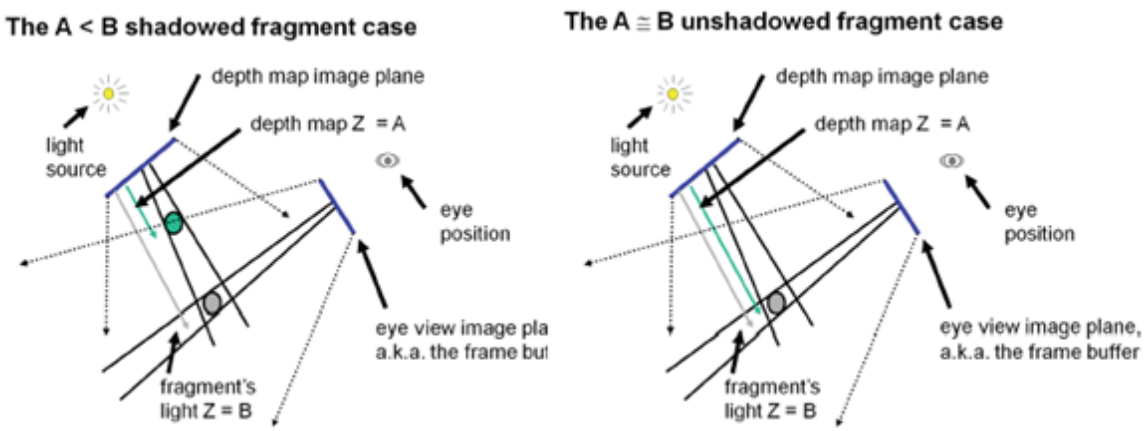


Figure 3.15. The basic shadow map algorithm, visualized. Image taken from [7].

3.4.1.1. Depth Bias. The shadow map stores only correct values for pixel centers. When more than one screen pixel corresponds to a single shadow map texel, then due to insufficient sampling and floating point imprecision, light receiving screen space pixels can erroneously shadow themselves, leading to an artifact called *shadow acne*, a term used to explain incorrect self-shadowing or shadow z-fight of the light receiving surfaces [8]. In order to address this problem, a small amount of depth bias is added to z-values written into the shadow map. But setting this bias correctly for the whole scene is problematic, since the amount of actual needed depth bias changes with the alignment of the current polygon with the light's view rays [6]. Most graphics APIs

provide a *slope scale bias* to add the z-values when rendering into the shadow map, which changes with the orientation of the currently rendered polygon. Unfortunately, the problem will still exist with these two bias modifications, since the shadow map depth buffer has a non-linear z-distribution. This can make the shadow acne problem worse when the objects far away from the light source, which share the lowest depth precision, are close to the view point, where we need the most precision. A naive solution would be to increase the precision of the shadow map depth buffer. When the light source is directional, the projection will be an orthogonal one which corresponds to a linear z-buffer, reducing the need of biasing. Another solution is to use an *ID-based shadow map* instead of a depth based one [56]. In this approach, one assigns different IDs to every shadow caster object. If a pixel reads the ID of its owner object from the shadow map, it means that it is in shadow. If the obtained value is zero or equal to the owner object's ID, the pixel is assumed to receive shadow. It should be noted that this method does not support self-shadowing by itself. The method can be extended by assigning a separate ID to each discrete polygon in the scene. But this is also problematic, since due to perspective projection, more than one polygon can assume sub-pixel sizes while rendering to shadow map, where we can write only a single value to a single pixel. *"Midpoint shadow maps"* is another attempt to solve the depth bias problem [57]. In this method, the scene is rendered two times from the viewpoint of the light. Both the back and front faces of each object are rendered and the average depth is stored into the depth buffer. This method can lead to *incorrect self-unshadowing*, when the stored average depth is larger than an actually shadowed pixel's distance to light and incorrect self-shadowing, when the opposite occurs. The mostly used and practical method to address depth biasing is the *"Second-Depth Shadow Mapping"* method introduced by Wang and Molnar [58]. The method assumes closed objects as shadow casters and proposes to render the second nearest surface to the light into the shadow map. The method is very simple to implement with current graphics APIs, by letting only the back-faces of objects to be drawn into the shadow map [8]. The method practically introduces a shadow depth bias in the size of the shadow caster. By rendering back faces only, we transfer the shadow acne problem as the light leaking problem to the back faces of caster objects. But the lighting model already assumes

only the ambient lighting for those faces and any light leaking artifact is successfully hidden with it, making the method most practical solution for depth biasing. To address the problems of the midpoint shadow maps, Weiskopf and Ertl propose to combine the average depth with a fixed bias [59]. They choose the minimum of a fixed bias and the current averaged depth and write the selected one into the shadow map. An interesting problem with depth biasing is called as *"Peter-Panning"* which results from too large depth bias added to shadow values. This artifact makes objects on the ground detached from their shadows and makes them appear as "floating" [60].

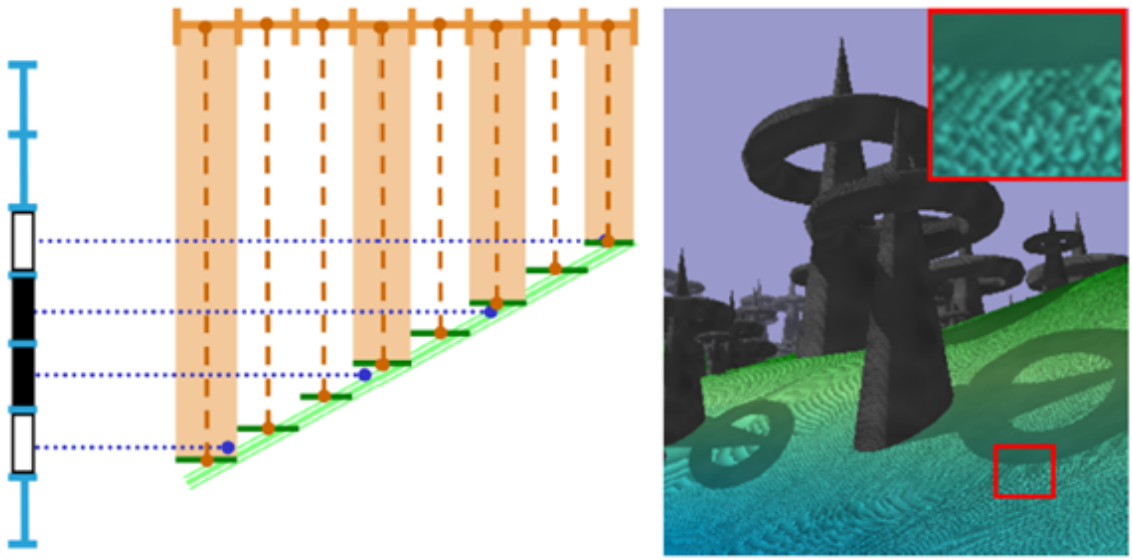


Figure 3.16. The z-fighting and incorrect self shadowing error caused by shadow map undersampling and imprecision. Image taken from [8].

3.4.1.2. Aliasing. Since shadow mapping is an image based method, aliasing is a natural side effect of it. The main reason of aliasing is the limited resolution of the shadow map. Shadow map is a discretely sampled representation of the visible scene and when reconstructing the visibility information in the view space, more than one screen pixels can correspond to a single shadow map texel or the contrary. This failure of one-to-one correspondence of screen pixels and shadow map texels result in different visual artifacts. The most detectable and disturbing artifact is the stair-step artifacts at shadow

edges. The survey of Scherzer *et al.* [8] defines four different types of aliasing errors. These are the following:

- Undersampling errors: This is the most common source of shadow map aliasing. It occurs when the shadow map texels projected on the scene have a lower sampling rate than screen pixels. As a result, more than one pixel will read the same depth value from the same shadow map texel, leading to stair-step edge artifacts.
- Oversampling errors: These occur when the projected shadow map texels have a higher sampling rate than screen space pixels.
- Reconstruction errors: These are the result of nearest-neighbor sampling of shadow maps. With the undersampling of the shadow map, they lead to stair-step artifacts [8].
- Temporal aliasing, flickering errors: If the rasterization of the shadow map changes each frame, pixels seem to come in and out of shadows very quickly as the camera moves or the light position or direction changes. The non-optimal reconstruction in an undersampled shadow map makes the artifact worse. For directional lights, the problem is reduced by making the texel boundaries constant in world space [6, 60]. However if the light position freely rotates and translates in the world space, controlling this becomes very hard. Hiding the error with a good reconstruction filter is the best solution.

The initial sampling stage of shadow mapping, in which we render the scene from the viewpoint of the light, is where the shadow map texel/screen pixel correspondence ratio is determined. If this sampling stage can be executed optimally, such that this correspondence ratio is as close as possible to the ratio of shadow map resolution to screen resolution, then all the errors listed above can be alleviated [8]. Therefore, the most intense effort for shadow mapping research is dedicated to finding new algorithms and methods, which aim to optimize this initial sampling step.

In order to optimize the sampling step, a mathematical model which shows how the sampling errors arise may be helpful. An attempt in this direction was made by Stamminger and Drettakis in their *"Perspective Shadow Maps"* paper [61]. They propose a simplified sampling error analysis which is 2D, assumes a directional, overhead light source and a parallel to view z-axis shadow map, which spans exactly from the viewer near plane to far plane. Figure 3.17 depicts a sample setting for the simplified sampling analysis.

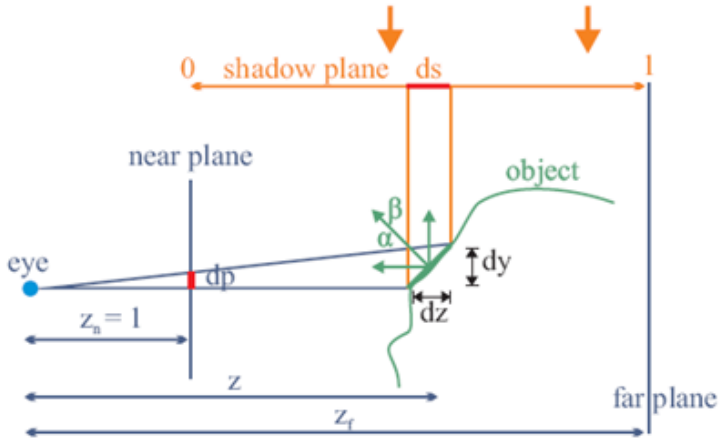


Figure 3.17. Simplified sampling error analysis for shadow mapping aliasing. Image taken from [8].

Following Figure 3.17, ds is the smallest unit (texel) of the normalized shadow map. The rays from both ends of a shadow map texel intersect with an object; in the z-axis, the length of this intersected small part is dz, which is related to ds with $dz = (z_f - z_n)ds$. In the y-axis, the height of the intersected part can be shown with $dy = dz\frac{\cos\alpha}{\cos\beta}$. If the near plane distance is assumed to be one and the intersected part's distance from near plane is z, the projected length on the near plane can be shown as $dp = \frac{dy}{z}$. From these definitions, the shadow map aliasing error is defined as the following:

$$\frac{dp}{ds} = \frac{dz\cos\alpha}{zds\cos\beta} \tag{3.4}$$

In this equation, if dp is greater than the size of a screen pixel, or $\frac{dp}{ds}$ is larger than $\frac{Resolution_{ShadowMap}}{Resolution_{Screen}}$, shadow map undersampling occurs. Stamminger and Drettakis define two different undersampling types. The first is the perspective aliasing and it appears when $\frac{dz}{zds}$ is large. When we get closer to a shadow receiving surface in the eye space, the surface will cover more screen area due to perspective projection. This means that more screen pixels will correspond to a single shadow map texel when we project them onto the light plane. Since z is the free parameter in the aliasing equation, we can easily see that a smaller z results in greater amounts of aliasing. This type of aliasing is especially visible in places near to the viewer camera and therefore it is global and generates very disturbing stair-step artifacts [55]. Since it directly relates to the insufficient sampling of the shadow map and generates the most disturbing artifacts, most of the literature on shadow mapping try to reduce *perspective aliasing*. The second type of aliasing called as projection aliasing. It happens when $\frac{\cos\alpha}{\cos\beta}$ is large; when a surface is almost parallel to the incoming light's direction. This is a local error and is directly related with the orientation of a specific surface and therefore it is much harder to detect and to reduce. Methods which aim to improve projection aliasing need a detailed scene geometry analysis to detect surfaces with large $\frac{\cos\alpha}{\cos\beta}$. Projection aliasing is still an open problem for real-time shadow mapping approaches [55]. Clever scene design and blur filters are the most practical solutions for this kind of error. Stamminger and Drettakis point out that the illumination model can automatically help with fighting projection aliasing like it does in depth biasing: For surfaces where $\frac{\cos\alpha}{\cos\beta}$ is large, this mathematically leads to the fact that the dot product between the surface normal and light direction, $\overrightarrow{N}.\overrightarrow{L}$, is small. A small $\overrightarrow{N}.\overrightarrow{L}$ means a darker shaded surface; as the projection aliasing gets worse, the surface will become darker. This can effectively "hide" projection aliasing artifacts [61]. In Figure 3.18, these two different types of undersampling aliasing are shown. In the left figure, the orientation of the

tree surface against the light direction causes projection aliasing. In the right one, a shadow map texel covers a larger area when near to the camera, leading to perspective aliasing.

An accurate, 3D sampling error analysis is more complicated but possible. Lloyd defined such an analysis in his Ph.D. thesis [62]. We refer the reader to it for details.

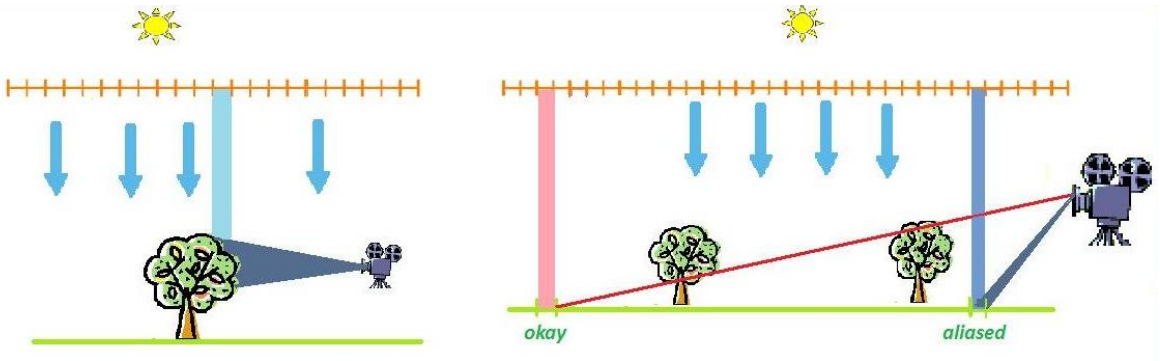


Figure 3.18. Projection aliasing (left) and perspective aliasing (right). Image based on [8].

3.4.2. Hard Shadow Mapping Methods and Reduction of Sampling Error

Sampling error reduction methods aim to modify the initial shadow map sampling stage, such that the projected shadow map texels correspond to the lowest amount of screen pixels, as possible. Many different methods have been proposed in order to realize this goal. We give a summary of the prominent sampling error reducing methods.

3.4.2.1. Focusing (or Fitting) the Shadow Map. A quick observation to reduce the sampling error is one that we must utilize the highest possible area of the shadow map image to cover the whole scene, which is visible to the camera. We should avoid rendering empty and unnecessary (invisible) parts of the scene into the shadow map. To this end, before rendering the scene into the shadow map, the convex hull of the

view frustum and light position is taken and this convex hull is clipped to scene bounding volume and light frustum. Brabec *et al.* showed the importance of focusing the shadow map when trying to reduce the sampling error [63]. Most of the error reduction methods assume that the shadow map is already focused on the visible parts of the scene and therefore, focusing is a fundamental need for a robust shadow map implementation [55]. Since the shadow map rasterization changes every time when the camera or the light source moves, focusing causes temporal aliasing or so called "flickering" errors [6].

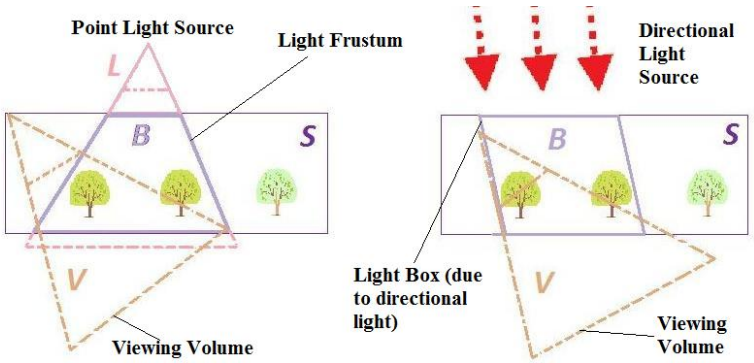


Figure 3.19. Shadow map focusing under point light (left) and directional light (right). Image based on [6].

3.4.2.2. Warping Based Methods. We have already shown that perspective aliasing becomes worse on surfaces near the camera. If we had a way to redistribute the shadow map samples more densely for locations near to camera and more sparsely for locations far to the camera, the perspective aliasing would benefit from it. Warping methods exactly do that. The first work to propose the redistribution of shadow map samples is the *"Perspective Shadow Maps"* of Stamminger and Drettakis [61]. They first transform the visible scene from the eye space to eye's post perspective space and after that; they render the scene into the shadow map. In the eye space, this assigns a larger shadow map sample distribution to near objects. Although a clever idea, this method had some robustness and quality issues. First, shadow map quality degenerates too quickly for objects far away from the camera. Secondly, the post perspective space

changes the direction of the light sources and there are many special cases to handle, making prediction of the shadow quality and overall implementation hard. Finally, the shadow quality changes so rapidly from frame to frame as to cause disturbing flickering artifacts. Alternative methods for improving the clever idea of warping, introduced in [61], while trying not to retain its disadvantages, have been proposed. *"Trapezodial Shadow Maps"(TSM)*, proposed by Martin and Tang in [64], uses a so-called *"trapezoidal space"*. They use the fact that when projected on the 2D light plane, the view frustum is geometrically most similar to the shape of a trapezoid. They define a trapezoid approximation of the view frustum and from this, a transformation is extracted which converts the light's post perspective space to the trapezoidal space. By doing this conversion, firstly they minimize the wasted portions of the shadow map used for empty regions and secondly they redistribute the shadow map samples to alleviate the perspective aliasing problem. A second and similar method is called as *"Light Space Perspective Shadow Maps" (LiSPSM)*, developed by Wimmer *et al.* in [65]. As their main contribution, they decouple the warping from the eye's post perspective space and define a new, independent light space warping transformation after the light space is determined. This perspective transformation is specified with respect to the coordinate axes of light space, such that lights do not change their directions and types after the transformation, which is the case in perspective shadow maps [65]. Moreover, by defining a free parameter which controls the strength of the perspective warp, they ensure that the perspective transformation does not degrade the obtained shadow quality below the standard shadow mapping. The formula for calculating this free warping parameter is derived from perspective aliasing error analysis [55]. As a result, for situations where light and view directions are perpendicular, they obtain the same transformation as perspective shadow maps introduced in [61]. For the worst case, where the view and light directions are parallel, the perspective transformation loses its effect and the method converges to uniform shadow mapping [65]. Following the simplified error analysis of Stamminger and Drettakis, where $ds = \frac{dz}{z}$, Wimmer *et al.* define in [65] the optimal shadow map sample distribution as the following:

$$s = \int_0^s \mathrm{d}s = \int_{z_n}^{z} \frac{1}{z}\,\mathrm{d}z = \ln \frac{z}{z_n} \tag{3.5}$$

Equation 3.5 shows that the optimal distribution of shadow map samples for an ideal directional overhead lighting setting, for which all other requirements of the Equation 3.4 hold, is logarithmic. In [62], Lloyd derives a logarithmic perspective warping transformation, based on his accurate sampling error analysis. While very effective, this *"Logartihmic Perspective Shadow Mapping (LogPSM)"* approach lacks hardware support, because the interpolation stage between vertex shaders and pixel shaders is hyperbolic on the current graphics hardware, not logarithmic [8]. To address that, Lloyd proposed hardware modifications to support his warping method. Figure 3.20 shows a general image representation of the perspective warping approach.

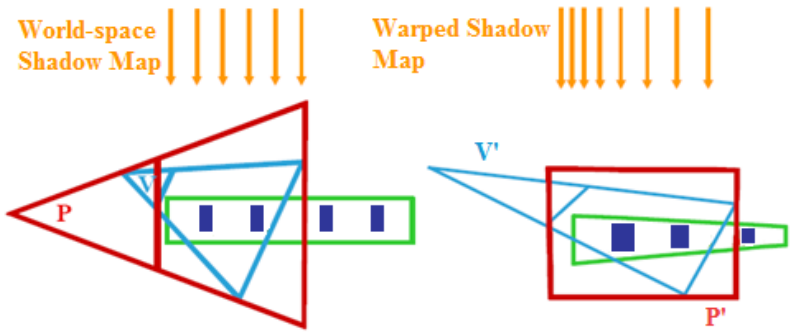


Figure 3.20. On the left, the uniform shadow map sampling is shown. On the right, the reparametrization of shadow map samples is shown, after the perspective warp. Image based on [8].

3.4.2.3. Global Partitioning: Parallel Split Shadow Maps. The most practical and prominent approach to reduce the sampling error is to divide the viewing frustum into different parts and to construct a separate shadow map for each frustum slice [8]. If the viewing frustum is sliced cleverly, the shadow map resolution can be increased at

needed locations very easily by allocating the needed amount of resolution to a specific frustum slice. The most popular algorithms which uses the partitioning approach are *"Parallel Split Shadow Maps(PSSM)"* of Zhang *et al.* [66] and *"Cascaded Shadow Maps"* of Engel [67]. These papers actually describe the same algorithm under different names. The fundamental challenge of partitioning is to determine where to slice the view frustum. According to Equation 3.5, an optimal splitting scheme will be one which makes the aliasing error $\frac{dp}{ds}$ constant over the whole depth range. The logarithmic split scheme, derived from Equation 3.5 is stated as following in [66]:

$$C_i^{log} = z_n(\frac{z_f}{z_n})^{\frac{i}{m}}$$

Where C_i is the splitting location, m is the number of splits and i is the current split. Zhang *et al.* argue in [66] that the logarithmic split scheme is not practical by itself since it allocates too much resolution to small slices near to the camera, where only few objects would fit, if any, and too low resolution to far away parts of the viewing frustum, where most of the actual scene resides. Therefore, they devised a second, uniform splitting scheme, which linearly partitions the view space z-axis. As the final approach; they propose a practical splitting scheme, which linearly interpolates between logarithmic and uniform splitting according to the needs of the current setting. Those are given as the following:

$$C_i^{uniform} = z_n + (z_f - z_n)(\frac{i}{m})$$

$$C_i^{practical} = \alpha C_i^{uniform} + (1 - \alpha)C_i^{log}$$

Partitioning approaches of [66] and [67] are preferred as the most practical and efficient algorithms to reduce the sampling error. They do not suffer from the special cases from which warping methods do; their shadow quality does not degrade at some specific viewer and light direction settings. Therefore, it is the choice of most state-of-the-art graphics applications; like *CryEngine 2*, the graphics engine of the popular video game *FarCry* [68]. In [68], it is reported that perspective shadow maps [61] and trapezoidal shadow maps [64] are tried along with cascaded shadow maps [67] and cascaded shadow maps gave better shadow quality overall. In [22] and [69], implementation strategies for cascaded shadow maps are given, for the popular graphics APIs, OpenGL and DirectX, in respective order. Partitioning and warping strategies can be combined together: For example each frustum slice can be rendered into its corresponding shadow map by using any of the algorithms of [61], [64] and [65]. But this is deemed as unnecessary, even prohibitive, because it introduces a disturbing amount of flickering errors and incoherence of shadow quality at frustum slice borders while the reported shadow quality improvement is insignificant ([6, 8, 69]).

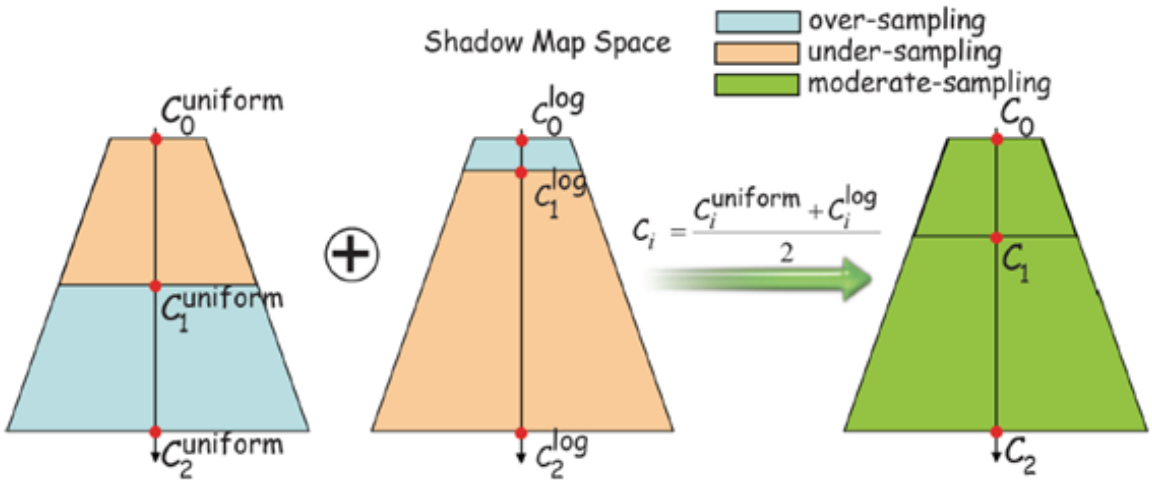


Figure 3.21. Parallel splitting of the view frustum. On the right, logartihmic and uniform splittings are mixed and the practical splitting scheme is obtained. Image taken from [7].

3.4.2.4. Other Strategies For Sampling Error Reduction. There are other error reduction techniques, which differ from the former fitting, warping and partitioning techniques. Those did not reach the popularity of the previous methods, as they are either impractical to implement and maintain, lack hardware support or have serious performance drawbacks.

Adaptive partitioning is a family of algorithms which try to allocate shadow map parts in a more optimal way by doing a detailed analysis of the scene. More shadow map resolution is allocated for locations where the most resolution is needed. The shadow map resolution allocation is usually managed by a hierarchical structure, like a quadtree. The detailed scene analysis provides reduction of the projection aliasing, which the former methods cannot address. This allocation management involves many read backs from the graphics hardware and therefore these algorithms usually only reach interactive frame rates, but the obtained shadow quality is very good for general cases [6, 8, 55]. Some prominent algorithms of the adaptive partitioning approach are *"Adaptive Shadow Maps"* [70], *"Queried Virtual Shadow Maps"* [71], *"Tiled Shadow Maps"* [72] and *"Fitted Virtual Shadow Maps"* [73].

Irregular sampling approach arises from the fact that eye space pixels do not correspond to a regular grid anymore when they are projected onto the light plane. These methods try to improve the shadow quality by sampling the shadow map on an irregular basis, exactly at locations where eye space pixels actually correspond. Since the graphics hardware only supports regular sampling patterns, achieving this aim is challenging. Johnson *et al.* propose hardware modifications for that in [74].

Temporal reprojection aims to increase shadow quality by increasing shadow map sampling rate by reprojecting samples from older frames onto the current frame's shadow map. In each frame, the shadow map is jittered differently and when the results are combined into a final image, its effect is increasing the quality of shadows [75].

3.4.3. Filtering Hard Shadows and Shadow Map Reconstruction Techniques

Shadow mapping is an image based method and therefore, when reading data from it, we need a reconstruction step when reading a data which is not discretized on the shadow map image. Color textures used in common texture mapping use hardware supported bilinear and trilinear filtering for this purpose but we cannot use bilinear filtering for shadow maps; the final result of a shadow map read is a binary value: A pixel is either in shadow or not, according to the comparison of the sampled depth value with the current pixel's depth. This binary query results in jagged, stair-stepped hard shadow edges where the shadow map is undersampled. Using bilinear filtering like regular textures will not improve this, they will erroneously blend discrete depth values in the shadow map and since the final sampling result is binary, this will result only in moving of the hard jagged shadow edges back and forth instead of a blurring effect.

The first attempt to address this reconstruction problem was proposed by Reeves *et al.* under the name of *"Percentage Closer Filtering (PCF)"* in [9]. The basic idea of percentage closer filtering is to apply a filter to the shadow test results instead of depth values of the shadow map. For this purpose, for each pixel, its corresponding location on the shadow map and additional locations near to it are sampled and the binary shadow comparison results are averaged. The selection of additional locations can vary; a kernel of equal distance samples, a Poisson disk sampling pattern and a random or jittered sampling pattern can be used. Most PCF implementations use an equal distance regular grid kernel. The size of the PCF kernel directly influences the thickness and softness of the produced soft shadow edges; a larger kernel means a softer shadow and better hiding of hard, jagged edges. But since the PCF is executed for every rendered pixel, larger kernels will reduce general performance because of limited texture memory bandwidth and sampling latency. Although it is dated, it is a very clever and practical idea and therefore, PCF is still the most common shadow map filtering technique.

The problem of shadow map filtering is, as said before, its binary and unlin-

ear nature. Recent research on shadow map reconstruction offers some alternatives to correct this. The most prominent and effective approach on this direction is the *"Variance Shadow Mapping (VSM)"* technique, proposed by Donnely and Lauritzen in [76]. Main idea of this technique is to store both the z-depth and its square in a shadow map texel. By doing that, they assume that they store the mean and mean squared of the distribution of depths which correspond to the footprint of the current shadow map texel in the world space.

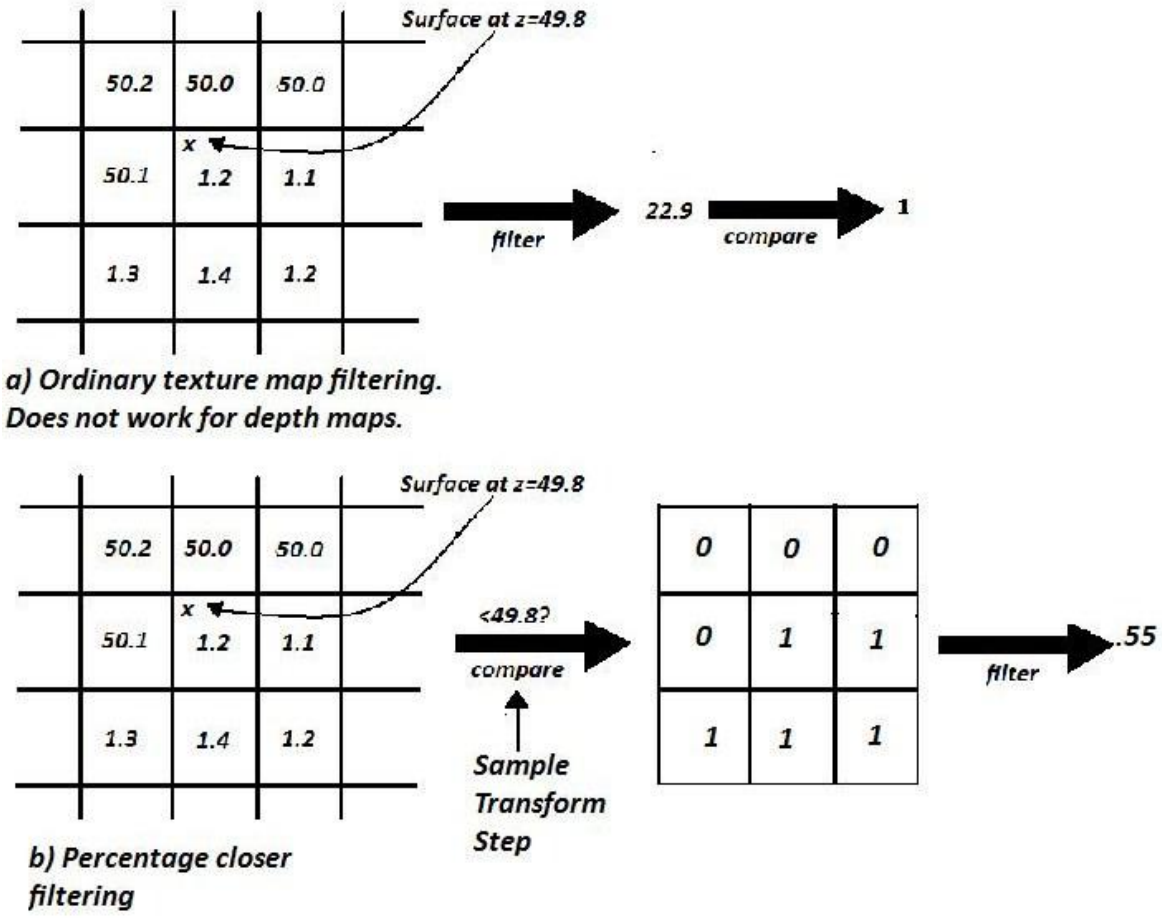


Figure 3.22. Percentage closer filtering. Directly filtering depth values does not produce any blur, since the result will again be binary. Filtering shadow test values produce a soft shadow, instead. The depth values are courtesy of [9].

This assumption linearizes the shadow map and it can be pre-filtered. Donnely and Lauritzen propose a separable Gaussian blur filter as the prefiltering operation,

since its cost grows linearly with the kernel size, not quadratic as PCF. In the pixel shader, the shadow lookup can be done using regular bilinear texture filters. From the sampled mean and mean squared values, the variance δ^2 is calculated and for a pixel with depth value t and sampled mean μ, if it is $t > \mu$, then the amount of shadow is given by Chebyshev's inequality:

$$P(x \geq t) \leq p_{max} \equiv \frac{\delta^2}{\delta^2 + (t - \mu)^2}$$

The problem with this approach is that high depth complexity, especially stacked occluders, can produce high variance which in turn leads to disturbing light leaking artifacts. To address this issue, Lauritzen and McCool devised *"Layered Variance Shadow Maps"* which slices the depth range into multiple layers to isolate occluders in different depth layers [77]. While reducing light leaks, many layers needed for better quality and that can result in too much layers to handle; especially when combined with the cascaded shadow mapping technique and therefore it is not recommended [69].

While not popular as variance shadow maps, alternative pre-filtering approaches exist. *"Convolution Shadow Maps"* proposed by Annen *et al.* use Fourier expansion instead of a statistical test for depth testing. For a practical approximation, 16 sine and 16 cosine coefficients are needed to be stored and this limited approximation leads into light leaks and ringing artifacts [78]. *"Exponential Shadow Maps"* by Annen *et al.* approximates the depth test as a step function and uses an exponential to represent this function [79]. This approach leads to artifacts when the kernel size grows since the approximation becomes wrong for such kernels.

3.5. Physically Based Soft Shadows

The hard shadow filtering methods we discussed in the previous subsection provides soft edges which leads to a more realistic shadow appearance. But the actual

job of such methods is to provide blur filters to reduce jagged-edge aliasing for hard shadows. The resulting soft edges are not physically based. In order to be considered as physically based, the penumbra calculation must take the size and shape of the light source, the distance of the light source from the blocker and the distance of the blocker from the receiver, into account. The blurring methods from the previous subsection operate only on a fixed sized blurring window; they do not include light-blocker and blocker-receiver distances in their soft edge calculations. The soft shadowing effect resulting from those filtering methods usually give visually satisfactory results but this plausible soft shadow appearance is only a welcome side effect of these methods [6].

Physically based soft shadowing is a very challenging problem because one must consider the global geometry of the whole scene for potential shadow casters. Since in our work we only concentrate on soft shadows resulting from the Sun and we propose some approximations to simulate the real soft shadowing effect, we do not attempt to do this and limit ourselves with a basic discussion. For detailed reviews of image based and geometry based soft shadowing approaches, we refer the reader to the surveys of Hasenfratz *et al.* in [80] and Eisemann *et al.* in [6].

We have discussed that a larger PCF kernel will lead to softer shadow edges and a smaller one results in a thinner penumbra area. The *"Percentage-Closer Soft Shadows (PCSS)"* technique of Fernando in [10] is built on this observation. The algorithm uses a blocker distance estimation heuristic for each pixel, and from the obtained distance; it adjusts the size of the applied PCF kernel for the current pixel. For the blocker distance estimation, the algorithm checks an area in the shadow map and averages the z-distances which are closer to the light source than the current pixel as the blocker distance. The search region is a rectangular kernel around the pixel's projected foot print on the shadow map. The search region's dimension depends on the size of the light source and pixel's distance from the light source. According to the returned blocker distance, the algorithm calculates the penumbra size with the following formula:

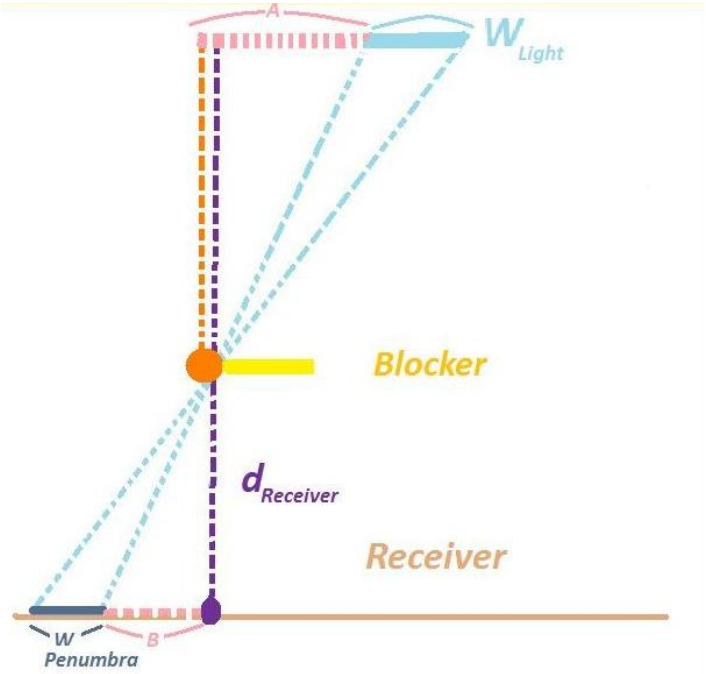


Figure 3.23. The penumbra region of an area light source changes with the distances between the source-blocker and blocker-receiver. Image based on [10].

$$w_{penumbra} = \frac{(d_{receiver} - d_{blocker})w_{light}}{d_{blocker}} \tag{3.6}$$

The size of the PCF kernel to be applied in the current pixel is then proportional to $w_{penumbra}$. While providing a good approximation for soft shadowing, PCSS has some drawbacks. First, the algorithm uses a parallel light source – plane blocker approximation, which is not valid in all cases. The blocker search step is not sensitive to whether a point on the shadow map, which is closer to the light source than the current pixel, does actually block the pixel in question. Therefore, this step can lead to overestimated, large shadows. Moreover, varying PCF kernel size will lead to serious self-shadowing artifacts.

PCSS is a good example to show how physically based soft shadowing algorithms

take light-blocker and receiver distances and sizes into account. There are other similar soft shadowing algorithms which use different approximations to calculate these parameters. A detailed review of such algorithms is out of scope of this work and we refer the reader to the corresponding surveys in [6] and [80] for detailed information.

3.6. Methods Specialized for Terrain and Height Map Shadowing

Height fields or height maps are the most common method to represent terrain data. Height map structure is usually tessellated into a huge number of triangles and therefore rendering it in a reasonable time is a hard problem. Shadowing of height maps is another hard problem. When talking about shadowing of a height map, we mean shadows "generated" by the height map only, meaning the self shadowing of the height map and shadows cast on other objects by the height map itself. Mainstream shadowing algorithms like projection shadows, shadow volumes and shadow mapping run into different difficulties when used for height maps. Projection shadows are immediately eliminated since they only work with planar receiver surfaces. Shadow volumes algorithm runs into performance issues since it is geometry sensitive; it needs to extract the height map silhouette to obtain the actual shadow volume. The work which is needed to be done in order to do that is huge, since a height map consists of thousands of triangles. Moreover, height map is not a closed object which is a requirement for silhouette extraction. The most logical choice for height map shadowing is to use one of the many different shadow mapping methods. While it is possible to use shadow mapping as a solution for height map shadowing, it will run into the following difficulties:

- Aliasing errors: Shadow mapping will suffer from more serious aliasing errors when used for a height map; since the geometry rendered into the shadow map is very large. This is due to the fact that the limited resolution of the shadow map has to cover a larger area in world space measurements. If we refer to Equation 3.4, since the shadow map spans a larger area of the world geometry, ds becomes smaller and the aliasing, $\frac{dp}{ds}$, becomes larger.

- Self shadowing issues: Height map is a very large self-shadowing geometry. This fact, when we keep in mind that the limited shadow map resolution covers huge world space geometry and the shadow map has limited floating point precision, leads into disturbing depth biasing problems [81]. Determining a depth bias which minimizes global shadow acne errors is hard in the case of a height map, since the amount of self-shadowing is very high.
- Additional rendering of height map geometry: Shadow mapping requires that the scene is rendered from the view point of the light before the actual rendering step. When shadowing a height field, this means that the extra rendering of a highly tessellated geometry will be a performance bottleneck, especially in the vertex shader stage of the graphics pipeline.
- Handling LOD algorithms: Rendering a high polygon height map model is a very expensive operation unless some kind of Level of Detail algorithm is utilized which modifies the height map's polygon count according to its distance from the camera. In order to increase performance, shadow maps usually render LOD simplified height map geometry into the shadow map image. This is suboptimal, since constantly changing world geometry will cause the rendered shadows to rapidly change with camera movements. Small changes in the world geometry caused by LOD and GeoMorphing algorithms may reflect as big changes to the rendered shadows, due to the applied projection on wide surfaces.

To provide high quality and high performance shadowing for height maps, specialized shadow algorithms have been devised. The most intuitive way to do terrain shadowing is to shoot a visibility ray from each height map vertex to the Sun and to check, whether the current vertex receives sunlight or is blocked by the terrain. Terrain occlusion is checked with many terrain-ray intersection tests as we march from the current vertex to the Sun. This can be called as the *"Brute Force Ray Marching"* method. Each shadowed vertex receives only ambient lighting. The result of the algorithm is saved into a *light map* or *illumination texture* which is then applied to terrain by multi texturing. In [82], the implementation of this brute force approach is given in detail. The obvious drawback of the algorithm is its long running time. For a N^2

sized height map, a vertex will need to scan through the whole terrain, diagonally, in the worst case. Therefore, the running time of the algorithm becomes $O(N^3)$, which is not suitable for real-time applications, even for interactive ones. Anyhow, this basic algorithm constitutes a valuable basis for more complex terrain shadowing algorithms. Moreover, it can still be used as a preprocess for an application, if the Sun does not change its position during the run time (static light source).

"Horizon Mapping" by Max N. L. is another specialized height map shadowing method [83]. The original paper in [83] aims shadowing for bump mapped surfaces only, which are derived from height maps. In [84], the algorithm is extended to cover height maps by using the OpenGL graphics API. This method is based on the definition of a *horizon angle*. For a point p on the height map, the horizon angle ω, for the azimuthal direction φ is the smallest angle, at which a light ray on the direction φ can reach the point p unblocked by the height map. For each point on the height map, we calculate the horizon angles for an arbitrary number of azimuthal directions and store it into a texture. Usually, the horizon angles are calculated for 8 primary directions (N,E,S,W,NE,SE,SW,NW). The horizon angle calculation is again an $O(N^3)$ operation for a single given direction on a N^2 sized height map; since we need to take samples which scan the whole height map in the current direction, in order to determine the correct horizon angle. For a azimuthal directions, the total calculation cost will become $O(aN^3)$. But despite this high calculation cost, horizon mapping has a very important advantage over brute force ray marching; it is the way we determine whether a point on the height map is shadowed. If the light comes from the direction $\overrightarrow{d}$ and $\overrightarrow{d_{xz}}$ is $\overrightarrow{d}$'s projection on the xz-plane, we calculate the horizon angle for $\overrightarrow{d_{xz}}$, ω, by interpolating between the horizon angles for the two closest primary directions. Finally, we compare the angle between $\overrightarrow{d}$ and $\overrightarrow{d_{xz}}$, i.e., $\cos^{-1}(\overrightarrow{d}.\overrightarrow{d_{xz}})$ with ω and if ω is larger than the incoming light's angle, this means the light ray cannot reach the current point: It is blocked by the terrain and we shadow the current point. If ω is smaller, the point receives light unblocked and is fully lit. By interpolating between the horizon angles for primary directions, horizon mapping can be used as an efficient real-time algorithm for terrain shadowing if the light source is dynamic. But the interpolation step can cause the algorithm to easily miss shadows cast by heights between the sampled primary

directions.

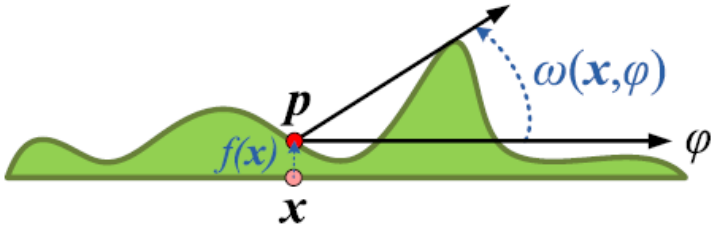


Figure 3.24. Horizon angle in a given direction is the smallest angle at which a point p on the terrain can see the sky unblocked. Image taken from [11].

Horizon mapping considers only points on discrete directions as possible occluders and interpolate between horizon angles for undiscretized directions. To improve this aspect of the algorithm and to create a method which considers all height map vertices as potential occluders for every single vertex, Stewart developed a new approach in [85]. For every point p on the height map, he defines s sections which divide the circle with the center p into equal parts, with each part having the angle $\frac{2\pi}{s}$. From all points inside of a section, he computes the maximum elevation and horizon angle for that section by utilizing computational geometry. Observing that different points have similar points in their overlapping sectors, he optimizes his algorithm by using the already computed elevation data from former points. While his algorithm does not miss terrain heights like standard horizon mapping does, the algorithm is extremely slow and is not suited for real-time applications.

New methods for terrain shadowing which are built on horizon calculation are proposed more recently. The method of Snyder and Nowrouzezahrai in [11] calculates the horizon information in real-time for small sized height maps $(256^2, 512^2)$. The algorithm optimizes the running time of $O(N^3)$ for a single direction horizon extraction for all terrain points by using a multi resolution height map pyramid. This pyramid is created from the full detail height map by applying filtering to generate denser levels, like the mipmapping operation for regular color textures. Instead of sampling the height map with a constant sampling rate like in the standard horizon mapping,

they take coarser distanced samples from height maps with lower detail levels as a function of the distance from the current point. For shading the height map, they use fourth order spherical harmonics in order to model the occlusion, which is obtained from the horizon extraction step and to model incoming light. The spherical harmonics approximation for occlusion and lighting causes the algorithm to over soften shadows. They try to reduce this artifact by defining partial lighting swaths around receiver points, assuming light will only reach from these directions. Additionally, the multi-resolution pyramid approximation unavoidably causes the algorithm to miss shadows cast by high frequency hills in the distance, over the current receiver point. They report a speed of 7.05 Hz for a 1024^2 height map, which is only enlightened by key lights. This speed makes the algorithm only interactive for larger sized height maps like 1024^2.

Another height map shadowing method is *"Ambient Aperture Lighting"* which is proposed by Oat and Sander in [12]. In this method, surface and light source visibility is represented by spherical caps projected on the hemisphere which surrounds a terrain point. The light source is considered as a circular area source, which is the case of the Sun. The amount of illumination for a point is calculated with the intersected area between its visibility aperture on the hemisphere and the light source's circular projection on it. As a preprocessing step, for each terrain vertex they calculate the amount of total visibility on the hemisphere. The amount of visibility is scaled by the area of the unit hemisphere and the circular area of the visibility aperture is then projected on the unit hemisphere in the direction of the bent normal. Bent normal is defined as the average of the directions at which the terrain points receives light unblocked. In the rendering step, the spherical light source is project on the visibility hemisphere of the terrain point. The intersection area between the visibility aperture and light source projection gives the amount of illumination for this point. In general, the method is a clever solution to the height map soft shadowing problem. Its biggest shortcoming is its expensive preprocessing step. A change in the height map elevation data requires execution of the visibility determination process again.

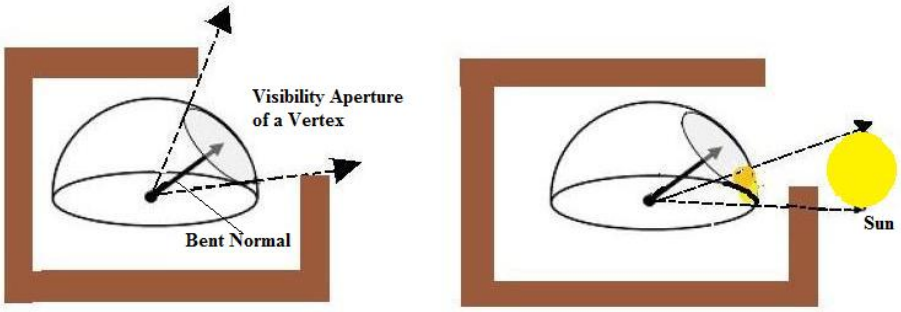


Figure 3.25. The circular visibility aperture on the unit hemisphere(left) and the projection of the light source on it(right). The intersection area between the circles gives amount of illumination. Image based on [12].

A method with the basic idea similar to Ambient Aperture Lighting is proposed by Chang *et al.* in [13]. In this method, the visibility of each height map vertex is represented by a cone instead of a circular aperture like in [12]. Similar to [12], the method consists of preprocessing and rendering steps. In the preprocessing step, for each height map vertex, 360 angles are sampled to provide the occlusion contribution of the surrounding geometry. The steepest visibility angle is then recorded for each direction, which is to horizon angle in the current direction. After that, 360 sample angles are divided into four groups with each group representing the average of the 90 horizon angles corresponding to it. The four averaged horizon angles are then recorded into a four channel texture to be used in the rendering step. In the rendering step, a spherical light source is projected on the height map's tangent space as a circle. The amount of illumination is determined from the intersection area between the light circle and quad-directional cones. In order to simplify the intersection calculation, the four partial cones are interpolated into a single cone with more weight given to light's direction. Like Ambient Aperture Lighting, the method has an expensive preprocessing step for visibility determination.

Timonen and Westerholm introduced another horizon mapping based height map shadowing method in [14]. Their most important contribution is a new horizon extraction algorithm, which uses the coherence between equally distant height map samples

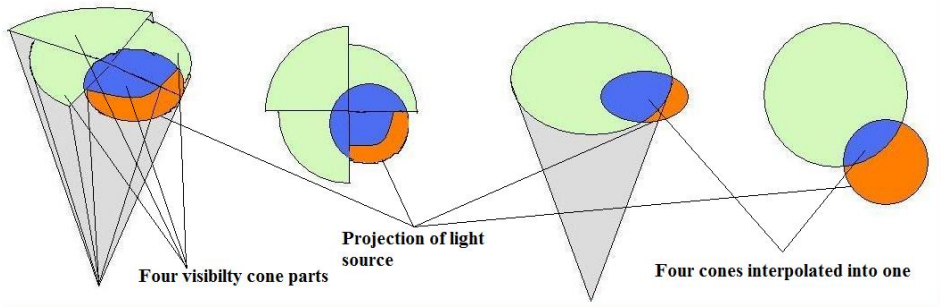


Figure 3.26. The quad-directional visibility cones are interpolated into a single cone for light visibility test. Image based on [13].

in a given direction and extracts horizon angles for these samples in $O(N)$ time where N is the total number of samples in the current sweep direction (see Figure 3.27). For a N^2 sized height map, the horizon extraction time for a single direction becomes $O(N^2)$ as the result of this linear time algorithm. Since every height map sweep line is independent from other lines, GPGPU technology can be utilized to extract horizon angles for many different directions in parallel. Timonen and Westerholm used NVIDIA's CUDA technology for this parallel execution goal. Their second contribution is a lighting model derived from the Rendering Equation of Kajiya [37]. They use a cube map to represent a complex lighting environment, which corresponds to the hemisphere around the height map with each pixel of the cube map shows whether light is coming from that specific direction or not. By projecting the normal of a height map point onto the azimuthal plane of a specific light direction sampled from the cube map, they calculate the contribution of that light direction as a function of the horizon angle and projection of the normal on the current azimuthal plane. Uniform ambient lighting can be readily calculated in an efficient way with this method since it is assumed to be reaching a point from all directions on its hemisphere. For a 1024^2 height map, they report a maximum performance of 76 Hz on a NVIDIA GTX 280 video card with 16 azimuthal sweep directions. The performance decreases proportionally with the number of sweep directions; for 64 directions the maximum speed becomes 24 Hz. To the best of our knowledge, this method is the first and only height map shadowing method which utilizes GPGPU technology to introduce parallelism and to boost performance.

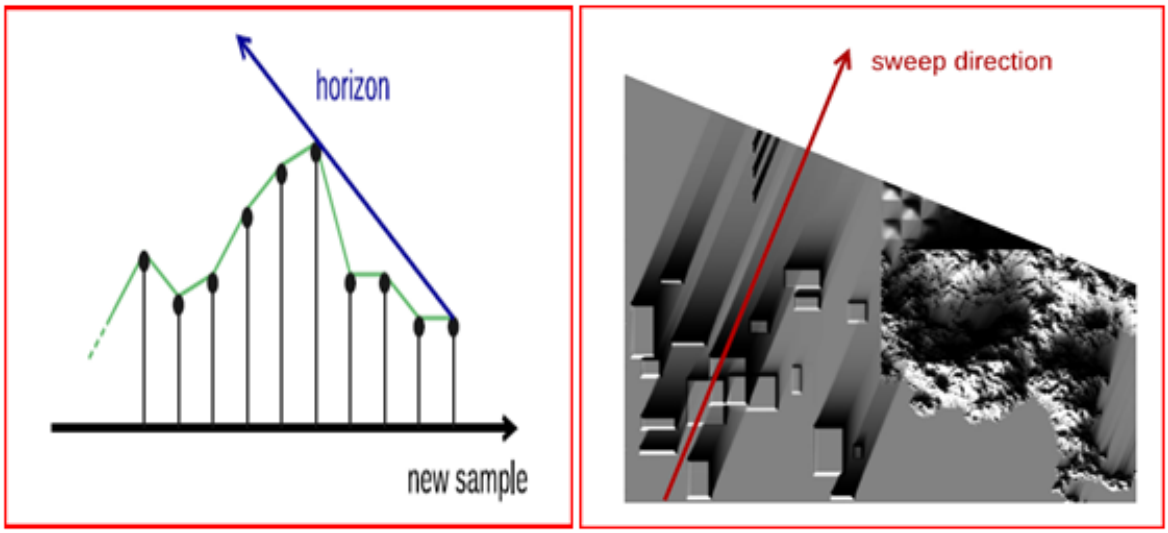


Figure 3.27. The linear horizon extraction algorithm outputs the horizon angle of a new sample in constant time, using the horizon information from older samples (left). A sweep direction on the height map can be seen on the right. Images taken from [14].

"Hybrid Terrain Shadow Ray Casting" is a height map shadowing method proposed by Sakalauskas in [81]. As its name implies, the method borrows elements from mainstream shadowing algorithms like shadow mapping, shadow volumes and ray tracing. Sakalauskas' method is based on the brute force approach of [82]. He observed that the amount of ray-triangle intersections of the brute force approach can be greatly reduced since only those height map positions, which constitute a silhouette when looked from the direction of the light, are significant for the shadow calculation. According to Sakalauskas, a height map point T^i is a silhouette point, if the height map segment between T^i and the previous height map sample T^{i-1} faces to light and the segment between T^i and the next height map sample T^{i+1} faces backwards from light. When casting a ray from an arbitrary point on the height map to the sun, processing only those silhouette points brings a huge performance improvement. Sakalauskas reports that on his sample data, among all the sampled height map positions, only %5.7 of them are actually silhouette points. In his algorithm he builds an image called as the SMap every frame which holds the silhouette positions according to the current light direction. Since SMap elements are based on the regular height map structure, logarithmic time search algorithms are used to detect each silhouette point on every SMap

column. After the preprocessing step, in which the SMap is generated, in the pixel shader stage, every height map pixel runs a ray casting operation by querying the SMap texture for blockages by silhouette points. Where the SMap is not discretized, texture linear interpolation can be safely used since the terrain is readily a structure which consists of linearly connected discrete height map points. Sakalauskas reports an average speed of 42.61 Hz on its 512^2 sized sample terrain data with the optimized version of the algorithm. The biggest advantage of this method is that it supports shadows cast by the height map onto other objects; it is not limited with self-shadowing like other height map shadowing methods explained to this point. Moreover, it provides alias free, pixel correct shadows. As a limitation, this method only produces binary hard shadows (shadowed or lit), which is not very realistic in the case of Sun illuminated outdoor scenes, since the Sun casts soft shadows with penumbrae. Additionally, the visibility test of this method is still based on ray casting in the pixel shader stage; if the geometry complexity on the screen increases with many additional objects, this could create a processing bottleneck caused by long pixel shader execution times. Lastly, the amount of %5.7 for silhouette points can vary from terrain to terrain (to a maximum of %50) making the algorithm susceptible to terrain elevation complexity. Anyhow, this method is a clever solution to the height map shadowing problem in general.

The last height map shadowing method we will examine is introduced by Tatarchuk in [15]. This method is actually devised for parallax mapping but it can be used with any height map based geometry. In this method, n samples, $(h_0 - h_n)$, are taken on the height map in the direction of the light. As the we trace a light ray from the starting sample, h_0, to the last sample, h_n, the height of the light ray is subtracted from the current sample's height. If the height map sample is larger than light ray's height at this point, this sample is occluding the initial sample, h_0. Instead of generating a binary hard shadow query, which will stop ray tracing at the first intersection with the height map, Tatarchuk calculates the occlusion contribution of each height map sample iteratively. If the current sample is above the light ray, then its height above the initial sample height, $h_i - h_0$, is found. By using this height difference, she computes the amount of occlusion contribution for each height sample, according to Equation 3.6. The occlusion contributions are scaled with the distance to the starting sample as in

the physical ground truth. Among all of the occlusion contributions, the maximum one is selected. We saw in Equation 3.5 that this is the lower bound for the occluder fusion problem.

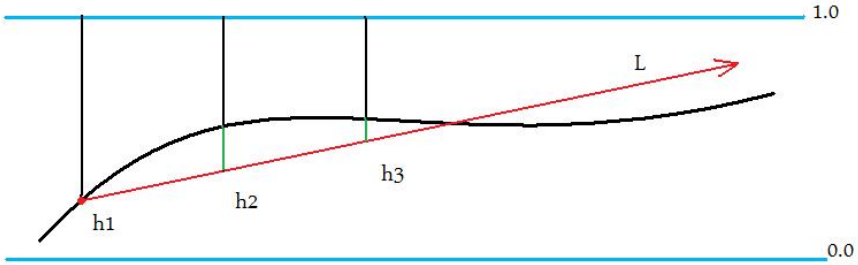


Figure 3.28. Height samples are taken along the light direction to calculate occlusion contributions of each sample in Tatarchuk's method. Image based on [15].

4. SHADOW HEIGHT MAPS FOR TERRAIN SHADOWING

4.1. Basic Idea of Shadow Height Maps and Summary of Our Method

Shadowing of height maps is a challenging task as we saw in the previous chapter. State-of-the-art versions of shadow mapping and shadow volumes algorithms run into numerous difficulties when used to shadow height map geometry. Specialized height map shadowing algorithms are devised to this end. They offer practical and good quality shadowing for large height maps but usually they depend on some kind of pre-processed visibility information, like horizon angles in different directions. Calculation of this visibility information is often very expensive as we saw in cases of brute force ray casting method [82] and horizon mapping based methods [83], [12, 13, 84, 85]. Methods like [11] accelerate this visibility determination process with optimizations like multi-resolution height map pyramids. Such optimizations decimate the height map data in order to generate lower resolution pyramid levels, which in turn causes the algorithm to miss shadows cast by high frequency details in the distance. Moreover, the algorithm still runs only at interactive speeds with this optimization. The method in [14] is utilizing GPGPU technology in order to calculate shadowing of the height map in parallel and it reaches better performance than [11] in determining light visibility. But its performance is dependent on the number of azimuthal sweep directions it takes into account. Among these methods, only the method in [81] does not have an expensive preprocessing step and it can be extended to support dynamic height maps. Another and a very serious shortcoming of these specialized methods is that they only support self-shadowing of the height map; this means they cannot produce shadows which the height map casts onto other objects, with the exception of the method in [81].

To address the shortcomings of these specialized shadowing algorithms, an alternative shadowing approach for height maps is used in this thesis. This approach is based on the use of so-called *"Shadow Height Maps"*. A shadow height map is an

enhanced version of the regular height map data. A height map describes a 3D surface with 2D data; each vertex of the 3D surface can be shown with $(x, H(x,z), z)$ where $H(x,z)$ is the discretized height map value. If $H(x,z)$ is not discretized at the current x and z coordinates, bilinear interpolation can be used to produce the height data; which makes the 3D surface a collection of bilinear surface patches. A shadow height map has a similar definition; it contains the height of the shadow which the height map casts. The 3D surface which is defined by a shadow height map can be shown with $(x, S(x,z), z)$ where $S(x,z)$ gives the height of the shadow on the current x and z coordinates. Bilinear interpolation can be safely used to reconstruct data where shadow height map is not discretized, since the information stored in a shadow height map is geometrically meaningful, as we will show in coming sections. Since it represents the shadow volume data of the height map in discrete locations, shadow height maps are sometimes named discretized shadow volumes [86]. Shadow height map of a height map can be generated both for point and directional light sources. The most important advantage of the shadow height map over other height map shadowing techniques are that it is not limited with the self-shadowing of the height map and it can naturally cast shadows onto other objects from the height map. Since the shadow height map contains the geometrical height of the shadow volume, every object in the same coordinate space with the shadow height map can query it for the height of the height map's shadow at the current x and z coordinates. This query mechanism is most suited for the pixel shader stage: If the pixel has its corresponding x,y and z world coordinates, the following simple procedure is applied to find out whether the pixel is shadowed:

$v = sgn(SHM(x_{pixel}, z_{pixel}) - y_{pixel})$;
if $v > 0$ **then**
  Shadow(Pixel);
else
  !Shadow(Pixel);
end if

Figure 4.1. The basic Shadow Height Map visibility test algorithm.

If the current pixel's height smaller than the shadow height sampled from the

shadow height map using the x and z coordinates, then the pixel is under the shadow boundary and is occluded by the height map. Such pixels receive only ambient lighting. If the pixel's height is larger than the shadow height, this means the pixel is not occluded by the height map and receives diffuse and specular components of the light. Shadow height maps are not limited with regular grid height map structures; in theory, a shadow height map of every surface can be generated, if the surface in question can be exactly defined or approximated with a 3D function of the form $f(x, z) = y$. Moreover, objects which are placed on the height map can be unified with the height map surface in order to generate shadows from them. For example, buildings on an open terrain can be sampled as if they were elements of the height map and their shadows are automatically added to the cumulative shadow volume of the terrain. The visibility querying mechanism does not differ between the pixels of the height map or pixels of any other arbitrary object and this is the most powerful aspect of the shadow height map method. Figure 4.2 illustrates a sample setting for this case.

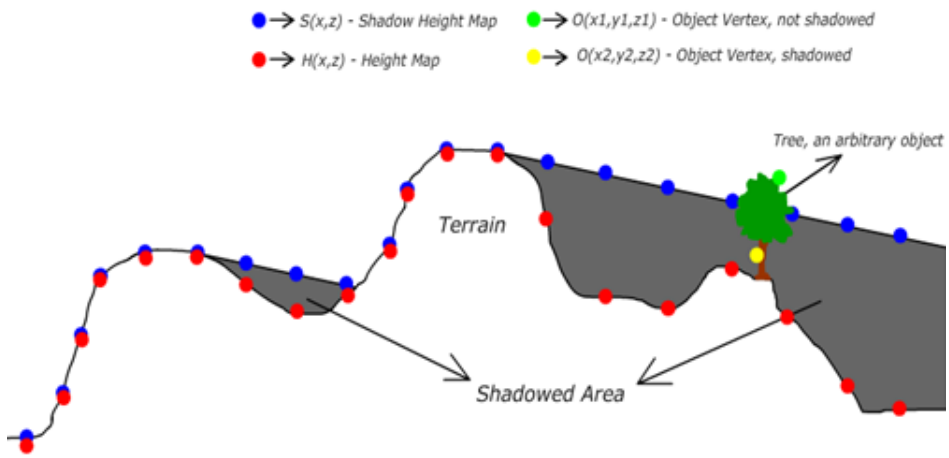


Figure 4.2. Shadow height map of a height map based terrain. Both the terrain and the tree receive shadows from the shadow height map, without distinction.

Another advantage of the shadow height map technique is that it does not need an expensive preprocessing step to extract visibility information. Generating the shadow height map as a texture can be done extremely fast for a given light direction. In our work, we show that it can be done in a similar fashion to the horizon extraction

technique of [14], with parallel light rays from a directional light source tracing the height map iteratively in itself and independently from each other.

In this thesis, we present a new height map shadowing method which is based on the concept of shadow height maps. It is a hybrid method which borrows elements from shadow mapping, shadow volumes and ray casting; by doing so, it shows similarities to Sakalauskas' hybrid terrain shadowing system in [81]. Our system supports static and dynamic, movable directional light sources, which is the common case of outdoor scenery illuminated by the Sun but it is possible to extend it for point light sources. There are only a few sources in the literature which use the shadow height map concept for shadowing; our method improves those in many aspects. First, we cast rays from an arbitrary sun direction onto the terrain. Each sun ray traverses a 2D slice of the height map and creates a discrete representation of the actual shadow volume of the height map on this 2D slice by using a simple iterative algorithm. The number of rays cast is parametric, proportional to the size of the height map terrain. (For example, 1024 rays for a 1024^2 terrain). For each 2D slice, we store the discrete shadow volume representation into a shadow height map texture, which is rotated such that each sun ray is stored into a single column of the image. In the pixel shader stage, we calculate the corresponding texture coordinates on the shadow height map and sample the height of the shadow for this point by a simple texture lookup. If the world y-coordinate of the current point is smaller than the sampled shadow height, it is shadowed by the terrain. If the texture coordinate is not among the discrete values on the SHM, we use simple texture bilinear interpolation. Since our lookup is not binary in its nature and geometrically meaningful instead, bilinear interpolation can be used with little computational cost. We execute ray casting in parallel on NVIDIA's CUDA framework and therefore, calculation of the shadow height map for a 1024^2 terrain is done extremely fast (under 1 ms for a single frame in our test system). For filtering hard shadow edges and eliminating any depth fighting errors, we define a constant penumbra height. This, by itself, provides a physically inaccurate but visually plausible soft shadow effect. Additionally, we propose a modified version of the constant penumbra algorithm, in which we add a geometrically calculated amount of unit penumbra height to the constant penumbra in each step of the iterative 2D shadow

volume extraction algorithm, as the distance from the last blocker increases. With this, we provide harder shadows near casters and softer shadows with larger penumbrae in the distance. We obtain this physically based soft shadow approximation by storing the penumbra height for each shadow height map sample along with the corresponding shadow height. In addition to terrain self shadowing; the objects on the terrain can receive shadows from terrain by sampling the shadow height map. Our approach does not support shadows cast by other objects onto the height map or on themselves. To include this, our method should be combined with a shadow map or shadow volumes based method. Using dynamical, movable light sources introduced temporal aliasing errors due to changing shadow height map sampling locations. We were able to reduce that to acceptable levels by applying shadow interpolation between larger light direction intervals and averaging shadow heights of several successive frames.

In this section, we made a basic introduction to the shadow height map concept and the basic idea behind it and gave a summary of our height map shadowing approach which is based on shadow height maps. In the coming sections of this chapter, we will firstly give a literature survey on previous work which utilized shadow height maps. Then we will describe our proposed method in detail: How is it implemented, its comparison with state-of-the-art algorithms, problems and artifacts we run into and our solutions for them.

4.2. Literature Survey: Shadow Height Maps in Previous Works

Although not a common approach, the idea of building a shadow height map using the original height map data is used before in some studies. In our literature survey, we were able to find a few examples which directly or indirectly use shadow height maps in order to address the height map shadowing problem.

According to our survey, in [16], Loscos *et al.* use the term "shadow height map" for the first time. Their target application is a virtual city simulation with thousands of human impostors with a crowd animation. They have a large city setting which has the characteristics of a 2.5D height map, meaning the "floor", the buildings and all

other additional features can be described as a function of the form $f(x, z) = y$. They have observed that the shadows, which the city elements such as buildings cast, can be represented with a second height map, which they call the "shadow height map" of the city. Their definition of the shadow height map is the same with ours: A discretized shadow volume representation in the image domain. Dynamic elements, like human impostors, can receive shadows from the buildings by converting their world coordinates into texture coordinates and sampling the shadow height map in appropriate locations. The work was accomplished in 2001, a date in which programmable graphics pipeline functions were not yet introduced and therefore they implemented the shadowing system with the fixed pipeline multi-texturing features of the OpenGL API. They render human impostors, some of which are completely shadowed, blended with a gray texture and the ones which are fully lit blended with a white texture. If an impostor is partially shadowed, then they calculate the amount of the shadowed and lit parts and execute the multi-texturing according to this lit/shadowed ratio. Since no per-pixel calculations are possible in the lack of pixel shaders, the exact shape of the shadow boundary cannot be represented on the impostor. They assume that they have a directional light source, which is static during the run-time of the application. Moreover, the city is assumed to be static, too. As a result of the 2.5D nature of shadow height maps, they cannot represent city elements like roofs overhanging the edge of the buildings, bridges and other "non-2.5D" objects. Such elements cannot be represented with a function of the form $f(x, z) = y$. This is a limitation which all shadow height map based methods share. Their shadow height map approach supports binary hard shadows only. Loscos *et al.* create the shadow height map with a hybrid approach which is a mixture of projection shadows and shadow volumes; [39], [41]. They render a large plane under the city. Then they place the camera viewpoint over the model and with an orthogonal projection, they render each model edge with vertices projected on the floor plane, offscreen. For each model edge this will produce a shadow plane. They store the z-buffer (depth buffer) of this scene as a separate image and finally they subtract the original height map from this image. The resulting image will contain 0's in areas which do not receive shadows and the actual height of the shadows over the ground, otherwise. This approach is applicable where the total silhouette edges are

in reasonable numbers. In a general fractal or real world elevation (DEM) data based height map, detecting such edges would be extremely slow since there will be hundreds of thousands of triangles to check for silhouette extraction.

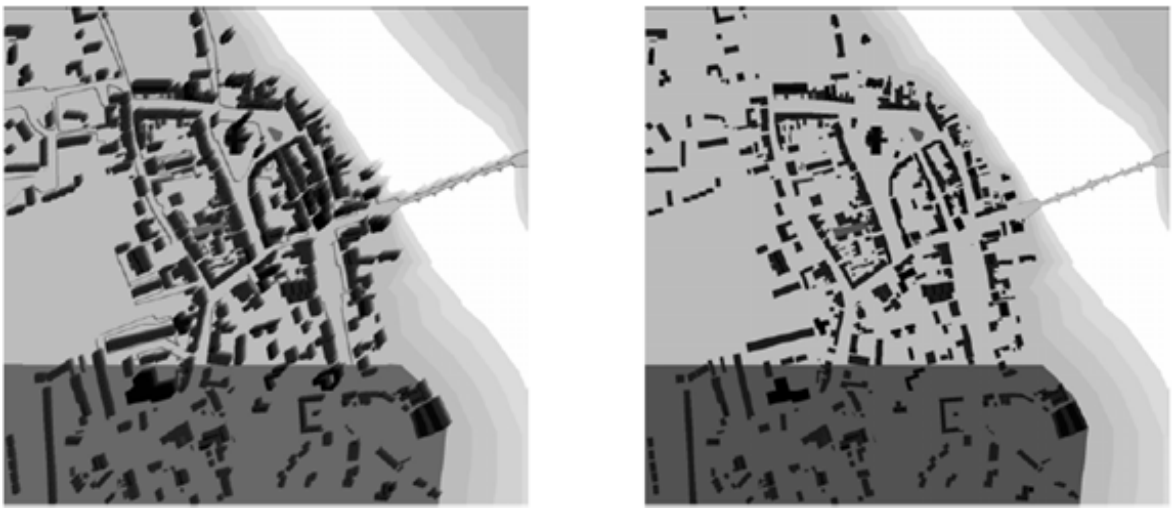


Figure 4.3. On the left, the discretized shadow volume, i.e. the shadow height map is shown. The original height map is shown on the right. Image taken from [16].

McGuire and Sibley used a shadow height map based terrain shadowing solution in [86]. They are the first to derive a recursive definition for the 2D discrete shadow volume extraction process. They call shadow height maps as "discretized shadow volumes" for the height map data. In their setting, the Sun is a directional light source at infinity. As they load chunks of terrain, the shadow volume extraction algorithm is run on the new height map data and shadow height information is stored on each vertex. The shadow height data on the vertices is interpolated on the polygons in the rasterization stage. Similar to our method, they define a penumbra height, in which it is assumed that pixels see the Sun only partially. The visibility test of Figure 4.1 is executed in pixel shaders. They allow the Sun to be dynamic but its movement is restricted only to a rotation around the z-axis; this means the incoming sun light vector is only a function of the horizon angle θ, which the light vector makes with the x-axis: $L(\theta) = \langle \cos\theta, sin\theta, 0 \rangle$. This is not a precise simulation of the Sun movement; such an exact East-West rotation is only valid on the equator at solstices. According to

our findings, this method is the most similar one to our method in the literature. Our shadow height map based solution does not impose a limitation on the Sun movement; all types of rotations and arcs are valid in our system. Moreover, our system implements the shadow extraction algorithm completely on the GPU in parallel, instead of on CPU. Additionally, we propose an extension to the constant penumbra height approach, where we increase the penumbra height as the distance from the last blocker increases.

Lee and Kim presents a height map shadowing algorithm based on the use of shadow height maps [17]. For every directional light source, they generate a different shadow height map. They do not restrict the light direction like McGuire and Sibley does in [86]; according to their method, there are two types of shadows. Type 1 shadows are directly aligned with x or z axes. Type 2 shadows are coming from arbitrary directions; they are not aligned with any of the main axes. Extraction of Type 1 shadows are based on a recurrence relation which very similar to the one used in [86];. Since Type 2 shadows are passing through locations where the height map is not discretized, use of linear interpolation is needed. They devised the following relation for the generation of such shadows, which holds for a light direction $L = \langle l_x, l_y, l_z \rangle$, where $0 < l_x < l_z$ and s is a scaling constant, indicating the size of a height map cell:

$$
\begin{aligned}
S(0,0) =& H(0,0) \\
S(0,z) =& H(0,z) \\
S(x,0) =& H(x,0) \\
S(x,z) =& max(H(x,z), S(x - \frac{l_x}{l_z}, z-1) - s\frac{l_y}{l_z})
\end{aligned}
$$

There are eight such formulas for eight possible l_x and l_z relations, each covering a 45° portion of the xz-plane. For approximate soft shadows, they define a constant penumbra height like in [86];. Their biggest advantage over [86];, is the capability of providing arbitrary light directions with Type 2 shadows. But their extraction method

for this type of shadows is sequential: Each shadow height map element $S(x, z)$ depends on the previously processed element. This makes the algorithm $O(N^2)$ iterations with no parallelization possibility; the extraction formula cannot benefit from the GPGPU technology. An additional disadvantage is the memory transfer overhead. Since they execute the shadow height map extraction algorithm on the CPU, the resulting image data must be transferred to GPU every frame in order to be accessed by the graphics pipeline. The bus traffic between CPU and GPU is a common bottleneck in computer graphics applications; since the bus bandwidth is limited and the transfer speed is relatively slow, applications need to avoid CPU-GPU memory transfers as possible for high performance. Lee and Kim indicate that this is their main source of performance degradation. Anyhow, this work is the most precise and detailed paper on the concept of shadow height maps.

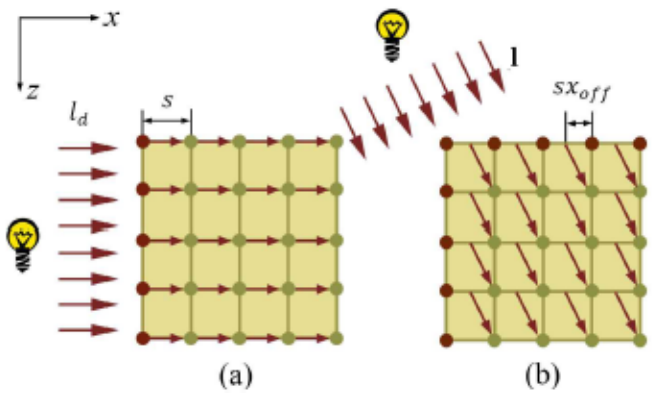


Figure 4.4. Type 1 shadows which are exactly aligned with x or z axes (a) and Type 2 shadows which have an arbitrary direction (b). Image taken from [17].

The concept of discretized shadow volumes in the form of shadow height maps is not used only in the area of computer graphics. In [87], Ratti and Richens use a similar shadow extraction algorithm to compute daylight shadows in the field of city and environmental planning. From the digital elevation model (DEM) of a surface, they generate the shadow volume of the height map data as an image processing macro. They assume z-axis as the up vector. The sun light direction vector is scaled such that the bigger of the x and y axes becomes 1 pixel of length. The z axis is adjusted according

to the image calibration. They translate the image by the x and y components and in the same time they subtract z component from the image height. They continue to translate by this vector and take the maximum of the current shadow volume with the previously calculated, until all height levels reach 0 or the shadow volume is shifted right off the image. Although this is implemented only as an image processing macro, the basic idea of building a discretized shadow volume is relevant to our work.

The last work we will mention belongs to Robertson [88]. The algorithm does not actually involve the idea of shadow height maps, but the way he calculates shadows for a given surface data is similar to our proposed method. He observed that for a single light source, if the processed image is transformed in a way such that any arbitrary point can only be occluded by other points in front of it, the visibility determination problem for the whole surface will be reduced to processing of independent scan lines (rows or columns) of the image and therefore can be executed extremely rapidly. He pointed out that this transformation also allows parallel processing of the visibility determination problem since the scan lines are independent from each other. His algorithm consists of five steps. In the first step the image is rotated such that the light direction becomes aligned with the columns of the image. A point light source is used and in order to make the light beams emanating from this source parallel to each other, a 1D rhombic transformation is applied to the base image. The effect of this transformation is to convert the point light source into a directional one. This makes the second step of the algorithm. In the third step, each scan line is processed. He projects the current pixel's height onto the light plane. If the current pixel's projection is bigger than all other previous pixels', this pixel is not shadowed and vice versa. The biggest projection is stored for comparison with newer pixels. It should be noted that this iterative process is similar to the ones used in [86] and [17] but instead of calculating a shadow height, it only stores binary information of "shadowed" or "lit" for the current pixel. The last two steps include the inverse rhombic transformation, which effectively decompresses the surface and the rotation of the surface back to its original direction, in this order. The original algorithm generates hard shadows for surface data but Robertson proposes another version which calculates soft shadows by storing values between 0 and 1 instead of only 0 and 1 but he does not give any implementation details in his paper.

Our shadow height maps based height map shadowing solution has similar features with all of the above methods but it provides numerous advantages compared to them. We, too, like Robertson in [88], rotate our height map data such that it becomes vertically aligned with the sun light direction. For each column, we cast a separate sun ray and determine the visibility information with a simple iterative algorithm, which lasts $O(N)$ iterations where N is the number of sampling locations (pixels) on the current column. We use GPGPU technology to execute this operation in parallel for each ray. Since today's GPUs can process many threads in parallel, visibility determination for the whole height is completed extremely fast, such that its speed becomes comparable to shadow mapping; the fastest known shadowing solution. To the best of our knowledge, our method is the only one to utilize GPGPU technology to implement height map shadowing in parallel, with the exception of the method in [14] and the first one to do height map shadowing with the use of shadow height maps and GPGPU acceleration. Our algorithm supports dynamic directional light sources; we do not restrict sun movement around a fixed axis like [86]. Moreover, it does not have a preprocessing step and therefore it can be adapted easily to support dynamic height maps, like [81]. Since we use GPGPU technology, our whole shadow height map generation process takes place on the graphics card (GPU) and we can directly write our data into a texture image which is already allocated on the video card memory. By doing this, we avoid being stalled by the high cost CPU-GPU memory transfers, a performance degradation source from which [17] suffers. Our visibility determination for individual scene pixels is executed on custom programmed pixel shaders. This allows us to generate visually plausible and physically approximated penumbra effects, which the method in [16] cannot achieve. In summary, our shadow height map based approach is more advanced from older similar examples in the literature in many aspects.

4.3. The Rendering Pipeline and The GPGPU Technology

In our work we make extensive use of graphics hardware related technologies like rendering libraries and the GPGPU infrastructure. It is necessary to use one of the rendering APIs to be able to communicate with the graphics hardware. Addition-

ally, we need another framework to utilize the GPGPU infrastructure which provides necessary abstractions. Exact definitions and functionality of such frameworks differ between each other but since the implementation of our algorithm heavily depends on them, we deem it necessary to give a basic explanation of these technologies.

4.3.1. The Rendering Pipeline and Shader Technology

For graphics applications, it is very important to be able to communicate with the graphics hardware. There are various standardized collections of libraries which provide this communication with a hardware abstraction layer (HAL). Most prominent among these are the OpenGL of the Khronos Group [89] and the Direct3D of Microsoft [18]. In our work, we use the 9.0c version of the Direct3D library.

The process of rendering graphics to a display involves various steps of calculations which begin with loading of the initial geometry into the video memory and end with rasterization of the final color data to the pixels on the screen. This sequence of different calculations is called as the *"Graphics Rendering Pipeline"*. Each graphics library implements its own version of the rendering pipeline. Figure 4.5 shows the rendering pipeline of Direct3D 9.0c.

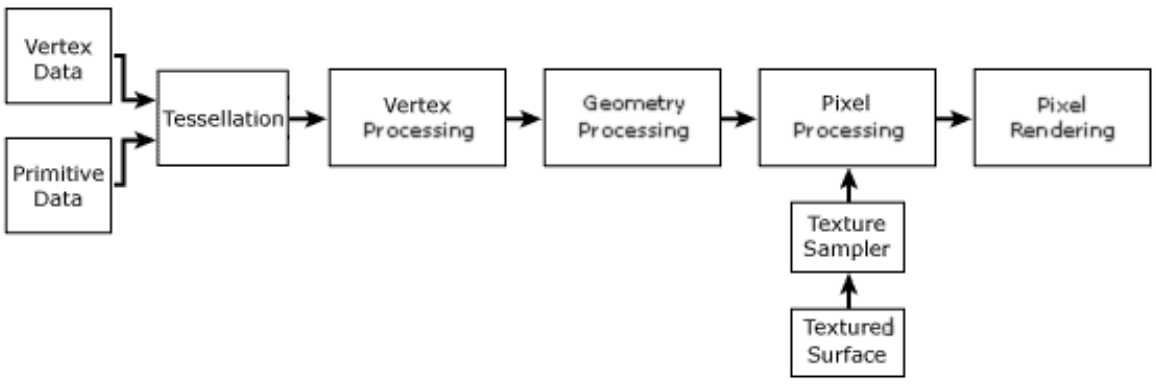


Figure 4.5. The Direct3D 9 rendering pipeline. Image taken from [18].

The "Vertex Data" and "Primitive Data" blocks in Figure 4.5 represent the input

data feed into the rendering pipeline. The "Vertex Data" block contains the untransformed model vertices and "Primitive Data" contains index data, which is used by the rest of the pipeline to index the "Vertex Data". The "Tessellation" unit converts higher order primitives like quads into triangle lists, which is the smallest, atomic polygon used in the pipeline. In "Vertex Processing" stage, various coordinate transformations are applied to incoming vertex data. Those include world, view and perspective projection transformations. This stage is applied independently to each vertex. Current state-of-the-art graphics libraries like Direct3D and OpenGL allow developers to write their custom transformation programs, which are applied to each single vertex. These programs are called as the *"Vertex Shaders"*. A vertex shader program takes a single vertex as its input. At the very least, a single vertex has to contain its untransformed world coordinates for world-view and projection transformations. In addition to that, it may contain texture (UV) coordinates, normal data, color data or any arbitrary user-defined data, as long as it is declared as a vertex format declaration via the graphics library before. In order to write vertex shaders, Microsoft devised the *High Level Shader Language (HLSL)*, which has a C++ like syntax and is specialized for programming graphics hardware. With a vertex shader program, we are able to modify all the embedded data on the processed vertex according to our needs. A vertex shader program outputs the modified vertex data to be processed by the later Geometry and Pixel Processing units. The data which it outputs is fed into the following "Geometry Processing" stage. In this stage space, the viewport transform, clipping of the polygons, which is rendered outside of the screen borders, back face culling and attribute evaluations are applied to the transformed vertices. The most important process of this stage is rasterization. After the projection of vertices on the 2D screen, the data which the transformed vertices contain is interpolated along the scan lines of 2D triangles. This process is called as the rasterization step. In the newer versions of the Direct3D library (10 and 11) the geometry processing stage is also programmable with the use of *"Geometry Shaders"*. In our work, we did not need those. The "Pixel Processing" step is another programmable stage of the rendering pipeline. The programs used to this end are called *"Pixel Shaders"*. Like vertex shaders, in Direct3D, pixel shader programs are written with HLSL. Pixel shaders take the interpolated data from the

rasterization stage as the input. Texture mapping through texture samplers are done in pixel shaders; all effects related to per pixel lighting, texturing, coloring, shadowing, etc. are realized in this stage. Pixel shaders output a single color value as their result. The final stage, "Pixel Rendering", is responsible with applying alpha, stencil and depth testing. Moreover, here, alpha blending and the fog effects can be applied to the final pixel color values, too. The resulting pixels, which pass all culling tests and which go through alpha and fog blending, if enabled, are presented to the frame buffer and drawn onto the display.

The above is only an abridged and simplified version of the whole Direct3D rendering pipeline. A more detailed discussion of this technology is out of the scope of this text. For a good source on Direct3D programming, we refer the reader to the book of Luna F. D. [33].

4.3.2. The GPGPU Technology

The term GPGPU stands for *"General Purpose Computing on Graphics Processing Units"*. Due to the nature of the graphics processing, video cards contain many small, equivalent chips, which are suited to execute similar operations in parallel. For example, vertex shaders and pixel shaders work on numerous vertices and pixels, which only differ in their embedded data. Therefore, a GPU is a highly parallel, multithreaded, many core processor with tremendous computational capability and very high memory bandwidth. A GPU offers a SIMD (Single Instruction Multiple Data) environment and it is ideal for the execution of tasks which operate on identical structures with only the input data varying among them. In theory, such tasks are not limited with the processing of vertices and pixels in the area of computer graphics; many different problems can be formulated in a way such that the SIMD principle of a GPU can be applied to solve them in parallel. GPGPU is the step taken in this direction, the purpose of this technology is to provide developers with a framework, such that they can benefit from the multithreaded-many core architecture of GPUs in a standardized way.

The *Compute Unified Device Architecture (CUDA)* is NVIDIA's framework for the GPGPU technology [19]. Each GPU which supports the CUDA framework has several multiprocessors and each multiprocessor has multiple cores. In CUDA, the basic unit of processing is a thread. Threads are created within blocks and grids can be launched with multiple blocks. The launching mechanism can correspond to a 1D, 2D or 3D structure, according to the needs of the problem at hand. The basic piece of the program, which all CUDA cores execute in parallel, is called as a *kernel*. Each CUDA core has its own ALU (Arithmetic Logic Unit) but there is only a single Control Unit per multiprocessor. This structure is the result of GPU's SIMD architecture: It is assumed that the processing cores will run the same piece of program on different data, with minimum conditional branching. Therefore, handling of conditional data is the main weakness of the SIMD architecture. Threads in the same block are processed in groups of 32, which are called as warps. Threads in the same warp execute in lockstep, this means that they must process the same line of code in a given clock cycle. If the threads in the same warp follow different conditional paths, all execution paths have to be serialized and this degrades the performance immensely. For this reason, care should be taken when implementing an algorithm for any GPGPU architecture, such that the number of conditional branches is minimized. Compared to a general CPU architecture, the GPU has much more raw processing capability. A CPU has a large portion of its transistors dedicated to complex control logic circuitry, which the GPU architecture lacks. For example, GPUs do not have any branch prediction capabilities like CPUs. But since its raw processing power is so high, GPU architecture tends to complete data independent tasks with few conditional statements in a much shorter time than a CPU.

CUDA has various memory types with different features. *Global memory* is the slowest (off-chip) memory. But it is very large and accesses to it can be accelerated if threads in the same *half warp* (two 16 thread sized halves of a warp) access sequential addresses on it. This is called as a *coalesced access* and it is advantageous over randomly scattered access because memory transfers for sequential address blocks can be combined into a single request in this way. Any thread in any block or grid can access global memory without any limitations. In latest GPUs with CUDA compute

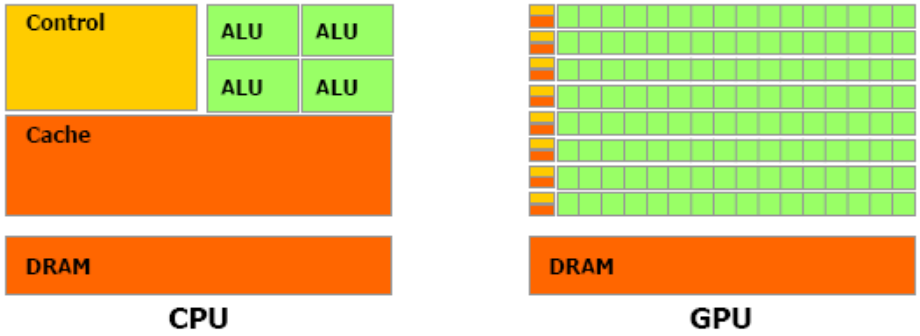


Figure 4.6. CPU architecture compared with GPU architecture. Note the immense amount of ALU circuitry in the GPU. Image taken from [19].

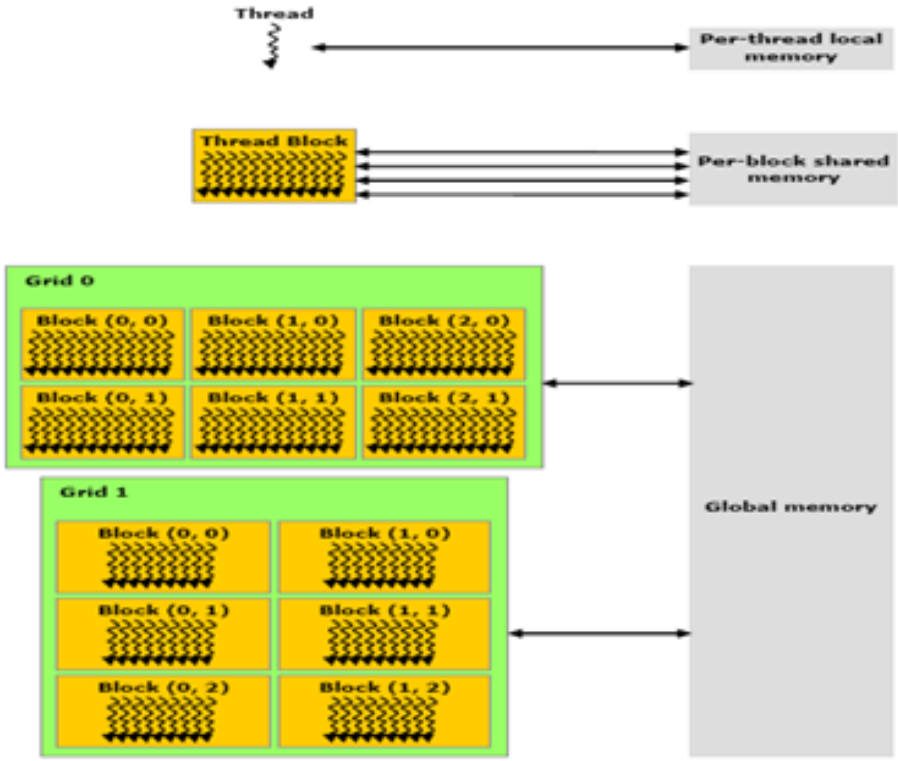


Figure 4.7. Different types of memories in CUDA and how they correspond to the thread hiearchy. Image taken from [19].

capability 2.0 or higher, the global memory is cached; an important gain for performance. The second memory type is the *constant memory*. It is a very limited (usually 64 KB) memory but it is on-chip and cached. Reads from it can be done very fast. As

its name implies, it is a read-only memory. It is loaded with data by application-side calls to the CUDA runtime; device-side kernel code cannot modify data stored in the constant memory. *Shared memory* is a per multiprocessor (per block) memory. It is used as a high speed cache for the threads in the same block. It provides communication between the threads in a block. It should be used with care in order to avoid race conditions and deadlocks. *Texture memory* is the last type of memories which CUDA provides. It represents the equivalent of the texture sampling mechanism found in the rendering pipeline; it actually uses the same underlying hardware [19]. It presents a 2D spatial caching mechanism. If threads tend to read from close spatial areas on the texture, the access speed will be increased tremendously. Moreover, since it uses the same texture sampling circuitry with the graphics rendering pipeline, it provides hardware accelerated bilinear interpolation when reading from locations not discretized on the texture. Texture memory is read-only like the constant memory. Figure 4.7 shows the relationship between different types of memories and the thread, block and grid organization in CUDA.

We use CUDA to execute the ray casting step in our algorithm in parallel. Since our ray casting scheme consists of identical, data independent tasks it can be perfectly mapped to thread hierarchy present in CUDA. For more detailed information on the CUDA architecture, we refer the reader to NVIDIA's CUDA C Programming Guide [19].

4.4. The Terrain Shadowing Algorithm: The Basic Case for a Static Directional Light

We start describing our algorithm with the basic case of a static directional light source. We assume that the y-axis is our up vector and the sunlight can reach our terrain from any arbitrary direction. Our algorithm for static light sources consists of five different steps:

In the following subsections, we will explain each step in detail. Since we use the height map data directly to generate our terrain mesh, we use the terms of height map

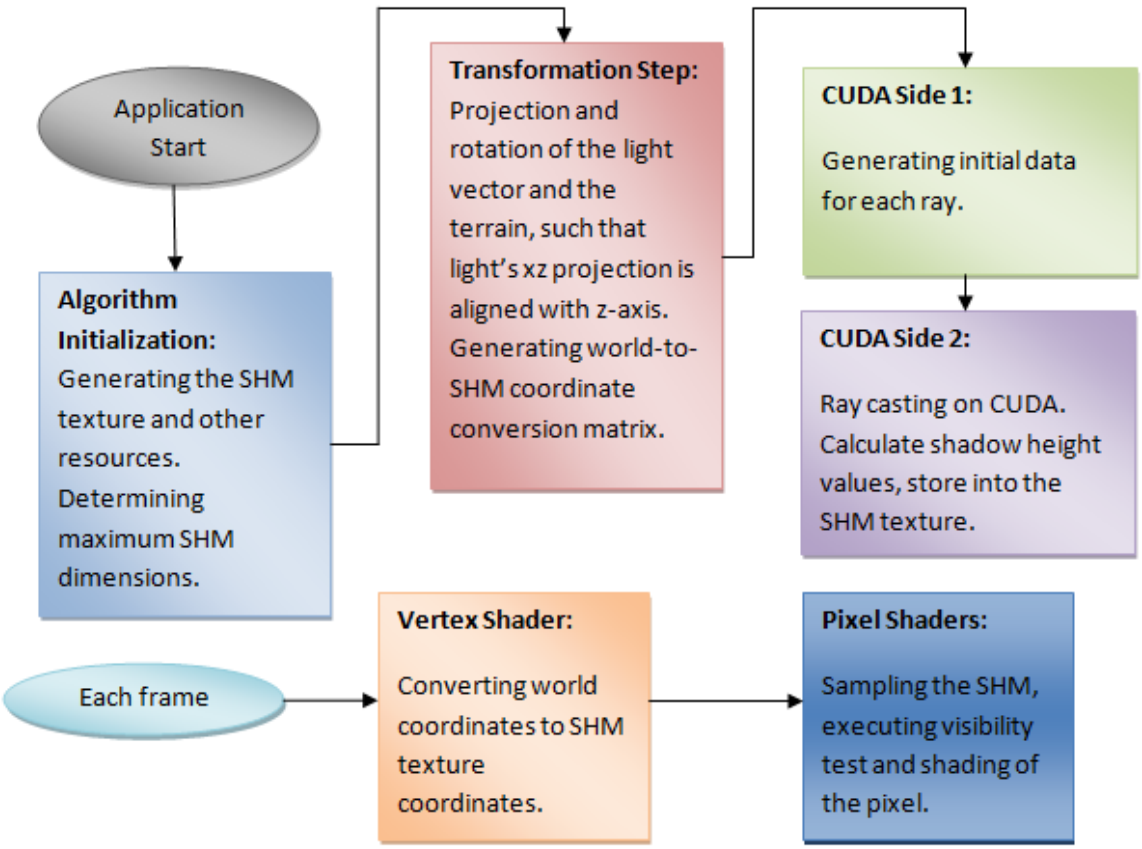


Figure 4.8. The flow diagram of our algorithm, when used with a static light source.

and terrain interchangeably in the rest of the chapter.

4.4.1. Algorithm Initialization

This initial step is always executed once, independent of whether the light source is static or dynamic. Here, we determine the dimensions of the shadow height map (SHM) texture and allocate a Direct3D texture resource to contain it. The width of the SHM texture in pixels is equal to the maximum number of sun rays cast onto the terrain. We adjust the number of rays with a variable which represents the ray/height map resolution ratio. If this ratio, R, is equal to 1, this means that N rays will be cast for a terrain of the size N^2 if the sun light vector's xz-plane projection coincides exactly with x or z axes. Moreover, we can imagine the terrain as an image with the dimensions of R^2N^2 with this ratio. As mentioned before, we rotate our system, such

that the sunlight's projection on the xz-plane will exactly coincide with the negative z-axis and so we can process sun rays as vertical scan lines on the SHM texture. Since the sun light may have an arbitrary direction, after this rotation, the maximum number of rays and the SHM texture dimensions are functions of the sun light direction's x and z coordinates. In order to cover all possible sun directions, we need to find the biggest possible SHM dimensions which can be assumed after the rotation. The sun light's projection on the xz-plane makes an angle with the negative z-axis, which we call α (Figure 4.9). Finding the maximum dimension is easy, if we can express it as a function of α. We will show how this angle is related to the sunlight's x and z coordinates in the next section. In general, if we have a rectangle shaped height map with dimensions of $W * H$, after the rotation, the bounding rectangle of the height map will have the following dimensions (Figure 4.9):

$$\begin{aligned} W(\alpha) &= H\cos\alpha + W\sin\alpha \\ H(\alpha) &= H\sin\alpha + W\cos\alpha \end{aligned}$$

For finding the biggest possible values of $W(\alpha)$ and $H(\alpha)$, we take the first derivatives of both and solve for α's, which make the derivatives zero:

$$\begin{aligned} \frac{dW}{d\alpha} &= -H\sin\alpha + W\cos\alpha \\ 0 &= -H\sin\alpha_1 + W\cos\alpha_1 \\ \tan\alpha_1 &= \frac{W}{H} \\ \alpha_1 &= \tan^{-1}\frac{W}{H} \end{aligned} \qquad \begin{aligned} \frac{dH}{d\alpha} &= H\cos\alpha - W\sin\alpha \\ 0 &= H\cos\alpha_2 - W\sin\alpha_2 \\ \tan\alpha_2 &= \frac{H}{W} \\ \alpha_2 &= \tan^{-1}\frac{H}{W} \end{aligned}$$

The above equations show us that maximum width and height of the new height map bounding box is $W(\tan^{-1}\frac{W}{H})$ and $H(\tan^{-1}\frac{H}{W})$, respectively. In fact, these two

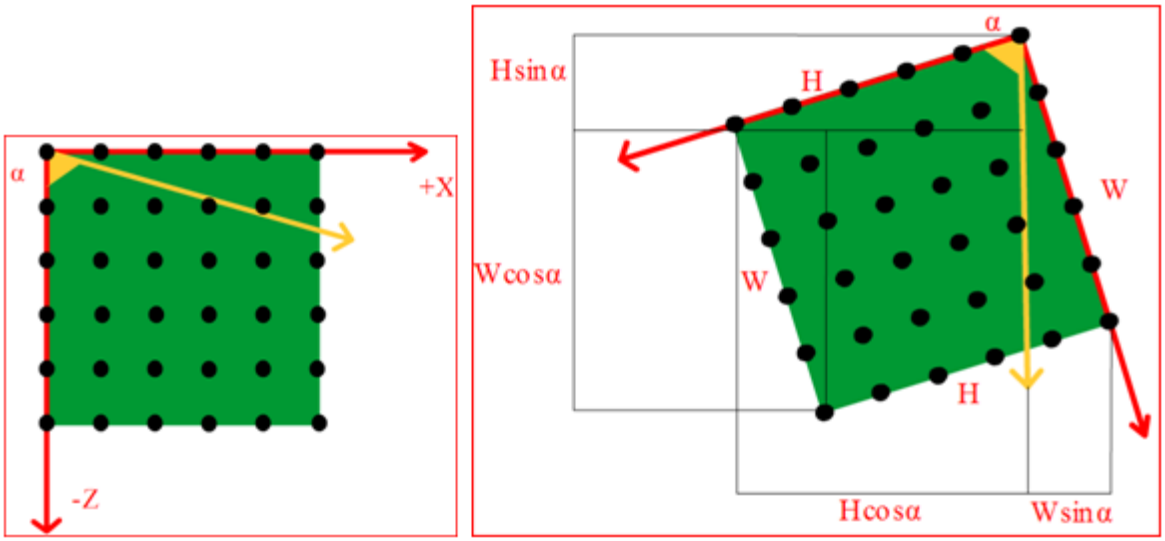


Figure 4.9. On the left, the sun light's projection on xz-plane makes the angle α with negative z-axis. On the right, the terrain is rotated, such that the sun light exactly coincides with negative z-axis.

equations are equal to each other. Since $\tan^{-1}\frac{H}{W} = 90^o - \tan^{-1}\frac{W}{H}$, we can express $H(\tan^{-1}\frac{H}{W})$ as the following:

$$\begin{aligned} H(90^o - \alpha_1) &= H\sin(90^o - \alpha_1) + W\cos(90^o - \alpha_1) \\ H(90^o - \alpha_1) &= H\cos\alpha_1 + W\sin\alpha_1 \\ H(\alpha_2) &= W(\alpha_1) \end{aligned}$$

So the dimensions of the shadow height map texture and the maximum number of sun rays can be found with the following formula:

$$M = \lceil R(W\cos(\tan^{-1}\frac{W}{H}) + H\sin(\tan^{-1}\frac{W}{H}))\rceil + 1$$

If the height map is a square with the dimensions of N^2, the formula becomes:

$$M = \lceil \sqrt{2}RN \rceil + 1$$

Following these calculations, we create a Direct3D texture with M^2 of size. This means that the maximum number of sun rays we will cast is also M. We will generate a 32-bit floating point texture to write shadow height values into. In the later parts of this chapter, we will show a compression method with which we are able to store SHM textures using 16-bit unsigned integer textures. In addition to the texture, we generate input buffers from which CUDA kernels for ray casting read their initial values. Some of these buffers will be copied into the constant memory. There are two additional, important operations we do in this initialization step. The first one is registering the Direct3D texture with the CUDA. This is required for CUDA-Direct3D interoperability; we use this feature in order to write into the Direct3D textures from ray casting kernels. According to the CUDA C Programming Guide, registering Direct3D resources with CUDA is an expensive operation which we should avoid from doing on a per-frame basis [19]. We only do this registration at the program start.

The last operation we execute in the initialization step is to transfer the original height map data to a CUDA texture, which is referenced by a CUDA array. This texture will be sampled by the ray casting CUDA kernels in the later steps. The bilinear interpolation hardware and 2D spatial texture cache make the texture memory the ideal place to store the height map data in.

4.4.2. The Transformation Step and CUDA Side 1

In this step, we first rotate the terrain to make the sun light vector aligned with the negative z-axis. We need to find the relation between the angle α and sunlight direction vector's x and z axes to do this rotation. Let the vector $\overrightarrow{L_s} = \langle x_s, y_s, z_s \rangle$ be

the sunlight's normalized direction vector, with its y component, y_s, being negative (stemming from the fact that the Sun is always at the top of the landscape). We calculate this vector's normalized projection on the xz-plane as the following:

$$\overrightarrow{L_{xz}} = \frac{1}{\sqrt{x_s^2 + z_s^2}} \langle x_s, 0, z_s \rangle = \langle x_p, 0, z_p \rangle$$

Following this, the sine and cosine of the angle α can be calculated as:

$$\sin(\alpha) = \begin{cases} \sqrt{1 - z_p^2}, & \text{if } x_p > 0, z_p \leq 0 \\ \sqrt{1 - z_p^2}, & \text{if } x_p \geq 0, z_p > 0 \\ -\sqrt{1 - z_p^2}, & \text{if } x_p < 0, z_p \geq 0 \\ -\sqrt{1 - z_p^2}, & \text{if } x_p \leq 0, z_p < 0 \end{cases}$$

$$\cos(\alpha) = -z_p$$

We can build a simple 2D rotation matrix, after we calculate $\sin(\alpha)$ and $\cos(\alpha)$. By using this matrix, we rotate the height map in its world coordinate system. This rotation will align the sun light direction with the negative z-axis and will greatly simplify our shadow height (visibility) calculations in CUDA kernels. Simplified kernels will contain shorter code with lesser complexity which increases CUDA-side performance. This approach is similar to the scan line method of Robertson [88].

We do not apply the 2D rotation matrix to all height map vertices; just rotating the corners of the terrain is enough for our purpose. The four rotated corners of

the rectangular height map data will constitute four rotated edges. From the rotated world coordinates of the terrain corners we will calculate 2D line equations of each terrain edge in the form of $f(x) = Ax + B$. We use these line equations for the edges to determine the entry and exit points for each sun ray in the CUDA kernels. We assumed in the initialization step that the height map can be represented with an image of the size R^2N^2 and from this size we calculated the dimension of the SHM texture, which is M^2. After rotating the height map, we further assume that the rotated "image" of the height map is placed in the SHM texture, such that their middle points exactly coincide. At this stage, each SHM column corresponds to a sun ray and each SHM texel corresponds to a sampling point, at which we calculate the shadow height. In practice, this means that we are shooting M equally distanced, parallel sun rays onto the terrain and we take shadow height samples in equal distances along these rays. For each SHM column, we need the intersection points with the rotated terrain rectangle. Edge equations of the rotated terrain rectangle are used for this purpose. The 2D distance between each SHM column (sun ray) and each SHM sample location corresponds to a world coordinate system distance, D_{world}, which is calculated with the equation $D_{world} = \frac{C}{R}$, in which C is the cell length; the distance between each height map vertex in world coordinates and R is the ray/height map resolution ratio as we defined before. Due to the rotation of the terrain, some of the SHM columns will not intersect with the terrain rectangle. We can save resources if we avoid such columns and only process the ones which actually intersect with the terrain. If we assume in the world coordinate system that the square SHM texture covers an area of the size $M^2D^2_{world}$ and its center point exactly coincides with the world origin $\langle 0, 0, 0\rangle$, we can calculate the column indices of the first and last SHM columns, which intersect with the terrain, with the following formulas:

$$C_{start} = \lceil \frac{X_{min}}{D_{world}} + \frac{1}{2}(M + 1) \rceil$$

$$C_{finish} = \lceil \frac{X_{max}}{D_{world}} + \frac{1}{2}(M - 1) \rceil$$

In the equations above, X_{min} and X_{max} represent the minimum and maximum x-coordinates of the rotated terrain rectangle, in the world coordinate system. We will only launch CUDA kernels for the SHM columns in the range $[C_{start}, C_{finish}]$.

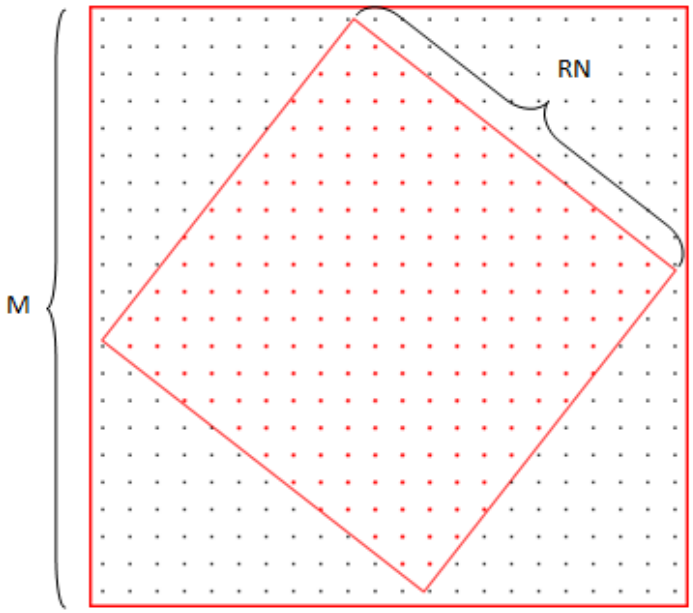


Figure 4.10. The SHM texture with the size of M^2 and the height map placed in the center of it, with the size of R^2N^2. Each SHM column represents a sun ray. The red dots indicate locations of shadow height samples.

Another important piece of data calculated in this step is the difference vector $\overrightarrow{L_D} = \langle \Delta x, \Delta y, \Delta z \rangle$. We define it as the vector which gives the x and z coordinate differences between two successive texels on the same SHM column, in the height map texture coordinate space. We use these in the ray casting kernel, in order to sample landscape heights from the height map texture we uploaded into CUDA in the initialization step. The vector also gives the amount of height, Δy, which a sun ray loses when traveling from a SHM texel to the next one. These three components are calculated as the following:

$$\Delta x = \frac{x_p D_{world}}{C(W+1)}$$

$$\Delta y = \frac{y_s D_{world}}{x_s^2 + z_s^2}$$

$$\Delta z = -\frac{z_p D_{world}}{C(H+1)}$$

In the equations above W and H are height map width and height in cell counts. Since D_{world} corresponds to the 2D world space distance on the xz-plane between two SHM texels, we scale the unit height loss of the sun light, y_s, with $\frac{D_{world}}{\sqrt{x_s^2+z_s^2}}$ in order to find the actual height loss between two SHM texels in world space.

Later in the pixel shader stage of each object, for each pixel, we will need to sample the SHM texture in the appropriate location in order to get the corresponding shadow height for the current pixel. We need to obtain the texture coordinates of this sampling location from the world coordinates of the pixel in question. To this end, we generate a world coordinate-to-SHM texture coordinate conversion matrix. From the x and z world coordinates, we first generate the 2D height map texture coordinates. Then we transform the height map(HM) coordinates into SHM coordinates by taking the rotation and translation of the height map image on the SHM texture, into account. The formulation of the corresponding transformation matrix is given in the next page.

$$M_0 = \begin{bmatrix} \frac{1}{WC} & 0 & 0 \\ 0 & -\frac{1}{HC} & 0 \\ -\frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix} (World - to - HM\ space\ conversion)$$

$$M_1 = \begin{bmatrix} \frac{WR}{M} & 0 & 0 \\ 0 & \frac{HR}{M} & 0 \\ -\frac{WR}{2M} & -\frac{HR}{2M} & 1 \end{bmatrix} (Translation\ to\ origin\ and\ scaling)$$

$$M_2 = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} (Sunlight\ z - axis\ alignment\ rotation)$$

$$M_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix} (Translation\ from\ origin)$$

$$M = M_0 M_1 M_2 M_3 = \begin{bmatrix} \frac{R\cos\alpha}{CM} & \frac{R\sin\alpha}{CM} & 0 \\ \frac{R\sin\alpha}{CM} & -\frac{R\cos\alpha}{CM} & 0 \\ \frac{1}{2} - \frac{WR\cos\alpha}{M} & \frac{1}{2} - \frac{WR\sin\alpha}{M} & 1 \end{bmatrix} (Final\ conversion\ matrix)$$

$$[x_{SHM}, z_{SHM}, 1] = [x_{World}, z_{World}, 1] \begin{bmatrix} \frac{R\cos\alpha}{CM} & \frac{R\sin\alpha}{CM} & 0 \\ \frac{R\sin\alpha}{CM} & -\frac{R\cos\alpha}{CM} & 0 \\ \frac{1}{2} - \frac{WR\cos\alpha}{M} & \frac{1}{2} - \frac{WR\sin\alpha}{M} & 1 \end{bmatrix} \quad (4.1)$$

Equation 4.1 shows generation of SHM texture coordinates from the world x

and z coordinates. After all the transformations and necessary calculations have been finished, we run the first CUDA kernel in our algorithm flow. Each ray we will cast in the next step needs some initial values to trigger the shadow volume extraction algorithm. There are six such initial values we need to calculate:

- $H_{initial}$ - The initial height: The ray casting algorithm starts with this height first. It is the height sampled just at the 2D intersection point (x_i, z_i) where the current SHM column enters the height map rectangle. Line equations of the rotated edges are used to calculate this value.

- (x_0, z_0) - First sampling point: The 2D coordinates of the first height map sampling location, in the height map texture coordinate space.

- Δy_0 - The initial height difference: The height difference between the intersection point (x_i, z_i) and the first sampling point (x_0, z_0). It is calculated as:

$$\Delta y_0 = \Delta y \frac{\sqrt{(x_0 - x_i)^2 + (z_0 - z_i)^2}}{D_{world}}$$

- (T^c_{Start}, T^c_{End}) – Column start and end indices: These show the index of the first and last texels on the current column c, at which the shadow heights for this column are calculated for the first and last time. The shadow volume extraction CUDA thread for the column c will process and generate meaningful values for the texels in the interval $[T^c_{Start}, T^c_{End}]$.

Since the calculation of these initial values can be independently done for each column, it is a data independent process and therefore it can be parallelized. A serial CPU implementation of this step tends to work very slow and thus we decided to parallelize this step in CUDA, as well. We decided to implement a separate CUDA kernel for the calculation of these initial values with a temporay global memory buffer.

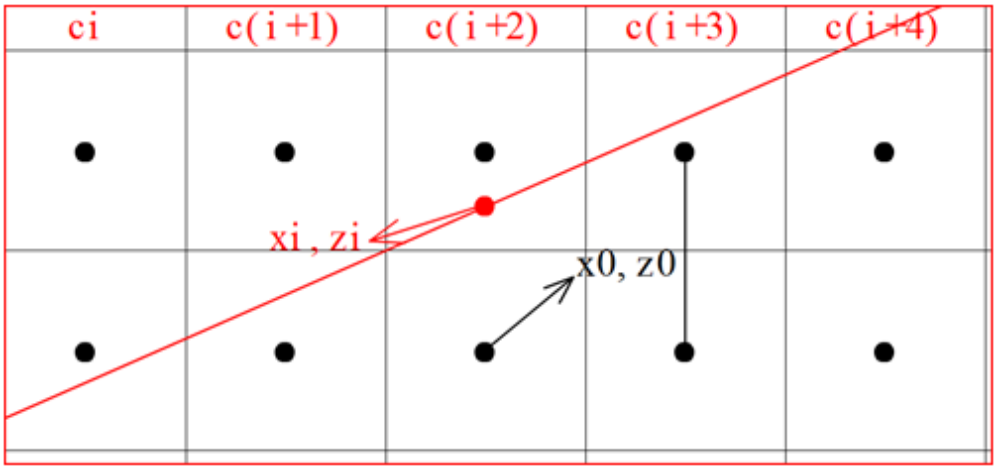


Figure 4.11. Initial values of the columns $[C_i, C_{i+4}]$. C_{i+2} enters the terrain at (x_i, z_i).

The parallelization is done on per-column basis: There is a single CUDA thread for each SHM column. Each thread calculates the above data by executing 2D geometry intersection tests between the current column and the height map rectangle. The line equations of the height map edges and other needed data for this calculation are loaded into the CUDA constant memory before launching the kernel. Each thread writes its results into a CUDA global memory array, into the appropriate location, which is calculated by using the thread and block IDs of the current running thread. The later shadow volume extraction threads will read these values from exactly same locations since both CUDA kernels will use exactly the same thread hierarchy.

4.4.3. CUDA Side 2: The 2D Shadow Volume Extraction Algorithm

This part is where we actually populate SHM with useful data. This CUDA kernel invocation lies at the core of our whole algorithm. Our thread hierarchy is the same with the previous CUDA launch: We will launch the kernel for the columns in the interval $[C_{start}, C_{finish}]$. Each single CUDA thread is responsible for generating data for its corresponding SHM column. Like in the initial data generation kernel, we load the necessary data like the maximum SHM dimension, the difference vector $\overrightarrow{L_D}$ and the maximum ray count $C_{finish} - C_{start} + 1$, into the constant memory before the actual kernel launch. We use the height map texture we loaded into the texture

memory to sample terrain heights in each SHM texel location. We set its sampling filter to bilinear; bilinear interpolation will provide a good reconstruction method to obtain data where the height map texture is not discretized. This allows us to shoot sun rays from any arbitrary direction. The approach of loading the height map into CUDA's texture memory and sampling it with bilinear interpolation is used in [14], too. Another advantage of the texture memory is the hardware supported address modes. There are different address modes which determine how the texture sampling hardware should behave if the sampling location is outside of the original texture borders (repeating, wrapping, mirroring, etc.). This automatically handles memory access violation cases and simplifies the kernel code. For a total ray count of $T = C_{finish} - C_{start} + 1$ and for a maximum thread-per-block count of B for the current system, $\lceil T/B \rceil + 1$ thread blocks are generated at the kernel launch. For a height map size of 1024^2 and ray/height map resolution ratio of 1 there will be 1449 rays to be shot. If the maximum thread-per-block count is 512, there will be 3 CUDA blocks assigned to do the job.

The shadow volume extraction algorithm we use is actually an inductive recurrence relation. Its ground logic is very simple. For the current SHM column, c, we set the current shadow height value, R^c, equal to $H^c_{initial}$ at the start. $H^c_{initial}$ is the height at the intersection point of the current SHM column with the height map and it was generated by the previous CUDA kernel launch. Again at the start, the next height value is sampled at the texture coordinates (x_0, z_0) as $H^c_0 = HM(x_0, z_0)$ where HM is the height map texture. R^c is our propagated shadow height; we update it in each step such that it is always equal to the maximum possible shadow height. In the end of each inductive step, we save the most recent value of R^c into the current texel in the output buffer. In general, at the kth step of the inductive process, we check whether the current height map sample, H^c_k is higher than the next sample, H^c_{k+1}, if we subtract the sun light height difference, Δy, from it. If it is $H^c_k - \Delta y \geq H^c_{k+1}$, this means that H^c_k can cast shadow on H^c_{k+1}, since the height map segment between H^c_k and H^c_{k+1} is back-facing from the Sun. But this condition, by itself, does not mean that H^c_k actually casts shadow on H^c_{k+1}. H^c_k can be already in shadow, if the propagated shadow height, R^c, is higher than it. So, if both $H^c_k > R^c_k$ (the value of R^c, at kth step) and $H^c_k - \Delta y \geq H^c_{k+1}$ are true, then we say that H^c_k is a *"silhouette point"* which ac-

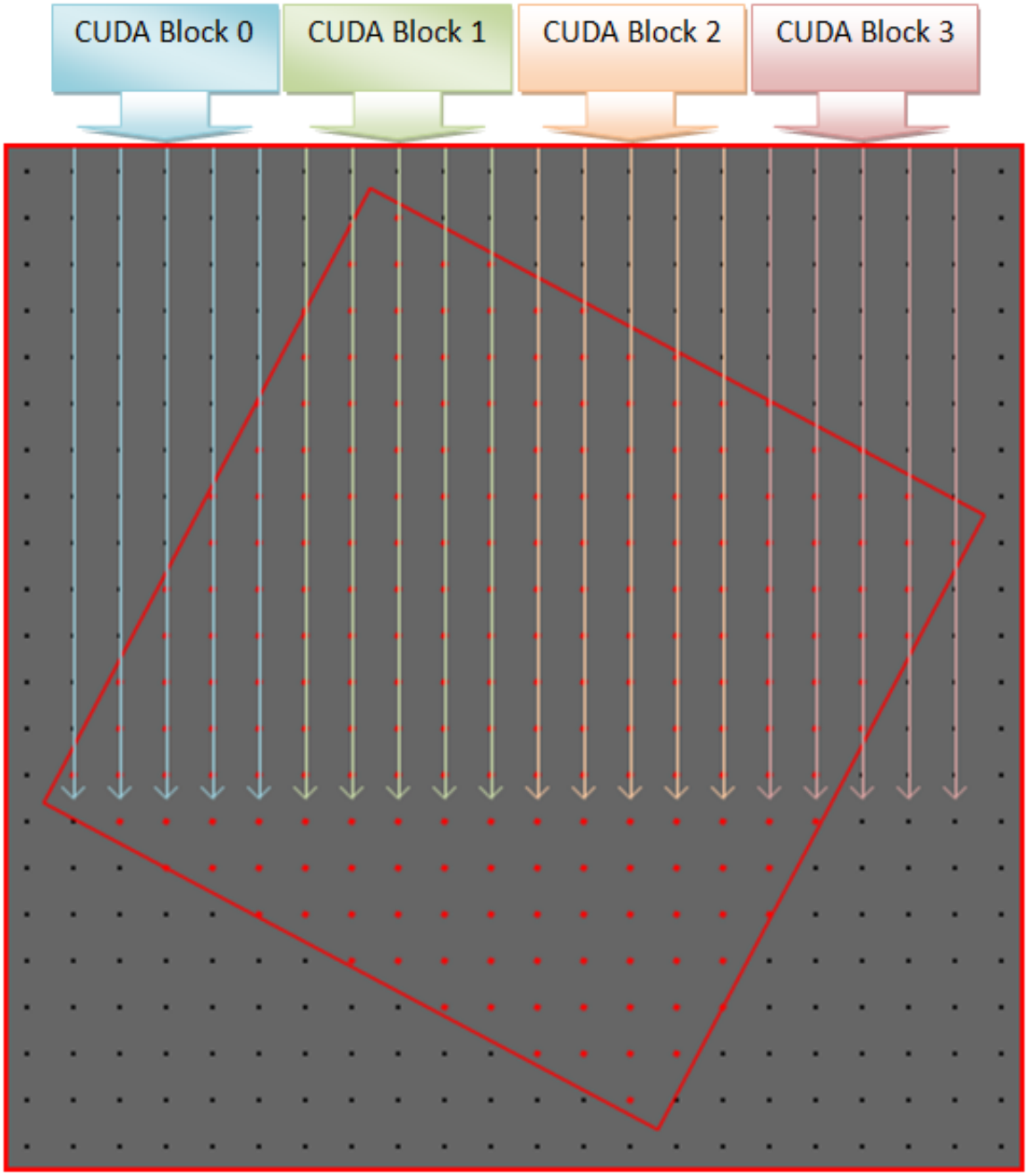


Figure 4.12. In this sample figure, 20 CUDA threads for 20 SHM columns have been launched. Assuming an imaginary system where the maximum thread/block number is 5, 4 different CUDA thread blocks are used in the kernel launch.

tually casts shadow on the height map points next to it. At such silhouette points, we set R^c_{k+1}, the next value for the current shadow height, equal to $H^c_k - \Delta y$ and thereby updating it effectively to the maximum possible shadow height. If it is $H^c_k - \Delta y < H^c_{k+1}$

in the current step, then we simply update R_k^c as $R_{k+1}^c = R_k^c - \Delta y$, since the height map segment between H_k^c and H_{k+1}^c is front-facing to the Sun. If we iterate this inductive operation from the texel T_{Start}^c to T_{End}^c, we will generate the maximum shadow heights for all texels in the interval $[T_{Start}^c, T_{End}^c]$ in the SHM column, c. The mathematical formulation of this inductive process can be written as the following:

$$R_0^c = H_{initial}^c$$

$$R_{k+1}^c = \begin{cases} max(H_k^c, R_k^c) - \Delta y & \text{if } H_k^c - \Delta y \geq H_{k+1}^c \\ R_k^c - \Delta y & \text{if } H_k^c - \Delta y < H_{k+1}^c \end{cases}$$

Figure 4.13 illustrates this process on an imaginary 2D height map slice:

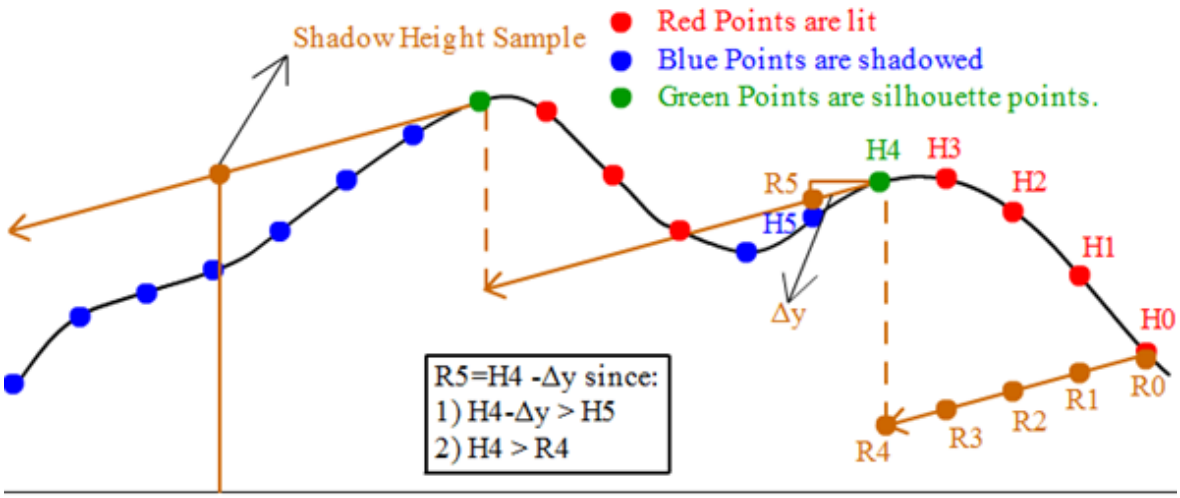


Figure 4.13. Shadow volume extraction on a 2D height map slice. Notice how the current shadow height jumps to assume the heights of the silhouette points.

The shadow volume extraction algorithm on a single column is an $O(N)$ algorithm, where N is the height map dimension. Extracting the shadow heights for all columns is an $O(N^2)$ process, but since we are processing each column with a CUDA

thread in parallel, the performance impact caused by that is greatly hidden.

We need to copy the results of the shadow volume extraction kernel into Direct3D textures in order to use them with the rendering pipeline. CUDA provides two separate ways to modify a Direct3D texture; we either allocate a linear global memory array into which the CUDA kernels will write their shadow height data and after the kernel invocation finishes, we copy the data into the target texture by using the CUDA array reference for the texture, as a distinct step (It should be noted that regular texture memory is read-only). As with the arrival of the CUDA version 4.0, a new memory type, called as the surface memory, has been introduced [19]. Surface memory actually uses the same hardware infrastructure with the texture memory but as a new feature, it allows the programmer to write directly into the texture memory, removing the read-only restriction. In our implementation, we tried both approaches. We found out that a ray casting kernel which writes into a global memory array performs faster than an identical kernel, which writes into the Direct3D texture directly via the surface memory. The reason behind this difference is the coalesced access optimization of the global memory. But after the global memory kernel invocation finishes, we additionally need to copy the global memory array into the Direct3D texture with a "cudaMemcpy" call, as an extra step. This memory transfer introduces a significant latency, especially for larger height maps. When combined with the cost of this memory transfer, the global memory kernel becomes slower than the one which uses the surface memory. Moreover, by utilizing the surface memory, we don't need to allocate a separate global memory array which we only use as a temporary buffer, thereby saving from general video memory. Therefore, we prefer directly writing into the surface memory over the global memory in our implementation.

In general, this process is executed very quickly with CUDA parallelization. For a 1024^2 sized height map, the completion time of the SHM generation kernel is around 1-1.5 ms. It should be noted that the shadow volume extraction algorithm is totally independent from the complexity of the height map. It does not differ between a low and high frequency height map, since its only connection with the height map data is a single bilinear interpolated texture lookup. The most similar approach to our algorithm

in the literature is Timonen and Westerholm's horizon extraction method [14].

4.4.4. SHM in the Rendering Pipeline: Vertex and Pixel Shaders

After we populated the SHM texture with appropriate data in the CUDA side, we need to feed it into the Direct3D rendering pipeline. The vertex shaders for the SHM do simple operations. We copy the world-to-SHM space conversion matrix of the Equation 4.1 into the vertex shader. For any arbitrary mesh (height map chunk or any other arbitrary object), we multiply the vertex coordinates with this matrix and obtain SHM sampling coordinates. The vertex shaders output those coordinates into the rasterizer stage and for each triangle on the screen, they are interpolated across the pixels of the current triangle. The pixel shader takes these interpolated SHM coordinates as input. Here, they are fed into the visibility determination functions in order to sample the SHM in appropriate locations. The visibility determination logic will be detailed in the next section.

4.5. Visibility Determination and Soft Shadow Simulation

In our method, we are easily able to generate soft shadows, by using two approaches. Our first approach uses a physically inaccurate but visually plausible soft shadow approximation. The second approach uses a simple geometrical model to calculate soft shadow penumbrae which takes the light source size, distance and the distance from the last blocker into account. In the two subsections below, we will explain these in more detail.

4.5.1. The Constant Penumbra Height Approach

In this approach, we define a constant, p, which we call as the *penumbra height*. This provides a cheap soft shadow filter. In the pixel shader stage, we sample a shadow height from the SHM. If the sampled shadow height is larger than the world y-coordinate of the current pixel but their difference is smaller than p, then we assume that the current pixel is partially shadowed, in other words, it is in the penumbra.

If the height difference is bigger than p, we say that it is completely shadowed; it is in the umbra. In order to calculate the occlusion amount of a pixel, we can use any continuous occlusion function $O(x)$ of the form:

$$O(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 0 \leq x \leq 1 & \text{if } 0 < x < p \\ 1 & \text{if } x \geq p \end{cases}$$

Simply subtracting the current pixel height from the sampled shadow height and scaling the difference by the penumbra height p gives a crude but useful occlusion estimation. If we use this as the occlusion function, the visibility determination for a pixel becomes the following for the constant penumbra approach:

$$O = min\left(1, max\left(0, \frac{SHM\left(x_{shm}, z_{shm}\right) - y_{world}}{p}\right)\right)$$

We can parametrically set p to different values to provide larger and smaller penumbrae. This constant penumbra height acts both as a filter for aliasing and depth fighting errors and as a simulation for soft landscape shadows. We experimentally determined that a penumbra height equal or close to the height map world space cell length, C , provides a visually good soft shadow simulation.

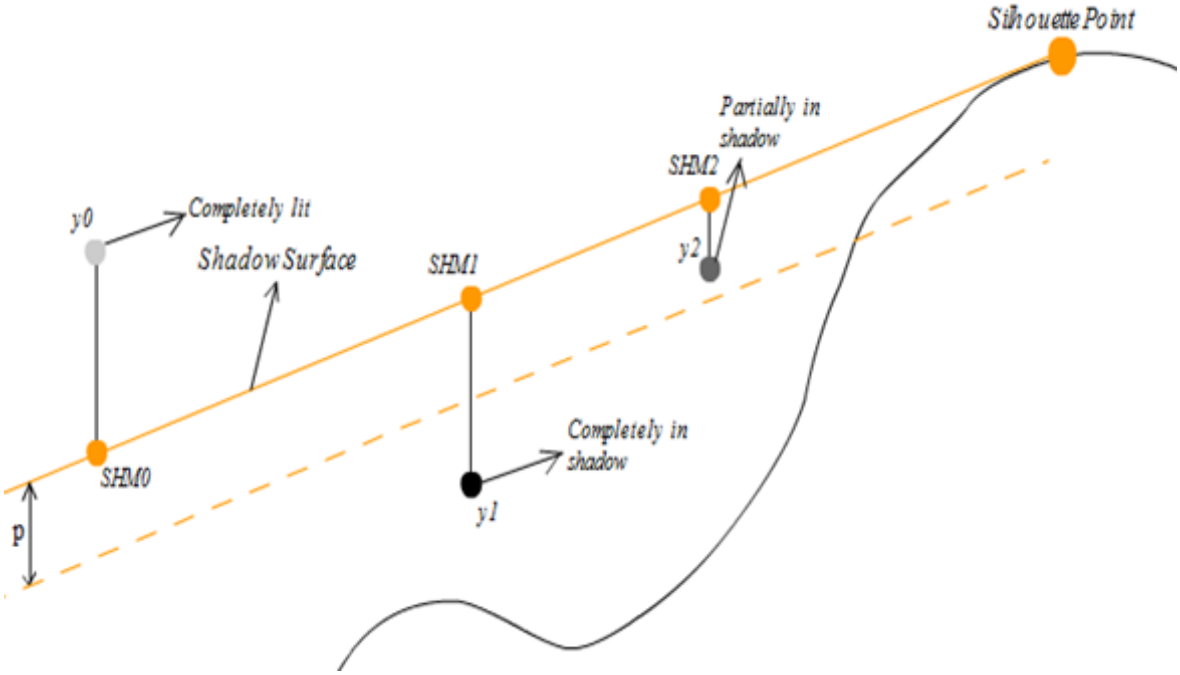


Figure 4.14. Visibility determination with constant penumbra height p. y_0 is completely lit since $y_0 > SHM_0$. y_1 is completely in shadow since $SHM_1 - y_1 > p$. y_2 is partially shadowed because $p > SHM_2 - y_2 > 0$.

4.5.2. The Varying Penumbra Height Approach

We devised a second soft shadow approach, which is based on a geometrical model and generates varying penumbra lengths instead of a globally constant one. In reality, the height of the penumbra depends on various parameters like the size and shape of the light source, its distance from the last shadow caster and the shadow caster's distance from the receiver. Moreover, shadows from other casters will mix in, leading to occluder fusion problem. In this approach we approximate the area light source as a linear one for each 2D height map slice. We define the angle β as the solid angle which the light source makes with the shadow caster. β is a function of the light source-caster distance and light source size such that it changes from caster to caster. Referring to Figure 4.15, if we assume that the line segment $\overline{BC}$ makes a right angle with $\overline{CA}$ and $|CA| = |DC|$, then it is $\tan \frac{\beta}{2} = \frac{|CA|}{|BC|}$. The distance $|BC|$ is a function of shadow caster's location and therefore β is not constant over the height map in reality.

But we focus on the case of the Sun illumination on landscapes and Sun's distance from the earth is $1.5 * 10^{11}$ meters while its radius is $7.0 * 10^8$ meters of length. Let's assume that we have two points, B and B', with B being exactly $1.5 * 10^{11}$ meters and B' $1.5 * 10^{11} + 1.0 * 10^5$ meters distant from the center of the Sun. (B and B' are approximately $\frac{100}{\cos\alpha}$ kilometers away from each other). The solid angles, β and β', are given as:

$$\begin{aligned}\beta =& 2\tan^{-1}\left(\frac{|CA|}{|BC|}\right) = 2\tan^{-1}\left(\frac{7.0 * 10^8}{1.5 * 10^{11}}\right) \approx 0.534757^o \\ \beta' =& 2\tan^{-1}\left(\frac{|CA|}{|B'C|}\right) = 2\tan^{-1}\left(\frac{7.0 * 10^8}{1.5 * 10^{11} + 1.0 * 10^5}\right) \approx 0.534756^o\end{aligned}$$

The Sun is so distant from the earth that its solid angle with arbitrary points on the earth surface barely changes as we can see in the calculations above. Therefore, we can safely assume β as constant for our height map data. We define the angle α as the elevation angle of the sun. With these definitions, on a 2D slice, we can calculate the penumbra height at the distance D from the last blocker as:

$$\begin{aligned}P =& D\left(\tan\left(\alpha + \beta\right) - \tan\left(\beta\right)\right) \\ P_{unit} =& D_{world}\left(\tan\left(\alpha + \beta\right) - \tan\left(\beta\right)\right)\end{aligned}$$

P_{unit} is the unit penumbra height, similar to the unit height difference Δy. We implemented a separate CUDA shadow volume extraction kernel for the varying penumbra height approach. In this kernel, we iterate the current penumbra height in a similar fashion and together with the current shadow height. We use 64-bit floating point textures for this type of soft shadowing; we store the shadow and penumbra heights into

two 32-bit channels.

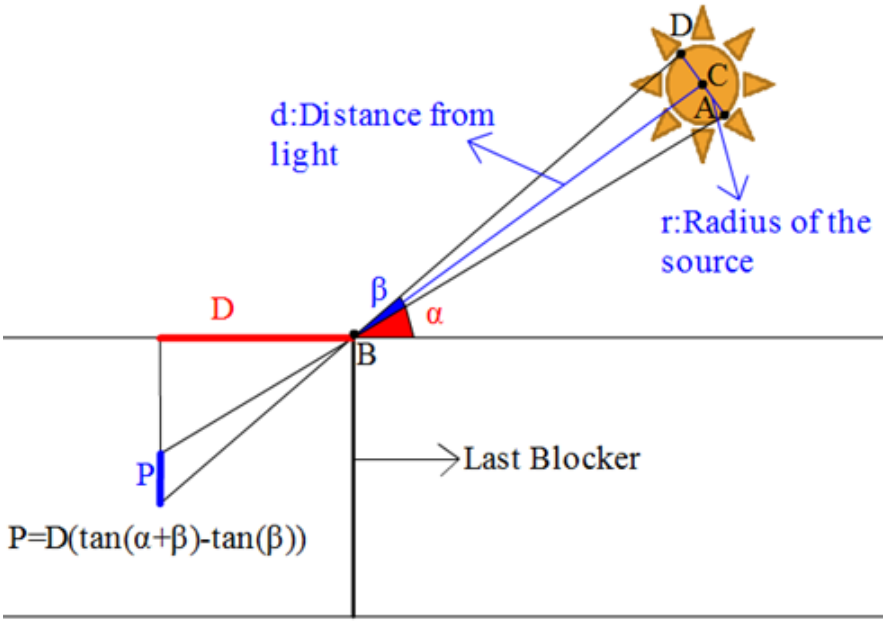


Figure 4.15. Calculation of the varying penumbra height.

The inductive recurrence relation of the penumbra height and the visibility determination function covering both shadow and varying penumbra heights are given in the following:

$$P^c_0 = 0$$

$$P^c_{k+1} = \begin{cases} P_{unit} & \text{if } H^c_k - \Delta y \geq H^c_{k+1} \; AND \; H^c_k > R^c_k \quad (1) \\ R^c_k - H^c_k + P_{unit} & \text{if } H^c_k - \Delta y \geq H^c_{k+1} \; AND \; R^c_k \geq H^c_k \geq R^c_k - P^c_k \quad (2) \\ P^c_k + P_{unit} & \text{if } H^c_k - \Delta y \geq H^c_{k+1} \; AND \; R^c_k - P^c_k > H^c_k \quad (3) \\ P^c_k + P_{unit} & \text{if } H^c_k - \Delta y < H^c_{k+1} \quad (4) \end{cases} \tag{4.2}$$

$$O = min\left(1, max\left(0, \frac{SHM_{ShadowHeight}\left(x_{shm}, z_{shm}\right) - y_{world}}{p + SHM_{PenumbraHeight}\left(x_{shm}, z_{shm}\right)}\right)\right) \tag{4.3}$$

There are four distinct cases of the Equation 4.2. In the first case, the current height map sample is a new silhouette point and therefore, we reset the penumbra height. The second case handles the occluder fusion problem. If we have a height map sample, whose height is in the penumbra region of the last blocker, we reduce the current penumbra height to the difference between the current shadow height and the height map sample. A more physically correct approach would take all previous height map samples into account to calculate the current penumbra height; but this would make the shadow volume extraction on a single SHM column a $O(N^2)$ process. Adjusting the penumbra like that in the case of occluder fusion preserves the linearity of the algorithm while obtaining a logical approximation to the ground truth. In the third case, the height map sample is in the umbra region of the last blocker and in the fourth case, the segment between H_k^c and H_{k+1}^c is front-facing to the Sun. We simply increase the current penumbra height by the unit penumbra height for these conditions. In general, we observed an extra running time around 0.3-0.6 ms with the modified CUDA kernel. This is caused mainly by the 64-bit memory writes we use in this kernel.

In the visibility determination of the Equation 4.3, we still add a constant amount of penumbra height, p, to the sampled penumbra. This is purely for filtering purposes, especially at locations near to the blockers, where the varying penumbra is too small. However, it is used with a smaller value in this case in order to avoid overly softened shadow edges near to blockers. Notice that shadow and penumbra heights are stored in distinct channels of the SHM texture and we only use a single texture lookup to load both of them.

4.6. Bilinear Interpolation as the Reconstruction Method

In Chapter 3, we saw that for shadow maps we cannot use texture bilinear filtering as the reconstruction filter, due to the binary nature of the visibility test and the geometric irrelevance of the data contained within the depth map. For SHMs, we can safely use bilinear filtering as the reconstruction filter since SHM is the discretized version of a shadow volume and hence the data contained in it is geometrically meaningful. We can exactly reconstruct a single shadow plane by using bilinear interpolation on discrete height samples taken from it. We can prove this like in the following:

Theorem 4.1. *Bilinear interpolation between height samples from the same shadow plane can exactly reconstruct it.*

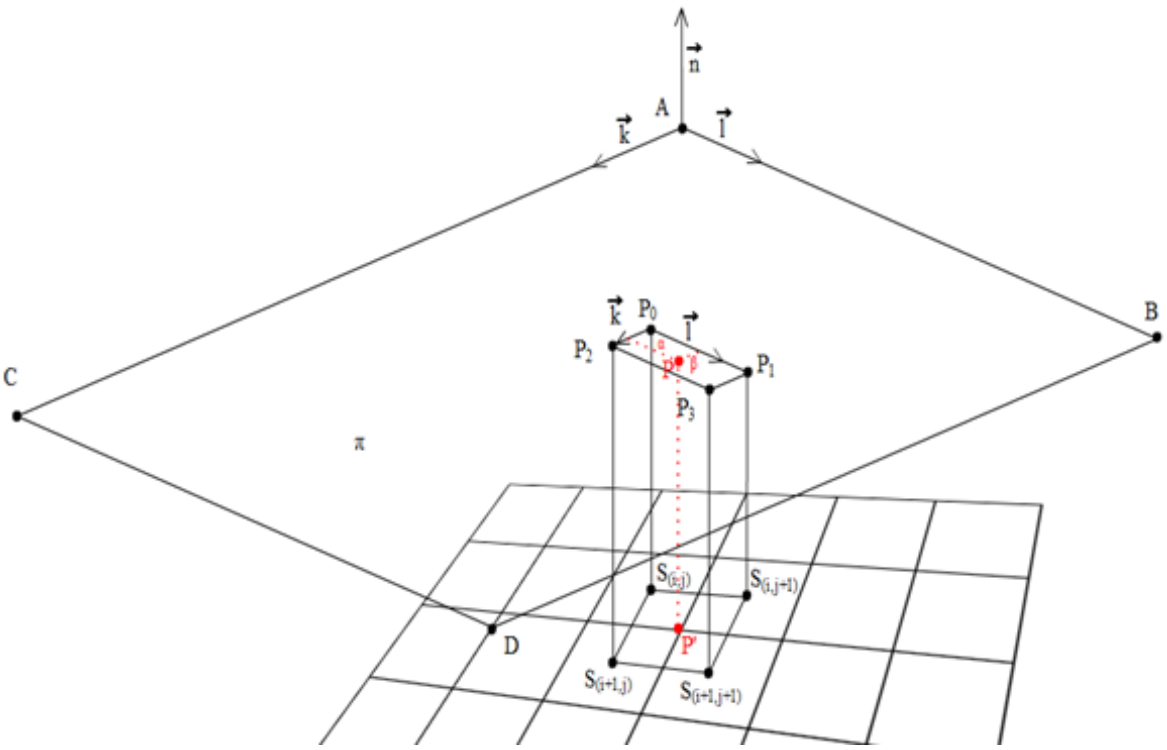


Figure 4.16. Reconstructing shadow heights from the SHM with bilinear interpolation.

Proof. Refer to Figure 4.16. $\overline{AB}$ is a silhouette edge, which is casting the shadow plane $\pi : \overrightarrow{n}(x-A) = 0$. $\overrightarrow{k}$ is the normalized vector of the light direction. $\overrightarrow{l}$ is equal to $\frac{B-A}{|AB|}$.

$\overrightarrow{n}$ is the normal vector of the plane π such that $\overrightarrow{n} = \frac{\overrightarrow{k} x \overrightarrow{l}}{|\overrightarrow{k} x \overrightarrow{l}|}$. $S_{(i,j)}$, $S_{(i,j+1)}$, $S_{(i+1,j)}$, $S_{(i+1,j+1)}$ are SHM sampling locations containing shadow heights (y-coordinates) of the points P_0, P_1, P_2 , P_3, respectively. The SHM is aligned with the height map such that both have the y-axis as the up vector. The sampling location P' corresponds to the point P which is bilinear interpolated from the points P_0, P_1, P_2 , P_3 according to the coefficients α and β. We need to show that P is always on the plane π.

We know that the points P_0, P_1, P_2 , P_3 are on the plane π. For $0 \leq i \leq 3$, it is:

$$\begin{aligned} \overrightarrow{n}\,(P_i - A) &= \overrightarrow{n}\,P_i - \overrightarrow{n}\,A = 0 \\ \overrightarrow{n}\,P_i &= \overrightarrow{n}\,A \end{aligned} \tag{4.4}$$

Bilinear derivation of P from these points is formulated as:

$$P = (1-\beta)\,(1-\alpha)\,P_0 + (1-\beta)\,\alpha P_1 + \beta\,(1-\alpha)\,P_2 + \beta\alpha P_3$$

We put P in the plane equation for π. If P is on this plane, the equation must give 0:

$$\begin{aligned} \overrightarrow{n}\,(P - A) &= \overrightarrow{n}\,((1-\beta)\,(1-\alpha)\,P_0 + (1-\beta)\,\alpha P_1 + \beta\,(1-\alpha)\,P_2 + \beta\alpha P_3 - A) \\ &= (1-\beta)\,(1-\alpha)\,\overrightarrow{n}\,P_0 + (1-\beta)\,\alpha\,\overrightarrow{n}\,P_1 + \beta\,(1-\alpha)\,\overrightarrow{n}\,P_2 + \beta\alpha\,\overrightarrow{n}\,P_3 - \overrightarrow{n}\,A \end{aligned}$$

Using Equation 4.4, we substitute each $\overrightarrow{n}\,P_i$ $(0 \leq i \leq 3)$ with $\overrightarrow{n}\,A$:

$$\begin{aligned} \overrightarrow{n}\,(P - A) &= (1-\beta)\,(1-\alpha)\,\overrightarrow{n}\,A + (1-\beta)\,\alpha\,\overrightarrow{n}\,A + \beta\,(1-\alpha)\,\overrightarrow{n}\,A + \beta\alpha\,\overrightarrow{n}\,A - \overrightarrow{n}\,A \\ &= ((1-\beta)\,(1-\alpha) + (1-\beta)\,\alpha + \beta\,(1-\alpha) + \beta\alpha)\,\overrightarrow{n}\,A - \overrightarrow{n}\,A \\ &= \overrightarrow{n}\,A - \overrightarrow{n}\,A = 0 \end{aligned}$$

□

As long as the sampling locations of the SHM correspond to the same shadow surface plane, we can exactly reconstruct it with simple bilinear interpolation. If they belong to different shadow planes, interpolation will give an approximation of the shadow surface. Since our sampling locations are located on the regular grid of the SHM image, we will always obtain an approximation of the actual shadow volume. We can adjust the correctness of this approximation by using the ray/height map resolution ratio, R. For our case with regular grid height maps, $R = 1$ provides a very good approximation to the exact shadow volume. From Theorem 4.1, we reach the following important result:

> *Our method will always generate pixel-correct shadows for the currently approximated shadow volume of the height map by bilinear interpolating between SHM samples.*

4.7. Moving Light Source and Temporal Aliasing Problem

4.7.1. The Reason

In the previous section, we showed how we generate an approximate shadow volume by interpolating between SHM samples. As the directional sun light rotates on the terrain, the orientation of the SHM texture changes with it as well. As we cast rays for each successive frame, our discrete approximation for the height map shadow volume slightly changes. The reason for this is the changing locations of the shadow height samples, in other words the temporal incoherence between shadow height samples. The ground truth in this case would be extracting the silhouette of the height map for each new light direction and extruding it, which is the case of the shadow volumes method [39]. But as we showed, we are shooting rays with equal distance to each other and in every frame; we obtain a slightly different approximation of the real shadow volume. This minor frame-to-frame incoherence of the shadow volume approximation results in so-called "flickering" artifacts. Flickering shows itself as rapid changes and

oscillations of the shadow edges as the sun light direction rotates and it is a quietly disturbing visual artifact. This is similar to the temporal aliasing artifacts of shadow maps, which is caused by the changing rasterization of the shadow map for each frame as the camera or light moves, as described in [6] and [8]. The temporal incoherence of the sampling locations is visualized in Figure 4.17. In it, the blue square and samples belong to the current frame. Red samples belong to the previous frame but rotated according to the current light direction. The red square shows the alignment of the height map according to the previous light direction. It can be clearly seen how the sampling positions differ slightly. In Figure 4.18, the difference between the ground truth of silhouette extraction and SHM ray casting is shown.

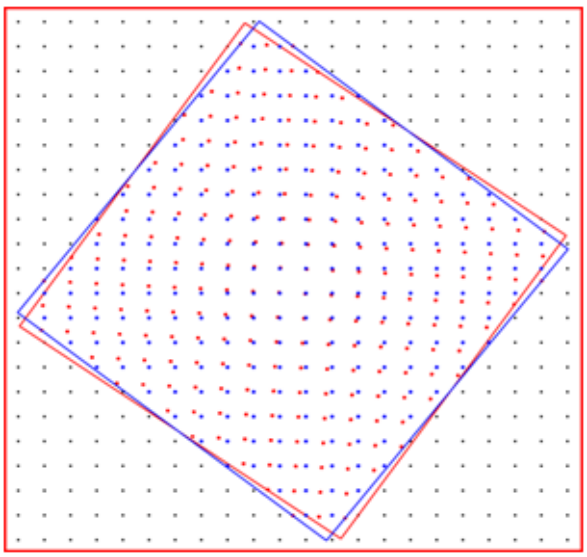


Figure 4.17. Two SHMs of successive frames for a rotating light direction. Notice the changing sampling locations.

In our algorithm, this temporal incoherence error was the most important and difficult problem which we have addressed. In the following subsections we explain our proposed methods to reduce the problem.

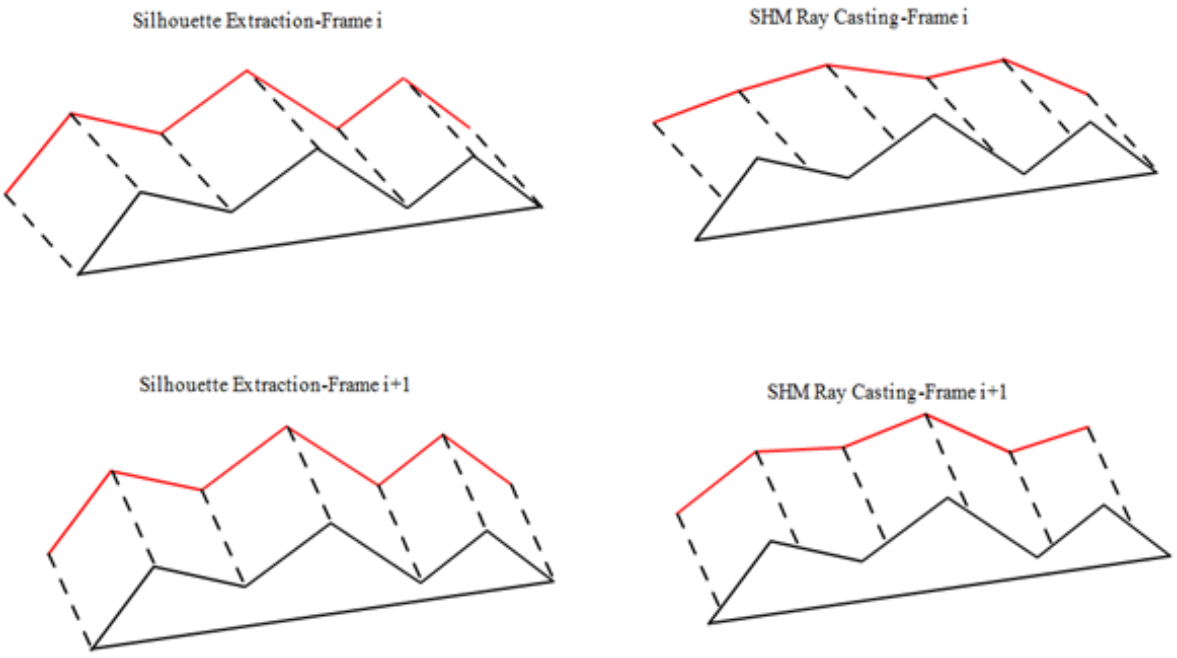


Figure 4.18. Difference between silhouette extraction (ground truth) and SHM ray casting. Notice how the changing ray locations cause the shape of the shadow to differ between successive frames in SHM method.

4.7.2. Method 1: Interpolation between Two SHMs

Our first solution to the temporal incoherence problem is defining an interpolation interval over a certain amount of frames. If the light direction changes with the angle θ each frame, we define the interval $\Delta\alpha = N\theta$, where N is the number of frames which we are going to interpolate over. Assuming light's current angle to its starting direction as α, we calculate two distinct SHMs on the CUDA side, one for the current light direction, SHM_{α}, and one for the light direction at the end of the interpolation interval, $SHM_{\alpha+\Delta\alpha}$. We generate two transformation matrices to sample from these two textures and pass them to the pixel shader stage. For each pixel, we sample two shadow heights from SHM_{α} and $SHM_{\alpha+\Delta\alpha}$ and execute visibility test for both of them. According to our current frame in the interval, we linearly interpolate between the shading results from two SHM textures and this becomes our final amount of shadow. Mathematically, this can be shown as in the following for constant penumbra soft shadowing:

$$
\begin{aligned}
O_{\alpha} =& min\left(1, max\left(0, \frac{SHM_{\alpha}\left(x_{SHM_{\alpha}}, z_{SHM_{\alpha}}\right) - y_{world}}{p}\right)\right) \\
O_{\alpha+\Delta\alpha} =& min\left(1, max\left(0, \frac{SHM_{\alpha+\Delta\alpha}\left(x_{SHM_{\alpha+\Delta\alpha}}, z_{SHM_{\alpha+\Delta\alpha}}\right) - y_{world}}{p}\right)\right) \\
O_{Final} =& (1 - \frac{n}{N})O_{\alpha} + \frac{n}{N}O_{\alpha+\Delta\alpha}
\end{aligned} \tag{4.5}
$$

Where n is our current number in the interpolation interval. Extending Equation 4.5 for varying penumbra shadowing is trivial, we just execute varying penumbra visibility test of Equation 4.3 for O_{α} and $O_{\alpha+\Delta\alpha}$ instead of the constant penumbra one.

By interpolating over a number of frames, we reduce frame-to-frame high frequency shadow edge flickers. Temporal aliasing still exists between SHM_{α} and $SHM_{\alpha+\Delta\alpha}$ but since a new shadow edge is introduced once in N frames, the frequency of oscillations reduces and the flickering effect becomes less disturbing. Increasing N reduces flickering more but as we interpolate on a larger $\Delta\alpha$, shadow movement starts to lose its smoothness: The interpolation effect becomes apparent in which shadows of the angle α slowly disappear while the ones of $\alpha + \Delta\alpha$ emerge. This effect of shadow interpolation is more detectable for high features (like tall hills), due to higher angular displacement of their shadows. According to the frequency of the height map, the amplitudes of its features and the angular velocity of the light rotation, a good value for N is one which reduces the flickering as possible while not exposing the interpolation effect and damaging the shadow movement smoothness in an unacceptable amount, so there is a trade-off.

Shadow interval interpolation provides us the opportunity to cache SHMs during the interpolation. We only need to calculate a new SHM at the end of each interval. This is a very important source of optimization such that the cost of generating a new SHM is amortized over N frames and our general frames-per-second rate vastly increases. With this caching ability, our algorithm performs in a comparable speed to

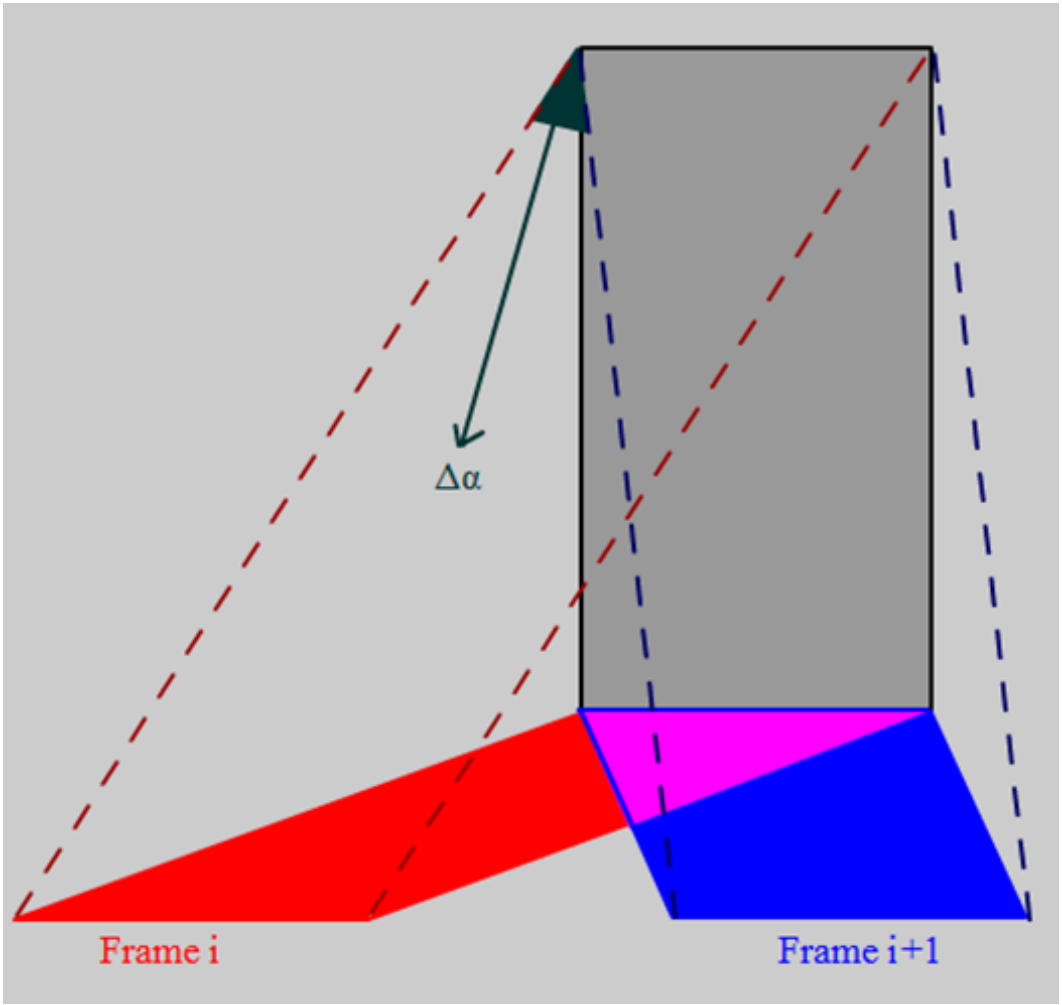


Figure 4.19. Shadow interpolation illustrated: We interpolate over an angle of $\Delta\alpha = N\theta$, as the shadow of the $Frame\ i$ disappears, the shadow of the $Frame\ i+1$ will start to appear.

shadow mapping, which is the fastest available shadowing algorithm in literature. We owe this caching mechanism to the camera independence of our algorithm. This will be explained in more detail in the following sections.

4.7.3. Method 2: Temporal Low Pass Filter

Our second method to reduce temporal aliasing and flickering artifacts is a temporal averaging (mean) filter. We use this filter in conjunction with the interval interpolation. In this method, we define a filter size K. Assuming we have a in-

terpolation interval angle of $\Delta\alpha$, we generate $2K + 2$ SHMs, one for the current frame, SHM_α , one for the next interval, $SHM_{\alpha+\Delta\alpha}$, K SHMs for the previous intervals, $SHM_{\alpha-\Delta\alpha}$, $SHM_{\alpha-2\Delta\alpha}$, ..., $SHM_{\alpha-K\Delta\alpha}$ and K SHMs for the next intervals, $SHM_{\alpha+2\Delta\alpha}$, $SHM_{\alpha+3\Delta\alpha}$, ..., $SHM_{\alpha+(K+1)\Delta\alpha}$. We generate appropriate sampling matrices for each SHM and pass them to the pixel shader stage. Our visibility test in the pixel shader is same with Equation 4.5 in the principle. We will again interpolate between the shadowing values of the current light angle, α and the next one, $\alpha + \Delta\alpha$. This time, the shadow heights we use in visibility tests are not single samples from both SHM_α and $SHM_{\alpha+\Delta\alpha}$, they are the average of $2K + 1$ heights we sampled from previous and next intervals for each one, instead. The current pixel's world coordinates are transformed with the SHM matrix for each light direction and the shadow height is sampled from the appropriate SHM location for that direction. In general, we need to execute $2K + 2$ texture sampling operations for the whole procedure, since $2K$ samples overlap for the current and the next angle. The core idea of this procedure is to introduce a temporal low-pass frequency filter. By averaging heights from successive frames, we reduce high frequency height changes which cause the disturbing flickering artifact at shadow edges. As we increase the filter size K, we observe more stable, oscillation-free shadow edges due to elimination of high frequencies. But increasing the filter size K has its adverse effects. First one is the increased amount of texture samples. Texture sampling is a performance bottleneck in graphics processing due to the limited texture memory bandwidth [33] and therefore increased filter size results in longer rendering times. Another issue is memory consumption. Since we need to cache $2K + 2$ SHM textures, we introduce two new SHM textures to the video memory for each larger temporal filter. We addressed this memory consumption issue in an extra effort which we will talk about in coming sections.

In order to manage $2K + 2$ SHM textures, we need to allocate a large texture, which is large enough to contain that amount of smaller textures, without overlapping. We use a greedy algorithm at the program start to determine the dimensions of such a tightest containing texture. Packing many textures into a larger one and sampling them by using scaling and offsets in the pixel shader stage is a common method, which is called as the *texture atlasing* [33]. Only for the first frame of the program, we calculate

$2K + 2$ SHMs at the CUDA side in a single kernel invocation. For the next intervals, we only calculate a single SHM, $SHM_{\alpha+(K+2)\Delta\alpha}$, and write that over the one which is the last recently used in the previous frame, namely $SHM_{\alpha-K\Delta\alpha}$. The mathematical representation of the temporal mean filter is given in the Equation 4.6. Notice for $K = 0$, the temporal filtering turns into plain interval interpolation of the Equation 4.5. n is, again, our current frame number in the interpolation interval.

$$
\begin{aligned}
O_\alpha =& min\left(1, max\left(0, \frac{\frac{1}{2K+1}\sum_{i=-K}^{K} SHM_{\alpha+i\Delta\alpha}\left(x_{SHM_{\alpha+i\Delta\alpha}}, z_{SHM_{\alpha+i\Delta\alpha}}\right) - y_{world}}{p}\right)\right) \\
O_{\alpha+\Delta\alpha} =& min\left(1, max\left(0, \frac{\frac{1}{2K+1}\sum_{i=1-K}^{K+1} SHM_{\alpha+i\Delta\alpha}\left(x_{SHM_{\alpha+i\Delta\alpha}}, z_{SHM_{\alpha+i\Delta\alpha}}\right) - y_{world}}{p}\right)\right) \\
O_{Final} =& (1 - \frac{n}{N})O_\alpha + \frac{n}{N}O_{\alpha+\Delta\alpha}
\end{aligned}
\tag{4.6}
$$

We implemented a temporal median filter, in addition to the mean. This time, we do not take the average of the shadow heights, we find the median of them for both the angles α and $\alpha + \Delta\alpha$. Then we linearly interpolate between the median shadow heights. The effect of the temporal median filter is almost identical to the temporal mean filter, without any clear improvement on the temporal aliasing artifacts.

The filter size we are going to use depends on the needs of the current height map geometry. For low frequency, low amplitude height maps, $K = 1$ or $K = 2$, provide acceptable flickering reduction in conjunction with shadow interval interpolation. If there are tall features on the height map due to high amplitude, a large interpolation interval N could damage shadow movement smoothness due to large angular displacement. In such cases larger filters like $K = 3$ or $K = 4$ generate visually satisfactory results with a smaller interval size. Moreover, high frequency features like very small and short hills consisting of a few vertices tend to produce more serious flickering artifacts since their shadow volume approximations change very rapidly from frame to frame due to the undersampling of the SHM rays. Therefore, filter sizes of $K = 3$ or $K = 4$ could be

desirable for high frequency height maps containing such features, as well. Increasing the penumbra height in order to over-soften or eliminate their shadows is another solution since the shadows of such small features do not make significant contributions to overall rendering quality, because they cannot or can only be barely distinguished from the general shading of the height map and therefore are often negligible. Increasing the ray/height map resolution ratio is the ultimate solution for a better shadow volume approximation, but the associated memory cost can be high.

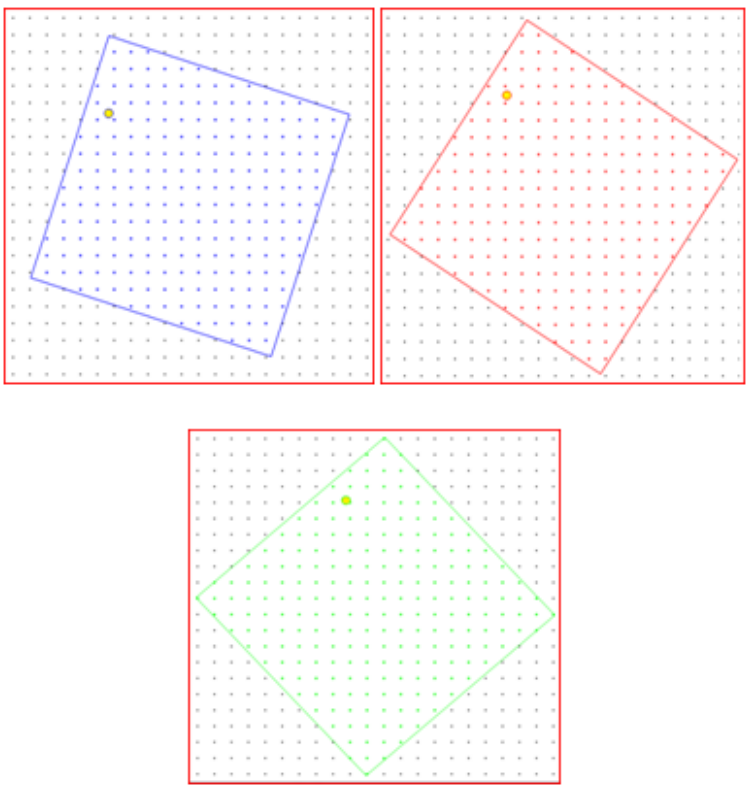


Figure 4.20. Three successive SHMs for a temporal filter of size $K = 1$. The current pixel's location in each SHM is shown by the yellow dot.

4.7.4. Method 3: Jittering

In Section 4.4.2, we described that we place the height map in the SHM such that their middle points coincide. Our first idea was to give small, random offsets to the height map for each interval such that its center point becomes shifted for

small amounts every time. Instead of rotating in a regular 2D fashion, new sampling locations gets irregular coordinates with this method and that helps to reduce shadow edge oscillations. We experimentally deduced, that translating the height map on the SHM such that its left-most corner lies on the left edge and top-most corner lies on the top edge of the SHM introduces a good source of jittering and reduces oscillations. We simply adjust the world-to-SHM space conversion matrix in Equation 4.1 to add this modification into our pipeline, so adding jittering support does not bring any overhead.

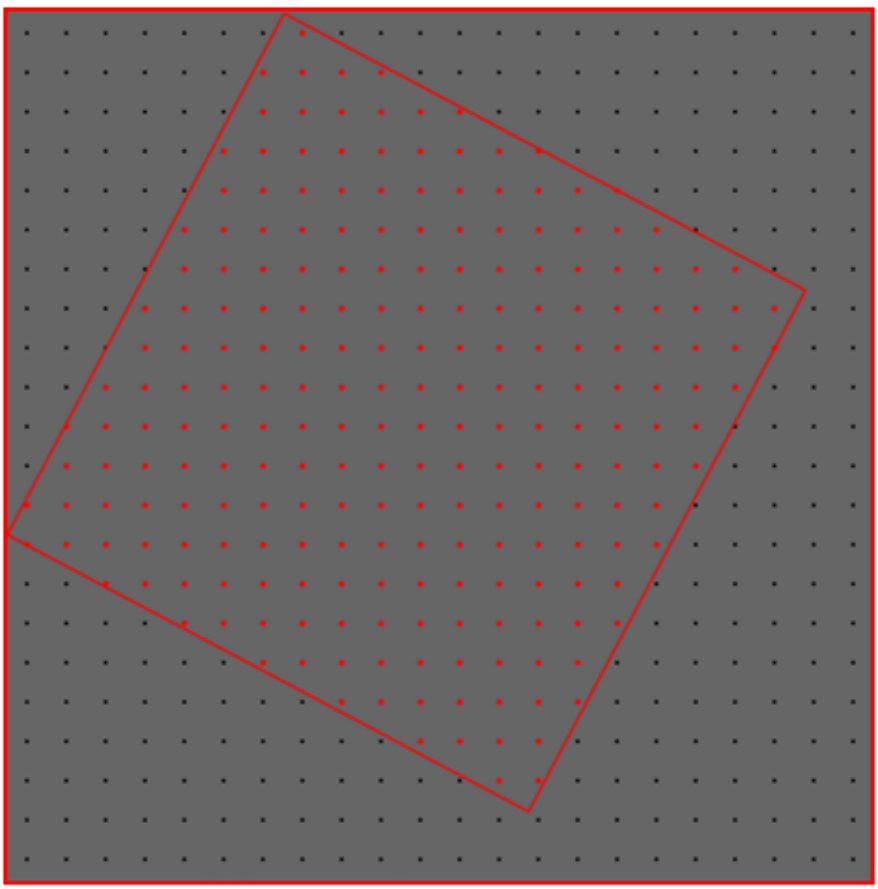


Figure 4.21. Aligning the left most and top most corners of the rotated height map with SHM edges provides a natural source of jittering.

4.7.5. Method 4: Defining a Threshold for Penumbrae

We observed that flickering artifact is most serious in the penumbra region of the shadows, where amount of shadowing (occlusion) is low and the terrain ground nearly receives sun light completely. We introduced a threshold value, $1 \geq T \geq 0$, and we set all occlusion values smaller than this threshold directly to zero (making completely lit). For occlusion values $1 \geq O \geq T$, we renormalize them according to the interval $[T, 1]$. This provides smaller but more stable soft shadow edges because we pruned high frequency changes at lower occlusion values. In order to make the pruned shadow edges wider, we add a small parametric bias, b, to the shadow height values at the CUDA side and then increase the constant penumbra height, p. T is a parametric value and can be changed according to needs of the height map geometry at hand. Additionally we can control the rate at which the shadow hardness change in the penumbra region by defining a "hardness exponent", h, for the interval $[T, 1]$. With this final modification, the Equation 4.6 becomes the Equation 4.7 as follows (Notice that in Equation 4.7, the bias b is added at the CUDA side actually):

$$
\begin{aligned}
O_\alpha =& min\left(1, max\left(0, \frac{b + \frac{1}{2K+1}\sum_{i=-K}^{K} SHM_{\alpha+i\Delta\alpha}\left(x_{SHM_{\alpha+i\Delta\alpha}}, z_{SHM_{\alpha+i\Delta\alpha}}\right) - y_{world}}{p}\right)\right) \\
O_{\alpha+\Delta\alpha} =& min\left(1, max\left(0, \frac{b + \frac{1}{2K+1}\sum_{i=1-K}^{K+1} SHM_{\alpha+i\Delta\alpha}\left(x_{SHM_{\alpha+i\Delta\alpha}}, z_{SHM_{\alpha+i\Delta\alpha}}\right) - y_{world}}{p}\right)\right) \\
O'_\alpha =& \begin{cases} 0 & \text{if } O_\alpha < T \\ \left(\frac{O_\alpha - T}{1-T}\right)^h & \text{if } O_\alpha \geq T \end{cases} \\
O'_{\alpha+\Delta\alpha} =& \begin{cases} 0 & \text{if } O_{\alpha+\Delta\alpha} < T \\ \left(\frac{O_{\alpha+\Delta\alpha} - T}{1-T}\right)^h & \text{if } O_{\alpha+\Delta\alpha} \geq T \end{cases} \\
O_{Final} =& (1 - \frac{n}{N})O'_\alpha + \frac{n}{N}O'_{\alpha+\Delta\alpha}
\end{aligned}
\tag{4.7}
$$

4.8. Other Minor Problems of the Algorithm and Our Solutions

4.8.1. Shadow "Sides" Problem

In our experiments we observed that sometimes the "sides" of the shadows seem to be cut off abruptly: The hard shadow (umbra) area immediately passes to completely lit area without or with a very narrow area of soft shadow (penumbra). This gives an unrealistic look to shadows, especially to the shadows of small features. The cause of this artifact is the rapid change in the shadow elevation data at shadow edges. The shadow of the current height map feature mixes with the shadow of a former shadow casting feature at such locations and since those shadow heights belong to different casters, bilinear interpolation does not return a physically meaningful result: Higher shadows of the current caster get rapidly interpolated with the much lower shadows of a former caster, whose heights are lower than the terrain surface. We implemented a routine at the CUDA side which detects such texels as we generate the shadow height information. We need to determine texels at the "sides" of shadows and give them a bias such that their heights become equal to the terrain surface at the corresponding locations. This will avoid rapid change in the shadow height elevation and provide softer transitions. In general, detecting such texels would require an expensive edge detection operation on the SHM image. But since the shadow volume extraction operation is vertically aligned and done column-wise, we can use CUDA shared memory to check neighboring threads and determine whether our current texel is neighboring to a shadowed texel, only with a small modification to the original ray casting kernel. All threads keep the record of their "shadowed" - "not shadowed" states and write this information into the shared memory location indicated by their thread IDs. If the current and previous texels, $T_{(i,j)}$ and $T_{(i-1,j)}$, are not shadowed while at least one of the left, top left, right or top right texels ($T_{(i,j-1)}$, $T_{(i-1,j-1)}$, $T_{(i,j+1)}$,$T_{(i-1,j+1)}$ respectively) is shadowed, then the current texel resides at the "side" of a shadow and we set the shadow height in this texel equal to the terrain surface. (Additionally, the penumbra height is set to zero if we are using varying penumbrae). Figure 4.22 illustrates a sample situation from a colored SHM texture.

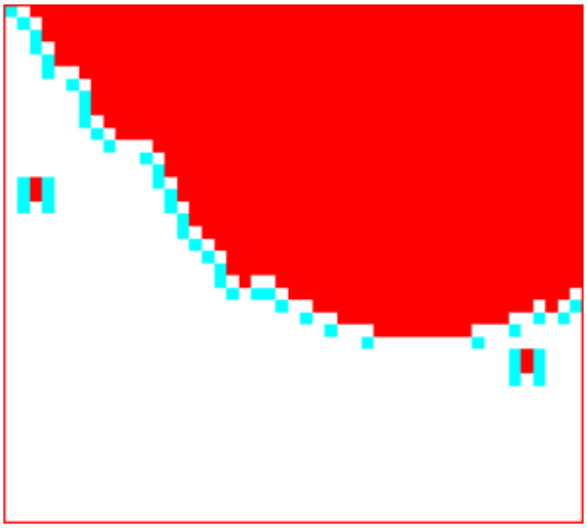


Figure 4.22. A part of the colored SHM texture. Red pixels show samples shadowed, white pixels show unshadowed sampling locations. Cyan pixels show potential "shadow side" artifact locations. At these pixels we add a bias to the shadow height.

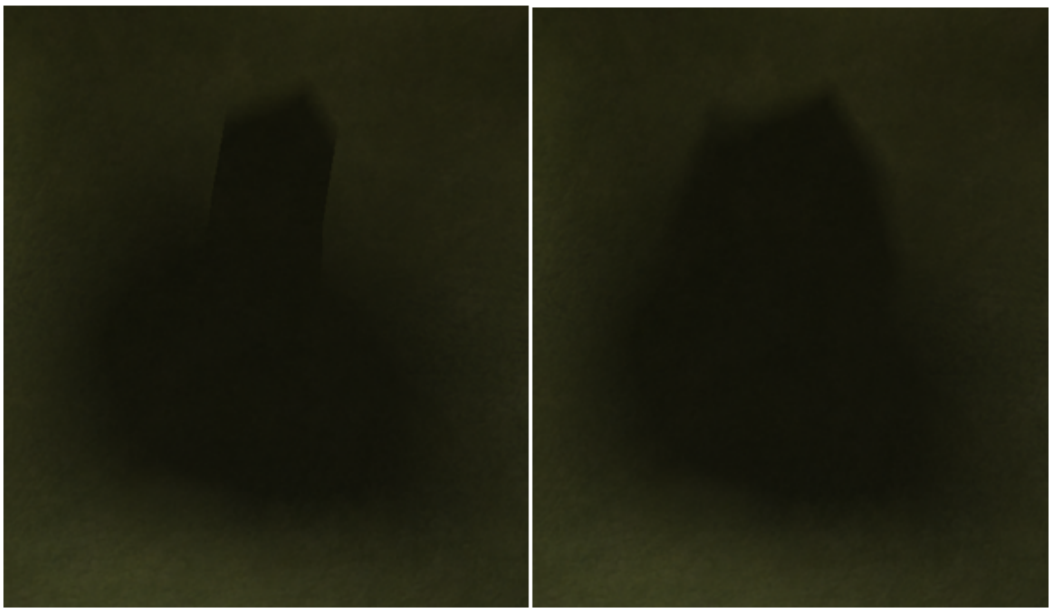


Figure 4.23. Actual images from our program showing the "shadow side" error. On the left image, we can see how the shadow is cut off abruptly. The image on the right shows the same location with our shared memory fix.

4.8.2. Reducing Memory Consumption

The last issue we addressed is the memory consumption of our algorithm. As we increase the size of our temporal flicker reducing filter, we add two new SHM images at each larger filter size. For $K = 5$, we would need to cache 12 SHM textures at any given frame. Moreover, the memory consumption grows quadratically with the height map size. For example, a height map with the size of 1024^2 needs 4 times more space compared to a size of 512^2. We can reduce the ray/height map resolution ratio, R, but as R becomes smaller than 1, temporal aliasing errors increase rapidly since the obtained shadow volume approximation diverges from the ground truth due to undersampling.

We implemented an alternative approach in which we store constant and varying penumbra SHM textures into 16 and 32-bit Direc3D unsigned integer textures, respectively, instead of 32 and 64-bit floating point ones. This effectively provides a $\frac{1}{2}$ reduction for the total memory space needed. For constant penumbra SHM textures, we define a height interval and quantize this interval into 2^{16} unit heights. We store into the unsigned integer texture the coefficient with which we can most closely represent the current floating point shadow height, if multiplied with the unit height. As the interval to quantize, we chose the minimum and maximum height values of the height map data. The shadow height can never be larger than the maximum elevation of the height map data so it provides an optimal upper bound. Our ray casting algorithm may decrease the shadow height below of the minimum elevation value of the height map but since shadow heights smaller than the minimum elevation value will never cast shadows on the height map surface, we can safely clamp such heights to the minimum elevation and obtain an optimal lower bound. On the pixel shader stage, we reconstruct the shadow by multiplying the coefficient sampled from the SHM texture with the unit height and adding it to the minimum height map elevation. For the varying penumbra soft shadows, we quantize the penumbra height with the shadow height as well and store them into 16-bit channels of a 32-bit unsigned integer texture. The unit penumbra height for an arbitrary shadow interval is multiplied with a parametric coefficient and this becomes our quantization unit. We divide the floating point

penumbra height at each texel by this quantization unit and store the coefficient into the 16-bit channel. For a moving light source setting, there will be a separate quantization unit for each $2K + 2$ SHM texture and we load all of those into the constant registers of the pixel shaders. In the pixel shader stage, we simply multiply the sampled penumbra coefficient with those quantization units and reconstruct actual penumbra heights. Since we map continuous floating point values into discrete intervals, these compression schemes are lossy. But in practice, we observed in our experiments with different height maps that the intervals are always small enough to represent shadow and penumbra height changes and using discrete integer textures did not introduce any visual artifacts. Therefore we deduce that we can safely use half precision textures instead of full ones, without introducing any artifacts.

The process of quantizing and reconstructing shadow height can be given as the following:

- At CUDA side:

$$H_{unit} = \frac{H_{max} - H_{min}}{2^{16}}$$
$$SHM_{16-bit} = \lfloor \frac{H_{ShadowHeight} - H_{min}}{H_{unit}} + 0.5 \rfloor$$

- At Pixel Shader:

$$H_{ReconstructedShadow} = H_{unit} * SHM_{16-bit}\,(x_{world}, z_{world}) + H_{min}$$

The process of quantizing and reconstructing penumbra height is given in the following. c is the parametric coefficient we define. At the pixel shader side we additionally make a multiplication with 2^{16} since Direct3D normalizes unsigned integer textures into $[0, 1]$:

- At CUDA side:

$$SHM_{16-bitPenumbraChannel} = \lfloor \frac{P_{PenumbraHeight}}{cP_{unit}} + 0.5 \rfloor$$

- At Pixel Shader:

$$H_{ReconstructedPenumbra} = 2^{16} * cP_{unit} * SHM_{16-bitPenumbraChannel}\,(x_{world}, z_{world})$$

4.9. Comparison of Our Method with Shadow Mapping

Our shadow height map based method is most similar to shadow mapping, since both are image based methods. Our method has some advantages and disadvantages compared to shadow mapping and in this section we will give a comparative analysis of their differences.

Our method is completely camera independent. Our method is independent from the position and direction of the camera, the size and shape of the viewing frustum or distances of the camera near and far planes. Our shadow quality and algorithm performance does not vary with movement of the camera. Since the SHM is aligned with the base height map, camera properties do not affect shadow generation. In comparison, shadow mapping is directly related with the camera position and direction. Moreover, the size and shape of the viewing frustum affects the generated shadow quality and performance. Real-time shadow mapping quality improvement methods like focusing [63], warping [61], [62], [64], [65] and partitioning [66], [67] depend on the camera position and direction. Focusing tries to maximize shadow quality by concentrating the shadow map resolution on the view frustum – scene intersection. The quality improvement gained with warping based methods is dependent on the relation of the light – camera directions. Such methods give optimal result when the camera and light directions are perpendicular to each other. In the case of the "dueling frusta" where the camera look vector and light direction vector are parallel

to each other, these methods degrade back to uniform shadow mapping which is a big disadvantage. Partition based methods do not suffer from special camera-light settings as warping based ones do and they provide more robust shadow quality. But they are dependent on the size of the viewing frustum. Since the viewing frustum is split into different parts for different shadow maps, a smaller viewing frustum is always advantageous since more shadow map resolution is used for smaller world geometry. In our setting where we render a large terrain, the viewing frustum is especially large in order to render distant details and this is disadvantageous for shadow mapping where our method is not affected by the frustum size. Additionally, camera location and direction are important factors for shadow mapping performance. If the camera-scene intersection is large, we need to render a huge amount of geometry into the shadow map and this degrades overall performance. Especially in our case of terrain rendering, in a camera setting with which we can view a large portion of the terrain geometry (a typical fly-over camera, commonly used with flight simulators, for example), we observe a rapid drop in the frame rate. In our SHM based method, we do not suffer from such performance hits caused by different camera settings.

Since our method is camera independent, the SHM image is only a function of the light direction whereas the shadow map is a function of the camera location-direction and light direction. We can describe this mathematically as the following:

$$SHM = f(\overrightarrow{l})$$
$$ShadowMap = f(\overrightarrow{l}, \overrightarrow{v}, \overrightarrow{p}, n, f, FoV, r)$$

Where $\overrightarrow{l}$ is the light direction, $\overrightarrow{v}$ is the camera view direction, $\overrightarrow{p}$ is the camera position, n and f are view frustum near and far plane distances, FoV is the camera field of view angle and r is the screen width-height ratio.

Our camera independence provides us the opportunity to cache SHMs of successive light directions and to interpolate between them, as mentioned in Section 4.7.2.

This postpones SHM generation cost to the start of each interpolation interval and allows us to implement temporal aliasing reduction strategies we discussed in Section 4.7. Shadow mapping does not have this advantage since it depends on six more parameters which are related to the camera and every time the camera moves, the corresponding shadow map must be recalculated. We cannot cache shadow maps for different light directions because any of these parameters may change and render the cached shadow maps useless during the interpolation interval. (For example the camera can move or rotate during the interpolation interval and the cached shadow map images may not contain depth information for newly rendered world geometry.) For shadow maps, temporal reprojection with a so-called history buffer is proposed to address this caching problem in [75], but this method deals only with scenes with static objects and static light sources and cannot handle dynamic scenes where we have a movable light source.

If the light source is completely static or moves only at certain times, without the need of a smooth movement, our method's camera independence becomes a very precious feature, since a single SHM can be calculated as a very rapid preprocess and can be used throughout the whole application runtime only at the cost of a single texture lookup per pixel, whereas shadow mapping needs recalculation each time the camera moves. In general, we deem the camera independence as the most important advantage of our method over shadow mapping.

Our method is less affected by aliasing and uses cheap hardware bilinear interpolation as the reconstruction method. Aliasing is the most serious problem of shadow mapping as we mentioned in Chapter 3. Shadow mapping suffers from projection and perspective aliasing due to the positions and orientations of the camera and individual polygons, respectively. Additionally, it suffers from temporal aliasing errors, which results in flickering of the shadow edges, since the rasterization of the shadow map changes whenever the camera or the light source moves. To address the aliasing errors caused by undersampling, shadow mapping needs special filtering and reconstruction methods like Percentage Closer Filtering (PCF) [9] or Variance Shadow Maps [76]. Hardware texture bilinear filtering cannot be used with shadow maps due to their unlinear nature. Such reconstruction methods are usually expensive since they require

many texture lookups. (A $N*N$ PCF kernel needs N^2 texture lookups). As mentioned before, shadow mapping is especially prone to generate aliasing artifacts with height map based terrains since hugely tessellated and large height map chunks are rendered into the limited resolution of shadow maps, leading to undersampling. We observed each of the three aliasing errors often in our observations with shadow mapping: Close up camera positions to the shadow edges cause stair-step edges as the result of perspective aliasing. When a polygon is nearly parallel to the sunlight direction, projection aliasing errors appear. Finally, camera and light movements will introduce very disturbing flickering artifacts due to temporal aliasing. To reduce and hide the underlying undersampling error, which causes all those artifacts, shadow mapping needs high resolution images and PCF kernels with large dimensions in our terrain rendering system, which degrade the performance and increase the needed memory storage.

Although our method is image based also, it does not suffer from projection and perspective aliasing like shadow mapping. In Section 4.6 we showed that the SHM contains an approximated discrete version of the actual shadow volume of the height map and it can generate pixel-correct shadows for this approximation by using simple linear interpolation. Therefore our method is free from stair-step artifacts caused undersampling, unlike in the case of shadow mapping. Unfortunately, our shadow volume approximation slightly changes every frame as the sunlight moves and this causes flickering errors. But our algorithm can reduce that kind of temporal aliasing errors more successfully than shadow mapping: Shadow mapping mostly relies on spatial reconstruction filters to hide the undersampling which in turn hides flickering, to a degree. Methods like [75] exist, but they are limited with static scenes. Our method's camera independence allows us to cache SHMs of successive light directions and interpolate between them. Additionally, we devised a temporal smoothing filter to further reduce flickering errors. Since we are camera independent, flickering will only occur with sunlight movements in our case, whereas shadow mapping suffers from flickering as the result of both camera and sunlight movements; separately or together. In short, we are much less affected by aliasing errors and furthermore, our reconstruction filter is simple hardware bilinear interpolation whereas shadow mapping needs costly, large PCF kernels.

Our method does not suffer from depth bias – self shadowing issues. Shadow maps suffer from depth bias errors, caused by low image resolution and insufficient floating point precision. This results in self shadowing errors called as shadow acne and light leaking and it is a major source of problem for shadow maps. Like aliasing, self shadowing errors become more serious for terrain rendering since height map is a highly self-shadowing geometry by its nature [81]. Moreover, large PCF kernels worsen self-shadowing errors [69] and this makes the problem more serious in our case with terrain rendering, since we need to use large PCF kernels in order to hide undersampling errors satisfactorily. Defining an amount of bias to add depth values sampled from the shadow map reduces self shadowing errors, but this is not a global solution since the needed bias changes according to the slope of polygons. Therefore, a certain amount of shadow acne and/or light leaking artifacts always existed in our experimentations with shadow mapping.

Our algorithm does not suffer from depth biasing artifacts due to the nature of its visibility test. Our visibility test does not depend on a binary comparison with a single value stored in the depth buffer texel center, we make a bilinear interpolation between the nearest texels and reconstruct a geometrically meaningful shadow height value instead. Since we are able to generate pixel-correct shadows for the approximated height map shadow volume our system does not allow any self shadowing artifacts to occur.

We can generate constant and varying penumbrae soft shadows very cheaply. With our definition of a constant penumbra height, we can generate approximate soft shadows just at the cost of a single texture lookup. Shadow mapping needs large and expensive PCF kernels to obtain similar soft shadowing effect. Additionally, we implemented a geometrical based varying penumbra soft shadowing approach. With this method, we are able to generate harder shadows near to shadow caster and softer shadows in the distant. We are able to obtain this varying soft shadow effect with minimal performance hit. Shadow mapping needs specialized methods in order to generate varying penumbrae sizes. Regular, constant sized PCF kernels generate constant sized penumbrae regardless of the caster distance. A popular method which extends Per-

centage Closer Filtering to support varying penumbrae is the Percentage Closer Soft Shadows method [10]. This method adjusts the size of the PCF kernel according to the result of a blocker distance search heuristic for each pixel. The blocker search step involves many texture lookups which degrades the overall performance. In general, it is hard and costly to alter standard shadow mapping to make it support soft shadows with varying penumbrae, while we have managed to add this feature to our algorithm with a small modification and almost at no additional performance cost.

Our method can work flawlessly in conjunction with any height map LOD algorithm. As we mentioned before, we load the original height map elevation data as a texture into the CUDA framework and in the ray casting kernels, we sample the corresponding height data from it by using bilinear interpolation. This scheme is completely independent from any underlying LOD algorithm, which adjusts the mesh detail of the height map according to some screen-space error metric since it always works with the highest resolution height map data. As the y-coordinates of terrain vertices change according to the LOD of the current terrain node and the GeoMorphing algorithm, we do not pass their world y-coordinates directly to the pixel shader stage to use in the SHM visibility test. We know that their x and z-coordinates are not affected by the LOD system, so we load the same full resolution height map data as a texture into the Direct3D runtime and by passing the world x,z-coordinates of terrain vertices into the pixel shader stage, we sample this height map image to obtain the corresponding height map elevation data, in each pixel. This renders the terrain self shadowing completely independent from the side effects of the LOD: We do not observe terrain surfaces which start to or stop from receiving shadows suddenly due to changing mesh detail and y-coordinates, as the camera moves.

Shadow mapping, on the other hand, cannot work with LOD algorithms that well. Shadow mapping must either use the full detail height map geometry to generate depth data or the simplified LOD version of it. The second choice is more reasonable since the first pass rendering will perform faster due to lower detail height map mesh and using the full detail geometry for shadow generation would lead to inconsistencies with the actual LOD using geometry while generating shadows. There are two disadvantages

related with the usage of LOD geometry in shadow generation. First, the low detail geometry will morph constantly because of the underlying GeoMorphing algorithm, as the camera moves. Moreover, some elevation features will be eliminated when a smaller LOD is used for the rendering of a specific terrain node. Small changes or missing features in the height map geometry in the light space due to LOD algorithm could be reflected as dramatic changes in the world space and may affect overall shadow quality. Additionally, as the camera moves, some terrain features may enter into or exit from shadows. This is due to fact that the y-coordinates of terrain vertices are constantly changed by the GeoMorphing algorithm. This is an artifact which would damage the effort we spend in order to hide the popping effect of the GeoMipmapping algorithm. In short, our method can deal with LOD algorithms gracefully while shadow mapping runs into problems we described above.

Shadow mapping needs a second rendering pass, where our method does not. Shadow mapping generates its visibility information by rendering the scene geometry from the light's view point. Since shadow map is dependent on the camera location and direction, this rendering must be done every time the camera moves. The time spent on this preprocess rendering is proportional with the amount of the geometry visible by the camera, such that there will be an important performance loss if the view frustum contains a large amount of the scene geometry. Our method does not need a second pass rendering. We only execute our ray casting CUDA kernels at the start of the interpolation intervals and its cost is amortized over the number of frames we are interpolating in.

Shadow mapping is a global method. Shadow mapping's advantage over our method is its generality: Shadow mapping can be used with any form of rasterizable geometry. Our method is limited with 2.5D geometry, which can be mathematically described with a function of the form $f(x, z) = y$. Height map geometry satisfies this requirement gracefully. Our method can generate shadows which the height map casts over itself and over other objects. To include shadows casted by arbitrary objects onto the height map or over other objects, we must combine our method with a shadow mapping or shadow volumes based approach. But since shadow mapping does not

work with height map geometry very well, we can decouple terrain shadowing from the shadow map and concentrate it over other arbitrary objects while we are handling shadows from the terrain with our method. In this aspect, our approach and shadow mapping are complementary to each other: We can combine the shadows generated by both algorithms, by using the occluder fusion methodology as mentioned in Chapter 3. Such a combined shadowing solution could yield much better shadow quality and performance, compared to a single, global shadow mapping approach, in the case of terrain rendering. Relieved from the burden of terrain shadowing by our method, shadow mapping can be optimized for rendering arbitrary objects. For example, it can be focused on small object geometries and can utilize its limited resolution much better. In this case its undersampling problem will be reduced or even more, it can be completely solved, which in turn leads to the usage of much smaller PCF kernels for reconstruction. Smaller PCF kernels will increase the overall performance since the needed amount of texture lookups reduce. Rendering many instances of the same geometry is a common case in terrain rendering; for example identical tree or building models can be duplicated and placed on different locations in order to build a forest or a city. In such a case, handling terrain shadowing with our algorithm and leaving other geometry to the shadow mapping allows the usage of an interesting shadow mapping modification, the *Instanced Shadow Maps* [90]. This method proposes a way to reuse the shadow map of an object for its other identical instances on the scene. On the other hand, features residing on the surface of the height map, like buildings, cars, trees, etc. can be approximated to the height map itself and our SHM method can generate their shadows as a part of the height map shadow volume. In such a case, shadow mapping will only be responsible for the shadows of objects, which are "flying". In general, combining our method with the shadow mapping for terrain rendering is an interesting topic and provides different opportunities for optimizing both the shadow quality and performance.

5. EXPERIMENTS

5.1. The Methodology

After implementing our algorithm, we conducted some tests which compare its performance and quality with the mainstream shadow mapping algorithm. Shadow mapping became our choice of competitor algorithm since it is image based like our method and it is the fastest general shadowing solution for directional outdoor lights, like the Sun [8]. As we mentioned in our literature survey in Chapter 3, standard shadow mapping as proposed originally in [40] is not enough by itself from a quality point of view. Therefore, we implemented the Parallel Split Shadow Mapping (PSSM) approach as our competitor algorithm [22], [66], [67], [69]. The partitioning approach of PSSM is considered more useful compared to the warping based approaches of [61], [62], [64] and [65] since it does not suffer from quality degradation caused by special camera-light settings as warping algorithms do. PSSM is compared to the warping approach and reported to be generating better shadow quality, according to the designers of the popular and state-of-the-art video game, FarCry 2 [68]. Additionally, its core algorithm is more simple and intuitive to implement. For each PSSM partition, we used the focusing approach of [38] in order to further increase the shadow map resolution allocated to the visible terrain parts. As the shadow map reconstruction method, we used the popular Percentage Closer Filtering (PCF) method [9]. We implemented the PCF totally in the pixel shader code and used the hardware acceleration presented by the modern GPUs. In order to be able to use the hardware PCF support, we used regular N^2 grid PCF kernels. Each sample of the PCF kernel is a hardware accelerated bilinear lookup returning a value in the interval $[0, 1]$. The results of N^2 samples are then averaged.

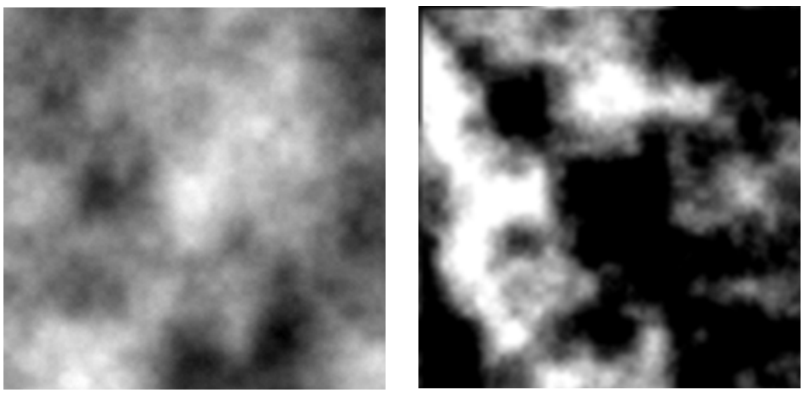
We compared our method with the PSSM both in quality and performance aspects. For the quality comparison, we focused the camera on two different locations on our two different test height maps. These locations are specially selected since they constitute difficulties for PSSM to produce a good shadow quality: They are closely

zoomed on the shadow receiver, causing perspective aliasing and the polygons receive the sunlight with a wide angle which causes projection aliasing. We aimed to show that our method does not suffer from these aliasing sources which are the main area of problem for shadow maps. We presented six different parameter settings for the PSSM for the both of the problem locations: We changed the split count, the shadow map resolution and the PCF kernel size among those settings. For our SHM method, for each focus location, we gave four snapshots zoomed on the same problematic area, using exactly the same camera position and direction as the PSSM snapshots. Two of the SHM snapshots show the constant penumbra approach; one with the full 32-bit and one with the 16-bit SHM texture resolution. The other two contain the result of the varying penumbra approach, one for the 64-bit and one for the 32-bit compressed SHM textures. Due to the dynamic nature of the flickering problem, we are not able to show the results of our reduction methods with still images.

For the performance comparison tests, we used two different height map geometries. These are 1024^2 sized procedural height maps which are generated with the Perlin Noise algorithm we implemented [23]. Both of these are high frequency height maps originally, generated with the Perlin Noise parameters of 0.004 frequency, 1.0 amplitude, 8 octaves and 0.5 persistence. We refer the reader to [23] and to [24] to get a detailed explanation of the algorithm and its parameters. We kept one of the high frequency height maps as it is generated by the algorithm and we applied a low pass averaging filter with the kernel size of 15 to the other one in order to eliminate high frequency components from it. We used those height maps for the snapshots we used in the quality comparison as well. The performance tests are conducted with a fly over camera, which looks exactly at the world space origin, $\langle 0, 0, 0 \rangle$ and rotates around the y-axis. In order to contain different amount of world geometry inside the viewing frustum, the camera is distanced 5000 world coordinate system units from the origin and five different tests are conducted with the same PSSM or SHM parameters with the camera elevation angle from the xz-plane set as 75°,60°, 45°, 30° and 15°. For each single PSSM and SHM parameter setting, the average of these five runs with different camera elevation angles are taken. By doing this we aim to include different amounts of geometry loads and their effect on the rendering speed. Each test run executes for

$500ms$ and finishes. Tests are done for a rotating light source environment. The sunlight rotates around the axis vector of $\langle 1, 1, 0 \rangle$ with a speed of $\frac{1.0*10^{-4}rad}{frame}$. For PSSM, a single parameter setting consists of the view frustum split count, ranging from one to four, shadow map resolution, one of the 1024^2, 2048^2 and 4096^2 for each frustum split and the PCF kernel size ranging from two to five. For each height map, this is equal to $4*3*4*5 = 240$ test runs, with a total of 480 for both height maps. For our SHM based method, we have two different parameters. First one is the length of the interpolation interval. We use the values $N = 25$, $N = 50$, $N = 75$ and $N = 100$ for the interval length. The second parameter is the temporal filter size. We use the values $K = 2$, $K = 3$, $K = 4$ and $K = 5$. In all of our SHM tests, the ray/height map resolution ratio, R , is equal to 1. Both PSSM and SHM performance test runs use completely identical camera and height map settings in order to make a fair comparison. In order to receive shadows from the height map, we included test objects into our scene. These are standard Utah Teapots which are provided by the Direct3D API as direct function calls.

The test machine on which we have conducted our experiments has a 2.6 GHz Intel Core2Quad Processor, 4GBs of RAM and a GeForce 450 GTS GPU with 1024 MBs of video memory and 192 CUDA cores. As the operating system, we use Windows Vista, 32-bit edition. We run our tests with a screen resolution of $1440 * 900$ pixels.



(a) (b)

Figure 5.1. (a) On the left, our high frequency height map. (b) On the right, our low pass filtered height map.

5.2. Quality Comparisons

5.2.1. Results

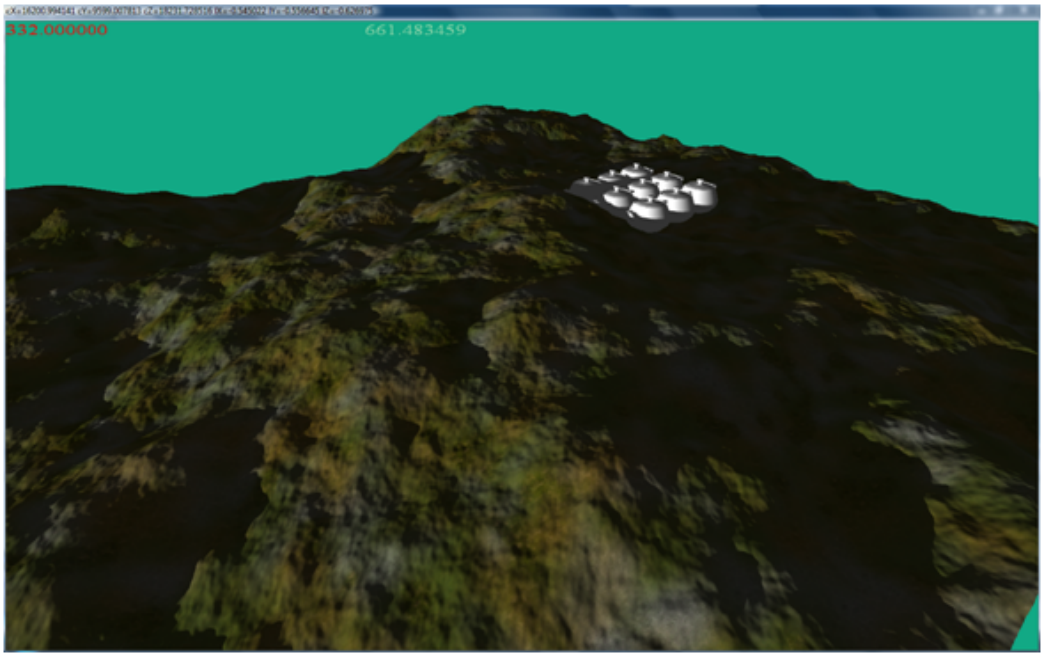


Figure 5.2. Actual rendered version of the high frequency height map of Figure 5.1a.

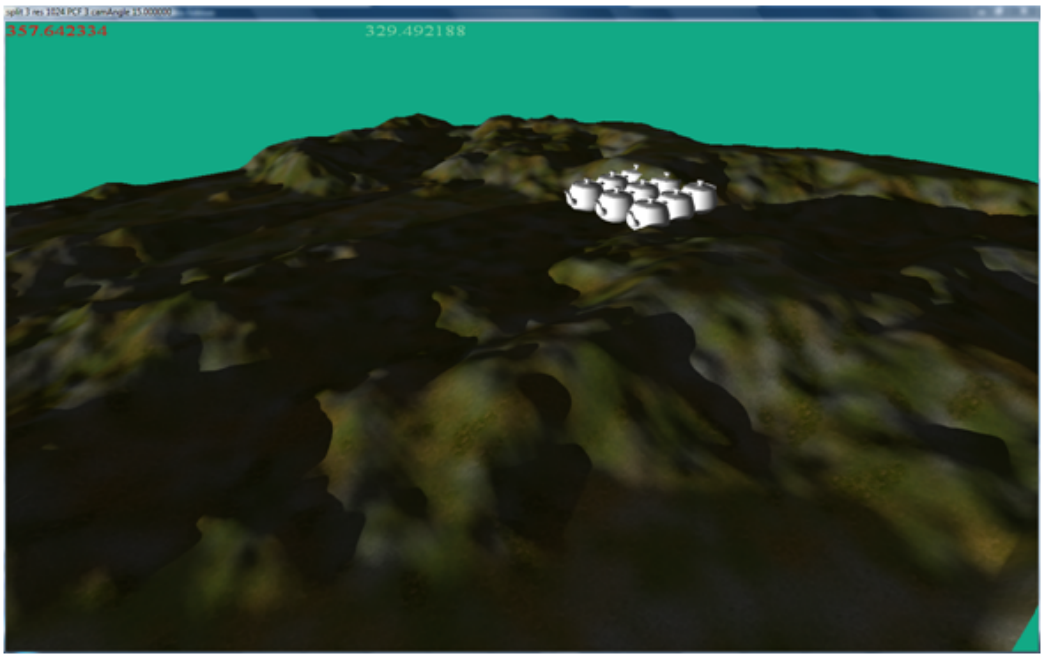


Figure 5.3. Actual rendered version of the low frequency height map of Figure 5.1b.

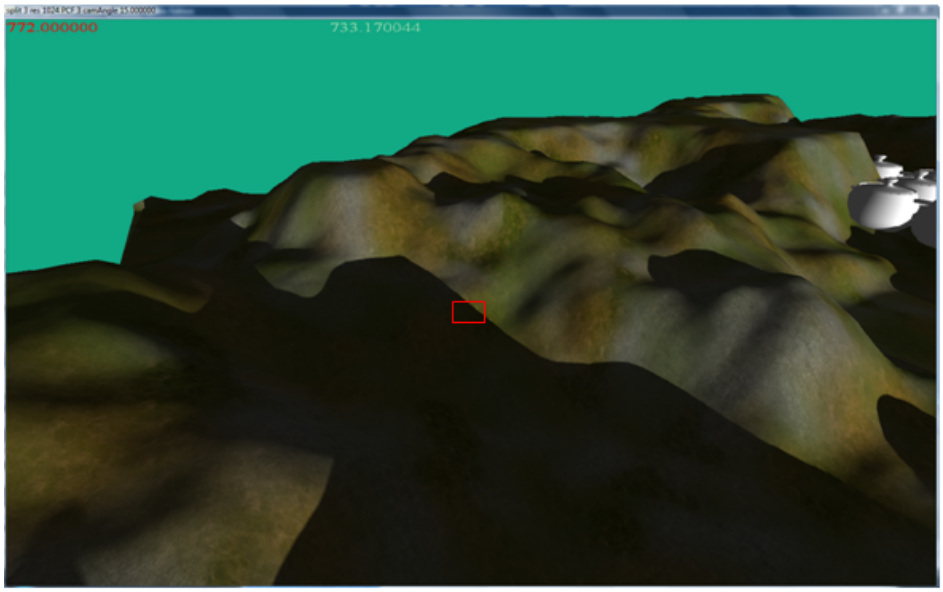


Figure 5.4. SHM Distance Shot. Const. Pen. SHM Falloff:0.55 Penumbra Height:1.5 Bias:30 FPS:772

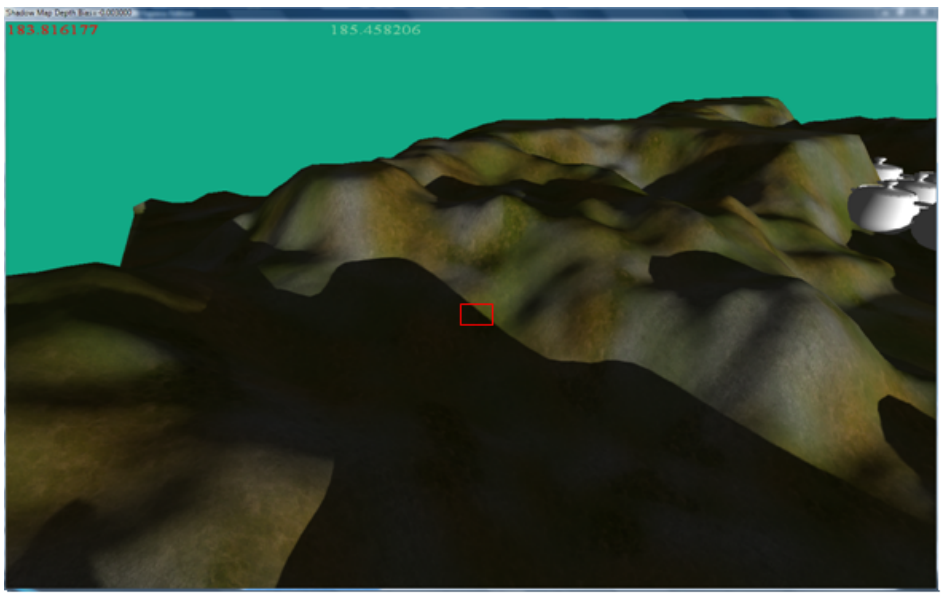


Figure 5.5. PSSM Distance Shot. PSSM Split Count:3 Shadow Map Resolution:1024^2 PCF Kernel:3^2 FPS:183

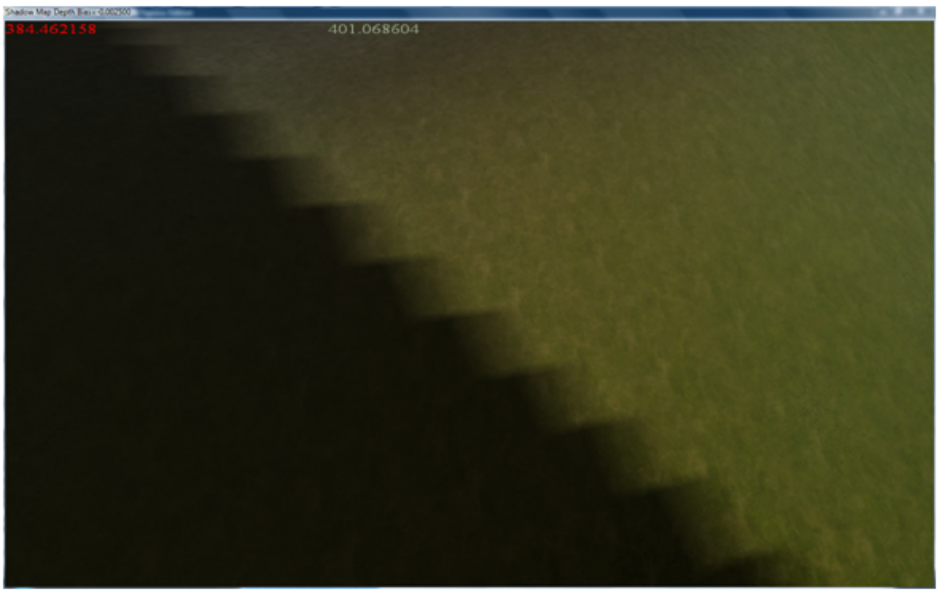


Figure 5.6. PSSM Split Count:1 Shadow Map Resolution:1024^2 PCF Kernel:2^2 FPS:384

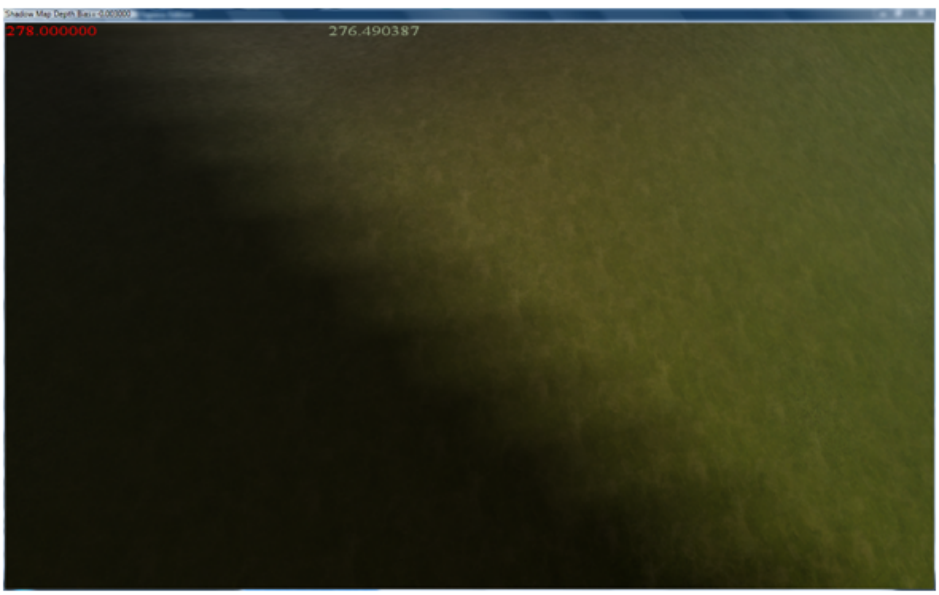


Figure 5.7. PSSM Split Count:1 Shadow Map Resolution:1024^2 PCF Kernel:5^2 FPS:278

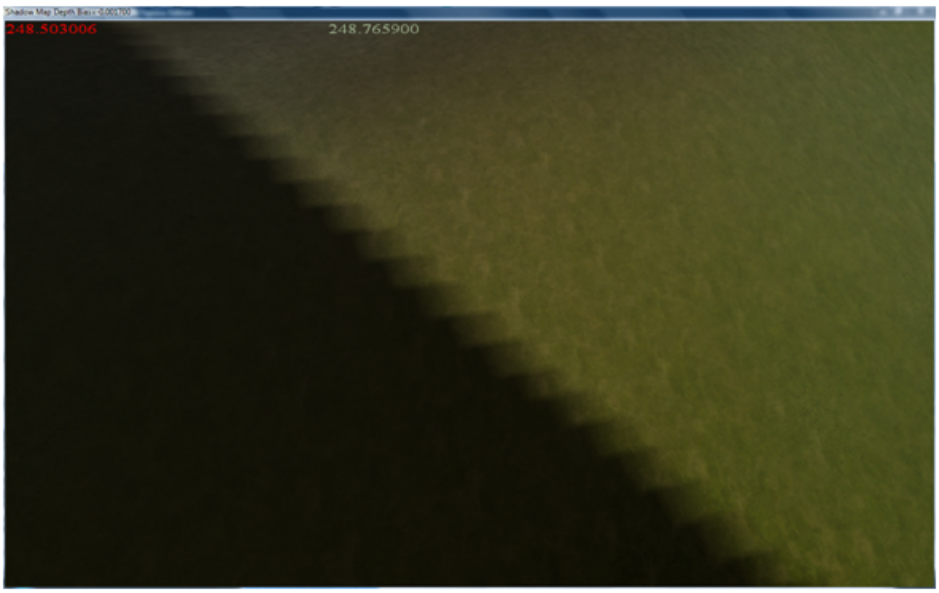


Figure 5.8. PSSM Split Count:1 Shadow Map Resolution:2048^2 PCF Kernel:3^2 FPS:248

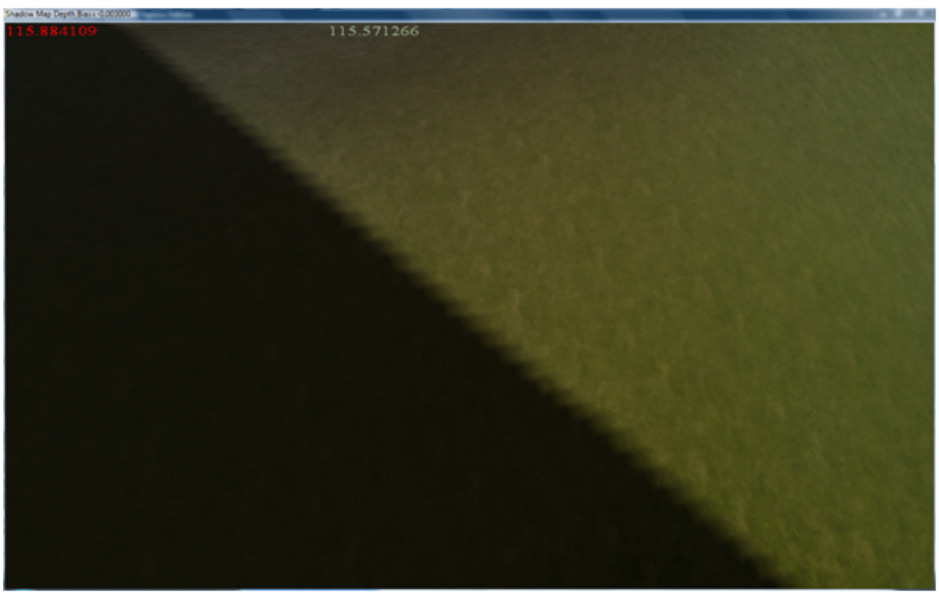


Figure 5.9. PSSM Split Count:3 Shadow Map Resolution:2048^2 PCF Kernel:3^2 FPS:115

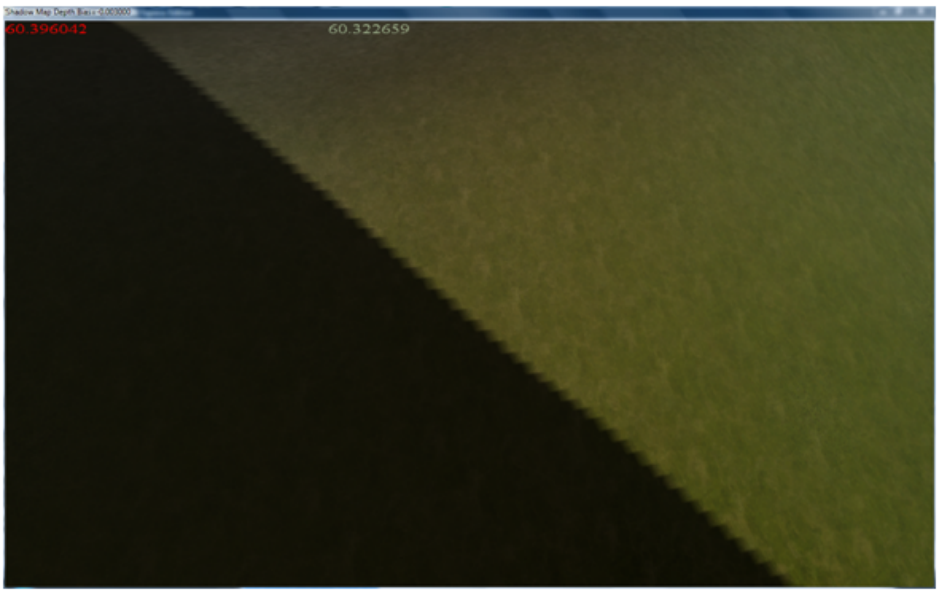


Figure 5.10. PSSM Split Count:2 Shadow Map Resolution:4096^2 PCF Kernel:2^2 FPS:60

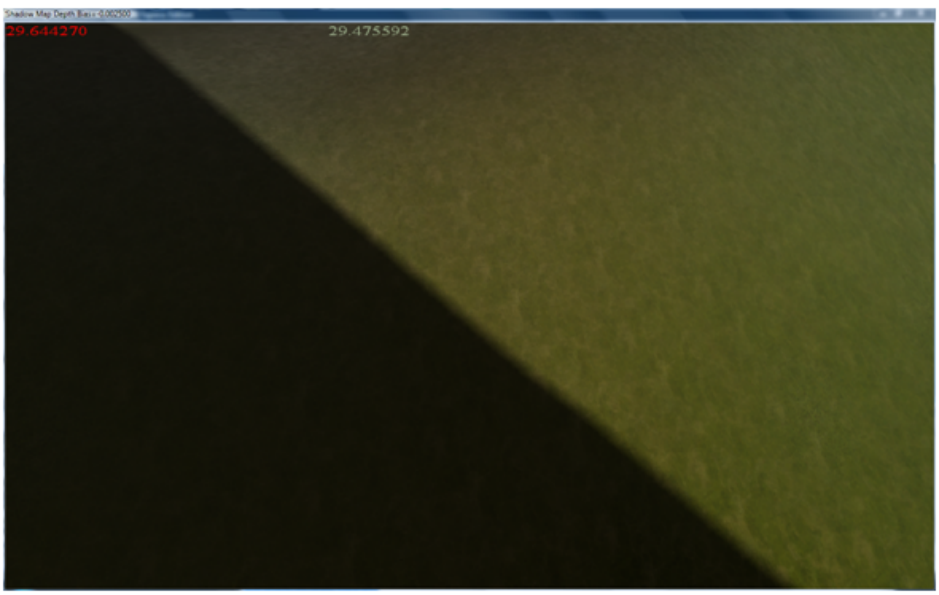


Figure 5.11. PSSM Split Count:4 Shadow Map Resolution:4096^2 PCF Kernel:5^2 FPS:29

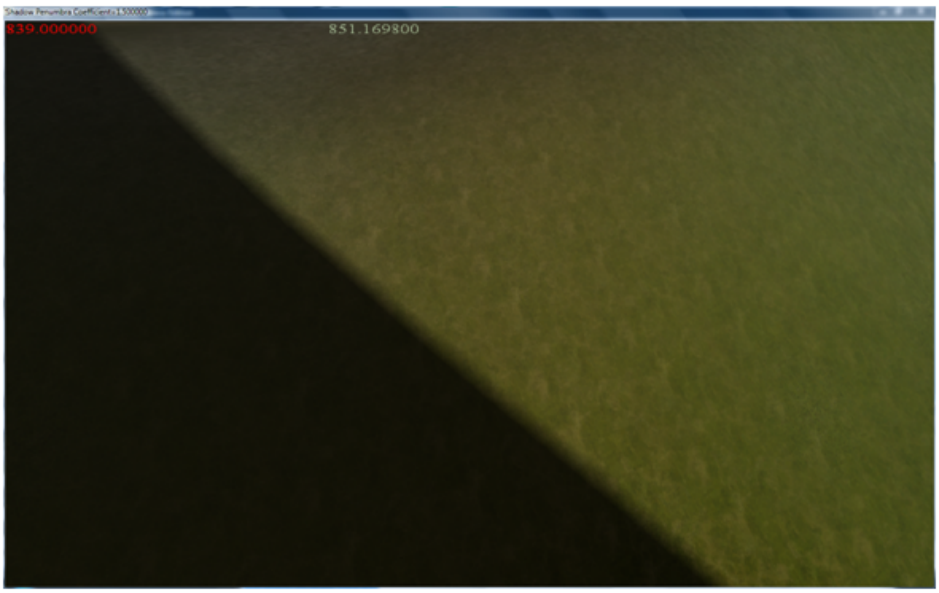


Figure 5.12. Constant Pen. 32-bit SHM Falloff:0.55 Penumbra Height:1.5 Bias:30 FPS:839

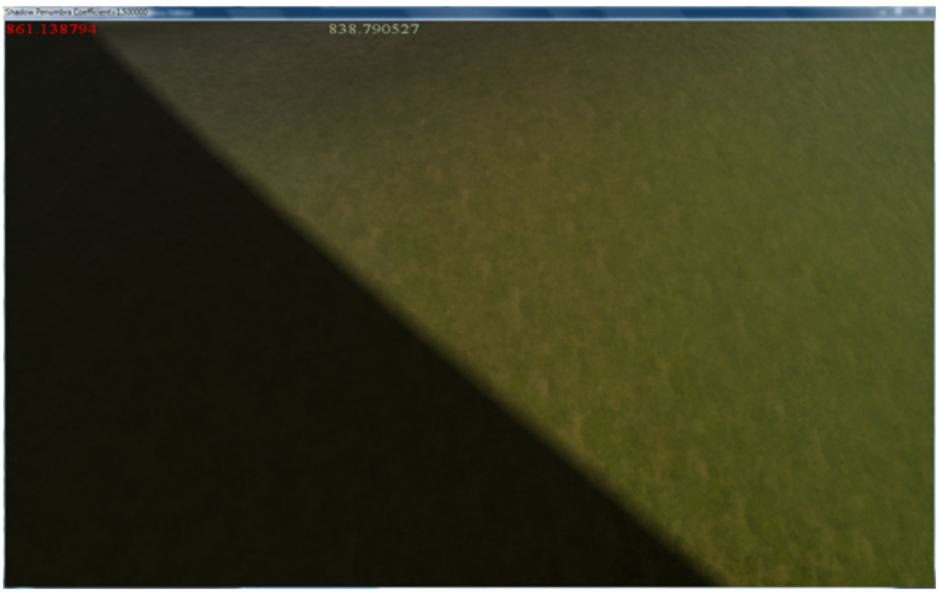


Figure 5.13. Constant Pen. 16-bit SHM Falloff:0.55 Penumbra Height:1.5 Bias:30 FPS:861

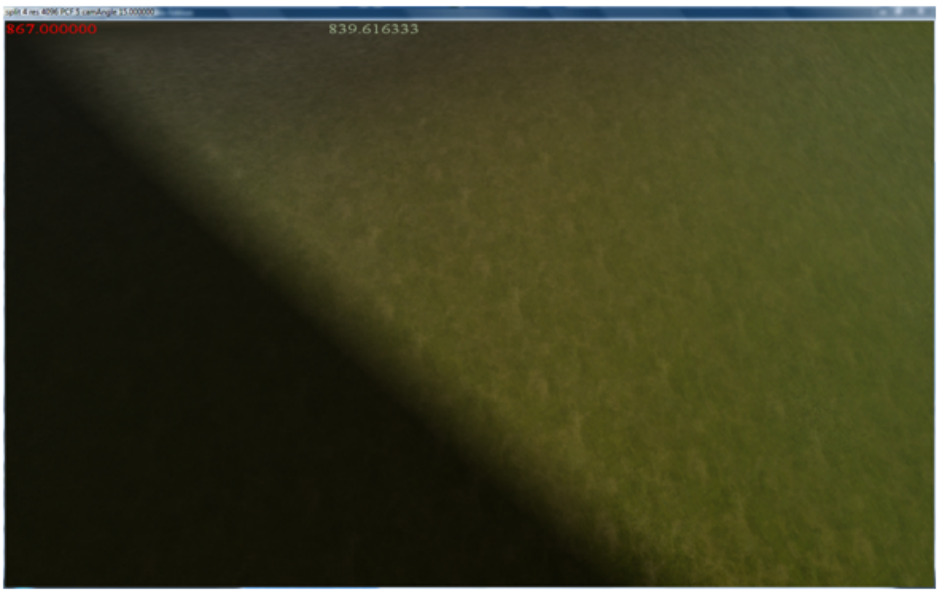


Figure 5.14. Varying Pen. 64-bit SHM Falloff:0.2 Penumbra Height:0.2 Bias:0 FPS:867

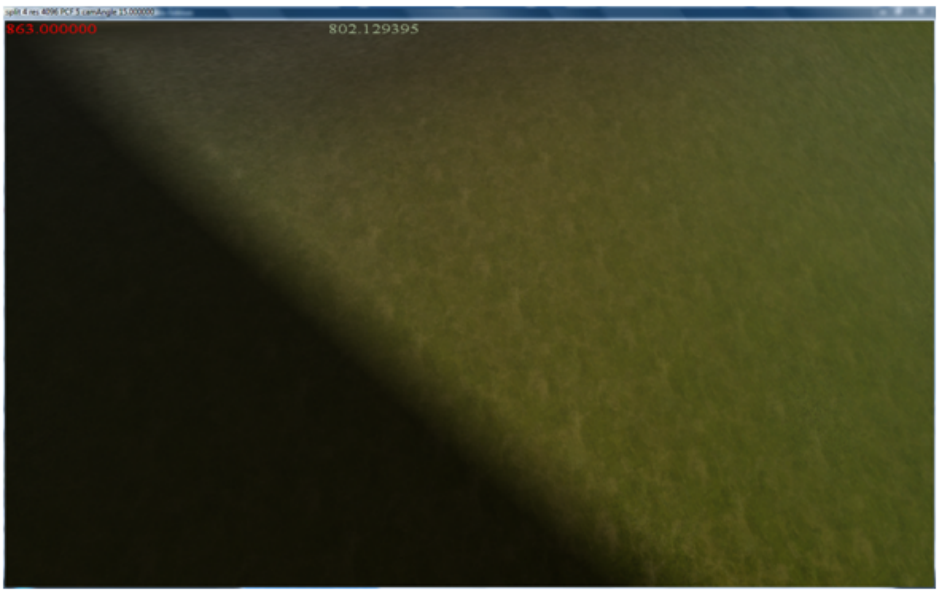


Figure 5.15. Varying Pen. 32-bit SHM Falloff:0.2 Penumbra Height:0.2 Bias:0 FPS:863

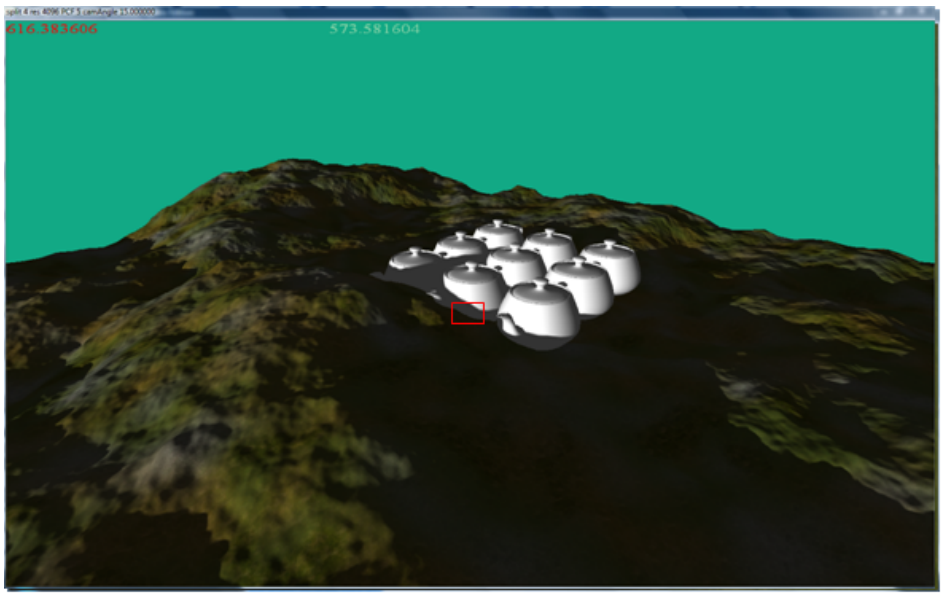


Figure 5.16. SHM Distance Shot. Const. Pen. SHM Falloff:0.55 Penumbra Height:1.2 Bias:30 FPS:616

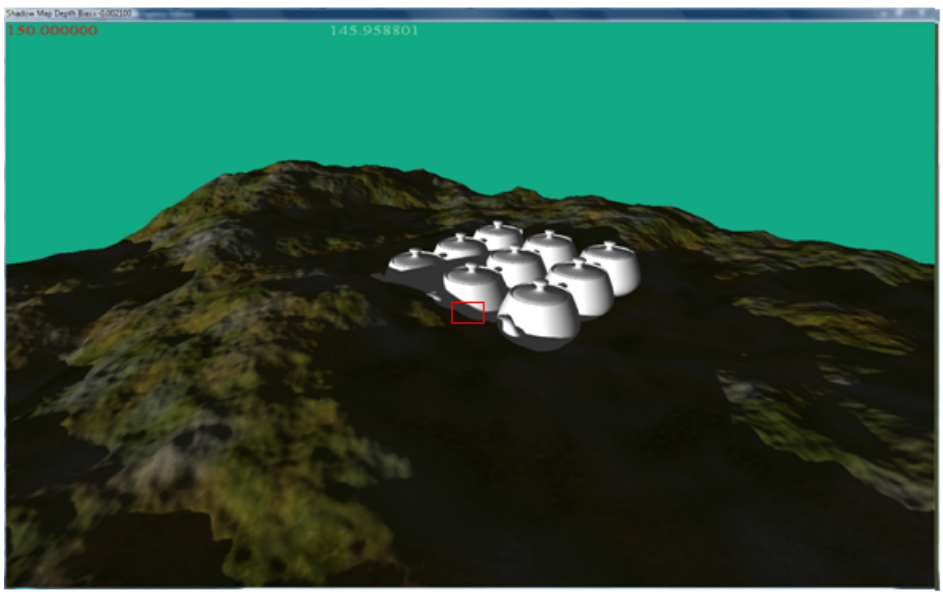


Figure 5.17. PSSM Distance Shot. PSSM Split Count:3 Shadow Map Resolution:1024^2 PCF Kernel:3^2 FPS:150



Figure 5.18. PSSM Split Count:1 Shadow Map Resolution:1024^2 PCF Kernel:2^2 FPS:374



Figure 5.19. PSSM Split Count:3 Shadow Map Resolution:1024^2 PCF Kernel:5^2 FPS:133



Figure 5.20. PSSM Split Count:1 Shadow Map Resolution:2048^2 PCF Kernel:2^2 FPS:281



Figure 5.21. PSSM Split Count:3 Shadow Map Resolution:2048^2 PCF Kernel:5^2 FPS:97



Figure 5.22. PSSM Split Count:1 Shadow Map Resolution:4096^2 PCF Kernel:2^2 FPS:134



Figure 5.23. PSSM Split Count:4 Shadow Map Resolution:4096^2 PCF Kernel:5^2 FPS:32



Figure 5.24. Constant Pen. 32-bit SHM Falloff:0.55 Penumbra Height:1.0 Bias:30 FPS:632



Figure 5.25. Constant Pen. 16-bit SHM Falloff:0.55 Penumbra Height:1.0 Bias:30 FPS:645



Figure 5.26. Varying Pen. 64-bit SHM Falloff:0.55 Penumbra Height:0.2 Bias:30 FPS:647



Figure 5.27. Varying Pen. 32-bit SHM Falloff:0.55 Penumbra Height:0.2 Bias:30 FPS:632

5.2.2. Comments

The locations and camera positions above are selected because they constitute problematic cases for shadow mapping and we aimed to show that our method does not suffer from the aliasing errors of shadow mapping. The first case, shown in the figures between 5.4 and 5.15, belongs to a location close to the height map surface. Since the surface is the shadow receiver and it is close to the camera near plane, the perspective aliasing increases rapidly. Additionally, terrain triangles in this location receive the sunlight with a wide angle, this increases the aliasing error caused by projection, since many pixels correspond to the same shadow map texel due to the projection of the surface triangles onto the light plane. We can see the aliasing error in its worst form in Figure 5.6, in the shape of the stair step shadow edges, where we have a single PSSM split, a shadow map resolution of 1024^2 and a PCF kernel size of 2^2. Each "step" corresponds to a single shadow map texel and therefore all surface corresponding to that texel becomes either shadowed or not. In Figure 5.7 we can see the effect of increasing the PCF kernel size to 5^2, it blurs the results obtained from different shadow map texels; such that it blurs the hard stair step edge in general. Despite the absence of hard jagged edges, the stair step can be clearly seen, this time blurred, in Figure 5.7. In figures 5.8 and 5.9 the shadow map resolution is increased to 2048^2. For a single split in Figure 5.8, we can see smaller stair steps compared to previous images, since the area of the shadow map texels reduced by $\frac{1}{4}$ due to increasing resolution. In Figure 5.9 we increase the split count to three. Since a single shadow map buffer has been focused on a smaller world height map part, we can see a degree of alleviation in the aliasing error in Figure 5.9, compared to 5.8. With shadow map resolution of 4096^2, we see more reduction of the aliasing error in Figure 5.10 with a split count two and PCF size 2^2. In the Figure 5.11, the PSSM parameters are set for the maximum quality; four splits, 4096^2 resolution and 5^2 PCF size. With this setting the aliasing error became finally invisible. The important point is that aliasing only became invisible for the current camera location, if the camera moved closer to the shadow edge, we could begin to see the jagged edge again, as the perspective error would get larger. In general, this causes the shadow edges to become instable and makes them prone to temporal aliasing artifacts if the camera moves, even under a

static light setting.

We can see in figures 5.12 and 5.13 the same location, rendered with our method and with the constant penumbra approach. Compared to the PSSM, our method generates alias free, smoothly softened shadow edges. Getting close to the caster surface would not generate stair steps caused by perspective aliasing in our case, since the way we generate shadows is alias free for a static light environment: We obtain pixel correct shadows for the height map's shadow volume approximation under the current light direction, as we have shown in the Section 4.6. Note that our shadow generation method is free from the perspective aliasing as well. Figures 5.14 and 5.15 show the results of the varying penumbra approach. We can notice the wider penumbra due to the large distance of the shadow casting terrain hill, compared to the constant penumbra approach; which is a more correct approximation of the reality. Between the figures 5.12 – 5.13 and 5.14 – 5.15 we can see that the quantized unsigned integer SHM textures do not exhibit any difference from their full precision, floating point counterparts. We did not encounter any artifacts from this quantization in our other experiments, neither. From these experiments, we deduced that we can safely use integer SHM textures at the half memory cost, without introducing any artifacts.

Figures between 5.16-5.27 belong to the second location. This time we focused on the shadow casted by the terrain on one of the Utah teapots. We again selected a camera location close to the shadow receiving surface and a surface which receives the sunlight with a relatively wide angle. The six figures between 5.18-5.23 belong to the PSSM, with same parameters as the previous location. We receive the same results under the same parameters: Increasing frustum splits and shadow mapping resolution alleviates the stair stepping aliasing artifact while increasing PCF kernel size blurs the jagged edge and hides it to a degree. But this time, we can clearly see that even with the PSSM parameters set for the maximum quality, the aliasing artifact is still visible in the Figure 5.23, unlike the previous location. Theoretically, we cannot eliminate the aliasing effect unless we use an unlimited resolution shadow map buffer and some degree of aliasing will be always there for shadow mapping. In figures 5.24 and 5.27, we see that our method works alias-free, again. Notice that this time the

varying penumbra approach generates smaller penumbrae, because the shadow casting hill is relatively close to the shadow receiving surface. Again, the quantized integer SHM textures did not exhibit any artifacts compared to floating point ones. Finally, in all these images, we can see the current frames-per-second (FPS) values. Notice how the performance of PSSM rapidly degrades as we set the parameters to larger values in order to increase quality. Under the static light environment, our method reaches nearly two times or higher speeds, compared to the fastest PSSM setting with the lowest parameters. Moreover, our method always produces better quality, alias free shadows.

We are not able to show the temporal aliasing artifacts of the both methods, since it cannot be observed on still images. But the in our experiments, we saw that the flickering artifact of the PSSM is more disturbing compared to our method. This flickering is especially distracting in PSSM when the PCF reconstruction is insufficiently blurs shadow edges.

5.3. Performance Comparisons

5.3.1. Results

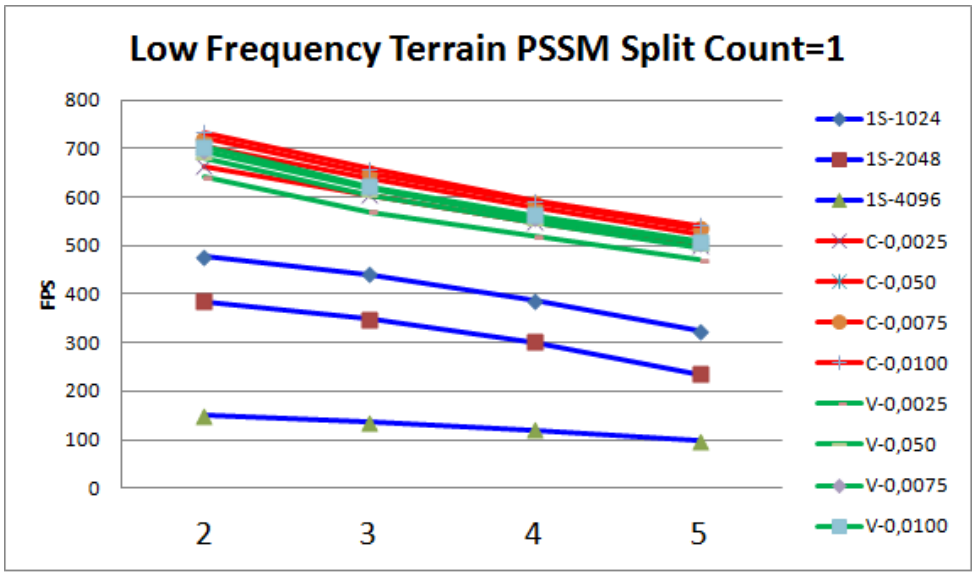


Figure 5.28. Results for 1 split PSSM vs. SHM on the low frequency terrain.

Table 5.1. Results for 1 split PSSM vs. SHM on the low frequency terrain.

Method	SM Res. Int. Length	2 PCF/K	3 PCF/K	4 PCF/K	5 PCF/K
1 Split	$\mathbf{1024^2}$	477.5	440.4	386.0	323.9
1 Split	$\mathbf{2048^2}$	384.4	347.6	300.7	235.7
1 Split	$\mathbf{4096^2}$	149.5	136.2	121.0	97.6
C. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	663.9	603.6	549.0	500.1
C. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	705.7	639.3	579.9	523.8
C. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	721.5	649.1	589.0	534.7
C. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	733.5	657.8	592.3	541.0
V. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	640.3	570.0	519.0	470.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	681.2	605.2	547.7	495.2
V. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	695.0	618.1	556.3	504.4
V. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	703.6	621.7	563.1	508.6

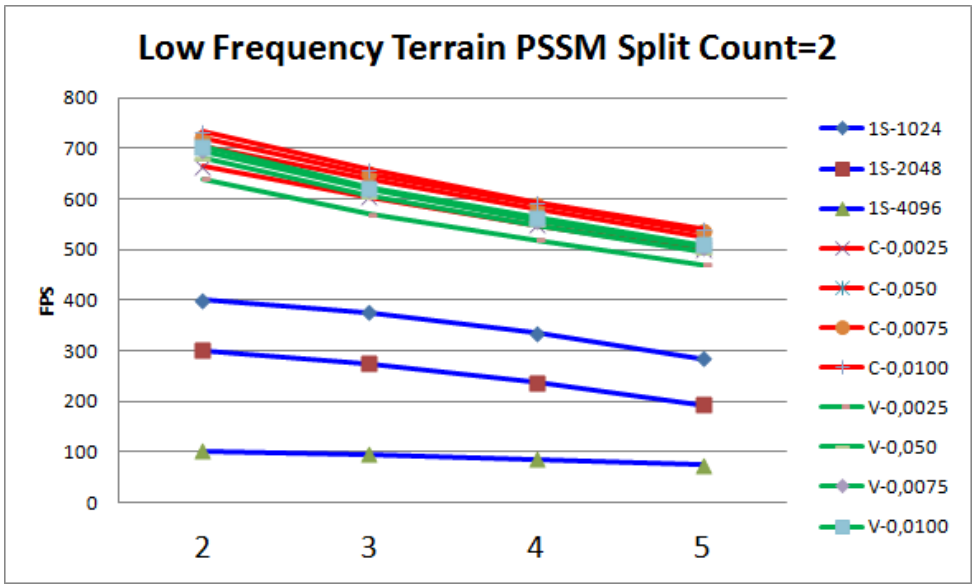


Figure 5.29. Results for 2 split PSSM vs. SHM on the low frequency terrain.

Table 5.2. Results for 2 split PSSM vs. SHM on the low frequency terrain.

Method	SM Res. Int. Length	2 PCF/K	3 PCF/K	4 PCF/K	5 PCF/K
2 Splits	$\mathbf{1024^2}$	399.7	376.3	335.7	284.4
2 Splits	$\mathbf{2048^2}$	301.1	274.9	237.4	193.3
2 Splits	$\mathbf{4096^2}$	101.4	95.2	85.9	73.7
C. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	663.9	603.6	549.0	500.1
C. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	705.7	639.3	579.9	523.8
C. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	721.5	649.1	589.0	534.7
C. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	733.5	657.8	592.3	541.0
V. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	640.3	570.0	519.0	470.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	681.2	605.2	547.7	495.2
V. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	695.0	618.1	556.3	504.4
V. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	703.6	621.7	563.1	508.6

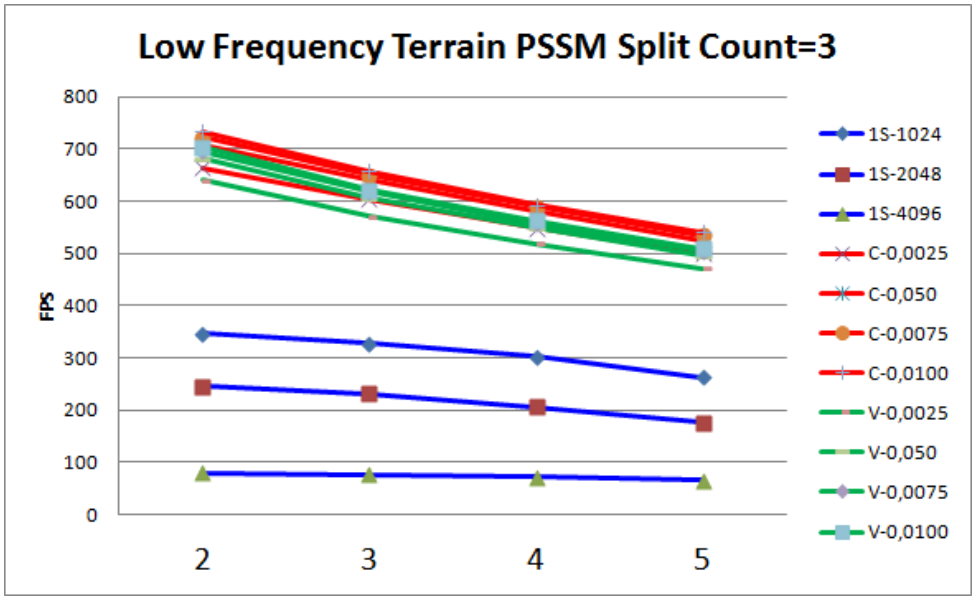


Figure 5.30. Results for 3 split PSSM vs. SHM on the low frequency terrain.

Table 5.3. Results for 3 split PSSM vs. SHM on the low frequency terrain.

Method	SM Res. Int. Length	2 PCF/K	3 PCF/K	4 PCF/K	5 PCF/K
3 Splits	$\mathbf{1024^2}$	346.2	327.2	301.4	262.4
3 Splits	$\mathbf{2048^2}$	245.4	231.0	206.1	174.9
3 Splits	$\mathbf{4096^2}$	79.5	75.9	72.1	64.5
C. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	663.9	603.6	549.0	500.1
C. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	705.7	639.3	579.9	523.8
C. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	721.5	649.1	589.0	534.7
C. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	733.5	657.8	592.3	541.0
V. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	640.3	570.0	519.0	470.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	681.2	605.2	547.7	495.2
V. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	695.0	618.1	556.3	504.4
V. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	703.6	621.7	563.1	508.6

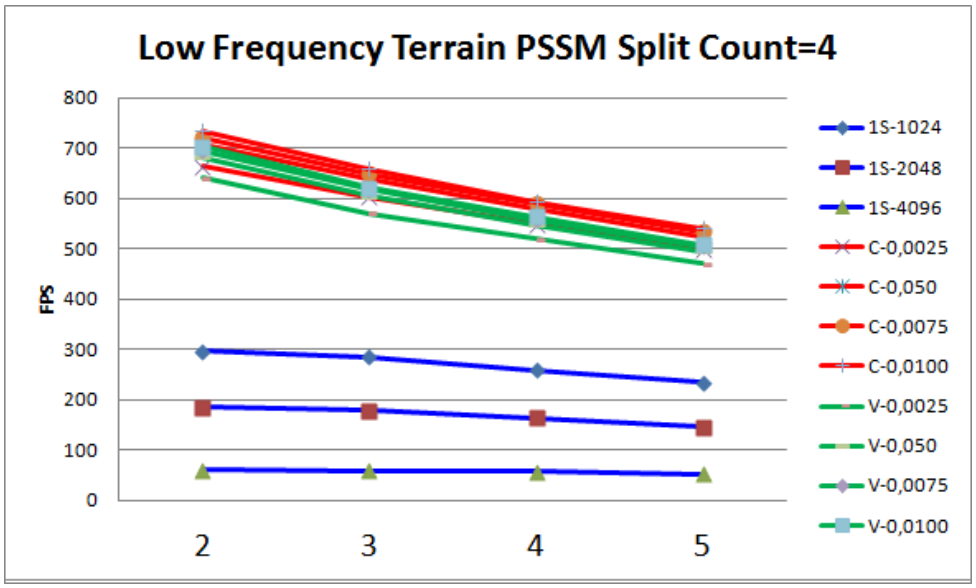


Figure 5.31. Results for 4 split PSSM vs. SHM on the low frequency terrain.

Table 5.4. Results for 4 split PSSM vs. SHM on the low frequency terrain.

Method	SM Res. Int. Length	2 PCF/K	3 PCF/K	4 PCF/K	5 PCF/K
4 Splits	$\mathbf{1024^2}$	296.0	284.2	259.5	234.4
4 Splits	$\mathbf{2048^2}$	185.0	179.2	164.4	146.1
4 Splits	$\mathbf{4096^2}$	61.3	59.0	57.3	52.8
C. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	663.9	603.6	549.0	500.1
C. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	705.7	639.3	579.9	523.8
C. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	721.5	649.1	589.0	534.7
C. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	733.5	657.8	592.3	541.0
V. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	640.3	570.0	519.0	470.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	681.2	605.2	547.7	495.2
V. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	695.0	618.1	556.3	504.4
V. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	703.6	621.7	563.1	508.6

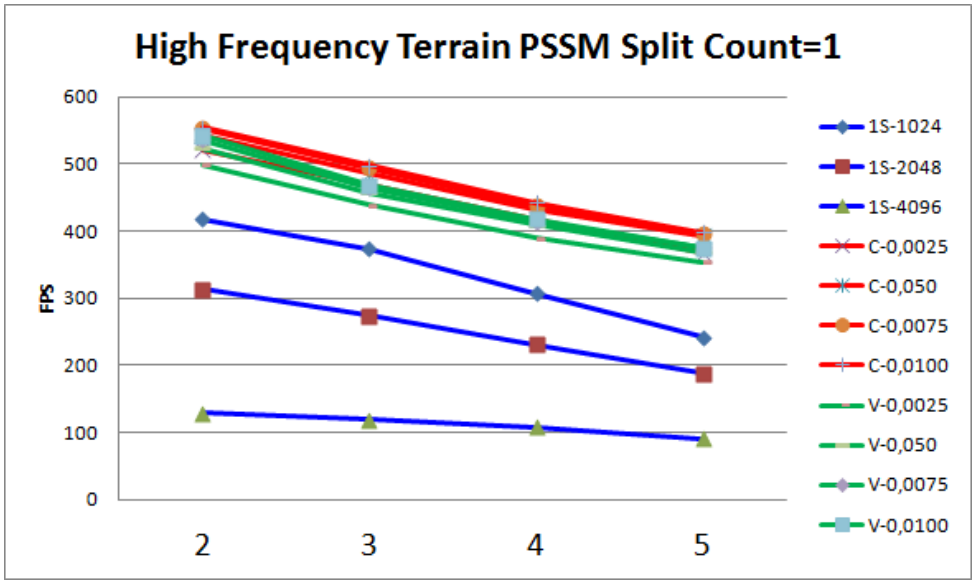


Figure 5.32. Results for 1 split PSSM vs. SHM on the high frequency terrain.

Table 5.5. Results for 1 split PSSM vs. SHM on the high frequency terrain.

Method	**SM Res. Int. Length**	**2 PCF/K**	**3 PCF/K**	**4 PCF/K**	**5 PCF/K**
1 Split	$\mathbf{1024^2}$	418.2	372.9	307.7	241.9
1 Split	$\mathbf{2048^2}$	313.6	274.1	230.5	187.4
1 Split	$\mathbf{4096^2}$	128.5	118.9	108.4	90.3
C. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	521.5	468.5	418.1	367.4
C. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	542.9	484.9	432.4	393.0
C. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	552.7	494.2	438.0	395.8
C. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	554.5	497.6	441.8	398.5
V. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	449.2	438.5	389.5	353.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	523.9	457.1	409.2	369.1
V. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	535.6	464.7	413.7	374.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	541.8	468.4	417.6	375.8

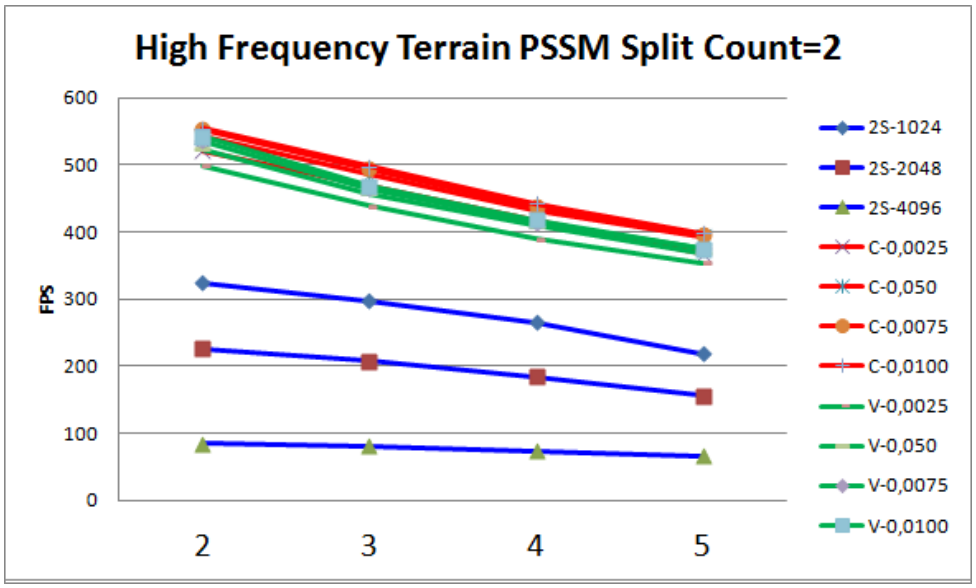


Figure 5.33. Results for 2 split PSSM vs. SHM on the high frequency terrain.

Table 5.6. Results for 2 split PSSM vs. SHM on the high frequency terrain.

Method	**SM Res. Int. Length**	**2 PCF/K**	**3 PCF/K**	**4 PCF/K**	**5 PCF/K**
2 Splits	$\mathbf{1024^2}$	324.2	297.8	265.6	218.6
2 Splits	$\mathbf{2048^2}$	226.6	207.3	183.8	155.4
2 Splits	$\mathbf{4096^2}$	84.8	80.4	74.2	66.2
C. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	521.5	468.5	418.1	367.4
C. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	542.9	484.9	432.4	393.0
C. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	552.7	494.2	438.0	395.8
C. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	554.5	497.6	441.8	398.5
V. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	449.2	438.5	389.5	353.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	523.9	457.1	409.2	369.1
V. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	535.6	464.7	413.7	374.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	541.8	468.4	417.6	375.8

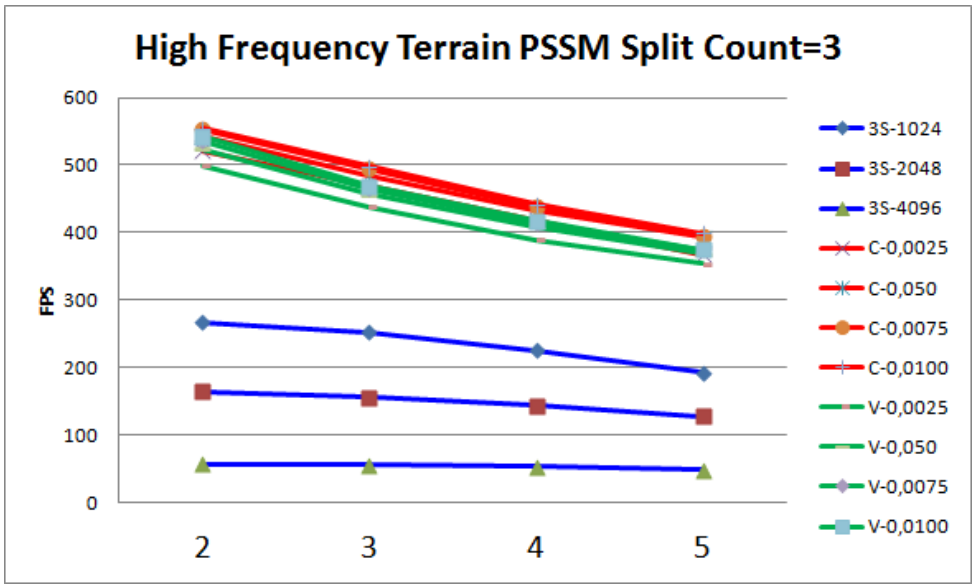


Figure 5.34. Results for 3 split PSSM vs. SHM on the high frequency terrain.

Table 5.7. Results for 3 split PSSM vs. SHM on the high frequency terrain.

Method	**SM Res. Int. Length**	**2 PCF/K**	**3 PCF/K**	**4 PCF/K**	**5 PCF/K**
3 Splits	$\mathbf{1024^2}$	266.6	251.8	226.0	191.9
3 Splits	$\mathbf{2048^2}$	164.2	156.3	143.1	127.5
3 Splits	$\mathbf{4096^2}$	56.5	55.0	52.6	48.5
C. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	521.5	468.5	418.1	367.4
C. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	542.9	484.9	432.4	393.0
C. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	552.7	494.2	438.0	395.8
C. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	554.5	497.6	441.8	398.5
V. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	449.2	438.5	389.5	353.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	523.9	457.1	409.2	369.1
V. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	535.6	464.7	413.7	374.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	541.8	468.4	417.6	375.8

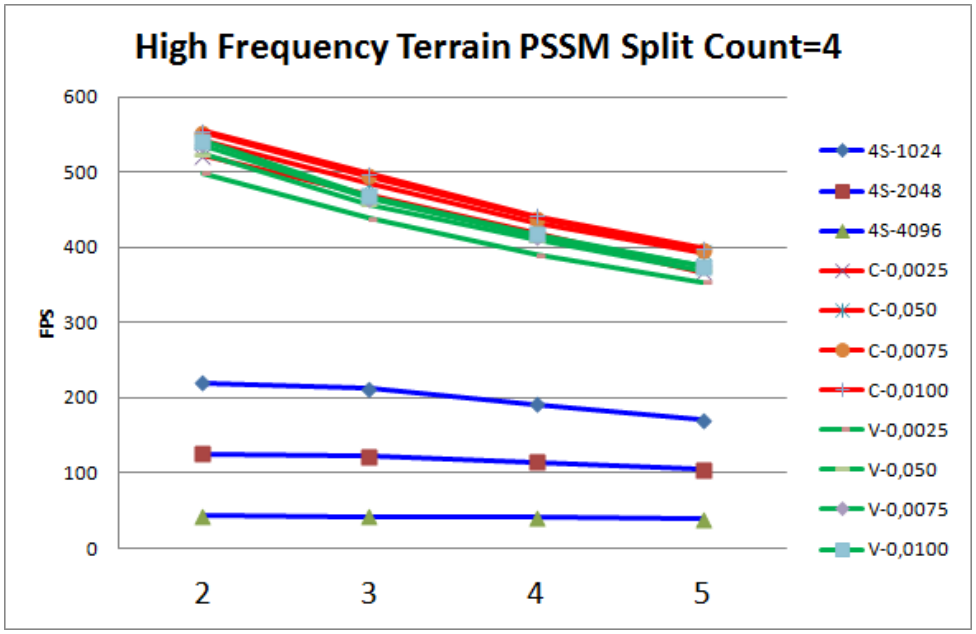


Figure 5.35. Results for 4 split PSSM vs. SHM on the high frequency terrain.

Table 5.8. Results for 4 split PSSM vs. SHM on the high frequency terrain.

Method	SM Res. Int. Length	2 PCF/K	3 PCF/K	4 PCF/K	5 PCF/K
4 Splits	$\mathbf{1024^2}$	220.3	212.1	191.3	170.8
4 Splits	$\mathbf{2048^2}$	125.8	122.3	114.2	104.8
4 Splits	$\mathbf{4096^2}$	42.9	42.2	40.8	38.5
C. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	521.5	468.5	418.1	367.4
C. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	542.9	484.9	432.4	393.0
C. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	552.7	494.2	438.0	395.8
C. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	554.5	497.6	441.8	398.5
V. Pen.	$\mathbf{\Delta\alpha = 0.0025}$	449.2	438.5	389.5	353.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0050}$	523.9	457.1	409.2	369.1
V. Pen.	$\mathbf{\Delta\alpha = 0.0075}$	535.6	464.7	413.7	374.7
V. Pen.	$\mathbf{\Delta\alpha = 0.0100}$	541.8	468.4	417.6	375.8

5.3.2. Comments

In the graphics above, the x-axis shows the PCF kernel size for the shadow mapping and the temporal filter length for our algorithm. The results we obtain both for high frequency and low frequency terrains indicate that our SHM based method outperform PSSM in all test environments we have used. The main reason of our better performance is our algorithm's camera independence due to the alignment of the SHM with the height map and our ability of caching the results over a number of frames. In contrast to our method, PSSM needs to re-render the shadow map each time the camera moves, since the shadow map is aligned with light plane.

We can clearly see as the number frustum splits increase, the performance of the PSSM drops. This is an expected result since the geometry load to be processed increase with each split. Moreover, due to the large polygons drawn into the shadow map buffers, the fill rate begins to become another bottleneck as well. As can be clearly seen from the graphs, the most dramatic source of performance hit for shadow mapping is the shadow map resolution. This is an expected result as well, since with shadow map buffers with dimensions two times higher, the algorithm needs to clear and fill four times more pixels. In this case, the fill rate quickly becomes the main source of bottleneck; as the raster operators become overwhelmed by the amount of pixels needed to be drawn and as the result, the whole graphics pipeline stalls. In order to reduce depth bias errors, we render both sides of the height map and the tea pots into the shadow map, disabling the depth testing. This worsens the fill rate bottleneck as pixels may get overdrawn multiple times in a single scene. The performance loss caused by the shadow map resolution becomes more critical with the increased PSSM split count as each new split introduces a new shadow map buffer to render into. The effect of the increasing PCF kernel size on the shadow mapping performance is clearly visible, as well. Since PCF sampling works on the pixels which are drawn into the frame buffer and not on the shadow map image, it is not as much susceptible to the increasing split count like the shadow map resolution. An interesting fact which can be observed is that the PCF kernel size affects the performance more for lower shadow map resolutions and frustum split counts. For such settings, the fill rate and geometry

processing load is rather low and the texture bandwidth consumption due to hardware PCF samples begin to stall the pipeline. As the amount of geometry which is drawn into the shadow map buffers and the fill rate consumption increase with more splits and shadow map resolution, the bottleneck quickly shifts to the fill rate. In such settings the performance hit caused by larger PCF kernels become less important. This is a fact observed by Zhang *et al.* as well [66]. They report that the PCF cost becomes insignificant as the geometry load increases.

Our SHM based method's performance changes both with the temporal filter and interval length. It seen from the graphs that increasing filter length affects the performance more. A single increase in the filter size introduces two new texture samples. In this aspect, our temporal filter is more efficient than the PCF, since our sampling cost grows with $O(N)$ where the PCF kernel grows with $O(N^2)$. The small performance drop resulting from smaller interval lengths can be explained with the increased number of CUDA kernel invocations in the course of $500ms$ test runs. An interval length of $N = 25$ frames means four times more CUDA calls compared to $N = 100$. Anyhow, the amount performance drops caused by this is very small. We can see that constant penumbra SHM is slightly faster than the varying penumbra one. This is caused by the increased cost of the texture reads; the SHM textures for the varying penumbra approach contain two channels both for the shadow and penumbra heights. So we will make a 64-bit texture read instead of 32-bit and this increases the texture bandwidth consumption. (Note that we make 32-bit reads instead of 16-bit, when using integer SHM textures).

The effect of different height map geometries can be seen between the performance graphics of high frequency and low frequency height maps. The performance is higher for both of the algorithms in the low frequency height map. Since the low frequency height map contains less elevation details, its mesh blocks switch to coarser LOD levels much faster, compared to the high frequency one. Therefore, the amount of polygons sent to the pipeline reduces and the rendering time decreases. This effect is totally independent from both of the shadowing algorithms.

6. CONCLUSION AND FUTURE WORK

In thesis, we presented our work on the problem of terrain shadowing under directional sunlight. We designed a shadowing algorithm, specialized for the 2.5D height map geometry, based on the idea of shadow height maps. We implemented it mostly on the GPU in parallel, using the CUDA framework. Our most serious problem was the temporal aliasing errors which cause flickering of the shadow edges in a moving sunlight environment. By using methods like defining an interpolation interval and designing a temporal low pass filter, we managed to alleviate it such that it reduces to negligible or acceptable levels. In our experiments, we compared our algorithm with a state-of-the-art shadow mapping variant, Parallel Split Shadow Mapping (PSSM) and we showed that our algorithm does not suffer from the perspective and projection aliasing errors, the main problem of the shadow mapping. Additionally, we saw that the temporal aliasing errors of our algorithm are much more tolerable compared to the temporal aliasing errors of the PSSM. The performance results turned out to be superior as well. Our method outperformed its PSSM counterpart with equivalent parameters in all tests we have conducted. From these results, we can say that we reached our aim of developing a reliable terrain shadowing algorithm, which can compete or even outperform the current solutions in this particular area.

In addition to the development of a new terrain shadowing algorithm, we solved two critical problems of a popular terrain LOD algorithm, GeoMipmapping, which are namely the edge cracks between blocks of different resolutions and the vertex popping effect observed at LOD switches. Our index buffer based edge stitching and accompanying GeoMorphing solutions can be generalized for other tile-based DLOD algorithms. In general, we developed a fully parametrical terrain rendering system with a source code over 20000 lines, developed with C++.

Finally, we compiled a detailed review of the dated and state-of-the-art hard and approximated soft shadow techniques found in the literature, in the Chapter 3 of this thesis. This survey can be useful for readers which are interested in this area.

As future work, we think about extending our shadowing algorithm to support out-of-core terrain rendering. Out-of-core terrain rendering involves the visualization of very large height map data, which cannot be completely stored in the main memory, with a paging system which swaps height map chunks in and out from the hard disk, as needed. This can be a particularly interesting and challenging extension for our algorithm, since usually, the new terrain data is swapped in as the result of camera movements. This will introduce the camera as a new parameter to our algorithm; since we would need to calculate the SHM for the newly loaded height map data. An idea could be making use of the old SHM data which was already stored in the video memory: The newly introduced elevation data could be already receiving shadows from the existing height map geometry. So, we can develop a system which updates the SHM only incrementally, as new height map data is loaded.

Another research direction could be combining shadow mapping with our method in order to support shadows cast by the objects on the terrain and on themselves. We can devise many optimizations for the shadow mapping in this case both in quality and performance aspects, since it will only focus on the shadowing of arbitrary objects. Without doing any new, involved optimizations, this is actually already an improvement by itself, since the limited resolution of the shadow map will be focused on objects which are much smaller than the terrain, in world space measurements. This means a much more efficient distribution of the shadow map resolution. The needed PCF kernel size would be much smaller as well, since a more efficiently distributed shadow map resolution causes less aliasing artifacts.

We designed our algorithm with the case of the directional sunlight shadows cast by the landscape in mind. Our method can be easily extended to support shadows caused by the point and spot lights, as well. The 1D rhombic compression and decompression techniques used by Robertson in [88] can be applied to our algorithm and we can easily modify our ray casting CUDA kernels to support shadowing from such kind of lights.

Finally, we think about improving our low pass filter approach further. For example, a good research direction would be the development of an incremental, temporal exponential smoothing filter which reduces temporal aliasing. Such a filter could incrementally update an accumulation SHM texture, after a new SHM texture for a new light direction has been generated. Such a filter could further improve our performance while reducing our memory consumption.

Our terrain shadowing method can be used in many applications like strategy games, MMORPGs or flight simulators where a large landscape is needed to be visualized. In our experience, we saw that many current games do not use shadows for terrains and the ones which do use static, precalculated shadows for landscapes, which do not change during the runtime of the program and cannot be cast on other game objects. This is mostly due the fact that dynamic terrain shadowing is a very problematic issue and it is hard to obtain in a cheap and visually plausible way, as we explained in the whole of this thesis. We would like to see that our terrain shadowing algorithm finds its usage in such applications.

REFERENCES

1. Vogelsang, S., *Progressive Geometrical Mipmapping – A Runtime-editable Terrain System*, Master's Thesis, Georg-August-Universitat, 2007.

2. Boer, W. H. D., *Fast Terrain Rendering Using Geometrical MipMapping*, 2000, `http://www.flipcode.com/articles/article_geomipmaps.pdf`, accessed at May 2012.

3. Möller, T. and E. Haines, *Real-Time Rendering*, A. K. Peters Limited, 2002.

4. Woo, A., P. Poulin and A. Fournier, "A Survey of Shadow Algorithms", *IEEE Computer Graphics and Applications*, Vol. 10, No. 6, pp. 13–32, 1990.

5. Lloyd, B., J. Wendt, N. K. Govindaraju and D. Manocha, "CC Shadow Volumes", *In Proceedings of the Eurographics Symposium on Rendering*, 2004.

6. Eisemann, E., U. Assarsson, M. Schwarz and M. Wimmer, "Casting Shadows in Real Time", *SIGGGRAPH Asia Course Notes*, 2009.

7. Cruz, D. D., "Shadow Maps: A Survey", *Technical Report - Universidade do Minho*, 2008.

8. Scherzer, D., M. Wimmer and R. Werner, "A Survey of Real-Time Hard Shadow Mapping Methods", *Computer Graphics Forum*, Vol. 30, No. 1, pp. 169–186, 2011.

9. Reeves, W. T., D. H. Salesin and R. L. Cook, "Rendering Antialiased Shadows with Depth Maps", *In Computer Graphics (SIGGRAPH '87 Proceedings) (July 1987)*, Vol. 21, No. 4, pp. 283–291, 1987.

10. Fernando, R., "Percentage-Closer Soft Shadows", *In ACM SIGGRAPH 2005 Sketches*, 2005.

11. Synder, J. and D. Nowrouzezahrai, "Fast Soft Self-Shadowing on Dynamic Height Fields", *Computer Graphics Forum*, Vol. 28, No. 4, pp. 1131–1139, 2009.

12. Oat, C. and P. Sander, "Ambient Aperture Lighting", *In ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, p. 61–64, 2007.

13. Chang, C.-F., B.-Q. Lin, Y.-C. Chen and Y.-F. Chiu, "Real-Time Soft Shadow for Displacement Mapped Surfaces", *IEEE International Conference on Multimedia & Expo (ICME) 2009*, pp. 1254–1257, 2009.

14. Timonen, V. and J. Westerholm, "Scalable Height Field Self-Shadowing", *In Proceedings of the Eurographics 2010 Conference*, Vol. 29, No. 2, pp. 723–731, 2010.

15. Tatarchuk, N., "Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows", *In Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (SI3D '06)*, pp. 63–69, 2006.

16. Loscos, C., F. Tecchia and Y. Chrysanthou, "Real-time Shadows for Animated Crowds in Virtual Cities", *In Proceedings of the ACM symposium on Virtual Reality Software and Technology, ACM Press*, p. 85–92, 2001.

17. Lee, S. H. and C. H. Kim, "Real-time Soft Shadowing of Dynamic Height Map Using a Shadow Height Map", *Korea Computer Graphics Society*, Vol. 14, No. 1, p. 11–16, 2008.

18. Microsoft, *Microsoft Developer Network (MSDN), Direct3D Architecture (Direct3D 9)*, 2012, `http://msdn.microsoft.com/en-us/library/windows/desktop/bb219679(v=vs.85).aspx/`, accessed at April 2012.

19. NVIDIA, *NVIDIA CUDA C Programming Guide*, 2012, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`, accessed at April 2012.

20. Ludvigsen, H. and A. C. Elster, "Real-Time Ray Tracing Using Nvidia OptiX", *Short Paper - Eurographics 2010*, 2010.

21. Phong, B.-T., "Illumination for Computer Generated Pictures", *Communications of the ACM (CACM)*, Vol. 18, No. 6, pp. 313–317, 1975.

22. Dimitrov, R., *Cascaded Shadow Maps NVIDIA Technical Document*, 2007, `http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf`, accessed at March 2012.

23. Ebert, D. S., F. K. Musgrave, D. Peachey, K. Perlin and S. Worley, *Texturing & Modeling: A Procedural Approach*, AP Professional, 1998.

24. Elias, H., *Perlin Noise*, 2000, `http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`, accessed at May 2012.

25. Lindstorm, P. and D. Koller, "Real-time Continuous Level of Detail Rendering of Height Fields", *In International Conference on Computer Graphics and Interactive Techniques - Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 109–118, 1996.

26. Duchaineau, M. and M. Wolinsky, "ROAMing Terrain: Real-time Optimally Adapting Meshes", *In IEEE Visualization - Proceedings of the 8th conference on Visualization '97*, pp. 81–88, 1997.

27. Röttger, S., W. Heidrich, P. Slussallek and H.-P. Seidel, "Real-Time Generation of Continuous Levels of Detail for Height Fields", *In Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization (Feb. 9-13, Plzen - Bory, Czech Republic)*, pp. 315–322, 1998.

28. Losasso, F. and H. Hoppe, "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids", *In International Conference on Computer Graphics and Interactive Techniques - ACM SIGGRAPH 2004 Papers*, pp. 769–776, 2004.

29. Ulrich, T., "Rendering Massive Terrain Using Chunked Level of Detail Control", *SIGGRAPH 2002 Course Notes*, 2002.

30. Pajarola, R. and E. Gobbetti, "Survey of Semi-Regular Multiresolution Models for Interactive Terrain Rendering", *The Visual Computer*, Vol. 23, No. 8, pp. 583–605, 2007.

31. Bloom, C., *Terrain Texture Compositing by Blending in the Frame-Buffer*, 2000, `http://www.cbloom.com/3d/techdocs/splatting.txt`, accessed at May 2012.

32. Ferraris, J. and C. Gatzidis, "A Rule-based Approach to 3D Terrain Generation via Texture Splatting", *In the Proceedings of the International Conference on Advances in Computer Entertaninment Technology (ACE'09)*, pp. 407–408, 2009.

33. Luna, F. D., *Introduction to 3D Game Programming With DirecX 9.0c A Shader Approach*, Wordware Publishing Inc, 2006.

34. Waveren, J. M. P. V. and I. Castano, *Real-Time Normal Map DXT Compression – NVIDIA Whitepaper*, 2008, `http://developer.download.nvidia.com/whitepapers/2008/real-time-normal-map-dxt-compression.pdf`, accessed at May 2012.

35. Portalés, J. G., *Techniques For Improving The Visualization Of Natural Scenes*, Ph.D. Thesis, Universitat Jaume I, Portugal, 2011.

36. Wanger, L., "The Effect of Shadow Quality on the Perception Spatial Relationships in Computer Generated Imagery", *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, pp. 39–42, 1992.

37. Kajiya, J. T., "The Rendering Equation", *ACM SIGGRAPH Computer Graphics*, Vol. 20, No. 4, p. 143–150, 1986.

38. Watt, A. and M. Watt, *Advanced Animation and Rendering Techniques-Theory*

and Practice, Addison-Wesley, 1992.

39. Crow, F. C., "Shadow Algorithms for Computer Graphics", *ACM SIGGRAPH Computer Graphics*, Vol. 11, No. 2, pp. 242–248, 1977.

40. Williams, L., "Casting Curved Shadows on Curved Surfaces", *ACM SIGGRAPH Computer Graphics*, Vol. 12, No. 3, pp. 270–274, 1978.

41. Blinn, J., "Me and My (Fake) Shadow", *IEEE Computer Graphics and Applications*, Vol. 8, No. 1, pp. 82–86, 1988.

42. Tessman, T., "Casting Shadows on Flat Surfaces", *Iris Universe*, pp. 16–19, 1989.

43. Heckbert, P. S. and M. Herf, "Simulating Soft Shadows with Graphics Hardware", *Technical Report CMU-CS-97-104, Carnegie Mellon University*, 1997.

44. Heckbert, P. S. and M. Herf, "Fast Soft Shadows", *Sketch in Visual Proceedings (SIGGRAPH '96)*, p. 145, 1996.

45. Gooch, B., P. J. Sloan, A. Gooch, P. Shirley and R. Riesenfield, "Interactive Technical Illustration", *Proceedings of 1999 Symposium on Interactive 3D Graphics*, pp. 31–38, 1999.

46. McReynolds, T., D. Blythe, B. Grantham and S. Nelson, "Advanced Graphics Programming Techniques Using OpenGL", *Course Notes at SIGGRAPH '99*, 1999.

47. Nguyen, H. H., "Casting Shadows on Volumes", *Game Developer*, Vol. 6, No. 3, pp. 44–53, 1999.

48. Segal, M., C. Korobkin, R. van Widenfelt, J. Foran and P. Haeberli, "Fast Shadows and Lighting Effects Using Texture Mapping", *ACM SIGGRAPH Computer Graphics*, Vol. 26, No. 2, pp. 249–252, 1992.

49. Heidmann, T., "Real Shadows, Real Time", *Iris Universe*, p. 23–31, 1991.

50. Carmack, J., "Development of the Game Doom 3", *Id Software*, 2000.

51. Bilodeau, W. and M. Songy, "Real Time Shadows", *Creative Labs Inc. Sponsored Game Developer Conferences*, 1999.

52. Hornus, S., J. Hoberock, S. Lefebvre and J. C. Hart, "ZP+: Correct Z-Pass Stencil Shadows", *In Proceedings of I3D (ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games)*, 2005.

53. McCool, M. D., "Shadow Volume Reconstruction From Depth Maps", *ACM Transactions on Graphics*, Vol. 1, No. 19, pp. 1–26, 2000.

54. Laine, S., "Split-Plane Shadow Volumes", *In Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2005.

55. Scherzer, D., "Robust Shadow Maps for Large Environments", *Proceedings of the 2005 Central European Seminar on Computer Graphics(CESCG)*, 2005.

56. Hourcade, J.-C. and A. Nicolas, "Algorithms For Antialiased Cast Shadows", *Computer & Graphics*, Vol. 9, No. 3, p. 259–265, 1985.

57. Woo, A., "The Shadow Depth Map Revisited", *Graphics Gems III*, p. 338–342, 1992.

58. Wang, Y. and S. Molnar, "Second-Depth Shadow Mapping", *Technical Report - University of North Carolina at Chapel Hill*, 1994.

59. Weiskopf, D. and T. Ertl, "Shadow Mapping Based on Dual Depth Layers", *In Short Paper Eurographics*, p. 53–60, 2003.

60. Tuft, D., *Common Techniques to Improve Shadow Depth Maps - Microsoft Technical Document*, 2011, `http://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx`, accessed at April 2012.

61. Stamminger, M. and G. Drettakis, "Perspective Shadow Maps", *In ACM Transactions on Graphics (Proceedings of SIGGRAPH) (July 2002)*, Vol. 3, No. 5, p. 557–562, 2002.

62. Lloyd, B., *Logarithmic Perspective Shadow Maps*, Ph.D. Thesis, University of North Carolina, Chapel Hill, 2007.

63. Brabec, S., T. Annen and H.-P. Seidel, "Practical Shadow Mapping", *Journal of Graphics Tools: JGT 7*, Vol. 4, No. 2, p. 9–18, 2002.

64. Martin, T. and T.-S. Tan, "Anti-Aliasing and Continuity with Trapezoidal Shadow Maps", *In Proceedings of Eurographics Symposium on Rendering 2004*, p. 153–160, 2004.

65. Wimmer, M., D. Scherzer and W. Purgathofer, "Light Space Perspective Shadow Maps", *In Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering)*, 2004.

66. Zhang, F., H. Sun, L. Xu and L. K. Lun, "Parallel-Split Shadow Maps for Large-scale Virtual Environments", *In Proceedings of Virtual Reality Continuum and Its Applications 2006*, Vol. 7, No. 8, pp. 311–318, 2006.

67. Engel, W., *ShaderX5: Advanced Rendering Techniques - Cascaded Shadow Maps*, Charles River Media, Inc., 2007.

68. Mittring, M., "Finding Next Gen – CryEngine 2", *In Advanced Real-Time Rendering in 3D Graphics and Games Course – SIGGRAPH*, 2007.

69. Tuft, D., *Cascaded Shadow Maps - Microsoft Technical Document*, 2011, `http://msdn.microsoft.com/en-us/library/windows/desktop/ee416307(v=vs.85).aspx`, accessed at April 2012.

70. Fernando, R., S. Fernandez, K. Bala and D. P. Greenberg, "Adaptive Shadow

Maps", *In SIGGRAPH 2001 Conference Proceedings (Aug. 2001), Fiume E., (Ed.), Annual Conference Series, ACM SIGGRAPH*, 2001.

71. Giegl, M. and M. Wimmer, "Queried Virtual Shadow Maps", *In Proceedings of I3D (ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games)*, 2007.

72. Arvo, J., "Tiled Shadow Maps", *In CGI '04: Proceedings of the Computer Graphics International (CGI'04)*, 2004.

73. Giegl, M. and M. Wimmer, "Fitted Virtual Shadow Maps", *In Proceedings of GI (Graphics Interface) (2007), Canadian Human-Computer Communications Society*, 2007.

74. Johnson, G. S., W. R. Mark and C. A. Burns, "The Irregular Z-Buffer and its Application to Shadow Mapping", *Research paper, The University of Texas at Austin*, 2004.

75. Scherzer, D., S. Jeschke and M. Wimmer, "Pixelcorrect Shadow Maps with Temporal Reprojection and Shadow Test Confidence", *In Rendering Techniques 2007 (Proceedings of Eurographics Symposium on Rendering)*, 2007.

76. Donnely, W. and A. Lauritzen, "Variance Shadow Maps", *In I3D '06: Proceedings of the 2006 Symposium on Interactive3D Graphics and Games (New York, NY, USA, 2006), ACM*, pp. 161–165, 2006.

77. Lauritzen, A. and M. McCool, "Layered Variance Shadow Maps", *In GI '08: Proceedings of Graphics Interface 2008 (Toronto, Ont., Canada, Canada, 2008), Canadian Information Processing Society*, pp. 139–146, 2008.

78. Annen, T., T. Mertens, P. Bekaert, H.-P. Seidel and J. Kautz, "Convolution Shadow Maps", *In Rendering Techniques 2007: Eurographics Symposium on Rendering (Grenoble, France, June 2007)*, 2007.

79. Annen, T., T. Mertens, H.-P. Seidel, E. Flerackers and J. Kautz, "Exponential Shadow Maps", *In GI'08: Proceedings of Graphics Interface 2008 (Toronto, Ont., Canada, Canada, 2008), Canadian Information Processing Society*, p. 155–161, 2008.

80. Hasenfratz, J.-M., M. Lapierre, N. Holzschuch and F. Sillion, "A Survey of Real-time Soft Shadowing Algorithms", *Computer Graphics Forum*, Vol. 22, No. 4, pp. 753–774, 2003.

81. Sakalauskas, T., "Hybrid Terrain Shadow Ray Casting", *Journal of WSCG, The International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2008.

82. Marghidanu, M., *Fast Computation of Terrain Shadow Maps*, 2002, `http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/fast-computation-of-terrain-shadow-maps-r817`, accessed at April 2012.

83. Max, N. L., "Horizon Mapping: Shadows for Bump-mapped Surfaces", *The Visual Computer*, Vol. 4, No. 2, pp. 109–117, 1988.

84. Sloan, P.-P. and M. F. Cohen, "Interactive Horizon Mapping", *In Proceedings of the Eurographics Workshop on Rendering Techniques 2000, Springer-Verlag*, p. 281–286, 2000.

85. Stewart, J., "Fast Horizon Computation at All Points of a Terrain with Visibility and Shading Applications", *IEEE Transactions on Visualization and Computer Graphics 4*, Vol. 4, No. 1, pp. 82–93, 1998.

86. McGuire, M. and P. G. Sibley, "A Heightfield on an Isometric Grid", *In Sketch Presented at the 31st Annual Conference on Computer Graphics and Interactive Techniques*, 2004.

87. Ratti, C. and P. Richens, "Raster Analysis of Urban Form", *Environment and Planning B: Planning and Design*, Vol. 31, No. 1, p. 297–309, 2004.

88. Robertson, P. K., "Spatial Transformations for Rapid Scan-line Surface Shadowing", *IEEE Computer Graphics and Applications*, Vol. 9, No. 2, pp. 30–38, 1989.

89. KhronosGroup, *The Open Graphics Library: OpenGL*, 2012, `http://www.opengl.org/`, accessed at April 2012.

90. Buntin, S. and M. Stamminger, "Instanced Shadow Maps", *In Proceedings of CGAMES'07*, pp. 135–142, 2007.