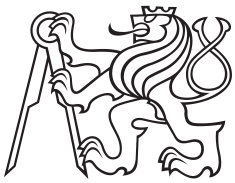
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Bc. Matěj Sakmary

Supervisor: doc. Ing. Jiří Bittner Ph.D.
Field of study: Open informatics
Subfield: Computer Graphics
May 2025

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Sakmary Matěj** Personal ID number: **487342**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Graphics**

II. Master’s thesis details

Master’s thesis title in English:

Virtual Shadow Maps

Master’s thesis title in Czech:

Virtuální stínové mapy

Name and workplace of master’s thesis supervisor:

doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **10.09.2024** Deadline for master's thesis submission: _____

Assignment valid until: **15.02.2026**

_____
Head of department’s signature

_____
prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the master’s thesis, the author must state the names of consultants and include a list of references.

_____
Date of assignment receipt

_____
Student’s signature

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Sakmary Matěj** Personal ID number: **487342**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Graphics**

II. Master’s thesis details

Master’s thesis title in English:

Virtual Shadow Maps

Master’s thesis title in Czech:

Virtuální stínové mapy

Guidelines:

Review existing methods for rendering shadows using shadow maps. Focus on virtual shadow map techniques that virtualize the shadow map by decoupling it from its storage [1,2]. Implement the virtual shadow map method using C++ and the Vulkan API. The implementation will support directional, point, and spotlights and allow the rendering of shadows for volumetric effects. Focus on optimizing rendering performance by exploiting visibility culling methods for rendering shadow maps [3, 4]. Perform a thorough measurement of the rendering performance and compare the quality of the rendered shadows with the reference generated by the path tracing on at least two complex scenes. Identify the strong and weak points of the implementation and discuss a possible combination of virtual shadow maps and ray-traced shadows.

Bibliography / sources:

[1] Unreal Engine 5.4 documentation. Virtual Shadow Maps. https://dev.epicgames.com/documentation/en-us/unreal-engine/virtual-shadow-maps-in-unreal-engine. Accessed 9/9/2024.
[2] Giegl, M., Wimmer, M. (2007). Queried virtual shadow maps. In Proceedings of the 2007 symposium on Interactive 3D graphics and games (pp. 65-72).
[3] Bittner, J., Mattausch, O., Silvennoinen, A., Wimmer, M. (2011). Shadow caster culling for efficient shadow mapping. In Symposium on Interactive 3D Graphics and Games (pp. 81-88).
[4] Greene, N., Kass, M., Miller, G. (1993). Hierarchical Z-buffer visibility. In Proceedings of the 20th annual conference on Computer graphics and interactive techniques (pp. 231-238).
[5] Boksansky, J., Wimmer, M., Bittner, J. (2019). Ray traced shadows: maintaining real-time frame rates. Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs, 159-182.
[6] Eisemann, E., Schwarz, M., Assarsson, U., Wimmer, M. (2011). Real-time shadows. CRC Press.

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s): Sakmary Matěj
Personal number: 487342
Programme name: Open Informatics

declare that I have elaborated the master's thesis entitled

Virtual Shadow Maps

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I did not use any artificial intelligence tools during the preparation and writing of my thesis. I am aware of the consequences if manifestly undeclared use of such tools is determined in the elaboration of any part of my thesis.

In Prague on 22.05.2025

Bc. Matěj Sakmary
.....
student's signature

Acknowledgements

I would like to thank to everyone who supported me throughout the creation of this work. Specifically, I would like to doc. Ing. Jiří Bittner Ph.D., my supervisor, for offering advice and guidance throughout the whole process. I would also like to thank to Patrick Ahrens for his valuable insights and expertise which were invaluable during the algorithm's design.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Abstract

In this work I present Virtual Shadow Maps (VSMs), a method for rendering shadow maps with greatly improved quality and memory efficiency compared to current methods. I separate logical address space from its physical backing by splitting each shadow map into a set of virtual pages. This achieves the appearance of a large contiguous memory without the need to reserve backing physical memory. For each frame, I find the set of visible pages by analyzing the depth buffer. Each visible page is assigned physical memory allocated from a designated memory pool. To efficiently fill visible pages with shadow map information, I utilize granular culling of the scene geometry. This is paired with caching of individual pages, which greatly reduces the number of pages that need to be filled each frame. I show that my implementation is well suited for multiple light sources of various types, including directional lights, point lights, and spotlights. Further, I show that this technique scales to an arbitrary number of cascades as only a fraction of virtual pages are visible and need to be backed each frame. The method can thus achieve any desired texel-to-pixel density at any distance with few wasted shadow texels.

Keywords: Shadows, Shadow map, Shadow Mapping, Virtual, Virtual textures, Memory Paging, Meshlets, Mesh Shading, Sparse, Culling, Real Time Shadows, Point Light Shadows

Supervisor: doc. Ing. Jiří Bittner Ph.D. Katedra počítačové grafiky a interakce, ČVUT FEL

Abstrakt

V této práci prezentuji Virtuální Stínové Mapy (VSMs), metodu pro vykreslování stínových map s výrazně lepší kvalitou a paměťovou efektivitou v porovnaní se současnými metodami. Metoda odděluje logický adresový prostor od jeho fyzické realizace rozdělením každé stínové mapy na sadu virtuálních stránek. Tímto je dosaženo zdánlivě vělkého a souvislého paměťového prostoru bez nutnosti kompletně rezervovat fyzickou paměť. V každém snímku jsou nalezeny viditelné stránky pomocí analýzy hloubkové textury. Fyzická stránka z designované paměťi je poté přiřazena každé viditelné stránce. Pro efektivní vyplění viditelných stránek využívám granularní ořezávání geometrie scény. Toto je zkombinováno s ukládáním jednotlivých stránek, což výrazně snižuje počet stránek, které musí být v každém snímku vyplněné. Ukazuji, že moje implementace je vhodná pro libovolný počet světelných zrojů různých typů, včetně směrových světel, bodových světel a reflektorů. Dále ukazuji, že tato technika škáluje na libovolný počet kaskád. Toto je způsobeno omezenou velikostí množiny viditelných stránek, které vyžadují přiřazenou fyzickou paměť. Tato metoda tedy může dosáhnout libovolné hustoty texelů na pixel v jakékoliv vzdálenosti s minimem vyplýtvaných stínových texelů.

Klíčová slova: Stíny, Stínová mapa, Stínové mapování, Virtuální, Virtuální textury, Paměťové stránkování, Meshets, Mesh Shading, Sparse, Ořezávání, Stíny v reálném čase, Stíny bodových světel

Contents

Figures

Tables

Chapter 1

Introduction

Figure 1.1 Image rendered using my application.

Shadows are one of the most important aspects of a three-dimensional scene. By providing visual queues on shapes and the relative positions of objects, they define how the scene is perceived and interpreted. Calculating and storing visibility, which is at the core of each shadowing solution, remains a challenging problem. Even though recent advancements in hardware technology make ray tracing a promising direction, shadow mapping remains a standard and widely used technique for solving visibility queries.

Despite that, current shadow mapping techniques still suffer from artifacts caused by insufficient shadow map resolution. Increasing resolution to alleviate these issues quickly becomes impractical due to memory consumption and hardware limitations. These issues are further accentuated if multiple light sources are considered, each requiring one or more separate shadow maps. Even when we ignore the memory consumption issues, naively rendering all shadow maps becomes prohibitive due to the sheer cost of rasterizing the scene for each shadow map.

Although many current techniques reduce issues caused by insufficient resolution, the fundamental compromise between shadow quality and memory

cost remains. It is common for rendering engines to provide several options for shadows, each with their own set of trade-offs. For example, screen-space ray tracing [MM14] can be used in addition to shadow mapping implementations. This is done to achieve sharper and more detailed contact shadows. Specialized techniques are often used for different light sources, further complicating the implementation.

In Section 2, I start by discussing the current solutions to the problems outlined above and their shortcomings. This is followed by the description of the scene and geometry representation in my implementation. A section describing the virtualization concepts in more detail follows. Finally, this chapter provides an overview of the entire algorithm. In the next chapter, I provide a description of the implementation of the described method. The Individual steps are discussed in detail, including the necessary optimizations, allowing for real-time framerates. The following chapter describes the results I was able to achieve using my technique. Here, I also discuss and analyze the performance of the method. In the last chapter, I conclude the work and offer promising directions for further research.

1.1 Goals

The goal of this work was to develop a technique capable of efficiently storing and rendering a large number of shadow maps. The technique needs to be unified and should not require any additional methods to provide consistent shadows at all ranges and in all scenarios. This means that this technique should be general enough to allow rendering of shadows from various light source types. It should also not rely on any offline baking. A part of the goal is to provide shadows, which can be applied to highly dynamic scenes. As such, the technique should be able to handle and represent shadows for moving, appearing, or disappearing objects.

The technique should be fast and efficient enough to run on consumer hardware. Further, since the main target of real-time shadows are games, the technique should be fast enough to be combined with many different aspects that go into rendering a game. Due to this, the performance of the technique needs to be roughly consistent and independent of the specific viewing conditions. The memory required to store the shadow maps also needs to be bounded and within the bounds present on modern hardware.

Finally, it should be possible to use the shadow maps from this technique for other effects often present in real-time applications. That is, it should allow for rendering of shadows in volumetric effects such as fog. It should also be possible to store shadows of volumetric effects themselves. For example, the technique should be able to represent shadows cast by clouds on the underlying terrain.

Chapter 2
Virtual Shadow Maps

In this chapter, I will provide the relevant background for the following chapter that describes the implementation. First, I will describe how previous work attempts to solve the issues introduced by shadowmapping. Next, I will talk about how the scene and geometry are represented in my implementation. Following this, the concepts relating to texture virtualization and their application to shadow maps will be explained. Finally, I will provide a high-level overview of the algorithm utilized by my method.

2.1 Previous Work

The fundamental issue with shadow mapping are the artifacts introduced by insufficient shadow-map resolution. These occur when a pixel projected into the shadow map covers less than a texel in the shadow map. The same shadow map texel is used to determine visibility for multiple pixels, resulting in blocky looking shadows. To avoid this, at least one-to-one mapping between shadow map texels and pixels needs to be ensured. An example of the discussed issues can be seen in Figure 2.1. As such many of the shadow mapping improvements lie in improving the shadow map texel distribution.

Cascaded Shadow Maps (CSM) is a popular algorithm used to ensure a better distribution of shadow map texels for directional lights [Dim07]. Instead of storing a single high-resolution shadow map, CSM represents the shadow map as multiple separate textures, also called cascades. The viewer's frustum is partitioned into smaller frusta such that each cascade can be paired with its own slice of the view range. Each cascade is then fitted with its own light space matrix covering the entiry-assigned frustum slice. The scene is then rendered once for each cascade using the appropriate cascade light matrix. Due to the shape of the perspective frustum, this results in a denser shadow map being placed near the viewer, where detail is most needed. In contrast, constant regions receive coarser shadows. It is also common for the resolution of textures representing individual cascades to vary. For example, two near-cascades can be used to represent detailed shadows near the viewer. This can be complemented with a third, lower-resolution cascade which is used to render all distant shadows.

Fitting the frustum to cover the entire frustum can be suboptimal in many

Figure 2.1 On the left undesirable effects caused by insufficient shadow map resolution which results in blocky looking shadows. On the right shadows produced by Virtual Shadow Maps.

situations. Rarely is the geometry uniformly distributed throughout the scene. An example of this would be a camera placed inside a room. The walls of the room obscure the rest of the scene from being seen; however, shadow maps corresponding to the rest of the scene are still rendered. These shadow maps will never be sampled and as such, they are completely wasted. This can, to some extent, be solved by manually tweaking the cascade placement for individual areas of the scene. However, this is both very tedious and still fails to achieve optimal results in many scenarios.

Sample Distribution Shadow Maps (SDSM) leverage this fact and improve CSM by using a per-frame analysis of the currently visible portion of the scene [LSL11]. The analysis is then used to maximize the efficiency of the shadow sample distribution for the current viewing angle. The current view analysis is performed by utilizing the depth texture rendered from the main viewpoint. The extent of the analysis can vary between individual implementations. For example, only the minimum and maximum visible depth can be used to omit placing cascades in regions of the viewer frustum that are obscured by geometry. Another example would be to construct a histogram of depth values, which can be used to further optimize the placement of individual cascades to regions with the highest concentration of samples.

In SDSM the individual cascade positioning is then updated every frame to best fit the geometry currently viewed. Due to these constant updates, SDSM often suffers from temporal instability. When the light cascades update in sub-texel increments, this can manifest itself as visible shimmering and aliasing of the shadow map. Because of this, care must be taken when placing the individual cascades.

Adaptive Shadow Maps (ASM) take a different approach [FFB01]. Instead of partitioning the shadow map into multiple cascades, ASM attempts to refine the quality of the shadow map locally. A hierarchical structure is introduced that is used to match the resolution of the shadow map with the resolution required by individual pixels. By evaluating the contributions of

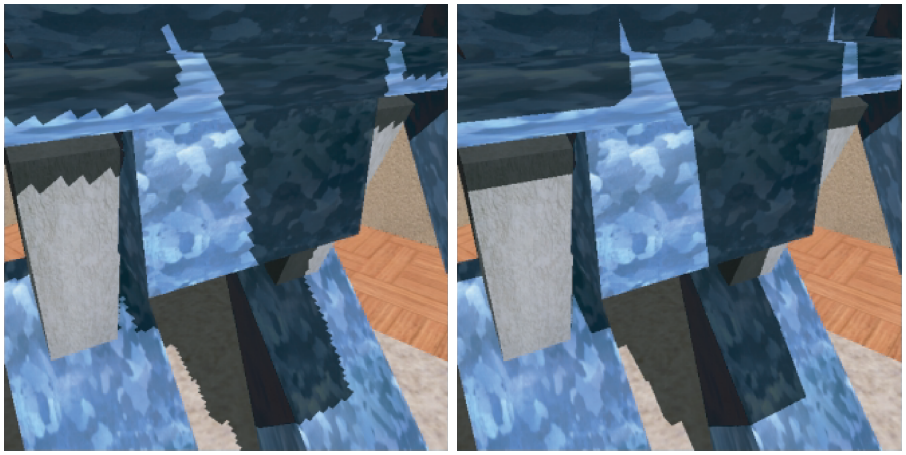


Figure 2.2 Image taken from [FFB01]. Showcase of adaptively refined shadow map on the right, compared to a conventional shadowmap with resolution 2048×2048 texels. This is very similar to the figure shown in 2.1 obtained by my method.

the individual shadow map pixels to the overall image quality, this structure is refined. This is done using an edge detection algorithm that ensures an efficient distribution of the texel shadow edges. The structure is thus iteratively refined to provide the best results for each viewpoint. A memory limit specified by the user is then used to bound the final quality of the shadow map obtained. The results obtained by this method can be seen in Figure 2.2.

A slightly different but still similar approach was proposed by *Resolution Matched Shadow Maps* (RMSM) [LSO07]. The same quadtree structure used in ASM was utilized to represent and refine the shadow map. A big downside of ASM is the highly variable amount of time that the iterative edge finding algorithm takes to converge. This makes ASM based shadows very expensive for dynamic scenes. To avoid this, RMSMs forgo the expensive iterative refinement step in favor generating a list of all shadow map texels requested by the by all pixels of the main view.

Another approach building upon and improving ASM is called *Queried Shadow Maps* (QSM) [GW07]. By using occlusion queries present in GPU hardware and improving the heuristics of the refinement algorithm, QSMs offer much better performance in highly dynamic scenes. This is mainly because the improved refinement algorithm can be executed completely in each frame, which is not true for ASMs, which rely on caching to deliver the best performance.

Recently, *Efficient Virtual Shadow Maps for Many Lights* introduced the concept of virtualizing shadow maps by separating the shadow map itself from its physical storage [OSK14]. They discuss hardware-supported virtualization of cubemaps used to store shadows. Every cube map is split into individual tiles that are then requested by individual pixels.

Similarly to ASM, RMSM and QSM, the proposed algorithm allows for

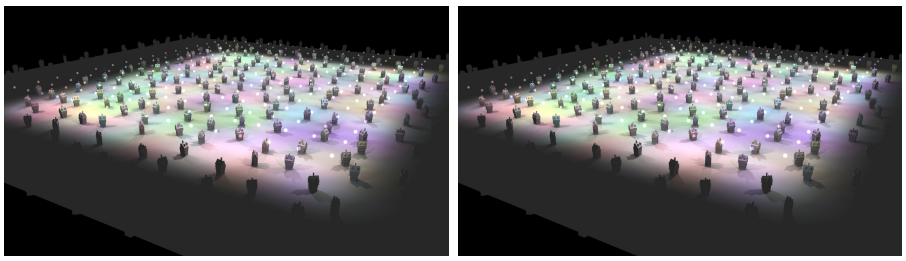


Figure 2.3 Image taken from [OSK14]. Two images obtained by utilizing virtual approach proposed by the authors of the Efficient Virtual Shadow Maps for Many Lights publication. Their method allows for rendering of large amounts of individual point lights. Specifically, there are 256 point lights in the left image and 356 lights in the right image.

dynamic scaling of the shadow map resolution. However, this is done on a much coarser level than the previous methods. As opposed to individual pixels requesting the shadow map tiles, the resolution of each cube map face is determined by taking the maximum resolution requested by all pixels.

This publication also proposes a highly efficient geometry culling scheme, which is required to achieve real-time frame rates. The culling is based on clustering the geometry into a Bounding Volume Hierarchy (BVH). Each leaf of the BVH stores batches of triangles represented by an axis-aligned bounding box (AABB). This hierarchical scheme allows for quick discarding of geometry in the top levels, improving culling performance.

Finally, to shade the extreme amount of individual lights, the authors propose a clustered shading approach. The entire view frustum is subdivided into a set of cells. First, the cells that contain geometry are used to analyze the rendered view. This is done in a fashion similar to how SDSM analyze the scene. Each cell containing visible geometry is then intersected with the spheres that represent the individual lights. This yields a list of contributing lights for each cell, which is used to drastically reduce the number of lights that need to be considered during shading of each pixel. The results of this method can be seen in Figure 2.3.

A work-in-progress version of this thesis was used to co-author *Implementing Virtual Shadow Maps* (VSM) and author *Resolution Matched Virtual Shadow Maps* (RMVSM). The first describes the complexities of implementing virtual shadow maps for directional lights [SRH24]. Although there is some overlap between this work, the technique I present here has been significantly improved and modified. The second publication describes a more up-to-date version of the algorithm. It provides an explanation of the generalized method that allows for point- and spot-light sources. In addition, the scene representation and culling scheme is described. This means that there is significant overlap between this work. However, the RMVSM publication only described a high-level overview of the method without delving into the implementation details. As such, parts of the RMVSM publication were taken from the work-in-progress version of this chapter.

2.2 Scene Description

In this section, I describe how the scene and geometry are represented and laid out in memory. I will discuss the pre-processing and optimization steps that geometry goes through to arrive at a format that is the most efficient for VSMs. Finally, I will talk about how the scene is parsed into a set of drawlists consumed by the GPU.

2.2.1 Graphics Library Transmission Format

The implementation supports loading files in the second version of the Graphics Library Transmission Format (glTF). Although this is not the most efficient when it comes to representing a scene, its flexibility makes it a very popular choice for development purposes. This is mainly due to the wide support in existing tools as well as the option to add custom features using extensions. Every glTF contains one or more binary files that store the data required to render the content of the scene. Binary files contain unique raw geometry and texture data and need to be accompanied by a tree-like hierarchical data structure. This structure describes the relationship between individual objects in the scene and is stored in a JSON file.

At the root of the scene hierarchy is the scene node. This contains one or more child nodes, which can themselves contain more child nodes. In glTF there are many types of nodes, but only a few are relevant to this work. Specifically, those are **transform**, **mesh** and **light** nodes. As the name suggests, the transform nodes contain only a transform and are used to affect the transforms of the objects directly below them in the hierarchy. In addition to containing a transform, mesh nodes are used to instance one or more meshes into the scene. As such, these contain a list of indices pointing to individual meshes. Similarly, the light nodes contain an index to the light that they are introducing into the scene.

A mesh in glTF represents a group of triangles with shared properties. The triangles contained in the same mesh use the same material and textures. To avoid storing duplicate vertices, index buffers are a de facto standard when representing geometry. Thus, each mesh is represented by two buffers, the first storing all vertices and the second storing the indices.

The scene in my framework adopts a similar structure; however, there is one key difference in the mesh representation. As already mentioned, VSMs require hierarchical culling in order to be efficient. To allow this, the geometry in the meshes also needs to be represented as a hierarchy. Instead of meshes containing the data of individual triangles directly, they contain a list of meshlets. Each meshlet contains information about a bounded number of triangles. The bound is typically set very low; around 64 to 128 is used in practice. This greatly aids in work distribution on the GPU by closely fitting the mesh shading pipeline.

The triangles of each meshlet are again represented as a list of indices. The index buffer for the entire mesh consists of 32bit entries to allow for very

large meshes. Because of the limited number of stored primitives and the relative close spatial locality of the vertices for each meshlet, the indices of individual meshlets do not need to be this large. For each meshlet, only a micro-index buffer is stored. It consists of a single 32 bit offset and a set of 8bit entries.

2.2.2 Geometry Processing

The meshlets and micro-indices are generated utilizing an open source library called Meshoptimizer [Kap25]. Meshoptimizer also generates axis-aligned bounding boxes (AABBs) and levels of detail (LODs) for both meshes and meshlets. AABBs are used to cull geometry during the drawing stage. The LODs are used to reduce the complexity of geometry that is farther away from the observer. Mesh processing can be quite costly; however, individual meshes are completely separate and can be processed in isolation. As such, I utilize a thread-pool approach to process multiple meshes in parallel.

2.2.3 Drawlists

All mesh data are processed on the CPU; however, immediately after the processing is done, the data are moved to the GPU. This is not true for the node graph that represents the scene. In contrast to meshes, the node graph is used purely on the CPU. At the start of every frame, the it is thus flattened into a set of draw lists. Furthermore, each frame, the combined transforms for each individual mesh are computed and stored in a GPU buffer. By doing this, I avoid the recursive process of parsing the node graph, which is not well-suited for GPU processing.

A draw list is a list of mesh indices that logically group meshes with shared properties. For the purposes of VSMs three draw lists are required. The first two are used when drawing the VSMs and split the geometry into alpha-masked and opaque meshes. This separation is important for performance reasons. Alpha-masked meshes require a texture fetch to determine the transparency of each texel. The third draw list required for VSMs holds all dynamic objects. That is, all objects for which the transform changed in between frames. This draw list is used to invalidate cached pages, as will be explained in further chapters. This also highlights an important property; a mesh instance can be simultaneously referenced by multiple draw lists at once.

2.3 Shadow Map Virtualization

As already stated above, the main issue that shadowmapping faces is insufficient resolution. However, naively increasing the resolution has its drawbacks. First, the memory cost scales quadratically with resolution, quickly making high-resolution shadow maps impractical. Second, higher resolution reduces the spatial locality for situations where a single pixel maps to too many

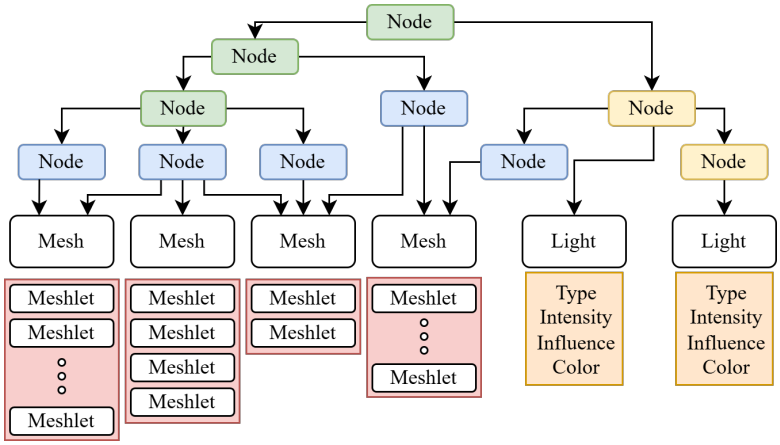


Figure 2.4 Scene representation Hierarchy. The bottom half shows the unique meshes and meshlets present in the scene as well as the unique lights. The upper part shows an example of the node graph. Nodes in blue reference at least one mesh, while green entities only contain a transform. Yellow nodes reference individual lights. Note that a single mesh or light can be referenced by more than one node in the entity graph.

shadow map texels. This increases the likelihood of cache misses, reducing overall performance. The last issue arises from the limits of the rasterization resolution placed by the target hardware. To overcome this, the scene would need to be rendered in multiple passes, further reducing the performance.

To avoid all shadow mapping artifacts, a one-to-one mapping between pixels and shadow map texels is necessary. However, the area of the shadowmap covered by each pixel is not uniform. The covered area depends on many factors, such as the distance from the viewpoint or the angle of light. Pixels belonging to geometry close to the viewpoint typically map to smaller areas of the shadow map and, as such, require the most resolution. Due to this non-uniformity, a different resolution is required for different parts of the shadow map in order to achieve the ideal mapping.

These requirements are very similar to the ones demanded from ordinary textures. Textures usually have the highest resolution possible to allow for good-looking closeups, while relying on mipmaps for more distant views. To combat the memory demands of high-resolution textures, a technique called virtual texturing is often used [Bar08; Wav09; WH10]. Virtual texturing leverages an important observation - even when the entirety of a texture is present in memory, only a limited subset will be sampled at any given frame. Due to this, memory can be reused to back only the sampled portions of the virtual texture. This allows for the appearance of a very large contiguous memory space without the need to fully reserve the physical memory. The required memory is no longer given by the resolution of the texture. Instead, it is given by the largest portion of the texture that can be visible at any given time. Utilizing this concept across all textures present in a scene achieves a high reduction in memory consumption.

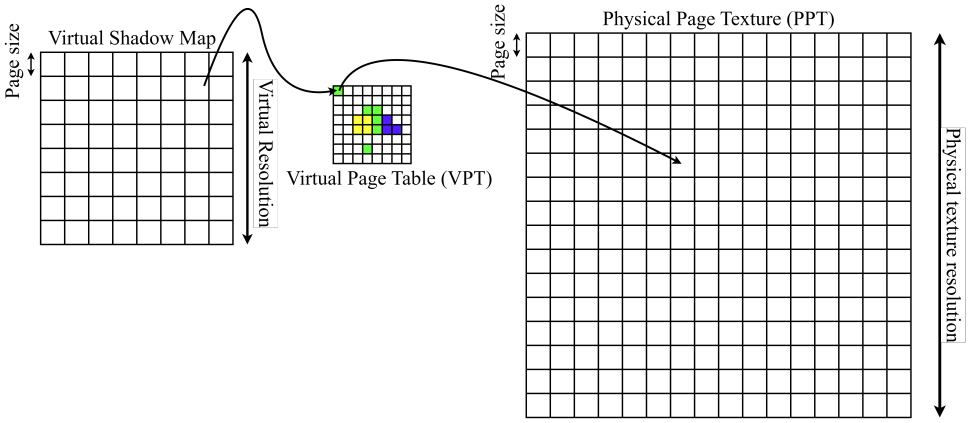


Figure 2.5 The virtualization layout adapted by our technique. Each entry stores metadata about the page, denoted by different colors in the image. In this example, white pages are not allocated, green pages are allocated and visible this frame, yellow pages are allocated but not visible this frame and finally blue pages are newly seen, and thus dirty and requiring a backing page from the PPT. The PPT itself is also denoted in the image. When a page is backed, the VPT entry stores coordinates to the corresponding page in the PPT.

By treating a shadow map as an ordinary texture, all of these concepts can be applied to increase the efficiency with which shadow maps are stored and rendered. Each shadow map is divided into a set of pages and is fully virtualized. It is thus represented by a single texture called Virtual Page Texture (VPT). VPT is of much lower resolution than the full virtual resolution of the shadow map. Every entry in the VPT contains the metadata for one page. That is, each entry in the VPT contains the allocation status and coordinates of the allocated page in the Physical Page Texture (PPT). The PPT is a second texture that I introduce. It is shared by all shadow maps and is used to provide the physical backing for the allocated pages in the VPT. Because all lights share the PPT, the page size is uniform across all shadow maps and light types. The virtualization scheme can be seen in Figure 2.5.

2.3.1 Page Caching

Shadow map caching utilizes the results from previous frames instead of redrawing the shadow map each frame. Although this can be a large optimization, it does come with caveats. Whenever any object that was previously rendered in the shadow map changes its position, rotation, or scale, the cached shadow map needs to be redrawn. Although most of the objects in a scene remain static, there are typically at least a few dynamic objects visible at any time. These objects force the shadow map to constantly redraw, greatly reducing the benefits of caching. For these reasons, shadow map caching is usually mainly utilized for smaller sources of light.

In VSMs, I have much more granular control over the entire shadow map. Instead of redrawing the entire shadow map, only the pages containing the dynamic objects need to be redrawn. By doing this, I minimize the number

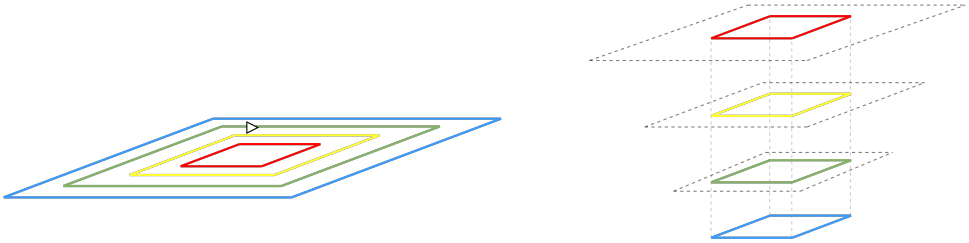


Figure 2.6 The virtualization scheme of the directional light. On the right we can see the Clipmap stack. With dotted lines I denote the real size of the mip level. In color is then the clipped level that is stored. All Clipmap stack levels share the same resolution. Because of this the portion of the underlying mip level covered by each clip level increases with increasing mip level. Increasing clip levels thus cover larger area in the world as can be seen in the left part of the image.

of required draws and allow for more efficient culling, which will be described in later chapters.

2.3.2 Light Type Specifics

My technique supports three types of light sources, directional lights, spot lights, and point lights. Although the three are similar in many ways, there are important differences in how I approach the virtualization of their shadow maps. In this part, I describe the specifics for each type of light.

Directional Lights

A typical scene contains only one or two directional lights. They always affect the entire scene and thus are particularly difficult to provide consistent shadows for. Due to their large area of influence, they are represented as a virtual texture spanning the entire scene. This texture needs to have a very high resolution, typically hundreds of thousands of pixels. To allow for this approach, I use a concept called Clipmap [TMJ98; AH05].

The main difference between a traditional mipmap and a Clipmap is that each level of a Clipmap is clipped to a specified maximum size. This divides the Clipmap into two parts, the set of mip levels that have been clipped by the maximum size, called the *Clipmap stack*, and the set of mip levels with the original size called the *Clipmap pyramid*. For the purposes of my solution, I utilize only the Clipmap stack.

A shadow map for a directional light is thus represented by a set of VPTs, each representing a virtual shadow map for a single Clipmap level. Because I utilize only the Clipmap stack, all levels maintain the same resolution. According to the original Clipmap technique, every level covers twice the area of the previous one. This ensures the varying texel density of the shadowmap. The higher the level, the larger the area covered and the larger each texel footprint becomes. A visualization of the directional light paging scheme can be seen in Figure 2.6.

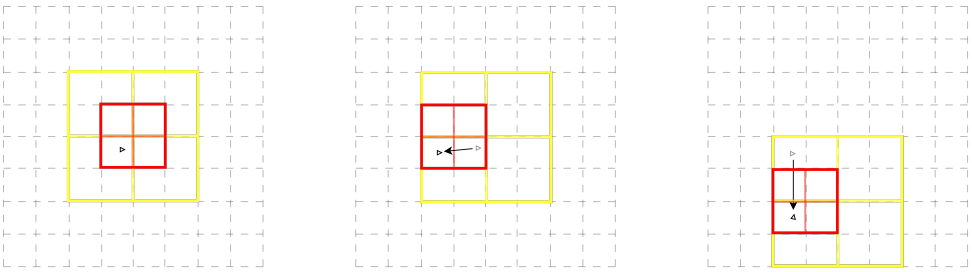


Figure 2.7 The snapping performed by the cameras corresponding to two clip levels demonstrated in top down view. This is a simplified scenario, where each virtual clip level has only four pages. Frustums of the individual Clipmap levels are snapped to their local page grid. At the start, the frustums of the two shown levels are aligned. When the camera starts to move the frustums become misaligned due to the differing world sizes of the page grid.

To support caching, the frustums of the individual Clipmap stack levels are snapped to the page grid given by that specific level. This means that the positions of the frustums for individual levels do not need to be the same, as can be seen in 2.7.

As the cascade frustum moves to follow the main camera, new pages, previously located on the edge just outside of the cascade frustum, might need to be drawn. In order to preserve previously cached pages, I utilize a sliding window, also called *Toroidal addressing* [TMJ98] or *2D wraparound addressing* [AH05]. With a sliding window, the newly entered pages are mapped to the location in the VPT previously occupied by old, exiting pages. Figure 2.8 shows a visualization obtained by projecting the directional VPT pages into the world space.

Spot Lights

A scene usually contains a larger number of spotlights. However, unlike for directional lights, such a large virtual resolution is no longer needed. This is mainly due to the fact that their effects are much more localized. As such, resolutions of four to two thousand texels are often sufficient.

Clipmaps are thus no longer needed, and the classic mipchain can be used. This is a key difference between spotlights and directional lights. Directional lights have clip levels with constant resolution that cover an increasingly large area. Spot lights have mipmap levels with decreasing resolution that cover the same area.

While slightly complicating rendering of the shadow map, described in the following chapters, it allows a significant reduction in the number of VPTs required to represent the light. Where directional lights require an entire VPT per clipmap, the mipchain of the VPT can be leveraged. Thus, a spotlight can be represented by a single VPT.

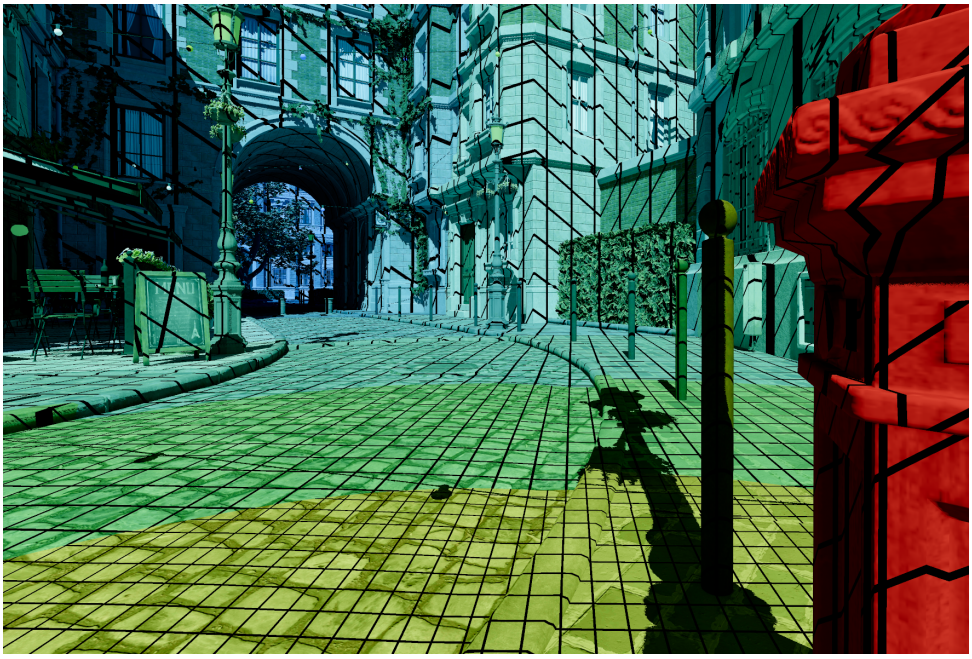


Figure 2.8 Visualization of the directional shadow map virtualization and paging scheme. Black lines denote individual shadow map tiles. Different colors denote different clip levels. We can clearly see how each clip level increases the page size. The objects close to the viewing position have low clipmap levels, because they require the most resolution. The further the objects get from the camera the less resolution are needed and so higher clip levels are requested.

Point Lights

The Last light type that will be described are point lights. Point lights are often represented as cubemaps. I model the cubemap by six rotated spotlights that share an origin. This makes point lights in many ways similar to spot lights. As such, point lights do not pose a significant difference for my technique. All shadow map textures are fully virtualized; a point light is thus simply represented as six VPTs, one for each face. Figure 2.9 shows a visualization obtained by projecting the point light VPT pages into world space.

2.4 Algorithm Overview

Next, I will describe a high-level outline of the proposed algorithm. The goal of this section is not to give a detailed description of each individual stage. Instead, this section should serve to conceptually explain the purpose of each of the steps. In addition to this, it should also provide an overview of how the individual stages interact and fit together. The algorithm is divided into four steps *Page invalidation*, *Page marking*, *Page allocation*, and *Page drawing*. Each of these steps can contain multiple individual substeps, which will be described in the later chapters.

Figure 2.9 Visualization of paging and virtualization of a single point light. Black lines denote individual shadow map pages. Different colors determine separate point light faces. The sizes of the pages imply the mip map from which the tile was selected. The tiles are now distorted by the perspective projection of the point light and no longer appear strictly rectangular. Additionally, the more complicated mip level selection can be observed. As the distance towards the light decreases, the resolution of the shadow map increases, which leads to lower mip map levels required (visible on the bottom, red, face of the point light).

2.4.1 Page Invalidation

Before any processing of the current frame can begin, changes from the previous frame need to be handled. These are mainly caused by the movement of the main camera as well as dynamic objects. Because all pages are cached, this requires an explicit step that frees pages that no longer contain valid information. For dynamic objects, all allocated pages they intersected the last frame, as well as all pages they intersect this frame, need to be redrawn. This needs to be done for all light source types. In addition, for directional lights, pages that fall outside of the wraparound addressing window need also be freed.

2.4.2 Page Marking

The second step of our algorithm is determining the shadow map pages that will be required for this frame. For these purposes, similarly to SDSM, the scene is rendered from the view of the main camera into a depth buffer. Following this, a pass over the acquired depth is performed. During this pass, each pixel is projected into VPTs of each light. This projection obtains the pixel footprint in the shadow map, which is used to determine the required Clipmap level or mip level. The page that this pixel requires is then marked. If this page is not currently allocated, it will be added to the set of VPT pages that require backing, which will be consumed in the next step.

2.4.3 Page Allocation

The marking step is followed by an allocation step. From the set of VPT pages marked in the previous step, those that do not have physical backing need to be allocated.

For the purpose of this, I first perform a pass over the entire PPT. This pass classifies all physical pages into three categories, free pages, allocated pages, and cached pages. Free pages are self-explanatory; these are the pages that are not currently used to back any VPT page. The allocated pages denote pages that contain valid data from previous frames, which will also be required in this frame. That is, the VPT page that is backed by this was also marked as required for this frame. As such, these cannot be used in the allocation process and remain untouched. Similarly to allocated pages, the set of cached pages denotes the pages that contain valid data from previous frames. However, they have not been marked by the Page marking step and their data will not be used in the current frame.

Once the physical pages are classified, the allocation step performs the assignment of the free or cached pages. I first attempt to back all the VPT pages produced by the previous step with the free pages. If there are no free pages remaining, I instead use the cached pages to back the requests. This requires the cached page to first be freed before the memory can be reused. Finally, all newly allocated pages need to be cleared so that their previous content does not obstruct the next steps of the algorithm.

2.4.4 Page Drawing

Drawing the pages of the shadow map is the last step of my algorithm. The rasterization itself is preceded by a number of geometry processing steps. First, to obtain the geometry instances that need to be drawn, the scene graph needs to be processed. This is followed by a series of work expansion and hierarchical geometry culling steps. Work expansion is necessary to ensure optimal work scheduling and sufficient GPU saturation. Culling massively reduces the amount of geometry that needs to be processed. This plays a fundamental role in obtaining real-time frame rates. To achieve the most efficient culling, a specialized data structure is required. This structure, which will be described in later chapters, is built as the next step of the page-drawing process. Finally, all geometry that survives the culling steps can be rasterized and drawn into the pages.

Support for Volumetric Effects

As stated in the goals section, the technique should allow for rendering shadows cast both by volumes and into volumes. The former is immediately supported by the described approach. All steps will remain the same up to the point of rendering the pages. This step would be replaced by the specific method used to store the shadowing information. For example, the volume that should be casting the shadow would be ray-marched for each

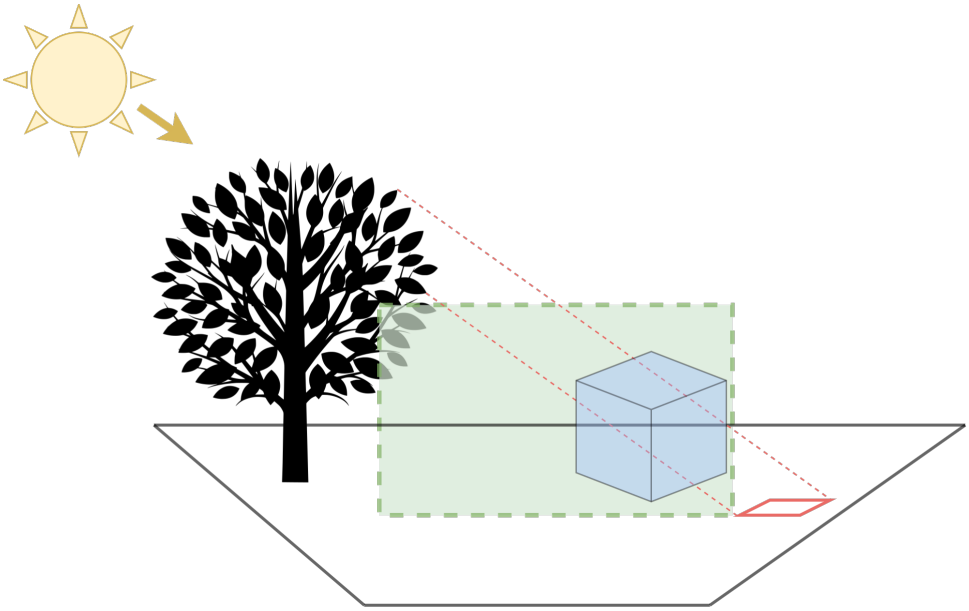


Figure 2.10 Demonstration of issues caused by page marking. Green rectangle denotes the area visible by the main camera. The blue box represents a volume in which should be shadowed by the tree on the left. Because the pages are only requested for pixels hitting opaque surfaces, the page in red will not be requested and as such will not contain any data. This is an issue, because this page is required to draw the tree shadow into the volume.

texel of every page, and the average depth stored. Depending on the method, the shadowing information stored in each page might differ from traditional shadow maps. As such, the format of the page and thus the format of the underlying PPT might be different. However, the steps of the algorithm remain exactly the same.

Sampling the shadow map when rendering volumetric effects will cause issues with the steps described so far. More specifically, the marking stage considers only visible surfaces when requesting the pages. That is, only the pages that would result in shadow cast on a surface in the view of the main camera are requested. Because the volume is not a solid surface, we might encounter a scenario in which no page covers a section of the volume. This can be seen in Figure 2.10.

This can be solved in various ways. The volume can be ray marched twice, first time the individual pages will be requested and once these are drawn, the volume will be ray marched for a second time, this time sampling the shadows and properly shading. This is the most precise approach; however, it is also the most expensive. Second, the volume can be only ray marched once and immediately shaded. When an unallocated page is encountered, it will be requested and drawn in the next frame. This will introduce some flickering; however, it might still produce acceptable results. Another approach would be to project the frustum of the main camera into a high level of the shadow map. All pages intersected by this projection would then be marked. Since

volumetric effects are only marched for the on-screen pixels, this would always guarantee the presence of the required pages. Because volumetric effects are often of low visual frequency, the shadow resolution required is also not high. As such, the frustum projection step might be skipped entirely, and an entire, high, level of each shadow map might always be rendered. This is especially viable for point lights and spot lights, as their highest levels cost only a fraction of the memory required for the lower ones.

Chapter 3

Implementation

In this chapter, I will describe the implementation details of the solution proposed in the previous chapter. The technique was integrated into an open source research framework called Timberdoodle [SA25]. Because of this, our solution was developed using the C++ language, using the Vulkan API to interact with the GPU. All shaders were implemented using the Slang [NVI25] shading language.

3.1 Virtualization Setup

I will start by describing the virtualization data structures that are the basis of the solution. These are utilized throughout most of the steps of the algorithm, and as such need to be described first. It is important to note that not all details regarding virtualization will be described in this section. Specific parts will be explained later in the chapter, once the appropriate context has been laid out.

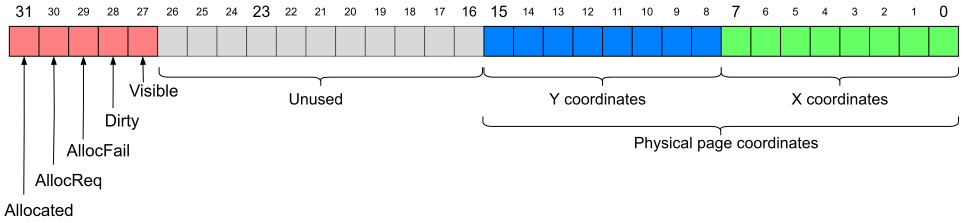


Figure 3.1 Format of a single VPT entry. Blue and green bits store the coordinates of the backing page in the PPT, if one is allocated. The grey bits are unused in our implementation. Finally the Red bits hold the meta information about the VPT entry.

3.1.1 Virtual Page Texture

The VPT represents the virtualized shadow map and contains meta-data entries for each page described in the previous chapter. To represent it, I use a single 32 bit unsigned int texture. The resolution is given by the as:

$$Resolution_{VPT} = \frac{Virtual\ resolution}{Page\ resolution}. \tag{3.1}$$

The structure of each VPT entry can be seen in Figure 3.1. Each entry contains five meta-information bits:

- **Allocated** - when set, the entry is backed by a physical page.
- **AllocReq** - when set, the entry is requesting allocation.
- **AllocFail** - when set, the allocation for this entry failed.
- **Dirty** - when set, this page will need to be redrawn this frame.
- **Visible** - when set, this page will be sampled and requires backing.

Each entry also contains the coordinates for the physical page texture when the page is allocated. I utilize only 16 bits to store this information, leaving some bits unused. This limits the size of the PPT to only contain 256×256 pages, which is sufficient in my case. If the need for larger PPT would arise, the unused bits could be used to extend the addressable range.

Directional lights utilize Clipmap levels, which have a set resolution and thus require a VPT per level. However, for point lights and spotlights, the mipchain of the VPT can be utilized to store all levels at once. To avoid wasting memory by allocating mip levels for directional VPTs, I store the VPTs in two array textures. The first is used for all directional light sources, while the second is shared for point lights and spot lights.

Dynamically reallocating resources during the execution of the application often causes expensive copies frame stutters. Because of this, in GPU-based approaches, it is common that all textures are allocated once during the start phase of the application. They are allocated for the worst-case specified by the application. I follow this principle and allocate VPTs for a set number of light sources. In my implementation, this number is modifiable, but only during the compilation of the application itself. The actual amount of shadowcasting lights can change throughout the runtime of the application, however, it cannot exceed the worst-case boundary. The storage layout of all VPTs can be seen in Figure 3.2.

3.1.2 Physical Page and Meta Memory Textures

The PPT represents the memory pool, or shared storage, which is used to back individual virtual entries. In my implementation I use a single 32 bit floating point channel texture. Because of implementation details that will be described later, meta-information for each page in the physical texture also needs to be stored. In order to do this, I utilize a second texture, which I call the Meta-Memory Texture (MMT). Similarly to a VPT, the resolution is given by the resolution of the PPT divided by the page resolution. In contrast to the VPT, I require a bit more space to store all necessary information. Because of this the format is a single channel 64 bit unsigned integer.

The MMT is essentially a mirror double of the VPT. The structure of a single entry can be seen in Figure 3.3. Just like with VPT, each entry contains meta-information about the state of the underlying physical page:

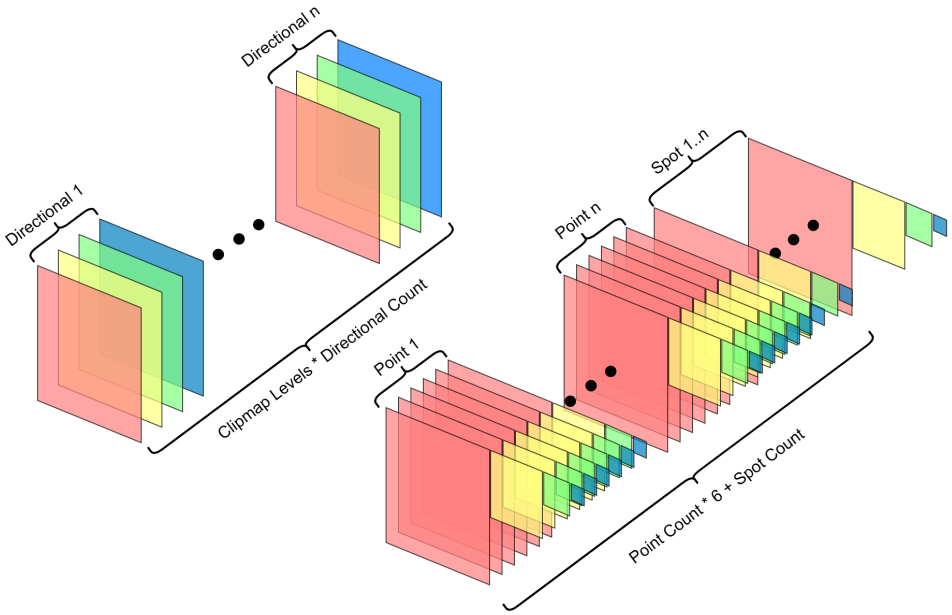


Figure 3.2 The layout of VPTs in my implementation. On the left we see the array of VPTs for directional lights. Each Clipmap level, denoted as different colors, is represented by a separate array layer. The Clipmap levels of individual directional lights are tightly packed in this array. Point lights and spotlights occupy the second array shown on the right. The individual levels, denoted as different colors, are represented by levels in the mipchain of the VPT. Each point light is represented by six faces, each stored as a single VPT. After all point lights, I tightly pack VPTs for spotlights. Each spotlight requires only a single VPT.

- **Allocated** - when set, this memory is backing a VPT entry.
- **Visible** - when set, this memory is backing a visible VPT page.
- **PointSpot** - when set, this memory is backing a spotlight or point light VPT entry.

Additionally, every entry contains the complete coordinates required to access the backed VPT entry. Using the *PointSpot* bit, the appropriate VPT array is selected. The process of determining the offset in the array of VPTs differs slightly depending on the type of light. For directional lights, the directional light index is used first to determine the 0th Clipmap level of the given light. To this, the clip level number is added, which obtains the correct array layer. Point lights and spotlights store the offset into the array directly. The bits used to represent the Clipmap level are used to represent the target mip level of the given VPT. With this, there is a double link between the VPT entries and backing PPT pages. This is especially useful because I often need to perform two types of traversal. I wish to both iterate over memory to resolve backed VPT entries and scan the VPT to find backing pages.

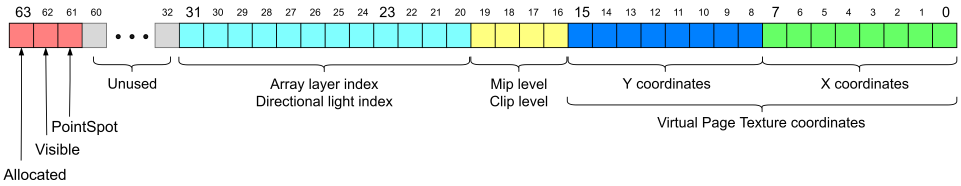


Figure 3.3 Format of a single MMT entry. Similarly to the VPT, the entries in red store the meta information about the page. Blue and greed bits store the coordinates of the backed page in the corresponding VPT. The yellow bits represent either the Clipmap level or the mip level, depending on the type of light. Finally the teal bits are used to represent the offset of the corresponding VPT in the array of VPTs.

3.2 Toroidal Addressing

As discussed earlier, the usage of Clipmaps demands a special addressing scheme in order to support caching. Whenever the camera representing the virtual Clipmap level snaps to another location, new pages at the edge of the new region need to be allocated. Similarly, the pages previously representing locations that are not covered by the updated frustum can be freed. This can be seen in Figure3.4. Without this mapping scheme, I would need to copy and shuffle the entries in the VPT each time the light frustum moves. This requires storing a per-Clipmap level offset of the respective light matrix position from the origin.

Listing 3.1 Function used to convert from coordinates obtained by projecting by clipstack level light matrix into wrapped coordinates.

```
1
2  int2 page_coords_virtual_to_wrapped(int2 virt_coords, int2 cliplevel_offset)
3  {
4      // Make sure that the virtual page coordinates are in
5      // page table bounds to prevent erroneous wrapping .
6      int2 bounds = int2(0, VSM_PAGE_TABLE_RESOLUTION - 1);
7      bool not_in_lower = any(lessThan(virt_coords, bounds.x));
8      bool not_in_upper = any(greaterThan(virt_coords, bounds.y);
9
10     if( not_in_lower || not_in_upper {
11         return int2(INVALID_COORDS);
12     }
13
14     int2 offset_coords = virt_coords + cliplevel_offset;
15     int2 wrapped_coords = mod(offset_coords, VSM_PAGE_TABLE_RESOLUTION);
16
17     return wrapped_coords;
18 }
```

This comes at the cost of more complicated addressing of the VPT. Now I need to distinguish between virtual coordinates and what I call wrapped coordinates. When determining which page a world position belongs to, I need to proceed in two steps. First, the virtual coordinates need to be obtained by

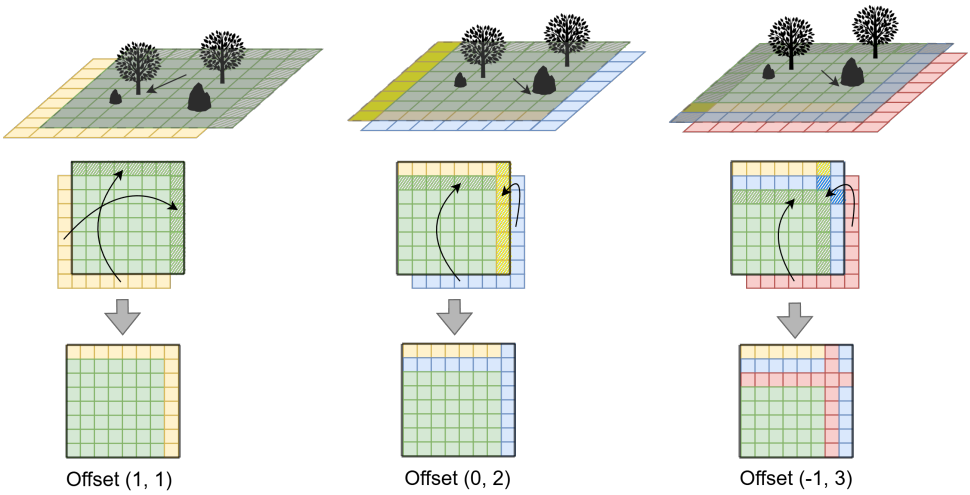


Figure 3.4 Left to right shows three consecutive frames in which the frustum of one of the Clipmap levels snaps to another position. The top row displays the covered world space area. In gray is the frustum of the previous camera position. In green are pages that were cached by the original frustum. Yellow, blue and red denote the pages that are covered by the new frustum. Hatched are the pages that are no longer covered by the new frustum. The middle row shows the mapping of the new pages. Finally, the bottom row shows how the state of the VPT after the mapping has been performed.

projecting the position by the light matrix. Second, these coordinates need to be transformed into wrapped coordinates. The wrapped coordinates can then be used to look up the corresponding VPT entry. Listing 3.1 shows the function used to perform the mapping from virtual to wrapped coordinates.

The last issue I would like to discuss occurs when sampling a cached page. To know if a position in the world is shadowed, it first needs to be projected into the shadow map. Then it is compared with the value stored in the shadow map to determine visibility. Because the light camera matrix changes between frames, it can be different from the camera with which the page was drawn. If I were to use the current matrix of the Clipmap level to project the world position, I would get invalid results. This is demonstrated in Figure 3.5.

To alleviate these issues, I introduce another texture that is paired with each VPT. I use this texture to store and reconstruct the view matrix with which each page was drawn. The entire matrix does not need to be stored as the only thing that changes between frames is the position of the camera. I can store only the elements which change with changing camera position, which are the three floating point values in the last row of the matrix. Unfortunately, there is nearly no hardware support for three channel 32 bit floating point textures, so I must resort to four channel texture instead. Because this texture has the same resolution as the VPT, combining them together would also be a viable approach. The first channel would be used to store the VPT information and the last three would be used to store the offset. In the end, I decided to keep these separate to reduce bandwidth and increase cache coherency

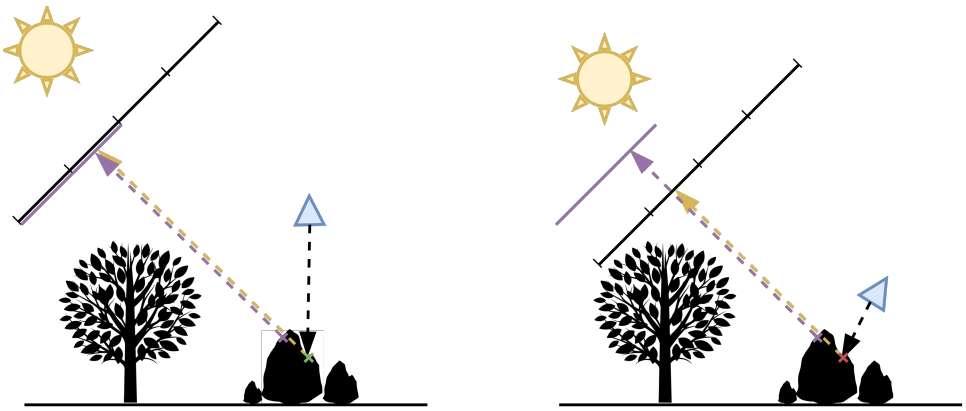


Figure 3.5 Demonstration of issues caused by caching and camera movement. The light camera of each Clipmap level follows the main camera. In the left image two pages, denoted in purple, get rendered and cached. The following shadow test determines the sample is shadowed because the distance in the shadow map is closer than the projected world position. In the right image, the main camera moves closer to the rock and the Clipmap camera follows. The shadow test erroneously uses the new camera and determines the sample is no longer in shadow. This is because the world position is not projected into the camera with which the cached page was drawn.

while sampling the VPT. This is because sampling the VPT is needed in many steps of our algorithm while sampling the offset happens only during the shading stage.

3.3 Page Invalidation

The algorithm begins by freeing cached pages that no longer contain valid data. This is done first so that the allocation step can immediately reuse the freed memory to back new pages. There are two primary mechanisms that can cause pages that have cached data to become stale. The first, discussed in the previous section, is by means of toroidal addressing. When a page is located outside of the current light frustum, it must be freed as it cannot no longer be addressed. The second is caused by a dynamic object intersection. Whenever the world area covered by a page contains a dynamic object, the data need to be updated to represent the new state of this object.

3.3.1 Toroidal Invalidation

I will start by describing the toroidal invalidation. However, before any pages need to be invalidated, the light camera for all Clipmap levels needs to be updated. This is the only step that is performed on the CPU and will be described in the next part.

Light Camera Update

To preserve correct caching, the camera of each Clipmap level needs to snap to the next position in page-sized increments. Because the pages sizes differ across individual levels, the update of camera for each Clipmap level must be done separately. While the view matrix of the clip-level camera changes, the projection remains the same for the entire run-time of the application. The projection matrix is thus calculated at the startup of the application once, and then only reused. Directional light sources utilize orthographic projection. As described in the previous chapter, each Clipmap level covers twice the world area of the previous level. The orthographic projection is then given by the clipmap level and the size of the zeroth level in world space.

The first step in determining the view matrix is to project the target position into the default view space. This view space is shared by all clip-level cameras that belong to the same light. It is given by a view matrix that is centered at the origin of the world coordinate system looking in the direction of the light source.

$$\mathbf{p_{def}} = V_{def}\mathbf{p_t} \tag{3.2}$$

Following this, I project the position in the default view space by the clip-level projection matrix to obtain the target position in normalized device coordinates.

$$\mathbf{p_{NDC}} = P_{clip}\mathbf{p_{def}} \tag{3.3}$$

Next, I calculate the NDC size of a single VSM page.

$$s = \frac{VSM_{page\ resolution}}{VSM_{texture\ resolution}} \tag{3.4}$$

And use it to scale the projected position.

$$\mathbf{p_{scaled}} = \frac{\mathbf{p_{NDC}}}{s} \tag{3.5}$$

This gives the position in the NDC space normalized to the size of the page. Finally, I take the ceil of this to arrive at the position that is aligned to the page grid.

$$\mathbf{p_{align\_scaled}} = \lceil \mathbf{p_{scaled}} \rceil \tag{3.6}$$

The page-aligned world position of the clip level is then obtained by unprojecting from this space.

$$\begin{aligned} \mathbf{p_{align\_scaled}} &= (\mathbf{p_{scaled}}[1], \mathbf{p_{scaled}}[2], \mathbf{p_{NDC}}[3]) \\ \mathbf{p_{align}} &= \mathbf{p_{align\_scaled}} * \mathbf{s} \\ \mathbf{p_{align\_world}} &= (P_{clip}V_{def})^{-1}\mathbf{p_{align}} \end{aligned} \tag{3.7}$$

This position is then offset by the height h at which the light should be above the target position. The offset is calculated along the normalized light direction d_{light} to preserve page alignment.

$$\mathbf{p_{final}} = \mathbf{p_{align\_world}} + h * -\mathbf{d_{light}} \tag{3.8}$$

Invalidation

Using the aligned page positions p_{align_scaled} from the previous and current frames, I calculate an offset per Clipmap level. This offset denotes the change in pages in the position of each Clipmap level. To free pages invalidated by this offset, I perform a single compute dispatch mapping one thread to each entry in every VPT table. The shader checks whether the given entry falls within the invalidation bounds, converts virtual coordinates into wrapped coordinates, and frees the page if needed. If a page needs to be freed, it needs to be freed both in the VPT as well as in the PPT. Before the VPT entry is reset, the information that it stores is used to calculate the coordinates of the appropriate PPT entry. These coordinates are then used to reset the appropriate entry in the MMT. Listing 3.2 shows the code of the compute shader. Notice that in addition to invalidating wrapped pages, whenever the sun moves, all pages are invalidated.

Listing 3.2 Function used to free invalidated pages.

```
1
2  // SV_DispatchThreadID.xy are the coordinates inside the VPT page
3  // SV_DispatchThreadID.z is the array level index
4  void main(uint3 svdtid : SV_DispatchThreadID)
5  {
6      const int2 clear_offset = wrapped_pages_info[svdtid.z].clear_offset;
7
8      int2 max_bounds = int2(
9          VSM_PAGE_TABLE_RESOLUTION + (clear_offset.x - 1),
10         VSM_PAGE_TABLE_RESOLUTION + (clear_offset.y - 1));
11
12     bool invalid = false;
13     invalid |= (clear_offset.x > 0) && (svdtid.x < clear_offset.x);
14     invalid |= (clear_offset.y > 0) && (svdtid.y < clear_offset.y);
15     invalid |= (clear_offset.x < 0) && (svdtid.x > max_bounds.x);
16     invalid |= (clear_offset.y < 0) && (svdtid.y > max_bounds.y);
17
18     const int3 wrapped_coords =
19         page_coords_virtual_to_wrapped(
20             vsm_page_coords.xy,
21             vsm_settings.clip_offsets[svdtid.z]
22         );
23
24     if(outside_new_window  || (vsm_settings.sun_moved != 0u))
25     {
26         uint vsm_page_entry = virtual_page_table.get()[wrapped_coords];
27         if(get_is_allocated(vsm_page_entry))
28         {
29             int2 MMT_coords = MMT_coords_from_entry(wrapped_coords);
30             meta_memory_texture.get()[MMT_coords] = 0u;
31             virtual_page_texture.get()[vsm_wrapped_page_coords] = 0u;
32         }
33     }
34 }
```

3.3.2 Dynamic Object Invalidation

Once the wrapped pages have been invalidated, I perform invalidation caused by dynamic objects. For these purposes, I utilize the dynamic mesh drawlist prepared when parsing the scene graph at the start of the frame. Before invalidating the pages, I first need to find the pages that are overleaped by the dynamic meshes. This is done by rasterizing the axis-aligned bounding box of each mesh. To rasterize a bounding box into the VPT of a given light, all corners of the bounding box are first projected by the corresponding light matrix. This gives the corners in the normalized device coordinates of the light. Then I calculate a Min-Max rectangle that bounds all points in the NDC. Following this, I convert the min-max bounds into a range of intersected pages. Finally, I iterate over this range and free every intersected page.

To perform the entire invalidation, once again, I dispatch a single compute shader. One logical mapping would be to dispatch a single thread per VPT per dynamic object. That is, every thread gets a single dynamic object and a single VPT into which this object should be rasterized. While this works, I found that this approach struggles with having a few outliers, which greatly slow the execution of this step. These outliers happen when a dynamic object intersects a larger part, or potentially the whole VPT. A single thread then has to iterate over all pages in a VPT, freeing every single one, which is very expensive.

To avoid outliers, for the purpose of this step, I split every VPT into a set of sections. I still launch a thread per dynamic object; however, instead of each thread having a single VPT to iterate over, it only iterates over a single section in the VPT. This increases both the amount of work and the number of threads that end up being launched. Previously, one dynamic object would cause dispatch equal to the number of VPTs. With this new mapping, one dynamic object causes a dispatch equal to the number of VPTs multiplied by the number of sections into which every VPT is split. The increased work comes from rasterizing a single bounding box multiple times into the same VPT. However, this approach is still faster than that initially suggested. This is because it limits the number of iterations that a single thread executes, which is the expensive part.

3.4 Page Marking

After freeing invalid pages, I move to marking the pages that will be required in this frame. First, the world-space footprint of each pixel needs to be calculated. To obtain a footprint for a single pixel, its four corners need to be projected into the world. For these purposes, I use the depth rendered from the main camera. The inverse of the view-projection matrix is used to reconstruct the world space position from the depth and uvs of the pixel.

Unfortunately, I cannot use the same depth to unproject all four corners of the pixel. I am trying to reconstruct the original world-space surface from

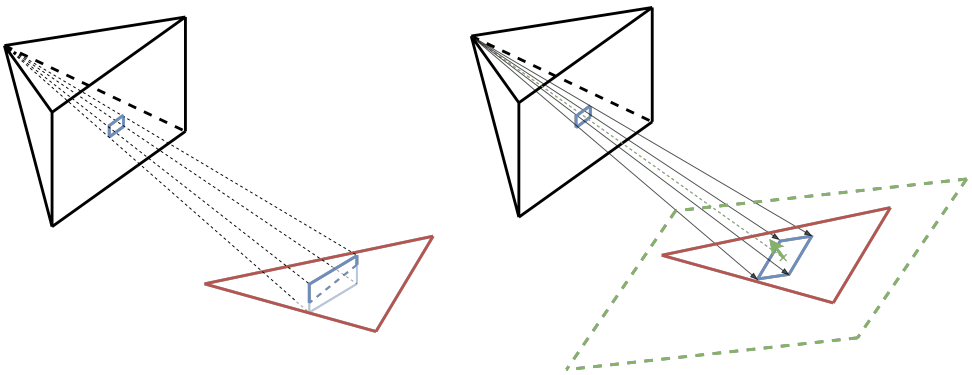


Figure 3.6 Incorrect and correct process of world-space footprint reconstruction. The pixel I wish to reconstruct the footprint for is marked in blue. Left part of the image demonstrates footprint reconstruction utilizing the same depth. This produces incorrect result with regards to the original geometry shown in red. The right part of the image shows the process of reconstructing a normal plane in green, and intersecting it with four rays. This obtains a footprint that more closely approximates the original surface.

the depth information. Using the same depth instead reconstructs a plane that is facing towards the camera. This is not good enough when my goal is to determine the footprint, which can be greatly deformed by the slope of the original surface. Instead, I use the depth and uvs corresponding to the center of the pixel to reconstruct a single world space position. At this position, I construct a plane angled according to the normal of the surface at that position. This normal is sampled from a texture that is rendered along with the depth texture. Finally, to obtain the footprint, I shoot four rays and intersect them with the constructed plane. Each ray originates at the main camera position and intersects one of the corners of the pixel. This process can be seen in Figure 3.6.

Once I obtain the world-space footprint, I can use it to determine the footprint of the original pixel in each shadow map. For directional light sources, this can be done directly in world space. The orthographic projection is defined by providing the world space size the frustum should span. From this I can calculate the world space size of each texel in the world. Once I know this, I can immediately determine the number of shadow map texels the world-space footprint covers.

Spotlights and point lights utilize perspective projection and as such cannot determine the footprint directly from world space. This makes marking the requested pages much more expensive. The marking can no longer be performed in view space and the footprint needs to be fully projected into shadowmap space. The footprint is then given as the bounding rectangle of these projected corners. Lastly, from the footprint I calculate the desired clip or mip level. This is given by the following formula:

$$\text{level} = \max\left(\left\lceil \log_2\left(max(T_W, T_H)\right)\right\rceil, 0\right) \tag{3.9}$$

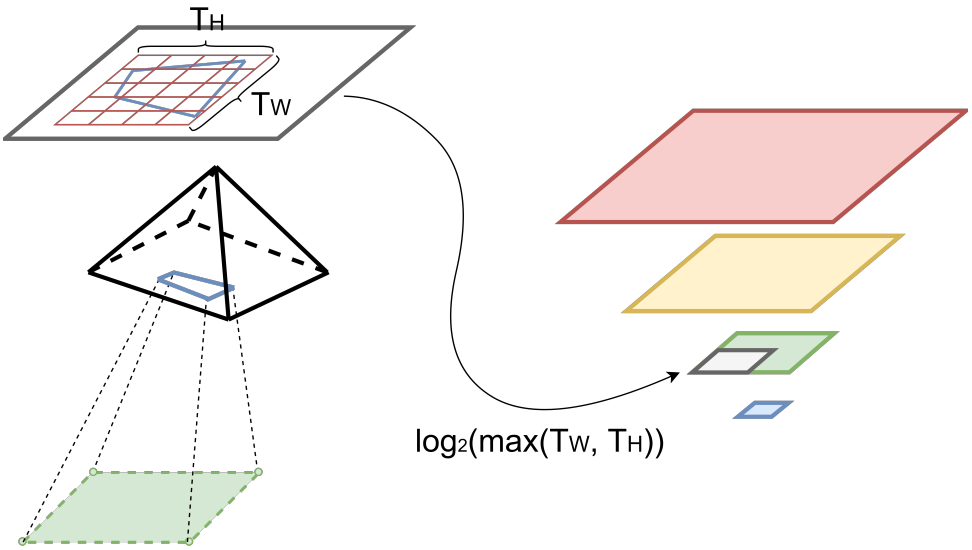


Figure 3.7 Visualization of the marking process for a single spot light. The world-space pixel footprint, shown in green, is projected into the uv space of the light. This gives the uv footprint, shown in blue which is then bounded by a rectangle shown in red. The width T_W and height T_H are then used to obtain the correct mip level of the VPT shown on right. Finally a page corresponding to the uvs of the center of the bounding rectangle is marked. This page is shown in gray.

Where T_W and T_H are the width and height of the footprint in texels. Once the level is obtained, the page corresponding to the uv coordinates at center of the footprint is marked. This process can be seen in figure 3.7. As already mentioned before, point lights are represented as a cubemap emulated by six spotlights. Because of this, the cubemap face needs to be manually determined first before the marking process can begin. This makes the marking process for point lights the most expensive one.

3.4.1 Processing Marked Page

Once a marked page has been identified, it needs to be processed. How a page is processed is determined by its current state. A marked page that is already allocated needs to be flagged as visible. In addition, the MMT entry backing the page also needs to be flagged so that the state remains consistent. Marked pages that are not currently allocated need to perform an allocation request. An allocation is requested by appending a request to a dedicated buffer. The request itself is represented by four integers that can fully identify the respective VPT page. That is, the allocation request holds the x and y coordinates of the page, the array layer index of the VPT, the mip level of the VPT in case the page belongs to a point light or spotlight.

To mark required pages for all pixels, I again utilize a single compute shader dispatch. I launch a single thread to process a single pixel. This ensures that all pixels are marked in parallel; however, this unfortunately introduces

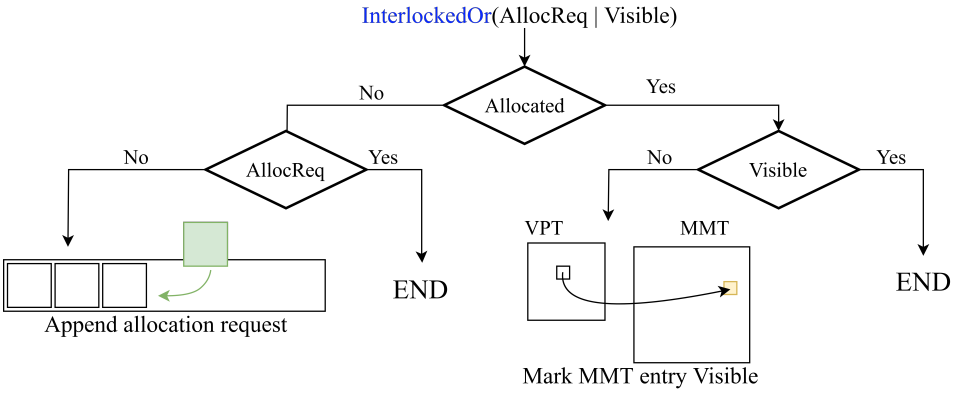


Figure 3.8 Flowchart depicting processing of a single marked page. The process starts with atomically setting the **AllocReq** and **Visible** bits. This operation returns the previous state of the page entry which is used in the following logic. If a thread is first to set the **AllocReq** bit for an unallocated page, it will insert the allocation request. Similarly, if a thread is first to set the **Visible** bit for an allocated page, it will mark the corresponding MMT entry as visible.

issues. Multiple pixels can mark the same page. An unallocated page marked by multiple pixels would result in the same request being inserted multiple times. This would both waste space and complicate further processing as these duplicates would need to be removed. Instead, all allocation requests should be unique and present only once.

To achieve this, I utilize atomic operations in combination with two status bits stored in each VPT entry. Every time I am processing a marked page, I start by using an *InterlockedOr* operation to set the entries' **AllocReq** and **Visible** bits. In addition to setting these bits, the *InterlockedOr* operation also returns the previous value of the entry. I use this to determine if this thread is the first to mark this page. If the thread is first, it can safely insert an allocation request, knowing that it will be unique. All other threads will see the **AllocReq** bit as already set and thus will not insert additional requests. The same mechanism is used to mark the MMT entry as visible when a page is allocated. While marking the MMT as visible multiple times would not produce any issues, it still saves bandwidth and thus increases efficiency of the shader. Figure 3.8 shows the complete process as a flow diagram.

3.4.2 Light Culling

The cost of the marking step increases linearly with the number of lights that should be marked. This is mainly due to the number of projections that need to be performed. The number of projections can be reduced by calculating the world-space footprint once and reusing it for all lights. However, the same cannot be done to obtain the footprint in the shadowmap space for point lights and spotlights. The footprint for each of these lights needs to be obtained individually, without the possibility of reusing the computation.

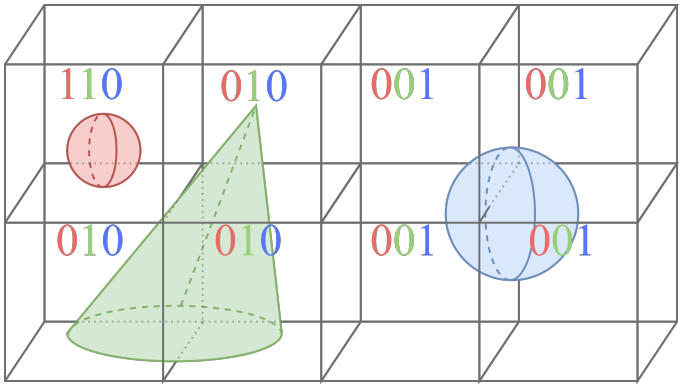


Figure 3.9 A simple example of a single slice of a light grid. Each cell holds a bitmask which represents influence of individual light sources. When a light source affects a cell the bit corresponding to its index is set.

Because spotlights and point lights make up the majority of light sources in typical scenes, this issue needs to be addressed.

Nothing can be done about the fact that I need to project into each light individually. However, when a given light is determined to have no influence on a specified world position, the marking process for that light can be skipped entirely. This is called light culling, and it plays a fundamental role in the efficiency of the marking step.

To determine which lights affect a given world position, I employ a world-space grid. Each cell of this grid contains a 256-wide bitmask where each bit corresponds to a single light source. A set bit at the index i means that the cell is influenced by the i-th light source. This grid is centered around the main camera and is updated at the beginning of each frame. The implementation of the grid update process was already present in the framework and was not developed by me. Only point light and spotlight indices are present in the grid, as directional lights have infinitely large influence, and thus would always be present in each cell of the grid.

In the marking shader, I first mark all directional light sources. Following this, I use the world space position of the center of the pixel to find the corresponding cell in the light grid. Then I extract all the sources of light that affect the cell and perform a marking pass for each one. The maximum amount of lights that need to be marked per pixel is no longer given by the total amount of lights in the scene. Instead, it is given by the number of lights that affect the same pixel. Due to the largely local influence of spot and point light sources, this number is much lower. A simplified example of a light grid slice can be seen in Figure 3.9.

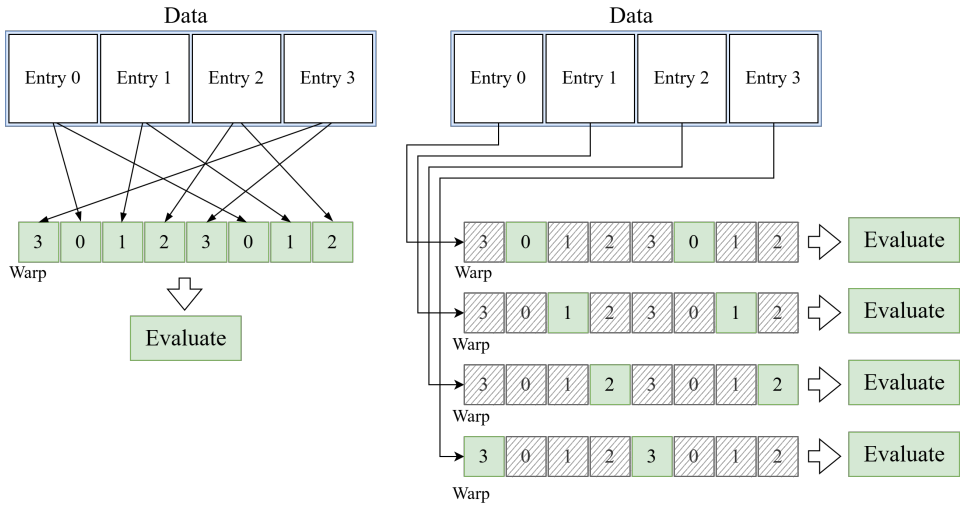


Figure 3.10 Scalarization of a single warp with eight threads. To evaluate every thread needs an individual entry stored in global memory. On the left is the usual approach, where each thread fetches required entry and stores it in unique registers. Following this all threads evaluate the function simultaneously. On the right is scalarized version of this process. Each entry is only fetched once into registers that can now be shared by all threads withing the warp. The function is then evaluated once for each entry. A thread is active only when its corresponding entry is currently being processed.

3.4.3 Scalarization

Preventing duplicate operations with **AllocReq** and **Visible** bits make the marking process much more efficient. That said, atomic operations are still very expensive, especially when large numbers of threads mark the same page. Because threads process pixels that are spatially coherent, there is a high probability that they mark the same page. This means that while only a few atomic marking operations might be required, the operation still needs to be performed multiple times, once for each thread. To reduce the number of these operations, I utilize a concept called execution scalarization. Scalarization is a process that takes advantage of specific properties of the GPU hardware. A full analysis of GPU architecture is outside of the scope of this work; however, for completeness, I provide a brief explanation of the key concepts in the following paragraphs.

Individual threads on a GPU are scheduled in small groups called warps on NVIDIA and wavefronts on AMD. For the rest of this section, I will utilize the naming by NVIDIA and thus refer to these groups as warps. Threads within a single warp typically execute the code in a lockstep. This means that all threads execute the same code at any given time, which is also called a single instruction multiple threads model (SIMT).

Whenever two parts of a warp disagree on the code that should be executed, the warp is said to be diverging. This typically occurs when threads within a warp evaluate a conditional statement differently, leading to different execution

paths. In such a case, both code paths will be evaluated in sequence, with parts of the warp active only when the code corresponding to its path is being executed. Divergence also applies to data dependencies. Two threads in a warp may wish to load data from different offsets in a single buffer or from completely different buffers altogether. This increases the number of registers that need to be allocated by the compiler, which in turn hurts occupancy and limits the number of warps that execute at the same time.

A second key property of warps that I would like to highlight is the highly efficient means of communication within a warp. Communication between arbitrary threads on the GPU is typically very problematic. It involves either writes to shared memory and expensive barriers, or atomic writes to global memory, which are even slower. Communication between threads in a single warp, however, is supported directly by hardware and is nowhere near as expensive as the above methods.

Scalarization utilizes these to unify the code and data dependencies within a warp. The first step is to collect the number of permutations, that is, the number of unique execution paths or data points within a single warp. Instead of allowing divergent execution of a warp, the shader iterates over individual permutations, manually masking threads not belonging to the active invocation. At first sight, this might seem like the same thing that hardware does; however, there is a key difference. With this approach, the compiler knows that the entire warp will follow a unified path. This gives a much larger opportunity to optimize the code. Instead of having to allocate one register per thread to account for unique values across the entire warp, only a single register, shared by the entire warp, can be allocated. Furthermore, this improves the pattern of accessing values, allowing for a more efficient loading of these values from memory. It is important to note that the effectiveness is highly dependent on the number of permutations being small. For a large number of permutations, a scalarization loop will be executed many times with a low number of active threads in each iteration. Figure 3.10 visualizes a simple case of scalarization.

In most cases, there will be a limited number of unique pages within a warp. This results in the same page being processed by multiple threads, increasing the contention of the atomic operation and slowing down the execution. Instead of each thread processing the marked page, the unique pages within a warp are identified. A single thread is then elected to process each unique page. The scalarized code can be seen in Listing 3.3.

3.5 Page Allocation

The page marking step produces the buffer that contains all unique allocation requests. This is consumed by the allocation step, which acquires the necessary pages from the PPT and assigns them to each allocation request. Additionally, it clears all newly allocated pages, removing data that have potentially been left by the previous allocation. To do this efficiently, page allocation is split into three steps, which will be described in the next sections.

Listing 3.3 Scalarized function used to process marked pages.

```
1
2  // SV_DispatchThreadID.xy are the coordinates of the processed pixel.
3  void main(uint3 svdtid : SV_DispatchThreadID)
4  {
5      uint4 point_light_mask = query_lights(world_pos);
6      Footprint ws_footprint = unproject(depth, normal);
7
8      // Loop through all lights.
9      while (any(point_light_mask != uint4(0)))
10     {
11         // Obtain the next light in the mask.
12         // Also modifies the mask.
13         uint light_idx = extract_light(point_light_mask);
14
15         // Obtain the VPT coordinates from the footprint.
16         int4 VPT_coords = project_into_light( light_idx, ws_footprint );
17
18         bool thread_active = true;
19         bool first_to_see = false;
20
21         // Scalarization loop.
22         while(thread_active)
23         {
24             // Obtain the coordinates of the first active thread.
25             const int4 sg_uniform_coords = WaveReadLaneFirst(VPT_coords);
26
27             // All threads with the same page enter.
28             if(all(equal(sg_uniform_coords, VPT_coords)))
29             {
30                 // Only one thread will process the marked page.
31                 if(WaveIsFirstLane()) { first_to_see = true; }
32
33                 // The page for active threads has been processed.
34                 // These threads can exit the loop.
35                 thread_active = false;
36             }
37         }
38
39         // One thread per page performs the processing.
40         if(first_to_see)
41         {
42             process_marked_page(VPT_coords);
43         }
44     }
45 }
```

3.5.1 Classification

The process begins by classifying PPT pages into three categories **Free pages**, **Allocated pages**, and **Cached pages**. As already described in a previous chapter, free pages are pages that currently do not back any VPT entry. In contrast, both cached and allocated pages are currently backing a VPT entry. The difference between those is that a cached page has not been marked as

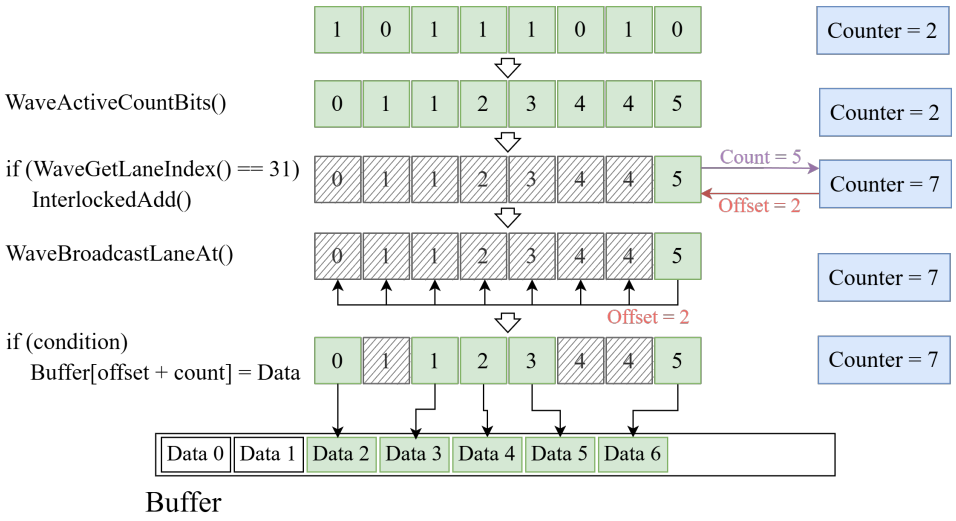


Figure 3.11 Scalarization of the writeout step visualized on a single warp with eight threads. Each thread holding value 1 in the first step wants to write out data into the buffer. First, an exclusive sum of the threads that want to write out is performed. Following this, the last thread reserves slots in the buffer for the entire warp. Next, threads that are writing out use the exclusive sum to calculate the offset into the buffer. Finally, once the offset is obtained, the write is performed.

visible by the marking process. It can thus be reclaimed and the memory used to back another, currently visible and not allocated page. Allocated pages cannot be freed and as such their memory is effectively constant. Because of this, they are ignored by the later steps of the algorithm.

The classification pass outputs two buffers, one containing free pages and one containing pages that are cached. During the marking pass, the visible bit is written for both the VPT and MMT entries. To classify all pages, only the MMT needs to be analyzed without the need to read individual VPTs. This is much more efficient, as the majority of the VPTs will contain unallocated entries. The classification is again performed by a single compute shader dispatch, which launches a single thread per MMT page. Every thread reads the corresponding MMT entry and determines the category into which this entry falls. If the entry corresponds to a cached or a free page, its coordinates are appended to the buffer of free or cached pages, respectively.

Because all threads run in parallel, there will be multiple threads attempting to append information into the same buffer. Without synchronization, this would result in a data race, which would produce undefined results. To synchronize the writes to the buffers, I utilize an atomic version of the linear allocation scheme. Both buffers store a single counter that denotes the offset at which the buffer was last written. Each counter starts at the value of 0. Whenever a thread wishes to write to a buffer, it starts by atomically increasing the appropriate counter. The previously stored value, returned by the atomic addition, is now reserved by this thread. Finally, the thread writes its data into the reserved slot.

This introduces a high atomic contention on the two counters, which in turn slows down the execution time. Similarly to the marking pass, communication within a wave is utilized to make this process more efficient. Instead of each thread attempting to reserve a slot, the number of threads in the warp that wish to reserve a spot in each buffer is counted. A single thread then executes the atomic operation and reserves slots for all threads. Each thread then determines the offset within the allocated window and writes the page information. The code for this is shown in Listing 3.4 and Figure 3.11 visualizes this process.

Listing 3.4 Scalarized function used to write classified pages into a buffer.

```
1
2  void write_classified_pages(
3      bool condition,
4      int2 PPT_coords,
5      Data buffer )
6  {
7      // Returns the number of threads with index
8      // lower than current thread with condition set to true.
9      const uint local_index = WavePrefixSum(condition);
10
11     uint allocation_offset = 0;
12
13     // Last thread allocates space for all threads.
14     if(WaveGetLaneIndex() == 31)
15     {
16         // Add this threads result to the exclusive sum.
17         uint page_count = local_index + uint(condition);
18
19         allocation_offset = InterlockedAdd(buffer.counter, page_count);
20     }
21
22     // All threads read value from last thread.
23     allocation_offset = WaveBroadcastLaneAt(allocation_offset, 31);
24
25     // Each thread writes data to its own spot determined by the
26     // order obtained before.
27     if(condition)
28     {
29         buffer.data[allocation_offset + local_index] = PPT_coords;
30     }
31 }
```

3.5.2 Allocation

The allocation step performs a simple matching between the two buffers produced by the classification pass and the allocation request buffer. For this, I utilize another compute dispatch, launching one thread for one allocation request. The number of allocation requests is known only on the GPU. To avoid expensive read-back to the CPU memory, I employ indirect dispatch. The classification pass described above reads the number of allocation requests and fills out the indirect dispatch structure. In addition to filling out the

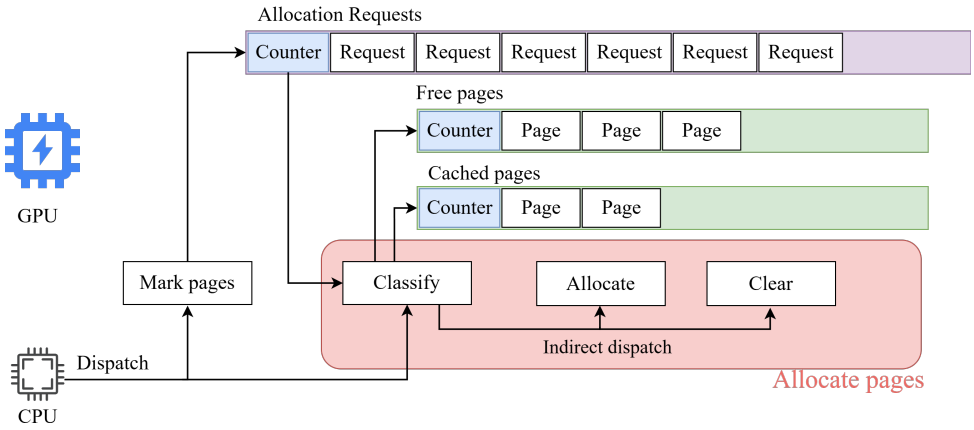


Figure 3.12 Visualization of the dependencies of the allocate pages step. The marking step is launched from the GPU just like the classfication part of the allocate pages. As a last step of page classification, the number of allocation requests produced by the marking pass is used to fill indirect dispatch structures. These are used to launch an appropriate number of threads for both the allocation as well as clearing of newly allocated pages.

indirect dispatch for the allocation, the classification also prepares the exact same structure for the clear pass that follows the allocation. This is visualized in Figure 3.12.

The counters of free and cached page buffers are used to partition the threads performing the allocation. This is done utilizing the thread index. All threads with an index smaller than the count of free pages allocate from the free page buffer. The rest of the threads use the same principle to allocate pages from the cached buffer. This ensures that cached pages are reallocated only when necessary.

Whenever cached pages need to be reallocated, they need to be freed first. This process is the same as the one described for page invalidation. The MMT entry is used to look up the corresponding VPT page coordinates. These coordinates are used to free the previously owning VPT page before assigning the new owner. When there are not enough pages to satisfy all allocation requests, the **AllocFail** bit is set for those pages that did not obtain an allocation. The **Dirty** bit is set for all pages that have been successfully allocated. This process can be seen in Figure 3.13.

3.5.3 Clear

The last step of this stage is to clear all newly allocated pages. As already said, this is performed by an indirect compute shader dispatch. In contrast to the previous part, instead of dispatching a single thread per page, one allocation request is responsible for launching a group of threads. That is, for each allocation request, the number of threads equal to the number of pixels in each physical page are launched. All threads in this group read the allocation status of the corresponding page. When the allocation was

successful, every thread clears a single pixel in the PPT. I again perform scalarization to fetch the allocation request and VPT entry only once per wave. The code of the shader performing the clear can be seen in Listing 3.5.

Listing 3.5 Shader performing clear of all newly allocated pages.

```
1
2  // SV_GroupThreadID.xy are the indices inside of each physical page.
3  // SV_DispatchThreadID.z is the index of the allocation request.
4  void main(
5      uint3 svdtid : SV_DispatchThreadID,
6      uint3 svgid : SV_GroupID,
7      uint3 svgtid : SV_GroupThreadID
8      )
9  {
10     AllocRequest request = allocation_requests[svdtid.z];
11     uint VPT_entry = 0;
12
13     // Only one thread will fetch the VPT entry.
14     if(WaveIsFirstLane())
15     {
16         if(request.point_spot)
17         {
18             // We are clearing a point or spotlight page.
19             VPT_entry = point_spot_VPTs[request.mip].get()[request.coords];
20         }
21         else
22         {
23             // We are clearing a directional light source page.
24             VPT_entry = directional_VPTs.get()[request.coords];
25         }
26     }
27
28     // Broadcast the entry to all other threads in the wave.
29     VPT_entry = WaveBroadcastLaneAt(VPT_entry, 0);
30
31     // If the page did not receive an allocation do nothing.
32     if(get_failed_alloc(VPT_entry)) { return; }
33
34     // Calculate the threads coordinates within PPT.
35     int2 memory_page_coords = MMT_coords_from_VPT_entry(VPT_entry);
36     // Start by calculating the top left corner of the page in PPT.
37     int2 PPT_corner_coords = memory_page_coords * VSM_PAGE_SIZE;
38     // Multiple work groups are clearing the same page.
39     int2 wg_offset = svgid.xy * CLEAR_PAGES_XY_DISPATCH;
40
41     // Write the cleared value.
42     uint2 PPT_coords = PPT_corner_coords + wg_offset + svgtid.xy;
43     PPT.get()[PPT_coords] = 0.0f;
44 }
```

3.6 Page Drawing

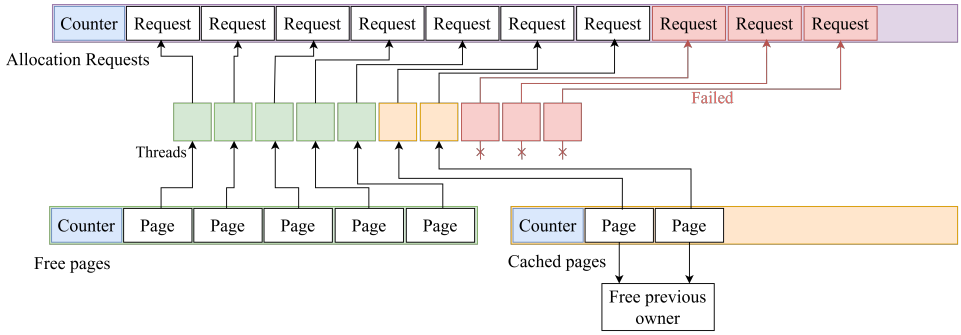


Figure 3.13 Visualization of the dependencies of the allocate pages step. The marking step is launched from the GPU just like the classfication part of the allocate pages. As a last step of page classification, the number of allocation requests produced by the marking pass is used to fill indirect dispatch structures. These are used to launch an appropriate number of threads for both the allocation as well as clearing of newly allocated pages.

The last phase of my algorithm populates newly allocated pages with depth data. To make this process efficient, I utilize mesh shaders. As mentioned during the scene description, the drawlists contain a list of mesh instances. However, mesh shaders require individual meshlets. This poses a significant challenge when scheduling work for the GPU. Due to the variable number of meshlets present in each mesh, I do not know how large the mesh shader dispatch should be. Because of this, the drawing starts with an additional drawlist expansion step.

Once the dispatch size is known, processing of individual meshlets can begin. Doing this naively would quickly become prohibitive due to the sheer amount of geometry that needs to be processed. In order to maintain real-time frame rates, the geometry needs to be culled to decrease the load on the rasterization stages. The culling is performed hierarchically to eliminate large groups of non-visible or irrelevant objects early in the process, reducing the computational overhead of checking each individual element.

3.6.1 Drawlist Expansion

Work expansion performs further flattening of the draw list, unwrapping meshes into a list of meshlet instances. Due to a relatively high number of meshlets present in a scene, this step is performed on the GPU. Storing every meshlet instance individually in a buffer would consume excessive memory, making it impractical for real-time applications. Additionally, processing them in a linear, unstructured manner would be computationally inefficient. To solve both of these issues, I utilize a data structure called the *power of two buffers*.

As the name suggests, this structure consists of a set of individual buffers.

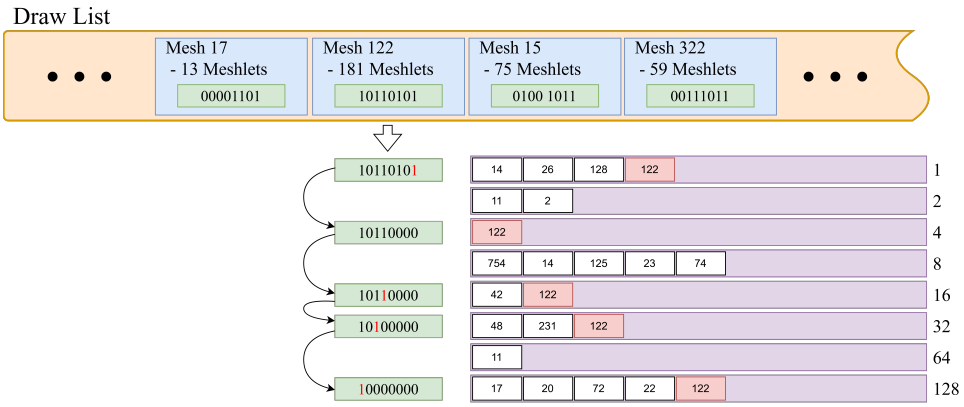


Figure 3.14 The process of draw list flattening during work expansion for a single mesh instance. First, the number of meshlets is read from the underlying mesh. Following this, the lowest bit from the meshlet number is extracted. A new entry, storing the index of this mesh, is created in a buffer corresponding to the index of the lowest bit. This bit is then masked out and the same process is repeated iteratively, until the number of meshlets falls to zero.

An entry in each buffer represents a set of meshlets belonging to an individual mesh. The number of meshlets represented by each entry is given by the buffer in which this entry is present. With an increasing buffer index, the number of meshlets represented by each entry doubles. This is where the "power of two" in the name comes from. That is, the first buffer represents single meshlets, the second buffer represents two meshlets, the third buffer represents four meshlets, and so on.

The meshlet expansion is performed in a compute shader, where a thread maps to a single meshlet instance in the target drawlist. Every thread reads the number of meshlets the mesh instance contains and distributes them across corresponding buffers. This is done by looping over the number of meshlets, extracting the first low bit, and writing an entry to the buffer given by the index of this bit. Before continuing to the next iteration, this bit is masked out from the meshlet number. This process can be seen in Figure 3.14.

The process of sorting into power of two buffers greatly increases the processing speed and reduces the memory required to store expanded draw lists. That said, it still needs to account for the worst case. For example, I need to allocate enough space for a scenario where all meshes in a scene contain exactly sixteen meshlets. Because the majority of the scenes have the meshlet numbers somewhat uniformly distributed, this wastes some memory. This is normally not that severe, as the wasted memory is still marginal compared to storing meshlets individually. For VSMs, however, the scene is drawn potentially hundreds of times per frame. Creating a separate power of two buffers for each of these draws would amplify the wasted memory.

To avoid this, I reuse the same power of two buffers for multiple draws. Because every power of two buffer expansion maps directly to a set of draw calls, this gives us the added benefit of dispatching fewer draw calls. There

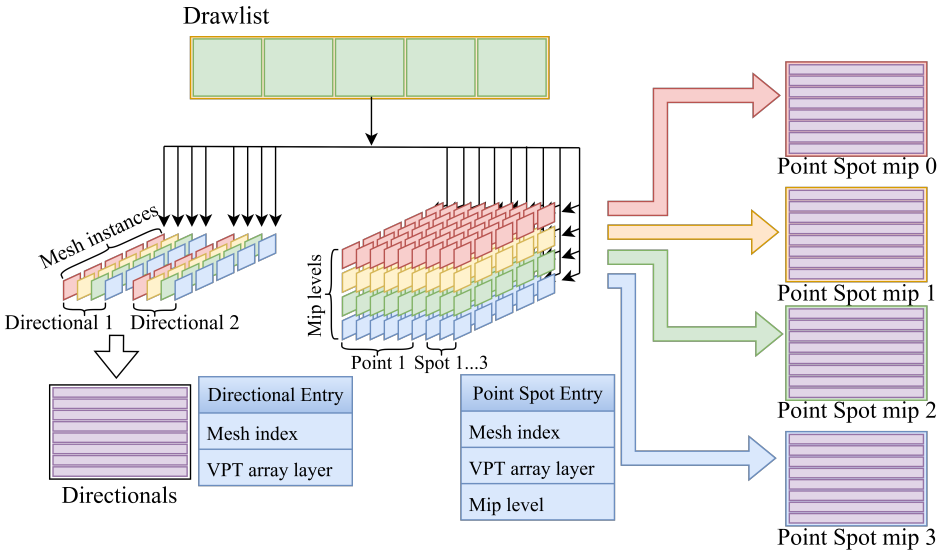


Figure 3.15 The dispatch scheme for a single draw list seen at the top of the image. For directional lights I launch a single thread for each entry for each VPT. These share single power of two buffers. For point and spot light, I launch one thread for each entry per VPT per Mip level. Every mip level has its own power of two buffer, however these are shared across all point and spot lights.

are slight differences in how individual light types are handled. As already described, every clip map of a directional light has the same resolution. I pack meshlets for all clip maps into a single meshlet list, resulting in a single indirect draw to render all clip maps at once.

Spotlights and point lights, however, use a mipmap chain, which has varying resolution. Because of this, I unfortunately cannot reuse the same power of two buffers as individual draw calls are required for each resolution. In contrast to directional lights, however, a typical scene contains multiple point lights and spot lights. This allows me to expand meshes for non-directional lights by the desired resolution. Assuming the same virtual resolution, only $log_2(resolution)$ power of two buffers and draw calls are required to draw shadows for all non-directional lights.

To make this work, the previous approach needs to be slightly modified. For work expansion, launching a single thread per mesh is no longer sufficient. Instead, for directional lights, a single thread per mesh per VPT is launched. For point lights and spotlights, a single thread per mesh per VPT per mip level of each VPT is launched. I also need to include additional information in the power of two entries themselves. In addition to storing the index of the respective mesh, the index of the VPT and, for non-directional light sources, the corresponding mip level need to be stored. The complete scheme of this process is shown in Figure 3.15.

Once a power of two buffer is prepared, it is used to dispatch a set of indirect draw calls. This is done by multiplying the number of entries in each buffer by the power of two it represents. A single task shader is launched for

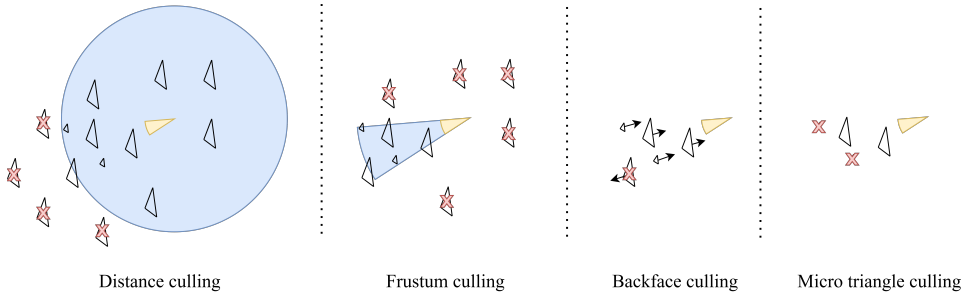


Figure 3.16 The process of culling with the standard culling methods. First, distance culling removes all objects that lie outsite of the lighs area of influence. Second, frustum culling removes all objecst that lie outside of the view frustum of the light. Following this, all backfacing triangles are culled. Finally, all triangles that would not result in any frament shader invocation are discarded.

each meshlet in every buffer. After processing the meshlet, the task shader dispatches a group of mesh shaders. Finally, each mesh shader processes a subset of the meshlets' vertices and writes the resulting triangles.

3.6.2 Geometry Culling

Sharing the same power of two buffers reduces the number of draw calls but does not reduce the amount of processed geometry. To allow for highly detailed scenes with large amounts of geometry, culling needs to be utilized throughout the pipeline. The proposed culling scheme closely matches the scene representation and the logical steps of the drawing pipeline. First, during drawlist expansion, mesh instances present in the draw list are culled. Only entire meshes are culled as part of this pass, and individual meshlets are not considered. Once a part of a mesh is determined visible, all of its meshlets are written in the power of two buffers.

Second, in the task shader, the culling of individual meshlets is performed. Every task shader thread processes a single meshlet instance. The task shader threads only launch mesh shader groups for visible meshlets. Similarly to meshes during drawlist expansion, meshlets that are not visible are thrown away. Lastly, during meshlet processing, individual triangles are culled. Although culling individual triangles is generally discouraged, it is critical in achieving the fastest possible results for VSMs. This is mainly due to the relatively expensive nature of fragment shader invocations.

Various standard methods were used to determine visibility. The fastest and most conservative are utilized first, and only after these determine the object visible are more expensive and precise culling methods employed. There are two specific methods that apply only to triangle culling: micro-triangle and backface culling. Except for those two, all other culling methods are utilized during each culling step described above. To cull meshes and meshlets, the axis aligned bounding boxes of the respective objects are used. Triangles are culled directly, using their vertices.

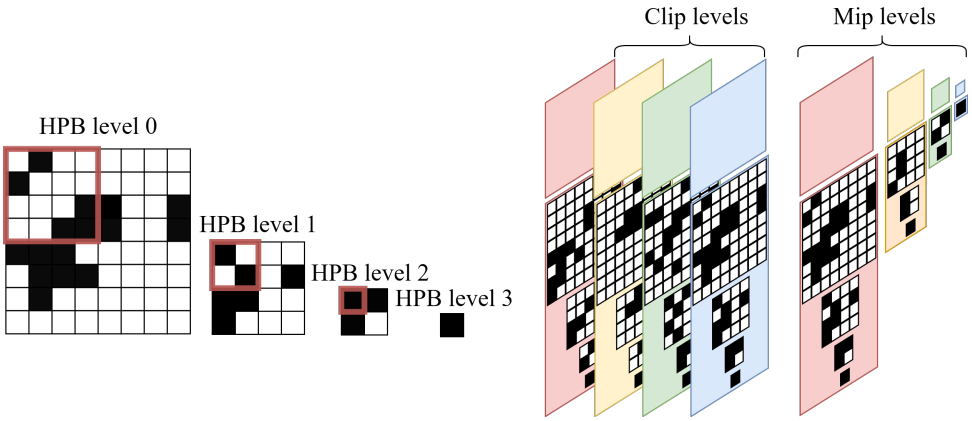


Figure 3.17 On the left four levels of Hierarchical Page Buffer (HPB). Black tiles denote dirty pages, that is, pages that will be drawn into this frame. Each level is a 2×2 logical "OR" reduction of the previous level. The red square denotes how areas across HPB levels map upon each other. On the right visualzation of HPBs for spot directional lights and spotlight. Directional light has four Clipmap levels, each represented by its own VPT and, as such, four full HPBs are constructed. Spotlight has four mipmap levels represented by the mipchain of a single VPT. As a result, the HBPs are smaller for each decreasing mip level.

For point lights and spotlights, distance culling is the first utilized method. Because the radius each light affects is known, objects that lie outside of this area can be discarded as their shadow will contribute nothing to the final result. This method is not valid for directional light sources. The radius of influence is infinite and as such would result in no objects being culled. Following distance culling, frustum culling is used, which is applicable for all types of light. The bounds of each object are checked against all frustum planes of the corresponding shadow camera. If an object lies outside the visible frustum, it is discarded. Next, the two triangle culling methods are used, starting with backface culling. This step is typically performed by the rasterization pipeline; however, with mesh shaders, this must be done manually. The second, called micro-triangle culling, removes all triangles that would not result in any fragment shader invocation. All standard culling methods can be seen in Figure 3.16.

Lastly, a method specialized for VSMs called Hierarchical Page Culling is utilized. This method is applicable to all light types as well as all three culling stages. This will be described in the next section.

Hierarchical Page Buffer

As already discussed, in most cases only a very limited set of pages is visible at one time. Furthermore, most of the visible pages will be cached and will not need to be redrawn. This makes the set of dirty pages, that is, pages that need to be drawn in any given frame, very small. Knowing this, all geometry that does not overlap a dirty page will not contribute anything to the final result and as such can be discarded. To make this process as efficient as

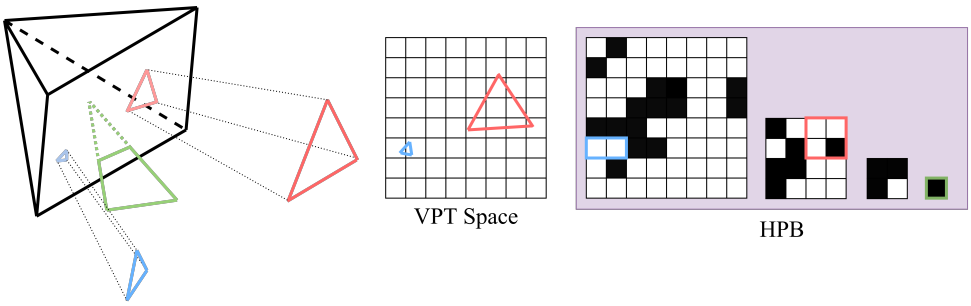


Figure 3.18 Culling performed with HPB. First individual objects are projected into the VPT space of the given light. Based on their footprint, the appropriate level of HPB is selected, such that at most four texels of the HPB need to be sampled. The footprints of objects intersecting the near plane of the lights camera cannot be determined. Because of this, these always sample the last level of the HPB. In this example scenario, blue triangle would end up being removed as it does not intersect any dirty page.

possible, a data structure called a Hierarchical Page Buffer(HPB) is used.

This structure and the culling process related to it are inspired by the way a hierarchical Z buffer [GKM93] is used for occlusion culling. Similarly to a hierarchical Z buffer, the HPB consists of multiple levels. Each level in the HPB is constructed as a 2×2 logical "OR" reduction of the level directly below it. That is, each level is half as large as the previous one. The construction starts with the lowest level, which is directly initialized by the **Dirty** bits of the corresponding VPT. As such, an HPB is built for each VPT. For directional light sources, an HPB is thus built for each clip level. Spot lights and point lights utilize mip maps and have a single VPT for all mip levels. Individual mip levels still need to be culled separately, and thus an HPB needs to be constructed for each level of the VPT.

Because HPBs are also used to cull meshes in the drawlist expansion step, they need to be constructed as a first step of the drawing process. This is done with a single compute dispatch. The construction process is heavily inspired by the single pass downsampler (SPD) released as a part of the FidelityFX SDK maintained by AMD [AMD25]. The original SPD processes the texture in patches of 64×64 texels, which are then synchronized using atomic variables. By limiting the maximal size of the VPT to the resolution of a single patch, synchronization can be omitted, and each HPB can be constructed with a single workgroup. This greatly simplifies the implementation as the majority of the complexity of SPD lies in the atomic synchronization.

Every thread within a workgroup processes a 4×4 region of the underlying VPT. As such, the size of each workgroup is 16×16 threads. Special care needs to be taken when constructing the HPBs for point lights and spotlights. Although the resolution of the 0th mip is 64×64 texels, higher mip levels have lower resolution. Because of this, the downsampling pass needs to terminate early to avoid writing out of bounds of the current HPB.

To cull using the HPB, the axis-aligned bounding box of the object is projected into the texture space of the light. From the projection area, the

appropriate level of the HPB is calculated. The level is determined so that only four texels need to be sampled in order to cover the whole projected area. If none of the texels are marked as dirty, the object is culled.

Extra care needs to be taken when projecting objects into point lights and spot lights. Because of perspective projection, an object that is behind or intersecting the near-plane results in incorrect results after applying the perspective division. For these objects, the highest level of the HBP is sampled instead. This is equivalent to a situation where an object would span the entire frustum of the light. Figure 3.18 shows the culling process.

Chapter 4

Results and Discussion

In this chapter, I will show the results and discuss the performance of my method. All measurements were taken with an NVIDIA GeForce RTX 4070 Ti SUPER GPU and an AMD Ryzen 7 7800X3D CPU. I will start with qualitative evaluation, comparing different configurations with a ray-traced reference. Following this will be a section that analyzes the performance. I used three test scenes to evaluate the quality and performance of my implementation. These were the Lumberyard Bistro, McQuire Archive San Miguel, and Unreal Engine Sun Temple. The configuration for each scene can be seen in Table 4.1.

4.1 Qualitative Analysis

	Bistro	**San Miguel**	**Sun Temple**
Triangles	4,144,001	9,895,141	608,161
Point lights	33	26	13
Spot lights	26	0	0
Directional lights	1	1	1

Table 4.1 Parameters of the three scenes utilized for qualitative and performance testing in our application.

In this section, I will evaluate the quality of the shadows produced by the implemented algorithm. For all results shown in this section, the size of the PPT was set to 16384×16384 texels. This was done to achieve the best looking results without worrying about memory consumption. That being said, all settings used were still within reasonable bounds and run in real-time.

4.1.1 Final Results

The first results that I would like to present can be seen in Figure 4.1. These are final images after shadow filtering and supersampling applied. Supersampling for the means of anti-aliasing was chosen due to its simplicity, but the same result could be achieved with other anti-aliasing methods.

Figure 4.1 Images obtained by my implementation. From top to bottom Amazon Lumberyard Bistro, McQuire San Miguel, and Unreal Engine Sun Temple. Every image was taken with supersampling enabled using 16 shadow map samples per pixel.

The purpose of these images is to show a fully polished look that can be achieved with VSMs. For all three images, virtual resolution of 4096×4096 texels for directional light sources and 2048×2048 texels for spotlight and point light sources.

Next, I show comparisons of different quality configurations as well as reference results obtained by ray tracing. The first of the tested configurations was the same as the one used to obtain the first set of images. That is, the resolution of 4096×4096 texels was used for directional light and 2048×2048 texels was used for spotlights and point lights. For the second configuration, the resolution for all light sources was halved. That is, the resolution of 2048×2048 texels was used for directional light and 1024×1024 texels was used for point lights and spotlights. Halving the resolution is a good trade-off between the obtained quality and performance.

For this test scenario, only a single sample was taken for each light. This was done to ensure consistency between ray tracing and shadow mapping. In addition, it also better highlights the artifacts produced by shadow mapping and shows the difference between individual shadow map resolutions. Although shadow map filtering is used by pretty much all shadow mapping implementations, the quality of unfiltered shadows still strongly correlates with the quality obtained after shadow filtering. Furthermore, this also helps to separate the quality of the shadow map from the quality of the filtering algorithm.

For the ray-traced reference a single ray was shot towards the origin of the light. To achieve similar results between ray tracing and rasterization, the near plane of the perspective projection used during rasterization needed to be emulated. To do this, I added a distance equal to the near-plane to all shadow ray hits when evaluating if a ray reached the light. This is not a perfect emulation, because it discards geometry in a sphere around the light source. In comparison, the near-planes of point-light faces discard geometry in a cube around the light. As such, there might still be some slight differences caused by this between the reference and the rasterized shadows.

4.1.2 Bistro

In Figure 4.2 we can see two different view points of the Bistro scene. The first row shows the main square, which is the part that has the most lights visible at once. The lower resolution shadow map does not have enough texels to smoothly represent such a thin object. This is the most visible on the cables holding the light bulbs. This results in parts of the cable not casting any shadow, resulting in a dotted or dashed shadow, as can be seen in the highlighted rectangle on the right, or a visibly jagged shadow, as can be seen in the highlighted rectangle in the center. This is improved by the higher-resolution shadow maps, which are very similar to the reference.

In the second row, we can see a close-up on display cases illuminated by spotlights. Because spotlights, similarly to point lights, use a perspective projection, the texel density of the shadow map decreases with increasing distance from the light source. This can be seen in the rectangle highlighted

Figure 4.2 In the top row a view of the square, which contains the most visible lights for all tested configurations. Lower resolution shadow maps struggle to reproduce the detail of thin geometry such as the cables holding the light bulbs. In the bottom row a close up on a display case, shows the effect of decreasing shadow quality with increased distance for a spot light illuminating the display case.

on the left. In this view, even the higher-resolution shadow maps are not enough to hide all discontinuities. The highlighted rectangle on the right shows the effect of lowering directional light resolution, which results in visible texels on the eaves.

4.1.3 San Miguel and Sun Temple

Figure 4.3 shows the views of the other two scenes tested. In the upper row, we can see an image taken from the San Miguel scene. The rectangle in the top left highlights blocky looking shadows cast on a wall by leaves. This is in part due to the distance from the point light that casts the shadow. Similarly to spotlight artifacts discussed in the previous image, point-light shadows lose resolution as the distance from the source increases. Foliage shadows are often hard to represent by shadow-mapping techniques. This is mainly due to the high-frequency details, which are difficult to capture with the limited resolution of the shadow map. The rectangle in the middle of the image highlights the issues caused by a directional shadow stretched on a pillar. In general, shadow maps suffer from the worst artifacts when light rays strike a surface at near-grazing angles.

In the lower row, we can see an image of the Sun Temple scene. The upper right rectangle shows further problems caused by foliage, this time for directional light source. The lower rectangle shows an issue caused by our

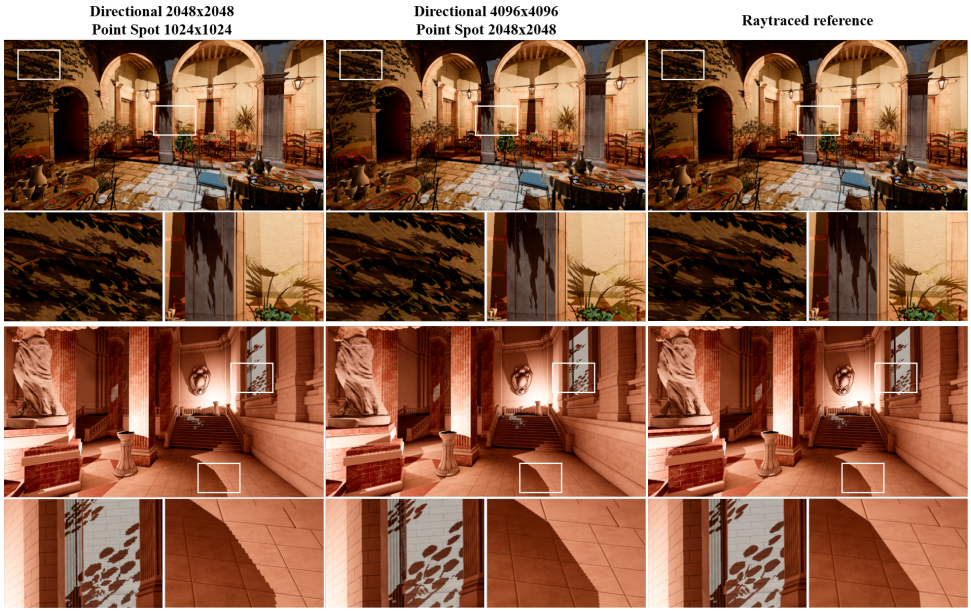


Figure 4.3 Upper row shows a view of the San Miguel Scene. Rectangle in the top left highlights issues caused by foliage which are worsened by the distance from the light source. The rectangle in the middle shows issues caused by near grazing angles of the incoming light. Bottom row shows a view of the Sun Temple scene. The Upper rectangle again shows issues caused by foliage, this time for directional light. The lower rectangle shows artifacts caused by our marking process which results in pages with lower resolution being marked.

marking logic. As shown in Figure 3.6, I use the normal of the underlying triangle to reconstruct the footprint of the pixel. However, this can result in the footprint being stretched when the surface is viewed at near-grazing angles. Because I only consider the bigger side of the footprint-bounding rectangle when selecting the mip level, pages with insufficient resolution are requested.

4.2 Performance Analysis

Next, I provide a performance analysis of my method. All measurements were taken by moving the camera along a predetermined path. This path was unique for each scene; however, for all measurements that involved one scene, the same camera path was always used. Moving the camera when collecting measurements is important for two reasons. First, the concentration of geometric complexity as well as individual lights varies across the scenen. This sometimes results in highly variable results in different locations of the scene. To make sure I provide correct results, the camera visits every section of each scene as a part of its path. The second reason why moving the camera is important is the shadow map caching described in 2.3.1. When caching is enabled, static cameras cache all pages. This results in no pages being redrawn, which skews the metrics.

4.2.1 Cached and Uncached Performance

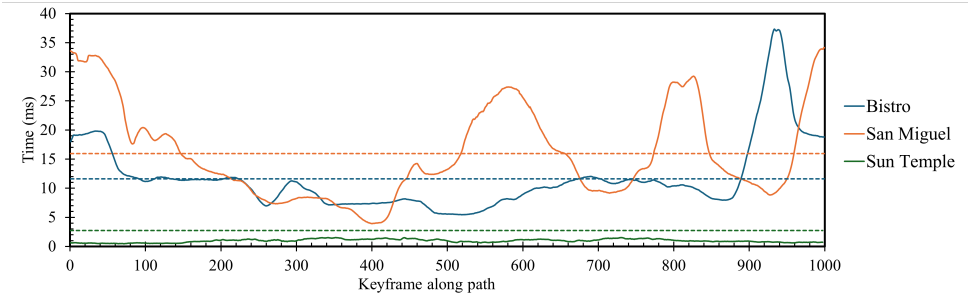
First, I provide graphs showing how performance varies along the predefined path in each scene, which can be seen in Figure 4.4. For these measurements, the resolution of 4096×4096 texels was used for directional lights and the resolution of 2048×2048 texels was used for both point lights and spotlights. It is important to note that these measurements do not include the sampling time of the shadow maps. These measurements consider only the time taken by the algorithm steps described in the implementation chapter.

From these measurements, we can immediately see the importance of caching. As soon as the geometric complexity of a scene rises above a certain level, caching increases the performance by factor of five to ten. This is because most of the geometric complexity is removed by the culling stages. When caching is disabled, the performance of each scene reflects that of its geometric complexity. Both San Miguel and Bistro scenes are five to in places forty times slower than the much simpler Sun Temple scene. In contrast, with caching enabled, the performance for all three scenes is in most parts comparable. This is despite Bistro having five and Sun Miguel ten times the triangles.

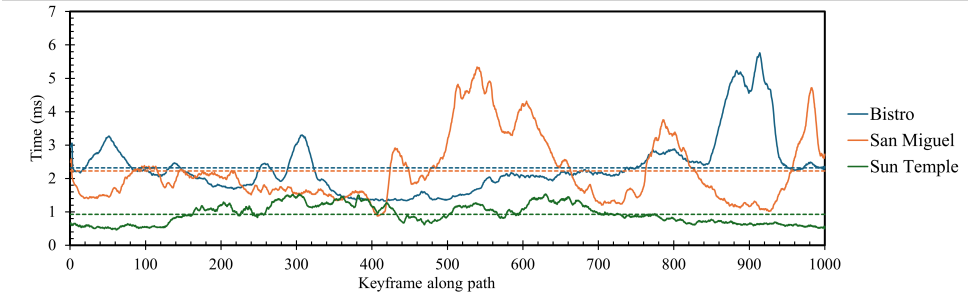
We can also see that, while caching improves performance, the trends of the individual sections of the path remain the same. Caching greatly reduces the number of pages that need to be redrawn, however, the pages still need to be drawn at least once. The peaks in the graphs of individual scenes correspond either to parts of increased geometric complexity or to parts along the path where the camera movement is fast. The increased geometric complexity is self-explanatory. With more overlapping geometry, comes more overdrawing, which increases the cost of drawing each page. Fast-moving camera, on the other hand, causes large dis-occlusions, which require many page draws each frame. Further, when the camera moves fast, more directional pages are invalidated due to toroidal addressing.

4.2.2 Performance of Algorithm Steps

Lastly, Figure 4.5 shows graph plotting the time taken by the most expensive stages of the algorithm. This measurement was taken on the Bistro scene with caching enabled. From this we can see that most of the spikes are caused by drawing point and spot lights. The first few smaller spikes are caused by the fast movement of the camera. As already discussed, this results in an increase in the number of point-light and spot-light pages drawn due to disocclusion and an increase in the number of directional pages drawn due to toroidal invalidation. The last spike is caused by the camera returning to the center Bistro area, where the path originally began. The center of Bistro contains the most lights. At the start of the path, the pages for all these lights were stored in cache. However, throughout the path, these got invalidated or freed to allow for rendering new pages along the cameras path. Upon reentry, the cache needs to be repopulated, which is what causes the large spike at the end of the path.



(a) **Uncached** performance for the three tested scenes.



(b) **Cached** performance for the three tested scenes.

Figure 4.4 The top graph shows the performance when caching is not enabled. The virtual resolution of directional lights was 4096×4096 texels and 2048×2048 texels was used for spotlights and point lights. While Sun Temple shows good results, the high geometric complexity of both San Miguel and Bistro scenes greatly reduce the performance. The performance with caching enabled can be seen in the bottom graph. We can see that caching successfully handles the high geometric complexity, resulting in comparable results for all three scenes.

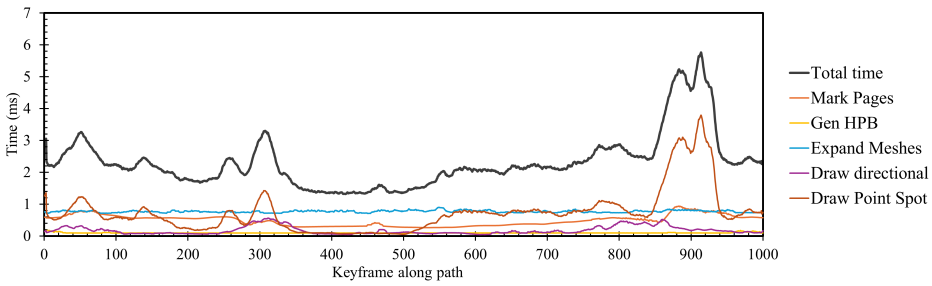


Figure 4.5 The timings for the most expensive stages through the algorithm along the path in the Bistro scene. The virtual resolution of directional lights was 4096×4096 texels and 2048×2048 texels was used for spotlights and point lights. We can see that the majority of the spikes are caused by the Point and Spot light drawing. While there are still some fluctuations in the directional light drawing, these are nowehere near as severe. The rest of the steps maintain steady performance throughout the entire camera path.

4.2.3 Scene Dependent Differences

Next, in Table 4.3 I show the times for individual stages of the algorithm. These were obtained by averaging all measurements made along the path in each scene. Measurements for two configurations are provided. These are the same as those used for the qualitative analysis. For reference, I also compare the time taken by the ray-tracing reference.

The ray-tracing was implemented utilizing inline ray queries. This is sub-optimal, as inline ray tracing cannot utilize many of the capabilities hardware has to improve the performance, such as shader execution reordering. The BVH used in my implementation was built by having one bottom acceleration structure per mesh. Then, all of them were grouped in one large top-level acceleration structure. No optimization or compaction of the acceleration structure was attempted. Both of these facts should be taken into account when interpreting the presented values. The ray-tracing timings are included for completeness, and are not meant to represent state-of-the-art performance. However, they still serve as a good baseline and can provide insight about how well VSMs compare to ray-tracing. The ray tracing uses the same light culling structure that was used during the marking process. As such, rays are only shot towards the lights that affect any given pixel, as opposed to shooting a single ray towards each light in the scene.

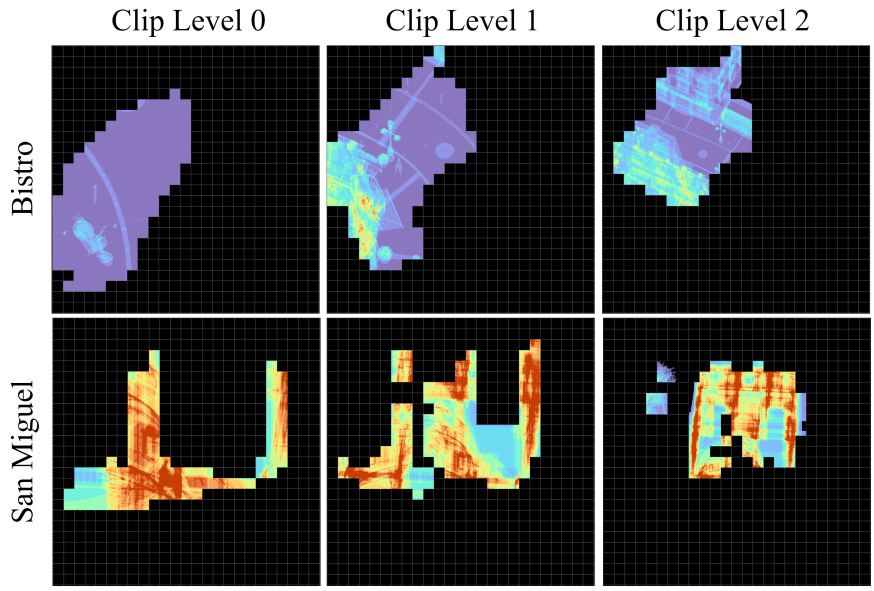


Figure 4.6 Overdraw from the perspective of the shadow map. In the upper row are three clip levels from the Bistro scene. Because Bistro is an outside scene. Not a lot of geometry is stacked on top of each other from the perspective of the light which results in little overdraw. In contrast, the three in the bottom show the same three clip levels in the Sun Temple scene. The roof and geometry of the temple cause significant overdraw, which increases the cost of the directional shadows.

Page Marking

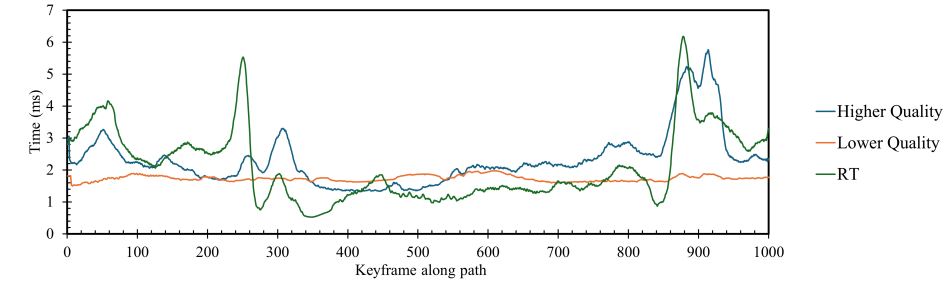
Unlike the previous graph, the timings presented in Table 4.3 also include the sampling cost. Like in the qualitative comparison, I only took a single shadow sample for each light. Looking at the timings presented, we can notice several differences between the individual scenes. First is the Mark Pages step. We can see that despite Bistro having the most lights out of the three scenes, the marking pass is the most expensive for the San Miguel Scene. San Miguel has multiple chandeliers that contain multiple candles. I modeled each candle as a separate point light source. This often results in many point lights being present in very close proximity to each other. As discussed previously, because of light culling, the cost of the marking pass is given by the number of lights that affect the same pixel. Both Bistro and Sun Temple have more uniform distribution of the lights in the scene. Because of this, the marking pass is faster even for Bistro, despite the larger number of present lights. Sampling the lights uses the same light culling, and as such the same arguments also apply.

Drawlist Expansion

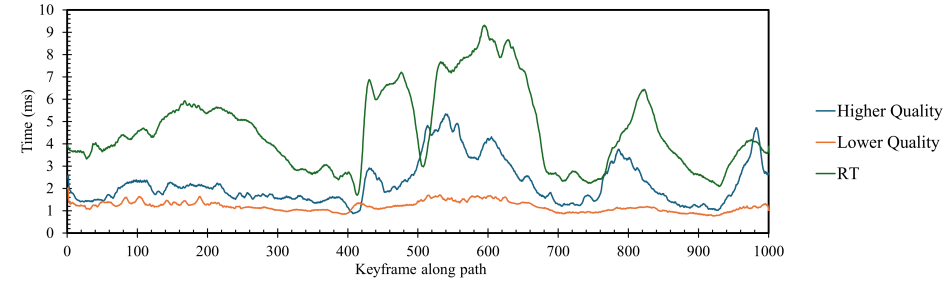
The next difference is in the drawlist expansion step. This time, the larger number of lights plays a direct role in the performance. Comparing Bistro and San Miguel we can see that Bistro has seven more point lights and twenty-six more spot lights. The drawlist expansion needs to be performed for 224 separate views. In contrast, the San Miguel drawlist expansion is performed only 156 times. Bistro also contains more individual meshes and is generally larger in the area it covers. Furthermore, the lights in the Bistro also have larger influences. This means that frustum and distance culling is not as efficient as in the San Miguel scene.

Directional Shadow Map Drawing

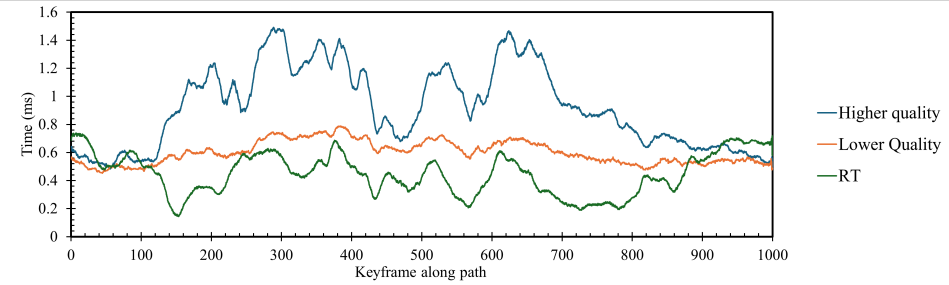
The last difference is in the drawing of directional shadow maps. This step is the most expensive for the Sun Temple in both configurations. Sun Temple is an entirely closed of scene. Because of this, when drawing the directional shadows, there is a lot of overdraw caused by the roof of the temple. This causes the directional shadows to be much slower. This is visualized in Figure 4.6. It compares overdraw in three clip map levels on Bistro and Sun Temple. Figure 4.8 then shows the same view from the perspective of the main camera with the overdraw projected into the scene.



(a) Bistro.



(b) San Miguel.



(c) Sun Temple.

Figure 4.7 The comparison of cached performance for two settings and ray tracing reference for individual scenes. Higher Quality refers to configuration with virtual directional resolution of 4096×4096 texels and virtual point and spot resolution of 2048×2048 texels. Lower Quality refers to configuration with virtual directional resolution of 2048×2048 texels and virtual point and spot resolution of 1024×1024 texels. RT refers to the performance obtained by the ray tracing reference.

4.2.4 Differences Across Configurations

Finally, I would like to discuss the differences in performance between the two configurations and the ray traced reference. We can see that most stages are completely independent of the VSM resolution. The marking pass is slightly cheaper, which is caused by the reduced resolution of the VPT. As we would expect, drawing of all directional, point and spot light pages is cheaper. This comes from the reduced number of shadow map texels that need to be rasterized.

In Figure 4.7 I also provide the comparison of all three methods in each

Samples taken	**1**	**2**	**4**	**8**
VSM sampling	295.7 μs	562.7 μs	618.9 μs	927.9 μs
VSM total	2615.1 μs	2882.3 μs	2938.6 μs	3247.6 μs
Ray tracing	2083.8 μs	4461.1 μs	9358.4 μs	19308.5 μs

Table 4.2 Comparison between ray tracing and VSMs when considering multiple shadow map samples. We can see that ray tracing is faster only for a single shadow map sample. As soon as the samples increase, the cost of ray tracing increases linearly with it. This is not true for VSMs. Due to the spatial locality of individual samples, the cache ensures faster following lookups, resulting in much better performance.

scene separately. From this we can see that the lower virtual resolution greatly reduces the spikes present for higher resolutions. Due to the reduced resolution, each shadow map is represented by fewer pages. As such, the PPT can store most of the pages, without having to throw any rendered pages out.

We can also see that while the average performance of the ray tracing is better, the spikes produced by this are much worse. This is caused by VSMs still being able to at least partially time slice the number of necessary computations. Even when a lot of pages need to be allocated in a certain section of the scene, these are still, at least partially, distributed through multiple frames.

4.2.5 Performance with Multiple Samples

The results shown in Table 4.3 might suggest that ray tracing outperforms and provides better detailed results than VSMs. Indeed, VSMs managed to outperform ray-tracing only in the San Miguel scene. However, things change when multiple samples are considered. This can be seen in Table 4.2. Due to the pages being loaded into the cache, taking repeated samples that are spatially close is significantly cheaper than the original sample. The same is not true for ray tracing. Every ray needs to be traced anew, resulting in a linear increase in cost. Another thing to consider is the typical presence of other effects that require shadowing information. For example, for volumetric effects, the same shadow maps could be reused. This would not be possible for ray tracing, which would further increase the cost.

Finally, I tested the influence of the PPT size on the performance achieved by my method. The result of this test can be seen in Figure 4.9. The measurements in this figure show the performance of the drawing step of the method. The test was run in the Bistro scene with the virtual resolution of the directional light sources set to 4096×4096 and the virtual resolution of the point and spot light sources set to 1024×1024. The resolution of the point and spot light sources needed to be reduced so that the method would not run out of memory.

From the figure, we can see a clear reduction in performance when the PPT size is decreased. This is correlated with a cache hit rate statistic. The cache hit rate was determined by dividing the number of pages marked and

previously cached by the total number of pages marked in the given frame. We can clearly see that in the parts of the graph where I measured spikes in the performance, the cache hit rate is decreased because more pages need to be drawn. The measurement with a larger PPT shows a much higher hit rate at the end of the camera path. As already discussed above, in the last section of the path, the camera reenters the center of the Bistro scene which contains the most light sources. As such, more pages remained cached from the start of the path, and thus the cache hit rate is much better.

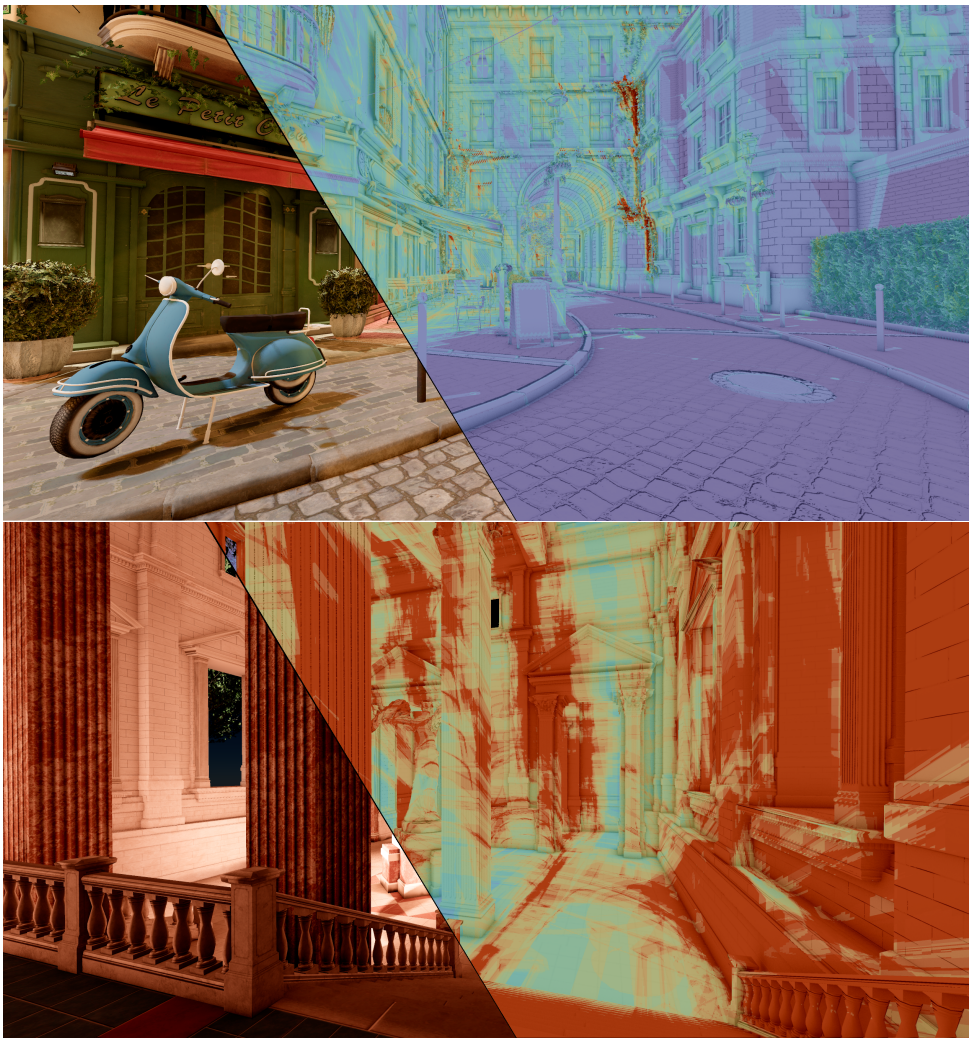


Figure 4.8 Shadow map overdraw Visualization on Bistro and Sun Temple. The overdraw visualization is obtained by projected data seen in Figure 4.6 onto the scene. Because Sun Temple is a purely indoor scene, shadow maps for directional lights encounter much more overdraw when compared to outside scenes like Bistro.

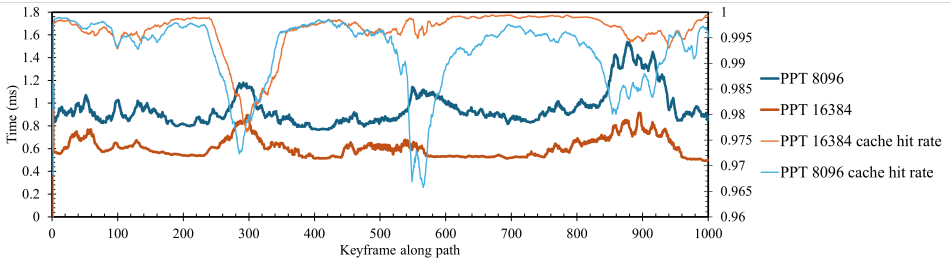


Figure 4.9 The correlation between the performance of the drawing step and cache hit rate measured on the Bistro scene. The virtual resolution of the directional light sources was set to 4096×4096 and the virtual resolution of the point and spot light sources was set to 1024×1024. Cache hit rate denotes the number of pages that were marked but not drawn compared to the total number of marked pages. We can clearly see, that the increase in PPT size allows for more pages being cached, which results in better cache hit rate, improving the performance.

4.2.6 PPT Size Analysis

Improved caching affects both the directional and point lights, as they share the PPT. That said, point lights are affected more because they benefit more from caching. For directional lights, pages still need to be invalidated and redrawn when the main camera moves. However, the above is not true for point lights. When the cache is large enough, all pages can be stored in the PPT and never redrawn. The performance of uncached shadow maps is independent of the PPT resolution as all pages are redrawn each frame.



	Bistro	**San Miguel**	**Sun Temple**
Directional 4096 × 4096 Spot and Point 2048 × 2048			
Invalidate Pages	6.3 μs	5.7 μs	5.0 μs
Mark Pages	489.4 μs	739.7 μs	236.4 μs
Classify Pages	10.4 μs	10.5 μs	12.2 μs
Allocate Pages	4.9 μs	4.9 μs	4.6 μs
Clear Pages	8.1 μs	6.0 μs	4.2 μs
HPB Directional	5.8 μs	5.6 μs	6.3 μs
HPB Point Spot	96.1 μs	69.8 μs	53.2 μs
Expand Directional	14.4 μs	11.0 μs	12.5 μs
Expand Point Spot	759.0 μs	307.2 μs	149.7 μs
Draw Directional	180.0 μs	303.0 μs	350.3 μs
Draw Point Spot	738.7 μs	755.8 μs	90.7 μs
Sampling	295.7 μs	592.6 μs	221.3 μs
Total	**2608.8 μs**	**2811.7 μs**	**1146.5 μs**
Directional 2048 × 2048 Spot and Point 1024 × 1024			
Invalidate Pages	5.8 μs	4.6 μs	4.7 μs
Mark Pages	442.2 μs	694.9 μs	247.2 μs
classify Pages	10.2 μs	10.3 μs	12.6 μs
Allocate Pages	4.1 μs	3.9 μs	4.8 μs
Clear Pages	4.0 μs	3.2 μs	3.7 μs
HPB Directional	5.2 μs	5.0 μs	6.9 μs
HPB Point Spot	101.2 μs	69.5 μs	52.5 μs
Expand Directional	14.4 μs	10.2 μs	12.8 μs
Expand Point Spot	779.1 μs	287.1 μs	164.5 μs
Draw Directional	46.9 μs	43.1 μs	69.3 μs
Draw Point Spot	75.8 μs	60.5 μs	20.9 μs
Sampling	323.8 μs	565.1 μs	284.4 μs
Total	**1812.5 μs**	**1757.2 μs**	**884.3 μs**
Raytracing			
Render Time	**2083.9 μs**	**4623.5 μs**	**446.3 μs**

Table 4.3 Times of individual stages of the algorithm for two test configurations which varied virtual resolutions. The used resolutions are denoted below each subtable. The times were calculated by averaging values measured along the entire pre-programmed path. For comparison, times to ray trace the scenes in the bottom.

Chapter 5

Conclusion and Future Work

In this work, I have presented an efficient method for rendering shadow maps called Virtual Shadow Maps. By separating the shadow map from the physical storage, I achieved a significant reduction in consumed memory and improved performance. I described the complexities of implementing the individual steps of the algorithm to allow for the fastest and best quality results. I showed that our method scales well for many light sources and generalizes to various light types, allowing for a unified approach. This is an improvement over existing methods that solve only a specific part of the problem. I have described how our method utilizes a specialized scene representation scheme to allow for highly efficient culling. Finally, I have shown the results obtained by my implementation. I have analyzed the quality of the resulting shadow maps compared to a reference obtained by ray tracing. In addition, I have also provided detailed performance metrics for multiple configurations and various scenes.

To further improve the performance of the Marking Pass, the hardware capabilities of modern GPUs could be utilized to perform the marking step for us. Modern graphics APIs expose functionality called Sampler Feedback, which allows for capturing and recording texture sampling information and location. This could be used to project the pixel into the shadow map. As projection is the most costly operation of the marking pass, this could significantly increase the performance.

Next, sparse (or tiled) resources in graphic APIs are a promising choice for VSMs. Currently, the API for sparse resources does not allow for an indirect GPU-driven approach. Backing individual pages would require GPU read-back; however, I believe these issues can be negated by a clever approach.

Finally, combining VSMs with Ray Tracing approaches could promise a further performance increase for scenes with enough complexity. Instead of rasterizing each page, the GPU hardware could be used to trace a single ray for each texel of every page. The VSM would then serve as a sort of cache for individual rays. This could especially be useful for directional light sources in scenes with large amounts of overdraw.

Bibliography

[AMD25] AMD. *Fidelity FX SDK*. https://gpuopen.com/amd-fidelityfx-sdk/. 2025. URL: https://gpuopen.com/amd-fidelityfx-sdk/.

[AH05] A. Asirvatham and H. Hoppe. "Terrain rendering using GPU-based geometry clipmaps". In: *GPU gems* 2.2 (2005), pp. 27–46.

[Bar08] S. Barrett. *Sparse Virtual Texture Memory*. https://www.gdcvault.com/play/417/Sparse-Virtual-Texture. GDC San Francisco CA, Feb. 19, 2008. URL: https://www.gdcvault.com/play/417/Sparse-Virtual-Texture (visited on 04/15/2024).

[Dim07] R. Dimitrov. "Cascaded shadow maps". In: *Developer Documentation, NVIDIA Corp* (2007).

[FFB01] R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg. "Adaptive shadow maps". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, pp. 387–390.

[GW07] M. Giegl and M. Wimmer. "Queried virtual shadow maps". In: *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. 2007, pp. 65–72.

[GKM93] N. Greene, M. Kass, and G. Miller. "Hierarchical Z-buffer visibility". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993, pp. 231–238.

[Kap25] A. Kapoulkine. *MeshOptimizer*. https://github.com/zeux/meshoptimizer. 2025. URL: https://github.com/zeux/meshoptimizer.

[LSL11] A. Lauritzen, M. Salvi, and A. Lefohn. "Sample distribution shadow maps". In: *Symposium on Interactive 3D Graphics and Games*. 2011, pp. 97–102.

[LSO07] A. E. Lefohn, S. Sengupta, and J. D. Owens. "Resolution-matched shadow maps". In: *ACM Transactions on Graphics (TOG)* 26.4 (2007), 20–es.

[MM14] M. McGuire and M. Mara. "Efficient GPU screen-space ray tracing". In: *Journal of Computer Graphics Techniques (JCGT)* 3.4 (2014), pp. 73–85.

[NVI25] NVIDIA. *Slang Shading language*. https://github.com/shader-slang/slang. 2025. URL: https://github.com/shader-slang/slang.

[OSK14] O. Olsson, E. Sintorn, V. Kämpe, M. Billeter, and U. Assarsson. "Efficient virtual shadow maps for many lights". In: *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 2014, pp. 87–96.

[SA25] M. Sakmary and P. Ahrens. *Timberdoodle*. https://github.com/Sunset-Flock/Timberdoodle. 2025. URL: https://github.com/Sunset-Flock/Timberdoodle.

[SRH24] M. Sakmary, J. Ryan, J. Hall, and A. Lustri. "Virtual Shadow Maps". In: *GPU Zen 3*. Black Cat Publising, 2024, pp. 319–336.

[TMJ98] C. C. Tanner, C. J. Migdal, and M. T. Jones. "The clipmap: a virtual mipmap". In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 1998, pp. 151–158.

[Wav09] J. van Waveren. *id Tech 5 Challenges*. Conference Presentation. 2009. URL: https://web.archive.org/web/20091007031619/http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf (visited on 05/01/2024).

[WH10] J. van Waveren and E. Hart. "Using Virtual Texturing to Handle Massive Texture Data". In: *GPU Technology Conference*. Vol. 10. 2010.

Appendix A

Attachment List

In this Appendix, I describe the structure of the attached application source code. Attached is the code for the entire framework. The entirety of the framework was not developed by me. However, the parts relevant to VSMs were uniquely written by me. As such, I will only describe the parts of the structure that are relevant to this work.

- **build** - This folder will be generated by the build process and will contain the resulting binaries.
- **deps** - This folder contains all the external libraries utilized by the implementation.
- **cmake** - Contains files relevant to the CMake build system.
- **media** - Contains various snapshots taken from the implementation.
- **settings** - Used to store the default settings used by the implementation.
  - **camera** - Stores keyframes of the paths used when obtaining the results.
  - **sky** - Stores the default configuration of the procedural sky used by the framework.
- **src** - contains the entire source code of the framework.
  - **multithreading** - source files relevant to the thread-pool used to load the scene.
  - **rendering** - source files related to rendering.
    - **virtual_shadow_maps** - Contains source files of shaders for individual steps, as well as all CPU side setup required in order to run the algorithm.
  - **scene** - source files handling the loading of scene and processing of geometry.
  - **shader_lib** - standalone contextless files used by shader code.
    - **vsm_sampling.hlsl** - Source file containing code related to sampling of VSMs.

- **vsm_util.hlsl** - source file containing various VSM helper functions.

- **shader_shared** - code that is shared between the CPU and GPU Code.
  - **vsm_shared.hlsl** - file that is shared between CPU and GPU which provides the configuration for VSMs and the declaration of shared structures.
- **ui** - souce files relating to the ui of the framework.

Appendix B

Manual

The framework that I used uses Cmake to build the entire application. All dependencies are managed by vcpkg. When vcpkg is not present in the system, it will automatically be downloaded into the project. The application was developed using vscode with the cmake extension, which is the recommended way of building this project. When using this step, the provided cmake configuration should automatically be detected.

When building the application manually through the command line, the standard CMake process should also work:

1. Go to the root directory.
2. From the command line call **cmake –preset=<specify preset here>** to configure and generate cmake files.
3. To list all available presets, call **cmake –list-presets**.
4. From the command line, call **cmake –build –preset=<specify build preset here>** to build the application.
5. To list all available build presets, call **cmake –build –list-presets**.

The MSVC compiler with version 19.42.34433.0 and Cmake with version 3.29 were used to build the application.

B.0.1 Controls

The application operates in two modes. The first is the control mode. This is the mode in which the application is initially in. It is used to control the user interface of the application.

The second mode is the interactive mode which locks the camera and allows the user to fly through the scene. This mode is entered and exited by pressing the ESC key. The control scheme in this mode is slightly different:

- WASD move forward, left, back and right, respectively.
- SPACE increase altitude.
- ALT decrease altitude.

- SHIFT to increase the flying speed

In addition to this, the cursor is disabled in the fly-through mode, and the user can control the view direction with the usage of mouse.

A new scene can be opened by navigating through the **File** menu in the top left corner. Similarly, individual widgets controlling the behavior of the entire application can be opened from the Widgets section.