CHALMERS

Irregular Adaptive Shadow Maps

Master of Science Thesis

FLAVIUS ALECU

Chalmers University of Technology
University of Gothenburgh
Department of Computer Science and Engineering
Göteborg, Sweden, May 2014

Irregular Adaptive Shadow Maps

Flavius Alecu

Examiner: Ulf Assarsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2014

Abstract

Shadow mapping is a common approach to visualizing shadows in real-time graphics applications such as games. Given the performant nature of these applications, often limited resources are reserved for this task despite shadows providing a strong and important visual cue in computer generated graphics.

With this work we introduce a new shadow mapping algorithm that aims for optimal resource usage while being comparable in quality to other popular approaches. A tile based partitioning scheme is provided to facilitate dynamic customizability and allow for adaptive distribution of resources at run time which leads to more efficient use of memory resources. Detailed results for memory and execution costs are presented and comparisons to commonly used previous work are made.

Acknowledgments

I would like to start by thanking Dean Calver for being an endless source of information and amazingly nice person in general. Thanks goes also to Luis Alvarado for all the fruitful discussions that sparked many an idea, Yohanna Adolfsson for taking the time to make sure this work is presented in an acceptable manner, and Manuel Virks for providing the beautiful tree model. I would also like to thank my supervisor, Ulf Assarsson, for all the help and for having the patience that allowed me to finish this insightful, if a bit long, journey.

Contents

List of Tables

List of Figures

1

Introduction

1.1 Background

Shadows are a fundamental visual cue detailing both object placement and movement in a scene. They make a significant contribution towards immersion in video games and computer animated films utilizing three-dimensional graphics. While films have reached a very high standard in visual fidelity, video games, being an interactive medium, require a more stringent approach. A lot of effort and resources are invested in many different aspects of the technology required to drive a modern high budget game. Player input needs to be handled, the game world needs to be simulated, audio needs to be processed and finally the world representation needs to be rendered. All of this should happen fast enough in order to have time to repeat the process several times per second and present the player with a continuous and smooth picture. In contrast, when creating computer generated graphics for film significantly more resources are available. Several orders of magnitude more time is available for each frame to render with the additional benefit of a controlled computer environment often offering an impressive level of capabilities. Nevertheless, the visual standard of real-time rendering quickly

approaches photorealism, yet shadows still remain a subject with much room for new research.

Shadows in computer and video games are usually calculated by having an understanding of the scene from the shadow casting light's point of view. The depth of the scene from this view is recorded and used to decide what lies in shadow and what does not. When rendering a frame of the game world to show on screen, each pixel queries the light view information and uses it to decide if it is in shadow or not. The camera view used to display the game world does not necessarily see the same part of the world as the shadow casting light. As such, each pixel in the camera frame does not always map to a pixel in the light view frame. Given the discrete nature of both the light and camera views, the resulting shadows will inevitably be based on incomplete information.

In practice, only a few resources are allocated to shadows in a game so that all other parts of the engine have room in the memory and processing time available. This dissertation explores a new algorithm for rendering real-time shadows and, like the majority of previous research in this field, we focus on increasing visual fidelity with a priority on real-time performance. Attempts at modifying established shadow rendering algorithms are made and new ideas are introduced. While some goals are achieved, certain weaknesses in our algorithm are identified that suggest a generic solution is not necessarily the optimal choice.

The requirements to render shadows vary between different circumstances. For example, a directional light representing the sun affects a larger amount of objects and casts a different type of shadow compared to an omnidirectional light, such as a desk lamp inside a small room. An implementation can cover all scenarios although the sacrifice of not being able to apply specific optimizations is then made. Thus this research focuses on a subset of shadow types, namely, shadows as cast by directional lights. We do not consider filtering, which is an algorithm agnostic detail and our implementation does not interfere with its application.

1.2 Outline

Shadows, being such an important part of computer graphics, is a major research subject. Chapter 2 gives a brief overview of previous research related to real-time shadow algorithms. We focus primarily on algorithms related to this thesis and what is utilized in modern video games and real-time rendering. In Chapter 3, the Irregular Adaptive Shadow Maps (IASM) algorithm - the primary focus of this dissertation - is described with comprehensive implementation details. Chapter 4 presents the results of the research associated with this thesis and gives an indication of how the new algorithm compares to commonly used previous work, more specifically Cascaded Shadow Maps (CSM). In Chapter 5 we discuss the dissertation and the various decisions that were made during implementation, as well as touch upon improvements that can be made and ideas to extend the work. Finally, Chapter 6 concludes the thesis with a concise summary.

2

Related Work

This chapter will give an overview of previous research done related to shadows in computer graphics. The focus will be shadow mapping algorithms, and more specifically, non-filtered partition based techniques as they apply to directional lights. However, for adequate coverage, additional work is mentioned that did not strictly influence the implementation of this dissertation.

2.1 Shadow Maps

The z-buffer is a two-dimensional array of values representing the depth of a scene as seen from an arbitrary point of view [6]. Such a buffer, populated with the distance to the scene from the point of view of a light source, stores depth information for any geometry "seen" by that light, as accurately as allowed by the resolution of the buffer. The original shadow mapping implementation [34] uses a z-buffer, when rendering the final image to display on screen, to decide what parts of the image should be shadowed. Each pixel is transformed to the view point of the light source and its depth is tested

against the values already present in the light depth buffer. If the pixel's depth is larger than the current value at the location it is not visible to the light and thus needs to be shadowed.

As noted in this initial shadow mapping proposal, this algorithm is not without problems. Due to the discrete nature of the depth buffer and the perspective of the camera view, transforming pixels from camera space to light space introduces aliasing and quantization artifacts. The limited resolution of the z-buffer can also introduce a type of artifact known as self-shadowing. In this scenario, a slanted plane, in respect to the shadow casting light, will inaccurately receive a shadow when no object is blocking the light from reaching the plane. This can partially be mitigated by applying a bias value [27] when sampling the depth buffer. This bias, however, is object specific and difficult to select for a globally optimal result.

The Midpoint Shadow Maps [36] implementation attempts to solve some of these problems by recording not just the closest distance to the light but also the second closest one. The average of the two is then used as the depth in the light z-buffer. This moves the depth comparison from the surface of an object into the middle of the geometry, hiding the artifacts caused by numerical imprecisions. For this to work, back-face culling cannot be applied, which is often a desired optimization where polygons facing away from the camera are not rendered and thus the execution cost is lowered. If back-face culling were to be applied, the second distance could potentially come from a far away surface. This would give a false representation of the average depth of shadow casting objects, leaving certain shadowed areas to pass the depth test and be illuminated. Dual Depth Shadows [33] addresses this by clamping the bias value to a predefined constant when the distance between the two registered depth values is too large. Conversely, Second-Depth Shadow Mapping [32] proposes that using the second closest distance is enough to not require a bias value at all to get rid of the self-shadowing artifacts.

The depth resolution when rendering a shadow buffer maps onto the space defined by the light frustum along its view direction. Noting this, a simple and algorithm agnostic improvement, known as fitting, can be implemented by optimizing the frustum to only include the space containing any shadow casting geometry that could potentially cast a shadow intersecting the view frustum [5].

2.2 Warped Shadow Maps

Directional light shadows are rendered with an orthogonal projection to correctly represent light sources far away. The lack of perspective produces a consistent sampling density across the entire shadow map. Since the final render of a scene is usually rendered with a perspective projection, to give an impression of depth and accurate object size and placement, the sampling density in the shadow map would ideally be greater closer to the viewer. This would minimize artifacts and unnecessary waste of memory for distant objects. An attempt to reduce perspective aliasing, an artifact caused by inadequate shadow buffer resolution, is made by Perspective Shadow Maps [30]. The idea is to map the view frustum to the unit cube, creating a perspective distortion where closer objects appear larger than farther ones. The scene in this state is rendered to the shadow map, resulting in a higher density close to the camera. With this algorithm, however, the shadow quality varies greatly with the light source position and direction since the light direction vector will be modified if the view direction is not perpendicular to it. The reparameterization of the shadow map also causes distant objects to have a reduced presence in the shadow map, lowering the quality of the shadow for these while self shadowing artifacts become more difficult to avoid as the bias value applied will need to vary based on the sample location in the shadow map.

Most issues introduced by Perspective Shadow Maps are caused by the shadow map warping being done in view space. This observation is made and the suggestion to apply the warp in light space instead is presented by Light Space Perspective Shadow Maps [35]. A frustum with the forward vector parallel to the shadow map plane is found and used to distort the scene before rendering to the shadow map. This affects the shadow map plane but leaves the light direction vector intact. Furthermore, the distance from the near plane to the perspective reference point can be freely chosen, which gives control over the level of distortion. This could be used to avoid drastic quality loss for objects far away.

The Perspective Shadow Maps approach can be extended with a low resolution pre-pass rendering of the scene where information is gathered and used to choose optimal perspective distortion parameters [7]. It is also proposed to split the shadow map rendering into several smaller light frusta in order to take advantage of local similarities in the shadow map.

A shadow map improvement similar to above mentioned perspective distortions is given by Trapezoidal Shadow Maps [24]. The camera frustum is approximated by a tightly bound trapezoid from the light's point of view which, when mapped to the shadow buffer, improves resolution for objects near the camera. In addition, the carefully selected trapezoid provides a closer fit around the area that falls inside the camera frustum, thus the resulting shadow map better utilizes each texel.

2.3 Partitioned Shadow Maps

Warping algorithms are primarily used to provide higher sampling density closer to the viewer while sacrificing, possibly unused, resolution farther away. A similar, although cruder, effect can be achieved by splitting the shadow processing along the view direction and allocating different resolutions to each split, as proposed by Cascaded Shadow Maps [11], Parallel

Split Shadow Maps [38] and Z-partitioning [22]. Warping alters the shadow map most when the incident light vector is perpendicular to the view vector and converges to the original shadow mapping algorithm when the two vectors are parallel. In contrast, partitioning does not depend on the position or direction of the light.

The algorithm splits the view frustum into an arbitrary number of cascades and a shadow map is rendered for each, causing some objects in the scene to potentially be rendered multiple times. The implementation of Parallel-Split Shadow Maps [37] shows how this can be avoided by utilizing the geometry shader stage of newer hardware.

Partitioning the shadow map presents the possibility to render different sized shadow maps for different parts of the scene, as we have used to great extent in our algorithm, further explained in Chapter 3. Culling algorithms need to be used in order to only render geometry needed for each cascade, although objects on split boundaries stand in need of being rendered in two, possibly several, shadow maps, depending on the partitioning scheme.

Separating-Plane Perspective Shadow Mapping [25] proposes a method where a plane can be found to split the light frustum such that one side uses a focused shadow map and the other a perspective distorted algorithm to optimally increase sampling density in each area under given circumstances. While this method does not extend any algorithms, it helps pick the best performant one at a minuscule performance cost.

Another example of the increased quality achieved from focused shadow maps is demonstrated by Sample Distribution Shadow Maps [19]. Here, regular z-partitioning is used with the addition of programmatically determining the tightest light frustum needed for each cascade. This can greatly increase the depth range available for the shadow map while at the same time removing the requirement for manual tweaking of constant parameters.

A slightly different partitioning algorithm, which has successfully been used in games on both personal computers as well as consoles, is demonstrated by Facetted Shadow Mapping [31]. The shadow map is divided into

facets, each with their own custom distortion matrix. When an object is rendered into the shadow map, the correct facet needs to be selected and its matrix used to transform the object before rendering. The same process is needed during shading when the shadow map is sampled. Using facets to partition the shadow map allows for circular distortion on the shadow map, providing higher precision in the center. This suits some scenarios better than others; for example, when the point of interest is not the camera.

2.4 Adaptive Shadow Maps

A scene very rarely has a constant distribution of geometry complexity across the entire frame. Moreover, in most scenarios, as the point of view moves around a virtual world, any one area in screen space experiences temporal differences in geometry. A static partitioning of a shadow map does not help in this case. The next logical step is to dynamically change the partitions and update them to better represent the current scene. This is exactly what is accomplished with Adaptive Shadow Maps (ASM) [12]. Instead of storing a single, two-dimensional shadow buffer, the information is stored in a quad tree where each node contains a part of the shadow map. This layout allows the shadow map to have a non-uniform resolution with the ability to update individual nodes as required by the scene. An edge detection algorithm is used to find shadow boundaries and refine the resolution needed for that part of the shadow buffer until the optimal resolution has been reached or the dedicated shadow memory has been depleted.

The iterative edge detection algorithm used by ASM has been found to be expensive and not always lead to accurate shadow resolutions. Resolution-Matched Shadow Maps [21] improves upon this research by observing that in most plausible scenes geometry continuity in view space leads to continuity in light view space. The edge detection step is removed and higher resolution is requested for all shadow texels that map to a pixel in the view image.

A very similar approach to Adaptive Shadow Maps was introduced by Queried Virtual Shadow Maps (QVSM) [14]. A quad-tree is again used to split the shadow buffer and refine the shadow resolution non-uniformly with the major difference being the heuristics used to determine when the optimal resolution for a node has been reached. While the ASM implementation relies on edge detection, QVSM uses the occlusion query mechanism in the Graphics Processing Unit (GPU) to count how many fragments have been output by the pixel shader for a single shadow map node. When the difference in fragment count between one refinement step and the next is below a set threshold the target resolution is deemed to have been reached.

Fitted Virtual Shadow Maps [13] explores a different method for splitting the available shadow buffer resolution by relying on deferred shading as the rendering setup. During a pre-pass render of the scene, the shadow space location of each view space pixel, as well as the required resolution at that location in the shadow buffer, is stored for later use. This data is then processed on the Central Processing Unit (CPU) and a kd-tree is created where each leaf node describes a part of the shadow buffer used to shadow the scene.

Tiled Shadow Maps [4] divides the light view into constant sized, rectangular tiles using a recursive binary cut algorithm. The input to this subdivision algorithm is a weight grid extracted during a separate low resolution rendering pass of the scene from the light's point of view. The weight values are calculated by observing pixels with depth discontinuity and using the distance between the shadow caster and the receiver along with the distance to the viewer. The observation that the larger the distance between shadow caster and receiver is the higher the shadow resolution required to minimize artifacts is made.

2.5 Sampling

The Shadow Silhouette Maps [29] algorithm builds upon the observation that shadow maps are only problematic at shadow boundaries. The work extends the traditional shadow map with an additional silhouette map where edges on geometry silhouettes are rasterized and stored for reconstruction during shading. When sampling the shadow map, the silhouette map provides a more accurate representation of the edges to improve the depth test performed on pixels close to shadow boundaries.

A slightly different approach to solving artifacts caused by lacking resolution is proposed by Alias-Free Shadow Maps [1]. The algorithm utilizes an unorthodox shadow buffer where the visible pixels in view space are transformed to light space and stored explicitly. The scene is then rendered from the light point of view and the previously transformed pixels are tested for exact visibility results. The shadow buffer stores information for only (yet all) view space pixels which leads to pixel perfect shadows. This algorithm can be taken one step further by treating each pixel as a projected facet from the tangent plane of the visible point [26]. Occluders are projected onto the facet and used to determine sub-pixel occlusion allowing for anti aliasing of shadow boundaries.

Yet another unconventional shadow map approach is presented in Reconstructable Geometry Shadow Maps [9]. Instead of the rasterized depth, the vertex information for the whole occluding triangle is store in the shadow map. This allows for reconstruction of the triangle and the occluding point when sampling the shadow map during shading. A potential problem is caused by GPUs not generating fragments for triangles that do not cover the center of the pixel, leading to incomplete information along shadow edges. Sub-Pixel Shadow Maps (SPSM) [20] extends this work with the addition of Conservative Rasterization [15], which guarantees the full area of a triangle has a presence in the buffer after rasterization. Since only the

triangle closest to the light is stored in each texel false-negatives can occur during shadow tests because of missing triangles in the buffer. To solve this SPSM recreates the eight additional triangles stored in the sampled texel's 3x3 neighborhood for more accurate results.

Further research explores adaptive sampling rates with Rectilinear Texture Warping (RTW) [28]. The algorithm starts with a scene analysis pass where a two-dimensional importance map is created. This step is done by rendering an initial geometry pass from the light point of view, the camera view with all shading information except shadows, or by combining the two methods. The importance map is then transformed into two one-dimensional warping maps, defining a warping grid used to shift texels when rendering into as well as sampling the shadow map. The RTW shadow map algorithm uses shadow buffer space more optimally than other similarly performant approaches. Unfortunately, the warping can also bend triangle edges causing artifacts on shadow boundaries unless the geometry is highly tessellated.

2.6 Filtering

Shadow mapping algorithms utilizing filtering approach the aliasing problem from a different direction. Instead of maximizing the effective resolution of the shadow map, the binary end result (lit or in shadow) for each pixel is filtered over an area. As such, some of these methods are often orthogonal to previously mentioned research, including the implementation of this thesis.

The ubiquitous Percentage Closer Filtering [27] samples the shadow map several times and stores the binary results in a table which is then averaged to get the final filtered value. This value is used to decide how strongly shadowed the pixel is, effectively creating a non physically based penumbra.

Deep Shadow Maps [23] modifies the shadow buffer format to store a visibility function for each pixel. These functions give the level of light at an arbitrary depth along the primary ray for each pixel. Percentage Closer

Filtering can be applied when computing the functions, which removes some of the work required when sampling the shadow map. However, between the cost of this preprocessing and the enlarged shadow buffer requirements, this algorithm is not suitable for real-time applications.

Expanding upon Deep Shadow Maps, Opacity Shadow Maps [17] utilizes the GPU by rendering the scene several times and clipping the geometry at different positions along the light direction vector. The resulting opacity maps provide several snapshots of the geometry along the depth of the scene and are used to compute the per pixel visibility functions.

Variance Shadow Maps [10] presents yet another different representation of the shadowing data required to shade a scene. Each texel of the shadow map stores the mean and mean squared of a distribution of depths. These values are used to compute the mean and variance of that distribution which in turn provide a bound on the percentage of the distribution that is greater than a given depth value. The mean and mean squared values can correctly be averaged, thus can the shadow map benefit from both software and hardware filtering.

A disadvantage of Variance Shadow Maps is highlighted when a distribution variance is large, which can be caused by a group of objects overlapping each other with large depth discontinuities. The visibility approximation in this situation is not accurate enough and light leaks can occur, illuminating areas that should be in shadow. Layered Variance Shadow Maps [18] provides a simple, although slightly more memory costly, solution. The depth range is split into several layers each with their own warping function optimal for a specific depth value. At sampling time the correct layer is chosen to give the best possible result.

Convolution Shadow Maps [2] provides a way to filter data required to shade a scene, similar to Percentage Closer Filtering with the additional benefit of supporting pre-filtering and hardware mip-mapping. This is achieved by replacing the shadow test function with a Fourier series expansion. A sum

of weighted basis functions is applied to the shadow map data and filtering the result of the basis function will directly filter the resulting shadow. In practice a relatively high memory cost is required for adequate results.

Exponential Shadow Maps [3] improves upon Convolution Shadow Maps by removing the costly Fourier expansion and instead storing e^{cz} in the shadow map, where z is the regular shadow space depth and c is an empirically chosen constant. Similarly to the basis functions used in Convolution Shadow Maps this value can be pre-filtered and is much cheaper to store. If the shadow map is filtered however, the chance to introduce artifacts grows with the size of the convolution kernel. The solution, as described by the Exponential Shadow Maps work, is to detect these artifacts and fall back to using Percentage Closer Filtering in these situations.

2.7 Shadow Volumes

Shadow Volumes [8][16] do not require the additional depth buffer to generate shadows like the previously mentioned algorithms. Instead, object silhouettes are extruded along the light direction and the volumes created define the shadowed areas. When shading is done, containment within a volume indicates the point is in shadow. The GPU's stencil buffer is used in order to split the projected scene into lit and shadowed areas. This requires that all shadow volume faces are rendered themselves and can incur a large rasterization overhead.

3

Irregular Adaptive Shadow Maps

3.1 Implementation Details

The Irregular Adaptive Shadow Maps algorithm has been implemented with the help of Microsoft Direct3D 11 API and requires a GPU with support for Shader Model 5 to run. Certain features, like unordered access resources and structured buffers, have been used that are not available in previous versions.

A framework was written in C++ for Microsoft Windows 7 to handle windowing, input and resource loading. Additionally, a Wavefront OBJ loader was developed for geometry import required for testing. The OBJ format was extended with a custom binary format for improved loading times.

A forward and a deferred rendering path were implemented to allow for thorough testing of the shadow algorithm. As described in this chapter, the two rendering paths both open up possibilities for various extensions and improvements to the implementation of IASM.

3.2 The Shadow Buffer

Traditionally, a shadow buffer is generated by rendering a depth only pass from the point of view of the light. The results are rasterized into a depth texture that is later used when shading the scene to decide which points lie in shadow and which are lit. Certain algorithms, e.g. partition based algorithms like CSM, can use several textures to represent different parts of the scene. Similarly, IASM splits the shadow buffer into several parts, however, it does not use texture resources to represent that data. Instead, a single structured buffer is used to cover the entire shadow area. Unlike CSM and other statically partitioned algorithms, the structured buffer we use allows for dynamic distribution of resolution based on arbitrary heuristics, which we give an example of in section 3.4. The structured buffer used in this work utilizes 32 bit floats to store the depth values. There is no restriction on what precision can be used however. On the contrary, since the structured buffer is not dependent on the GPU's support for texture formats, it becomes trivial to use any custom precision.

In our implementation, the shadow buffer is split into 4x4 square tiles where each tile covers the same amount of area in light space. In order to utilize most of the shadow buffer, the light frustum is placed in such a way that the camera moves along the circumference of the circle contained within the square covered by the orthographic light frustum. Figure 3.1 shows a top-down view of how the light frustum is placed, relative to the camera, as the camera rotates.

3.3 Scene Analysis

The structured shadow buffer allows for replication of a traditional texture based shadow buffer when resolution is split evenly across all tiles. As previously described, however, it also enables us to dynamically move resolution

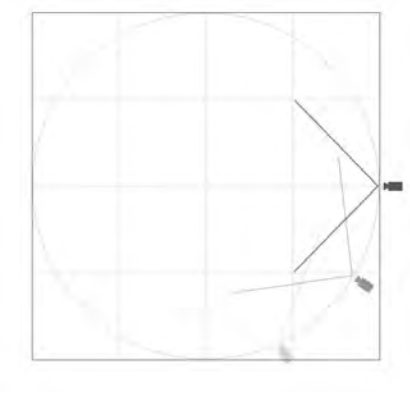

Figure 3.1: As the camera rotates, the light frustum is positioned in front of the camera to maximize the usage of the shadow buffer.

from one tile to another when needed. An important heuristic for this process is to analyze the scene, rendered from the point of view of the camera, and decide what parts of the resulting image require more or less shadow resolution.

During rendering of the scene we utilize a second structured buffer to store a per-tile summed importance value used to optimally set up the shadow buffer for the following frame. Each pixel is attributed a value based on how far away from the camera its world space position is. We use equation 3.1 to extract a pixel's importance value p_i, where w is the fourth component of the pixel's screen-space homogeneous coordinates and m and k are range constants. The higher the value of k, the farther away a pixel can be and still make a large contribution to the importance buffer. Constant m modulates the importance value based on the distance from the camera. The expression $(1 - w/k)$ requires clamping to range $[0, 1]$ to ensure the importance distribution is not ruined by negative values. This provides a mechanic for identifying tiles that cover the larger amount of geometry close to the camera.

$$p_i = 1 + m(1 - w/k) \tag{3.1}$$

Considering our tile to pixel ratio and the parallel nature of the GPU, we are guaranteed to have concurrent writes to the same memory location. To obtain identical results between frames and avoid jittering artifacts, the importance value writes will need to be atomic and thus utilize the DirectX 11 exposed function `InterlockedAdd`.

3.4 Tile Resolution Distribution

A varied set of heuristics has been explored to find a feasible and visually adequate solution to partitioning the finite resolution available for each tile. Chapter 5 describes a number of different algorithms. Here we present the best performing one as identified in the limited time we spent investigating.

A set of 16 resolutions are defined that serve as the total shadow buffer resolution after each is assigned to a tile. The per-tile importance data gathered during the scene analysis step is sorted, giving each tile a rank based on its importance value. This rank describes what resolution each tile gets, e.g. the highest ranked tile gets the highest resolution, as demonstrated by figures 3.2 and 3.3. This results in the tiles covering areas close to the camera receiving a larger resolution than tiles covering areas farther away when the importance buffer is built as per the steps presented in the previous section. A further possibility is to bypass the importance buffer entirely and set the tile resolutions to a predefined distribution when the layout of the scene is known, e.g. during in-game cutscenes.

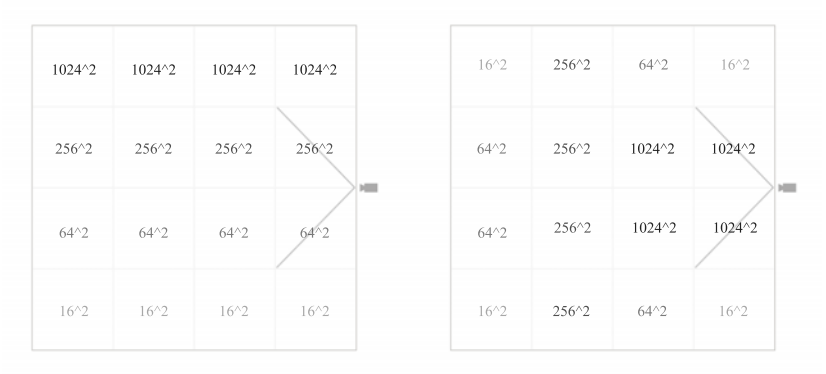

Figure 3.2: The shadow buffer resolution distribution before scene analysis (left) is adapted to the camera position within the light frustum (right). The tiles are ranked based on their importance with the closer tiles receiving the larger part of the available resolution.

3.5 Shadow Rendering

The shadow map is rendered in 16 passes, one for each tile. Before rendering the shadows, every object in the scene is frustum culled using the four side planes of each tile's frustum, ignoring the near and far planes. The near and far planes are calculated individually for each tile every frame depending on the scene geometry. In the forward renderer, the geometry is first partially culled with the four side planes after which the bounding spheres of the objects that pass the culling test are used to find the shortest and longest distances to the light for each tile. These two values define the near and far planes for the tile frusta. In the deferred renderer, the near and far planes are obtained from the pixel shader during the G-buffer pass where all geometry is rendered to extract the parameters required for the lighting passes. A shadow matrix is used to transform the pixel's position to light space and the resulting depth is atomically stored to a structured buffer after a min/max comparison with previous values. This provides us with a tight fitting frustum for each tile adaptively set based on the geometry in

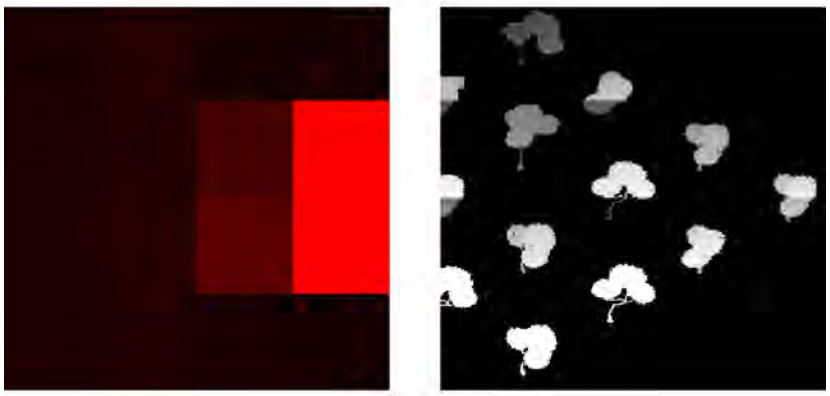

Figure 3.3: The tile importance buffer (left) is used to decide how to partition the available shadow buffer resolution (right). Note how the empty bottom and top right tiles are deemed unimportant and thus receive the lowest available resolution even though they are spatially close to the camera.

the scene.

The shadow buffer, as previously explained, is stored as a structured buffer. Considering that we will not be rendering to a traditional depth buffer, the shadow pixel shader does not require any render targets to be bound. Instead, we have our structured buffer bound as an Unordered Access View (UAV) and write the depth values to it manually. This is fortunate because the unorthodox split of the buffer into tiles results in a tiled memory layout incompatible with the conventional linear (or GPU specific tiled) texture access as performed by the hardware (figure 3.4).

Listing 3.1 shows how we translate a rasterized fragment's position in the pixel shader to its relevant linear memory space index. The tile resolution and its reciprocal are constants passed in to the pixel shader as t_r and t_s respectively. The reciprocal is used to transform the fragment position to the $[0, 1]$ range and the tile resolution allows for conversion of the resulting

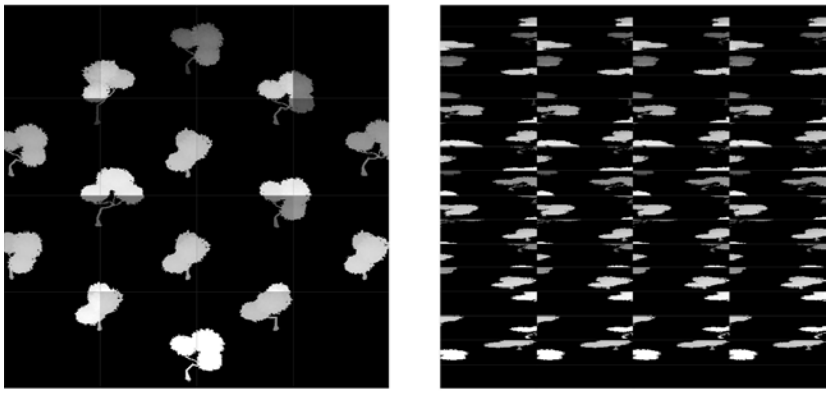

Figure 3.4: The pixel shader responsible for shading the scene requires a view of the shadow map as if stored in a single texture (left) while the memory layout used is per-tile linear (right).

normalized position to an index inside the tile. The third shader constant i_0 represents the current tile's first element's position in the structured shadow buffer. Finally, like all previous writes to structured buffers, we write the depth atomically.

```
float2 uv = float2(pos.x * ts, pos.y * ts);
uint idx = i0 + floor(uv.y * tr) * tr + floor(uv.x * tr));
uint depth = asuint(1.f - pos.z);

uint oldDepth;
InterlockedMax(shadowBuffer[idx].depth, depth, oldDepth);
```

Listing 3.1: The screen-space pixel position is used to find the pixel's location in the structured buffer. The depth value is then written atomically to avoid data loss when several writes to the same location in memory take place.

3.6 Scene Rendering

Mapping a texture onto a single tile, as described in the previous section, requires only a negligible amount of additional work. When shading the scene, however, we need to map a texture onto the entire shadow buffer. In this situation the tiles's sequential storage introduces an extra layer of complexity.

We begin by making the shadow matrices available to the shaders. The main shadow matrix $\mathbf{S_0}$ covers the entire shadow buffer while the tile matrices $\mathbf{S_{1-16}}$ represent the per-tile fitted frusta. The world position of the fragment is transformed into shadow space with the help of $\mathbf{S_0}$ and its projection is used to identify which shadow tile it overlaps, as illustrated by listing 3.2.

$$\mathbf{p_t} = \lfloor \mathbf{p_f} d/t \rfloor \tag{3.2}$$

$$i_t = \mathbf{p_{t_0}} + \mathbf{p_{t_1}} n \tag{3.3}$$

The tile index i_t is given by equations 3.2 and 3.3, where $\mathbf{p_f}$ represents the two-dimensional fragment coordinates on the image plane, t and d are the tile and shadow buffer dimensions respectively, and n is the width of the shadow buffer in tiles.

The implementation details for this step differ slightly between the forward and deferred renderers. The forward path uses the vertex shader to transform the world position into each shadow tile space with the matrices $\mathbf{S_{1-16}}$. This puts less of a burden on the pixel shader, which in practice is often the more costly one. The disadvantage, however, is made clear by the tile index not being available in the vertex shader, requiring all 16 matrix transforms to be calculated for the results to be available in the pixel shader.

```
float4 shadowPos = mul(worldPos, S0);
...
float2 uv = (shadowPos.xy + 1.f) / 2.f;
uv.y = 1.f - uv.y;

const float q = 4;
uint tileX = (uint)floor(uv.x * q);
uint tileY = (uint)floor(uv.y * q);
uint tileIndex = NUM_TILES_X * tileY + tileX;
```

Listing 3.2: The pixel's position projected to shadow space is used to identify what shadow tile the pixel is contained within.

The resulting depth values are stored in four *float4* vectors and passed to the pixel shader. A fragment's tile index is used to choose which of these 16 depth values is picked for the shadow comparison.

In a highly tessellated scene it is likely that a single matrix multiplication in the pixel shader is preferred instead. For the deferred rendering path it is not feasible to do the transforms in the vertex shader as that would mean all 16 depth values would need to be stored in the G-buffer, greatly increasing the memory cost. Instead, the tile index is stored in the G-buffer and the matrix multiplication with the correct tile matrix is done during the shading pass.

For the shadow comparison to be possible, we need to map a fragment's position, projected onto the shadow casting light's image plane, to an index in the structured shadow buffer. To facilitate this, a tile resolution array and an accumulated resolution array are passed as constants to the fragment shader. The tile resolution array stores each tile's resolution for that frame while the accumulated resolution array holds the sum of the squared resolutions for all previous tiles. The formula $\sum_{i=0}^{n-1} r_i^2$ is used to find the accumulated value for tile n, where r_i is the tile resolution for tile i. In essence, this provides each tile with the knowledge of where it is located in the structured buffer's memory. Adding the accumulated resolution to a

fragment's tile index gives us the fragment's index in the structured buffer and the location of the light view depth sample required for doing the shadow comparison for that fragment.

A fragment's tile index is calculated by finding where the image plane coordinates of the fragment overlap the tile and extracting the corresponding index for that location. The tile overlap is equivalent to the remainder from the component-wise division of the shadow buffer space coordinates by the tile dimensions. Since we work with screen-space coordinates, we first convert to the $[0, 1]$ range and then proceed by using equations 3.4 and 3.5 to find the fragment's index in the shadow buffer. Here, r is the resolution of the tile containing the fragment, i_0 is the accumulated resolution for that same tile, and i_b is the sought shadow buffer index.

$$\mathbf{p_b} = \lfloor (\mathbf{p_f} - \mathbf{p_t} d/t) nr \rfloor \tag{3.4}$$

$$i_b = i_0 + \mathbf{p_{b_0}} + \mathbf{p_{b_1}} r \tag{3.5}$$

4

Results

4.1 Test Environment

The implementation of Irregular Adaptive Shadow Maps for this dissertation was done exclusively on an Intel i7-950 3.07 GHz CPU running Windows 7 x64 with 12 GB of RAM. The GPU used was an NVIDIA GeForce GTX 660 Ti with 2 GB of GDDR5 video memory. Performance was measured with the Visual Studio edition of NVIDIA Nsight version 3.2.2. The Direct X 11 Cascaded Shadow Maps sample was integrated into the IASM framework and used for comparison.

Two different scenes were used for acquiring the results. The first one consists of a ground plane and several instances of a tree model. The tree's contiguous geometry results in a tight bounding box more likely to pass frustum culling tests. The second scene is made up entirely of one model depicting two buildings and adjacent objects like stones, etc. The large bounding box enclosing this scene virtually guarantees to fail all frustum culling tests for the shadow map tiles, demonstrating a worst-case scenario for the algorithm.

4.2 Visual Fidelity

The following images are captured at a resolution of 1280x720 pixels. The IASM tile resolutions are 1x1024, 2x512, 1x256, 4x128, 2x64, 2x32 and 4x16. The precision of the buffer, as mentioned before, was chosen to be 32 bits, resulting in a shadow buffer of 6860800 bytes, or 6.5 MB. The CSM implementation uses three cascades, each with a 1024 buffer. To match the IASM implementation, the resolution was chosen to be 32 bits as well, which results in a total of 12 MB used for the CSM shadow buffer.

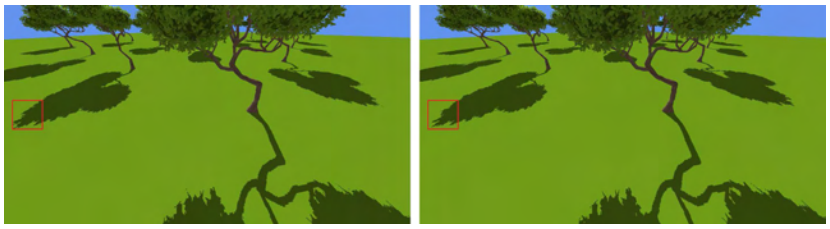

Figure 4.1: Cascaded Shadow Maps (left) compared to Irregular Adaptive Shadow Maps (right). In this scene both algorithms give a similar shadow resolution close to the camera. Small improvements can be noticed with IASM on the sides where the area is covered by the second CSM cascade (highlighted).

In the scene depicted by figure 4.1 IASM matches the visual performance of CSM closely. Near the camera, only minor differences can be seen caused by slightly different near/far planes for the shadow frusta and the fact that the CSM cascade covers a differently sized area compared to the IASM tile. In situations where an area is covered by one of the larger CSM cascades but falls on a high priority IASM tile, it can receive a higher resolution compared to CSM, resulting in a more detailed shadow, as demonstrated by the highlight in figure 4.2.

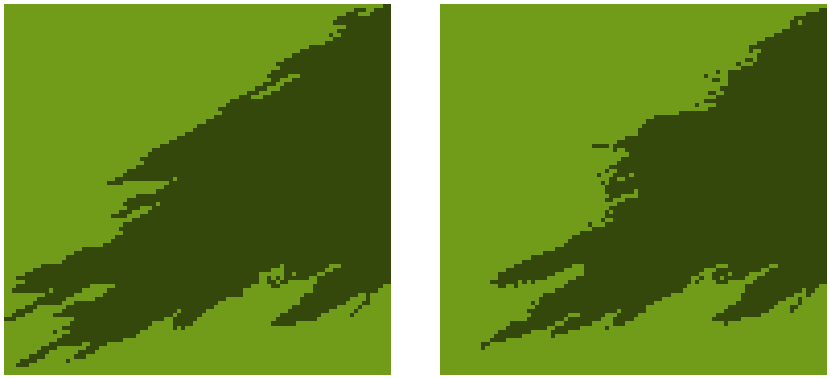

Figure 4.2: When using CSM and an area covered by the second or higher cascade (left) falls in an IASM tile that is deemed high priority enough (right) the higher resolution of the tile can improve the resulting shadow.

Both the first CSM cascade and the highest resolution IASM tile have a buffer of 1024x1024 texels. If the frustum does not differ greatly between the two, the shadows encompassed by those frusta will be very similar in quality. There is, however, a weakness in the IASM algorithm due to the way the camera is moved within the shadow frustum based on its orientation. In figure 4.3 we can observe the camera field of view relative to the shadow frustum. It is also easy to identify that whenever the camera is close to one of the four frustum corners, a boundary where four tiles meet is located very close to the camera. This is problematic not only because a tile boundary is now potentially visible, but also because this requires four tiles to have a high resolution in order to not introduce easily identified artifacts. We visualize this phenomenon in figure 4.4.

As previously mentioned, the forward renderer finds the near and far planes for the shadow tile frusta by iterating over all objects to be rendered and using their bounding spheres to find the shortest and longest distances to the light source. This same step could be applied to the deferred renderer

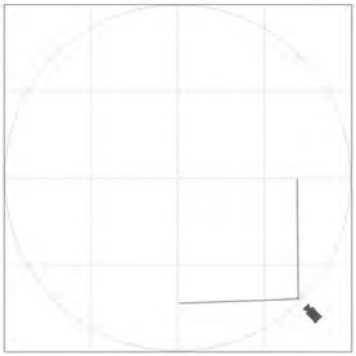

Figure 4.3: A camera orientation that positions it near one of the shadow frustum corners will result in the boundary where four tiles meet to be located very close to the camera.

Figure 4.4: When the boundary where four tiles meet falls close to the camera the variable resolution between tiles is easily noticed (left, middle). The CSM algorithm does not have this problem (right).

as well. In order to avoid this extra cost on the CPU, however, a GPU alternative was explored. Figure 4.5 shows a comparison between CSM and IASM running the deferred renderer. The near and far plane calculated for each shadow tile help produce a higher quality shadow in contrast to the CSM version.

In the same figure, on the far left, we can again see the four tile corner responsible for the accentuated resolution artifacts. In this situation, the tile covering the low resolution shadow contains little geometry compared

Figure 4.5: Cascaded Shadow Maps (left) compared to deferred Irregular Adaptive Shadow Maps (right). The highlight shows a four tile boundary giving rise to noticeable artifacts, although the overall shadow quality close to the camera is slightly better compared to CSM, thanks to the tight per-tile frusta.

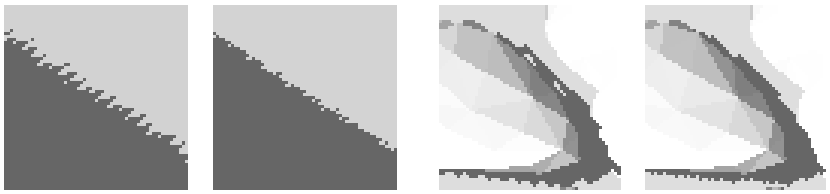

Figure 4.6: Pairs of crops from figure 4.5 showing differences between CSM (left) and IASM (right).

to other adjacent tiles. Even though the geometry is located relatively close to the camera, the importance heuristics are not enough to promote the tile to a higher resolution. We discuss possible solutions and improvements to this artifact in more detail in Chapter 5. Figure 4.7 visualizes the shadow buffer from this last scene. The bottom right tile is responsible for the low resolution shadow visible in figure 4.5.

4.3 Performance

The first scene analyzed is the tree scene depicted in figure 4.1 and the second is the village scene from figure 4.5. Table 4.1 lists the captured times

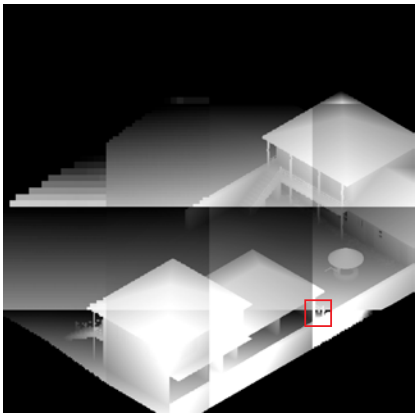

Figure 4.7: The per-tile shadow frusta with improved near and far planes help utilize the shadow buffer precision to a greater extent.

for both CSM and IASM. The importance data copy is arguably expensive in this context, yet it is a cost that can be hidden. The framework used in this research is made up of a simple renderer with very little processing outside of rendering requirements. The importance data gathered during scene rendering in frame n, which is the last pass rendered in our test case, is immediately required for shadow rendering, the first pass rendered, in frame $n + 1$. This creates a stall while the CPU is waiting for the GPU to copy the resource. The copy is asynchronous, however, and if meaningful work can be found to execute on the CPU at this time, the copy cost disappears. In a game engine, additional work is usually required for various simulations that would hide the cost of the GPU data copy.

The deferred renderer has a higher readback cost incurred by the additional buffers. While the forward renderer only needs the importance data, the deferred path copies the near and far plane values for the shadow tile frusta as well. This copy falls under the same time frame as the importance data copy and, as noted above, it can be hidden by additional work. Depending on the scene or engine design, however, this might not be enough.

	Readback	Shadows	Scene	Frame
Trees (CSM)	-	7.5ms	2.7ms	15.1ms
Trees (IASM)	4.6ms	6.4ms	2.7ms	18.8ms
Trees (deferred IASM)	7.9ms	6.5ms	2.6ms	22.6ms
Trees (deferred delayed IASM)	0.9ms	7.0ms	2.7ms	15.9ms
Village (CSM)	-	5.5ms	2.2ms	12.5ms
Village (IASM)	2.3ms	28.6ms	1.9ms	38.1ms
Village (deferred IASM)	8.5ms	28.9ms	2.0ms	45.0ms
Village (deferred delayed IASM)	0.0ms	28.6ms	1.9ms	36.0ms

Table 4.1: Performance cost of various stages for the two scenes visualized in figures 4.1 and 4.5.

It is possible to minimize the readback cost by delaying the schedule of the copy until after the shadows have been rendered. The importance data used for the tile resolution distribution will, in this scenario, be a frame old, as will the near and far plane values in the deferred renderer. While no artifacts were discovered for reasonable camera movement, it is likely the delayed data could introduce visible glitches if the camera moves a large distance between frames.

Compared to CSM, the shadow rendering stage performs slightly better with a cost of 6.4 milliseconds versus 7.5 milliseconds for the tree scene. The depth information is focused in the center of the shadow buffer, as visualized in figure 4.8, leading to some tiles not requiring any rendering, resulting in virtually no cost for that tile. Additionally, some evidence of bad frustum culling can be found in that same figure. The second tile from the left in the topmost row has a render cost of 0.24 milliseconds when it is obviously empty. This is caused by the tree close to the bottom right corner of that tile failing the frustum culling test and being scheduled for rendering in that tile, even though it later gets culled during clipping and does not contribute to the depth buffer at all. A tighter bounding shape, such as an axis-aligned or

oriented bounding box, would have most likely improved shadow rendering performance in this scene.

For the village scene the timings look significantly more dire. The total shadow rendering costs 28.6 milliseconds with each tile taking an average of 1.79 milliseconds. As mentioned before, this scene represents a worst case scenario that effectively removes frustum culling entirely, causing the whole scene geometry to be rendered in each shadow tile. This result reveals the significance of aggressive frustum culling and localized geometry clusters.

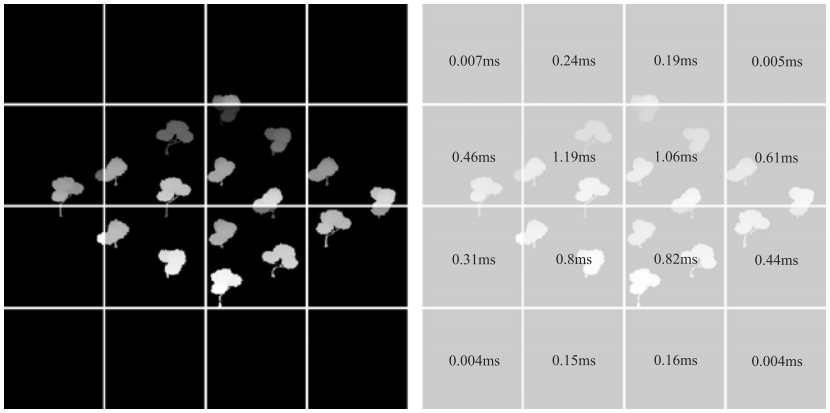

Figure 4.8: The shadow buffer from the scene in figure 4.1 visualized together with the cost of rendering each tile.

5

Discussion

Implementing an optimal shadow rendering solution for an interactive computer graphics system is not a straightforward process, especially as it relates to computer and video games. There are many different scenarios that give rise to different shadow algorithm requirements not always orthogonal to each other. A wide landscape rendered in an engine that allows large visibility distances can be approached in a different manner than a narrow urban scene with limited visibility. An open world game allowing the player to move the camera to any location and viewing shadows from any angle differs significantly from an "on rails" game with a deterministic and predefined camera movement. While this work tests and verifies a very limited subset of interactive shadow situations, it nevertheless introduces a new, feasible shadow mapping algorithm.

From the beginning, IASM was designed with adaptivity in mind. One of the most important decisions the algorithm needs to make is how to distribute the available shadow buffer resolution between the tiles. One of the first heuristics chosen to make this decision was to dynamically adapt the tile resolutions without predefined tile sizes. This meant the importance data and the memory available would define by how much tile sizes would

increase or decrease. The resulting arbitrary tile sizes introduced shadow acne and flickering too severe for this approach to be viable. This finding quickly lead to the final algorithm where tile resolutions are chosen from a predefined set of acceptable resolutions. It is worth noting, however, that a solution half way between the above two might perform even better. The set of predefined tile sizes does not necessarily have to contain only 16 sizes (one for each tile). Providing tiles with a resolution driven by what is required, as opposed to what is available, would result in higher quality shadows at the cost of potential memory spikes when many tiles require a high resolution. The inverse would also be true; when many tiles are empty, a very low resolution can be chosen, instead of an available high resolution, saving memory for that frame in the process.

In the average game utilizing three-dimensional graphics, common scenes tend to be quite dense. Because of this, shadow tiles with very low resolution are of limited use, if any. A low resolution is still needed to optimally use the memory available, but it must be used for geometry far away from the camera. A tile covering geometry close to the camera, no matter how little geometry, requires a reasonable amount of resolution to avoid introducing artifacts caused by perspective aliasing. This is the reason why the importance data calculated for IASM is heavily based on distance from the camera.

Using low tile resolutions when appropriate can have other benefits in addition to saving memory. IASM provides a trivial way to detect when a tile is empty before even attempting to render any depth values to it. By using the near and far plane pre-pass results, it can be detected if any object overlaps the tile frustum. If nothing overlaps, there are no near and far plane values, meaning nothing will render to the tile. Flagging a tile as empty can save additional post-processing on a buffer with no meaningful information.

A problematic situation can potentially arise when the tile resolution distribution is set in such a way that four adjacent tiles cannot have a similarly

high resolution, as demonstrated in figures 4.3 and 4.4. Here, the camera ends up pointing directly at the tile boundary where the edges are made visible by the varying tile resolutions. A different approach to tile placement, relative to the camera location, can diminish this issue. It can also be observed in figure 4.3 that once the camera is located on any corner tile, part of the tile area is wasted behind the camera. Once more, this would be less of an inconvenience with improved tile placement.

There are two major adaptive parts to IASM. The resources available are moved between tiles to provide the highest resolution where it is needed the most, usually close to the camera. The other adjustment is made to the near and far planes of the shadow frusta for each tile and thus further improves the resulting resolution of the shadow. An extension is possible by updating the sides of the shadow frustum as well. The frustum sides have been empirically chosen to encompass all geometry in the scene. More accurate values extracted from the geometry are likely to increase the shadow quality even more.

6

Conclusions

There exists a vast array of ideas presented by research in real-time shadow mapping. Some expand and improve upon previous work while others introduce entirely unique concepts. With computer hardware evolving at an incredible speed and new features, such as compute shaders, constantly being exposed, the capacity for new discoveries is larger than ever.

With the research presented in this dissertation, we have endeavored to produce an alternative shadow mapping algorithm for use in real-time applications such as games. New features from Direct3D 11 are utilized to access data in, until now, unorthodox ways and allow for more flexibility in general.

The IASM implementation focuses on shadows as cast by directional lights over a relatively wide area. Shadow filtering is considered an algorithm agnostic extension and was not applied in order to facilitate performance and quality comparisons between IASM and other algorithms.

Akin to Cascaded Shadow Maps and other partition based algorithms, IASM strives to save memory and improve execution speed by splitting the shadow map into tiles and focusing resources where they are most needed.

In contrast to previous work, however, IASM utilizes structured buffers to store the shadow buffer as opposed to traditional texture resources. The structured buffer opens up unique possibilities when it comes to data layout and access patterns. In addition, scene analysis is applied to make an informed decision when distributing shadow resolution across tiles and when extracting the data required to create optimal tile frusta.

The partitioning scheme implemented in IASM results in a relatively high number of tile edges, causing geometry to be rendered multiple times when it straddles any tile boundary. It becomes evident that aggressive per-tile frustum culling is required for optimal performance.

For both quality and performance comparisons an implementation of CSM is used. The test scenes were chosen to demonstrate performance when draw calls consist of small, localized geometry clusters as well as larger entities extending across the entire scene. The former performs really well and is comparable to CSM. The processing cost is spent on tiles where the most attention is required, generally meaning tiles that cover areas close to the camera. The latter, on the other hand, benefits very little, if at all, from frustum culling, resulting in geometry being rendered in every tile, effectively multiplying the rendering cost by the number of tiles.

Performance of IASM can vary greatly between different scenes, yet with reasonable geometry design competitive performance can be achieved while using less memory than other algorithms. With this work we have introduced a new and feasible shadow mapping implementation with many possibilities for future extensions.

Bibliography

[1] Timo Aila and Samuli Laine. Alias-free shadow maps. In *Proceedings of the Fifteenth Eurographics conference on Rendering Techniques*, EGSR'04, pages 161–166, Aire-la-Ville, Switzerland, 2004. Eurographics Association.

[2] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Convolution shadow maps. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, EGSR'07, pages 51–60, Aire-la-Ville, Switzerland, 2007. Eurographics Association.

[3] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. Exponential shadow maps. In *Proceedings of graphics interface 2008*, GI '08, pages 155–161, Toronto, Ont., Canada, 2008. Canadian Information Processing Society.

[4] Jukka Arvo. Tiled shadow maps. In *Proceedings of the Computer Graphics International*, CGI '04, pages 240–247, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Practical shadow mapping. *J. Graph. Tools*, 7(4):9–18, December 2002.

[6] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, University of Utah, 1974. AAI7504786.

[7] Hamilton Yu-Ik Chong. Real-time perspective optimal shadow maps. Technical report, Harvard University, 2003.

[8] Franklin C. Crow. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.*, 11(2):242–248, July 1977.

[9] Qinghua Dai, Baoguang Yang, and Jieqing Feng. Reconstructable geometry shadow maps. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 4:1–4:1, New York, NY, USA, 2008. ACM.

[10] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 161–165, New York, NY, USA, 2006. ACM.

[11] Wolfgang Engel. Cascaded shadow maps. In Wolfgang Engel, editor, *Shader X5: Advanced Rendering Techniques*, Shader X / ed. by Wolfgang Engel, pages 197–206. Charles River Media, 2006.

[12] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive shadow maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 387–390, New York, NY, USA, 2001. ACM.

[13] Markus Giegl and Michael Wimmer. Fitted virtual shadow maps. In *Proceedings of Graphics Interface 2007*, GI '07, pages 159–168, New York, NY, USA, 2007. ACM.

[14] Markus Giegl and Michael Wimmer. Queried virtual shadow maps. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 65–72, New York, NY, USA, 2007. ACM.

[15] Jon Hasselgren, Tomas Akenine-Möller, and Lennart Ohlsson. Conservative rasterization. In Matt Pharr, editor, *GPU Gems 2*, GPU gems / ed. by Randima Fernando, pages 677 – 690. Addison-Wesley, 2005.

[16] Tim Heidmann. Real shadows real time. *IRIS Universe*, (18):28 – 31, 1991.

[17] Tae-Yong Kim and Ulrich Neumann. Opacity shadow maps. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 177–182, London, UK, 2001. Springer-Verlag.

[18] Andrew Lauritzen and Michael McCool. Layered variance shadow maps. In *Proceedings of graphics interface 2008*, GI '08, pages 139–146, Toronto, Ont., Canada, 2008. Canadian Information Processing Society.

[19] Andrew Lauritzen, Marco Salvi, and Aaron Lefohn. Sample distribution shadow maps. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 97–102, New York, NY, USA, 2011. ACM.

[20] Pascal Lecocq, Pascal Gautron, Jean-Eudes Marvie, and Gael Sourimant. Sub-pixel shadow mapping. In *ACM SIGGRAPH 2013 Talks*, SIGGRAPH '13, pages 19:1–19:1, New York, NY, USA, 2013. ACM.

[21] Aaron E. Lefohn, Shubhabrata Sengupta, and John D. Owens. Resolution-matched shadow maps. *ACM Trans. Graph.*, 26(4), October 2007.

[22] D. Brandon Lloyd, David Tuft, Sung-eui Yoon, and Dinesh Manocha. Warping and partitioning for low error shadow maps. In *Proceedings of the 17th Eurographics conference on Rendering Techniques*, EGSR'06, pages 215–226, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[23] Tom Lokovic and Eric Veach. Deep shadow maps. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 385–392, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[24] Tobias Martin and Tiow Seng Tan. Anti-aliasing and continuity with trapezoidal shadow maps. In Alexander Keller and Henrik Wann Jensen, editors, *Rendering Techniques*, pages 153–160. Eurographics Association, 2004.

[25] Morten S. Mikkelsen. Separating-plane perspective shadow mapping. *J. Graphics Tools*, 12(3):43–54, 2007.

[26] Minghao Pan, Rui Wang, Weifeng Chen, Kun Zhou, and Hujun Bao. Fast, sub-pixel antialiased shadow maps. *Computer Graphics Forum*, 28(7):1927–1934, 2009.

[27] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *SIGGRAPH Comput. Graph.*, 21(4):283–291, August 1987.

[28] Paul Rosen. Rectilinear texture warping for fast adaptive shadow mapping. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 151–158, New York, NY, USA, 2012. ACM.

[29] Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Shadow silhouette maps. *ACM Trans. Graph.*, 22(3):521–526, July 2003.

[30] Marc Stamminger and George Drettakis. Perspective shadow maps. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 557–562, New York, NY, USA, 2002. ACM.

[31] Ray Tran. Facetted shadow maps. In Wolfgang Engel, editor, *Shader X7: Advanced Rendering Techniques*, Shader X / ed. by Wolfgang Engel, pages 217–244. Charles River Media, 2009.

[32] Yulan Wang and Steven Molnar. Second-depth shadow mapping. Technical report, Department of Computer Science, University of North Carolina, Chapel Hill, NC, USA, 1994.

[33] Daniel Weiskopf and Thomas Ertl. Shadow mapping based on dual depth layers. In *Proceedings of Eurographics '03 Short Papers*, pages 53–60, 2003.

[34] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, pages 270–274, New York, NY, USA, 1978. ACM.

[35] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In Alexander Keller and Henrik W. Jensen, editors, *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, pages 143–151. Eurographics, Eurographics Association, June 2004.

[36] Andrew Woo. The shadow depth map revisited. In David Kirk, editor, *Graphics Gems III*, pages 338–342. Academic Press Professional, Inc., San Diego, CA, USA, 1992.

[37] Fan Zhang, Hanqiu Sun, and Oskari Nyman. Parallel-split shadow maps on programmable gpus. In Hubert Nguyen, editor, *GPU Gems 3*, Lab Companion Series, pages 0–1. Addison-Wesley, 2008.

[38] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and*

its applications, VRCIA '06, pages 311–318, New York, NY, USA, 2006. ACM.