FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Shadow Volumes in Unreal Engine 4

An exploratory implementation of a custom lighting technique in a state-of-the-art rendering pipeline

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Bálint István Kovács
Registration Number 1227520

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Katharina Krösl

Vienna, 25th June, 2017

Bálint István Kovács

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Bálint István Kovács
Währinger Gürtel 84/11, 1090 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Juni 2017

Bálint István Kovács

Acknowledgements

First and foremost, I would like to thank my advisor Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer and the Institute of Computer Graphics and Algorithms for allowing me to work on such a current, real-world application topic. I would also like to express my deepest gratitude to my thesis assistant Dipl.-Ing. Katharina Krösl for her patience and guidance and for providing a clear direction for my work while allowing me the freedom to explore my own path towards finding answers to questions and solutions to problems.

I would like to thank my beautiful wife Judit for taking care of everything else while I devoted my time to this project and my mother, parents-in-law and two lovely daughters for putting up with me during this period.

Kurzfassung

Die vorgelegte Bachelor-Arbeit untersucht die Möglichkeiten der Implementierung von nicht inkludierten Beleuchtungstechniken in einer State-of-the-Art Spiel-Engine.

Im Speziellen wird demonstriert, wie Shadow Volumes in Unreal Engine 4 in einem Plugin implementiert werden können.

Diese Arbeit diskutiert die theoretischen und praktischen Aspekte der Unreal Engine im Zusammenhang mit Shadow Volumes und liefert detaillierte Informationen über alle Implementierungsschritte. Es werden die dabei zu beachtenden Details beschrieben und die erreichten Ergebnisse präsentiert.

Abstract

The presented bachelor thesis project explores the possibilities of implementing custom lighting techniques in a state-of-the-art game engine.

Specifically, Unreal Engine 4 is analyzed for the feasibility of implementing shadow volumes in a shader-centric plugin.

The thesis discusses the theoretical and practical background of Unreal Engine and of shadow volumes, and provides detailed information on every implementation step. It shows the challenges of customization and the results achieved.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Video and computer game development has never been more accessible than today. Not only is there a plethora of material available online and offline, in books, websites and video tutorials on the subject matter [1][2][3][4][5][6], but there are also countless libraries, frameworks and game engines to let content creators focus on the particular aspects of game design that they are most interested in. There is a broad spectrum of available tools from specific pre-defined functionality to all-in-one game design solutions. The Unreal Engine [7] is located on the latter side of this spectrum. It provides a complete solution for content creators with an exhaustive set of tools at their disposal to create games of any scope, from hobbyist projects through independent small games to massive AAA titles. Rivalry for market share with competing solutions (like *Unity* [8]) creates a healthy competition, but also some specialization between solutions. Fueled by its popularity, the Unreal Engine sets the industry standards and the state-of-the-art in real time rendering as well, balancing high performance with superior graphical fidelity, while providing cross-platform development solutions and ease of use. Therefore, for this thesis we choose to explore the possibilities of the Unreal Engine in terms of customization of the rendering module and especially for using custom shaders with the engine. This is motivated by the re-emerging interest in the Unreal development community in using custom shaders in Unreal Engine [9][10][11]. Custom geometry shaders are a special interest in the community [12][13], a topic for which very few information sources are available. In particular we choose to examine how shadow volumes can be implemented in Unreal Engine, because of their possibilities in special applications like virtual reality sci-fi/horror games and for their performance potential.

1.2 Problem Statement

The performance, rendering quality and relatively good accessibility of the Unreal Engine comes at a price. Although a lot can be achieved with the customization options provided in the editor, the realization process of specific methods can prove difficult. While a large number of users is content with the features and techniques provided out-of-the-box, even a short browsing of the community platforms (forums, wikis, etc.[5][6]) shows the ever-present wish of developers to push the spectrum of possibilities beyond what the Unreal Engine provides, extending its capabilities in all application contexts. There are different ways to create modifications, albeit often at the cost of performance and/or compatibility. This trend is aided by the fact that the complete Unreal Engine source code is accessible online, making the analysis of the inner workings of the engine from the highest level overview to the lowest level specifics possible. Therefore, exploring the possibilities of renderer customization while observing the related structures of the engine is of interest for both practical and scientific reasons as well.

1.3 Contributions

In the following report we provide a detailed description about an exploratory implementation process of a custom lighting technique in Unreal Engine 4. Its main goal is to discover how custom data structures, shader programs and interfaces can be integrated into a state-of-the-art, high performance game engine. This exploration is driven by several different motivating forces: Understanding the composition and organization of such a game engine yields highly desirable insight into current trends and techniques used in real-time rendering and in a game development work flow. Finding feasible ways to integrate one custom technique can help other similar endeavors achieve similar goals. Finally, adding the chosen technique (shadow volumes) to the current version of Unreal Engine (4.14) can open new possibilities in certain game projects. Through our exploratory implementation we show that implementing shadow volumes in Unreal Engine is a non-trivial task, particularly due to the way the engine handles custom rendering in plugins, and the missing topology descriptions needed to make use of custom geometry shader programs suitable for shadow volume construction. The limitations posed by the Unreal Engine on using stencil buffers and on accessing scene depth values of the main render thread from our plugin restricts our implementation to a simplified version of the shadow volumes algorithm, in which only one convex scene object is considered as shadow caster.

Still, although with these limitations, which we discuss in further detail in Section 4.3, our implementation succeeds in

- defining custom shader programs in a plugin,
- loading these shader programs into Unreal Engine,

- creating the shadow volumes in a custom geometry shader (with the help of core engine modifications),
- transferring the resulting render target data back to the editor,
- applying the shadows to the scene from the editor through a post-process material.

In this thesis we provide a detailed description of all the steps necessary to achieve the set goals, we present the evaluations of the developed algorithm and we formulate future improvement possibilities as well.

1.4 Thesis Structure

The remainder of this thesis is structured into four main chapters. In the following Chapter 2 we provide some background details about shadow calculations in general and shadow volumes in particular, as well as about the development history, current form and rendering-related specifics of Unreal Engine. Chapter 3 contains a detailed, step-by-step discussion of the exploratory implementation, focusing on insights gained about the rendering pipeline of the engine. In Chapter 4 we showcase the evaluation of the results, both in terms of performance and correctness. In Chapter 5 we summarize the conclusions of the research and implementation process, and discuss possibilities of future development.

CHAPTER 2

Background and Related Work

In this chapter we provide background information about the chosen technique and the environment in which it is implemented. In Section 2.1 we discuss the details of shadow volumes, their theoretical foundation, their past use and their limitations. In Section 2.2 we discuss Unreal Engine, its development, the specifics of its rendering pipeline and some details on custom programming possibilities and difficulties.

2.1 Shadow Calculations

Displaying shadows is one of the most important additions to computer graphics when it comes to realistic representations of a scene. Shadows contribute to realism by providing information about the relative positions of scene objects to each other and about the location of light sources in the scene to the observer. Without shadows, objects tend to look like they are floating in space. By including this graphics feature, the objects are perceived as affixed to their relative locations in the scene. Furthermore, shadows can have special importance in computer and video games by contributing to game mechanics, like aiding player character positioning onto platforms or providing additional information on objects outside the player's view, etc. [14]).

Because this is such an important feature in computer graphics, a substantial number of different approaches have been suggested and established over time. Some of them are a reflection of the time they were conceived in (e.g. single-sample soft shadows, penumbra maps, smoothies, etc. [14]), while others are well established and continually refined in an effort to optimize them for current hardware and software capabilities and requirements. We need to make one important distinction before continuing in the exploration of shadow calculation methods: Shadowing techniques in the Unreal Engine can be categorized as static or dynamic. Static shadows are not calculated at runtime, but are created in editor when constructing the game world and are baked into textures (lightmaps or light cache [15]) of object models. While these provide an extensive addition to the realism of a scene (and have in fact an important role in Unreal Engine as well), this report will not feature them further. We focus on dynamic shadow calculations, since these types of techniques provide the means to create shadows suitable for the aforementioned roles in computer and video games. This means that shadow locations and contributions to object color have to be calculated and applied in real-time, at the runtime of the application (game). For this thesis, we chose shadow volumes as an exemplary technique to implement.

2.1.1 Shadow Volumes

The concept of shadow volumes is one which has a rich history of development [16], but which already seems to have the period of its highest popularity behind it, as far as its use in computer and video games is concerned [16][17]. To explore the circumstances of why the use of shadow volumes is not found more often in current games, it is important to understand the advantages and disadvantages that this technique has to offer. A version of shadow volumes was first suggested by Franklin C. Crow in his 1977 paper *Shadow Algorithms for Computer Graphics* [18] and has been iteratively expanded upon in parallel with graphics hardware development [19][20][21][22]. The idea behind this technique is that the object casting the shadow is used to construct a new object, the shadow volume. This volume can then be used to check whether any given point of the scene is inside or outside of it, and therefore, whether the point is in shadow or not. The construction of the shadow volume utilizes the position of the light and the shadow caster object. In a first step, the outline/silhouette of the object has to be determined from the point of view of the light [14][23]. For this purpose, the face normals of the object are queried. The list of the silhouette edges can be constructed with the help of two observations about an edge: a.) One of the adjacent triangles faces the light source. b.) The other adjacent triangle faces away from the light source. If both conditions are met, the edge has to be part of the list of silhouette edges. To realize this efficiently, a suitable data structure has to be set up where faces can be efficiently queried through edges. Once the list of silhouette edges is constructed, the shadow volume can be constructed as well. This can be done by using the triangles of the model facing the light source as front cap of the volume and the triangles facing away from it as back cap. The two caps are connected through appropriately constructed new triangle strips. Figure 2.1 illustrates this idea. To actually yield a volume usable for shadow calculations, vertices of the back cap are projected along the direction vector from the light source to the respective vertex of the back cap. This projection can happen either to infinity or to an appropriately long distance. To prevent visual artifacts on the surface of the object facing the light source, the vertices of the front cap are also moved along the same direction vector. For the vertices of the front cap, only a very small bias is used. This way the faces of the front cap get positioned behind the faces of the object from the point of view of the light, preventing shadowed areas from appearing on the directly lit surface. The construction of the volume can be achieved through different methods. One common and modern way of doing so is by using the geometry shader stage of modern rendering hardware. With the appropriate setup, this allows for an efficient and straightforward evaluation of face normals, deconstruction and reassembly of models and insertion of new triangles [14][24]. Therefore, the exploratory implementation described in this report follows this approach.

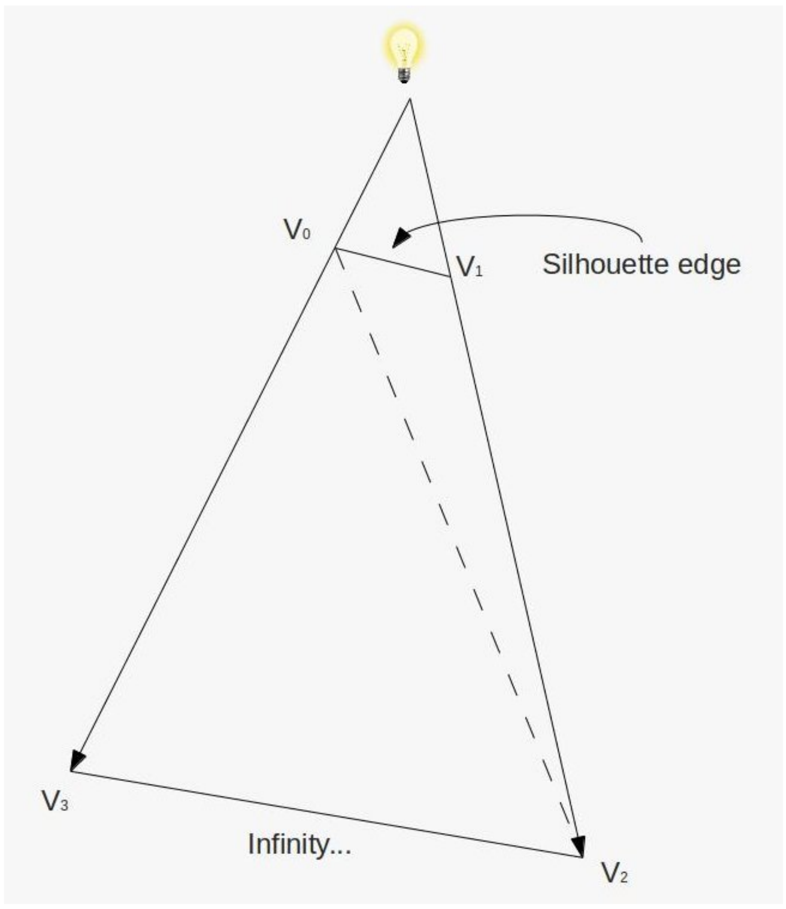


Figure 2.1: Projecting the silhouette edge and constructing the connecting triangle strip. Reprinted from [24].

To use this volume for shadow calculations, a technique is necessary to decide which points of a scene lie inside it and which do not. The points inside this volume are the ones where the caster object blocks the light emanating from the light source. This can be done in several different ways.

Most techniques use the stencil buffer, which is ubiquitous on modern rendering hardware. It provides an efficient way of stepping though the boundaries of the volume and counting the transitions. This can be done by shooting a ray from the camera to a scene object and incrementing or decrementing corresponding fragment values in the stencil buffer with each crossing of the hull of the volume. This requires different rendering steps: First, the scene is rendered without the volume to obtain z-buffer (depth) values for each fragment from the camera's point of view of the whole scene. Then the shadow volume is rendered twice, with back-faces and front-faces each rendered once to be able to perform the z-value tests on them. These z-tests can be realized in different ways, with each technique having advantages and disadvantages alike. The original technique, as introduced in [25], is based upon incrementing/decrementing the corresponding stencil buffer values if the z-test is passed by the object on comparison with the shadow volume rendered with back-face/front-face culling. To achieve correct shadowing in this way, three rendering passes are needed. In the first pass, the scene is rendered with specular and diffuse lighting into one target buffer, and with ambient lighting into another. In a second pass, only the stencil buffer is targeted. Shadow volume quads are rendered, with depth test enabled, into this stencil buffer. Stencil values are incremented for front facing shadow volume quads and decremented for back facing ones. In the third pass, the specular/diffuse render results are added to the ambient buffer only where the stencil buffer has the value of zero, ensuring only ambient lighting in shadowed areas [14].

This however can create problems in certain specific situations. For example, when the shadow volume is in a position where it gets clipped by the near-clip plane [19]. This problem can create unacceptable visual artifacts (missing shadows) and there is no straightforward solution to counter it when using this approach. An alternative method to z-value testing has also a strong connection to the 2004 computer game Doom 3 by ID software [26] and unofficially bears the name of the senior programmer of the game, for popularizing this concept. The z-fail method, also known as Carmack's reverse [19], is essentially comprised of the same steps, but reverses the order of the traversal. Here, stencil buffer values are incremented on depth-test fail with front-face culling and decremented on depth-test fail with back-face culling. Figure 2.3 provides an illustration of this approach. This technique offers a very elegant solution to the near-plane clipping problem. The near-plane clipping has no effect on the shadow rendering, because stencil buffer values are updated only if a quad's distance from the viewers position is bigger than the corresponding scene depth value[14]. However, the z-fail method has its own disadvantage of a different nature: Since it is protected by software patent, its use is restricted [14][27]. This is one of the reasons the source code of the aforementioned Doom 3 had to be altered when releasing it to the public in 2011 [28]. Motivated by certain particularities in the Unreal Engine rendering pipeline, the exploratory implementation presented in this report does not make use of the stencil buffer as featured in these most common approaches. Instead, we use a custom solution that integrates well into the structures provided by the Unreal Engine. Further implementation details are discussed in Chapter 3 .

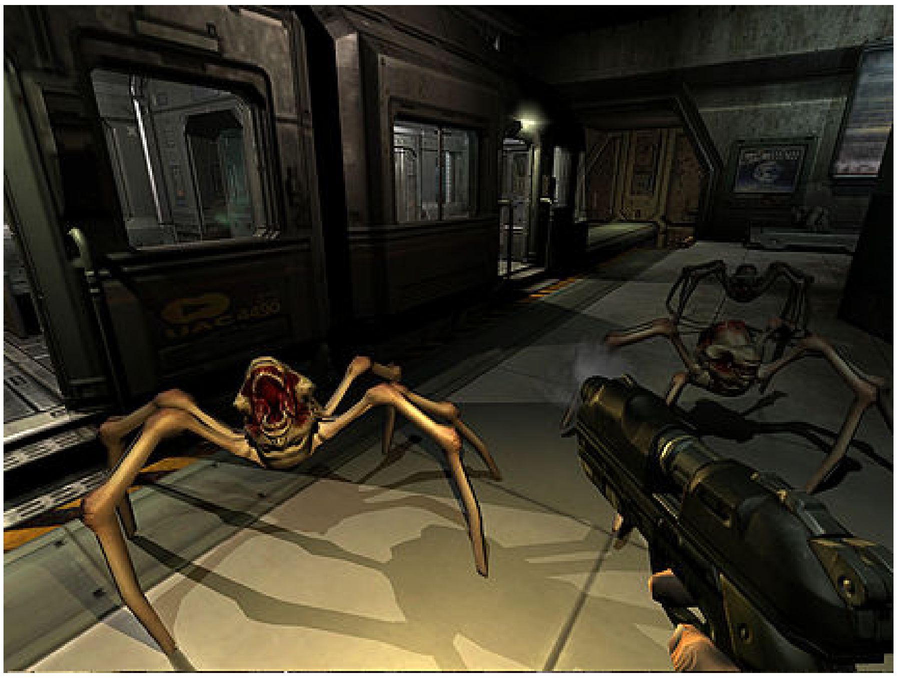


Figure 2.2: Volume shadows in Doom 3. Reprinted from [29].

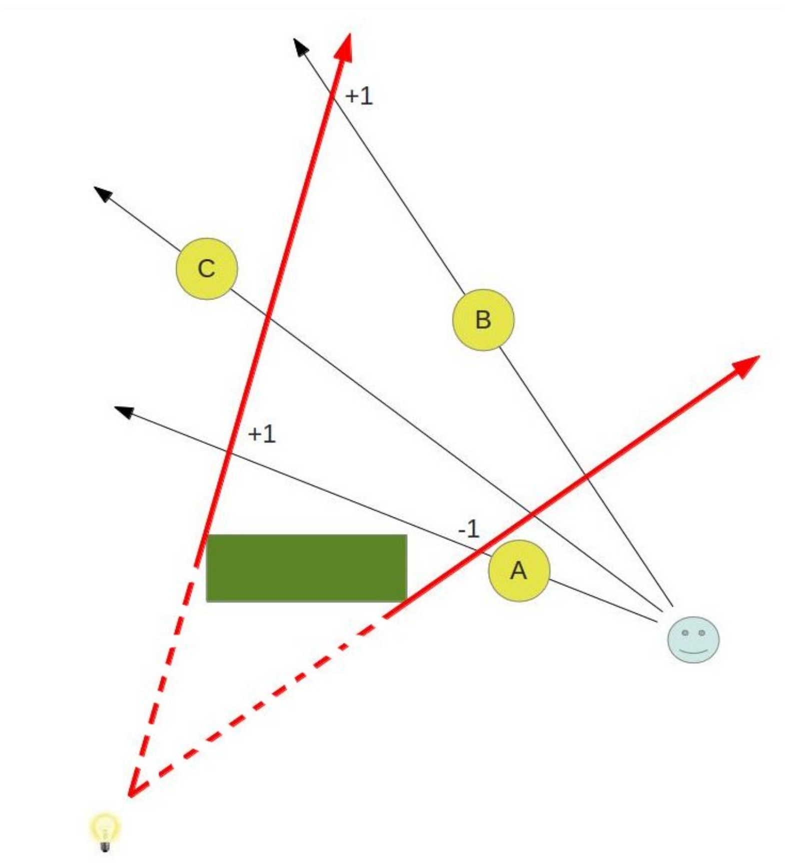


Figure 2.3: Stencil operations on shadow volume hull crossing. Reprinted from [24].

Factoring in all these particularities of the chosen technique, the advantages and disadvantages of shadow volumes are easily outlined. The biggest hindrance for using this technique in modern computer and video games is that the form of the volume, therefore the shape of the shadow, is directly connected to the shape of the casting objects mesh. For optimization purposes, a large number of types of in-game scene objects use alpha transparency to believably represent real-life objects exhibiting certain rich structural traits. These include foliage and flora in general, mesh materials (like nets, fences), flexible linkages, etc. [14][17]. Since the shadow volume is constructed from the edges of a primitive, alpha transparency cannot be taken into account. This causes the shadows cast by such objects to take the shape of those coarse primitives (faces or quads) from which the object is constructed. This in fact is a broader problem, affecting any kind of (partially) transparent object. The current requirements towards state-of-the-art graphical fidelity demand the extensive use of smoke, fog, water and similar effects, and the established implementations of these prohibit the use of shadow volumes in most circumstances.

A different, but important reason for the decline in use of shadow volumes in games is that since this technique produces hard, clean shadows, it only yields realistic results under special circumstances. Appropriate scenes have to play out in dark, fairly tight spaces (narrow corridors, small rooms) illuminated only by a small number of lights, since these circumstances would be the ones creating shadows with similarly hard, well defined outlines in real life situations. There are several suggestions of how to circumvent these shortcoming, like producing soft shadows from shadow volumes [21][22]. However, the added effort in implementation and the reduction in computational efficiency makes a strong case for applying the technique mainly for environments that are more suitable to its strengths. (In fact, one of the most notable examples of application of shadow volumes, the aforementioned Doom 3 [26] is set in the most archetypical scenario for this technique to be effective: In a dimly lit horror/sci-fi space (moon) station, explored from a first person view. Figure 2.2 depicts a scene from this game showcasing its use of shadow volumes.)

However, with the currently reignited interest in virtual reality, especially in the gaming industry, such application scenarios just might be in the center of attention. It can be difficult to asses the directions virtual reality gaming will take with the increase in technical capabilities and the decrease in price of consumer grade virtual reality hardware, but current trends show a boom in (survival) horror themed virtual reality games [30][31]. These type of games tend to set the scene for the player along the guidelines detailed above, and thus have the potential to take advantage of the strengths of this technique. At the same time, the impact of its shortcomings can be concealed by the fact that environments are often dark, narrow and sparingly but harshly lit for effect.

For virtual reality applications, performance considerations are of even higher importance, since low frame rates can be responsible for player health issues, such as headaches, motion sickness, etc. [32]. Variants of the chosen technique could be very well suited for this, since the shadows produced are fully dynamic and provide self-shadowing on any form of surface the shadows falls upon without additional impact on computational resources, above the resources already used for the shadow volume computation. These potential advantages for virtual reality gaming applications provide the reason for this report to explore the possibilities of how this technique can be implemented in a state-of-the-art game engine.

2.2 The Unreal Engine

This section discusses details of the chosen environment, Unreal Engine 4. In Subsection 2.2.1, we provide a brief overview of the development history, leading up to the current form of the engine. In Subsection 2.2.2, we take a quick glance at the rendering pipeline in Unreal Engine. In the final Subsection 2.2.3 we discuss details on the customization of the renderer.

2.2.1 Development of the Unreal Engine

The Unreal Engine [7] is one of the most notable competitors in video and computer game engines available to the industry today. It has been created by Epic Games [33], originally conceived for their 1998 video game Unreal [34]. Throughout several sequels and iterations of the game (Unreal Tournament, Unreal Championship, Unreal II: The Awakening, etc. [35]), the engine was continually developed and extended to industry standards in-house at Epic Software. In 2009, a major shift occurred in the corporate philosophy surrounding the engine, when the Unreal Development Kit, a (under specific circumstances) free version of the Unreal Engine 3 SDK, was made available for external game developers and content creators [36]. The Unreal Development Kit was a pre-built end-user game creation tool, which quickly gathered popularity among game developers and modders alike. The Unreal Engine known today is the result of a second major change on the part of Epic Software, with the release of Unreal Engine 4 in May 2012 [37]. This new version brought with it a large scale restructuring of financial plans for content creators in incremental steps, at the end making the complete version of the engine with all of its included tools and assets essentially free to use for developers under a certain annual revenue. Alongside these changes, Epic Software released the complete Unreal Engine source code to the public (naturally with intellectual property rights retained, as further detailed in [38]). It granted registered members of the Unreal community full access to study and modify the inner workings of the engine and associated first party tools [39]. This allowed the engine to gain increasing interest from small, independent content creators and hobbyist developers, forming a massive community around all the different aspects of Unreal Engine, from game and content creation to engine modification and development. This process helped the Unreal Engine to a very high level of popularity in the video game industry, which is still continuing today: High quality independent and studio-made games are released, powered by the Unreal Engine, and there is high interest from gamer-turned-developers in experimentation and content creation with the Unreal Engine. This widespread popularity is a motivating force for scientific, academical research as well, since the Unreal Engine continually pushes and defines the state-of-the-art in real-time rendering with every iteration.

For the popularity to emerge and be sustainable, the engine has to be in accordance with high standards. The ever growing, multi-billion dollar video game industry [40] demands the most cutting edge in graphical fidelity and computational performance alike. This means, that the rendering pipeline has to be adaptable for new techniques and developments, but, at the same time, has to be highly self-contained and optimized. Since, as discussed above, the Unreal Engine has a considerable development history, optimization is equivalent to a highly integrated interplay of different engine modules created and assembled over time to reach the set performance goals. This makes the Unreal Engine as a whole and its rendering pipeline in particular not only a highly interesting subject to study, but also a highly complicated structure to modify. Implementing custom techniques can also have a considerable performance impact on the rendering, depending on how much low-level requirements of said technique align with the structure of the rendering pipeline.

It is to be noted, that the Unreal Engine package, as it currently exists, is streamlined for the needs of content creators. Content creators require flexibility and technically assisted artistic freedom to an extent where they can realize their vision. This means, that the tools provided have to enable a customization to get content into a desired form in some way, but not necessarily the availability of a particular technique to do so. This has advantages and drawbacks as well, as already noted above: The tight, strongly coherent modules of the rendering pipeline allow for state-of-the-art performance and graphical fidelity partially by restricting the spectrum of methods employable by the user.

2.2.2 The Rendering Pipeline

To be able to implement custom lighting techniques, it is necessary to know the details about the rendering pipelines operation. Therefore, in this report we provide a brief overview of the high-level structuring of the Unreal Engine renderer and include references to sources that contain further details. The Unreal Engine employs a threaded rendering concept, where the renderer is run in its own thread, that is slightly delayed (1-2 frames) from the game thread [41]. This necessitates a careful memory management approach as to preserve data consistencies and avoid race conditions between the rendering and the game thread. To assure that all functionalities exhibit a deterministic behavior, all Unreal object types have two different internal representation: one in the game thread and one in the render thread. This is true for both object data and object functions, with a rigorous set of ownership and transfer rules to every one of them. This is one particularity that makes the unreal Engine highly efficient, but also fairly hard to customize, since back-and-forth of data transfer and inter-thread communication between the game and the rendering pipeline is very strictly regulated, with a deep hierarchy of objects encapsulating each other in different ways.

2.2.3 Renderer Customization

The complex structure of the rendering module makes its custom programming a fairly challenging task. The corresponding Unreal Engine documentation [42] provides some hints and pointers as to where to look for certain details, but first and foremost states that the best place to begin exploring this question is to review the code itself, starting with the *FDeferredShadingSceneRenderer::Render* method . Nonetheless, Unreal Engine documentation is very useful, because it provides the details on the inter-thread relations and connections of some graphics related objects. It is also essential for containing a high-level overview of the hierarchy of rendering-related modules and their execution order in each rendering cycle.

The Unreal Engine contains two different approaches (paths) for rendering meshes: A static and a dynamic rendering path. According to the information provided in [42] on the two different approaches, the static path is more efficient but also more rigid, while the dynamic path allows more control over certain parameters, but takes a higher toll on performance. They make use of *drawing policies*, which incorporate and organize geometry, materials and shader configuration into coherent units that are specific to a given render pass. Both rendering paths use *vertex factories* for this purpose, which contain specific representations of geometry and material definitions. Drawing policies are also responsible for providing data to the *render hardware interfaces* (RHIs). RHIs are a low level abstraction layer that allows to program the Unreal Engine rendering module in a platform independent manner. RHIs contain descriptions of feature sets that allow the targeting of required features instead of specific platforms, discarding the need of a graphics-API-level understanding of the environment. These feature sets are encapsulated in Unreal Engine feature level definitions (ERHIFeatureLevels). Examples given in the Unreal Engine documentation contain descriptions of the following ERHIFeatureLevels:

- *SM5*, mostly corresponding with Direct3D 11 Shader Model 5, but with limitations imposed by OpenGL 4.3 on the usable number of textures.
- *SM4*, being the same as SM5, but without cubemap arrays, tessellation capabilities and compute shaders.
- *ES2*, usable in most mobile environments, generally being in line with OpenGL ES2 capabilities.

The main advantage in using the RHI system lies in defining the program features in such feature sets, and letting Unreal Engine negotiate the supported features on the graphics-API-level. If a certain feature is not supported, Unreal Engine can drop down to the next feature level below until one is found where the platform supports all features required [42].

CHAPTER 3

Implementation

This report follows an exploratory approach for the purpose of assessing the possibilities of implementing a custom lighting algorithm in a state-of-the-art rendering engine. The presentation of the implementation steps therefore reflects on this by providing a step-by-step description of this exploration, resembling the actual progress of the realization process. The structure follows the logical, categorical hierarchy of different parts of the rendering pipeline and thus provides a reflection on the pipeline itself. Therefore, in this chapter we discuss the implementation process as follows: In Section 3.1 we describe the process of including custom shaders in Unreal Engine 4 in general. We present details of accessing the geometry shader stage of the rendering pipeline to prepare for the construction of the shadow volumes and the modifications to the Unreal Engine that make this possible. In Section 3.2 we discuss the actual construction of a shadow volume and the steps to make the custom render targets accessible from the Unreal editor. Finally, in Section 3.3 we show how to use these render targets to apply the shadow volume technique to the scene in the editor.

3.1 Custom Shaders in Unreal Engine 4

3.1.1 Implementation Approach

To use custom HLSL shader files in the Unreal Engine, the communication between the engine, the game and the shaders have to be set up. In the community surrounding the Unreal Engine there is a recurring interest in using custom shading algorithms. Yet, only a handful of actual realizations of such ideas exist, and even less tutorials are available that explain the implementation details developers have to be aware of. One very helpful starting point can be found in [43]. This demonstration however only shows the use of custom compute shaders and pixel shaders within the Unreal Engine. Our exhaustive online research did not yield any results about documentation or other kind of available information on how to enable the use of custom geometry shaders. Therefore, we gathered all implementation details for this project through a meticulous code review of the Unreal Engine source code. Each source file created in the implementation process includes detailed explanations about what steps are necessary to enable the use of the custom shader stages in the form of code comments, meaningful variable and method naming conventions and explicit engine source overrides. Our *ShadowVolumePlugin* class serves as a wrapper for the shader and provides means of transferring data between the editor and the shader and of executing draw commands. Our *ShadowVolumeShader* class contains all declaration and tools needed for the shader file to be loaded, integrated and used with the engine.

3.1.2 Implementation Details

We built our implementation upon one of the pre-packaged Unreal Engine templates. From the templates that are part of a standard Unreal Engine installation, the *First Person Template* [44] is one that is highly suitable for the task at hand. This has several reasons: This template is available out-of-the-box as a `C++` template (and not restricted to Unreal Engine's own blueprint customization). This enables the integration of the shader plugin by providing the possibility to implement helper methods for preparing model mesh and viewport data. The fist person template also has a well usable pre-defined scene with a small arena, movable obstacles and a first person player character. It provides working game logic, including physics, controls and other behavior for the player character and for the interactions with other scene objects. Both the implementation and the evaluation steps benefit from this pre-existing functionality and content contained in the template.

To construct the shadow volume, following the approach described in Subsection 2.1.1, an appropriate data structure has to be defined that contains adjacency information about which two faces share a particular edge. We need this information in order to find the silhouette edges of the shadow-casting model from the point of view of the light source. We have to do this in accordance with the specific details of the rendering pipeline of the Unreal Engine, meaning that it is actually two data structures that we need to employ: One representation in the game world and one in the rendering pipeline. Our solution for the representation used in the game world is a custom, intermediary structure. We modeled it after the halfedge mesh representation [45]. We added a custom method to the character actor class provided in the template project that contains the functionality for the construction of this structure from game models. (*AShadowVolume414Character::SetBuffers*). To achieve this, we pass a pointer to the main *UStaticMeshComponent* [46] of the game world object to this method, from which we extract all unique vertices. We insert them into a vertex vector that we use as vertex position buffer. We use the *TMap* [47] type for the halfedge-like edge map. In this map, keys are 2D vectors, containing the indices of two vertices making up a halfedge, while the associated value is the index of the vertex opposite of the halfedge. We insert all halfedges into the container in this manner. In a final step, we use this map to query the adjacent vertex indices for every vertex in the aforementioned vertex position buffer and construct the index buffer according to primitive topology requirements [48], as detailed in the following paragraph. Our implementation considers one world actor at a time as shadow caster. This is motivated by the aim of our implementation to prove the concept of implementing shadow volumes in Unreal Engine. To achieve this goal, the proposed setup is sufficient, demonstrative and has a reasonable performance. There are some limitations imposed by it, which we discuss in Section 4.3. In Section 5.2 we present an outlook on possible future work to overcome these limitations.

To be able to use the newly constructed mesh representation in the geometry shader stage of the graphics hardware, we have to pass it through the RHI system of the Unreal Engine to the underlying graphics API in a way that retains the adjacency information. As discussed in Subsection 2.2.3, the Unreal Engine includes rendering hardware interfaces to different lower-level graphics APIs, such as OpenGL, Direct3D, etc. [42]. The Direct3D input assembler stage provides a number of pre-defined primitive topologies, contained in the `D3D_PRIMITIVE_TOPOLOGY` enumerated type[48]. Of the types available in Direct3D 11, four can be used to include adjacency information. Two of these are line types, not suitable for the goals of our project. The other two are versions of the triangle list and the triangle strip topology with additional adjacency information encoded in the index buffers. Figure 3.1 shows the way the index buffers have to be set up for these two primitive types. To make use of these topologies in the geometry shader stage, we have to create the adjacency index buffer according to the specifications detailed in [48]. In the next step, we have to provide the buffers to the graphics hardware along with information about the specific topology, so that the information can be interpreted accordingly. However, the exploratory implementation revealed that the Unreal Engine does in fact not include the possibility to declare *triangle list with adjacency* or *triangle strip with adjacency* as the desired topology. This can be observed in the Unreal Engine source code in any of the RHI modules, but specifically at the two methods modified for this project (*D3D11Commands::GetD3D11PrimitiveType* and *RHIUtilities::GetVertexCountForPrimitiveCount*). There are several engine components involved in passing on the requested topology information, leading from higher to lower levels, from the renderer through the RHI to the graphics API (and to the hardware level). Unreal Engine actively checks for the requested topology through the use of the RHI utilities. The engine disallows any member of the aforementioned `D3D_PRIMITIVE_TOPOLOGY` enumerated type that is not explicitly listed in the RHI utilities.

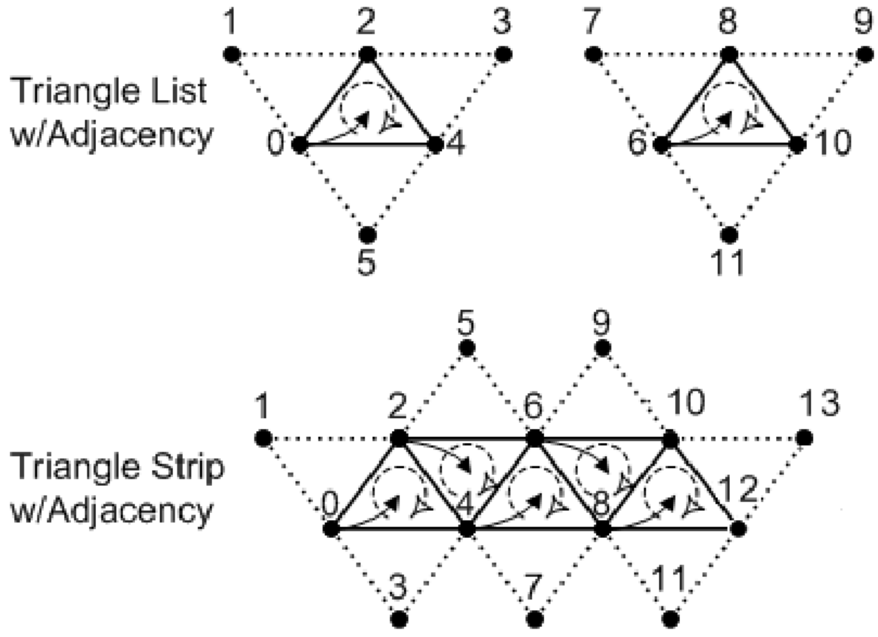


Figure 3.1: Direct3D primitive topologies including adjacency information. The numbers show how the vertex indicies have to be inserted into the index buffer in order for the Direct3D API to interpret them correctly. Reprinted from [48].

3.1.3 Engine Core Modifications

Since the RHI implementations do not contain any of the topologies needed, keeping the implementation of shadow volumes to a self-contained engine plugin is not feasible. Because of this, we propose a complementary core modification of certain engine modules. Since the RHI implementations, as explained above, do not include the necessary formats, it is necessary to modify all of them to retain Unreal Engines platform independence. For this exploratory analysis we chose to modify the Direct3D 11 RHI to demonstrate the concept. It is possible to implement similar modifications in an analogue manner in the other RHI modules. In the case of other Direc3D versions this can be accomplished by copying the corresponding sections. As for other Unreal Engine RHI implementations, like OpenGL, etc., additional research is needed to determine the corresponding primitive topologies. If no one-to-one structural matches exist in an RHI, we can modify the index and/or vertex buffer output of the algorithm responsible to convert Unreal Engine world objects into corresponding topological structures.

To accommodate for the Unreal Engine not providing a way of using the required topologies, we propose to convert one of the pre-existing definitions into a triangle list with adjacency, and another one into a triangle strip with adjacency topology. Converting topology types that are already in Unreal Engine instead of adding new types is motivated by two different circumstances: First, the Unreal Engine is dependent on a complex interplay between different modules, all of which would need enumeration and method extensions to incorporate a new topology type. Second, the two particular Unreal Engine *EPrimitiveType* [49] enumerators that we propose to repurpose for triangle list/strip with adjacency (`PT_26_ControlPointPatchList` and `PT_27_ControlPointPatchList`) are not used by other parts of the engine, so there are no complications to be expected from modifying these topology types. Only three Unreal Engine 4.14 source files, containing code for skeletal mesh handling, static mesh rendering and landscape rendering, make use of one such type (`PT_12`)), in contexts related to tessellation. The Unreal Engine documentation [3] does not specify any particular usage of the *EPrimitiveType* [49]by the engine. Since adding a new topology type would mean an unreasonably extensive engine modification, and at the same time there are types presumably reserved for extensibility, we propose the aforementioned type modifications on the `PT_26_ControlPointPatchList` and `PT_27_ControlPointPatchList` in order to prove the concept of a possible implementation of shadow volumes in Unreal Engine. We propose a cleaner but more extensive approach for future work in Section 5.2. We modify these types to be accepted by the rendering thread, allowing us to forward them to the Direct3D 11 RHI, where they are reconfigured to represent the triangle-list-with-adjacency and the triangle-strip-with-adjacency Direct3D topologies respectively. We modify the internal compatibility checks of the Unreal Engine (e.g. number of vertices compared to length of index array, etc.) accordingly on the internal types as well. With this solution, the model information can be directed through the Unreal Engine from the highest levels (editor, game world) through the intermediaries (rendering thread) to the lowest ones (RHI, graphics API) in a form suitable for shadow volume construction.

3.2 Shader Stages

After solving the problem of transferring model information with correct triangle adjacency topology to the graphics hardware, we can accomplish the actual shadow volume construction in the different shader stages. There are three shader stages, which we use while creating the volume and rendering it accessible for the editor: the vertex shader, the geometry shader and the fragment shader stage. The main part of our implementation is carried out in the geometry shader stage; therefore, we describe this stage in the most detail.

Shaders for the Unreal Engine are contained in engine specific *Unreal Shader Files* (USF), and have to be placed in the appropriate engine installation/build folder to be loaded at engine startup. (Or through the *recompile shaders* command.) The shaders are written in High-Level Shader Language (HLSL), from which the engine generates platform specific shader code on demand in accordance with the environment[50]. We organized our shader code incorporating all three shader stages in one file (*ShadowVolumeTechnique.usf*).

We set up the vertex shader in a way that allows the unmodified pass-through of data to the geometry shader. The *VertexData* structure that we pass as input to the vertex shader stage only contains the three dimensional vertex positions. In the geometry shader the assembled geometry data is received as four adjacent triangles (six vertices), as per specifications in the aforementioned Direct3D primitive topologies document [48]. A very well written and exhaustive explanation of the details of this process can be found in [23] and [24]. First, we calculate the edge vectors from the vertex position and adjacency information. After this information is accessible, we query each edge for information about the face normals of the two adjacent triangles. If one points toward the position of the light while the other does not, we designate the edge as a silhouette edge. For such an edge, we construct a new triangle strip in a reusable helper function, while we emit the two adjacent triangles on the output triangle stream as part of the front cap or the back cap respectively. We offset the former by a small bias to shift it below the surface of the shadow caster while we move the latter by a large number to position the back cap behind all possible scene objects. We move both along the direction vector from the light source towards the vertices included in the edge, as described in Subsection 2.1.1. However, we propose not to project the vertices of the back cap to infinity by the means of a zero value in the w coordinate component. Rather, we offset them by a large bias, while setting the w coordinate component value to 1. This is necessitated by our proposed approach to the data transfer between the shader plugin and the Unreal Engine editor, as described in the following Section 3.3.

We use the fragment (pixel) shader to configure the data in the render targets for the editor to use. We pass no color information through the shader stages and into the fragment shader. We insert only the depth values of each fragment into the current render target, into pixel color channels. A detailed discussion of the necessitating circumstances for this solution are provided in the following Section 3.3.

3.3 Applying Shadows to the Scene

Since the Unreal Engine has such a highly optimized and self contained rendering pipeline, it proves to be difficult to access internal information of the rendering stages. For the shadow volume algorithm to work in the form it is widely used [14], the depth information from the scene has to be accessible to facilitate the incrementing/decrementing of the stencil buffer values. However, since any draw call issued from an engine plugin is executed before Unreal Engines own draw calls of the current frame, this depth information is not ready when needed. Additionally, the data from the internal render targets of the engine's rendering pipeline is protected by the rendering pipeline and cannot be accessed from a plugin. This means that we render the front and back faces of the shadow volume in our plugin, but have no access to the scene depth data. Because of this fact, the scene depth information cannot be used to set the stencil buffer values from the plugin. To the best of our knowledge, the only viable solution to this problem is to combine the data in the editor, in form of a post-processing material, omitting the use of the stencil buffer.

This post-processing material (illustrated in Figure 3.2) is responsible for calculating which regions of the current view are shadowed and applying the appropriate color changes. In it, we compare the pixel color channel values of three render target textures to each other: The render result of the front-faces and the one of the back-faces of the volume, both containing depth values as color, as described above. The third one is a scene depth texture that we render with the help of an Unreal Engine *ScreenCaptureComponent2D* [51][52] tool. To align the view of the SceneCaptureComponent2D with the player's view, we connect it to the player's viewport (as a child object of the *FirstPersonCharacter->FirstPersonCameraComponent*). The screen capture component allows for rendering specific information into a render target defined in the editor. Such render targets can be set up for scene color in different formats, normals or scene depth as color information. We use this last format in the post-process material to compare depth values, since Unreal Engine's rendering pipeline hides all internal render targets where we could obtain this data from. In the Unreal Engine however, this *SceneDepth in R* format is automatically converted into world space distance measure rather than device-z coordinates. Therefore, we alter the shader responsible for its rendering as well. The pixel shader responsible for this particular depth rendering is located in the *SceneCapturePixelShader.usf* shader file. It only renders this particular format, and since the Unreal Engine does not utilize this type of rendering output internally, we can modify it without any side effects on other program parts. The modification simply constitutes a bypass for the method call to convert device-z coordinates into the world-space distance measure and writes them directly into pixel values. In order to get the the best available precision for the shadow calculations, we set up all three render targets in a *FloatRGBA* format. Furthermore, in early implementation experiments we found that in order for the scene viewport, the SceneCaptureComponent2D and the shader plugin render output to be consistent and usable, the render targets have to be configured to the same dimensions as the viewport. Even though matching values are determined before comparison through UV coordinates, the render targets get misaligned, causing visual artifacts (misaligned/shifted shadow regions), when we use their default (varying) sizes. Our solutions to this problem is to set the dimensions of all three render targets at the *BeginPlay* event to the dimensions of the current viewport, making them consistent in every program run, regardless of current viewport size. Figure 3.3 illustrates the three different render targets: Subfigures 3.3.1 and 3.3.2 show the front and back faces of the shadow volume. They contain depth values rendered as color information (all three color channels containing the same depth value) created by our custom shader. Subfigure 3.3.3 shows the render target set up for Unreal Engine's ScreenCaptureComponent2D to contain depth values of the scene in the red color channel. Subfigure 3.3.4 shows the results combined and applied to the scene through our post-processing material.

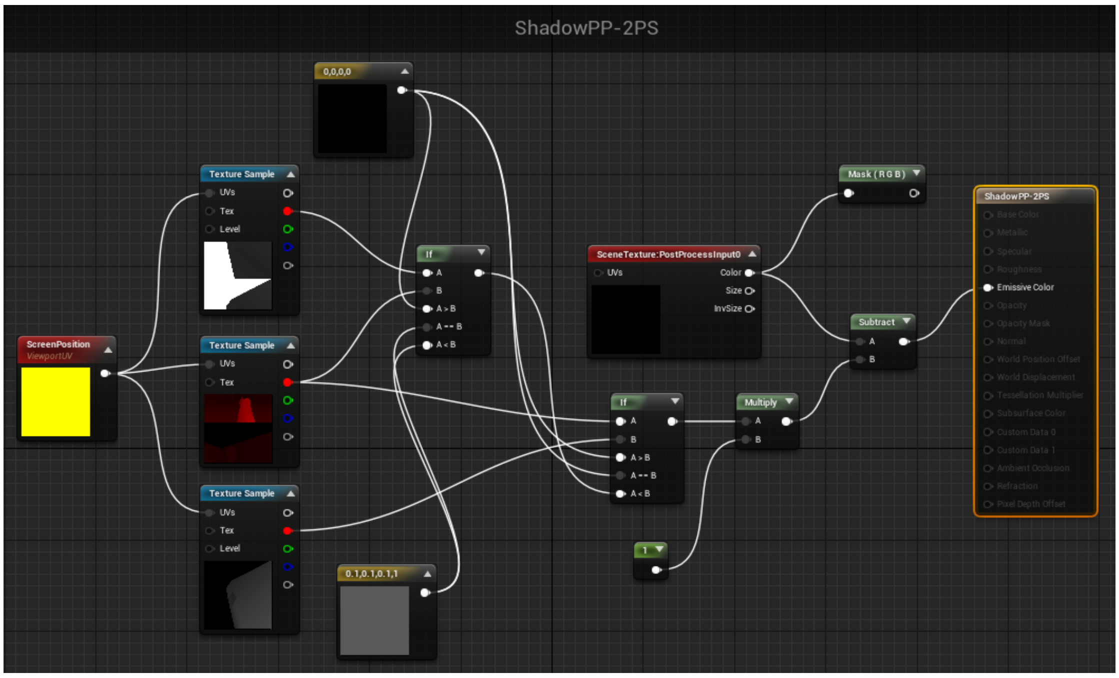


Figure 3.2: The post processing material responsible for deciding which screen pixel is in shadow and applying the appropriate color. Starting point on the left is the ViewportUV node aligning the compared pixels. Then, the three render target sampling nodes, followed by the logical comparison nodes. Finally, the scene texture is darkened in shadow regions and sent to the post-processing material output.

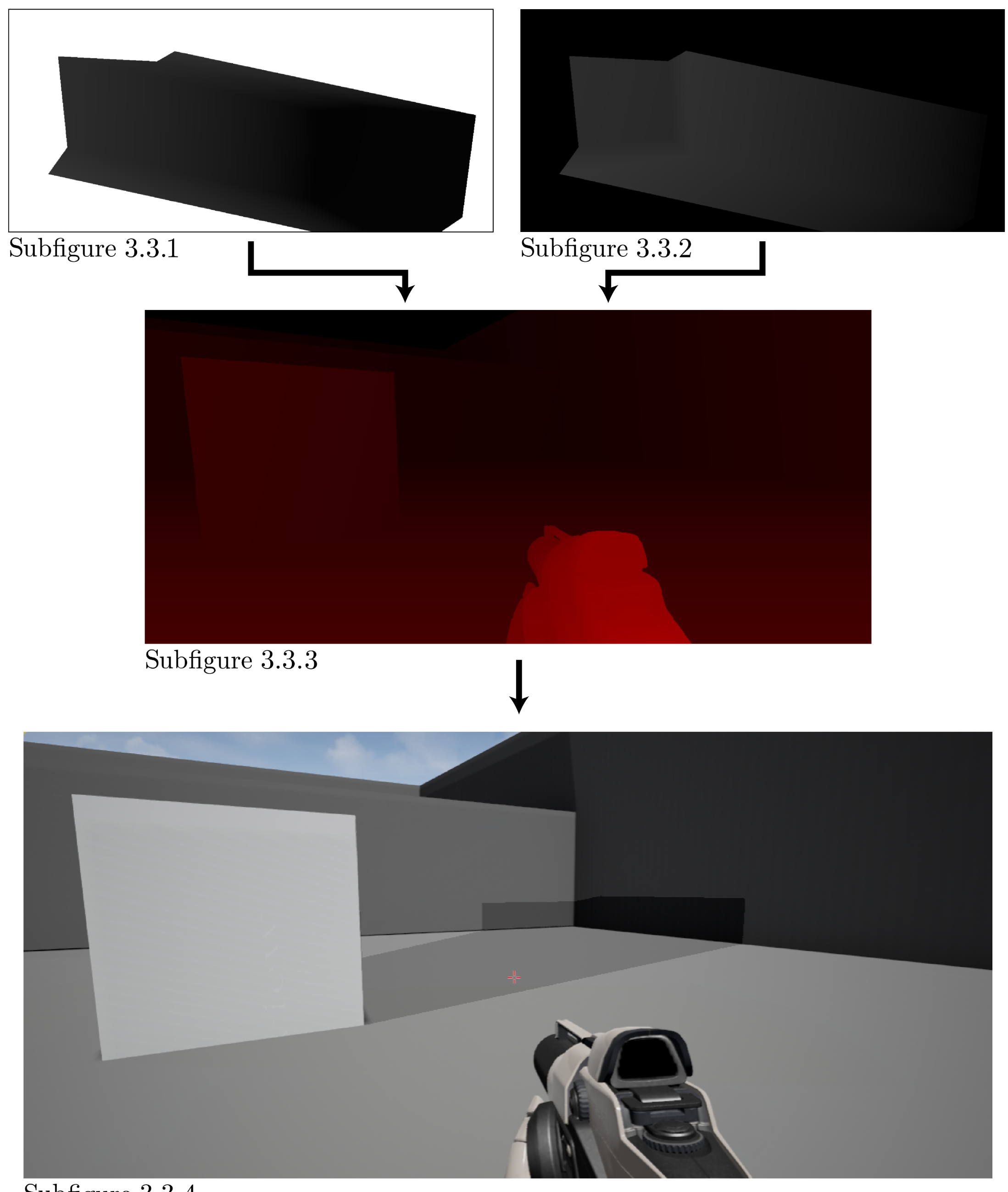


Figure 3.3: Illustration of the render targets. Subfigures 3.3.1 and 3.3.2 show the depth values of the front and back faces of the shadow volume, rendered into all three color channels. Subfigure 3.3.3 shows scene depth values rendered by Unreal Engine's ScreenCaptureComponent2D into the red color channel. Subfigure 3.3.4 shows the values combined and the shadow applied to the scene via the port-processing material.

We achieve the comparison of the depth values via simple logical nodes implemented in the post-processing material. We select pixels that we identify as being in the shadow region from the *SceneTexture* post-process output and alter (darken) their color to achieve the shadowed visuals. As a final step, we apply the material to the scene via a *PostProcessVolume* [53] enclosing the entire test level.

CHAPTER 4

Evaluation

We divide the presentation of our evaluations into four sections. Section 4.1 is a collection of illustrations showcasing the results of our implementation of shadow volumes in Unreal Engine. Section 4.2 contains a summary of the performance of our implementation. In the final Section 4.3, we discuss the known issues and limitations of the current realization.

4.1 Results

The following figures showcase the results of our implementation of the shadow volumes algorithm in Unreal Engine. We saved screenshots from our demo level running in the editor that show the graphical fidelity and correctness of the shadows created with our implementation.

- Figure 4.1 shows shadows of a spherical shadow caster object created with our algorithm.
- Figure 4.2 shows one test setup. We use the sphere annotated as *light source* in the image as omni-directional light source. The cube in the center is the shadow caster. The shadow created with our algorithm is cast on the ground and on the walls, annotated as *volume shadow* in the screenshot.
- Figure 4.3 showcases a visual comparison of the dynamic shadows created via Unreal Engine's own shadow mapping algorithm and the shadow created by our implementation of shadow volumes. Two light sources are used to offset the two shadows for visual comparison. The shadow closer to the player is the one created by Unreal Engines algorithm and is annotated as *Unreal shadow*. The shadow in the back is created with our implementation and is annotated as *volume shadow*. The shadows created by our algorithm retain their sharp edges throughout their

whole length. It can be observed that the summation of shadows in the overlapping shadowed areas are visually correct as well.

- Figure 4.4 allows a closer look at the scene shown in Figure 4.3. (The annotations are identical.) We can observe the aforementioned sharp edges and the correct summing of the two different shadows in the overlapping area.
- Figure 4.5 shows another example of a comparison similar to the one in Figure 4.3, but the shadow caster is a cone.
- Figure 4.6 shows the same test setup as Figure 4.5. We captured this screenshot with the shadow caster cone in motion. It can be observed that Unreal Engine's shadow (annotated as *Unreal shadow*) is blurry, while the shadow created by our algorithm (annotated as *volume shadow*) is sharp. In this situation, this is undesirable. The difference is the result of Unreal's shadow receiving motion blur post-processing, while the shadow created by our implementation does not. We present a suggestion for possible future work in Section 5.2 to counter this issue by adding a motion blur post-processing effect to our shadow volume shadows.

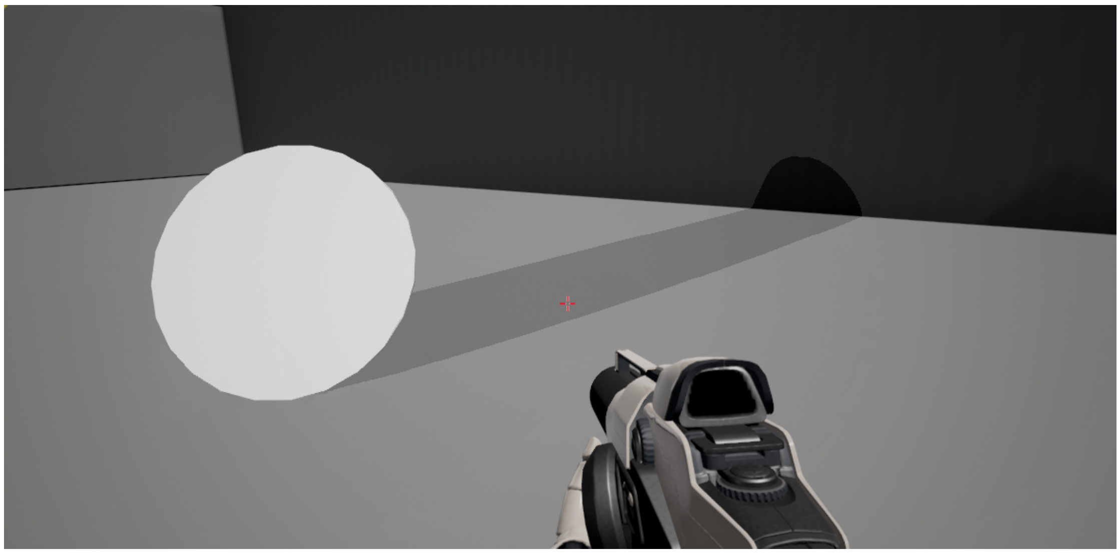


Figure 4.1: Shadows of a spherical shadow caster created with our algorithm cast on the ground/wall.

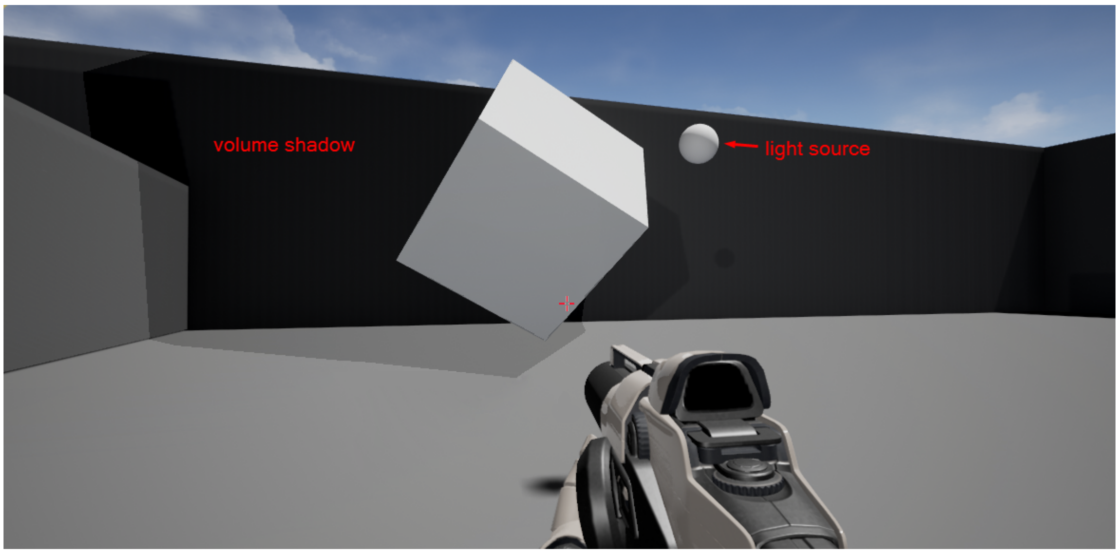


Figure 4.2: Exemplary test setup. A sphere is used as omni-directional light source (*light source*). The cube in the center is set up as shadow caster. The shadow created by our implementation can be observed on the ground and the walls.

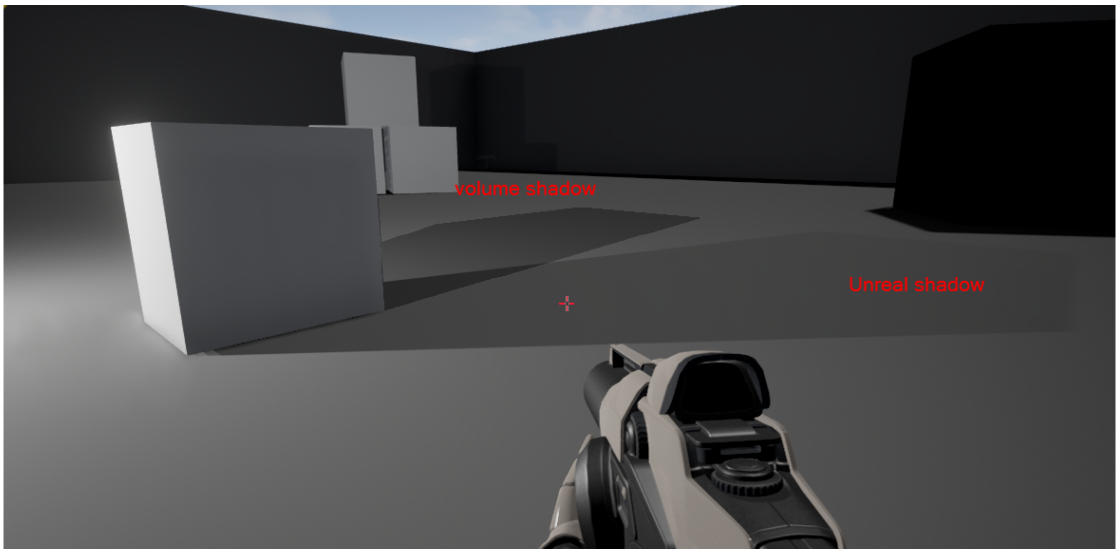


Figure 4.3: Visual comparison of Unreal's shadow map shadow (*Unreal shadow*) and our shadow (*volume shadow*). A cube is set up as the shadow caster. Two light sources are used to offset the two shadows for visual comparison. The sharp edges of our shadows and the correct summation of shadows in the overlapping area can be observed.

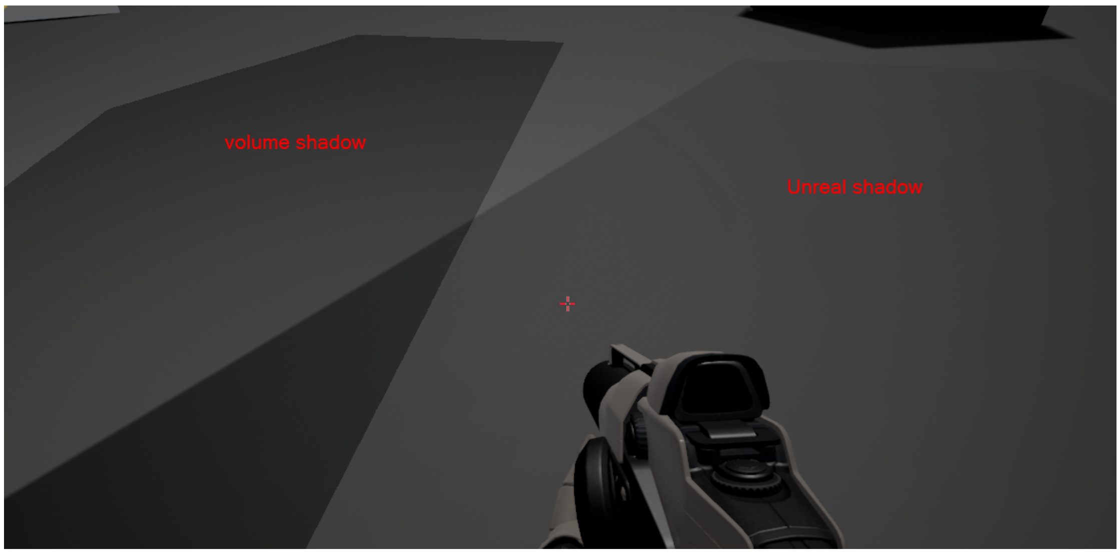


Figure 4.4: A closer look of the areas of interest from Figure 4.4.

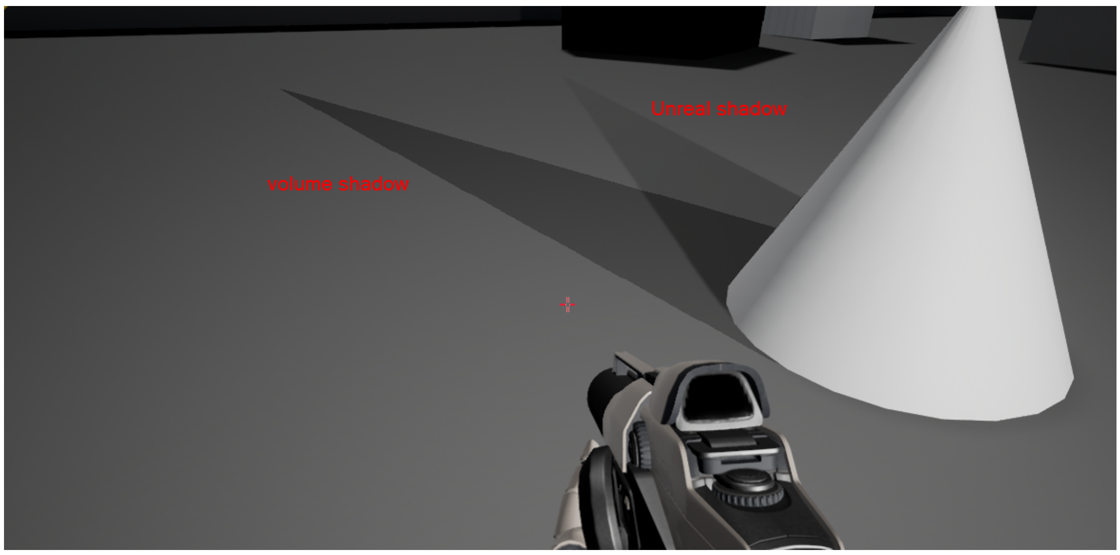


Figure 4.5: Visual comparison of Unreal's shadow map shadow (*Unreal shadow*) and our shadow (*volume shadow*). The test setup is similar to the one shown in Figure 4.3 and Figure 4.4. The sharp shadow edges created by our implementation can be observed even more prominently, especially at the pointed top of the cone's shadow.

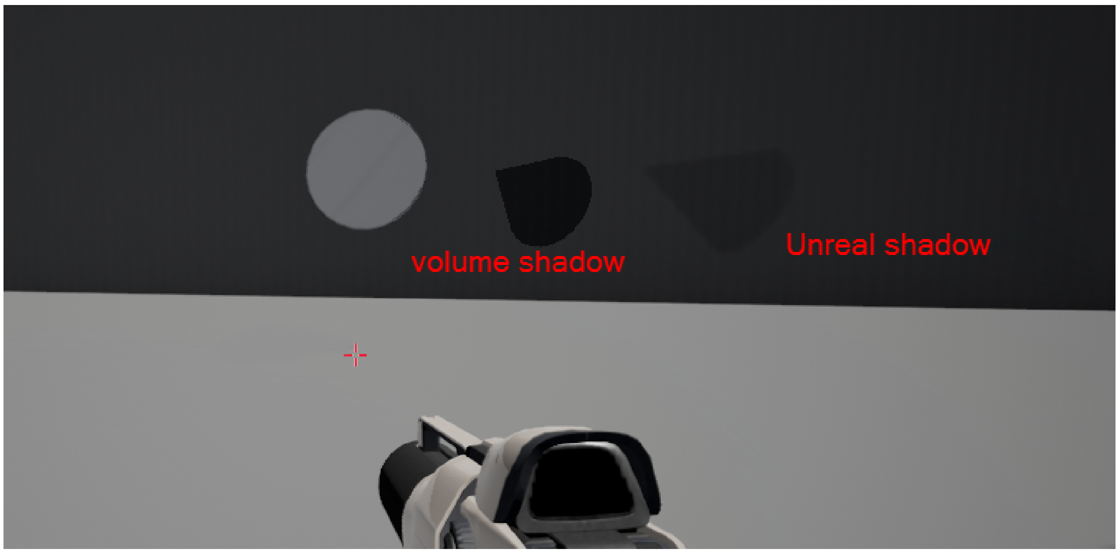
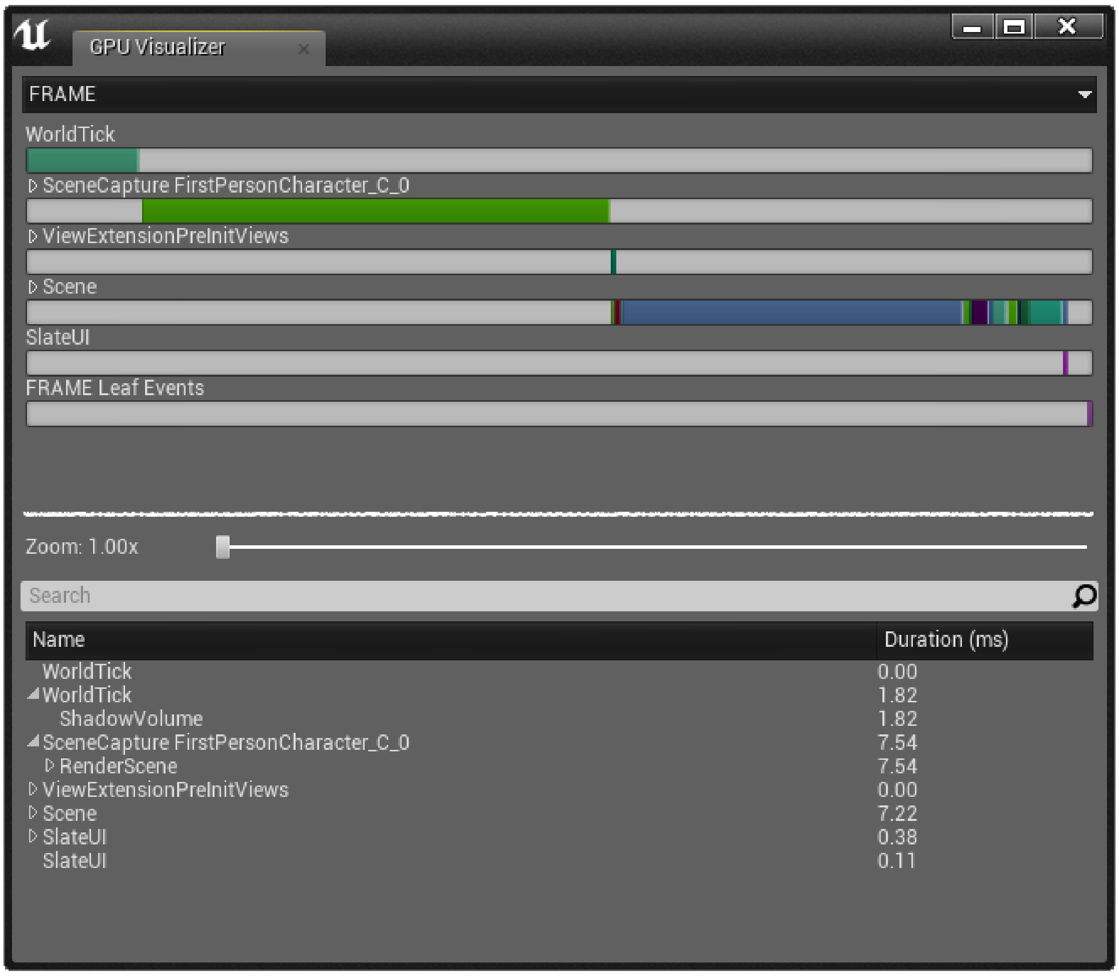


Figure 4.6: Visual comparison of Unreal shadow map shadow (*Unreal shadow*) and our shadow (*volume shadow*) with the same test setup as in Figure 4.5. We captured this screenshot with the shadow caster (cone) in motion. The shadow created by our algorithm is still sharp, while Unreal's own shadow map shadow is blurry. This is caused by a motion blur post-processing effect, which our algorithm does not include.

4.2 Performance

To assess the performance and limitations of our implementation, we derived a test level from the pre-packaged Unreal Engine First Person `C++` Template. The benefits of starting out from this template are discussed in Section 3.1. To test our implementation, we added our plugin to the template, and assigned a post-process material as a blendable to a post-process volume encompassing the entire game arena. We set up a sphere as omni-directional point light, by naming it *shVolLight*. We use different predefined Unreal static mesh objects to test our shadow volumes algorithm, by naming them *occluder2*. The algorithm in our plugin searches among the *AActor* objects in the game world for these two names to assign the roles of omni-directional light source and shadow caster object respectively. Any object that has a static mesh component can be renamed accordingly to take on the role of the shadow caster. The object designated as the light source does not have to be an actual light source used in the Unreal Engine. It can be an arbitrary world actor, since the program only takes into account the position to be used in the shadow volume construction. This solution allows for convenient and flexible testing.

The performance of our implementation can be measured through the built in GPU profiling tools of Unreal Engine. The hardware configuration used for testing was comprised of the following main components: Intel(R) Core(TM) i7-3930K CPU @ 3.20Ghz, 32 GB RAM, NVidia GTX 980ti. The output of the ProfileGPU tool is shown in Figure 4.7. It can be observed, that the impact of the shadow volume shader, with the shadow volume construction and two draw calls (with front-face and back-face culling respectively) at each frame takes 1.82 ms compared to the 7.22 ms needed for the rendering of the rest of the scene, including the application of the post-process material. This means an increase in rendering time of 25.21 percent per frame. This however is not the biggest impacting factor. Since the Unreal Engine does not provide access to the scene depth texture in any other usable way, we have to use the SceneCaptureComponent2D to obtain it, as discussed in Section 3.3. It can be observed in the GPU profiler that this component issues a command for another rendering of the complete scene, adding 7.54 ms additional render time per frame, i.e. a render time increase of 104.43 percent alone. Because of this, the additional render time of the whole technique is 9.36 ms per frame, i.e. an increase of 129.64 percent.



Name	Duration (ms)
WorldTick	0.00
WorldTick	1.82
ShadowVolume	1.82
SceneCapture FirstPersonCharacter_C_0	7.54
RenderScene	7.54
ViewExtensionPreInitViews	0.00
Scene	7.22
SlateUI	0.38
SlateUI	0.11

Figure 4.7: Frame render time contribution of rendering steps. It can be observed that while the construction and rendering of the shadow volume only adds 25.21 percent rendering time per frame, the SceneCaptureComponent2D adds an additional 104.43 percent, making it the biggest impact factor of our implementation on the performance.

4.3 Known Issues and Limitations

As we described in Section 2.2 and in Chapter 3, Unreal Engine imposes restrictions on the transfer of render target data from the rendering thread to the editor and on the use of the stencil buffer. Because of these restrictions, there are limitations present that prevent the full realization of the stencil shadow volumes algorithm as proposed in the related literature we presented in Subsection 2.1.1. The issues and limitations we present in the following section are a result of our adaptation of the shadow volumes technique, as discussed in Chapter 3.

The biggest drawback of not being able to use stencil buffer operations is the fact that our implementation yields correct shadows only for convex shadow caster objects. Since we render the depth values of the front and back faces of the shadow volume into the color channels of two textures, if the object has overlapping parts from the current viewpoint, the depth information of some of the overlapping faces is lost. In such a situation, the textures only contain one of the several corresponding depth values for a particular fragment, making a correct evaluation impossible. This causes incorrect shadows with any non-convex shadow caster.

Our implementation only considers one object in the scene as shadow caster. This results in the behavior that although shadows are cast correctly onto other objects behind the designated shadow caster, they appear to go through them. This causes the shadow of the shadow caster object to show up on the second, third, etc. surface behind it in its original shape. To mitigate this issue, all scene objects need to be occluders to the light source. This would ensure that the shadow of the next larger object covers the shadow of the object closer to the light source, removing this visually incorrect behavior. We discuss possible future work to achieve this in Section 5.2.

Not being able to use stencil operations, specifically the Z-fail method causes another problem as well. The shadow volume is rendered onto the shadowed side of the shadow caster object in a wrong way. We illustrate this issue in Figure 4.8. We can observe that the side of the shadow caster cube facing away from the light source and towards the player's view should be fully self-shadowed. Instead there is only a partial shadow, with an incorrect self-shadowed region (marked and annotated in the image). This issue occurs because in the erroneous region there are no two values to compare the scene depth against. If the player's viewpoint is located inside the shadow volume, only one of the render targets contain values in that region, since the back cap of the object is shifted into distance, as discussed in Section 3.2. Because of this, the player camera is never positioned behind the back cap, so the front faces of the back cap are not rendered. Therefore, if the player is inside the shadow, looking at the shadow caster, the self-shadowing is wrong.

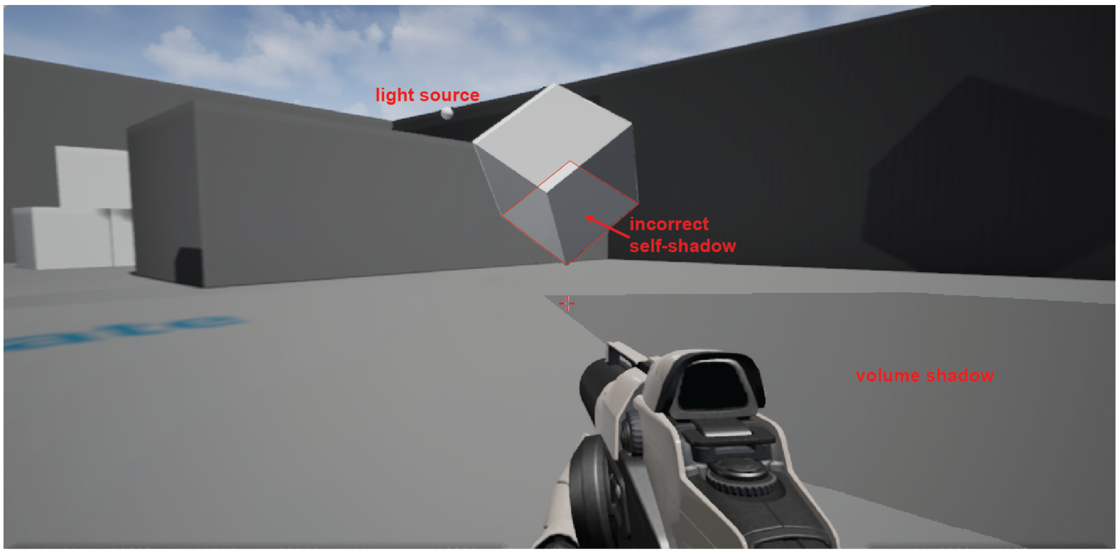


Figure 4.8: Visual artifacts in object self-shadows. The region annotated as *incorrect self-shadow* should be in shadow. This issue is caused by the fact that only one of the two render targets contains a value in the corresponding region, since the back cap of the volume is shifted into distance.

Additionally, even on the shadow casting objects surfaces where the self-shadows are correct, z-fighting can sometimes occur. The z-fighting is due to insufficient precision of the depth values when inserted into color channels and of the depth as color render target provided by Unreal Engine's SceneCaptureComponent2D. The results of this can be observed in Figure 4.9

One rare, yet possible problem is the misalignment of shadows in the viewport. Although this behavior is discussed in the Unreal developers community [54], no fix or information is provided by Epic Games. This is an error which is elusive enough that we can not illustrate it, as we cannot be reproduced at will. It happens because although the SceneCaptureComponent2D is a child object of the FirstPersonCharacter->FirstPersonCameraComponent as explained in Section 3.3, on random program starts it can get misaligned from it. This causes the movement of the two views (the actual scene render produced by Unreal Engine and the depth render target created by the SceneCaptureComponent2D) to only overlap when the view vector is parallel to the ground, but shift apart with increasing angle. For our implementation this means that the shadowed areas are not in the correct position, but also shifted with the view of the SceneCaptureComponent2D. In the the Unreal forums users suggest that this error is only present in the Play-in-Editor [55] test mode and does not occur in packaged games [54]. We find that restarting the editor several times fixes the issue temporarily for the Play-in-Editor mode as well.

Since the main scene rendering thread and the game thread are offset by one or two frames from each other, as discussed in Subsection 2.2.2, this delay can potentially be noticed on lower-end test hardware when rotating the camera quickly. This make for a "rubbery, flexible" feel of the shadows, catching up to the moving shadow caster object once it has settled. On the test hardware we used for evaluation, this delay is barely noticeable, since we achieved sufficient frame rates. (Frame render times can be observed in Figure 4.7.)

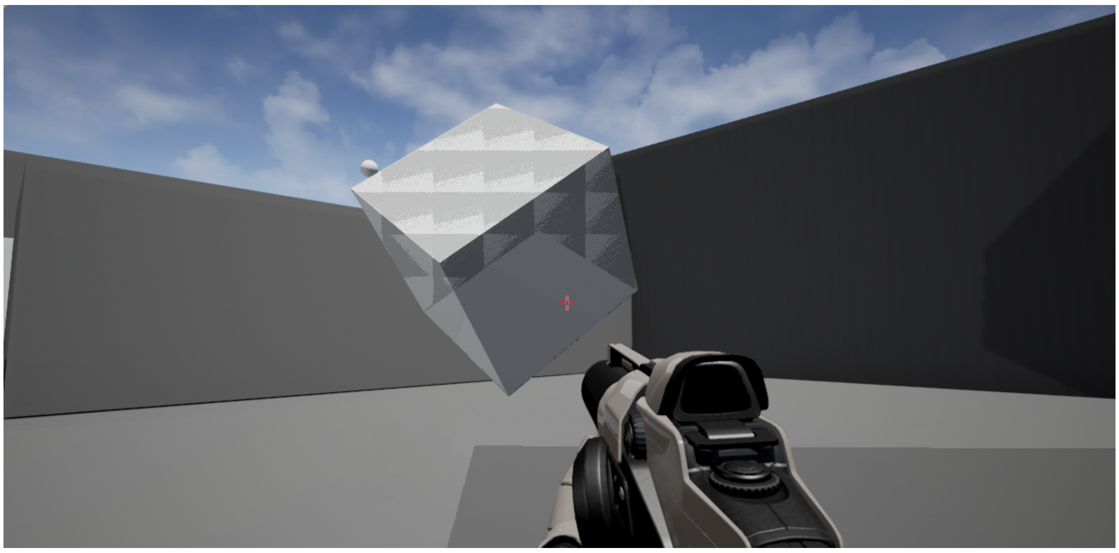


Figure 4.9: Occasional z-fighting in object self-shadow regions caused by insufficient precision of the depth values inserted into color channels.

CHAPTER 5

Conclusion

The final chapter of this thesis is divided into two parts. Section 5.1 contains a summary of our findings during the exploratory implementation process and the following evaluation and test phase. In Section 5.2 we provide a brief outlook on how the method could be improved in future work.

5.1 Summary

In this thesis we presented a way to implement the custom lighting algorithm of shadow volumes in the state-of-the-art game development tool Unreal Engine. We based our implementation process upon a comprehensive research into the Unreal Engine and a shadow volumes algorithm. Through an exhaustive code review, we devised a method to create a working example of a custom technique in the game engine.

We presented a way of integrating custom shaders, especially a custom geometry shader in Unreal Engine through the use of an engine plugin. We showed that for the realization of shadow volumes in Unreal Engine, the modification of core engine files is necessary. We made these modifications, because the platform-independent Rendering Hardware Interface (RHI) implementations of Unreal Engine do not include the primitive topology types describing adjacency information between mesh triangles. This adjacency information however is necessary to construct a shadow volume in the geometry shader stage. We implemented a method in the plugin to create the appropriate data structure from Unreal Engine's game world actors to use for the construction of the volume. This method uses a halfedge-like data structure where halfedges are associated with their opposing vertices in a triangle. We discovered that stencil buffer operations are not possible due to the necessary render target data not being available to our plugin. However, we devised a way to implement our shader while omitting the use of stencil buffers. We used fragment shaders to encode depth information in color channels, which we compared with scene depth values from Unreal Engine's SceneCaptureComponent2D depth-as-color output. We accomplished this via a post-processing material in the Unreal Engine editor. This way we solved the problem of transporting data back to the editor from our plugin. We used the same material to apply shadows to the scene.

We assessed the performance and limitations of our implementation in a test level devised from a Unreal Engine template. We found that the biggest performance loss is due to the use of the SceneCaptureComponent2D, while the construction and rendering of the shadow volume happens in relatively acceptable time. We encountered limitations and drawbacks caused by our implementation. Since we can not use stencil buffers, the current implementation of our algorithm only yields correct shadows for convex shadow caster objects. In its current form, our implementation only considers one scene object as shadow caster. Our algorithm produces wrong self-shadowing and z-fighting on the shadow caster due to the use of depth as color information instead of stencil operations. Due to a not well documented bug in Unreal Engine, the SceneCaptureComponent2D used for scene depth rendering can on rare occasion get misaligned from the player view, causing misaligned shadows. The delay between game and render thread in Unreal Engine can cause a temporary misalignment between the player view and the shadows as well during quick player motion on low performance hardware. We discuss possible future work to address some of these issues in the following Section 5.2.

Our implementation provides deep insight into the underlying structures and organization of a state-of-the-art, industry standard rendering pipeline, while also discovering difficulties in modifying it for specific techniques. Although the implementation has still room to improve on performance and graphical fidelity, it represents a proof of concept for the possibility of customization.

5.2 Future Work

Concerning the issues and limitations discussed in Section 4.3, and taking into account implementation details presented in Chapter 3, several improvements could be targeted in future work.

The current implementation should be extended by devising an appropriate data structure to hold and prepare all world actor objects to allow our implementation to consider them as light occluders. All shadow caster objects need to have shadow volumes constructed and those volumes need to be rendered into the render targets as well. This could be achieved correctly by implementing a fragment shader that is able to merge the depth values of the different volumes into one render target.

The issues of incorrect self-shadowing, z-fighting and of wrong shadows of non-convex objects could be solved by finding a way to use stencil buffers. For shadow volumes and other techniques that make use of the stencil buffer, it is essential to further investigate ways of using the stencil buffer directly on the graphics hardware as described in the related literature, and transfer only the results of the stencil operations back to the editor. For this to be possible, both the scene depth values and the depth values of the shadow volumes have to be made available in our plugin. This could be achieved through some from of low level data transfer (copy) directly from the different render targets of Unreal Engine's main internal scene rendering. This would make using the the SceneCaptureComponent2D unnecessary, solving the problem of random temporary misalignment as well. Making the use of the SceneCaptureComponent2D obsolete would be highly beneficial for the performance of our implementation, providing an almost 50 percent decrease in frame render times.

For the shadows created by our implementation to receive motion blur, additional nodes could be inserted into the post-processing material responsible for applying them to the scene. This could improve graphical fidelity, but would also add computational load.

For the Unreal Engine to retain platform independence, all RHI modules need to be modified to include the primitive topologies containing adjacency information. In future work, these could be overridden in the same manner as we proposed for the Direct3D 11 RHI. For a cleaner but more involved approach, the topologies could be added alongside the original ones without re-purposing any of Unreal's built-in types.

List of Figures

Bibliography

[1] E. Games, “Resources.” https://www.unrealengine.com/resources. [Online; accessed: 12.07.2017].

[2] E. Games, “Ue4 answer hub.” https://answers.unrealengine.com/index.html. [Online; accessed: 12.07.2017].

[3] E. Games, “Unreal engine 4 documentation.” https://docs.unrealengine.com/latest/INT/. [Online; accessed: 12.07.2017].

[4] E. Games, “Video tutorials.” https://docs.unrealengine.com/latest/INT/Videos/. [Online; accessed: 12.07.2017].

[5] E. Games, “Unreal engine community wiki.” https://wiki.unrealengine.com/Main_Page. [Online; accessed: 12.07.2017].

[6] E. Games, “Unreal engine forums.” https://forums.unrealengine.com/. [Online; accessed: 12.07.2017].

[7] E. Games, “Unreal engine.” https://www.unrealengine.com/. [Online; accessed: 20.06.2017].

[8] U. Technologies, “Unity - game engine.” https://unity3d.com/. [Online; accessed: 20.06.2017].

[9] P. St-Amand, “Writing shaders for ue4; where do i start?.” https://forums.unrealengine.com/showthread.php?97053-Writing-shaders-for-UE4-where-do-I-start, 2016. [Online; accessed: 25.07.2017].

[10] ggalt, “How to make custom shader?.” https://answers.unrealengine.com/questions/573425/how-to-make-custom-shader.html, 2017. [Online; accessed: 25.07.2017].

[11] Jobbobbo, “Writing your own custom shaders for ue4.” https://www.reddit.com/r/unrealengine/comments/3s6wqp/writing_your_own_custom_shaders_for_ue4/, 2016. [Online; accessed: 25.07.2017].

[12] P. St-Amand, “Is epic doing anything to address ue4’s crippling shader problem.” https://forums.unrealengine.com/showthread.php?106655-Is-Epic-doing-anything-to-address-UE4-s-crippling-shader-problem, 2016. [Online; accessed: 25.07.2017].

[13] MXADD, “Materials and geometry shaders.” https://forums.unrealengine.com/showthread.php?74531-Materials-amp-Geometry-shaders, 2015. [Online; accessed: 25.07.2017].

[14] E. Eisemann, M. Schwarz, U. Assarsson, and M. Wimmer, *Real-time shadows.* CRC Press, 2011.

[15] E. Games, “Static lights.” https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/StaticLights/. [Online; accessed: 20.06.2017].

[16] H. Kolivand and M. S. Sunar, “Survey of shadow volume algorithms in computer graphics,” *IETE Technical Review*, vol. 30, no. 1, pp. 38–46, 2013.

[17] J. Smart, “Why is shadow mapping the standard?.” https://gamedev.stackexchange.com/questions/106676/why-is-shadow-mapping-the-standard, 2015. [Online; accessed: 20.06.2017].

[18] F. C. Crow, “Shadow algorithms for computer graphics,” in *Acm siggraph computer graphics*, vol. 11, pp. 242–248, ACM, 1977.

[19] J. Carmack, “John carmack on shadow volumes....” https://web.archive.org/web/20090127020935/http://developer.nvidia.com/attach/6832, 2000. [Online; accessed: 20.06.2017].

[20] E. Lengyel, “Advanced stencil shadow and penumbral wedge rendering. presentation at game developers conference 2005,” 2005.

[21] U. Assarsson and T. Akenine-Möller, “A geometry-based soft shadow volume algorithm using graphics hardware,” in *ACM Transactions on Graphics (TOG)*, vol. 22, pp. 511–520, ACM, 2003.

[22] S. Laine, T. Aila, U. Assarsson, J. Lehtinen, and T. Akenine-Möller, “Soft shadow volumes for ray tracing,” *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3, pp. 1156–1165, 2005.

[23] E. Meiri, “Silhouette detection.” http://ogldev.atspace.co.uk/www/tutorial39/tutorial39.html. [Online; accessed: 20.06.2017].

[24] E. Meiri, “Stencil shadow volume.” http://ogldev.atspace.co.uk/www/tutorial39/tutorial39.html. [Online; accessed: 20.06.2017].

[25] T. Heidmann, “Real shadows, real time,” *Iris Universe*, vol. 18, pp. 28–31, 1991.

[26] id Software, “Doom 3.” [DVD-ROM], 2004.

[27] W. Bilodeau and M. Songy, “Method for rendering shadows using a shadow volume and a stencil buffer,” May 7 2002. US Patent 6,384,822.

[28] M. Larabel, “Doom 3 source code published under the gpl.” https://www.phoronix.com/scan.php?page=news_item&px=MTAxODk, 2011. [Online; accessed: 20.06.2017].

[29] “Doom3shadows.” https://en.wikipedia.org/wiki/Shadow_volume#/media/File:Doom3shadows.jpg. [Online; accessed: 20.06.2017].

[30] V. T. E. Team, “12 best vr horror games to play in 2017.” https://vrtodaymagazine.com/vr-horror-games/, 2017. [Online; accessed: 20.06.2017].

[31] M. Herst, “15 vr survival horror games of 2017 and beyond.” http://gamingbolt.com/15-vr-survival-horror-games-of-2017-and-beyond, 2017. [Online; accessed: 20.06.2017].

[32] J. Barrett, “Side effects of virtual environments: A review of the literature (dsto-tr-1419),” tech. rep., 2004.

[33] E. Games, “Epic games, inc..” https://epicgames.com/. [Online; accessed: 20.06.2017].

[34] E. Games, “Unreal.” [CD-ROM], 1998.

[35] C. Plante, “Better with age: A history of epic games.” https://www.polygon.com/2012/10/1/3438196/better-with-age-a-history-of-epic-games, 2012. [Online; accessed: 20.06.2017].

[36] I. Staff, “Epic games announces unreal development kit, powered by unreal engine 3.” http://www.ign.com/articles/2009/11/05/epic-games-announces-unreal-development-kit-powered-by-unreal-engine-3, 2009. [Online; accessed: 20.06.2017].

[37] S. Sakar, “Epic games debuts unreal engine 4.” https://www.destructoid.com/epic-games-debuts-unreal-engine-4-229158.phtml, 2012. [Online; accessed: 20.06.2017].

[38] E. Games, “Unreal engine end user license agreement.” https://www.unrealengine.com/eula, 2017. [Online; accessed: 20.06.2017].

[39] E. Games, “Unreal engine 4 on github.” https://www.unrealengine.com/ue4-on-github. [Online; accessed: 20.06.2017].

[40] E. McDonald, “The global games market will reach usd108.9 billion in 2017 with mobile taking 42 percent.” https://newzoo.com/insights/articles/global-games-market-reaches-99-6-billion-2016-mobile-generating-37/, 2017. [Online; accessed: 20.06.2017].

[41] E. Games, “Threaded rendering.” https://docs.unrealengine.com/latest/INT/Programming/Rendering/ThreadedRendering/. [Online; accessed: 20.06.2017].

[42] E. Games, “Graphics programming overview.” https://docs.unrealengine.com/latest/INT/Programming/Rendering/Overview/index.html. [Online; accessed: 20.06.2017].

[43] F. Lindh, “Hlsl shaders.” https://wiki.unrealengine.com/HLSL_Shaders. [Online; accessed: 20.06.2017].

[44] E. Games, “First person template.” https://docs.unrealengine.com/latest/INT/Resources/Templates/FirstPerson/. [Online; accessed: 22.07.2017].

[45] H. Bronnimann, “Designing and implementing a general purpose halfedge data structure,” in *Algorithm Engineering*, pp. 51–66, Springer, 2001.

[46] E. Games, “Ustaticmeshcomponent.” https://docs.unrealengine.com/latest/INT/API/Runtime/Engine/Components/UStaticMeshComponent/index.html. [Online; accessed: 20.06.2017].

[47] E. Games, “Tmap.” https://docs.unrealengine.com/latest/INT/API/Runtime/Core/Containers/TMap/index.html. [Online; accessed: 20.06.2017].

[48] M. W. D. Center, “Primitive topologies.” https://msdn.microsoft.com/en-us/library/windows/desktop/bb205124(v=vs.85).aspx. [Online; accessed: 20.06.2017].

[49] E. Games, “Eprimitivetype.” https://docs.unrealengine.com/latest/INT/API/Runtime/RHI/EPrimitiveType/index.html. [Online; accessed: 20.06.2017].

[50] E. Games, “Hlsl cross compiler.” https://docs.unrealengine.com/latest/INT/Programming/Rendering/ShaderDevelopment/HLSLCrossCompiler/index.html. [Online; accessed: 20.06.2017].

[51] E. Games, “Scene capture 2d.” https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/Reflections/1_7/. [Online; accessed: 20.06.2017].

[52] E. Games, “Add scene capture component 2d.” https://docs.unrealengine.com/latest/INT/BlueprintAPI/AddComponent/Rendering/AddSceneCaptureComponent2D/index.html. [Online; accessed: 20.06.2017].

[53] E. Games, “Post process effects.” https://docs.unrealengine.com/latest/INT/Engine/Rendering/PostProcessEffects/. [Online; accessed: 20.06.2017].

[54] MarcinW, “Issue with misaligned scene capture 2d.” https://forums.unrealengine.com/showthread.php?106957-Problem-building-anaglyph-post-process-effect-(shifting-red-and-green-blue-channels)&styleid=2, 2016. [Online; accessed: 20.06.2017].

[55] E. Games, “In-editor testing (play and simulate).” https://docs.unrealengine.com/latest/INT/Engine/UI/LevelEditor/InEditorTesting/. [Online; accessed: 20.06.2017].