HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Frau
Annabell Schütze

Vergleich der Effizienz verschiedener Möglichkeiten zur technischen Optimierung der Darstellung visueller Effekte in der Game Engine Unity

Mittweida, 2023

Fakultät Angewandte Computer- und Biowissenschaften

BACHELORARBEIT

Vergleich der Effizienz verschiedener Möglichkeiten zur technischen Optimierung der Darstellung visueller Effekte in der Game Engine Unity

Autor:
Frau

Annabell Schütze

Studiengang:
Medieninformatik und Interaktives Entertainment

Seminargruppe:
MI20w1-B

Erstprüfer:
Prof. Dr. rer. nat. Marc Ritter

Zweitprüfer:
Manuel Heinzig

Einreichung:
Mittweida, 19.09.2023

Faculty Applied Computer Science
& Biosciences

BACHELOR THESIS

Comparison of different efficient ways to optimize the technical aspects of rendering visual effects in the Unity game engine

author:
Ms.

Annabell Schütze

course of studies:
Media Informatics and Interactive Entertainment

seminar group:
MI20w1-B

first examiner:
Prof. Dr. rer. nat. Marc Ritter

second examiner:
Manuel Heinzig

submission:
Mittweida, 19.09.2023

Bibliografische Beschreibung:

Schütze, Annabell:
Vergleich der Effizienz verschiedener Möglichkeiten zur technischen Optimierung der Darstellung visueller Effekte in der Game Engine Unity - 2023
Verzeichnisse: 9 Seiten, Inhalt: 51 Seiten, Anhänge: 24 Seiten
Mittweida, Hochschule Mittweida, Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2023

Referat:

Die Anforderungen an die Qualität und den Realismus von Videospielen und deren visuellen Effekten steigen kontinuierlich. Dies führt zu der Herausforderung, beeindruckende Effekte und eine gute Performance zu gewährleisten. Die Auswahl einer geeigneten Technik für ein Leistungsproblem ist daher wichtig. Aus diesem Grund widmet sich die Bachelorarbeit dem Vergleich verschiedener Methoden zur Optimierung der Darstellung visueller Effekte in Unity. Neben einer Literaturrecherche werden ausgewählte Verbesserungsmöglichkeiten wie zum Beispiel ein Level-of-Detail System oder Batching implementiert. Als nächstes findet eine Evaluation dieser Techniken in einem Testszenario mit ausgewählten Metriken wie beispielsweise Bildrate und Grafikkartenauslastung statt. Anschließend erfolgt der Vergleich ihrer Effektivität in Bezug auf die originale Version des visuellen Effektes. Die Ergebnisse der Untersuchung zeigen, dass insbesondere das Level-of-Detail System und die Culling-Methode einen signifikanten Einfluss auf die Grafikkartenauslastung haben. Zudem weisen die meisten Optimierungstechniken in Bezug auf die Metriken Vor- und Nachteile auf. Obwohl die erzielten Ergebnisse nicht unmittelbar auf die praktische Anwendung in Spielen übertragbar sind, ermöglichen sie dennoch eine wertvolle Vergleichbarkeit der angewendeten Methoden.

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

VFX	visuelle Effekte
FPS	Frames per Seconds
CPU	zentrale Prozessoreinheit
GPU	Grafikprozessoreinheit
LOD	Level-of-Detail
ms	Millisekunden
GB	Gigabyte
MB	Megabyte
KB	Kilobyte
GHz	Gigahertz
DDR4	Double Data Rate 4
RAM	Random-Access Memory
PNG	Portable Network Graphics
bpp	Bits pro Pixel
ETC	Ericsson Texture Compression
DXT	DirectX Texture
GPGPU	General Purpose Computation on Graphics Procession Unit
API	Programmierschnittstelle

1 Einführung

1.1 Einleitung

Die Videospielindustrie erlebte in den letzten Jahren einen Aufschwung, wobei dies mit kontinuierlich steigenden Anforderungen an visuelle Effekte und ihre Grafikqualität sowie realistische Darstellung einhergeht. In der Zeit echt wirkender 3D-Grafiken und immersiver Spielerlebnisse stehen die Entwickler vor der Herausforderung, beeindruckende Effekte in Spielen zu integrieren, ohne die Gesamtperformance zu beeinträchtigen. Die Integration von mehreren komplexen Partikelsystemen kann beispielsweise zu Leistungsproblemen führen und so die Spielererfahrung erheblich beeinträchtigen.
Eine beliebte Entwicklungsplattform zur Umsetzung von Videospielen ist die Unity. Um der Herausforderung der genannten Leistungsprobleme zu begegnen, bietet die Engine für Partikelsysteme verschiedene Optimierungstechniken und -ansätze, um die Effizienz bei der Darstellung visueller Effekte zu verbessern. Diese Methoden umfassen beispielsweise Level-of-Detail Systeme, Batching und Texturkomprimierung. Allerdings existiert trotz der großen Auswahl an Ansätzen keine umfassende Untersuchung zur Effektivität dieser Techniken auf bestimmte Bereiche des Spiels.
Diese Bachelorarbeit widmet sich exakt dieser Thematik und zielt darauf ab, die verschiedenen Möglichkeiten zur technischen Optimierung der Darstellung visueller Effekte in der Engine Unity detailliert zu vergleichen. Dabei sollen ihre Auswirkungen im Spiel auf Metriken wie beispielsweise die Bildrate, CPU- und GPU-Auslastung, Anzahl der Draw Calls sowie den Speicherplatzbedarf in einem Testszenario analysiert werden. Durch den Vergleich der Techniken soll ein leichterer Optimierungsprozess für Partikelsysteme möglich sein, indem die Abwägung der Vorteile und möglichen Nachteile der Methoden erfolgt. Dies trägt zu einer verbesserten und einfacheren Entwicklung von Spielen in Unity bei, indem es Entwicklern eine leichtere Auswahl einer Methode zur Verbesserung der Leistung bei der Darstellung visueller Effekte ermöglicht.

1.2 Motivation, Problemstellung und Zielsetzung

Bei der Entwicklung von Computerspielen sollen möglichst realistische und beeindruckende visuelle Effekte, wie zum Beispiel Partikelsysteme, die Spielerfahrung verbessern. Allerdings führt die steigende Komplexität und Anzahl dieser Elemente meistens zu Herausforderungen wie beispielsweise Leistungsproblemen oder langen Ladezeiten, welche das Erlebnis des Spielers beeinträchtigen.
Eine effiziente Optimierung dieser Systeme ist daher entscheidend, um eine Menge interessanter Effekte ohne Einschränkungen im Spiel darzustellen. Die Engine Unity, eine beliebte Plattform zur Entwicklung solcher Anwendungen, bietet mehrere Möglichkeiten, um die Leistung von Partikelsystemen zu verbessern. Dabei ist es essenziell, die geeignete Technik für das jeweilige Problem auszuwählen, da die unterschiedlichen Ansätze ver-

schiedene Auswirkungen und Wirksamkeiten aufweisen. Diese Arbeit widmet sich daher dem Vergleich von Optimierungsmethoden für Partikelsysteme in Unity, um festzustellen, welche Konzepte am effektivsten sind. Um Leistungsunterschiede eindeutig den einzelnen Techniken zu zuordnen und zu bewerten, erfolgt die Untersuchung in einem Testszenario, in welchem nur der zu testende visuelle Effekt in der Szene vorhanden ist.
Ziel der Arbeit ist eine Vergleichbarkeit der Optimierungsmethoden in Unity zu schaffen. Dazu beginnt die Untersuchung mit einem umfassenden Literaturüberblick über die bestehenden Techniken in der Engine. Danach werden ausgewählte Ansätze in einer vordefinierten Testszene implementiert. Anschließend erfolgt eine Evaluation, um deren Effektivität und Auswirkungen auf bestimmte Leistungsparameter, wie beispielsweise die Bildrate, Grafikkartenauslastung oder Speichernutzung, festzustellen. Zum Schluss sollen die Ergebnisse bewertet und miteinander verglichen werden.

1.3 Aufbau der Bachelorarbeit

Das erste Kapitel bietet eine Einführung in das Thema und skizziert den Rahmen der Bachelorarbeit. Es umfasst die Einleitung, die Problemstellung, die Motivation sowie Zielsetzung und den Aufbau der Arbeit. Dabei wird die Notwendigkeit der Untersuchung und die Vorgehensweise erläutert. Daraufhin folgt die Literaturrecherche, in der eine Erklärung des Begriffes „visueller Effekt“ zuerst allgemein und darauf im Kontext von Unity erfolgt. Danach findet eine Auflistung und ein Vergleich der Möglichkeiten zur Erstellung von Partikelsystemen in der Engine statt. Die Beschreibung potenzieller Optimierungstechniken für diese Elemente erfolgt zum Schluss des Kapitels. Im nächsten Teil der Bachelorarbeit findet die Implementation des zu testenden visuellen Effektes und einer Auswahl der beschriebenen Methoden statt. Beispiele dafür sind das Level-of-Detail System, Batching, Texturkompression und ein Pooling-System. Deren Umsetzung wird ausführlich beschrieben und zum Schluss erfolgt die Auflistung von aufgetretenen Problemen. Das folgende Kapitel enthält die Evaluation der implementierten Optimierungstechniken. Dafür findet zunächst eine Erläuterung der Komponenten des Testgerätes und der Metriken, auf welche die Methoden geprüft werden sollen, statt. Daraufhin folgt die Auswertung des originalen visuellen Effektes, um eine Vergleichbarkeit der potenziellen Leistungsverbesserungen der Techniken zu ermöglichen. Als nächstes erfolgt die Evaluation der Optimierungsmethoden hinsichtlich der benannten Metriken. Eine zusammenfassende Auswertung und ein Vergleich der Effektivität der Ansätze findet am Ende statt. Das vorletzte Kapitel bietet eine Diskussion der Ergebnisse und fasst Erkenntnisse zusammen. In diesem Teil werden zudem unerwartete Resultate, Limitationen der Arbeit und ein möglicher Nutzen in der praktischen Anwendung in Videospielen geklärt. Das Ende der Bachelorarbeit bildet die Zusammenfassung und der Ausblick, in dem die wichtigsten Ergebnisse und Erkenntnisse sowie Ideen zu möglichen zukünftigen Untersuchen aufgelistet werden.

2 Theoretischer Rahmen

2.1 Grundlagen von visuellen Effekten

In Anbetracht einer fehlenden einheitlichen Definition von visuellen Effekten wird das Konzept als Häufungsbegriff betrachtet, der verschiedene Eingriffe in das filmische Bild umfasst. Dabei bezeichnen visuelle Effekte die digitalen Techniken, die verschiedene Aspekte wie z. B. Modellierung, Animation und Rendern abdecken. [Flückiger, 2022]
Im Kontext von Spielen sind sie Spezialeffekte, welche z.B. bei Charakterfähigkeiten, Wasserfällen und in der Benutzeroberfläche zum Einsatz kommen. Darüber hinaus können sie Beleuchtungen, Flüssigkeitssimulationen und Wetterveränderungen umfassen. Diese Spezialeffekte stellen ein wesentliches Mittel dar, um die visuelle Erfahrung im Spiel zu verbessern. [Zhang & Hu, 2017]
Diese wissenschaftliche Arbeit konzentriert sich ausschließlich auf den spezifischen Aspekt der Partikelsysteme im Kontext von visuellen Effekten.

2.2 Visuelle Effekte in Unity

2.2.1 Allgemeines zur Entwicklung von Partikelsystemen

In den letzten Jahren hat die Spiel-Engine Unity insbesondere bei unabhängigen Spieleentwickler und kleinen Teams an Popularität gewonnen. Ein Vorteil liegt in der Nutzerfreundlichkeit, welche sich unter anderem in der einfachen Programmierung widerspiegelt. [Xie, J., 2012] Auch der Visual Effect Graph, ein Werkzeug zur Entwicklung von Partikelsystemen auf einer knotenbasierten Logik beruhend, ist ein weiteres Beispiel für die gute Handhabbarkeit. [Unity Packages, 2023]
Im Allgemeinem sind unter visuellen Effekten in Unity simulierte, animierte Elemente zu verstehen. Diese werden in eine Szene integriert, um beispielsweise Wasserspritzer oder Feuerexplosionen darzustellen. Innerhalb der Unity-Engine werden visuelle Effekte in die zwei folgenden Kategorien unterteilt. Zum einem gibt es Umwelteffekte, welche physische Prozesse wie z.B. Feuer oder Regen abbilden und zum anderen Gameplay-Effekte. Diese bereichern bestimmte Benutzerinteraktionen oder lenken die Aufmerksamkeit des Spielers auf interessante Bereiche des Spieles. Ein Beispiel dafür ist ein Leuchteffekt beim Aufheben eines Objektes. [Unity Tutorial, 2023]
Visuelle Effekte werden in Unity auch als Partikelsysteme bezeichnet, die viele kleine 2D-Bilder oder Meshes, also 3D-Objekte, simulieren und rendern. Jedes dieser gerenderten Elemente wird als Partikel bezeichnet, das jeweils eine grafische Einheit im endgültigen System repräsentiert. Im Partikelsystem werden dann sämtliche Partikel gleichzeitig simuliert, um den Eindruck eines gesamten zusammenhängenden Effekts zu erzeugen, welcher den fertigen visuellen Effekt ergibt. Durch diese Art der Instanziierung eignen sich Partikelsysteme zur Darstellung von dynamischen Objekten in der Szene, wie z.B. Flüs-

sigkeiten oder Rauch, die ansonsten schwer mit herkömmlichen Meshes oder Bildern umsetzbar wären. [Unity Manual, 2023: Particle systems]
In Unity stehen zur Entwicklung von visuellen Effekten zwei Optionen zur Verfügung. Zum einem gibt es das eingebaute Partikelsystem, dass das standartgemäße System von Unity ist. Zum anderen ist der Visual Effect Graph vorhanden, welcher ein neueres, leistungsstarkes System ist, jedoch eine Lernkurve erfordert, um effektiv genutzt zu werden. Diese Option beruht zudem auf einer knotenbasierten Logik. [Unity Tutorial, 2023]

2.2.2 Eingebautes Partikelsystem

Das eingebaute Partikelsystem von Unity gestattet dem Nutzer vollständigen Lese- und Schreibzugriff auf das System und deren Partikel durch C#-Skripte. Zudem wird eine Partikelsystem-API bereitgestellt, die es ermöglicht, das Verhalten des Effektes anzupassen oder komplett neu zu erstellen. [Unity Manual, 2023: Choosing your particle system solution] Ein Vorteil dieses Partikelsystems besteht darin, Effekte für alle von Unity unterstützten Plattformen zu generieren. Weiterhin wird die Interaktion mit dem System und deren Komponenten über C#-Skripten bereitgestellt. Außerdem ist es mit dem Physiksystem von Unity verbunden, wodurch eine Aktion bei einer Kollision mit anderen Objekten in der Szene möglich ist. Das eingebaute Partikelsystem simuliert dabei das Verhalten der einzelnen Komponenten auf der zentralen Prozessoreinheit. [Unity Manual, 2023: Built-in Particle System] Um die Leistungsfähigkeit bei der Verwendung von z.B. Mesh-Partikeln zu steigern, ist es möglich, die Instanziierung dieser auf die Grafikprozessoreinheit auszulagern. Dafür muss zunächst der Rendermodus des Systems auf „Mesh“ eingestellt werden. Zudem ist die Auswahl einer Plattform für das Spiel sowie eines Shaders für den Effekt, welche die GPU-Berechnung unterstützten, notwendig. Zuletzt erfolgt die Konfiguration der Instanziierung im Rendermodul der Partikelsystems. [Unity Manual, 2023: Particle System GPU Instancing]

2.2.3 Visual Effect Graph

Der Visual Effect Graph ist ein visuelles Skriptingsystem, welches eine umfangreiche Bibliothek von integrierten Operatoren zur Steuerung des Verhaltens von Partikeln bereitstellt. Es ermöglicht die Erstellung von Echtzeit-Verhalten für visuelle Effekte und verwendet GPU-Compute-Shader in Verbindung mit einem nodebasierten Arbeitsablauf. [Zhang, 2020: S.31] Da der Visual Effect Graph auf der Grafikkarte ausgeführt werden kann, ist es möglich, Millionen von Partikel zu simulieren und große, komplexe Effekte zu erstellen. [Unity Manual, 2023: Choosing your particle system solution] Ein Unterschied zum eingebauten Partikelsystem besteht darin, dass das System eine höhere Anzahl an Elementen umfassen kann. Zudem zeichnet sich das Verhalten der Partikel durch eine hohe Anpassbarkeit aus. [Unity Manual, 2023: Visual Effect Graph]

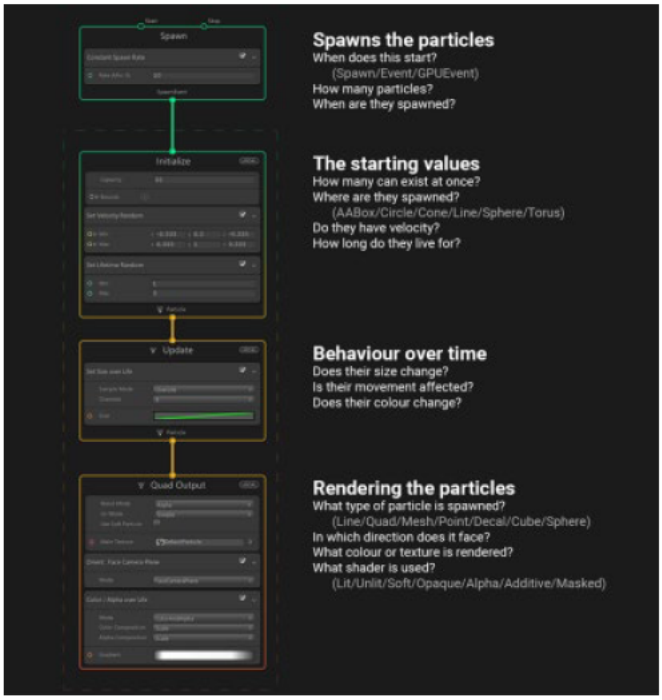

Abbildung 1: Aufbau eine VFX-Graphes
[Zhang, 2020: S.31]

Im Visual Effect Graph gibt es vier Hauptbereiche. Dies sind der in Abbildung 1 zu sehende Spawn-, Initialisierungs-, Update- und Output-Kontext. Im ersten Block erfolgt die Steuerung der Partikelerzeugung mit einschließlich der Anzahl, des Zeitpunktes und einer potenziellen Periodizität. Im Initialisierungs-Kontext werden Anfangswerte für jedes Element festgelegt. Der Update-Block ähnelt der gleichnamigen Methode in Unity Skripten und wird in jeden Frame aufgerufen, um z.B. laufende Eigenschaften der Partikel zu behandeln. Im Output-Kontext erfolgt die Render-Ausgabe und enthält beispielsweise Shader und Texturen. Die Interaktion mit anderen Systemen im Kontext des Visual Effect Graphes erfolgt entweder über das Event-System oder mittels C#-Skripten. [Zhang, 2020: S.31-32]

2.2.4 Vergleich der Varianten zur Entwicklung von Partikelsystemen

Bei den in Unity vorhandenen Methoden zur Erstellung von Partikelsystemen zeigen sich Unterschiede. Der Visual Effect Graph zeichnet sich durch einen stark anpassbaren Entwicklungsprozess in Form eines Graphen aus und ermöglicht die Ausführung auf der GPU, wodurch Millionen von Partikeln darstellbar sind. Der Erstellungsprozess beim eingebauten System findet im Gegensatz dazu mit einfachen Modulen mit vordefiniertem Verhalten für die Elemente statt. Dabei ist die Anzeige von lediglich tausenden Partikeln möglich, was geringer als bei dem VFX-Graph ist. Weiterhin erlaubt das eingebaute System die Interaktion mit dem zugrundeliegenden Physiksystem von Unity sowie die Wiedergabesteuerung und Anpassung der Partikel zur Laufzeit über C#-Skripte. Im Visual Effect Graph kann die Interaktion mit bestimmten Elementen wie beispielsweise dem Depth-Buffer zur Anwendung von Physik genutzt werden. Darüber hinaus ermöglichen C#-Skripte eine Wiedergabensteuerung und Veränderung von Elementen unter der Ver-

wendung von freigegebenen Eigenschaften sowie die Verwendung von benutzerdefinierten Ereignissen. In der High-Definition Render Pipeline ist im Visual Effect Graph zudem ein Farb- und Tiefenpuffer vorhanden. Ähnliches besitzt das eingebaute Partikelsystem nicht. [Unity Manual, 2023: Choosing your particle system solution] Ein weiterer Vorteil des VFX-Graphes besteht darin, dass ein Echtzeitverhalten für jeden Partikeln berechnet werden kann und die Kommunikation mit anderen Unity-Systemen möglich ist. [Zhang, 2020: S.32]

2.3 Optimierungstechniken für Partikelsysteme in Unity

2.3.1 Billboard

Billboards stellen besondere Transformationsgruppen dar, welche sich automatisch auf die Kamera in der Szene ausrichten und somit auf den Spieler fokussieren. Sie zeichnen sich oft durch ihre Einfachheit aus und enthalten in der Regel texturierte Vierecke. Dies bietet eine Leistungsoptimierung im Vergleich zur Darstellung von vollständigen 3D-Objekten. Daher ist es effizienter beispielsweise ein Bild eines Baumes mit transparentem Hintergrund anstatt eines Meshes zu verwenden. Ein Nachteil von Billboards besteht darin, dass der Spieler das Objekt stehts nur von einer Seite betrachten kann. Aus diesem Grund finden Billboards meist nur bei weit entfernten oder sehr kleinen Elementen in der Szene Verwendung. Sie werden oft für einzelne Partikel in visuellen Effekten eingesetzt. [Jung, 2019]
In der Unity-Engine sind Billboards als 2D-Texturen implementiert, die sich immer zu der Kamera in der Szene ausrichten. [Unity Manual, 2023: Glossary] Sie kommen oft in Level-of-Detail Systemen zum Einsatz um komplexe 3D-Modelle, welche weit entfernt von der Kamera sind, darzustellen. Gerendert werden Billboards Assets in Unity mithilfe des Billboard Renderers. [Unity Manual, 2023: Billboard Renderer component]

2.3.2 Level-of-Detail System

Level-of-Detail ist eine Optimierungsmethode, bei der die Präzision von Objektmodellen flexibel anhand der Entfernung zur Kamera gesteuert wird. Da das in Unity von der Kamera repräsentierte menschliche Auge weit entfernte Objekte weniger detailliert wahrnimmt, erfolgt die Reduzierung von Details entsprechend der Entfernung des jeweiligen Elements zum Betrachter. Dadurch rendern weniger komplexe Objekte in der Szene gleichzeitig, wodurch die Anwendung flüssiger läuft und weniger Geräteressourcen verbraucht werden. [Yan et al., 2023]

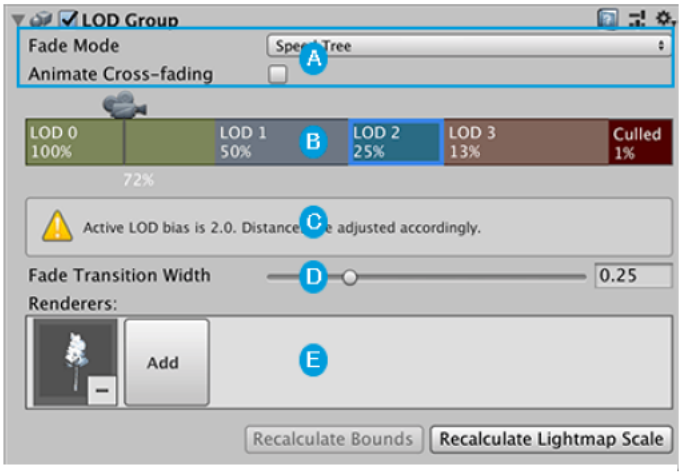

Abbildung 2: LOD Group Komponente
[Unity Manual, 2023: LOD Group]

Die Verwendung der LOD Group-Komponente, welche in Abbildung 2 dargestellt ist, bietet die Möglichkeit der Festlegung von Level-of-Detail für das jeweilige Objekt. Dies ermöglicht unteranderem die Einstellung der Anzahl an Ebenen und die Distanz beim Wechsel zwischen ihnen. [Aleksi, 2018: S. 24-25] Der Schwellenwert für diesen Übergang basiert auf dem Verhältnis der Größe des Elements auf dem Monitor zur Gesamtbildschirmhöhe. Der in Abbildung 2 mit dem Buchstaben „A“ gekennzeichneter Bereich kontrolliert diesen Wechsel zwischen den verschiedenen Leveln. Der sogenannten „Fade Mode“ bietet zur Auswahl „Cross Fade“, „Speed Tree“ oder keinem Übergangsmodus. Im Bereich „B“ des Bildes kann zwischen den einzelnen LOD-Leveln gewechselt und die Schwellenwerte beim Wechsel zwischen ihnen angegeben werden. Zudem wird eine Vorschau der gerenderten Ebene angezeigt. Die Nachricht in Abschnitt „C“ enthält Informationen über den LOD-Bias, falls dieser nicht auf eins festgelegt ist. Im Bereich „D“ der Abbildung werden die entsprechenden Renderer für jede Ebene gesetzt. Darüber hinaus befinden sich am unteren Rand des Bildes zwei Schaltflächen. Zum einem berechnet „Recalculate Bounds“ das Begrenzungsvolumen aller LOD-Gameobjekte neu. Zum anderen aktualisiert „Recalculate Lightmap Scale“ die Skalierung aller Elemente in den Leveln des Systems in der sogenannten „Lightmap“. [Unity Manual, 2023: LOD Group]

2.3.3 Culling

Die Culling-Technik ist in Unity-3D eine der effektivsten Optimierungsmethoden, da es die Anzahl der zu rendernden Objekte in der Szene reduziert, wodurch weniger CPU- und GPU-Ressourcen beansprucht werden. In der gewählten Engine sind drei Methoden von Sichtbarkeits-Culling verfügbar. Zum einem vermeidet das Back-Face Culling das Rendern von Geometrie, welche vom Betrachter abgewandt ist und daher außerhalb der Sichtfeldes liegt. Des Weiteren verhindert das in Abbildung 3 dargestellte Occlusion Culling das Elemente, die von anderen Teilen der Szene verdeckt sind, abgebildet werden. Die letzte in Abbildung 4 zusehende Methode des Fustum Cullings unterbindet das Rendern von Objekten, die außerhalb des Sichtfeldes der Kamera liegen. [Minh et al., 2022] Dieser Vorgang ist von Unity automatisiert, immer aktiv und erfordert keine Einrichtung

des Entwicklers. Das Occlusion Culling reduziert die Anzahl der Draw Calls für Objekte in Sichtweite. [Aleksi, 2018: S. 21-22] Dafür generiert die Methode Szenendaten im Editor und verwendet diese zur Laufzeit, um festzustellen, was von der Kamera aufgenommen wird. Dieser Prozess wird als „Baking“ bezeichnet. Dabei unterteilt Unity die Szene in Zellen und beschreibt die darin enthaltenden Geometrien, sowie die Sichtbarkeit zwischen benachbarten Zellen. Die Konfiguration dieses Prozesses erfolgt durch die Einstellung von Parameter im entsprechenden Fenster und die Verwendung von Occlusion Areas. Während der Laufzeit werden die gebackten Daten in den Speicher geladen und für jede entsprechend gekennzeichnete Kamera Abfragen mit ihnen durchgeführt, um festzustellen, was die Kamera aufzeichnet. [Unity Manual, 2023: Occlusion culling] Das Back-Face Culling wird vorwiegend bei Meshes mit einer Vielzahl von Triangeln angewendet, weshalb es in der Implementierung der Optimierungsmethoden für Partikelsysteme nicht zum Einsatz kam.

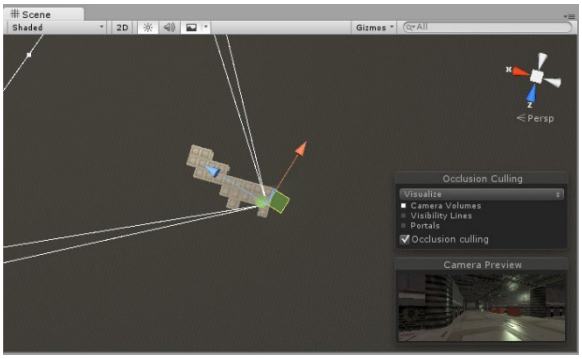

Abbildung 3: Occlusion Culling
[Unity Manual, 2023: Occlusion culling]

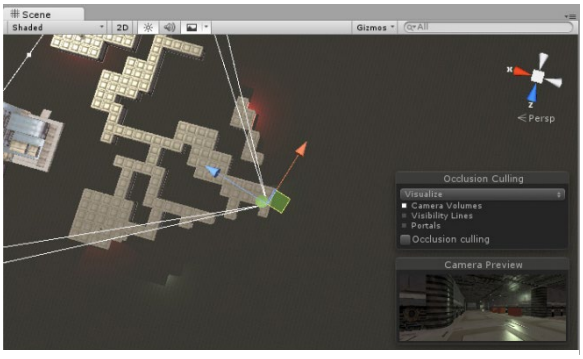

Abbildung 4: Fustum Culling
[Unity Manual, 2023: Occlusion culling]

2.3.4 Batching

Beim Batching handelt es sich um eine Optimierungsmethode, bei der die Anzahl der sogenannten „Draw Calls“ reduziert wird. Dies sind in Unity Anweisungen an die Grafik-API die angeben, wie bestimmte Elemente in der Szene gerendert werden sollen. Draw Calls finden somit Verwendung, um Geometrie auf den Bildschirm abzubilden und enthalten Informationen zu Texturen, Shadern und Buffern. Zudem können sie ressourcenintensiv sein. [Unity Manual, 2023: Optimizing draw calls]
Eine Variante zur Reduzierung der Arbeitslast der Grafikkarte in GPU gebundenen Projekten besteht darin, die Anzahl der Draw Calls zu minimieren. Neben der Entfernung von Objekten in der Szene kann dies durch Batching geschehen. Bei dieser Optimierungsmethode müssen die betroffenen Elemente dasselbe Material verwenden und im Editor als statisch markiert werden, was bedeutet, dass die Objekte sich nicht im Worldspace bewegen. Dies ermöglicht die Verwendung von Batching und somit das rendern mehrere Objekte in einem einzigen Draw Call. [Aleksi, 2018: S. 21]
Unity bietet zwei eingebaute Methoden für diese Optimierungsmethode. Zum einem werden unbewegliche Objekte durch das sogenannte „statische Batching“ kombiniert und

zusammen abgebildet. Zum anderem transformiert Unity die Eckpunkte kleiner Meshes im sogenannten „dynamische Batching“ auf der CPU. Daraufhin findet die Gruppierung ähnliche Ecken und deren Rendering in einem Draw Call statt. Das eingebaute Batching-System von Unity hat den Vorteil, dass es trotz der Zusammenführung von Meshes noch einzelne Objekte cullt. Nachteile sind jedoch, dass die statische Variante einen Speicherplatzoverhead verursacht und die dynamische Optimierung CPU-Ressourcen in Anspruch nimmt. [Unity Manual, 2023: Draw call batching]
Für Partikelsysteme wird die letztere Optimierungsmethode verwendet. Für die in diesem Fall dynamisch generierten Geometrien erstellt Unity für jeden Renderer einen großen Vertex-Buffer, welcher alle dynamischen, batchbaren Inhalte enthält. Der Renderer richtet dann den Materialzustand für das Batching ein und die Engine bindet den Vertex-Buffer an die GPU. Für jeden Renderer in der Batching-Gruppe aktualisiert Unity den Offset im Vertex-Buffer und sendet einen neuen Draw Call. [Unity Manual, 2023: Dynamic batching]
Mithilfe des von der Engine bereitgestellten Frame Debuggers kann überprüft werden, wann die Methode angewendet wurde und warum ein Draw Call nicht im vorherigen Batch mit gerendert wurde. [Aleksi, 2018: S. 21]

2.3.5 Texturkomprimierung

Da die GPU zur Laufzeit andere, spezialisierte Formate verwendet, welche für den Speicherverbrauch und die Geschwindigkeit von Abtastoperationen optimiert sind, führt Unity automatisch Texturkompression durch. Um die Speicherreduzierung zu optimieren, ist das Texturkompressionsformat für die verschiedenen Plattformen anpassbar. Der Wert Bits pro Pixel gibt an, wie viel Platz für einen Texturpixel benötigt wird. Bilder mit einem niedrigen bpp-Wert beanspruchen eine geringere Größe auf der Festplatte und die GPU erfordert eine geringe Speicherbandbreite zum Lesen der Pixel. Allerdings nimmt die visuelle Qualität mit sinkenden Bits pro Pixel ab. Die von Unity bereitgestellten Texturkompressionsformate sind alle in gewissem Grad verlustbehaftet und bieten daher unterschiedliche Abwägungen zwischen der Qualität und Größe der Datei. Eine weitere Möglichkeit zur Reduzierung der Bildgröße ist die Verwendung des Kompressionsformates Crunch. Dieses basiert auf DXT-Kompression oder der ETC-Methode und bietet zusätzlich eine variable Bitratenkompression. Beim Laden einer Crunch-Textur wird diese von der CPU zu einem DXT- oder ETC-Format dekomprimiert und auf die GPU geladen. Die Kompressionsmethode trägt dazu bei, Texturspeicherplatz auf der Festplatte zu minimieren, hat jedoch keine Auswirkungen auf den Speicherplatzverbrauch zur Laufzeit. [Unity Manual, 2023: Texture formats]

2.3.6 Pooling System

Das sogenannten „Pooling“ ist eine Methode zur Speicheroptimierung, welche dazu dient, die Häufigkeit der dynamischen Speicherzuweisung und -freigabe zu reduzieren. Dafür wird der voraussichtlich benötigte Speicher vorher allokiert und eine vordefinierte Anzahl an wiederverwendbaren Elementen initialisiert, was den Objektpool darstellt. Methoden können diese Objekte bei Bedarf abrufen und zurückgeben, anstatt sie wiederholt zu erstellen und zu zerstören.
Nachteile bei der Methode sind, dass die Freigabe des allokierten Speichers für eine längere Zeitspanne nicht möglich ist und der Pool die gleichzeitig verwendbaren Objekte aufnehmen können muss. Dies führt möglicherweise dazu, dass eine Reservierung von viel Speicherplatz erfolgt, auch wenn ein erheblicher Teil davon nicht benötigt wird. Das hängt von der Art der Verwendung der Elemente im Objektpool ab.
In Unity steht für die Implementierung dieser Optimierungsmethode eine eigene integrierte generische Objektpool-Schnittstelle zur Verfügung. [Ionin, 2023: S. 15] Diese verwendet einen Stapel, um die Objektinstanzen für die Wiederverwendung zu verwalten. Durch die Vermeidung von wiederholten Erstellungs- und Zerstörungsaufrufen wird zudem die CPU entlastet. [Unity Script, 2023: ObjectPool<T0>]

2.3.7 Compute Shader

Compute Shader sind spezielle Shader-Programme, die auf der Grafikprozessoreinheit ausgeführt werden. Sie sind in der High-Level Shadingsprache im Stil von DirectX11 verfasst. Diese Shader operieren außerhalb der herkömmlichen Renderpipeline und finden bei hochgradig parallelen GPGPU-Algorithmen oder der Beschleunigung einiger Teile des Renderns eines Spiels Anwendung. [Unity Manual, 2023: Compute shaders] GPGPU steht für General Purpose Computation on Graphics Processing Unit und erklärt die Nutzung der Rechenleistung der Grafikprozessoren für nicht grafische Aufgaben. [Mücke, 2007: S. 27]
Die Anwendung von Compute Shadern erfordert ein tiefes Wissen über die GPU-Architektur, parallele Algorithmen und Kenntnisse in verschiedenen Programmierschnittstellen wie DirectCompute, OpenGL Compute, CUDA oder OpenCL. In der Entwicklungsumgebung Unity weisen sie Ähnlichkeiten mit der DirectX11 DirectCompute-Technologie auf. [Unity Manual, 2023: Compute shaders]

3 Implementierung von ausgewählten Optimierungstechniken in Unity

3.1 Rahmen der Implementierung

3.1.1 Komponenten des Testgerätes

Das zur Implementierung der verschiedenen Techniken und zur Durchführung der Tests verwendete Unity Projekt wurde auf einem handelsüblichen Desktop-Computer ausgeführt. Dieses Gerät verfügt über die folgenden Hardware-Spezifikationen. Zum einem besitzt es eine NVIDIA GeForce GTX 1070 Grafikkarte und einen AMD Ryzen 3 1200 Quad-Core Prozessor. Zudem ist ein Arbeitsspeicher von 16 GB DDR4 RAM mit einer Geschwindigkeit von 2,6 GHz verbaut.

3.1.2 Aufbau der Testumgebung und des zu testenden Partikelsystems

Die Durchführung und Umsetzung der verschiedenen Techniken erfolgte innerhalb eines Unity 3D Projektes, welches auf der Universal Renderpipeline basiert. Weiterhin wurde das Visual Effect Graph Package integriert, um eine Optimierung über den VFX-Graphen zu ermöglichen. Im nächsten Schritt wurde die Testszene aufgesetzt. Sie enthält eine Kamera, die MainKamera, ein direktionales Licht und den zu prüfenden visuellen Effekt, welcher in Abbildung 5 dargestellt ist. Dieser besteht aus fünf individuellen Systemen, die mithilfe des eingebauten Partikelsystems von Unity erstellt wurden.

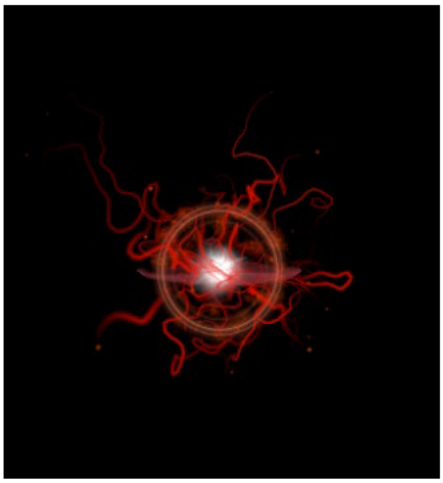

Abbildung 5: originaler visueller Effekt

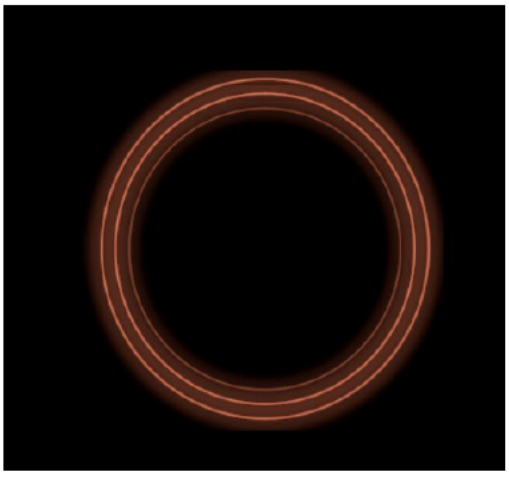

Abbildung 6: Kreis-Partikelsystem

Abbildung 7: Mesh-Partikelsystem

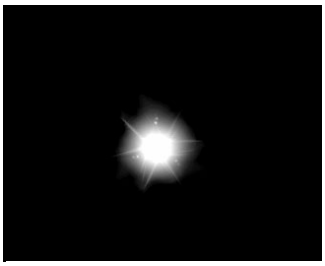

Abbildung 8: Lichtpunkt-Partikelsystem

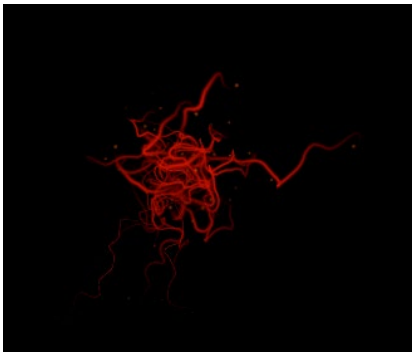

Abbildung 9: Partikel mit Trail Partikelsystem

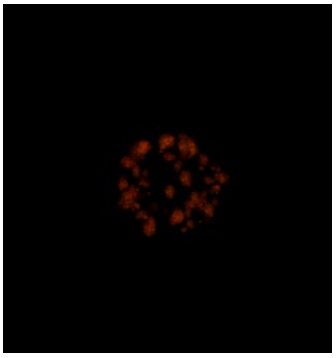

Abbildung 10: Rauch-Partikelsystem

Zum einem veranschaulicht Abbildung 6 einen Effekt mit bis zu maximal drei Kreisen, welche als Billboards rendern. Sie expandieren von der Mitte des Systems nach außen, wobei ihre Größe über die Lebensdauer von vier Sekunden zunimmt. Zudem loopt der Effekt, wodurch kontinuierlich aller zwei Sekunden die Erzeugung neuer Partikel stattfindet. Dieses Element ist relativ simpel aufgebaut, wie in den Attributen in Abbildung 56 dargestellt, um den Einfluss der Optimierungstechniken auf standardmäßiges System zu verdeutlichen.

Abbildung 7 zeigt ein Effekt mit einem Mesh in Form eines nicht vollständig geschlossenen Kreises. Dieses besitzt die in Abbildung 55 sichtbaren Attribute. Aufgrund der Dreidimensionalität des Objektes hat es eine zufällige 3D-Startrotation sowie 3D-Startgröße und erzeugt in einem Schleifendurchlauf bis zu zwei Partikel. Während ihrer Lebenszeit erhöhen diese ihre Größe und rotieren entsprechend ihres Tempos. Der in Abbildung 59 dargestellte Rendermodus wurde auf „Mesh“ gestellt, um den Einfluss der Optimierungstechniken auf ein dreidimensionales Partikelsystem zu testen.

In der Mitte des Effektes befindet sich, wie in der Abbildung 8 dargestellt, ein Lichtpartikelsystem. Bei diesem wird pro Schleifendurchlauf ein Partikel mit einer randomisierten Rotation erzeugt. Außerdem ist die gleichzeitige Darstellung von insgesamt 16 Partikel möglich. Diese drehen sich und verändern ihre Größe in Abhängigkeit zu ihrer Lebenszeit, wie in der Abbildung 60 abgebildet, um einen blinkenden Effekt zu erzeugen. Der Rendermodus dieses Partikelsystems ist auf Billboard eingestellt. Das Element soll den Einfluss eines weiteren über die Lebenszeit veränderten Attributes auf die Leistung demonstrieren.

Abbildung 9 zeigt den nächsten Teileffekt. Er besteht aus kleinen einfachen Billboard-

Partikeln, welche einen Trail aufweisen. In diesem System werden, wie in Abbildung 58 sichtbar, pro Schleifendurchlauf 15 Partikel in einer Kugel mit einem Radius von einer Einheit um die Mitte des visuellen Effektes erzeugt. Dadurch werden im Durchschnitt 55 und maximal 1 000 Partikel gleichzeitig angezeigt. Sie besitzen eine zufällige Bewegungsrichtung in Abhängigkeit zu ihrer Lebenszeit. Zusätzlich wird, wie in Abbildung 61 dargestellt, ein Rauschen auf ihre Bewegung angewendet, um das verworrene Erscheinungsbild des gesamten Effektes darzustellen und den Einfluss dieser Einstellung auf die Leistung testen zu können. Der Trail besteht aus mehreren Partikeln, hat eine Lebensdauer von einer Sekunde und rendert, wie in Abbildung 57 sichtbar, separat mit einem eigenen Material. Durch die Einbindung dieses Elements soll der Einfluss der Optimierungstechniken darauf untersuchbar sein. Außerdem ist im Gegensatz zu den anderen Teilsystemen die Erstellungsrate im Vergleich zu der maximalen Kapazität relativ gering, weshalb das Element mehr Zeit benötigt, seine vollständige Menge an Partikeln zu erreichen. Dadurch ist testbar, welchen Unterschied diese Eigenschaft bei den Optimierungsmethoden hat.
Im letzten Effekt sind Rauchpartikel vorhanden, welche in Abbildung 10 zu sehen sind und als Billboards rendern. Wie in Abbildung 63 dargestellt, können maximal bis zu 100 Partikel im System gleichzeitig vorhanden sein. Des Weiteren werden pro Schleifendurchlauf 50 Partikel mit randomisierter Größe, Rotation, Lebenszeit und Bewegungsrichtung in einer Kugel um den Mittelpunkt erzeugt. Über ihre Lebensspanne hinweg nimmt, wie in Abbildung 62 sichtbar, ihre Größe zu und sie rotieren sich. Dieser Teileffekt dient dazu, die Effektivität der Optimierungstechniken anhand eines relativ unkomplizierten Effektes, dessen Elemente in einer Kugel um den Mittelpunkt erstellt werden, zu testen.
Alle Partikel in den einzelnen Systemen besitzen einen anfänglichen Alpha Wert von Null, welcher sich im Verlauf ihrer Lebenszeit zuerst erhöht und am Ende abnimmt, um ein sanftes Verblassen darzustellen. Für den originalen Effekt wurden vier verschiedenen Texturen verwendet. Zum einem der in Abbildung 13 zusehende Kreis, der in Abbildung 11 gezeigte Lichtpunkt, der Lichtstrahl in Abbildung 12 und der Rauch in Abbildung 14.

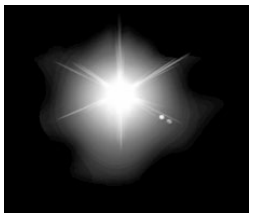

Abbildung 11: Lichtpunkt-Textur

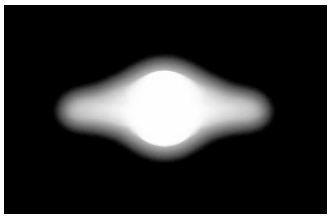

Abbildung 12: Lichtstrahl-Textur

Abbildung 13: Kreis-Textur

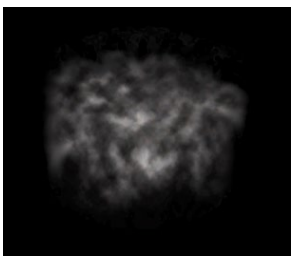

Abbildung 14: Rauch-Textur

3.2 Implementierung der Optimierungstechniken

3.2.1 Einsatz von ausschließlich Billboards

Um den visuellen Effekt ausschließlich mit Billboards darzustellen, wurden Änderungen am halbrunden Mesh-Partikelsystem vorgenommen. Dafür ist eine Abfolge von PNG‘s in einem Bild, ein sogenanntes „Sprite Sheet“, entstanden. Um dies als Animation in Unity flüssig abzuspielen, sind in Abbildung 15 zehn halbrunde Kreise abgebildet, welche die Rotation des Meshes nachbilden.

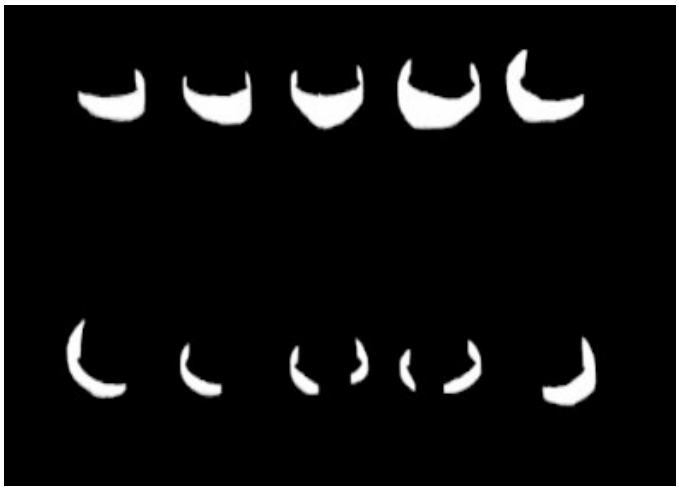

Abbildung 15: Sprite Sheet zur Nachbildung des Mesh-Partikelsystems

Zur Verwendung dieses Bildes in dem Mesh-Partikelsystem wurde der Rendermodus auf „Billboard“ gesetzt und ein Material, dass das PNG enthält, hinzugefügt. Des Weiteren fand die Entfernung der Rotation in Relation zur Geschwindigkeit, der Startrotation und 3D-Größe statt. Um die einzelnen Zeichnungen in der Abbildung nacheinander abzuspielen, wurde eine sogenannte „Texture Sheet Animation“ hinzugefügt. Dabei erfolgte die Festlegung der Tiles auf fünf mal zwei und die des Time Modes auf Lebenszeit. Hierbei entstand das in Abbildung 17 dargestellte Partikelsystem und der in Abbildung 16 zu sehende insgesamte visuelle Effekt.

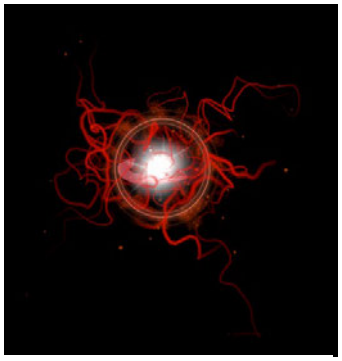

Abbildung 16: Effekt mit ausschließlich Billboards

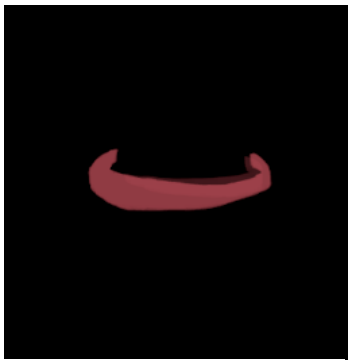

Abbildung 17: Nachbil-dung des Mesh-Partikelsystems

3.2.2 Einbindung eines Level-of-Detail Systems

Um ein Level-of-Detail System für den zu testenden visuellen Effekt zu implementieren, wurde zunächst ein leeres Objekt mit einer von der Unity-Engine bereitgestellten LOD-Group in der Testszene erzeugt, welche in Abbildung 18 abgebildet ist. Diese Komponente besitzt keinen Fad-Modus, da im Kontext von visuellen Effekten für jedes LOD-Level ein neues Partikelsystem konfiguriert werden muss und somit kein nahtloser Übergang zwischen den verschiedenen Stadien des Elements möglich ist. Insgesamt wurden zudem fünf unterschiedliche LOD-Level definiert und in das System integriert. Dies sollte die Darstellungsqualität und Leistung des visuellen Effektes optimieren und ermöglicht es, je nach Entfernung und Sichtbarkeit des Effektes automatisch zwischen den einzelnen Detailstufen zu wechseln. So sollen die Ressourcen effizient genutzt und trotzdem eine reibungslose Spielererfahrung gewährleistet werden.

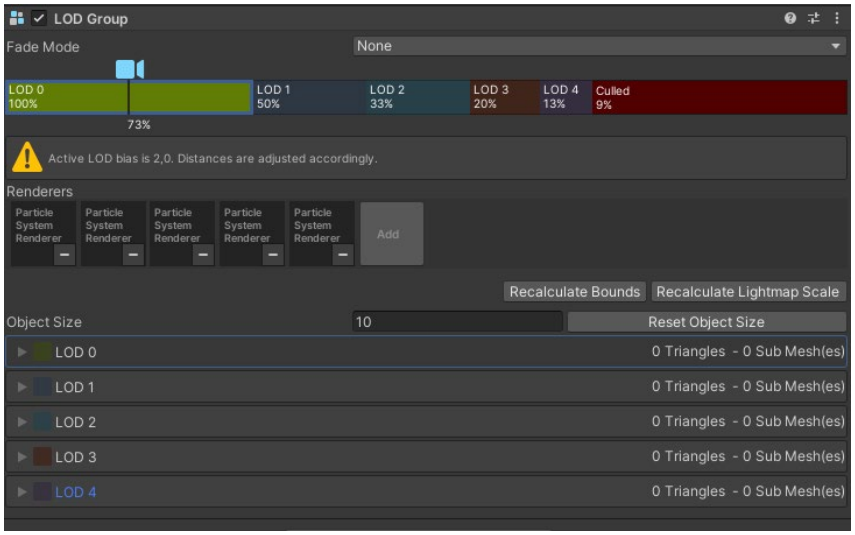

Abbildung 18: LOD-Group in der Testszene

Für die Implementierung wurden zudem fünf Instanzen des originalen Partikelsystems als untergeordnete Objekte der LOD-Group platziert und entsprechend ihrer Stufen benannt, um sie in den jeweiligen Level der LOD-Group einzusetzen. Als nächstes erfolgte die Anpassung der einzelnen LOD-Level Effekte.
Auf Stufe null, welche die höchste Detailstufe repräsentiert und so das größte sichtbare Partikelsystem darstellt, blieb das Original unverändert. Bei Level eins wurde, wie in Abbildung 20 zusehen, die Anzahl der erzeugten Partikel mit Trails und der Rauchpartikel reduziert. Es werden daher nur noch jeweils zehn größere Rauchpartikel und Partikel mit Trail pro Durchlauf erzeugt. Dies führt zu einer Verringerung der gleichzeitig vorhandenen Effektbestandteile und somit zu einer leichten Leistungsverbesserung. Wie in Abbildung 19 zu erkennen, wurden auf Stufe zwei die Rauchpartikel vollständig entfernt und die Anzahl der zu erzeugenden Partikel mit Trail weiter auf fünf pro Durchlauf reduziert. Diese entfallen beim LOD-Level drei vollständig. Der visuelle Effekt dieses Levels ist in Abbildung 22 zusehen. Bei Stufe vier wurde zudem das rechenintensivere Mesh-

Partikelsystem entfernt, sodass wie in Abbildung 21 sichtbar nur noch die Kreise und der zentrale Lichtpunkt zu sehen sind. Im letzten Level verschwindet das Gameobjekt gänzlich.
Zur korrekten Einstellung des LOD-Systems wurde zunächst die Funktion „Recalculate Bounds“ aktiviert, wodurch die Prozentzahlen der einzelnen Stufen sich neu konfigurierten. Diese repräsentieren die Schwelle, ab der das jeweilige Level aktiv wird, basierend auf dem Verhältnis der Größe des Elements auf dem Monitor zur Gesamtbildschirmhöhe. [Unity Manual, 2023: LOD Group] Nach weiteren Anpassungen ist Level null, wie in Abbildung 18 dargestellt, von 100% bis 50% zusehen. Ab dem niedrigeren Prozentsatz wird LOD eins bis 33% aktiv. Anschließend ist das Partikelsystem in Level zwei sichtbar, bis es nur noch 20% der gesamten Bildschirmhöhe einnimmt. LOD drei wird bis 13% angezeigt, gefolgt von Stufe vier bis 9%. Danach verschwindet der Effekt vollständig.

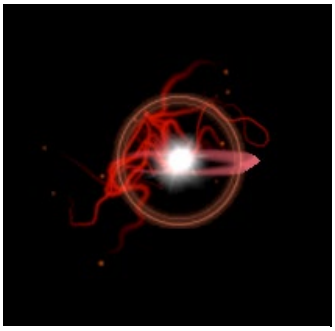

Abbildung 19:
LOD-Level2

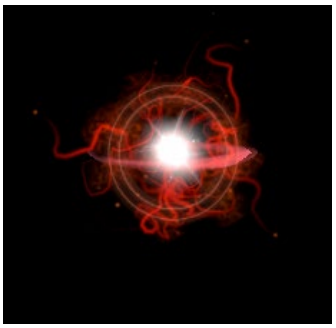

Abbildung 20:
LOD-Level1

Abbildung 21:
LOD-Level4

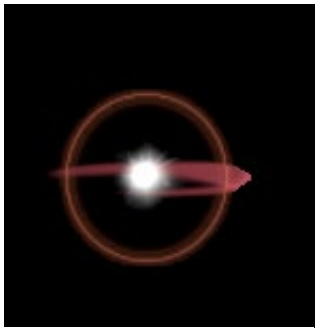

Abbildung 22:
LOD-Level3

Bei der Implementierung des Level-of-Detail Systems wurde darauf geachtet, dass die Veränderungen des Effekts zwischen den einzelnen Leveln so unscheinbar wie möglich sind. Dies geschah um einen nahtlosen Übergang zwischen den Leveln zu gewährleisten und dem Spieler den Eindruck eines gesamten, sich nicht ändernden Systems zu vermitteln.

3.2.3 Implementation eines Culling Systems

Um die Optimierungsmethode eines Culling Systems in Unity zu implementieren, wurden zunächst die Bestandteile des visuellen Effektes auf den statischen Modus gesetzt. Zusätzlich wurde der Culling-Modus in den jeweiligen Partikelsystem auf „Pause“ gestellt. Dadurch werden diese angehalten, falls sie nicht im Fokus der Kamera liegen und erst fortgesetzt, wenn sie wieder im Sichtfeld des Spielers erscheinen. Diese Funktion basiert auf einer Berechnung der Kamera, welche standardmäßig Objekte nicht rendert, die sich außerhalb ihres Betrachtungswinkels befinden. Dies trägt zur Verbesserung der Leistung des Spiels bei.
Allerdings wird der visuelle Effekt trotz dieser Einstellungen abgespielt, wenn er sich hinter einem Objekt befindet, welches im Blickwinkel der Kamera liegt. Der Spieler kann das System in diesem Fall jedoch nicht wahrnehmen. Um auch in diesem Szenario die Leistungsfähigkeit zu steigern, wurde das von Unity bereitgestellte Occlusion Culling eingebunden. Dies verhindert, dass die Durchführung von Renderberechnungen für Elemente stattfindet, die vollständig von anderen Objekten verdeckt sind. [Unity Manual, 2023: Occlusion culling] Die Konfiguration im Occlusion Tab umfasst zunächst die Festlegung der minimalen Größe eines Objektes, hinter dem keine Rendervorgänge durchgeführt werden sollen, wobei dieser Wert bei fünf Metern liegt. Weiterhin fand die Einstellung statt, ab welchem Durchmesser eines Loches das dahinter liegende Objekt noch rendert. Dies liegt bei 0,25 Metern. Als nächsten Schritt wurden aus Gründen der Berechnungsoptimierung diese Einstellungen zu einer Gruppe von Entitäten und Komponenten für eine bessere Leistung gebündelt und verarbeitet. Als Ergebnis dieses Vorganges entstand in der Szene eine Occlusion Area, welche in Abbildung 23 gelb gekennzeichnet ist. Sie definiert View Volumes, also Bereiche in der Szene in denen sich die Kamera zur Laufzeit im Occlusion Culling System befindet. Wenn sie sich während des Spiels in diesem Gebiet aufhält, führt Unity präzisere Berechnungen durch. [Unity Manual, 2023: Occlusion Areas] Die in Abbildung 23 dargestellten weißen Linien repräsentieren die aktuelle Sicht der Kamera. Wie ersichtlich ist, reicht diese nur bis zu dem quadratischen Objekt. Alles, was sich dahinter befindet, liegt nicht in der Sicht des Spielers.

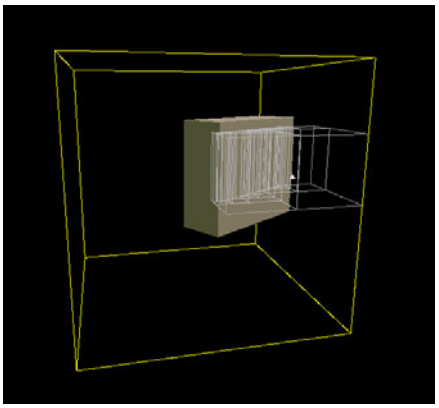

Abbildung 23: Occlusion Area

Zum Schluss des Verfahrens wurde das Occlusion Culling System einem Funktionalitätstest unterzogen. In Abbildung 24 ist auf der linken Seite die Szenen- und rechts die Game-Ansicht zu sehen. Es ist ersichtlich, dass der aktive visuelle Effekt nicht rendert, da er von dem Objekt verdeckt wird, welches sich vor der Kamera befindet. In Abbildung 25 fand die Verschiebung der Kamera in Richtung der X-Achse statt, wodurch das Partikelsystem zum Vorschein kam. Die weißen Linien in Abbildung 25 verdeutlichen, dass die Kamera nun auch einen Teil der Rückseite des Quadrates erfasst und somit der visuelle Effekt im Sichtfeld liegt und gerendert wird. Somit ist das Occlusion Culling System funktionstüchtig.

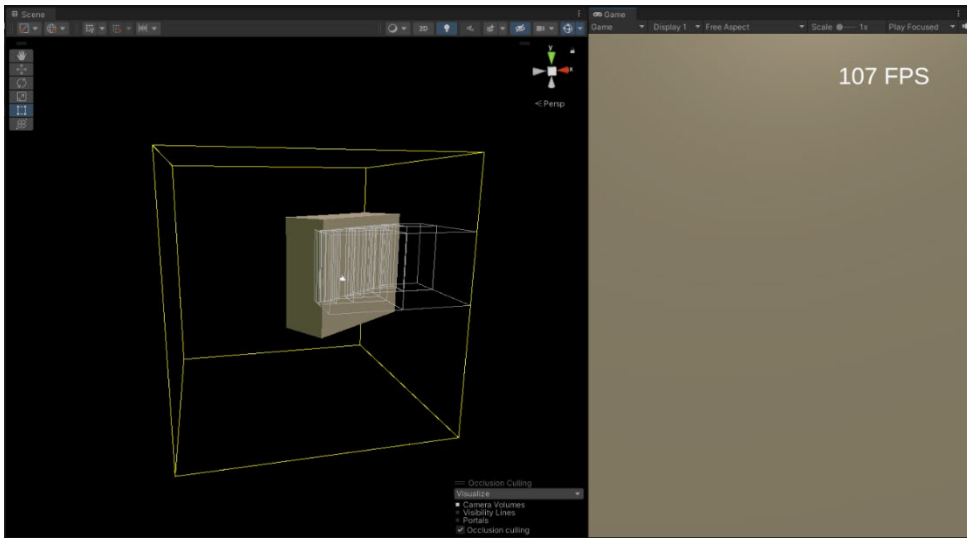

Abbildung 24: Occlusion Area mit blockierter Sicht

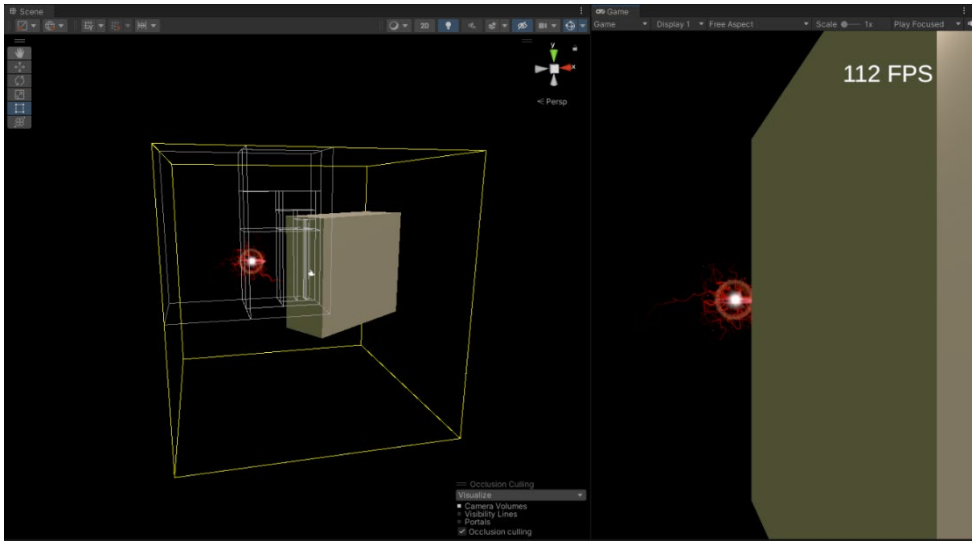

Abbildung 25: Occlusion Area mit freier Sicht

3.2.4 Durchführung von Batching

Um die Anzahl der Draw Calls, welche an die Grafik-API gesendet werden, zu reduzieren, wurde die Technik des Batching angewandt. Dies erfolgte durch die Minimierung der Menge an verwendeten Texturen und Materialen, da dies oft der Grund ist, warum Unity kein Batching durchführt. Zu diesem Zweck fand die Zusammenfügung der Texturen des Lichtpunktes, des Lichtstrahles, des Kreises und des Rauches in einer einzigen 4096 x 4096 Pixel großen Textur statt, welche in Abbildung 27 dargestellt ist. Es erfolgte ihre Zuweisung zu einem Material mit dem Shader Mobile/Particles/Additive, um sie in den einzelnen Partikelsystem zu verwenden.
Nachdem die ursprünglichen Materialen durch das Neue ersetzt wurden, fand in jedem Partikelsystem die Anpassung der Texture Sheet Animation statt, um das richtige Bild im Material auszuwählen. Dazu erfolgte die Einstellung des Modus auf Grid und der Tiles auf zwei mal zwei, da im Bild zwei untereinanderliegende Texturen in jeweils zwei Spalten vorhanden sind. Zusätzlich bekam Frame over Time einen konstanten Wert und eine Anpassung entsprechend des benötigten Bildes. Ein Beispiel dafür ist vom Kreis-Partikelsystem in Abbildung 26 zusehen. Dessen benötigte Textur ist im PNG rechts unten zu finden. Beginnend von Null und links oben zählend, handelt es sich um die dritte Textur im erstellten Bild, weshalb Frame over Time den Wert drei erhält.
Durch diese Anpassungen werden nun die Draw Calls der Rauch-, Kreis-, Lichtpunkt- und kleinen Partikelpartikelsystemen gebatcht. Bei den Trails kann das Material jedoch nicht geändert werden und sie werden zudem anders gerendert als die anderen Teileffekte. Die Erzeugung der Mesh-Partikeln erfolgt ebenfalls auf eine andere Weise, wodurch kein Batching mit den anderen Systemen stattfinden konnte.

Abbildung 26: Texture Sheet Animation Teil

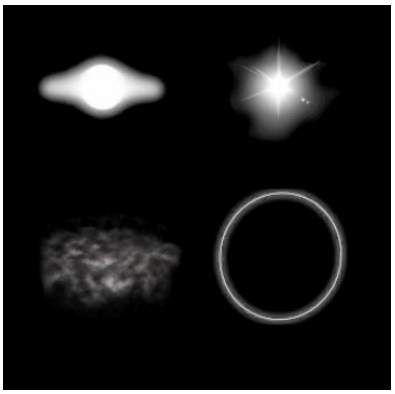

Abbildung 27: Batching-Textur

3.2.5 Umsetzung als Visual Effect Graph

Eine zusätzliche Möglichkeit zur Umsetzung des originalen visuellen Effektes besteht darin, anstelle der Verwendung des eingebauten Partikelsystems den Visual Effect Graph zu benutzen. Zu diesem Zweck wurde mithilfe des entsprechenden in Unity vorher integrierten Package ein leerer Graph im Asset Ordner erstellt. Insgesamt fand die Erzeugung von fünf Graphen in diesem Visual Effect Graph statt, welche ähnlich zu den einzelnen Partikelsystemen aufgebaut sind.
In Abbildung 65 ist der Graph der Kreise dargestellt. Diese erhalten bei der Initialisierung eine Lebensdauer von vier Sekunden und eine Größe von fünf Einheiten, welche zur Laufzeit erhöht wird. Im Ausgabeabschnitt ist der Blend-Modus auf Alpha eingestellt. Zudem fand dieselbe Textur wie im nachgeahmten Partikelsystem Anwendung und die Partikel im Effekt orientieren sich immer zur Kamera. Beim zweiten System, welches in Abbildung 64 zu sehen ist, wird das Kreis-Mesh erzeugt. Die Partikel darin werden mit einer Lebensdauer von vier Sekunden, einer Größe von drei Einheiten und einer 90 Grad Rotation in X-Richtung erzeugt. Im Update-Teil des Systems dreht sich jeder Partikel kontinuierlich um 0,18 Grad pro Geschwindigkeit. Als letztes wurden im Output Particle Mesh das zuvor schon für das nachgeahmte System erstellte Mesh und die Textur verwendet, um den gleichen Effekt zu erzeugen. Da dies ein 3D-Modell erstellt, fand keine Verwendung der Node Orient to Camera statt. Als nächstes ist das System des Lichtpunktes in Abbildung 67 dargestellt. Dessen Partikel erhalten zur Erzeugung eine Lebensdauer von fünf Sekunden, eine zufällige Größe zwischen zwei bis fünf Einheiten und eine randomisierte Rotation. Weil dieses System als Billboard rendert, verwendet es die entsprechende Lichtpunkt Textur aus dem dazugehörigen Partikelsystem und die Orientierung wird auf Face Camera Plane gesetzt. Die Erzeugung des blinkenden Effektes der Partikel erfolgt durch das Drehen und der Veränderung der Größe über die Lebenszeit hinweg. Als vorletzten Graphen ist die Erstellung der Rauchpartikel in Abbildung 66 zusehen. Bei ihrer Erzeugung erhalten sie eine randomisierte Lebensdauer von eins bis vier Sekunden sowie eine zufällige Farbe und Größe zwischen 0,15 bis 1,2 Einheiten. Ihre Anfangsposition wird auf zwei Einheiten um den Systemmittelpunkt gesetzt und ihre Bewegungsrichtung zufällig in alle Richtungen festgelegt. Die Partikel rendern ebenfalls als Billboards und besitzen daher eine Orientierung zu Kamera. Sie erhöhen ihre Größe und rotieren während der Lebenszeit. Der letzte Graph in diesem Effekt, welcher in Abbildung 68 dargestellt ist, erstellt die Partikel mit Trail. Diese erhalten bei ihrer Erzeugung eine randomisierte Lebensdauer zwischen zwei bis fünf Sekunden und eine zufällige Rotation und Größe zwischen 0,1 und 0,3 Einheiten. Ihre Position wird auf eine Einheit um den Systemmittelpunkt gestellt und ihre Bewegungsrichtung zufällig festgelegt. Die Partikel selbst rendern ebenfalls als Billboards und erhalten eine steigende Geschwindigkeit über ihre Lebenszeit hinweg. Im Updateteil erfolgt die Anwendung einer Turbulenz auf ihre Bewegung. Dies dient dazu, das Rauschen des entsprechenden Partikelsystems nachzuahmen und erzeugt die chaotische Bewegung, die im originalen Effekt zu beobachten ist. Darüber hinaus werden in diesem Abschnitt die Trails mithilfe der Node Trigger Event Rate (Over

Time) und eines GPU-Events erzeugt. Dies ermöglicht eine Verknüpfung mit dem jeweiligen erstellten Partikel und das Auslesen von Daten aus ihnen. Daher übernehmen die Trails ihre Position und Größe. Bei ihrer Initialisierung erhalten sie jedoch eine individuelle rote Farbe und Lebenszeit von fünf Sekunden. Sie werden als ParticleStrip Quad gerendert und orientieren sich immer zur Kamera. Während der Lebenszeit erfolgt zudem eine kontinuierliche Verkleinerung ihrer Größe.
Um sicherzustellen, dass diese Graphen im Visual Effect Graph genauso wie die Partikelsystem aussehen wurden Änderungen in der Ausgabereihenfolge des Renderelements vorgenommen. In Abbildung 28 ist zusehen, dass zuerst die Rauchpartikel, danach die Trails und die jeweiligen Partikel sowie anschließend der Lichtpunkt rendert. Daraufhin wird das halbrunde Mesh und zum Schluss die Kreise abgebildet. Durch diese Anpassungen erscheinen in der Szene die zuletzt gerenderten Partikel optisch vor den vorher gerenderten. In Abbildung 29 ist der fertig nachgebildete Effekt abgebildet. Er weist eine starke Ähnlichkeit zum originalen Effekt auf. Allerdings besitzen die Trails eine leichte abgehackte Erscheinung, da die Bewegungsrichtung der dazugehörigen Partikel nicht durch eine Noise-Funktion, sondern durch die Turbulenz-Node beeinflusst werden.

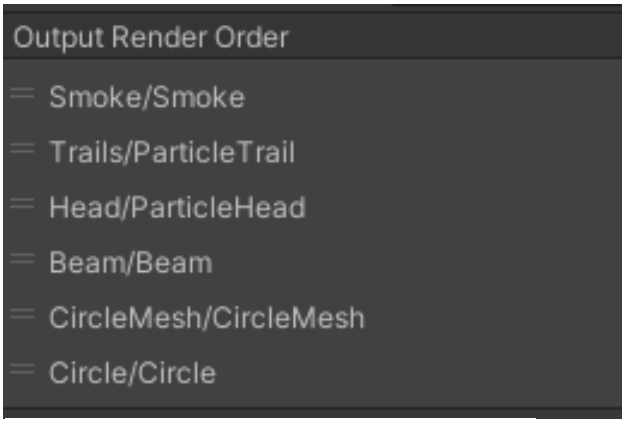

Abbildung 28: Renderreihenfolge des Visual Effect Graphes

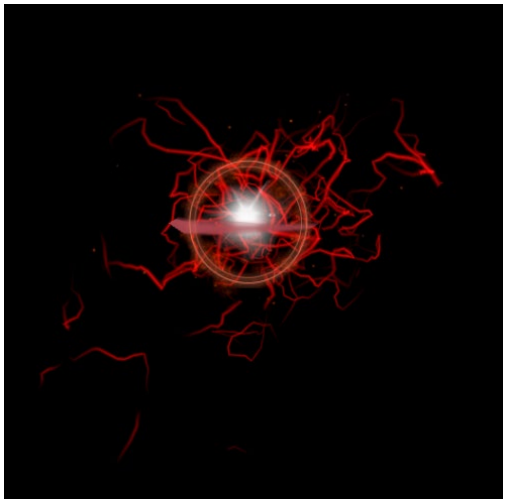

Abbildung 29: Effekt mit dem Visual Effect Graph

3.2.6 Kompression von Texturen

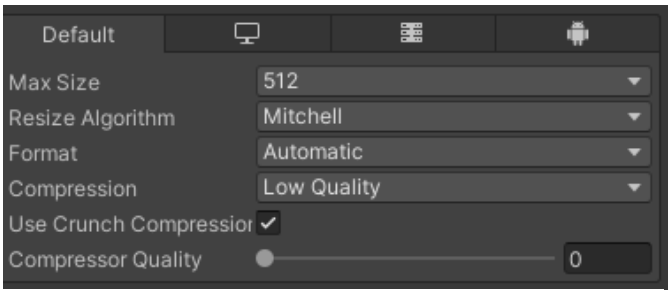

Abbildung 30: eigene Textur-Einstellungen

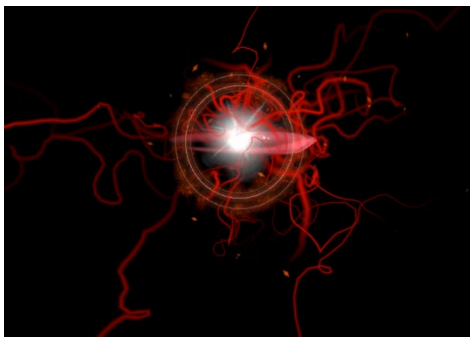

Abbildung 31: visueller Effekt mit komprimierten Texturen

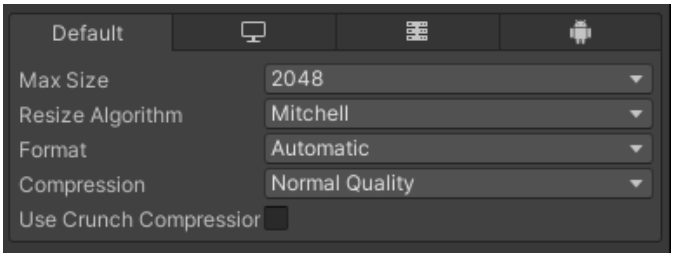

Abbildung 32: standardmäßige Textur-Einstellungen

Zur Implementierung dieser Optimierungsmethode wurden die verwendeten Texturen mithilfe der in Unity vorhanden Technik komprimiert. In Abbildung 32 sind die Standardwerte dargestellt, welche auch für die Texturen des originalen Effektes zum Einsatz kamen. Dabei komprimiert Unity die PNGs in normaler Qualität ohne Crunch zu verwenden. Dies entspricht auf der Windows-Plattform dem Texturformat RGB Compressed DXT1. [Unity Manual, 2023: Recommended, default, and supported texture formats, by platform] Für die Optimierungstechnik fand die Festlegung in Abbildung 30 dargestellten Einstellungen in den Importoptionen statt. Die maximale Größe wurde von 2 048 auf 512 Pixel reduziert und die Kompressionsart auf niedrig eingestellt. Zudem fand die Aktivierung der Crunch-Kompression statt, welche eine Qualität von null erhielt. Das entspricht auf der Windows-Plattform dem Texturformat RGBA Crunched DXT5. [Unity Manual, 2023: Recommended, default, and supported texture formats, by platform]
Dies führte zu einer deutlichen Verringerung der Bildqualität, wie Abbildung 31 für den gesamten Effekt dargestellt. Trotz der visuellen Verluste war es wichtig, eine möglichst starke Komprimierung vorzunehmen, um einen signifikanten Effekt bei der Evaluation zu ermöglichen, selbst wenn bereits im originalen System eine von Unity voreingestellte Komprimierung stattfand.

3.2.7 Einbindung eines Pooling Systems

Um die Instanziierung mehrerer Instanzen des visuellen Effektes effizienter zu gestalten, wurde ein Pooling System implementiert. Dieses generiert einmalig eine Vielzahl von Exemplaren des Systems und schaltet sie je nach Bedarf in den aktiven bzw. passiven Zustand. Dadurch wird die Berechnung des wiederholten Instanziierens und Löschens des gleichen Effektes vermieden. Die Zuweisung eines neuen Objekt-Pools vom Typ „GameObject“ zu der Klassenvariable erfolgt, wie in Abbildung 33 zu sehen ist, in der Start-Methode des Skripts „PoolingSystem“. Dieses Element verfügt für seine Funktionalität über vier Methoden und drei Werte. Die Variable „collectionCkeck“ überprüft, ob ein zurückgegebenes Objekt schon im Pool vorhanden ist, um so Fehlermeldungen zu verhindern. Die „defaultCapacity“ gibt den zu erwartenden Wert an, wie viele Instanzen des Objektes benötigt werden. Diese Kapazität kann bei Bedarf auf den Wert von „maxSize“ erhöht werden, falls mehr Elemente benötigt werden als ursprünglich angenommen.

```
public class PoolingSystem : MonoBehaviour
{
    public ObjectPool<GameObject> _pool;
    [SerializeField] private GameObject original;  Pooling
    public Vector3 spawnPosition;  Serializable
     Event function
    private void Start()
    {
        _pool = new ObjectPool<GameObject>(createParticleSystem, actionOnGet: takeParticelSystemFromPool,
            actionOnRelease: returnParticelSystemToPool, destroyParticelSystemInPool, collectionCheck: true,
            defaultCapacity: 200, maxSize: 300);
    }
```

Abbildung 33: Klassenvariablen und Start-Methode des Pooling-Systems

Eine Methode des Pooling-Systems, wie in Abbildung 34 gezeigt, erstellt neue Instanzen des zuvor im Unity-Inspector zugewiesenen Gameobjektes und weist ihnen die Position in der Variable „spawnPosition“ zu.

```
    private GameObject createParticleSystem()
    {
        //new instance off the particle System
        GameObject particleSystem = Instantiate(original, spawnPosition, transform.rotation);
        return particleSystem;
    }
```

Abbildung 34: Erstellungs-Funktion des Pooling-Systems

Die nächste Methode, welche in Abbildung 35 zu sehen ist, wird aufgerufen, wenn die Aktivierung eines Partikelsystems aus dem Pool erfolgen soll. Dabei werden zunächst die Position und Rotation auf die Anfangswerte zurückgesetzt, um sicherzustellen, dass das wiederverwendete Objekt auch bei der Reaktivierung wie neu erscheint. Nach diesen Einstellungen wird das Partikelsystem auf aktiv gesetzt und ist sichtbar.

```
private void takeParticelSystemFromPool(GameObject particleSystem)
{
    //reset transform + rotation
    particleSystem.transform.position = spawnPosition;
    particleSystem.transform.rotation = transform.rotation;

    //activate particelSystem
    particleSystem.gameObject.SetActive(true);
}
```

Abbildung 35: Aktivierungs-Methode des Pooling-Systems

Die Verwendung der oberen Methode in Abbildung 36 erfolgt, wenn ein visueller Effekt in der Szene nicht mehr sichtbar sein soll und somit die Zurückgabe des deaktivierten Elements in den Objekt-Pool erfolgt. Die andere Methode in Abbildung 36 wird genutzt, um überschüssige Instanzen des Gameobjekts, die die „maxSize“ überschreiten, dauerhaft zu löschen, sobald deren Rückgabe an den Pool erfolgt.

```
private void returnParticelSystemToPool(GameObject particleSystem)
{
    particleSystem.gameObject.SetActive(false);
}
1 usage
private void destroyParticelSystemInPool(GameObject particleSystem)
{
    Destroy(particleSystem.gameObject);
}
```

Abbildung 36: Deaktivierungs- und Lösch-Funktion des Pooling-Systems

Zur Überprüfung der Funktionalität und für die spätere Evaluation wird die in Abbildung 70 gezeigte Coroutine alle sieben Sekunden ausgeführt. Dies ist eine Methode, welche ihre Ausführung unterbrechen und in einem anderen Frame an der gleichen Stelle fortführen kann. [Unity Manual, 2023: Coroutines] Dabei werden 100 Elemente mit unterschiedlichen Positionen erstellt. Durch die Erhöhung der X- und Z-Position entsteht ein Gitternetz aus den visuellen Effekten. Des Weiteren erfolgt die Weitergabe der Position der Systeme an die „spawnPosition“ Variable im Pooling Skript. Wenn die Effekte mit dem Pooling-System angezeigt werden sollen, findet deren Aktivierung mit der Get()-Methode statt. Erfolgt die Evaluation ohne die Optimierungsmethode, werden die Elemente instanziiert.

In Abbildung 69 ist das Skript zusehen, welches auf dem Prefab des visuellen Effektes liegt. Bei der Aktivierung der Elemente startet eine Coroutine, die nach zehn Sekunden das System zum Objekt-Pool zurückgibt und somit deaktiviert. Ohne das Pooling-System wird das jeweilige Element stattdessen zerstört.

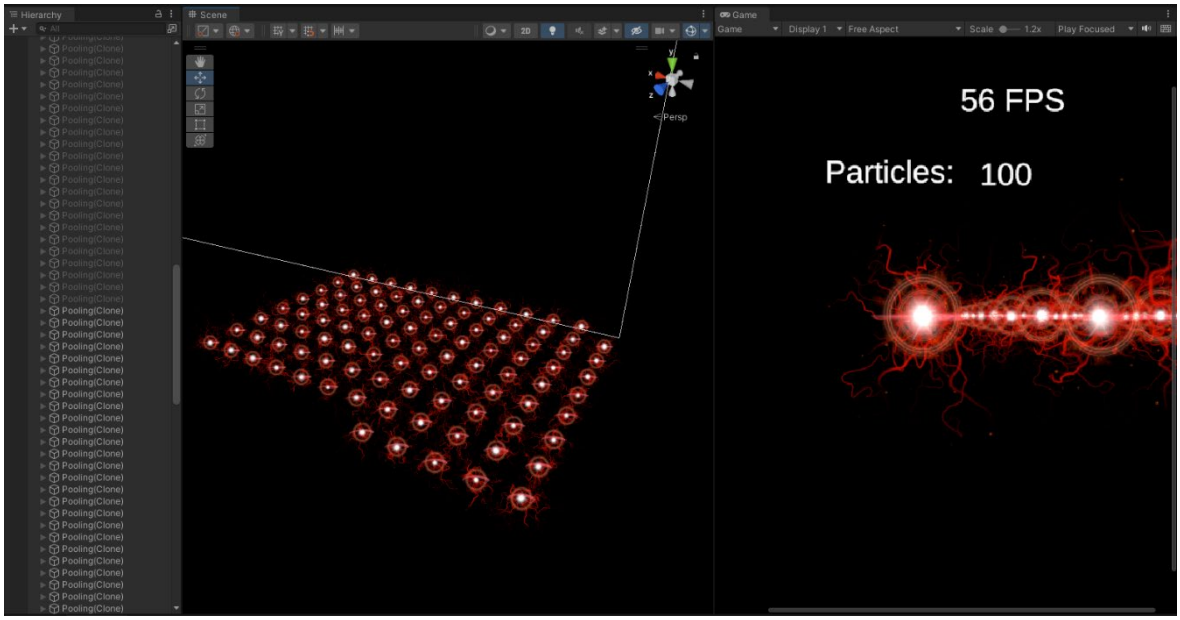

Abbildung 37: Test des Pooling-Systems

In Abbildung 37 ist ein Test zur Überprüfung der Funktionalität des Pooling-Systems dargestellt. In der Game-Ansicht ist die aktuelle Partikelanzahl und am linken Rand des Bildes die in der Szene enthaltenen Gameobjekte sichtbar. Obwohl 200 visuelle Effekte initialisiert wurden, sind in diesem Ausschnitt nur die Hälfte davon aktiv, da zurzeit nur 100 Systeme benötigt werden. Die Aktivierung von den inaktiven Objekten erfolgt, wenn die Verwendung von mehr Elementen notwendig ist.

3.3 Aufgetretene Probleme bei der Implementierung

Ein Problem bei der Implementation der Komprimierungstechnik bestand darin, dass Unity standardmäßig bereits die Texturen im originalen Effekt komprimierte. Dies führte dazu, dass anfänglich keine Veränderungen in Bezug auf die Speichergröße oder die Auslastung der GPU bei der Anwendung einer leicht stärkeren Methode erkennbar waren. Um dennoch einen Effekt bei diese Optimierungsvariante zu erzielen, war es erforderlich, zusätzliche Verluste bei der visuellen Qualität durch die Verwendung von der Crunch-Operation zu akzeptieren.
Für das Pooling-System war es notwendig, zunächst eine sinnvolle Initialisierungsmenge und -art für die visuellen Effekte zu ermitteln, um die Funktionsweise der Technik und die einzelnen Systeme deutlich herauszustellen. Zu diesem Zweck wurden mehrere Erstellungsintervalle getestet, bis dieser Wert auf sieben Sekunden festgelegt wurde.

4 Evaluation

4.1 Beschreibung der Testwerkzeuge

Da der schon integrierte Unity Profiler als Hauptwerkzeug zur Untersuchung von Projekten in der Engine fungiert, kam dieser zum Einsatz, um die CPU- und GPU-Auslastung zu überwachen. Außerdem wurden damit verschiedene Render-Informationen wie z.B. die Anzahl an Draw Calls, Batches, Dreiecken oder Eckpunkten erfasst. Er ermöglicht zudem das Auslesen der oben genannten Metriken für jeden Frame. Des Weiteren stellt der Profiler detaillierte Informationen darüber zur Verfügung, wie viel Zeit die CPU oder GPU für bestimmte Operationen aufwenden.
Um präzisere Informationen über die Verwendung der Draw Calls zu erhalten, wurde der bereits in Unity integrierte Frame Debugger eingesetzt. Dieser gewährt Einblicke in die Renderprozesse und ermöglicht unteranderem die Identifizierung von gebatchten Draw Calls.
Um eine genauere Analyse der Speichernutzung durchführen zu können, wurde zusätzlich der Memory Profiler verwendet. Dies ist ein externes Unity-Projekt, welches eine Integration in die Engine benötigte. Es liefert detailliertere Informationen zur Speicherauslastung und erstellt davon Momentaufnahmen. Darüber hinaus erfolgt eine Aufteilung des Speichers nach Objekttypen, wodurch die Auswertung der Daten erleichtert wird. [Aleksi, 2018: S. 14]

4.2 Verwendete Metriken

Die verschiedenen Optimierungstechniken sowie das originale Partikelsystem wurden anhand folgender Metriken untersucht. Zum einem wurde, wie in Abbildung 72 zusehen, die Frames per Second des gesamten Spiels unter Verwendung eines simplen C#-Scripts und einer Anzeige in der Benutzeroberfläche erfasst. Zum anderen erfolgte die Analyse der CPU- und GPU-Auslastung im Profiler von Unity. Die Ergebnisse wurden ebenfalls in FPS angegeben und zeigen, welche Zeitspanne die jeweilige Einheit für die einzelnen Rechenprozesse benötigte. Des Weiteren wurde die Anzahl der Draw Calls und Batches für jede der angewandten Techniken mithilfe des Profilers und Frame Debuggers untersucht. Hierbei ist ebenfalls ermittelbar, aus welchen Gründen kein Batching erfolgte. Draw Calls werden in Unity als Zeichenaufrufe definiert und enthalten Informationen darüber, wie die Darstellung der einzelnen Elemente geschehen soll. Sie umfassen beispielsweise Details zu Texturen, Shadern oder Buffern. Diese Aufrufe können ressourcenintensiv sein. [Unity Manual, 2023: Optimizing draw calls] Aus diesem Grund wird unter anderem Batching eingesetzt, um Draw Calls zusammenzufassen und somit ihre Effizienz zu optimieren. [Unity Manual, 2023: Draw call batching] Diese Kombinationen werden von Unity erfasst. Mithilfe des Memory Profiler wurde der Speicherbedarf des Spiels festgehalten und insbesondere auf 2D-Texturen und Partikelsysteme hin untersucht, da diese beiden Komponenten für die Untersuchung von visuellen Effekten besonders von Bedeutung sind.

Darüber hinaus fand die Dokumentation eventueller Auffälligkeiten oder Besonderheiten, welche während des Tests festgestellt wurden, statt. Dies umfasst beispielsweise die Eckpunkt- und Dreieck-Anzahl.

4.3 Leistungsvergleich der implementierten Techniken

4.3.1 Auswertung des originalen Partikelsystems

Bei der Leistungsauswertung des originalen Partikelsystems wurde eine durchschnittliche Bildrate von 60 FPS festgestellt. In Abbildung 71 ist ersichtlich, dass 92,5% der CPU-Auslastung auf die Methode „DXGI.WaitOnSwapChain“ entfielen. In dieser Funktion wartet der Prozessor auf die Beendigung der GPU Renderprozesse für den aktuellen Frame.

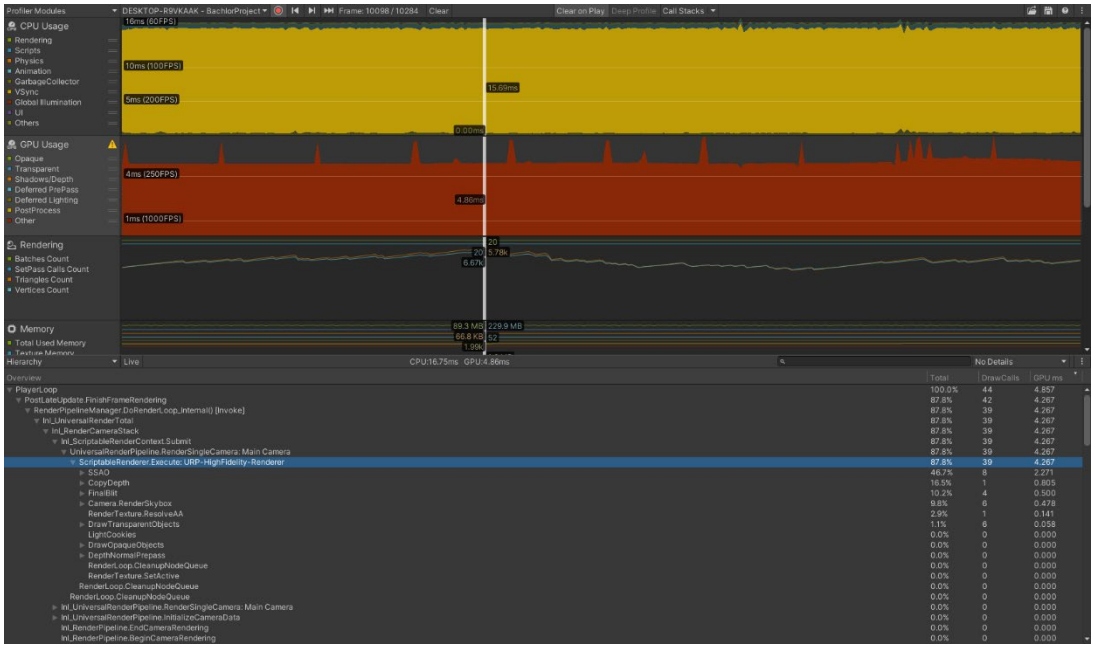

Abbildung 38: GPU-Auslastung des originalen visuellen Effektes

Wie in Abbildung 38 sichtbar ist, wurde 87,8% der gesamten Berechnungszeit der GPU von der Methode „ScriptableRenderer.Execute: URP-HighFidelity-Renderer“ in Anspruch genommen. In diesem Schritt führte die GPU die Universal Render Pipeline für jeden Frame aus, wobei ein bedeutender Teil von 46,7%. für die Operation „SSAO“ verwendet wurde. Diese Methode berechnete die Tiefen- und Normaltextur der Kamera. Das Rendern des visuellen Effektes erfolgte in der Funktion „DrawTransparentObjects“ und lastete die GPU zu 1,1% aus. Die Darstellung des Systems erforderte sechs Draw Calls, da fünf verschiedene Partikelsysteme und der Trail mit jeweils unterschiedlichen Materialien abgebildet werden mussten. Die gesamte GPU-Berechnung pro Frame dauerte durchschnittlich ca. 5 ms. Die Analyse des Rendering im Profiler, welche in Abbildung 74 dargestellt ist, zeigte, dass keine Gruppierung stattfand, da die Anzahl der Batches der der Draw Calls gleicht. Es erfolgte zudem lediglich ein dynamisches Batching, da der visuelle

Effekt als dynamisches Objekt zusammengefasst wird. Jedoch fand bei dem originalen System keine Reduzierung der Batches statt. Weiterhin ist zusehen, dass ungefähr 5 800 Dreiecke und 6 700 Eckpunkte pro Frame gerendert wurden, jedoch ihre Anzahl bei jedem Frame etwas schwankt und unter anderem von der aktuellen Partikelzahl im visuellen Effekt abhing.
Im Frame Debugger fand eine detailliertere Auflistung aller Renderschritte pro Frame inklusiver der Draw Calls statt. So ist in Abbildung 73 zusehen, dass sechs Draw Calls für transparente Objekte ausgeführt wurden, welche den visuellen Effekt darstellten. Im ersten Draw Call wurde das Material mit der PointLight-Textur gerendert, das unter anderem im halbrunden Mesh-Partikelsystem Anwendung fand. Im zweiten Aufruf war das Material der Trails zusehen, das nicht mit der vorherigen Operation batchbar war, da diese unterschiedliche Materialien verwendeten. Im dritten Draw Call wurde erneut ein auch in den kleinen Partikeln vorhandenes Material mit der PointLight-Textur gerendert. Im darauffolgenden Aufruf fand die Abbildung des Materials des Lichtpunkt-Systems statt. Darauf folgte die Verarbeitung des Materials der Kreispartikel und zuletzt der Draw Call des Rauches. Die Ausführung all dieser Renderschritte erfolgte mit dem Event „Draw Dynamic“. Es fand zudem kein Batching statt, da die Komponenten des Effektes unterschiedliche Materialien verwendeten.
Im Memory Profiler von Unity erfolgte die Aufzeichnung von 467 MB der 630 MB des Speichers. Davon wurden, wie in Abbildung 76 zusehen, insgesamt 349,5 MB für den Speicherbedarf und 7,8 MB für Grafik und Grafiktreiber verwendet. In Abbildung 39 ist der verwendete Speicherplatz als Baumstruktur dargestellt. Zu sehen ist, dass die Render Texturen den meisten Speicherplatz beanspruchten, gefolgt von den 2D-Texturen, welche eine Größe von 6,3 MB besaßen und die benötigten Texturen der Partikelsysteme enthielten. Am unteren Rand von Abbildung 39 sind die Texturen Beam, Circle, Smoke und PointLight aufgelistet, wobei die beiden ersten mit 1,3 MB den meisten Speicherbedarf hatten. Darüber hinaus wurde erkenntlich, dass die PointLight-Textur jeweils im Material vom Mesh-Partikelsystem und den kleinen Partikeln Anwendung fand und daher zwei Referenzen aufwies. Beim Hineinzoomen in das Diagramm ist der Speicherplatzbedarf der Partikelsysteme zu sehen gewesen. Abbildung 75 stellt dies dar und zeigt die Auflistung der fünf Elemente. Diese besaßen eine Gesamtgröße von 275,4 KB, wobei das System der kleinen Partikel mit Trail davon den größten Anteil mit 212,1 KB hatte.

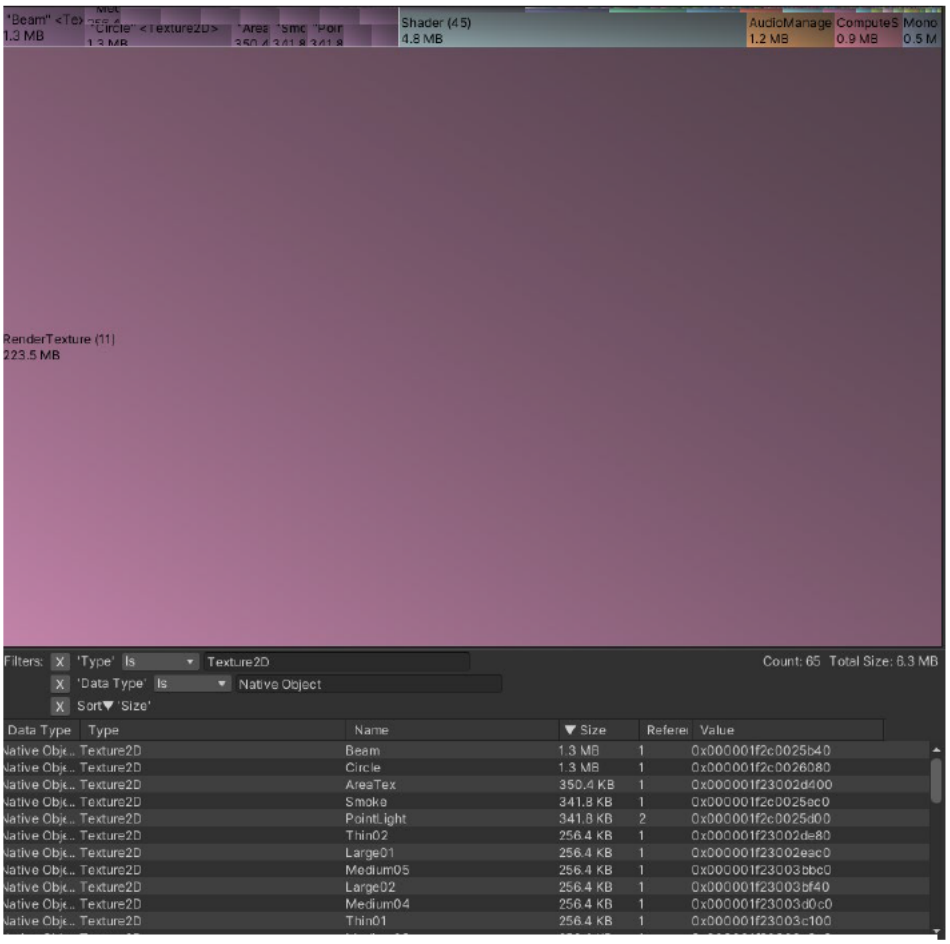

Abbildung 39: 2D-Texturen in der Baumstruktur des Memory Profilers des originalen visuellen Effektes

4.3.2 Vergleich mit ausschließlich Billboards im Partikelsystem

Im Verlauf der Evaluierung dieser Optimierungsmethode wurde eine Bildrate von 60 FPS ermittelt. Die durchschnittliche Auslastung der CPU belief sich auf 16 ms pro Frame, wobei das Warten auf die Beendigung der Operationen der Grafikkarte am meisten der Zeit benötigte.

Die GPU-Auslastung betrug im Durchschnitt etwa 3,7 ms pro Frame, was eine um 1,3 ms schnellere Durchführung im Vergleich zum originalen Effekt darstellte. In Abbildung 40 ist zu erkennen, dass die Grafikkarte 76,9% dieser Zeit für die Ausführung des URP-HighFidelity-Renderers und 23% für die Methode „Graphics.PresentAndSync“ verwendete. Diese Operation stellte die Wartezeit während der Durchführung des Hauptthreads „Present“ dar. Im Rendering-Teil des Unity-Profilers, welcher in Abbildung 77 abgebildet ist, fand keine Veränderung bei den Draw Calls und dynamischen Batches im Vergleich zum originalen Effekt statt. Allerdings reduzierte sich die Anzahl der Dreiecke auf 4 100 und die der Eckpunkte auf 4 500, da keine Mesh-Partikel mehr gerendert wurden. Diese wiesen eine höhere Dreieck- und Eckpunktzahl als die anderen Systeme auf.

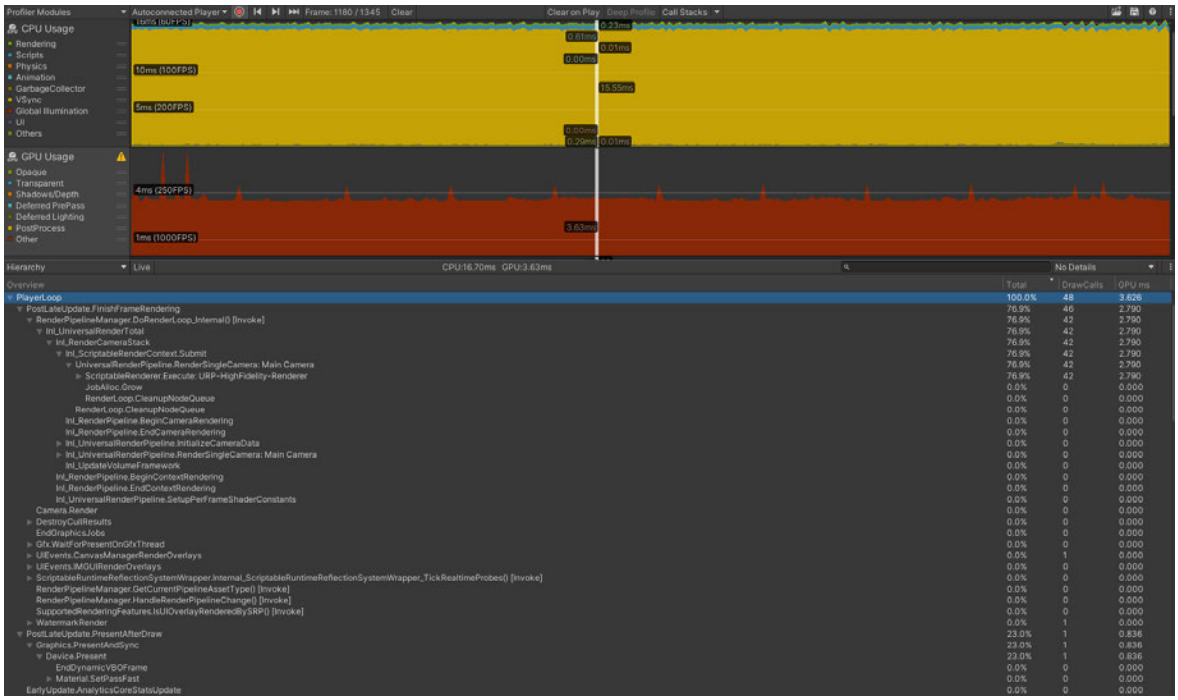

Abbildung 40: GPU-Auslastung bei der Verwendung von ausschließlich Billboards

Im Frame Debugger war zu erkennen, dass es sechs Draw Calls für transparente Objekte gab. Jedoch wurde, wie in Abbildung 78 dargestellt, der Grafikkartenaufruf des Mesh-Partikels durch den der hinzugefügten Textur namens HalfCircleBillboard ersetzt. Die in Abbildung 41 erkennbare Baumstruktur des Memory Profilers zeigt eine Zunahme des Speicherplatzverbrauches der 2D-Texturen auf 9,1 MB, was unter anderem auf das neu installierte Unity Package für das TextMeshPro-Asset zurückzuführen war. Außerdem besaß die neue Textur eine Größe von 2,7 MB, was sie zur ressourcenintensivsten in dieses Objekttyps machte.

Der in Abbildung 41 abgebildete Speicherplatzverbrauch der Partikelsysteme zeigt, dass das CircleMesh-Partikelsystem mit 13,8 KB eine um 1,5 KB reduzierte Größe als beim originalen Effekt aufwies. Durch die Verwendung von ausschließlich Billboards verringerte sich somit die Gesamtgröße dieses Objekttypes auf 273,9 KB.

Zusammenfassend lässt sich feststellen, dass die Optimierungsmethode die GPU-Auslastung, unter anderem aufgrund der reduzierten Anzahl an zu renderten Dreiecken und Eckpunkten, minimierte. Die Veränderungen im Mesh-Partikelsystem haben zudem zu einer Verringerung des Speicherbedarfs beigetragen. Allerdings erhöhte sich die Größe der 2D-Texturen durch die neu verwendete Textur leicht.

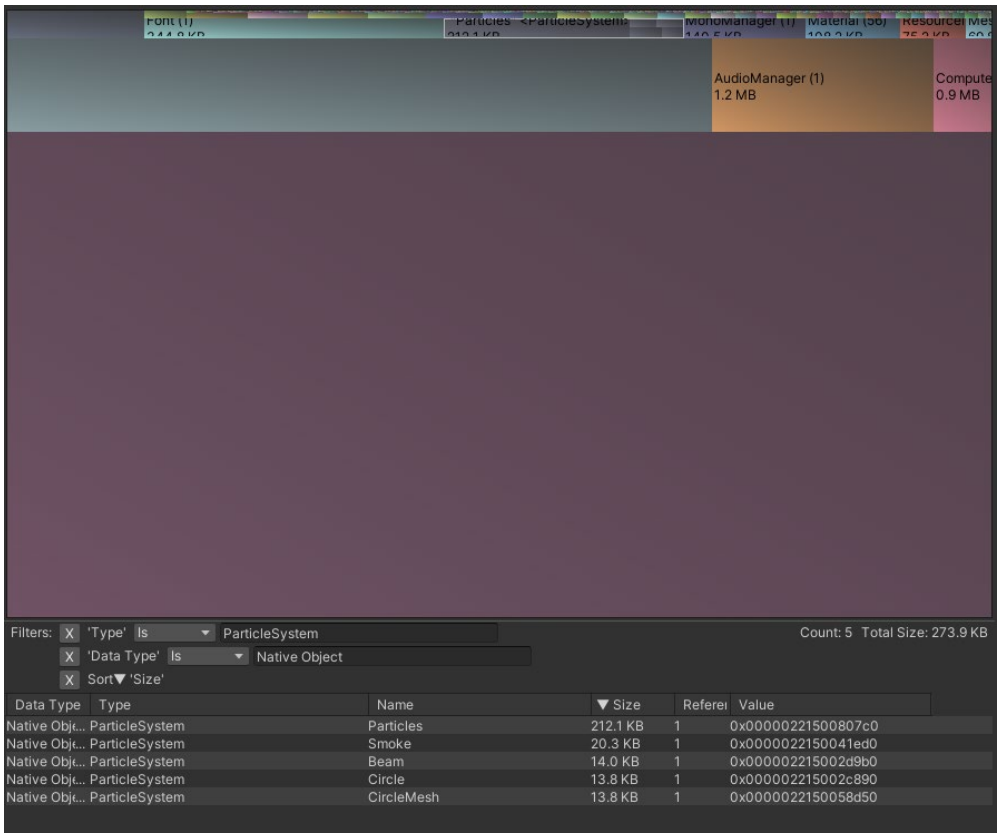

Abbildung41: Partikelsysteme in der Baumstruktur des Memory Profilers bei Verwendung von ausschließlich Billboards

4.3.3 Vergleich mit dem Level-of-Detail System

Bei der Evaluation des Level-of-Detail Systems wurde in jedem Stadium eine Bildrate von 60 FPS aufgenommen. Die Auslastung der CPU betrug ungefähr 16,7 ms pro Frame und war damit in jedem Level gleich hoch. Die Hauptbelastung resultierte jedes Mal aus dem Warten auf die Beendigung der GPU-Prozessen. Die GPU-Auslastung variierte zwischen den einzelnen Stadien. Bei Level null benötigte die Grafikkarte für ihre Berechnungen 5,5 ms bis 10,38 ms und beim ersten Stadium ungefähr 5,14 ms pro Frame. Im zweiten LOD änderte sich im Vergleich zum vorherigen nichts Wesentliches. Im dritten und vierten Level reduzierte sich die GPU-Auslastung auf durchschnittlich 1,65 ms pro Frame. Bei der letzten Stufe, bei welcher der Effekt verschwunden war, lag die Grafikkartenauslastung bei knapp über 1 000 FPS. Unabhängig vom LOD-Stadium verursachte die Ausführung des URP-HighFidelity-Renderers den höchsten Leistungsverbrauch.
Im Rendering-Teil des Profilers für das Level waren insgesamt 21 Draw Calls und sechs dynamische Batches zu erkennen. Zudem wurden 5 400 Dreiecke und 6 300 Eckpunkte gerendert. Im nächsten Stadium blieb die Anzahl der Grafikkartenaufrufe gleich. Jedoch reduzierte sich die Zahl der Dreiecke auf 3 500 und die der Eckpunkte auf 4 200, da einige kleine Partikel mit Trail entfernt wurden. Im zweiten Level verringerte sich die Anzahl der Draw Calls auf 20 und die der dynamischen Batches auf fünf, da die Entfernung der Rauchpartikel in diesem Schritt aus dem System stattfand. Dadurch und durch die Reduzierung der kleinen Partikel mit Trail hatte sich die Zahl der Dreiecke auf 1 600 und die

der Eckpunkte auf 1 900 verringert. Im dritten LOD-Level waren nur noch drei dynamischen Batches und 18 Draw Calls zu sehen. In diesem Schritt entfielen zwei der Grafikkartenaufrufe, da zusätzlich die kleinen Partikel mit Trail entfernt wurden und in deren Partikelsystem zwei Materialien renderten. Daher reduzierte sich auch die Anzahl der Dreiecke auf 442 und die der Eckpunkte auf 768. Da im vierten Level des Weiteren das halbrunde Mesh-Partikelsystem entfernt wurde, verringerte sich die Zahl der Draw Calls auf 17 und die der dynamischen Batches auf zwei. Dies ist in Abbildung 81 dargestellt, wobei sich die Anzahl der Dreiecke auf 58 belief und die der Eckpunkte bei 128 lag. Wenn der Effekt verschwunden war, entfielen die dynamischen Batches und es gab nur noch 15 Draw Calls. Die Zahl der Dreiecke reduzierte sich auf 44 und die der Eckpunkte auf 100, wobei einige davon für die Benutzeroberfläche mit der FPS-Anzeige verwendet wurden. Der Frame Debugger zeigte deutlich die Reduzierung der Draw Calls über die Level hinweg. So beliefen sie sich bei dem nullten und ersten Stadium auf sechs, im zweiten Level auf fünf, in dem dritten LOD auf drei und bei dem letzten auf zwei Aufrufe.

In der Baumstruktur des Memory Profilers von Unity blieb die Speichernutzung bei allen

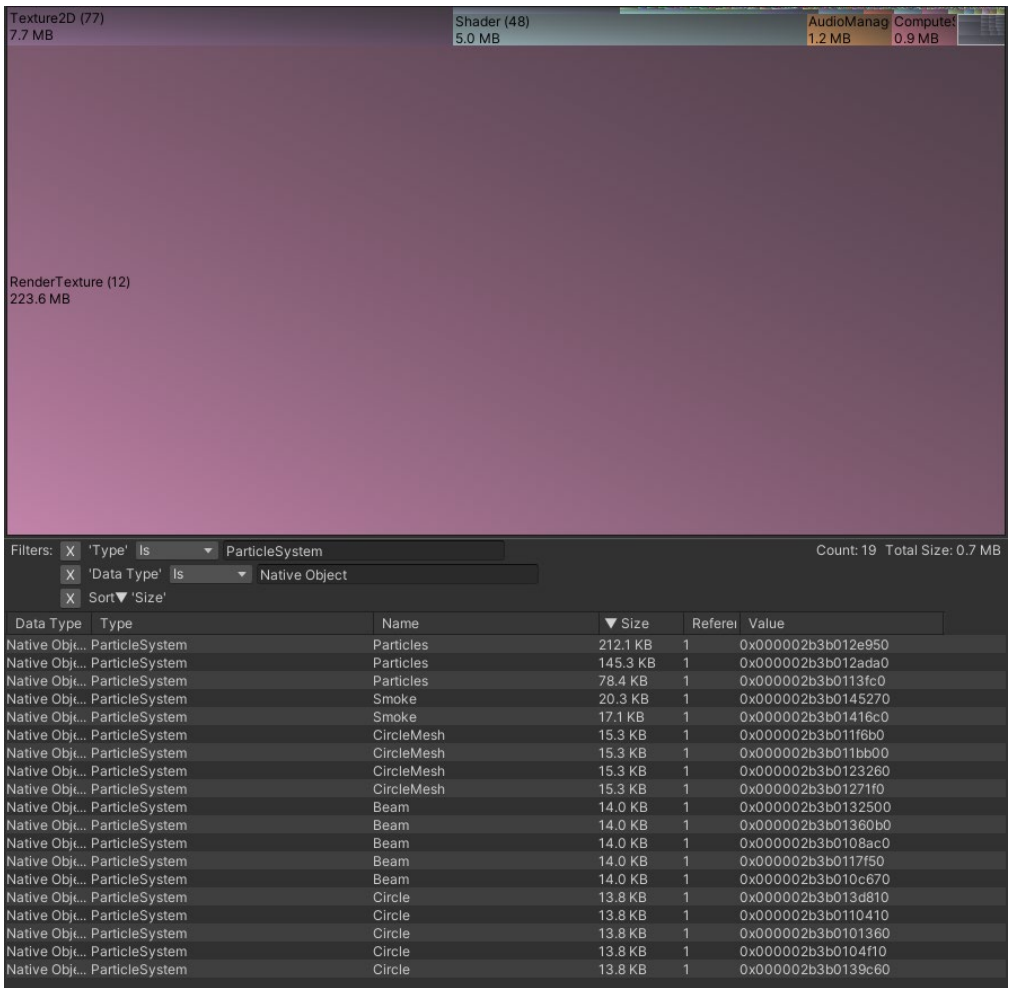

Abbildung 42: Partikelsysteme in der Baumstruktur des Memory Profilers des LOD-Systems

Leveln gleich. Die Größe der 2D-Texturen hatte sich im Vergleich zum originalen Effekt

nichts verändert. Allerdings erhöte sich der Speicherplatzverbrauch und die Anzahl der Partikelsysteme, da jedes Level einen eigenen visuellen Effekt und die entsprechend benötigten Bestandteile besaß. Insgesamt wurden 19 Systeme mit einer Größe von 0,7 MB abgespeichert. Im Vergleich zum originalen Effekt stellte dies eine Steigerung von ca. 425 KB dar. Wie in Abbildung 42 sichtbar ist, variierte die Größe der gleichnamigen Elemente je nach Anzahl der Partikel in dem jeweiligen Level und Effekt. Zudem war das Rauch-Partikelsystem, welches mit „Smoke“ bezeichnet war, nur zweimal aufgelistet, da es nur in den Leveln null und eins vorkam.
Insgesamt zeigte diese Optimierungsmethode, dass die CPU-Auslastung unverändert blieb, doch die Graffikkarte bei zunehmen LOD-Level entlastet wurde, insbesondere in Bezug auf die Anzahl der Draw Calls, Dreiecke und Eckpunkte. Jedoch hatte sich der benötigte Speicherplatz für die Partikelsysteme etwas erhöht.

4.3.4 Vergleich mit dem Culling System

Um die Optimierungstechnik des Culling Systems zu testen, fand die Durchführung von Experimenten statt, bei denen Daten sowohl ohne als auch mit einem Objekt vor der Kamera und dem Effekt erhoben wurden. Bei beiden Varianten war eine Bildrate von 60 FPS zumessen. Die Auswertung der CPU- und GPU-Auslastung ergab keine signifikanten Unterschiede zwischen den Methoden. Die Verarbeitungszeiten pro Frame lagen beim Prozessor bei ca. 16 ms und bei der Grafikkarte bei ungefähr 5,2 ms bis 5,4 ms. Abbildung 80 zeigt die Operationen der GPU. Es ist erkennbar, dass mit 54 Draw Calls mehr Aufrufe auf der Grafikkarte stattfanden als im Vergleich zu Abbildung 82. Bei diesem befand sich das Quadrat vor der Kamera, wodurch insgesamt nur 48 Draw Calls vorhanden waren. Beim Rendering-Teil des Profilers, welcher in Abbildung 83 zu sehen ist, wurde offensichtlich, dass bei dem verdeckten Effekt keine Durchführung des dynamischen Batching stattfand, was bei der normalen Variante der Fall war. Zusätzlich hatte sich die Dreieck- und Eckpunktanzahl deutlich reduziert. In der Variante mit Quadrat vor der Kamera renderten nur 118 Dreiecke und 248 Eckpunkte, wobei bei dem sichtbaren Effekt 4 600 Dreiecke und 5 600 Eckpunkte vorhanden waren.
Im Frame Debugger wurde bei dem Vergleich der beiden Test festgestellt, dass in der verdeckten Variante keine Renderaufrufe für transparente Objekte, wie den visuellen Effekt, stattfanden. Daher erfolgte kein Rendering der Partikel, wenn diese von einem anderen Element verdeckt waren. Ist der Effekt wieder sichtbar, fanden die sechs Draw Calls für transparente Objekte statt.

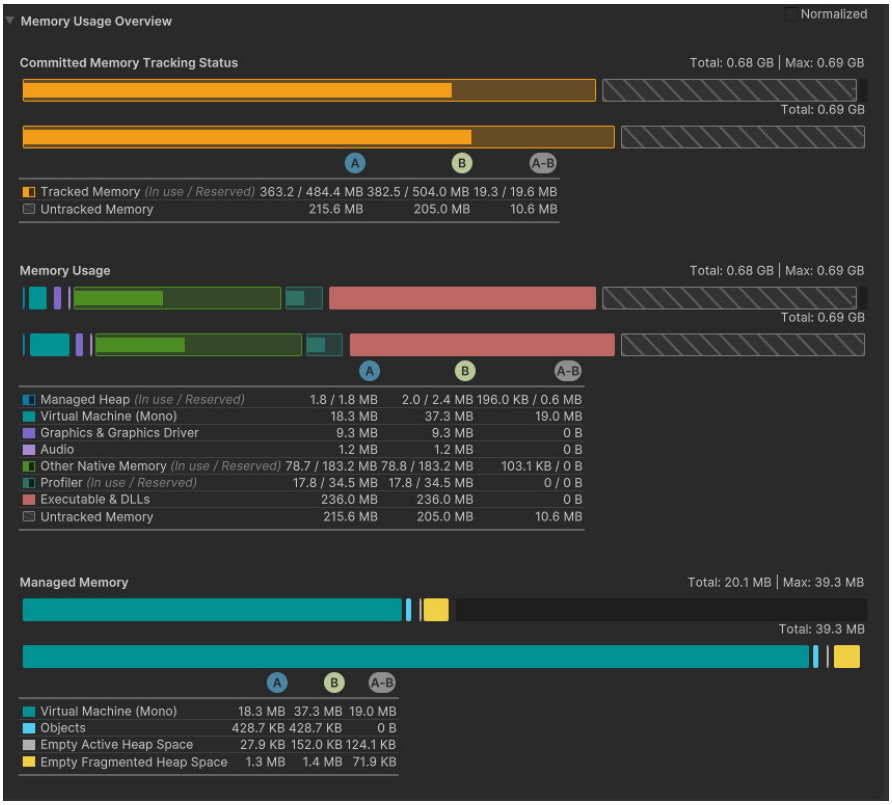

Abbildung 43: Übersicht der Speichernutzung des Culling-Systems im Memory Profiler

In Abbildung 43 ist der Vergleich der Speichernutzung zwischen den beiden Varianten dargestellt, wobei „A“ für den verdeckte und „B“ für den sichtbare Effekt steht. Insgesamt wurde bei der ersten Variante ungefähr 20 MB weniger des aufgezeichneten Speichers verwendet. Wie eine Analyse der Speichernutzung zeigte, war die virtuelle Maschine bei Variante B um 19 MB größer. Die Betrachtung der beiden Baumstrukturen des Speichers machte deutlich, dass auch bei der Variante A alle 2D-Texturen, die die Partikelsysteme verwendeten, gespeichert wurden. Somit waren bei diesem Objekttyp keine Speicherunterschiede vorhanden. Jedoch gab es bei Variante A keine gespeicherten Partikelsysteme, welche in der Variante B vorhanden waren. So wurden ca. 275,4 KB an Speicher gespart.

Insgesamt war bei der Optimierungsmethode des Culling Systems insbesondere eine Verbesserung in Bezug auf die Speicherplatzgröße, sowie die Anzahl der Draw Calls, Dreiecken und Eckpunkte erkennbar. Die CPU- und GPU-Auslastung blieb jedoch weitgehend unverändert, da der Effekt nur einen geringen Einfluss darauf besaß.

4.3.5 Vergleich mit Batching

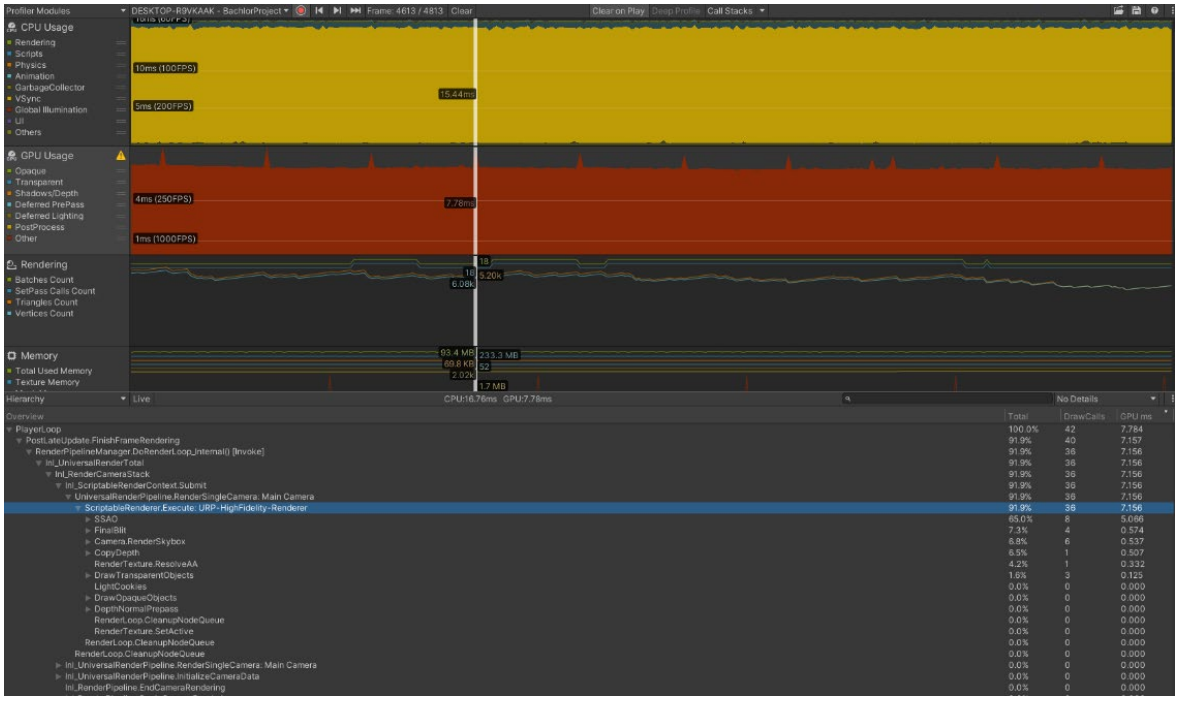

Abbildung 44: GPU-Auslastung beim Batching

Bei der Optimierungstechnik des Batchings war bei den Tests im Durchschnitt 60 FPS zu messen. Zudem benötigte die CPU etwa 16,7 ms pro Frame für ihre Operationen, wobei 91,5% dieser Zeit für das Warten auf die Beendung der GPU-Prozesse aufgewendet wurden. Die Grafikkartenauslastung ist in Abbildung 44 dargestellt. Das Bild zeigt, dass im Durchschnitt die Berechnungen der GPU 7,8 ms pro Frame beanspruchten. Davon wurden 91,9% zur Ausführung des URP-HighFidelity-Renderer verwendet, während das Zeichnen der transparenten Objekte des visuellen Effektes 1,6% der Auslastung veranschlagte. Zudem war feststellbar, dass im Vergleich zum originalen Effekt nur drei anstelle von sechs Draw Calls benötigt wurden, um die transparenten Elemente darzustellen. Im Rendering-Teil des Unity Profilers, welcher in Abbildung 85 zu sehen ist, wurde deutlich, dass bei einigen der Draw Calls des visuellen Effektes Batching stattfand und daher beim dynamischen Batching sechs Draw Calls in drei Gruppen bzw. Batches aufgeteilt waren. Die Anzahl der Dreiecke und Eckpunkte blieb größtenteils unverändert.

Beim in Abbildung 84 dargestellten Frame Debugger Ausschnitt wird sichtbar, dass bei der Operation „DrawTransparentObjects“ nur drei Aufrufe pro Frame erfolgten. Der erste davon renderte das Material des Mesh-Partikelsystems, welches die Batching-Textur enthielt. Der nächste Aufruf bildete das Material mit der Batching-Textur, dass im Kreis-, Lichtpunkt-, Rauch- und den kleinen Partikel-Partikelsystemen vorkam, ab. Durch die gemeinsame Nutzung desselben Materials konnten die Draw Calls der drei Elemente in einem Batch rendern und so die Grafikkarte entlastet werden. Jedoch fand kein Batching mit dem vorherigen Aufruf statt, obwohl beide dasselbe Material besaßen. Der Grund dafür war, dass die Objekte verschiedene Batching-Schlüssel aufwiesen, was bei Partikel-

systemen unter anderem durch unterschiedliche Vertex Streams oder der Vermischung von Linien und Trails bzw. von lit und unlit Geometrie hervorgerufen werden kann. Da das Mesh-Partikelsystem einen anderen Rendermodus als die anderen Systeme hatte, war keine Gruppierung möglich und ein eigener Draw Call erforderlich. Beim letzten Aufruf fand das Rendern des Trail-Materials statt, da sie, auch wenn sie ein Teil der kleinen Partikel waren, eine eigene Textur besaßen. Diese wurde von Unity automatisch festgelegt und konnte daher nicht wie bei den anderen Systemen in einem gemeinsamen Bild bzw. Material zusammengefasst werden.

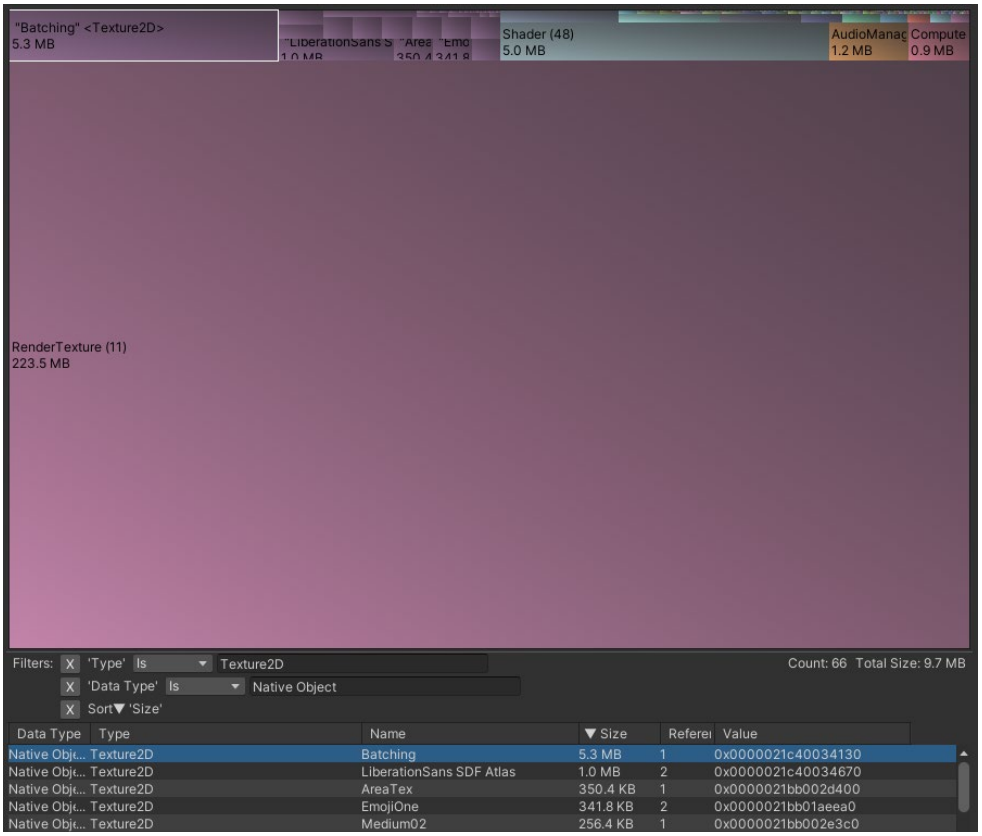

Abbildung 45: 2D-Texturen in der Baumstruktur des Memory Profilers beim Batching

Der in Abbildung 45 dargestellte Memory Profiler von Unity zeigt in der Baumdarstellung des Speichers, dass die 2D-Texturen etwas mehr Speicherplatz beanspruchten, da eine zusätzliche Textur, die Batching-Textur, vorhanden war. Diese hat eine Größe von 5,3 MB. Obwohl die einzelnen im originalen Effekt verwendeten Texturen nicht mehr gespeichert wurden, war die Batching-Textur so speicherintensiv, dass die Gesamtgröße der 2D-Texturen sich trotzdem auf 9,7 MB erhöhte. Es wäre jedoch möglich das neue Element mit einer geringeren Auflösung zu speichern, um dadurch Speicherplatz zu sparen. Die Partikelsysteme selbst nahmen genauso viel Speicherplatz ein wie beim originalen visuellen Effekt.

Zusammenfassend lässt sich bei dieser Optimierungsmethode festhalten, dass sich die Anzahl der Batches halbiert hat und dadurch weniger Aufrufe an die GPU getätigt wurden. Allerdings konnten die Durchführung nicht bei allen Partikelsystemen stattfinden, da durch

die Verwendung des Mesh-Rendermodus und der Trails verschiedene Materialien und Rendermethoden zustande kamen. Zudem hatte sich durch die Verwendung einer neuen Textur der benötigte Speicherplatz der 2D-Texturen erhöht. Weiterhin wurde eine Leistungsverschlechterung bei der GPU festgestellt, da die Draw Calls und somit auch das Batching nur ein geringer Anteil bei den Berechnungen der GPU ausmachten und die Größe der neuen zu rendernden Textur viel höher war. Bei komplexeren Effekten und mehreren gleichzeitig abgebildeten Instanzen ist dennoch eine Optimierung zu erwarten. Das wurde jedoch in dieser Arbeit nicht weiter getestet.

4.3.6 Vergleich mit dem Visual Effect Graph

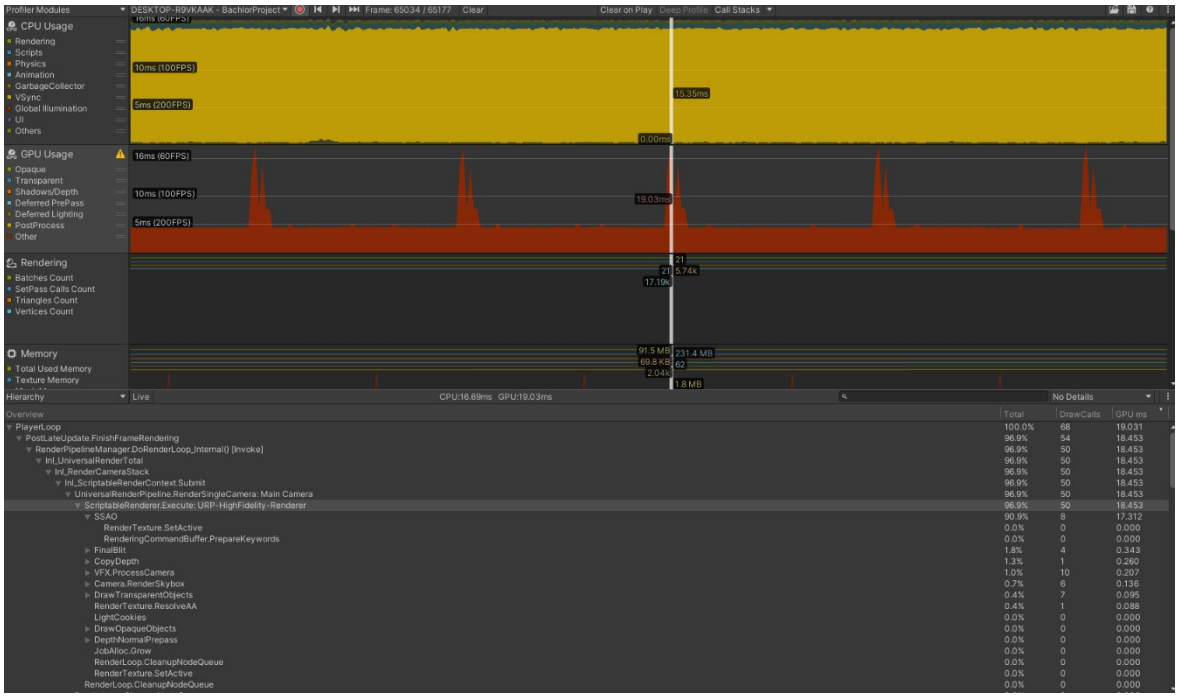

Abbildung 46: GPU-Auslastung des Visual Effect Graphes

In der Evaluation der Umsetzung des originalen Systems als Visual Effect Graph wurden im Spiel durchschnittlich 60 Frame pro Sekunde gemessen. Die CPU benötigte ebenfalls ca. 60 FPS, um alle Prozesse abzuschließen. Davon entfielen 91,4% auf das Warten auf die Beendung von GPU-Prozessen. Abbildung 46 zeigt die aufgezeichnete Grafikkartenauslastung des Profilers von Unity. Dabei entfielen 96,9% der Auslastung auf den URP-HighFidelity-Renderer. Das Zeichnen transparenter Objekte machte dabei lediglich 0,4% aus. Ein neuer Prozess namens „VFX.ProcessCamera“ war zudem sichtbar und benötigte 1,0% der Auslastung sowie zehn Draw Calls. Weiter unten in den Aufzeichnungen des Profilers wurde das VFX-Update mit 4,6% angezeigt. Die Berechnung der SSAO-Methode erzeugte die Spitzen der GPU-Nutzung, da diese 40% bis 90% der Verarbeitung in Anspruch nahm. Insgesamt benötigte die Grafikkarte für die Berechnungen bei den höchsten Auslastungen 19 ms und normalerweise 5,5 ms. Bei den Aufzeichnungen des Renderings in Abbildung 86 wird sichtbar, dass keine Gruppierung stattfand, da die Anzahl der Draw

Calls und Batches gleich war. Jedoch erfolgte beim Visual Effect Graph kein dynamisches Batching und insgesamt gab es 21 Draw Calls und somit mehr als beim originalen System. Die Anzahl der Dreiecke war ca. gleichgeblieben, jedoch hatte sich die Eckpunktzahl mit 17 200 im Vergleich zum ursprünglichen Effekt mehr als verdoppelt.
In Abbildung 87 des Frame Debuggers wird ersichtlich, dass die transparenten Objekte in sechs Batches und sieben Draw Calls renderten. Zudem fand in jedem Aufruf die Verwendung der Methode „Draw Procedural Indexed Indirect“ statt, um die Texturen abzubilden. Bei dem Graphen mit den Partikeln mit Trail wurde diese Funktion zweimal aufgerufen, da zwei Output-Teile mit verschiedenen Texturen renderten.

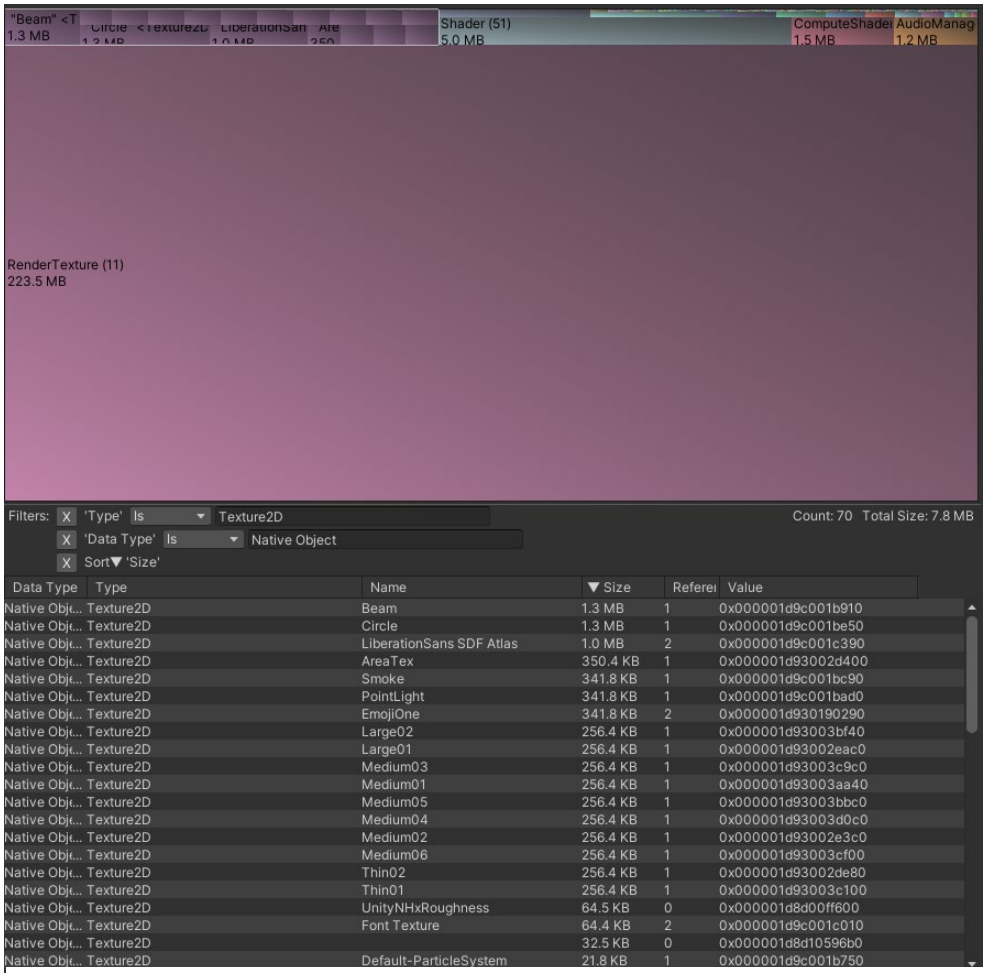

Abbildung 47: 2D-Texturen in der Baumstruktur des Memory Profilers des Visual Effect Graphes

Im in Abbildung 47 dargestellten Memory Profiler von Unity ist sichtbar, dass die 2D-Texturen ungefähr 1 MB mehr Speicherplatz benötigten. Dies war auf ein neu integriertes Text-Asset Pack und die damit eingefügte 1 MB große LiberationSans-Textur zurückzuführen. Ansonsten wurden im Visual Effect Graph alle Texturen wie im ursprünglichen Effekt verwendet, weshalb sie auch dieselbe Größe besaßen. Im Vergleich zum Original fand keine Auflistung der einzelnen Partikelsysteme statt, sondern wie in Abbildung 48 sichtbar, nur die Anzeige des VFX und eines VFX-Assets, welche eine Größe zwischen

18,6 KB und 20,8 KB besaßen. Dies liegt daran, dass die einzelnen Graphen in einem Asset vereint sind und keine eigenständige Gameobjekte in Unity darstellen.

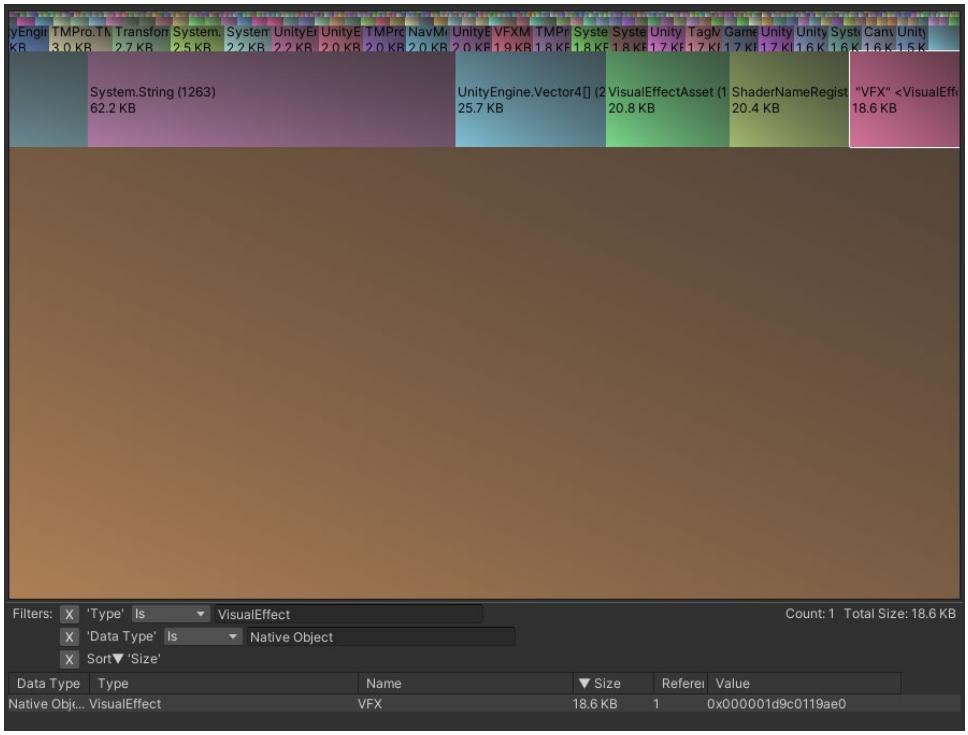

Abbildung 48: VFX in der Baumstruktur des Memory Profilers des Visual Effect Graphes

Insgesamt zeigte sich im Vergleich zum originalen Effekt keine Verbesserung. Tatsächlich hatte die GPU eine höhere Auslastung und es wurden mehr Eckpunkte gerendert. Die einzige positive Änderung bestand im Speicherplatzbedarf, da der einzelne Visual Effect Graph eine geringere Größe als die Partikelsysteme gemeinsam aufwies. Eine wirkliche Verbesserung dürfte wahrscheinlich erst bei Systemen mit mehreren tausend Partikeln oder komplexeren Shader sichtbar werden. Die Behandlung eines solchen Falls erfolgt jedoch in dieser Arbeit nicht.

4.3.7 Vergleich mit der Kompression von Texturen

In der Evaluierung der Optimierungsmethode wurde eine Bildrate von 60 FPS ermittelt. Die CPU-Auslastung betrug ca. 16 ms pro Frame. Die zeitaufwendigste Methode bestand im Warten auf die Beendigung der GPU-Operationen.

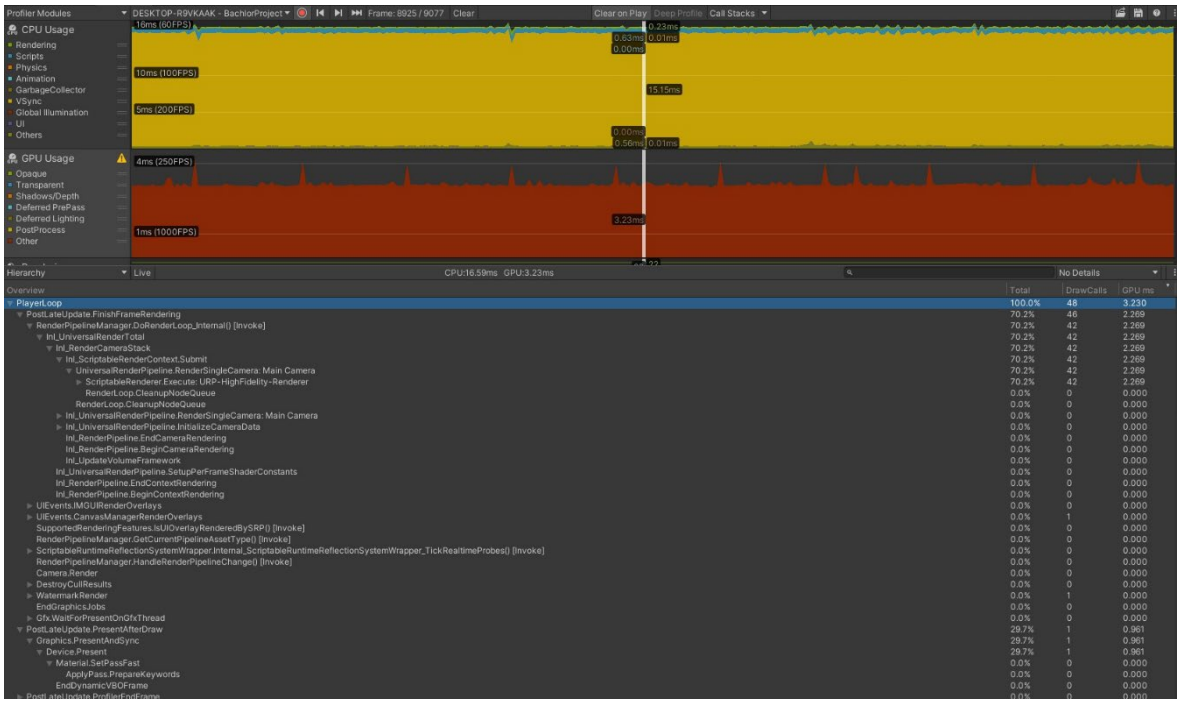

Abbildung 49: GPU-Auslastung bei der Texturkompression

Die Grafikkartenauslastung betrug, wie Abbildung 49 dargestellt, im Durchschnitt 3,3 ms pro Frame. Dies entspricht einer Reduzierung um etwa 1,5 ms im Vergleich zum originalen Effekt. Während der Durchführung der GPU-Operationen entfielen 70,2% der Zeit auf die Ausführung des URP-HighFidelity-Renderers und 29,7% auf das Warten der Fertigstellung des Hauptthreads. Die Anzahl der Draw Calls, dynamischen Batches, Dreiecke und Eckpunkte im Rendering-Teil des Profilers blieb im Vergleich zum originalen Effekt unverändert. Ebenso waren im Frame Debugger keine Unterschiede zum ursprünglichen System ersichtlich.

In der Baumansicht des Memory Profilers, welche in Abbildung 50 dargestellt ist, wird ersichtlich, dass sich die Speichergröße der 2D-Texturen um 2 MB auf 5,7 MB verringerte. Diese Reduzierung war auf die Verwendung der komprimierten Bilder in den Partikelsystemen zurückzuführen. Alle Texturen besaßen eine Größe von 341,8 KB und sind in der Abbildung 50 mit „_Compressed“ gekennzeichnet.

Insgesamt hatte sich durch diese Optimierungsmethode im Vergleich zum originalen Effekt die Auslastung der Grafikkarte und die Speichergröße der 2D-Texturen verringert.

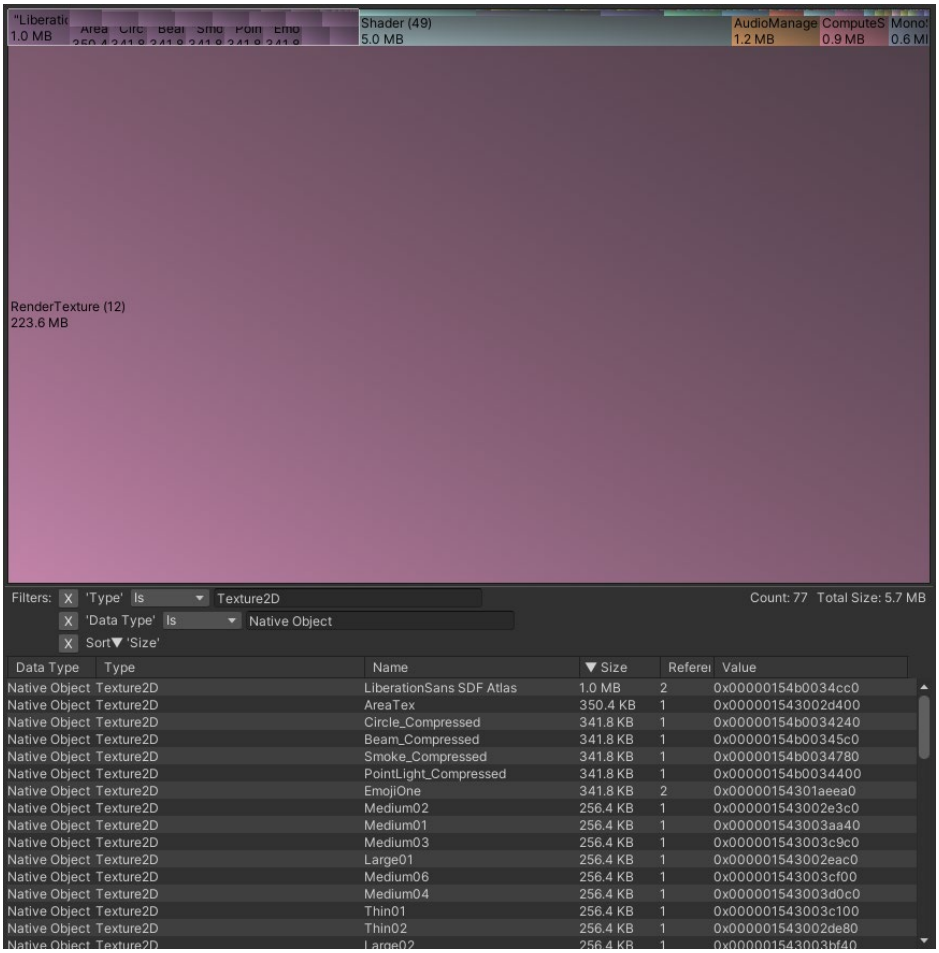

Abbildung 50: 2D-Texturen in der Baumstruktur des Memory Profilers bei der Texturkomprimierung

4.3.8 Untersuchung der mehrfachen Instanziierung des Partikelsystems

4.3.8.1 Allgemeines

Um die Effizienz der Optimierungstechnik des Pooling-Systems auswerten zu können, wurden zunächst Tests ohne diese Methode durchgeführt. Dabei erfolgte lediglich eine Änderung in der Aktivierung und Deaktivierung, während der Zeitintervall bei sieben Sekunden und die Anzahl der Effekte bei 100 bzw. 200 blieben. Die Menge der sichtbaren Elemente variierte, um die Funktionsweise des Pooling-Systems eindeutiger darstellen und den Einfluss von deaktivierten Elementen auf die betrachteten Metriken beobachten zu können.

4.3.8.2 Originale Instanziierung

In der Variante ohne die Optimierungstechnik wurde bei der Anzeige von 100 oder 200 Systemen eine Bildrate von 60 FPS beobachtet, wobei diese bei der Initialisierung von neuen Effekten kurzzeitig auf 55 FPS sank. Dies verdeutlichte den Einfluss der Erstellung von Objekten auf die Frames pro Sekunde. Die Auslastung der CPU blieb konstant bei 16 ms pro Frame. Bei der Darstellung von 200 Effekten stieg dabei der Anteil des Render-Loops an der Gesamtzeit auf ca. 20 %. Dadurch verringerte sich die Wartezeit auf die Beendigung der GPU-Prozesse auf 56,2% der Gesamtzeit.

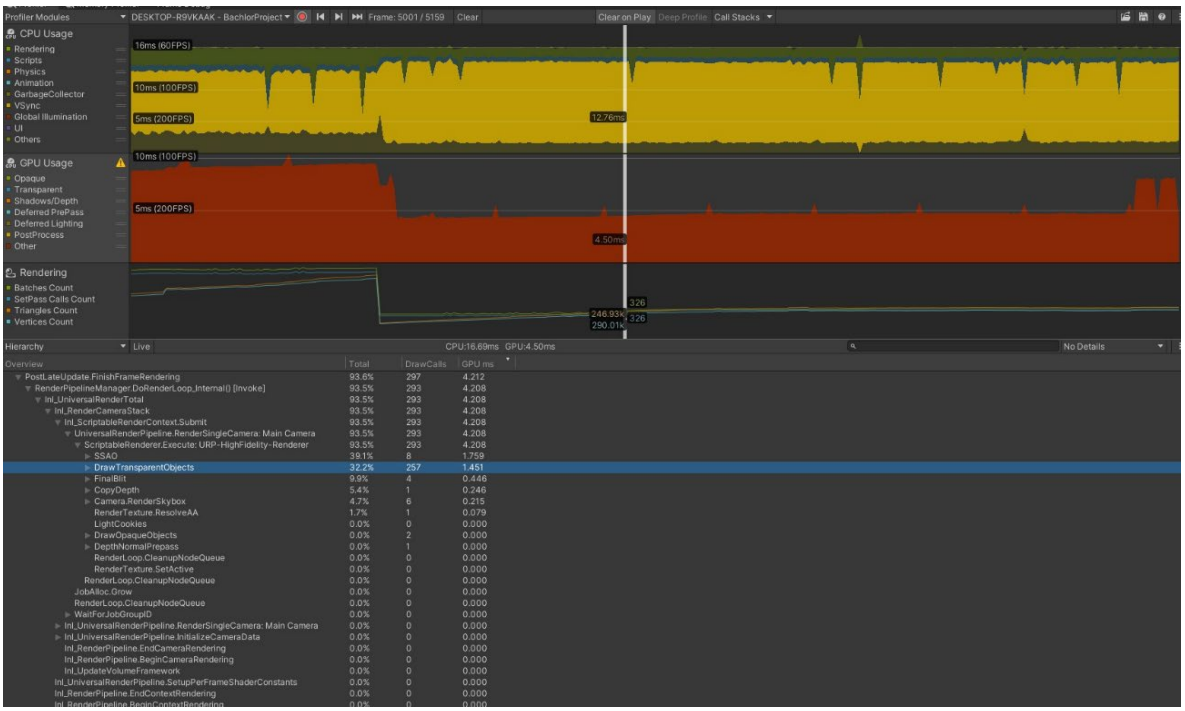

Abbildung 51: GPU-Auslastung der originalen Instanziierung

Bei der Auslastung der Grafikkarte zeigten sich, wie in Abbildung 51 zu sehen, signifikante Unterschiede zwischen 100 und 200 Effekten. Die GPU benötigte bei der geringen Anzahl an Elementen ca. 4,5 ms und bei der größeren Menge an Systemen ca. 9,5 ms pro Frame. Bei beiden Varianten beanspruchte die Ausführung des URP-HighFidelity-Renderers die meiste Zeit. Der in Abbildung 88 und Abbildung 90 abgebildete Rendering-Teil des Unity-Profilers wies eine klare Veränderung zwischen 100 und 200 angezeigten Effekten auf. Bei der geringeren Menge an Systemen waren 326 Draw Calls und 311 dynamische Batches vorhanden. Die Anzahl der Dreiecke belief sich auf 246 900 und die der Eckpunkte auf 290 000. Bei 200 angezeigten Effekten betrug die Zahl der Draw Calls 660 und die der dynamischen Batches 627. Zudem wurden 457 600 Dreiecke und 545 300 Eckpunkte in der Szene gerendert. Im Frame Debugger belief sich die Menge der einzelnen Draw Calls für die transparenten Objekte bei 100 Elementen auf eine Zahl von 293 und bei 200 Effekten auf 623.

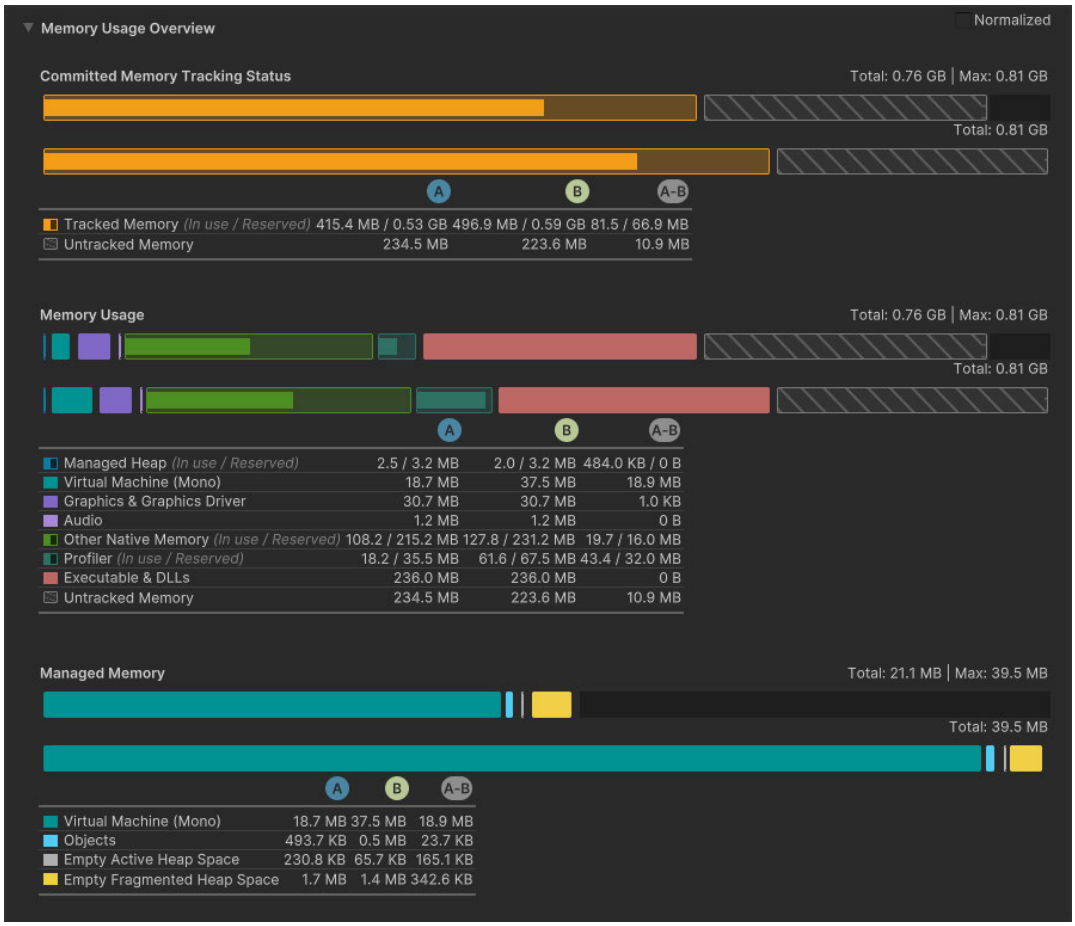

Abbildung 52: Vergleich der Speichernutzung der beiden Mengenvarianten bei der originalen Instanziierung

Beim Vergleich der in Abbildung 52 abgebildeten beiden Mengen an visuellen Effekten im Memory Profiler, wobei „A“ die kleinere und „B“ die größere Variante darstellt, wird deutlich, dass 200 Elemente im Vergleich zu 100 Systemen etwa 50 MB mehr Speicherplatz verwendeten und reservierten. Die virtuelle Maschine trug mit einer Zunahme um 18,9 MB maßgeblich dazu bei. In der Anzeige des Speicherplatzes als Baumstruktur war erkennbar, dass die Größe der Partikelsysteme bei 200 visuellen Effekten bei 44,5 MB lag, wobei sich die Anzahl der derzeit verwendeten Elemente auf 1 000 belief. Auffällig war, dass die Hälfte der Partikelsysteme der kleinen Partikel mit Trail eine Größe von 212,1 KB aufwies, wobei die anderen nur einen Speicherplatzbedarf von 116,1 KB besaßen. Dies kam durch die unterschiedlichen Initialisierungszeiten zu Stande, weil die Überprüfung des Speicherplatzes kurz nach der Erstellung einiger der visuellen Effekte stattfand. Da die kleinen Partikel mit Trail im Verhältnis zu ihrer maximalen Kapazität eine geringe Initialisierungsanzahl aufweisen, benötigen sie etwas Zeit, um ihr Maximalzahl und vollständige Größe zu erreichen. Das ist bei den anderen Partikelsystemen nicht der Fall, weshalb sie kurz nach der Erstellung ihren maximalen Speicherbedarf erlangten. Bei 100 angezeigten Elementen belief sich die Größe der Partikelsysteme auf 27 MB und ihre Anzahl auf 500. Die Speicherausnutzung der 2D-Texturen blieb in beiden Varianten unverändert.

4.3.8.3 Instanziierung mit dem Pooling System

In der Evaluierung der Optimierungsmethode des Pooling-Systems wurde bei beiden Effektanzahl-Varianten eine konstante Bildrate von 60 FPS gemessen. Die Auslastung der CPU betrug stehts ungefähr 16 ms pro Frame. Die zeitintensivste Methode war bei beiden Fällen mit durchschnittlich 70 % das Warten auf die Beendigung der GPU-Prozesse. Anschließend folgte mit ungefähr 13 % die Durchführung des Render-Loops.

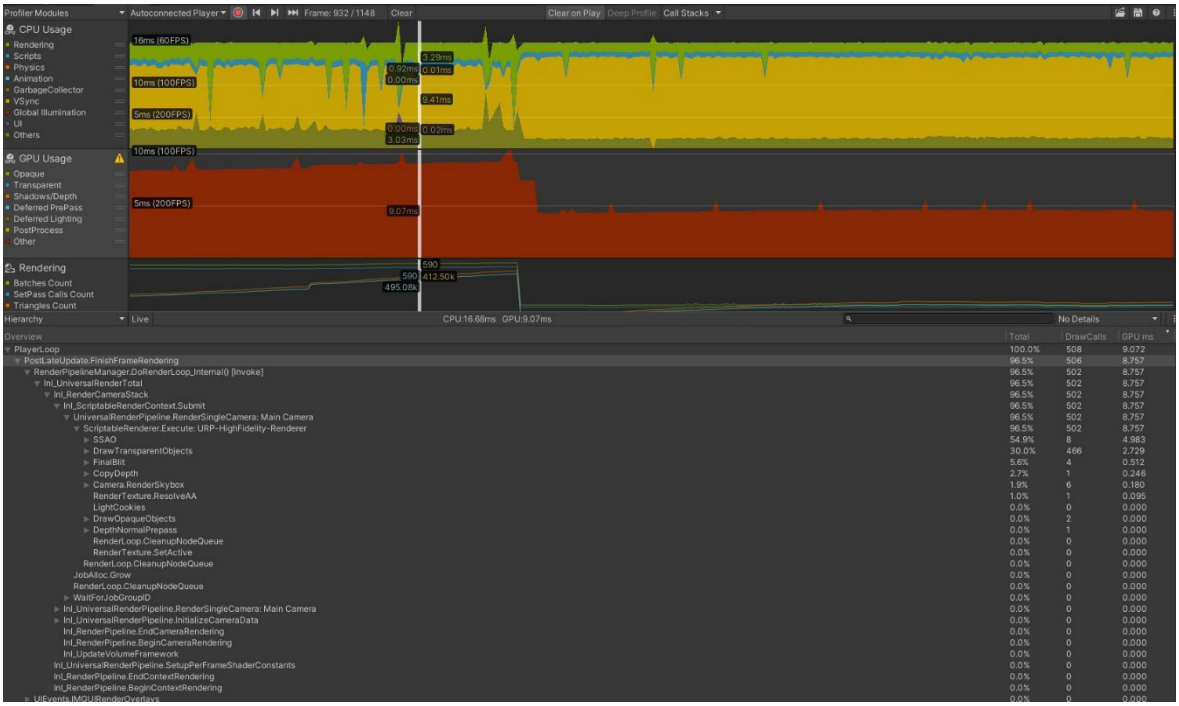

Abbildung 53: GPU-Auslastung mit Pooling-System

Die Auslastung der Grafikkarte zeigte, wie in Abbildung 53 zusehen, signifikante Unterschiede zwischen den beiden Varianten an Partikelsystemmengen. Bei 100 Effekten benötigte sie ungefähr 4,4 ms pro Frame für die Ausführung der Operationen, während es bei 200 Elementen ca. 9 ms waren. Diese Zeiten waren im Vergleich zur Instanziierung ohne Pooling-System nur minimal niedriger. In beiden Fällen beanspruchte die Ausführung des URP-HighFidelity-Renderers den Großteil der Gesamtzeit. Der Rendering-Teil des Unity-Profilers zeigte zudem große Unterschiede zwischen der Anzeige von 100 und 200 Effekten. So waren bei der geringeren Anzahl 324 Draw Calls und 290 dynamische Batches vorhanden. Außerdem wurden 245 800 Dreiecke und 287 900 Eckpunkte gerendert. Bei der Variante mit 200 Elementen fanden 631 Draw Calls und 603 dynamische Batches statt. Die Anzahl der Dreiecke belief sich auf 424 400 und die der Eckpunkte auf 507 300. Im Frame Debugger war außerdem zu sehen, dass bei der Darstellung von 100 Effekten 290 Draw Calls für transparente Objekte durchgeführt wurden. Bei der Anzeige von 200 Elementen beliefen sich die Aufrufe auf 603.

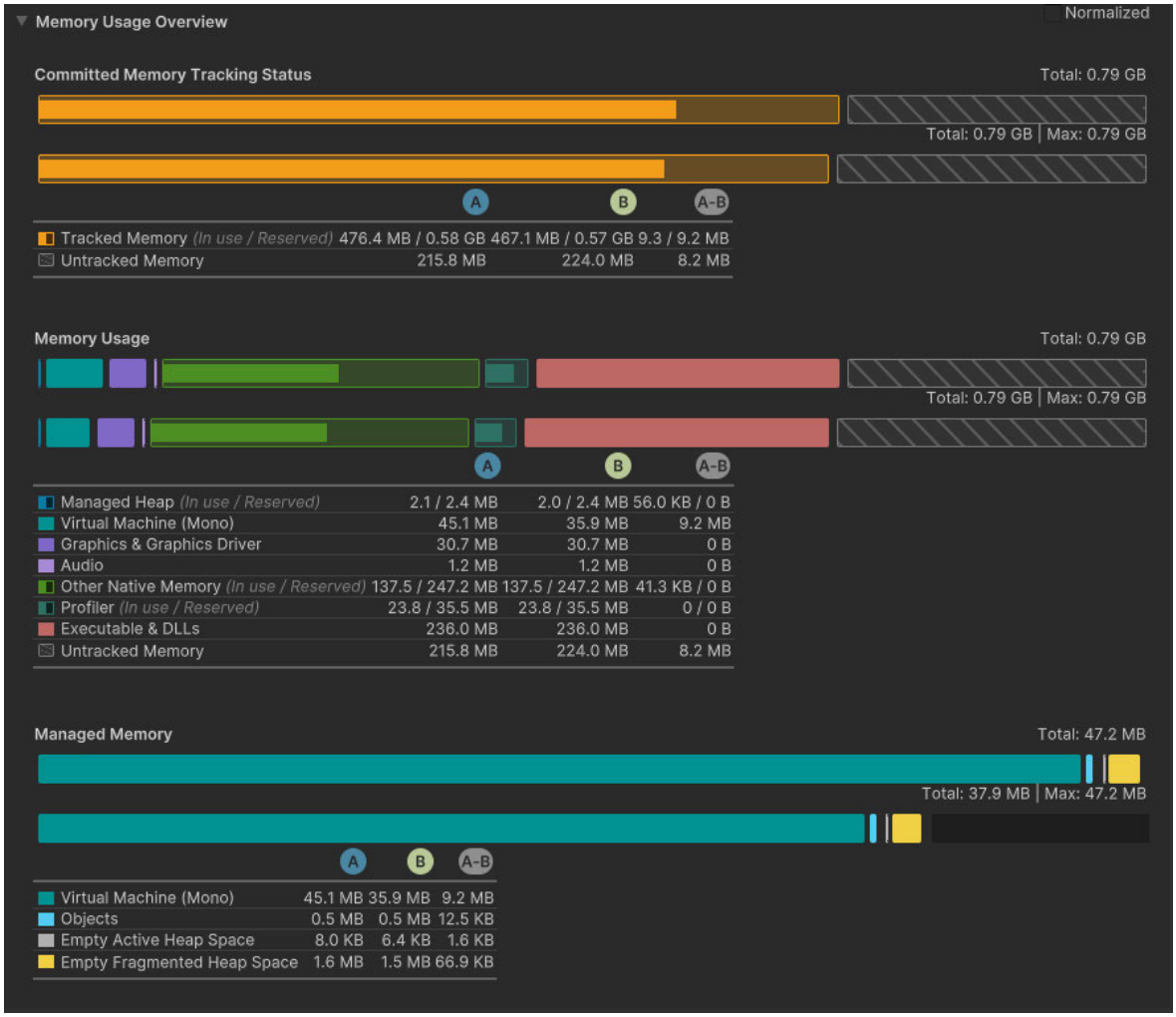

Abbildung 54: Vergleich der Speichernutzung der beiden Mengenvarianten mit Pooling-System

Die Darstellung des Memory Profilers in Abbildung 54 zeigt bei den beiden verschiedenen Mengen an Effekten keinen signifikanten Unterschied im Speicherplatzverbrauch. Die im Bild mit „A“ gekennzeichnete Variante mit 100 Elementen verwendete jedoch 9,2 MB mehr Speicher für die virtuelle Maschine. Im Vergleich mit der Variante ohne Pooling-System wurden zwar 30 MB mehr Speicher bei Darstellung von 100 Effekte benötigt, aber 20 MB eingespart, wenn die Anzeige von 200 Elemente erfolgte. Die Baumdarstellung des Speichers blieb mit einer Größe von 53,8 MB der 1 000 Partikelsysteme bei beiden Effektanzahl-Varianten gleich. Die Menge dieses Datentypes glich zwar der Variante ohne Pooling, jedoch war der Speicherplatzverbrauch um 9,3 MB höher. Dies lag daran, dass die Partikelsysteme bei der Optimierungstechnik nicht jedes Mal neu initialisiert wurden. Somit mussten sich die Partikel mit Trail nicht ständig neu aufbauen und waren daher bereits an ihrer Maximalanzahl und höchstem Speicherplatzverbrauch angelangt. Die gleichbleibende Größe der Partikelsysteme bei 100 sowie bei 200 Effekten zeigte, dass zu jeder Zeit alle Objekte im Speicher vorhanden waren, auch wenn diese nicht angezeigt wurden. Bei dem Speicherplatzverbrauch der 2D-Texturen erfolgte keine Veränderung. Insgesamt ergab sich bei der Optimierungstechnik eine signifikante Veränderung der Bild-

rate während der Initialisierung bzw. Aktivierung der Effekte. Mit dem Pooling System war die Anzahl der Draw Calls und dynamischen Batches etwas höher, jedoch blieb der Speicherplatzbedarf konstant und die Größe der Partikelsysteme stieg etwas.

4.4 Zusammenfassung der Evaluation

In der Evaluation wurde festgestellt, dass jede Methode unterschiedliche Auswirkungen auf die verschiedenen betrachteten Metriken hatte. Ein Vergleich der Optimierungstechniken ist in Tabelle 1 dargestellt. Diese zeigt, dass lediglich zwischen der Instanziierung mehrerer Effekte und der Verwendung des Pooling-Systems eine Steigerung der Bildrate stattfand. In diesem Fall erhöhte sich dieser Wert während der Erstellung neuer Systeme um ungefähr 5 FPS. Bei der CPU-Auslastung waren bei allen Methoden keine signifikanten Unterschiede erkennbar. Die GPU-Beanspruchung nahm hingegen bei der Verwendung von ausschließlich Billboards und komprimierten Texturen sowie dem Einsatz des LOD-Systems ab Level zwei ab. Jedoch erhöhte sich diese bei der Optimierungstechnik des Visual Effect Graphs und des Batchings. Die Anzahl der Draw Calls verringerte sich bei dem LOD-System, da weniger Partikelsysteme während des Fortschreitens der Level gerendert wurden. Die Gruppierung der Grafikkartenaufrufe fand ausschließlich bei der Batching-Technik statt, bei der nur noch drei Batches vorhanden waren. Die Anzahl der Dreiecke und Eckpunkte reduzierte sich bei der Verwendung von ausschließlich Billboards leicht sowie signifikant bei den höheren LOD-Leveln und der Nutzung von Culling. Der Einsatz des Visual Effect Graph erhöhte hingegen die Eckpunktanzahl. Der Speicherplatzbedarf der 2D-Texturen reduzierte sich lediglich bei der Komprimierungsmethode, während er bei der Nutzung von ausschließlich Billboards, Batching und dem VFX zunahm. Die Verwendung von nur Billboards, dem Cullings-System und dem Visual Effect Graph führte zu einer Verringerung der Größe der Partikelsysteme, jedoch nahm diese beim Einsatz von LOD-Leveln zu. Auch bei der Anwendung des Pooling-Systems erhöhte sich der Speicherplatzbedarf der Partikelsysteme im Vergleich zur einfachen Instanziierung.
Insgesamt war zu erkennen, dass bei den meisten Techniken einige der betrachteten Metriken verbessert wurden und andere sich verschlechterten. Somit war keine der untersuchten Methoden optimal, da der Großteil Stärken und Schwächen aufwieß. Bei der betrachteten Technik des Visual Effect Graphs konnte zudem keine Verbesserung der Messwerte festgestellt werden.

Methodik	**Bildrate in FPS**	**CPU- / GPU-Auslastung**	**Draw Calls / Batches**	**Dreiecke / Eckpunkte**	**Größe; 2D-Texturen / Partikelsysteme**
Original	60	16,7 ms / 5 ms	6 / 6	5 800 / 6 700	6,3 MB / 275,4 KB
Billboard	60	16 ms / 3,7 ms	6 / 6	4 100 / 4 500	9,1 MB / 273,9 KB
LOD Level 0	60	16,7 ms / 5,5 ms – 10,38 ms	6 / 6	5 400 / 6 300	6,3 MB / 0,7 MB
LOD Level 1	60	16,7 ms / 5,14 ms	6 / 6	3 500 / 4 200	6,3 MB / 0,7 MB
LOD Level 2	60	16,7 ms / 5,14 ms	5 / 5	1 600 / 1 900	6,3 MB / 0,7 MB
LOD Level 3	60	16,7 ms / 1,65 ms	3 / 3	442 / 768	6,3 MB / 0,7 MB
LOD Level 4	60	16,7 ms / 1,65 ms	2 / 2	58 / 128	6,3 MB / 0,7 MB
LOD Effekt weg	60	16,7 ms / 1 ms	0 / 0	44 / 100	6,3 MB / 0,7 MB
Culling	60	16 ms / 5,3 ms	0 / 0	118 / 248	6,3 MB / 0 KB
Batching	60	16,7 ms / 7,8 ms	6 / 3	5 200 / 6 100	9,7 MB / 275,4 KB
VFX	60	16,7 ms / 5,5 ms - 19 ms	6 / 6	5 700 / 17 200	7,3 MB / VFX: 18,6 KB Asset: 20,8 KB
Textur-Kompression	60	16,6 ms / 3,3 ms	6 / 6	5 800 / 6 700	5,7 MB / 275,4 KB
100 Instanzen ohne Pooling	55 -60	16,6 ms / 4,5 ms	311 / 311	246 900 / 290 000	6,3 MB / 27 MB

Methodik	**Bildrate in FPS**	**CPU- / GPU-Auslastung**	**Draw Calls / Batches**	**Dreiecke / Eckpunkte**	**Größe; 2D-Texturen / Partikelsysteme**
200 Instanzen ohne Pooling	55 -60	16,6 ms / 9,5 ms	627 / 627	457 600 / 545 300	6,3 MB / 44,5 MB
100 Instanzen mit Pooling	60	16,6 ms / 4,4 ms	290 / 290	245 800 / 287 900	6,3 MB / 53,8 MB
200 Instanzen mit Pooling	60	16,6 ms / 9 ms	603 / 603	424 400 / 507 300	6,3 MB / 53,8 MB

Tabelle 1: Vergleich der Ergebnisse der Evaluation

5 Diskussion und Erkenntnisse

5.1 Interpretation der Ergebnisse

Durch die Resultate der Evaluation wird eine Vergleichbarkeit der verschiedenen Optimierungsmöglichkeiten erzielt. Auf diese Weise findet die Veranschaulichung der Effektivität der Techniken in Hinblick auf die unterschiedlichen Metriken statt und Nachteile werden bei der Verwendung der Methoden aufgezeigt. Die Ergebnisse zeigen, dass bei zunehmendem LOD-Level sich die GPU-Auslastung deutlich reduzierte. Ebenso war eine signifikante Verringerung der Grafikkartenauslastung bei der Kompression der 2D-Texturen festzustellen, was darauf hinweist, dass die Größe der zu rendernden Texturen die Leistung der GPU beeinflusst. Weiterhin führte die Implementierung des Pooling-Systems zu einer Erhöhung der Bildrate bei der Erstellung bzw. Aktivierung neuer Effekte.
Ein unerwartetes Ergebnis ergab sich bei dem Visual Effect Graph, bei welchem keine Verbesserung der Messwerte erfolgte. Dies könnte auf die Tatsache zurückzuführen sein, dass die Evaluierung dieser Technik lediglich anhand eines visuellen Effektes stattfand und das Systems zur Darstellung von mehreren Millionen Partikeln optimiert wurde. Eine Verbesserung im Vergleich zum eingebauten Partikelsystem könnte daher erst bei einer größeren Anzahl von Instanzen erkennbar sein. Weiterhin stieg die GPU-Auslastung trotz reduzierter Grafikkartenaufrufe bei der Optimierungsmethode des Batching. Der Umstand könnte mit der verwendeten Textur zusammenhängen, welche eine höhere Speicherauslastung als die PNGs im originalen Effekt aufweist und daher mehr Ressourcen beim Rendern beansprucht. Ebenfalls blieb die GPU-Auslastung bei der Culling-Technik unverändert, selbst wenn das Element verdeckt war. Dies könnte auf die Berechnungen für die Occlusion Area zurückzuführen sein.

5.2 Limitation der Untersuchung

Die direkte Übertragung der Ergebnisse auf die praktische Anwendung in Videospielen gestaltet sich problematisch, da sich die Untersuchung auf die Analyse der Optimierungstechniken in einem Testszenario ausrichtete. Die Evaluationsszene weicht erheblich von der realen Spielumgebung ab, da bedeutende Komponenten wie beispielsweise der Spielercharakter und interaktive Objekte fehlen. Die Anwesenheit dieser Elemente könnte die Leistung der Anwendung und die Auswirkungen der Optimierungstechniken beeinflussen. Dennoch ermöglicht diese Arbeit den Vergleich zwischen den einzelnen Methoden, insbesondere da im Rahmen des Testszenarios eindeutig festgestellt werden konnte, wie sich die einzelnen Techniken auf die Leistung und die Metriken auswirkten. Dies gibt Entwicklern die Möglichkeit die für das jeweilige Problem effektivste Optimierungsmethode auszuwählen, um die Gesamtleistung im Spiel zu steigern. Außerdem ist das Abwägen von etwaigen Nachteilen der Ansätze möglich. In der Praxis können zudem mehrere Techni-

ken gleichzeitig verwendet werden, um eine bessere Performancesteigerung zu erzielen und Nachteile der Methoden auszugleichen.
Eine weitere Limitation der Untersuchung besteht darin, dass die Evaluation der Optimierungsmöglichkeiten, mit Ausnahme des Pooling-Systems, lediglich an einem visuellen Effekt stattfand. Obwohl die Ergebnisse dadurch detaillierter sind, könnten manche Leistungsverbesserungen erst bei der Anwendung auf eine größere Anzahl an Partikelsystemen zu eindeutigen und aussagekräftigen Ergebnissen führen. Dies war bereits bei der Evaluation des Pooling-Systems sichtbar, bei welchem die Anzeige von 100 bis 200 Effekte gleichzeitig erfolgte und eine Verbesserung der Bildrate wahrgenommen werden konnte. Eine ähnliche Änderung trat bei den anderen Methoden nicht auf. Die Untersuchung zielte jedoch auf detailliertere Ergebnisse ab, weshalb die Untersuchung der restlichen Techniken mit einer größeren Anzahl an Effekten ausblieb.

6 Zusammenfassung und Ausblick

In dieser Arbeit erfolgte der Vergleich der Effizienz verschiedener Möglichkeiten zur technischen Optimierung der Darstellung visueller Effekte in der Game Engine Unity. Das Hauptziel bestand darin, Varianten zur Verbesserung der Leistung bei Partikelsystemen aufzuzeigen, ihre Effektivität zu untersuchen und sie miteinander zu vergleichen. Dazu begann die Untersuchung mit einem umfassenden Literaturüberblick über bestehende Techniken zur Optimierung der Systeme. Als nächstes fand die Erstellung des zu testenden visuellen Effektes und die Implementierung ausgewählter Ansätze in einer vordefinierten Testszene statt. Die anschließende Evaluation erfolgte anhand einer Reihe von Metriken, um deren Auswirkung und Effektivität auf bestimmte Leistungsparameter zu untersuchen. Zum Schluss fanden die Bewertung und Diskussion der Ergebnisse statt. Hauptergebnisse der Arbeit waren die Reduzierung der GPU-Auslastung bei der Verwendung des LOD-Systems und der Komprimierung von 2D-Texturen sowie die Erhöhung der Bildrate während der Initialisierung bzw. Aktivierung von Effekten unter Nutzung eines Pooling-Systems. Insgesamt war ersichtlich, dass viele der untersuchten Techniken Vorteile und Nachteile in Bezug auf die evaluierten Metriken aufwiesen. Eine Limitation der Arbeit besteht darin, dass bis auf beim Pooling-System die Tests der Methoden lediglich an einem visuellen Effekt stattfanden. So wurden detaillierte Ergebnisse erlangt, jedoch bleibt offen, welche Effektivität die Optimierungstechniken für mehrere gleichzeitig angezeigte Elemente aufweisen. Zudem können die Ergebnisse nicht direkt auf die praktische Anwendung in Spielen übertragen werden. Dennoch ist ein Vergleich der Methoden und somit die Auswahl der effektivsten Optimierungsvariante für das jeweilige Leistungsproblem mit Abwägung der etwaigen Nachteile möglich. Es bleibt Raum für zukünftige Forschung, um den Einfluss dieser Methoden auf die Leistung von Spielen sowie deren Anwendbarkeit in der Praxis zu erforschen.

Literatur

[Zhang & Hu, 2017] Zhang, B. & Hu, W. (2017): Game special effect simulation based on particle system of Unity3D, https://doi.org/10.1109/ICIS.2017.7960062, verfügbar am 17.09.2023, 19:00

[Xie, J., 2012] Xie, J. (2012): Research on key technologies base Unity3D game engine, https://doi.org/10.1109/ICCSE.2012.6295169, verfügbar am 17.09.2023, 19:00

[Unity Packages, 2023] https://docs.unity3d.com/Packages/com.unity.visualeffectgraph@12.0/manual/index.html, verfügbar am 17.09.2023, 19:00

[Unity Tutorial, 2023] https://learn.unity.com/tutorial/get-started-with-vfx?pathwayId=61a65568edbc2a00206076dd&missionId=61a6488fedbc2a0020607444, verfügbar am 17.09.2023, 19:00

[Unity Manual, 2023] https://docs.unity3d.com/Manual/, verfügbar am 18.09.2023, 14:00

[Flückiger, 2022] Flückiger, B.: Visuelle Effekte, In: Hartmann, B., Kuhn, M., Schick, T., Wedel, M.: Handbuch Filmwissenschaft, Stuttgart, Weimar: Metzler, 2022

[Zhang, 2020] Zhang, Y.: Using the Unity Game Engine as a Platform for Prototyping Cinematic Visual Effects (Dissertation, Department of Media), Aalto University, 2020

[Jung, 2019] Jung, B., Vitzthum, A.(2019): Virtuelle Welten, In: Dörner, R., Broll, W., Grimm, P., Jung, B. (eds): Virtual und Augmented Reality (VR/AR), Berlin, Springer Vieweg, https://doi.org/10.1007/978-3-662-58861-1_3, verfügbar am 17.09.2023, 19:00

[Yan et al., 2023] Yan, X., Liang, J., Xie, P., Wu, Y. (2023): Research on Unity Scene Optimization Based on Fast LoD Technique Performance Comparison on Android Mobile Platform, https://doi.org/10.1109/ISCTIS58954.2023.10213077, verfügbar am 17.09.2023, 19:00

[Aleksi, 2018] Aleksi, L.: Optimizing Unity Projects (Dissertation, Business Information Technology), Kajaanin University, 2018

[Minh et al., 2022] Minh, L., Minh, T., Sang, L., Khanh, H. (2022): Object level frustum culling based frame rate acceleration method, https://doi.org/10.54939/1859-1043.j.mst.CSCE6.2022.28-40, verfügbar am 17.09.2023, 19:00

[Ionin, 2023] Ionin, K..: Comparing performances between different methods of using large numbers of ParticleSystem effects in Unity games on Android OS (Dissertation, Information and Communication Technology), Turku University, 2023

[Unity Script, 2023] https://docs.unity3d.com/ScriptReference/, verfügbar am 17.09.2023, 19:00

[Mücke, 2007] Mücke, F.: Analyse GPU-basierter Feature Tracking Methoden für den Einsatz in der Augmented Realität (Dissertation, Angewandte Informatik), Universität Augsburg, 2007

Anlagen

Anlagen, Teil 1

Abbildung 56: Aufbau Kreis-Partikelsystem

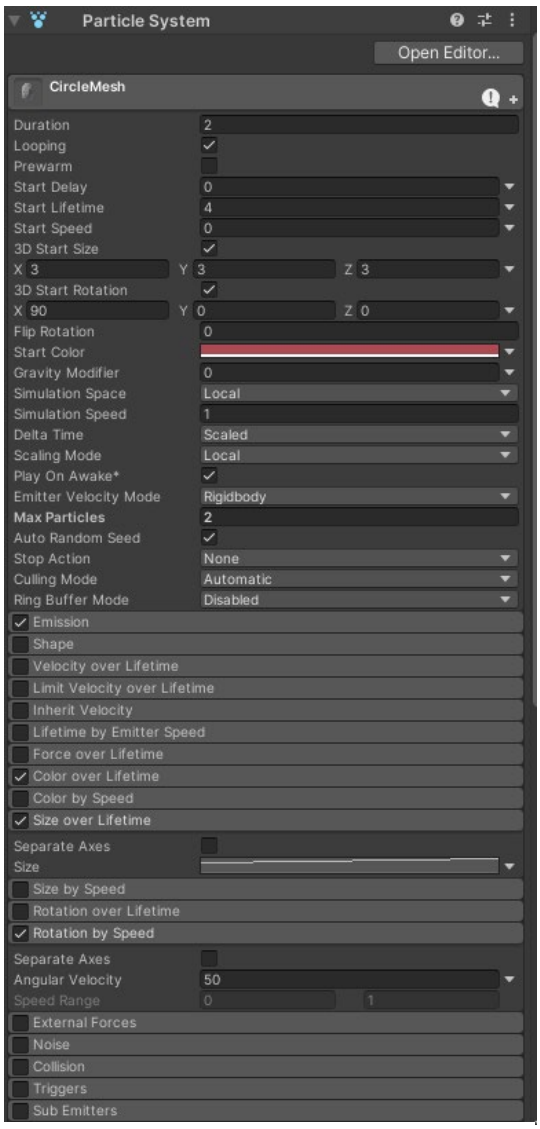

Abbildung 55: Aufbau Mesh-Partikelsystem 01

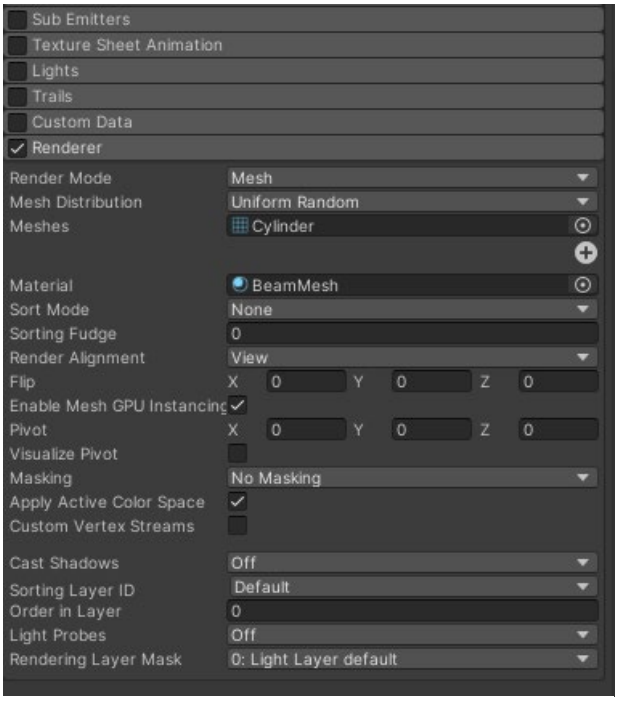

Abbildung 59: Ausbau Mesh-Partikelsystem 02

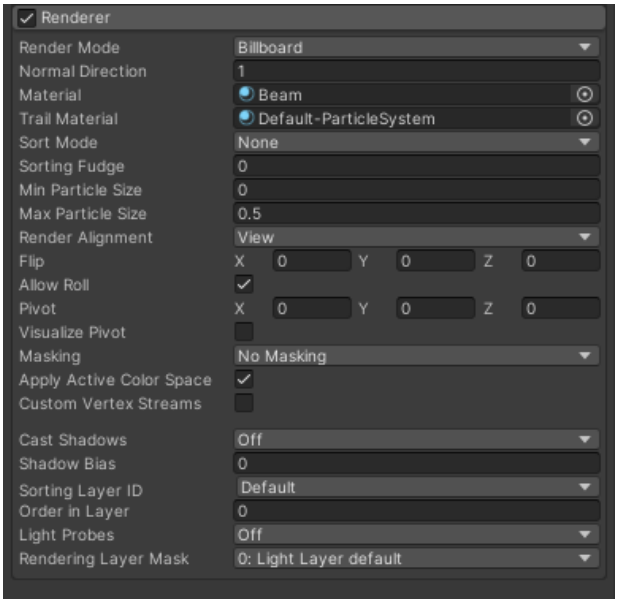

Abbildung 57: Aufbau Partikel mit Trail Partikelsystem 03

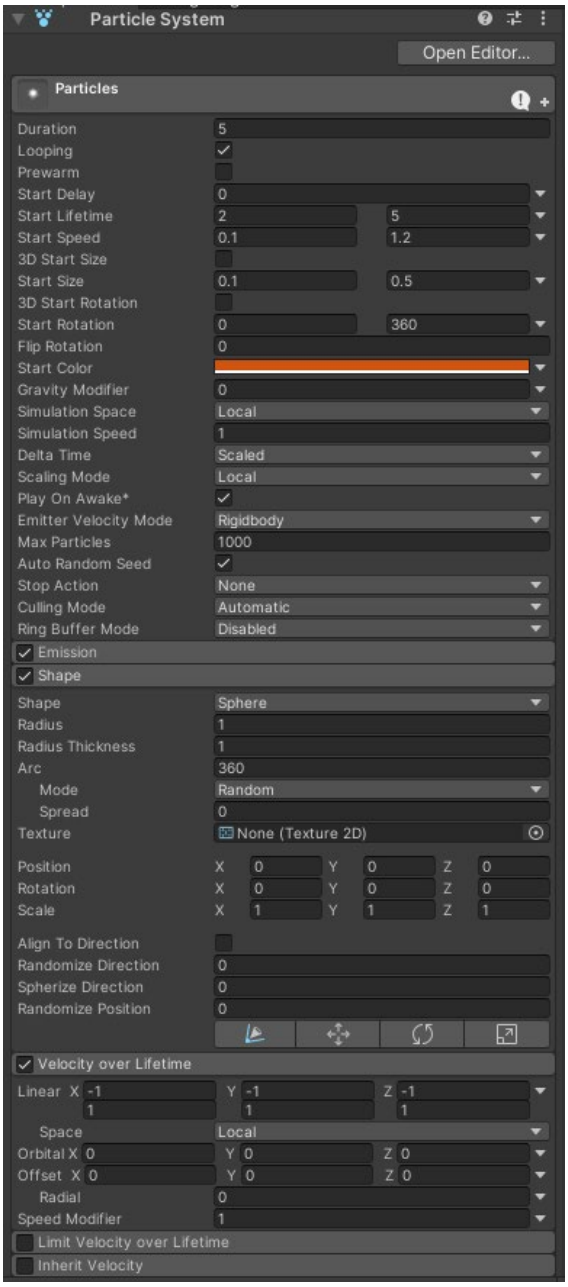

Abbildung 58: Aufbau Partikel mit Trail Partikelsystem 01

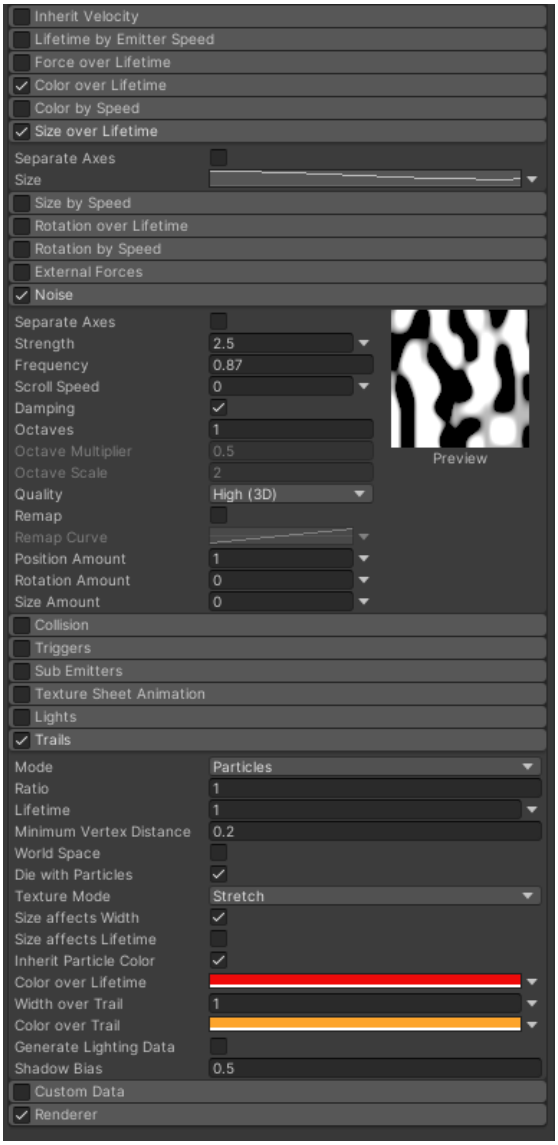

Abbildung 61: Aufbau Partikel mit Trail Partikelsystem 02

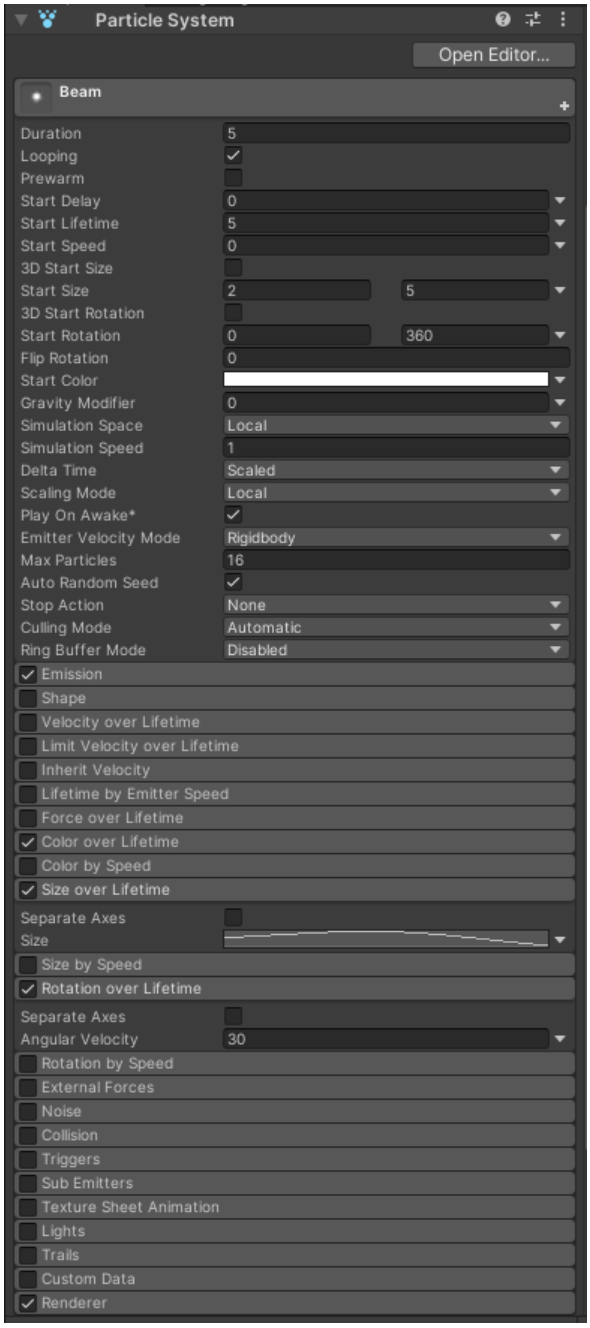

Abbildung 60: Aufbau Lichtpunkt-Partikelsystem

Abbildung 63: Aufbau Rauch-Partikelsystem 01

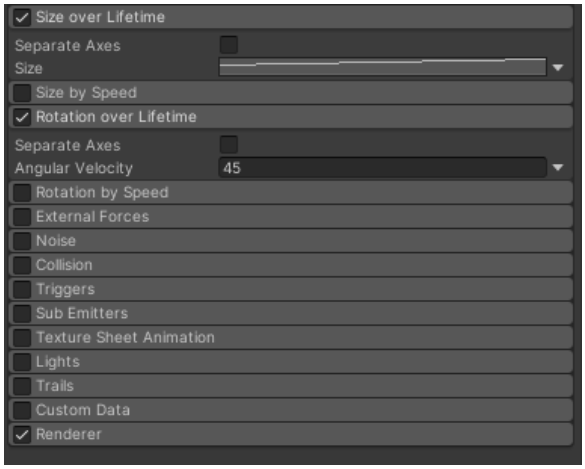

Abbildung 62: Aufbau Rauch-Partikelsystem 02

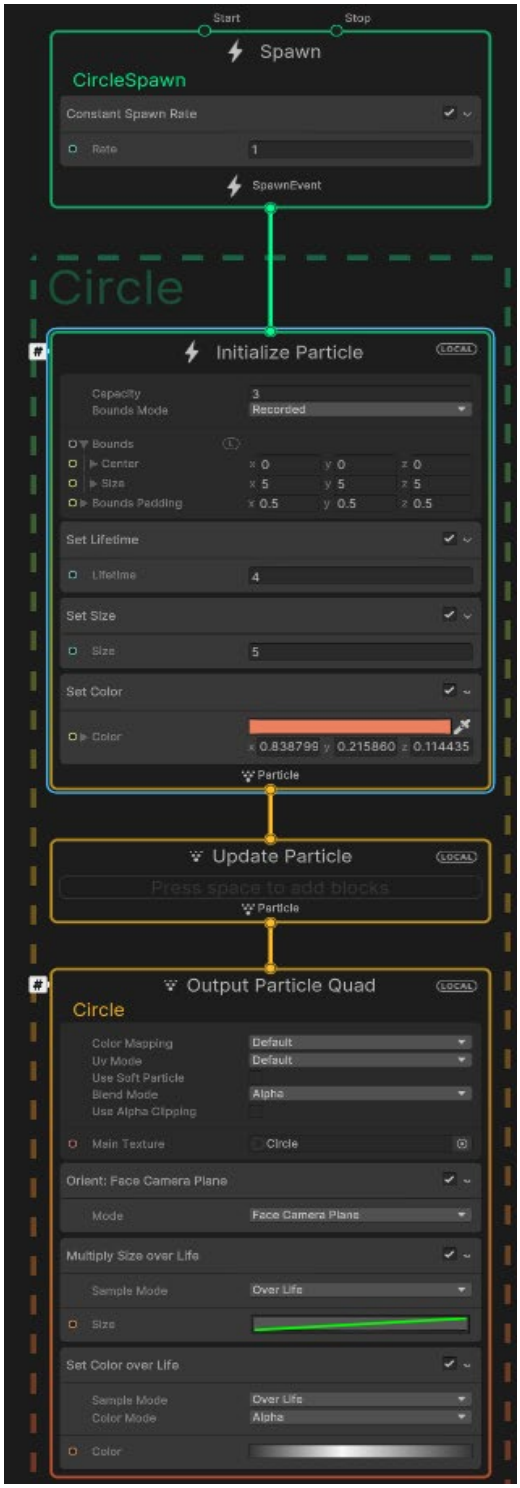

Abbildung 65: Kreis-Graph des VFX

Abbildung 64: Mesh-Graph des VFX

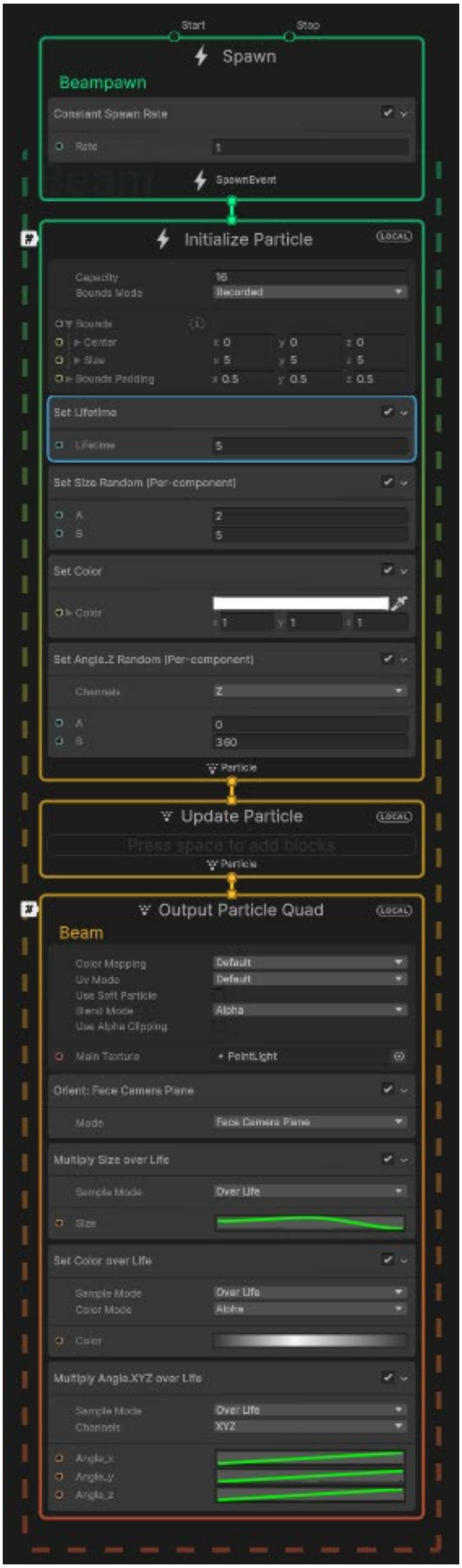

Abbildung 67: Lichtpunkt-Graph des VFX

Abbildung 66: Rauch-Graph des VFX

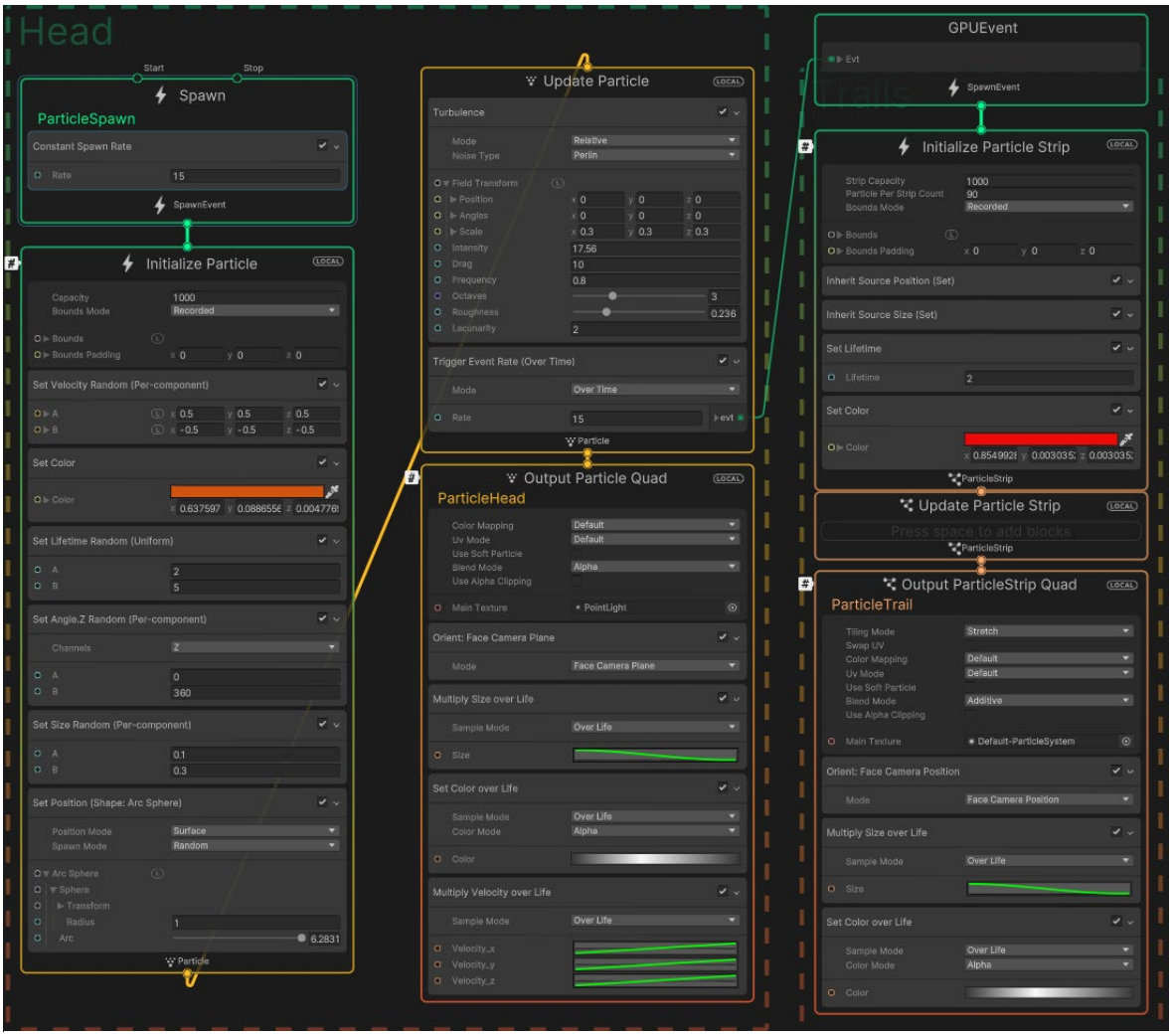

Abbildung 68: Partikel mit Trail-Graph des VFX

```
public class ParticlePool : MonoBehaviour
{
    private ObjectPool<GameObject> _pool;
    private Coroutine deactivateGameobject;
    private float destroyTime = 10f;

    Event function
    private void Start()
    {
        _pool = GameObject.FindGameObjectWithTag("MainCamera").GetComponent<PoolingSystem>()._pool;

    }

    Event function
    private void OnEnable()
    {
        deactivateGameobject = StartCoroutine( routine: DeactivateGameobject());
    }

    Frequently called  1 usage
    private IEnumerator DeactivateGameobject()
    {
        float time = 0f;
        while (time < destroyTime)
        {
            time += Time.deltaTime;
            yield return null;
        }

        if (_pool != null)
        {
            _pool.Release(gameObject);
        }
        else
        {
            Debug.LogError( message: "Kein PoolingSystem");
        }
    }
}
```

Abbildung 69: Deaktivierungs-Skript des Pooling-Systems

```
IEnumerator SpawnParticle()
{
    int particleCount = 0;
    Vector3 position = new Vector3(transform.position.x, transform.position.y, transform.position.z + 10f);
    int increase = 0;
    float xPosition = transform.position.x;
    while (particleCount <= 99)
    {
        if (increase <= 10)
        {
            position.x += 10;
            increase += 1;
        }
        else
        {
            position.z += 10;
            position.x = xPosition;
            increase = 0;

        }
         //with Pooling-System
        _poolingSystem.spawnPosition = position;
        _poolingSystem._pool.Get();

        //without Pooling-System
       // Instantiate(original, position, transform.rotation);

        particleCount += 1;
        allParticleCount += 1;
    }

    time = 0f;
    yield return null;
}
}
```

Abbildung 70: Coroutine zur Anzeige der visuellen Effekte

Anlagen, Teil 2

```
public class FPSDisplay : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI text;  Changed in 2 assets
    private float pollingTime = 1f;
    private float time;
    private int frameCount;

    // Update is called once per frame
    Event function
    void Update()
    {
        time += Time.deltaTime;

        frameCount++;

        if (time >= pollingTime)
        {
            int frameRate = Mathf.RoundToInt(frameCount / time);
            text.text = frameRate.ToString() + " FPS";

            time -= pollingTime;
            frameCount = 0;
        }
    }
}
```

Abbildung 71: Berechnung der Bildrate

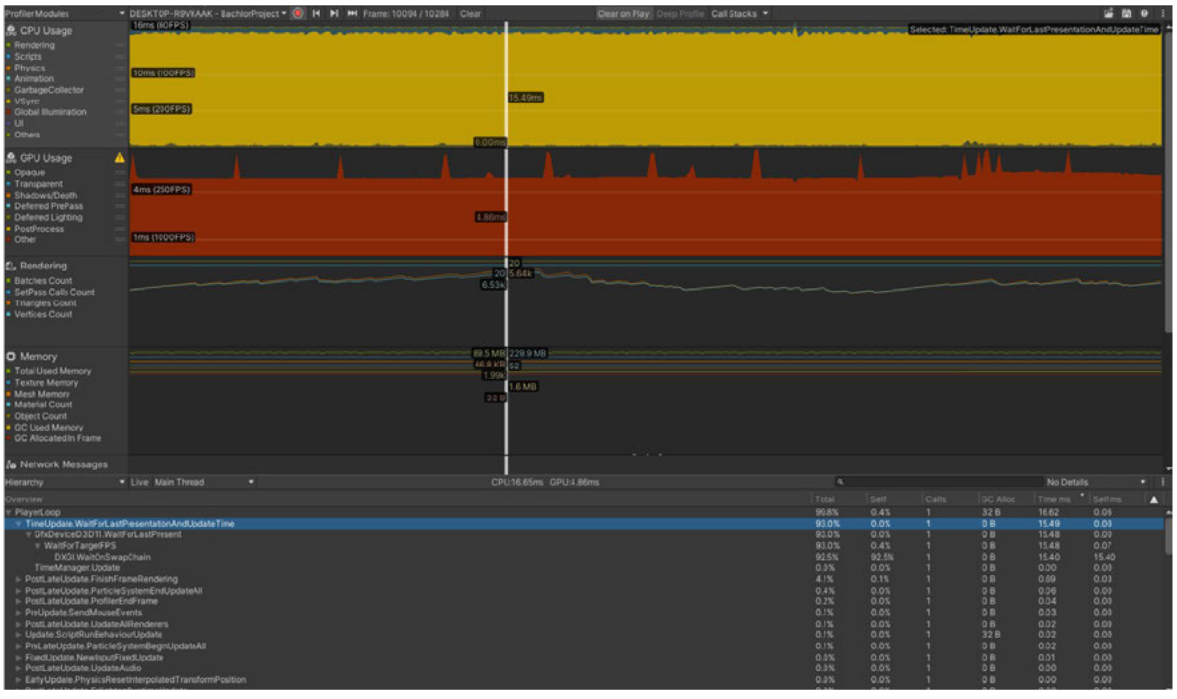

Abbildung 72: CPU-Auslastung des originalen visuellen Effektes

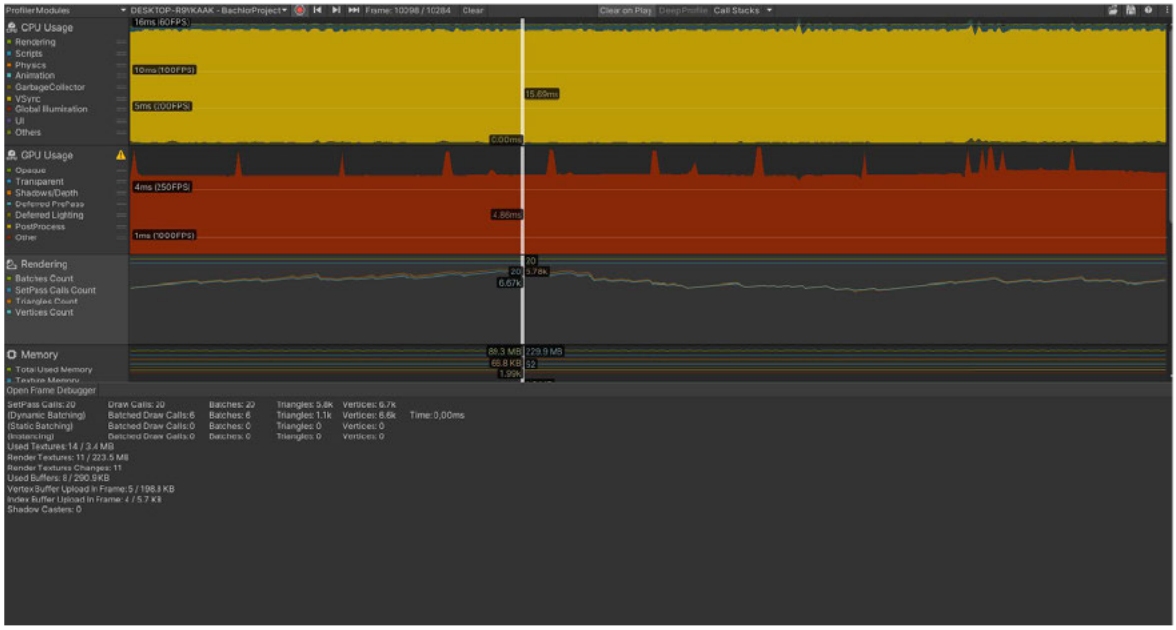

Abbildung 73: Frame Debugger des originalen visuellen Effektes

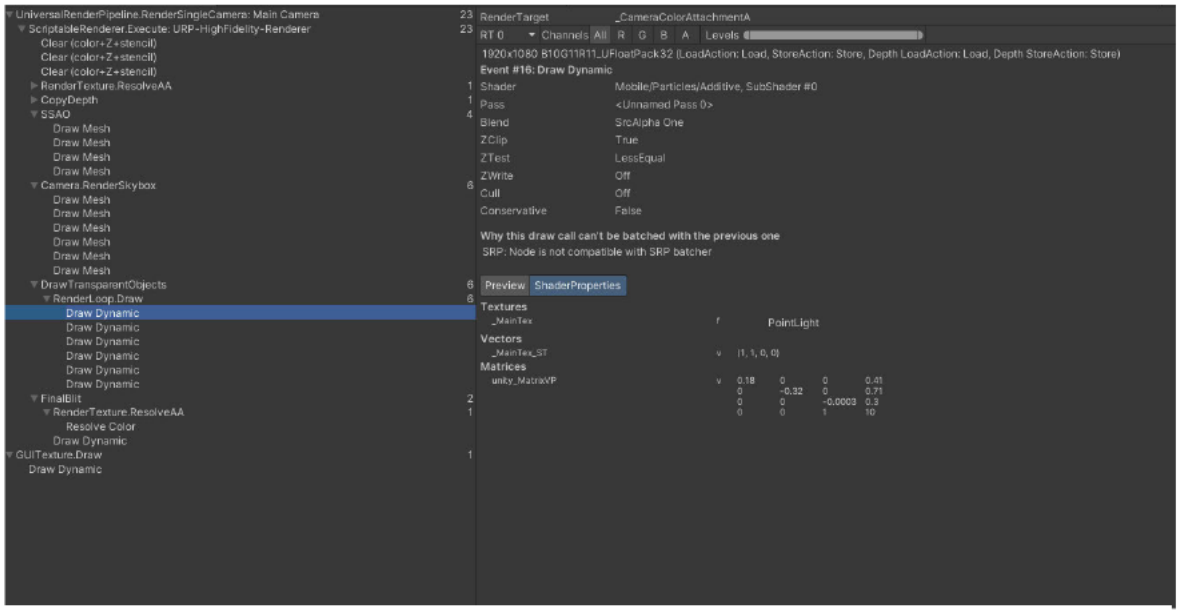

Abbildung 74: Rendering-Teil des originalen visuellen Effektes

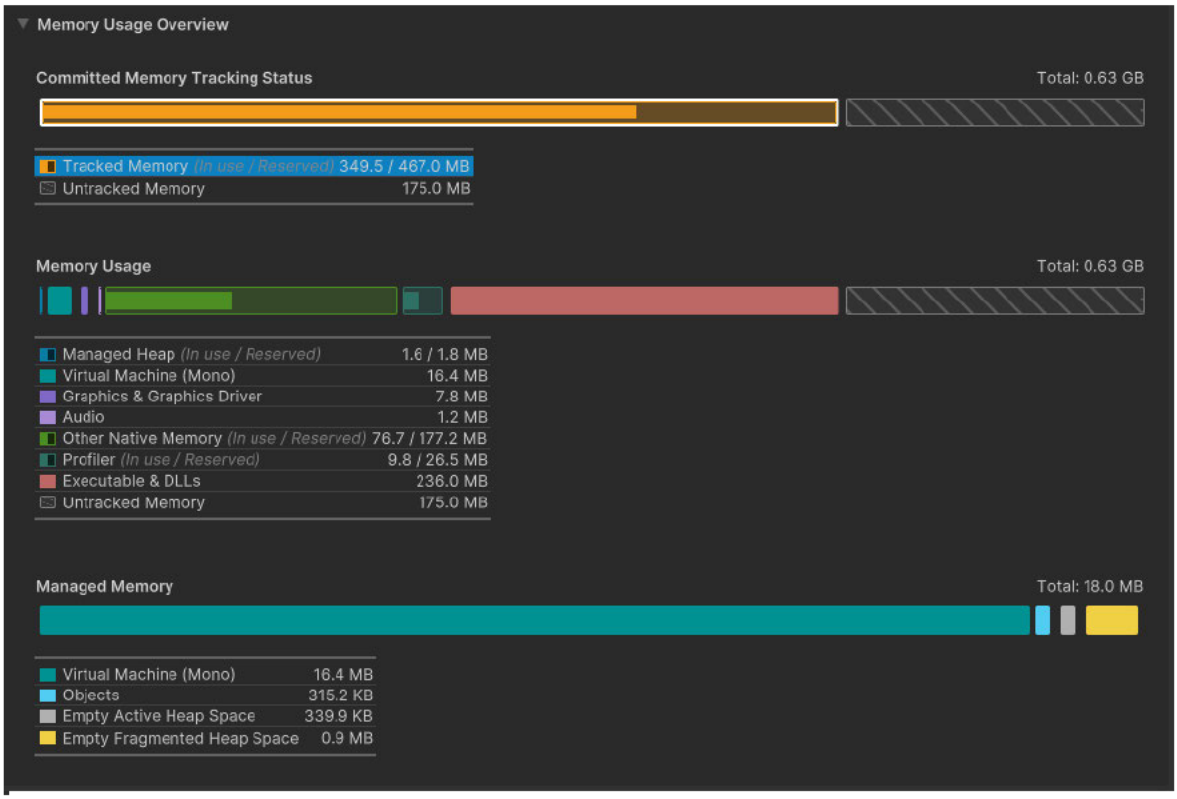

Abbildung 76: Übersicht der Speichernutzung im Memory Profiler des originalen visuellen Effektes

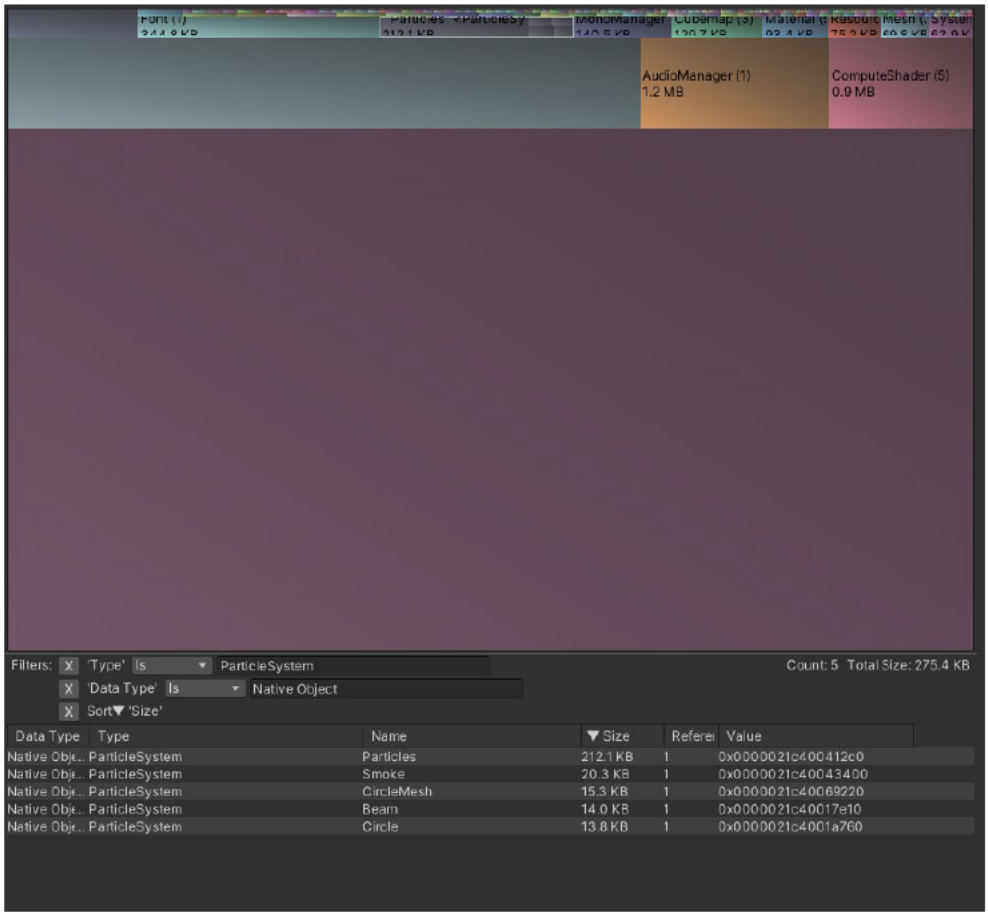

Abbildung 75: Partikelsysteme in der Baumstruktur des Memory Profilers des originalen visuellen Effektes

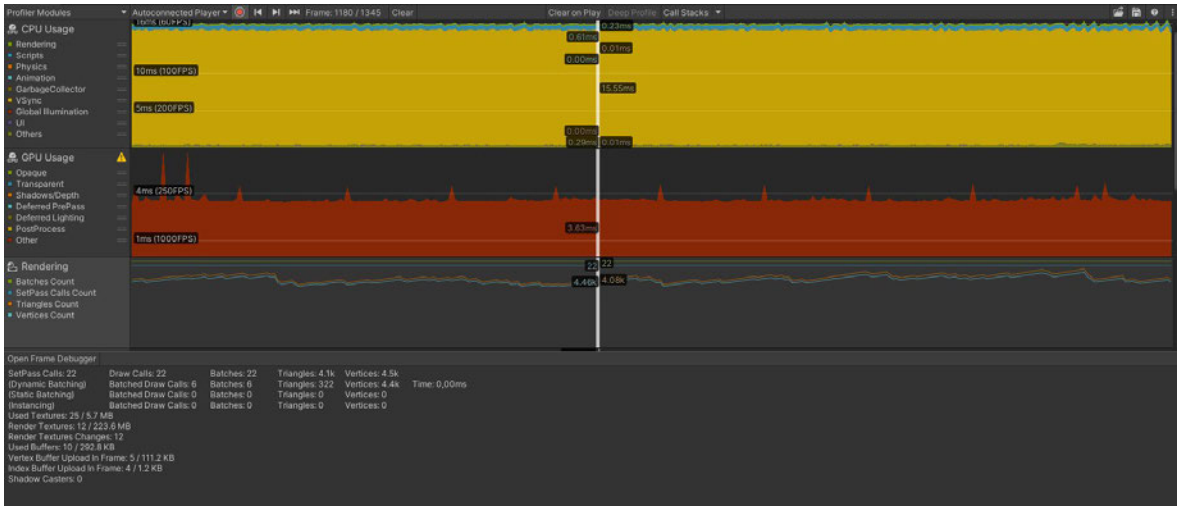

Abbildung 77: Rendering-Teil bei Verwendung von ausschließlich Billboards

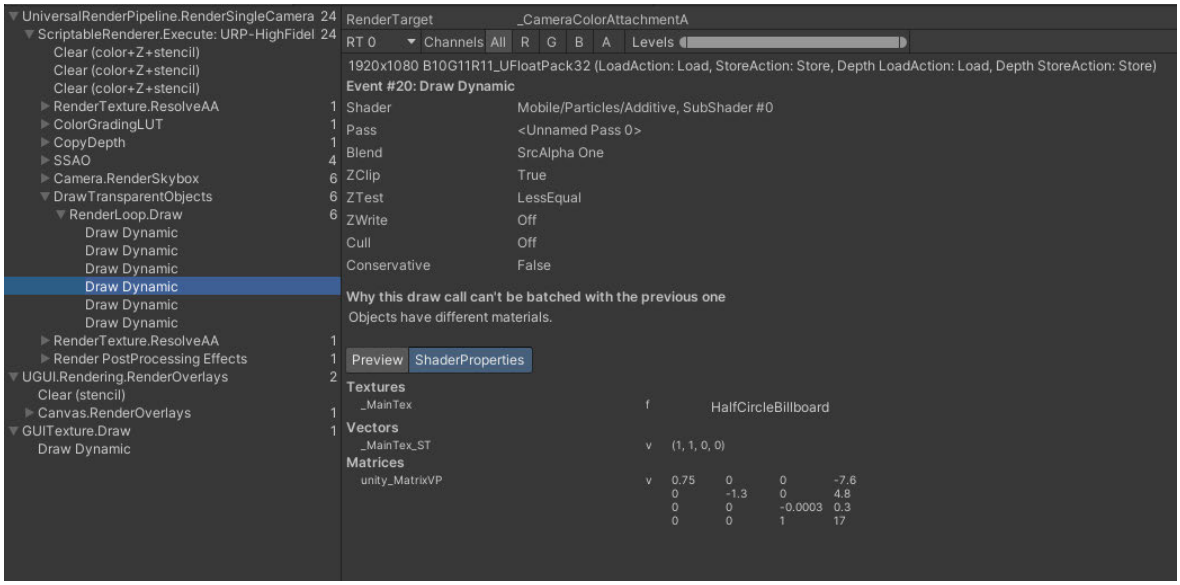

Abbildung 78: Frame Debugger bei Verwendung von ausschließlich Billboards

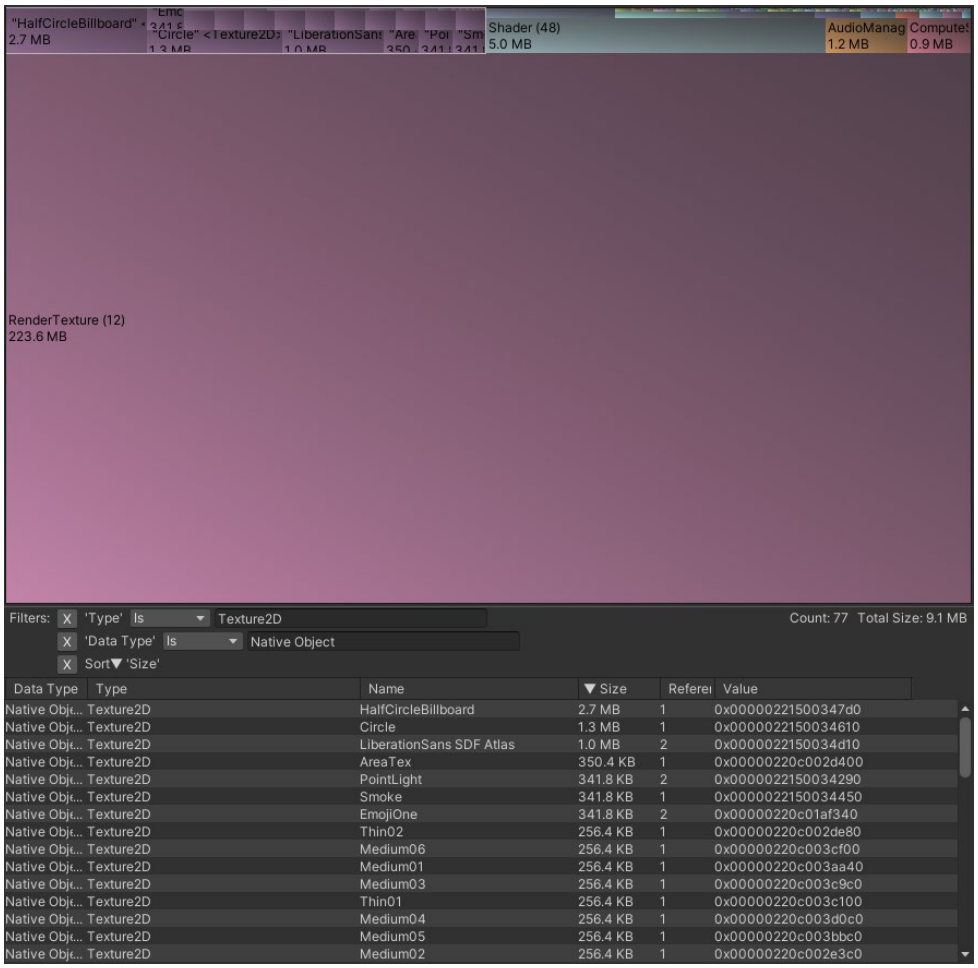

Abbildung 79: 2D-Texturen in der Baumstruktur des Memory Profilers bei Verwendung von ausschließlich Billboards

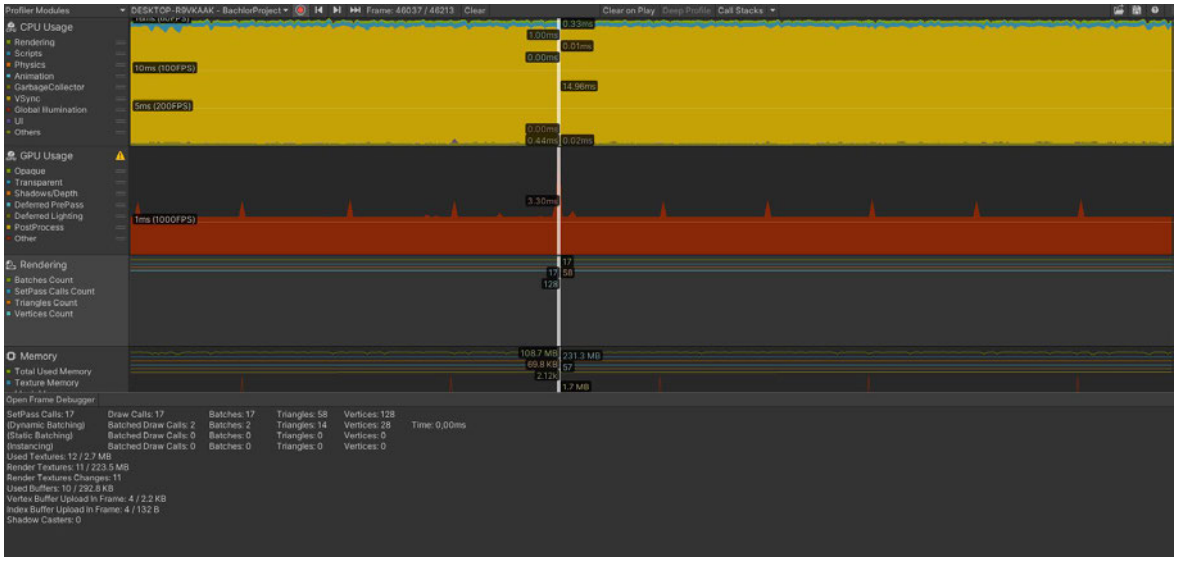

Abbildung 81: Rendering-Teil LOD-System Level 4

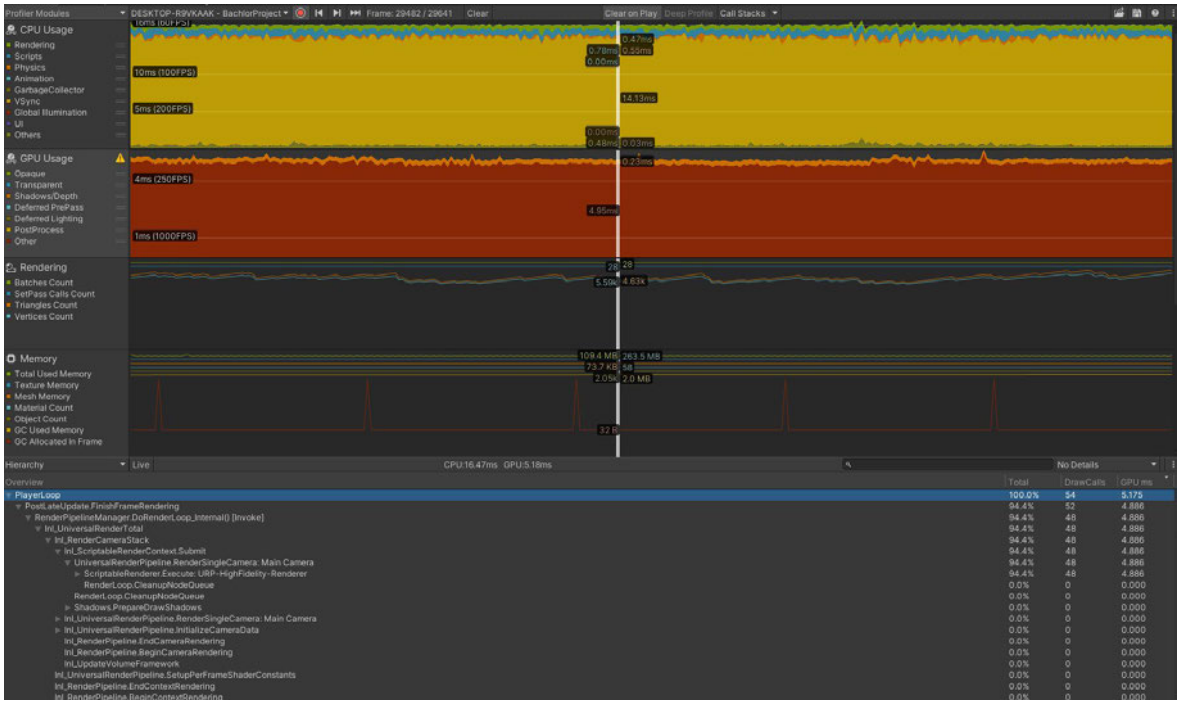

Abbildung 80: GPU-Auslastung des Culling-System ohne blockierte Sicht

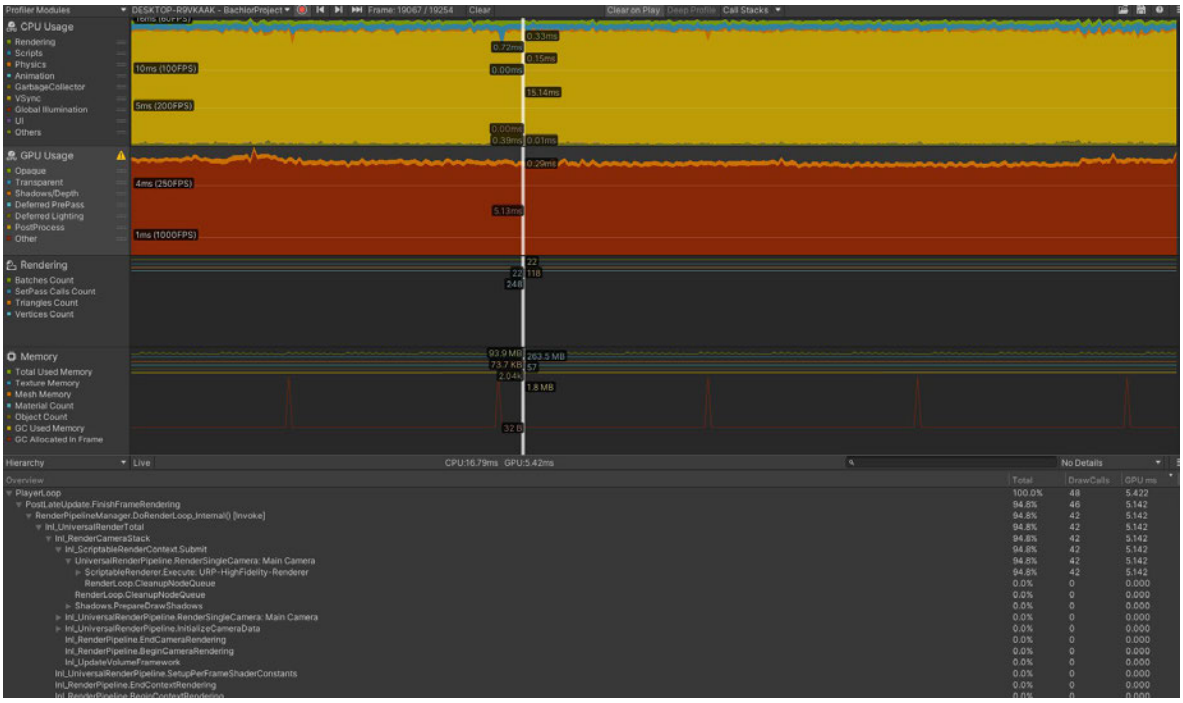

Abbildung 82: GPU-Auslastung des Culling-Systems mit blockierter Sicht

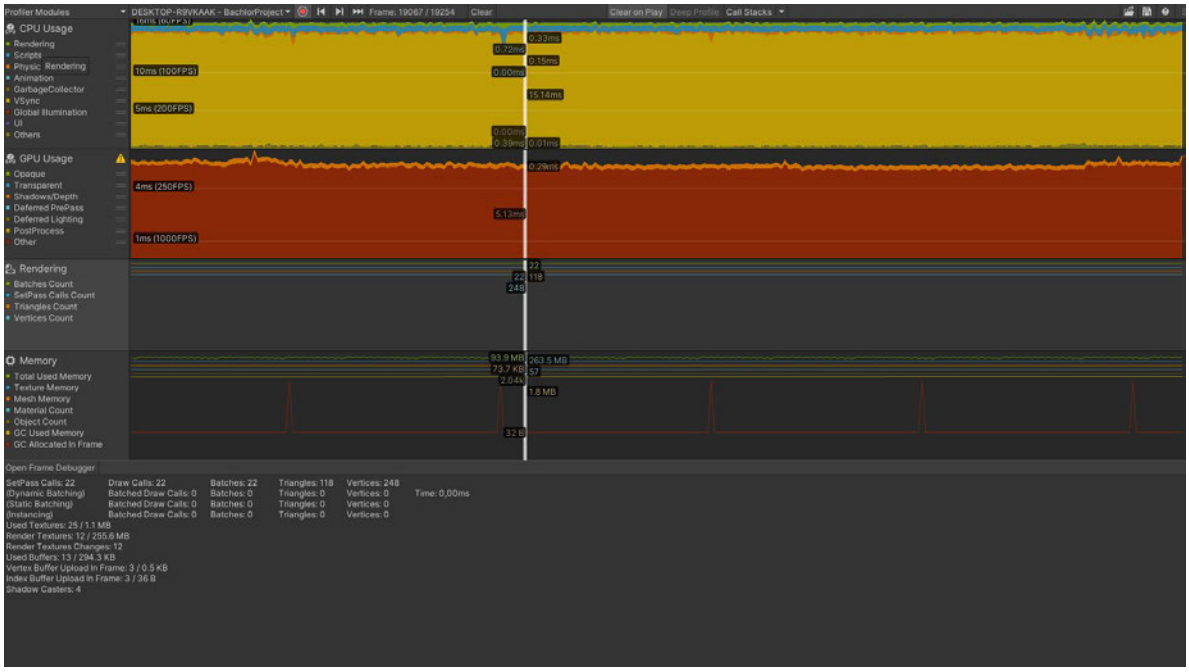

Abbildung 83: Rendering-Teil des Culling-Systems mit blockierter Sicht

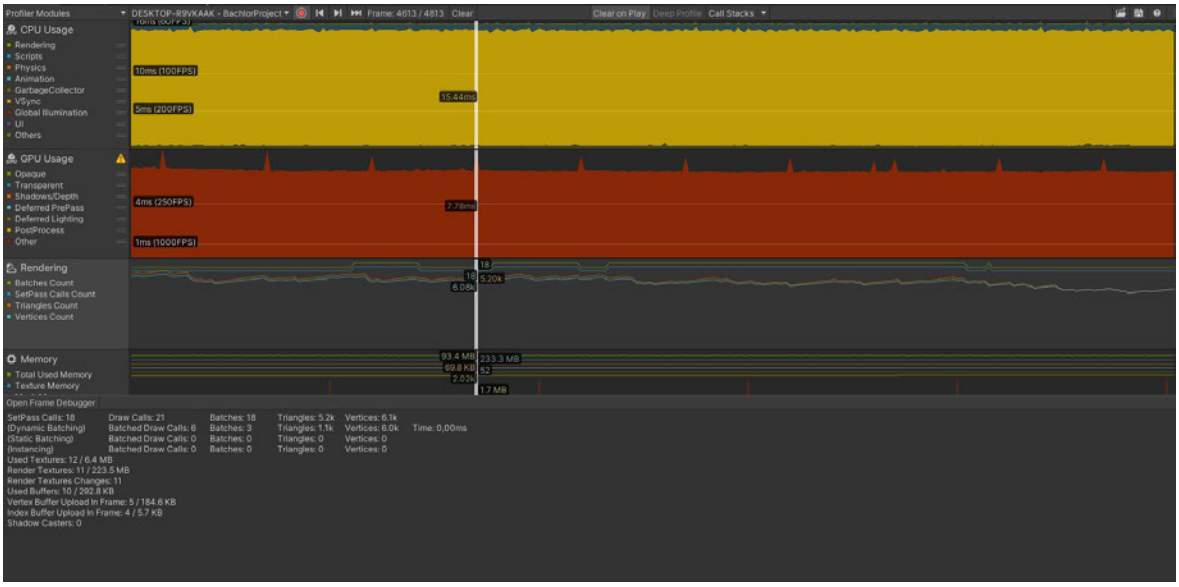

Abbildung 85: Rendering-Teil beim Batching

Abbildung 84: Frame Debugger beim Batching

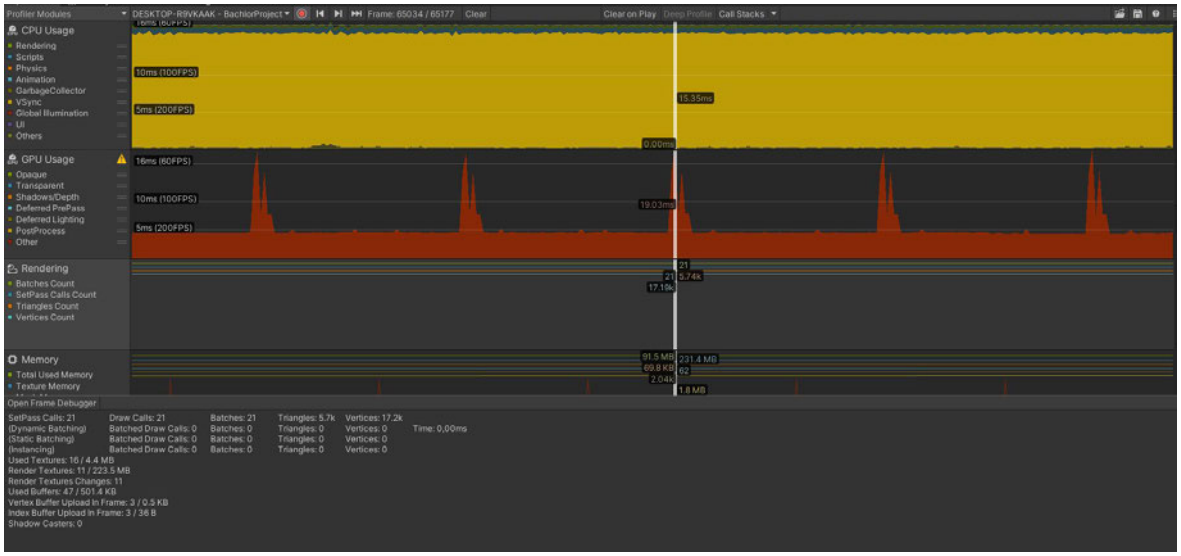

Abbildung 86: Rendering-Teil des Visual Effect Graphes

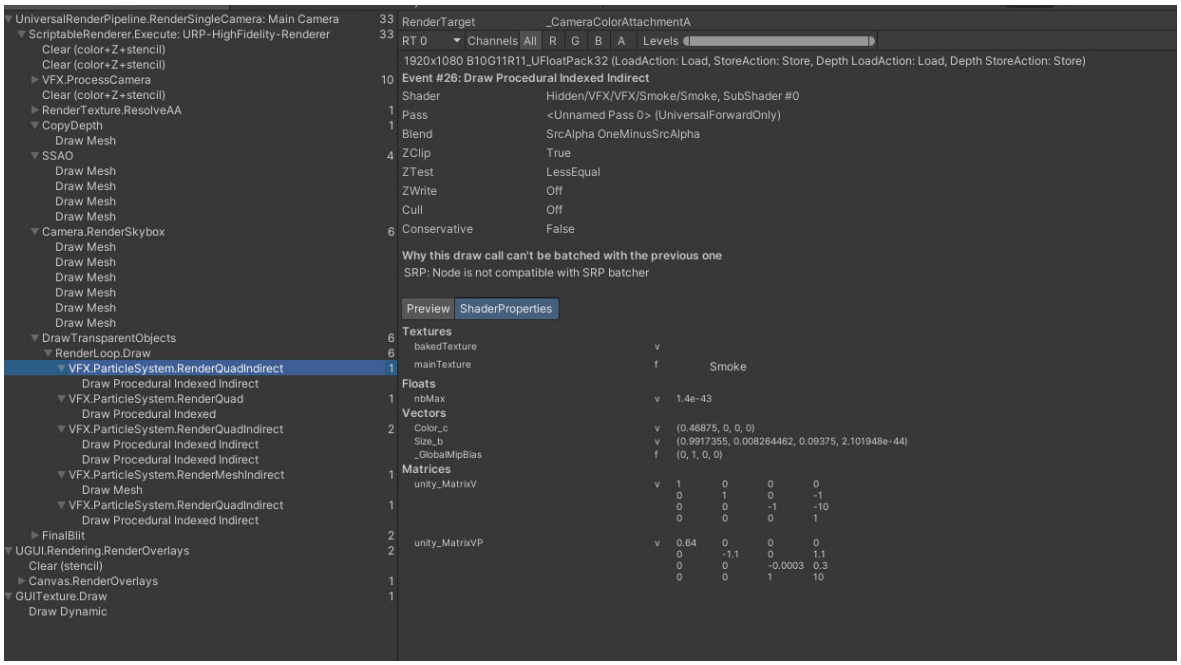

Abbildung 87: Frame Debugger des Visual Effect Graphes

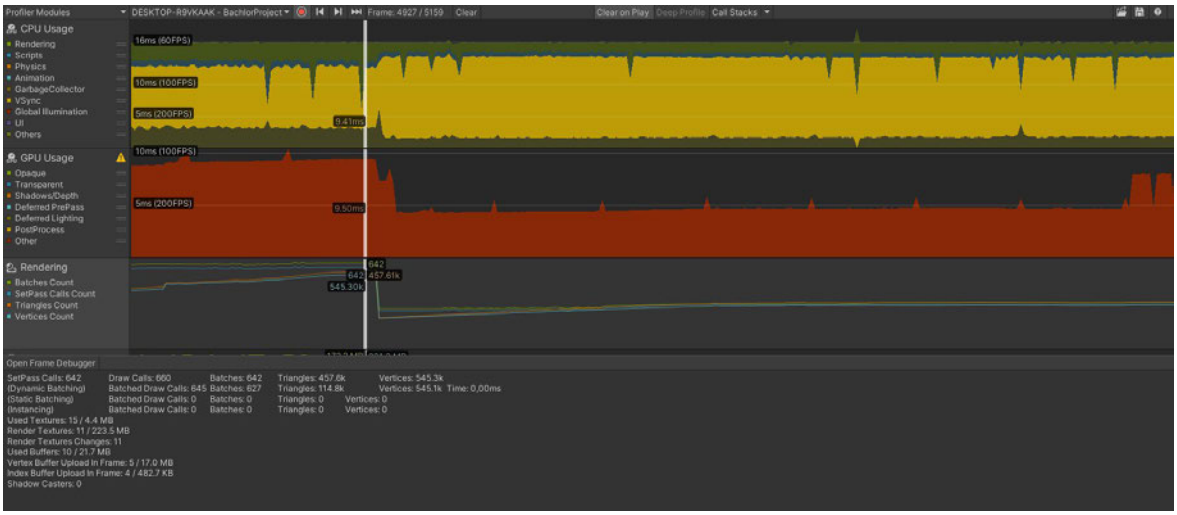

Abbildung 89: Rendering-Teil bei 200 Effekten bei der originalen Instanziierung

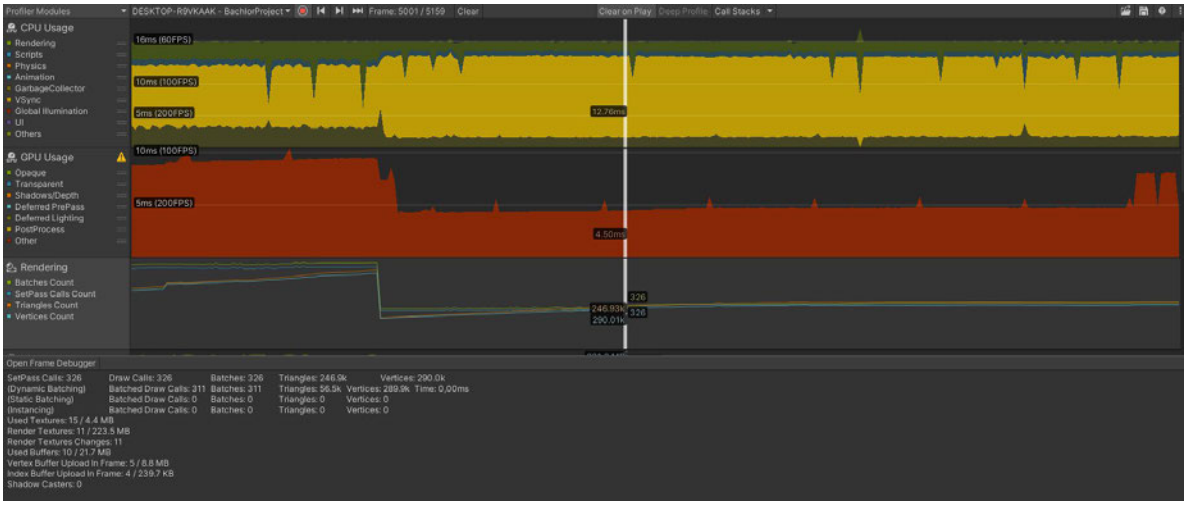

Abbildung 88: Rendering-Teil bei 100 Effekten bei der originalen Instanziierung

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, den 19.09.2023

Annabell Schütze