Bachelorarbeit

Dynamische Billboard-Vereinfachungen für große Szenen

Wilhelm Schnepp
Studiengang: Informatik
7. Fachsemester
Matrikelnummer: 466646

Datum: 5. Dezember 2018

Eingereicht bei:
Prof. Dr. Thorsten Grosch
Institut für Informatik
Abteilung für Graphische Datenverarbeitung und Multimedia
Technische Universität Clausthal

Zweitgutachter:
Prof. Dr.-Ing. Michael Prilla
Institut für Informatik
Abteilung für Human-Centered Information Systems
Technische Universität Clausthal

Betreuer:
M.Sc. Johannes Jendersie
Institut für Informatik
Abteilung für Graphische Datenverarbeitung und Multimedia
Technische Universität Clausthal

Abstract

Sollen große Szenen mit vielen komplexen Objekten dargestellt werden, so muss die Geometrie der Szene reduziert werden. Denn selbst modernste Grafikkarten bieten nicht genügend Rechenleistung, um eine große Szene mit vollem Detaillevel darstellen zu können. Zur Komplexitätsreduktion der Geometrie existieren eine Vielzahl verschiedener Ideen, wobei Billboards eine der Populärsten sind. Jedoch erhalten diese Tiefeninformationen nicht, wodurch die Billboards nur bei sehr großer Distanz zur Kamera gut aussehen. Deswegen werden in dieser Arbeit G-Buffer-Billboards entwickelt, welche unter anderem besagte Tiefeninformationen erhalten und dadurch eine bessere Bildqualität auch bei geringerer Distanz zur Kamera ermöglichen.

Inhaltsverzeichnis

1 Einleitung

Eine der Anwendungen der Computergrafik ist das Zeichnen komplexer Szenen oder Modelle. Dabei ergibt sich jedoch häufig das Problem, dass die zu zeichnende Geometrie zu komplex ist. Selbst modernste Grafikhardware bietet oft nicht genügend Rechenleistung, um diese Modelle so schnell zu zeichnen, dass eine Interaktion mit dem Nutzer möglich ist. Abhilfe schaffen Verfahren zur Detailreduktion der Geometrie, besonders solcher Objekte, die weit vom Betrachter entfernt liegen und dementsprechend auch nur sehr wenige Pixel des endgültigen Bildes ausmachen.

In dieser Arbeit soll ein Verfahren präsentiert werden, welches einen populären Ansatz zur Detailreduktion, die Verwendung von Billboards, mit in moderner Grafikhardware verfügbarer Technologie kombiniert. Dadurch kann das Aussehen einzelner Objekte der Geometrie zu einem bestimmten Zeitpunkt exakt eingefangen und anschließend wiederverwendet werden, ohne diese Objekte jedesmal erneut zeichnen zu müssen.

Das zweite Kapitel widmet sich zuerst einigen allgemeinen Grundlagen der Computergrafik, stellt anschließend die zwei verschiedenen Verfahren des Forward und Deferred Renderings vor und erklärt daraufhin verschiedene Ansätze zur Geometriereduktion. Im dritten Kapitel wird der Ansatz der Billboards weiterentwickelt, indem diese anstatt normaler Texturen, mit einem G-Buffer wie beim Deferred Rendering ausgestattet werden. Dadurch können Informationen wie bspw. die Tiefe erhalten werden. Im vierten Kapitel wird anschließend die Performance der G-Buffer-Billboards evaluiert und im fünften Kapitel dann einige aufgetretene Probleme erklärt und Lösungen für diese vorgeschlagen.

2 Grundlagen

In diesem Kapitel werden zuerst einige allgemeine Grundlagen der Computergrafik erklärt.

Anschließend werden zuerst das Forward und das Deferred Rendering einander gegenübergestellt und darauf folgend verschiedene Ansätze zur *Level of Detail* Re-

duktion präsentiert.

2.1 Allgemeine Grundlagen

In der Computergrafik werden aus meist dreidimensionalen Daten durch geeignete Verarbeitung, Transformation und Beleuchtung zweidimensionale Bilder oder Bildsequenzen erzeugt. Dieser Prozess wird *Rendern* genannt.

2.1.1 Offline- und Echtzeitrendering

Es wird zwischen sogenanntem *Offline-* und *Echtzeit*rendering unterschieden. Beim Offlinerendering muss die Berechnung der Einzelbilder einer Bildsequenz nicht mit einer interaktiven Framerate stattfinden, d.h. die Erzeugung eines Einzelbildes kann beliebig viel Zeit einnehmen, weil keine Reaktion auf Nutzereingaben nötig ist. Ein Einsatzgebiet für solches Offlinerendering ist die Produktion von CGI[1]-Inhalten für Filme wie bspw. „Avatar“.

Demgegenüber sollen oder müssen für viele Anwendungen der Computergrafik die Nutzer mit der dargestellten Szene interagieren können. Dazu sind höhere Framerates nötig, wobei eine Framerate ab ca. 6 Bildern pro Sekunde (Frames per Second, FPS) als *interaktiv* bezeichnet wird. Eine Spezialform der interaktiven Framerates stellt das Echtzeitrendering dar. Hier sollen die Framerates nicht nur interaktiv sein, sondern auch als Bildsequenz wahrgenommen werden, wofür mehr als ca. 15 FPS nötig sind (vgl. [Ake+18, p. 1]). Eines von vielen Einsatzgebieten dieser Technik sind Videospiele, wobei diese jedoch Framerates von sogar 30 oder 60 FPS oder mehr anstreben.

Bei solch hohen Framerates wird die Zeit, die für die Berechnung eines Einzelbildes zur Verfügung steht, stark eingeschränkt. Bei 60 FPS stehen beispielsweise pro Frame zur Berechnung nur 16,6 ms zur Verfügung. Deshalb sind für das Echtzeitrendering trotz immer leistungsfähigerer Hardware sehr effiziente Algorithmen notwendig.

[1] Computer Generated Imagery, Computergenerierte Bilder/Szenen

2.1.2 Vertices

Zumeist soll in dieser Zeit eine (komplexe) dreidimensionale Szene dargestellt, beleuchtet und eventuell noch mit Post-Processing-Effekten verbessert werden. Diese Szene besteht dabei aus vielen einzelnen Datenpunkten, den *Vertices* (Singular: Vertex), welche bspw. Position, Farbe und Normale jeweils eines Punktes dieser Geometrie angeben. Diese Vertices werden später zu einfachen Formen (meistens Dreiecken), den *Primitiven*, verbunden und bilden dadurch die Oberflächen der zu zeichnenden Geometrie. Andere übliche Typen von Primitiven sind bspw. Punkte, Linien oder Rechtecke (welche Quads genannt werden).

2.1.3 Kamera

Für die Betrachtung der Szene wird eine *Kamera* definiert. Im Wesentlichen besteht die Kamera aus einer Position im Weltkoordinatensystem, einer Blickrichtung und einem sogenannten Up-Vektor, der die vertikale Ausrichtung der Kamera in Relation zu den anderen zwei Vektoren bestimmt.

Aus den drei Vektoren lässt sich jetzt ein neues Koordinatensystem, das *Kamerakoordinatensystem*, sowie eine Transformationsmatrix zur Umrechnung von Welt- in Kamerakoordinaten konstruieren. In diesem ist die Kamera immer im Ursprung zentriert, und guckt bei OpenGL per Definition in die negative z-Richtung.

Zur perspektivischen Verzerrung der Szene wird eine zweite Transformationsmatrix mit einer zusätzlichen vierten Komponente bestimmt. Indem später die ersten drei Komponenten durch die vierte Komponente geteilt werden, kann eine perspektivische Verzerrung erreicht werden. Dafür wird ein Frustum konstruiert, welches den sichtbaren Bereich darstellt. Dieses wird definiert durch die *Near*- und *Far*-Plane, sowie einen Öffnungswinkel der Kamera. Die Near-Plane ist dabei die der Kamera zugewandte Ebene des Frustums, auf welche das Bild der Szene projiziert wird, während die Far-Plane die der Kamera abgewandte Seite bildet und damit die maximale Sichtweite einschränkt. Die Ebenen, die in den Grafiken 6 und 7 (Kapitel 3.4) mit $-n$ bzw. $-f$ bezeichnet sind, sind genau diese Near- und Far-Plane der Kamera. Über den Öffnungswinkel der Kamera kann die Größe dieser Ebenen kontrolliert werden, während über deren z-Koordinate die Entfernung von der Kamera eingestellt wird.

2.1.4 Axis Aligned Bounding Box

In vielen verschiedenen Situationen muss die Position eines Objektes in der Szene abgeschätzt werden können, als Beispiel für eine Anwendung seien hier Sichtbarkeits- oder Kollisionstests genannt.

Da die Überprüfung jedes einzelnen Vertex eines Objektes viel zu aufwändig ist, werden für diesen Zweck sogenannte *Bounding Volumes* eingesetzt. Dies sind einfache dreidimensionale Formen, die so um das Objekt herumgelegt werden, dass dieses komplett in dem Bounding Volume enthalten ist.

Eines der meistgenutzten Bounding Volumes ist die *Axis Aligned Bounding Box*, kurz AABB. Dies ist ein an den Achsen des Koordinatensystems ausgerichteter Quader. Der Vorteil dieser AABB ist ihre einfache Berechnung, indem aus allen Vertices des Objektes für die drei Achsen getrennt Minimum und Maximum bestimmt werden, sowie dessen geringer Speicherbedarf. Denn es müssen nur das zuvor bestimmte Minimum und Maximum gespeichert werden.

Wird ein Objekt von einem in ein anderes Koordinatensystem transformiert, so muss auch dessen AABB in diesem Koordinatensystem erneut bestimmt werden. Da die Iteration über alle Vertices des Objektes zu aufwändig ist, werden stattdessen nur die acht Eckpunkte der alten Bounding Box transformiert und aus diesen transformierten Punkten ein neues Minimum und Maximum bestimmt, die dann wiederrum eine AABB in dem Koordinatensystem bilden.

Da nur die acht Eckpunkte der alten Bounding Box berücksichtigt werden, beinhaltet diese Box zwar das gesamte Objekt, ist jedoch nicht optimal. Es kann also passieren, dass eine auf diese Weise bestimmte Bounding Box sehr viel größer als das Objekt in diesem Koordinatensystem ist.

Ein solcher Fall ist in Abbildung 1 dargestellt. Die Bounding Box des Weltkoordinatensystems (Blau) der Vespa wurde mit dem beschriebenen schnellen Algorithmus in das Kamerakoordinatensystem transformiert (Orange). Zu Vergleichszwecken wurde außerdem durch Iteration über alle Vertices der Vespa die optimale Bounding Box im Kamerakoordinatensystem bestimmt (Rot).

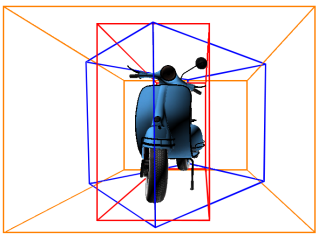

Abbildung 1: *Die mit dem schnellen Algorithmus aus der Welt-AABB (Blau) in das Kamerakoordinatensystem transformierte AABB des Objektes (Orange) ist deutlich größer als die exakt bestimmte AABB der Vespa im Kamerakoordinatensystem (Rot). Die exakte Bounding Box wurde durch Iteration über alle Vertices des Objektes gewonnen. Das Modell ist dem Außenbereich des „Amazon Lumberyard Bistros“ ([Lum17]) entnommen.*

2.1.5 Vertex Array Object

Die Vertices werden in einem *Vertex Array Object* gespeichert. Dies ist eine Datenstruktur in OpenGL, die es erlaubt die verschiedenen Informationen der Vertices kompakt in einem *Vertexbuffer* abzuspeichern. Die Vertices eines Vertex Array Objects einer Szene können nun mit einem sogenannten *Draw-Call* gezeichnet werden. Dieser Funktionsaufruf erteilt der Grafikkarte den Auftrag, die als Argumente übergebenen Vertices des Vertex Array Objects zu zeichnen. Die Reihenfolge der Vertices bestimmt dabei, wie diese später zu Primitiven verbunden werden. Deshalb muss ein in mehreren Primitiven verwendeter Vertex auch mehrfach in dem Buffer abgespeichert werden.
Um dies zu Verhindern besitzt ein Vertex Array Object einen optionalen *Indexbuffer*. In diesem werden die Indices der zu zeichnenden Vertices gespeichert. Hier kann der Index eines mehrfach verwendeten Vertex auch mehrfach verwendet werden. Anschließend wird statt direkt die Vertices zu Zeichnen, stattdessen der Indexbuffer gezeichnet, wobei die Reihenfolge der Indices im Indexbuffer nun die Reihenfolge, in der Vertices zu Polygonen verbunden werden, angibt.

2.1.6 Grafikpipeline

Die Transformation von Vertices zu der gezeichneten Szene erfolgt in einer sogenannten Grafikpipeline, von der heutzutage große Teile programmierbar sind.

In einem ersten Schritt werden die Vertices aus dem Weltkoordinatensystem

zunächst in das Kamerakoordinaten- und dann in das Projektionskoordinatensystem transformiert. Dies geschieht im *Vertex-Shader*.

Ein Shader ist ein auf der Grafikkarte vielfach parallelisiert ausgeführtes Programm. Es existieren verschiedene Arten von Shadern, die unterschiedliche Funktionalitäten erfüllen, wobei die wichtigsten der schon erwähnte Vertex-Shader und der *Fragment*-Shader sind. Dieser ist später für die Beleuchtung und Texturierung der Geometrie zuständig. Zusätzlich existieren noch vier weitere Shaderarten: zwei verschiedene Tessellation- sowie Geometry- und Compute-Shader.

Mit Tessellation kann bestehende Geometrie verfeinert werden, und das Verfahren stellt somit eine spezielle Form von *Subdivision* dar. Dadurch kann bspw. ein relativ niedrig aufgelöstes Modell durch Tessellation und Displacement Mapping mit hohem Detaillevel dargestellt werden, wie von Nießner und Loop in [NL13] demonstriert.
Der Geometry-Shader hingegen erlaubt es, Primitive nachträglich verändern, erweitern, duplizieren oder sogar weglassen zu können. So können bspw. aus Punktprimitiven Rechtecke generiert werden.
Ein Compute-Shader hingegen kann beliebige Informationen berechnen und sogar außerhalb der Grafikpipeline eingesetzt werden. Dadurch ist dieser von der verfügbaren Funktionalität vergleichbar zu OpenCL- oder CUDA-Programmen, kann jedoch bei Einsatz in der Pipeline zum Beispiel für die Berechnung von Post-Processing-Effekten verwendet werden.

Auf den Vertex-Shader folgen die Operationen *Primitive Assembly*, *Clipping* und der *Rasterizer*. Diese Stufen sind nicht frei programmierbar, können jedoch durch einige Optionen beeinflusst werden (bspw. den Typ der verwendeten Primitive).

Beim Primitive Assembly werden die Vertices, die der Vertex-Shader ausgegeben hat, zu Primitiven (meistens Dreiecken) verbunden. Ganz oder teilweise außerhalb des sichtbaren Bereiches liegende Primitive werden anschließend beim Clipping am Rand des sichtbaren Bereiches abgeschnitten. Außerdem wird die Perspektivische Division und die Viewport-Transformation durchgeführt. Der Viewport bezeichnet den aktuell genutzten Bereich des Fensters. Dieser wird angegeben durch die Position der unteren linken Ecke sowie dessen Breite und Höhe. Durch die genannten Transformationen werden die Vertices entsprechend der Kameraperspektive verzerrt und in Fensterkoordinaten umgerechnet. Das durch die Perspektivische

Division erhaltene Koordinatensystem wird als *Normalized Device Coordinates* bezeichnet, da in diesem alle sichtbaren Vertices in allen drei Achsen im Bereich [-1, 1] liegen. Der Eindruck von dreidimensionaler Tiefe wird durch eine nicht-lineare Skalierung der z-Achse erreicht.

Der Rasterizer erzeugt aus den Primitiven nun Sequenzen an Fragmenten. Ein Fragment enthält die Informationen, die nötig sind, um ein Pixel des Bildes der aktuellen Geometrie zu berechnen. Vertexinformationen werden dabei zwischen den Vertices des Primitivs interpoliert, wobei die Art der Interpolation beeinflusst werden kann.

Werden Tessellation- oder Geometry-Shader verwendet, so werden diese Programme vor Primitive Assembly, Clipping und Rasterizer ausgeführt. Jedoch wird dann bereits vorher eine reduzierte Form des Primitive Assembly genutzt, da diese Shader als Eingaben Primitive erhalten. Das Clipping und darauf folgende Stufen arbeiten in diesem Fall mit den Outputs dieser Shader weiter, nicht mit dem des Vertex-Shaders.

Nach dem Rasterizer folgt jetzt der bereits erwähnte Fragment-Shader, der die Beleuchtung und Texturierung der Fragmente übernimmt. Die Ergebnisse des Fragmentshaders werden nun nach einigen (optionalen) Tests, wie bpsw. dem Tiefentest, in einen *Framebuffer* geschrieben. Der Tiefentest entscheidet auf Basis einer vom Nutzer festgelegten Funktion, welche Objekte vorne bzw. weiter hinten in der Szene liegen und löst damit Verdeckungskonflikte zwischen Objekten auf. Ein zweiter Test ist der Scissortest, der es erlaubt den Bereich, in den ein Shader im Framebuffer zeichnen kann, einzuschränken.

Eine schematische Darstellung dieser Grafikpipeline ist in Abbildung 2 zu sehen. Alternativ ist in [CR12b] eine sehr detaillierte Map dieser Pipeline für OpenGL 4.4 zu finden, welche für das Buch „OpenGL Insights“ ([CR12a]) erstellt wurde.

2.1.7 Framebuffer

Ein Framebuffer ist eine Kollektion von verschiedenen Buffern, die durch Operationen der Grafikpipeline gefüllt werden. Ein Framebuffer hat dabei meistens einen Depth-Buffer, der Tiefeninformationen speichert, und außerdem einen oder mehrere Color-Buffer. Diese Buffer werden häufig Framebuffer-*Targets* oder *Attachments* genannt. Die Tiefeninformationen des Depth-Buffers sind dabei nicht-linear ska-

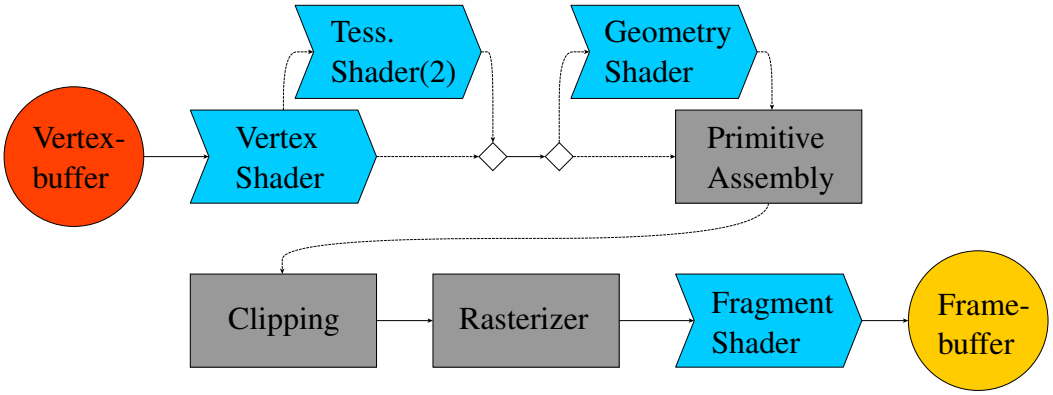

Abbildung 2: *Schematische Darstellung der OpenGL-Grafikpipeline. In Rot-orange eingefärbt ist der Vertexbuffer des Vertex Array Objects. In Cyan sind die programmierbaren Teile der Pipeline markiert, also die unterschiedlichen Shader, wohingegen die nicht-programmbierbaren Teile der Pipeline in Grau eingefärbt sind. Der Framebuffer mit dem finalen Bild ist in Gelb dargestellt.*

liert, um den erwähnten Tiefeneffekt zu erzielen.

Es wird außerdem unterschieden zwischen dem *Default*-Framebuffer und von der Anwendung erstellten Framebuffer-*Objekten* (FBOs). Der Default-Framebuffer ist ein von der OpenGL-Implementation bereitgestellter Framebuffer, der auf dem Bildschirm dargestellt wird. Dieser hat nur einen Colorbuffer, in den die Ausgabe des Fragmentshaders hineingeschrieben wird, und das Hinzufügen von Attachments ist nicht möglich.

Demgegenüber erlaubt ein FBO es, einen Framebuffer mit mehreren Attachments zu erstellen und diese mit einem entsprechend darauf vorbereiteten Fragment-Shader auch gleichzeitig zu beschreiben. Das gleichzeitige Rendern in mehrere Attachments wird als *Multiple Render Targets* bezeichnet. Außerdem können die Attachments eines Framebuffers als Texturen verwendet werden.

Durch Unterschiede in der Wiederholfrequenz des Bildschirms und der Anwendung kann es mit nur einem Default-Framebuffer auftreten, dass einzelne Objekte löchrig erscheinen oder komplett fehlen. In diesem Fall können diese Objekte nicht mehr komplett in den Buffer eingezeichnet werden, bevor dieser auf dem Bildschirm dargestellt wird.

Dieses Problem kann mit Double-Buffering nicht auftreten, denn hier existieren zwei Default-Framebuffer, in die abwechselnd ein Frame gezeichnet wird. Der je-

weils andere Buffer wird währenddessen auf den Bildschirm gezeichnet. Weil die Buffer erst nach Abschluss eines Frames getauscht werden, kann es hier nicht mehr auftreten, dass ein unvollständiger Framebuffer gezeichnet wird.
Der aktuell auf dem Bildschirm dargestellte Buffer wird *Frontbuffer* genannt, während der jeweils andere, aktuell zum Rendern verwendete Buffer als *Backbuffer* bezeichnet wird.

2.1.8 Texturen

Häufig sollen die Oberflächen der Primitive jedoch nicht einfarbig gezeichnet werden, sondern feinere Strukturen wie bspw. die Steine einer Hauswand darstellen. Zu diesem Zweck werden die Oberflächen mit Texturen belegt. Dies sind Bilder der Oberfläche, wobei es verschiedene Arten von Texturen gibt, die jeweils unterschiedliche Eigenschaften des Materials der Oberfläche enthalten. Die *diffuse* Textur gibt beispielsweise die Farbe eines Materials an, während eine *spekulare* Textur die Parameter für Lichtreflexionen bereitstellt. Eine *Normalentextur* hingegen liefert detaillierte Informationen über die Oberflächennormalen und erlaubt dadurch eine feinere Berechnung der Beleuchtung des Objektes.

Um eine Textur auf eine Oberfläche aufzutragen, wird *Texture-Mapping* genutzt. Dies ist ein Verfahren, bei dem die Vertices des Primitivs jeweils einen Punkt in der Textur, der für diesen Vertex genutzt werden soll, angeben. Diese Werte werden Texturkoordinaten genannt und für zweidimensionale Texturen mit s und t abgekürzt. Für zwischen verschiedenen Vertices liegende Bereiche der Oberfläche wird zwischen den Texturkoordinaten der angrenzenden Vertices interpoliert. Beim Zeichnen greift der Fragment-Shader jetzt auf die interpolierte Position in der Textur zu und verwendet den an dieser Position ausgelesenen Farbwert für die Beleuchtungsberechnungen. Dadurch wird das Primitiv mit dem zwischen den Texturkoordinaten der Vertices gelegenen Bereich der Textur belegt. Die verschiedenen Arten von Texturen können dabei nebeneinander und gleichzeitig genutzt werden.

Texturen werden in OpenGL standardmäßig zur Verwendung an eine von mehreren Textureinheiten gebunden und können vom Fragment-Shader dann über die Textureinheit genutzt werden. Durch diese Struktur müssen Objekte mit unterschiedlichen Texturen in getrennten Draw-Calls gezeichnet werden.

Alternativ existiert noch ein weiterer Modus, durch welchen die Texturen nicht mehr vor Verwendung an die Textureinheiten gebunden werden müssen. Durch das Entfallen der Binde-Operationen von Texturen an Textureinheiten entfällt eine Menge Overhead, außerdem erlaubt dies auch die Vereinigung der Draw-Calls verschieden texturierter Objekte zu einem einzigen Call. Diese Methode nennt sich *Bindless Textures*. Hier wird für eine Textur ein zusätzliches 64-Bit Handle erstellt, welches die Textur eindeutig identifiziert. Nach Erstellung dieses Handles kann der Zustand einer Textur größtenteils nicht mehr verändert werden. Hiervon explizit ausgeschlossen sind jedoch Änderungen am *Inhalt* der Textur.
Um die Bindless Texture zu verwenden, ist jedoch noch ein weiterer Schritt nötig: Die Textur muss *resident* gesetzt werden. Dies bedeutet, dass die Textur danach garantiert in einem Speicherbereich liegt, auf den die Grafikkarte zugreifen kann.

Ist die Bindless Texture nun resident, kann das Handle der Textur an den Fragment-Shader übergeben werden und dieser kann mithilfe des Handles auf die Textur zugreifen.

2.2 Forward vs. Deferred Rendering

Je nach Anzahl der Lichtquellen und der Beschaffenheit der zu zeichnenden Szene ist die Beleuchtung eine der aufwändigsten Operationen beim Zeichnen eines Frames. Deshalb existieren verschiedene Ansätze, wie die Beleuchtung der Szene ausgeführt werden kann, um die benötigte Zeit zu reduzieren.

2.2.1 Forward Rendering

Das historisch ältere Verfahren wird als *Forward Shading* oder *Forward Rendering* bezeichnet.

Beim Forward Rendering werden die Objekte zuerst im Vertex-Shader transformiert und die vom Rasterizer für dieses Objekt generierten Fragmente anschließend sofort vom Fragment-Shader beleuchtet. Dadurch kann es passieren, dass Objekte, die im finalen Bild durch Verdeckung durch ein anderes später gezeichnetes Objekt gar nicht sichtbar sind, trotzdem beleuchtet werden. Dieses Phänomen wird *Overdraw* genannt.

Die Menge aller in einem Frame generierten Fragmente sei mit D bezeichnet und enthält also auch diese später durch ein anderes Fragment verdeckten Fragmente.

Damit kann die Beleuchtung im Forward Shading in Pseudocode in einer Form ähnlich zu Algorithmus 1 beschrieben werden.

```
for all m: Lichtquellen in der Szene do
  for all f : Fragmente aus D do
    Beleuchte_Fragment(f, m)
  end for
end for
```

Algorithmus 1: *Struktur der Beleuchtungsberechnungen beim Forward Shading. Alle Lichtquellen beleuchten alle aus den Objekten der Szene generierten Fragmente, unabhängig davon, ob diese später verdeckt werden.*

Da der Forward Renderer die Fragmente des Objektes sofort nach Erzeugung beleuchtet, besitzt dieser eine Zeitkomplexität von $O(|D| * m)$, wobei m die Anzahl der Lichtquellen in der Szene ist, und führt deshalb in Szenen mit viel Overdraw zu einer schlechten Performance.

2.2.2 Deferred Rendering

Deswegen wurde bereits 1988 von Deering et al. [Dee+88] ein alternatives Konzept vorgeschlagen, welches von Saito und Takahashi in [ST90] zu dem heute üblichen Verfahren des *Deferred Shadings* oder *Deferred Renderings* weiterentwickelt wurde. Die Autoren beider Publikationen haben diese heute für das Verfahren gebräuchlichen Bezeichnungen jedoch nicht benutzt. Die Idee hinter dem Verfahren ist die Entkopplung von Geometrie- und Beleuchtungsberechnungen durch die Einführung eines zusätzlichen Buffers.

In einem ersten Renderpass werden die zur Beleuchtung nötigen Informationen wie Farbinformationen, Normalen und Tiefeninformationen der Vertices in einen zusätzlichen, *G(eometry)-Buffer* genannten Buffer geschrieben. Zu beachten ist, dass zu diesem Zeitpunkt noch keine Beleuchtungsberechnungen ausgeführt werden. Für den G-Buffer wird ein zusätzlicher Framebuffer, meist mit derselben Auflösung wie der Backbuffer, verwendet.
Anschließend wird der Inhalt des G-Buffers im sogenannten *Lighting-Pass* mit den Lichtquellen der Szene beleuchtet.

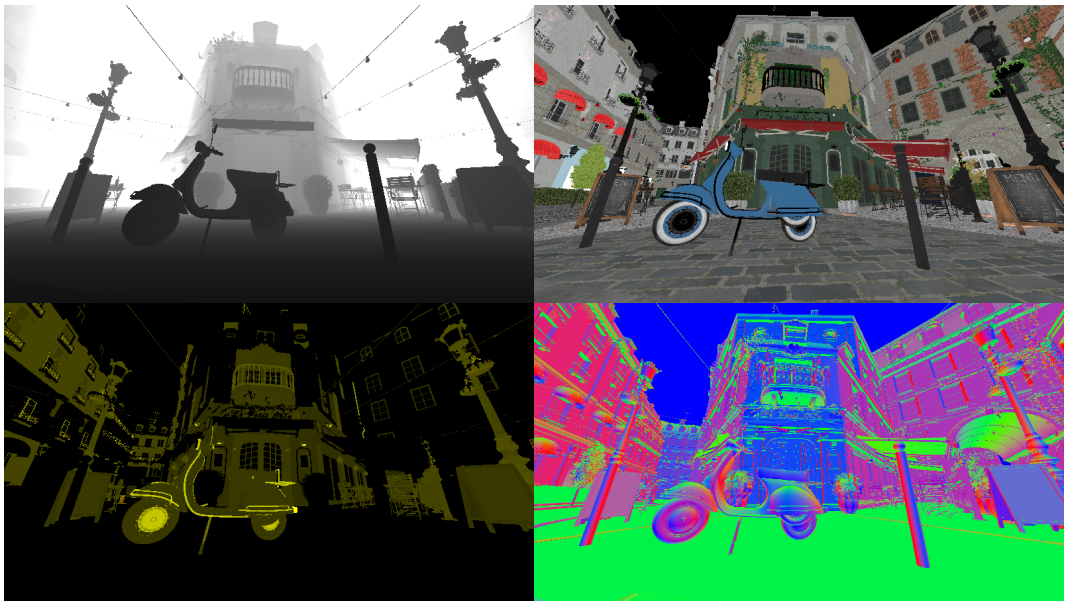

Abbildung 3: *Die vier Attachments des G-Buffers. Oben links ist der Depth-Buffer zu sehen, oben rechts ist die diffuse Farbe dargestellt, unten links die spekularen Informationen und unten rechts die Normalen. Das verwendete Modell ist der Außenbereich des „Amazon Lumberyard Bistros“([Lum17]).*

Zum Speichern der verschiedenen Informationen verwendet dieser G-Buffer mehrere Attachments, in die die unterschiedlichen Informationen per MRT parallel hineingeschrieben werden können.
Der später verwendete Deferred Renderer benutzt zum Beispiel vier Attachments: Das Erste ist der Depth-Buffer, welcher für den Tiefentest benötigt wird. In diesem Depth-Buffer sind die Tiefenwerte nicht-linear skaliert im Wertebereich [0, 1] gespeichert. Weiterhin existieren noch ein Attachment für die diffusen Farben mit einem Alphakanal. Zusätzlich existiert noch je ein Attachment für die Reflexionsinformationen und die Normalen.
Insgesamt verwendet der G-Buffer des Deferred Renderers dadurch 13 Byte pro Pixel, wobei drei Byte auf den Depth-Buffer, 4x1 Byte auf die diffuse Textur, 2x1 Byte auf die Reflexionsinformationen und 2x2 Byte auf die Normalentextur entfallen. Die Normalen werden komprimiert gespeichert, wozu ein Verfahren namens „Octahedron normal vector encoding“, wie in [Nar14] demonstriert, angewendet wird.

Dieses Setup ist in Abbildung 3 dargestellt, wobei oben links die Tiefen, oben rechts die diffusen Farben, unten rechts die Normalen und unten links die Reflexionsinfor-

mationen zu sehen sind. Für die Darstellung wurden jedoch die Tiefenwerte aus dem Depth-Buffer zuerst in eine lineare Skalierung umgerechnet und anschließend durch 1000 geteilt. Ohne diese Umrechnung ist das Bild der Tiefen sehr hell, da durch die nicht-lineare Skalierung sehr viele Pixel einen Wert nahe 1 haben. Außerdem wurden die Normalen aus der komprimierten Speicherung zur Darstellung dekomprimiert. Zur Bestimmung des Farbwertes. den eine Normalen in der Darstellung erhält, wird der Betrag genutzt, wodurch für Normalen mit entgegengesetzter Richtung dieselbe Farbe verwendet wird.

Da die Beleuchtung mit diesem Verfahren erst stattfindet, nachdem alle Objekte in den G-Buffer gezeichnet wurden, werden jetzt nur noch tatsächlich sichtbare Objekte bzw. deren Fragmente beleuchtet, die unnötigen Berechnungen für verdeckte Objekte entfallen also.

Dieses Verfahren kann in Pseudocode also wie in Algorithmus 2 dargestellt werden. Sei D wie beim Forward Renderer die Anzahl aller für einen Frame erzeugten

```
for all n: sichtbare Objekte der Szene do
  RenderInGBuffer(n)
end for
for all m : Lichtquellen in der Szene do
  BeleuchteGBuffer(m)
end for
```

Algorithmus 2: *Struktur der Beleuchtungsberechnungen beim Deferred Shading. Zuerst werden alle Objekte in den G-Buffer geschrieben. Anschließend wird dieser beleuchtet.*

Fragmente, und außerdem E die Menge der im endgültigen Bild tatsächlich sichtbaren und beleuchteten Fragmente. Damit kann die Zeitkomplexität des Deferred Renderers ausgedrückt werden als $O(|D| + |E| * m)$, wobei m wieder die Anzahl der Lichtquellen darstellt.
Der Deferred Renderer kann weiter optimiert werden, indem für jede Lichtquelle zuvor ein möglichst kleiner Einflussbereich berechnet wird. Dadurch sinkt die Größe der Menge E besonders für kleine Lichtquellen.

2.2.3 Vor- und Nachteile

In einer Szene mit viel Overdraw und vielen Lichtquellen mit kleinem Einflussbereich und damit einer geringen Größe von E im Vergleich zur Menge aller Fragmente D ist der Deferred Renderer also deutlich effizienter. Im Worst Case, wenn kein oder kaum Overdraw vorhanden ist und viele große Lichtquellen existieren (E also nicht viel kleiner als D ist), ist dieser jedoch langsamer als der Forward Renderer, da durch den zusätzlichen Framebuffer Overhead verursacht und zusätzliche Speicherbandbreite benötigt wird.

Ist die Anzahl der Lichtquellen in einer Szene nicht vorhersehbar, so kann eine Beleuchtungsengine nur als *Multi-Pass*-Architektur entworfen werden, da Arrays auf der Grafikkarte nur statische Größen haben können. Multi-Pass bedeutet, dass der für die Beleuchtung zuständige Fragment-Shader so entworfen ist, dass dieser in einer Ausführung die Fragmente nur mit einer Lichtquelle beleuchtet. Zur Beleuchtung eines Objektes mit m Lichtquellen muss der Fragment Shader also auch m-mal die Fragmente des Objektes beleuchten. Hierzu muss in einem Forward Renderer auch der Vertex-Shader m-mal ausgeführt werden, wobei jedoch $m - 1$ dieser Aufrufe redundant sind.
Für einen Deferred Renderer ist eine solche Multi-Pass-Architektur jedoch kein Problem, da durch die Entkopplung von Beleuchtung und Geometrietransformation der Vertex-Shader unabhängig von der Anzahl an Lichtquellen für jedes Objekt nur einmal ausgeführt wird.

Leichte Variationen des Deferred Renderings vereinfachen auch vorher nur schwer umzusetzende globale Post-Processing Effekte wie Ambient Occlusion oder Motion Blur. Eine solche Implementation für Motion Blur als Post-Processing Effekt eines Deferred Renderers haben bspw. McGuire et al. [McG+12] präsentiert.

Einer der größten Nachteile ist jedoch die Inkompatibilität dieses Verfahrens mit transparenten Objekten. Denn da der G-Buffer pro Pixel nur die Farbinformationen des am nächsten an der Kamera liegenden Primitivs speichert, in diesem Fall also das transparente Objekt, sind die Farbinformationen für dahinter liegende Objekte im Lighting-Pass nicht vorhanden, sodass diese auch nicht beleuchtet werden können. Häufig wird dieses Problem gelöst, indem der Deferred Renderer nur für opake Objekte genutzt wird. Anschließend werden die transparenten Objekte dann in einem zweiten Forward-Rendering-Pass in der Szene platziert.

Ein zweiter Nachteil ist der hohe Verbrauch von Speicher und Bandbreite auf der GPU durch den zusätzlichen Framebuffer mit mehreren MRTs. Weiterhin ist bspw. Multi-Sample-Anti-Aliasing (MSAA) nur mit deutlich mehr Entwicklungsaufwand als für einen Forward MSAA Renderer umzusetzen, da die Lighting-Stage für jedes Sample ausgeführt werden muss und dies selbstständig implementiert werden muss.

2.3 Level of Detail

In einer Szene existiert oftmals eine Vielzahl an Objekten, mit einer riesigen Zahl an Polygonen. So besteht bspw. das in dieser Arbeit häufig verwendete „Amazon Lumberyard Bistro“ [Lum17] bereits aus ca. 3,85 Millionen Dreiecken. Mit der kompletten Darstellung sehr komplexer Szenen wie den Spielwelten in Open-World-Titeln der letzten Jahre (bspw. „GTA V“ oder „The Witcher 3“) wäre aber auch heutige Grafikhardware aufgrund der schieren Anzahl an Vertices maßlos überfordert, sodass die Darstellung in Echtzeit, die für diese Videospiele nötig ist, auf diese Weise nicht umgesetzt werden kann.
Aufgrund der Größe und des Detailreichtums dieser Spielwelten ist bereits das komplette Laden der Spielwelt in den Arbeitsspeicher nicht möglich, da diese selbst komprimiert bereits über 10 Gigabyte groß sind. Deshalb wird nur ein kleiner aktuell und demnächst sichtbarer Teilbereich der Welt geladen und bei Bedarf dann weitere angrenzende Teilbereiche nach- oder entladen.

2.3.1 Reduktion des Detaillevels

Aber auch diese Teilbereiche sind häufig geometrisch noch so komplex, dass sich diese nicht in Echtzeit rendern lassen. Deshalb muss auch die geometrische Komplexität dieser Teilszenen noch reduziert werden. Dabei kann eine weitere Reduktion des geladenen Bereiches der Welt nicht verwendet werden, da ansonsten sichtbare Teile der Welt fehlen würden.

In einer Szene existieren jedoch oft Objekte, die vom Betrachter weit entfernt sind, und dementsprechend nur sehr wenige Pixel des endgültigen Bildes ausmachen. Normal gezeichnet benötigen diese jedoch genauso viel Rechenzeit für die Geometrietransformationen, wie wenn das Objekt direkt vor der Kamera steht. Das führt zu einem überproportional großen Rechenaufwand für solche kleinen, weit von der Ka-

mera entfernt liegenden Objekte. Deshalb wurden bereits viele verschiedene Ideen entwickelt, wie die Rechenzeit für weit von der Kamera entfernte Objekte verringert werden kann, ohne die Bildqualität sichtbar zu verschlechtern.

Im Folgenden werden einige ausgewählte dieser Ideen präsentiert, es existieren jedoch noch eine Vielzahl weitere Ansätze. Einen detaillierteren Überblick über die verschiedenartigen Entwicklungen zur LOD-Reduktion gibt [JWP05].

Die verwendeten Ansätze reduzieren den Rechenaufwand zum Zeichnen weit entfernter Objekte alle durch eine Reduktion der geometrischen Komplexität dieser Objekte. Dabei erzeugte unterschiedliche Detaillevel werden als *Level of Detail*, kurz LOD, bezeichnet. Die präsentierten Ansätze unterscheiden sich stark in der Art und Weise, wie diese verschiedenen LODs erzeugt und verwendet werden.

Eine offensichtliche Möglichkeit ist, dass der Designer eines Modells zusätzlich zu der Full-Detail-Version auch noch Versionen mit reduziertem Detaillevel erstellt. Da dies jedoch einen hohen zusätzlichen Aufwand (und damit Entwicklungskosten) bedeutet, haben Heckbert und Garland [HG94] ebenso wie Schaufler und Stürzlinger [SS95] Algorithmen entwickelt, welche Low-Detail-Versionen eines Modells automatisch erzeugen können.

2.3.2 Impostors

Alternativ kann anstatt einer Vereinfachung der Geometrie diese auch komplett ersetzt werden, entweder mit sogenannten *Impostors* oder einer Spezialform dieser, den *Billboards*. So kategorisieren Sillion et al. [SDB97] bspw. ein Modell des Pariser Stadtteils Montmatre zuerst in (Straßen-) Blöcke, die dann bei großer Entfernung zur Kamera durch einen Impostor ersetzt werden können. Dazu wird der zu ersetzende Stadtteil zuerst in eine Tiefentextur gerendert und anschließend noch in eine Farbtextur gezeichnet. Daraufhin wird, basierend auf der Tiefe, neue Geometrie generiert, auf welche die Farbtextur dann aufgetragen wird. Diesen Ansatz entwickeln Décoret et al. in [Dec+99] weiter, indem sie diese Impostors für kleinere Bereiche generieren, um Tiefeninformationen besser zu erhalten.

2.3.3 Billboards

Mit Billboards werden die Objekte durch noch weniger Geometrie ersetzt: Ein Billboard besteht nur aus einem Quad bzw. zwei Dreiecken. Diese werden mit einer Textur des ursprünglichen Objektes belegt, wobei diese Textur entweder mit dem Modell durch den Designer erstellt sein kann, oder dynamisch generiert wird. Schaufler und Stürzlinger [SS96] verwenden bspw. dynamische Texturen, um Teile der Szene durch Billboards zu ersetzen. Diese Billboards speichern jedoch nur Farben und Normalen eines Objektes, sodass die Tiefeninformationen der Objekte verloren gehen. Dadurch sehen diese Billboards nur für sehr weit entfernt liegende Objekte gut aus.

Deshalb schlagen Décoret et al [Déc+03] für höhere Detaillevel die Verwendung von mehreren, sich überlagernden Billboards vor. Diese Gruppierungen von Billboards nennen die Autoren *Billboard Clouds*. Diese werden allerdings nicht dynamisch generiert, sondern müssen in einem Pre-Processing-Schritt berechnet werden.

Dynamisch generierte Billboards müssen außerdem hin und wieder aktualisiert werden, um weiterhin eine akkurate Repräsentation des ersetzten Objektes darzustellen. Dazu schlägt Schaufler [Sch95] die Verwendung von zwei Heuristiken bei *seitlicher* Kamerabewegung und einer Kamerabewegung in Richtung des Objektes vor. Sehr ähnliche Heuristiken werden auch hier verwendet werden, da diese einfach zu implementieren sind.

2.3.4 LOD-Entscheidung

Existieren verschiedene Detaillevel eines Modells, muss außerdem zur Laufzeit noch entschieden werden, wann welches Detaillevel eingesetzt werden soll. Neben anderen verwendet Blake [Bla87] hierfür eine statische Heuristik basierend auf der Entfernung eines Objekts zur Kamera.
Wie Funkhouser und Séquin [FS93] bemerken, kann dies zu stark schwankenden Framerates führen, und schlagen stattdessen einen Algorithmus vor, der die LOD-Entscheidung auf Basis einer angestrebten Framerate trifft. Ist die verfügbare Frametime knapp, verwenden die Autoren anstatt des für die optimale Bildqualität nötigen LODs ein schlechter aussehendes, aber schnelleres LOD.

3 Billboards für große Szenen

In diesem Kapitel wird erklärt, wie Billboards weiterentwickelt werden können, um ihre Qualität besonders bei mittlerer Entfernung zur Kamera zu verbessern. Anschließend widmet sich dieses Kapitel der Erklärung der Berechnungen, die nötig sind ein solches verbessertes Billboard zu erstellen und anzuzeigen.

3.1 G-Buffer Billboards

Im vorherigen Kapitel wurden das Forward und Deferred Rendering vorgestellt, ebenso wie ein Überblick über verschiedene LOD-Methoden. Diese verwenden für den Erhalt der Tiefe eines ersetzten Objekts oftmals neue, zusätzliche Geometrie (Impostors) oder verlieren die Tiefeninformation (Billboards).

Indem für die Billboards ein Framebuffer mit demselben Attachment-Layout wie für einen Deferred Renderer verwendet wird, kann diese Information jedoch erhalten werden.

Als erster Schritt wird die Größe des Modells auf dem Bildschirm sowie die Entfernung zur Kamera bestimmt. Ist das Objekt weit genug von der Kamera entfernt (genauere Definition von „weit genug“ folgt später), wird das Objekt in einen zusätzlichen Framebuffer mit Multiple Render Targets gezeichnet, statt es normal in der Szene zu platzieren.

In der Szene wird nun, wie bei den Billboards, anstelle des Objektes nur noch ein Quad platziert, welches an den Achsen des Kamerakoordinatensystems ausgerichtet ist. Als Texturen für dieses Rechteck werden die MRTs des zuvor erstellten Framebuffers verwendet. Da das Objekt in folgenden Frames nicht noch einmal in diesen Framebuffer gezeichnet werden muss, kann Renderzeit eingespart werden.

Zu bemerken ist dabei, dass auch im Falle eines Deferred Renderers die G-Buffer-Billboards zuerst in den Szenen-G-Buffer eingefügt werden, und erst danach der Lighting Pass ausgeführt wird. Hierdurch sind keine Modifikationen der Shader für den Deferred Renderer nötig.

Allerdings muss das G-Buffer-Billboard aufgrund von Kamerabewegungen gelegentlich neu gezeichnet werden, um Änderungen des Blickwinkels und/oder der

Größe des Objektes zu berücksichtigen. Selbst bei starker Kamerabewegung ist jedoch das erneute Zeichnen des G-Buffer-Billboards nicht in jedem Frame nötig, da sich das Aussehen des Objektes von Frame zu Frame nur geringfügig ändert.

In jedem Frame, in dem das G-Buffer-Billboard wiederverwendet werden kann, kann also Rechenzeit eingespart werden, da das Zeichnen der Billboards schneller ist als das Zeichnen des originalen Objektes. Aber auch die Ausführung einer gewissen Anzahl an Updates pro Frame ist effizient möglich, da der Updatevorgang nur geringfügig aufwändiger ist als das Zeichnen in die normalen Buffer der Grafikpipeline.

Im folgenden werden die Begriffe G-Buffer-Billboard und Billboard synonym verwendet.

3.2 Billboard Framebuffer

Da in diesem Setup besonders die Operationen zum Erstellen, Verändern (in der Größe) und Löschen von Framebuffern aufgrund von hohem Treiberoverhead viel Rechenzeit benötigen, ist es ineffizient für jedes erstellte Billboard einen eigenen, kleinen Framebuffer zu verwenden.
Deshalb werden eine bestimmte Anzahl Billboards in einem entsprechend größeren Framebuffer zusammen gespeichert. Dadurch kann die Anzahl verwendeter Framebuffer und dynamischer Texturen stark gesenkt werden und die Performance im Vergleich zur Verwendung eines Framebuffers pro Billboard signifikant verbessert werden. Allerdings macht dies die Einschränkung der verfügbaren Auflösungen der Billboards nötig, da sonst die Verwaltung in den Framebuffern zu aufwändig wäre. Aus diesem Grund hat ein Billboard immer eine Dimension B_D, wobei diese aus den Dimensionen von 16x16, 32x32, 64x64, 128x128, 256x256 oder 512x512 Pixeln ausgewählt wird. Für die später verwendeten Modelle wurde jedoch besonders die größte Klasse nur für sehr wenige Billboards benötigt, wie in Abbildung 15 (Seite 49) zu erkennen.

Diese Framebuffer werden im folgenden als Billboard-Framebuffer bezeichnet. Der interne Aufbau dieser Buffer, der in Abbildung 4 dargestellt ist, soll jetzt kurz erläutert werden.

Für jede dieser verfügbaren Dimensionen wird zu Beginn ein Billboard-

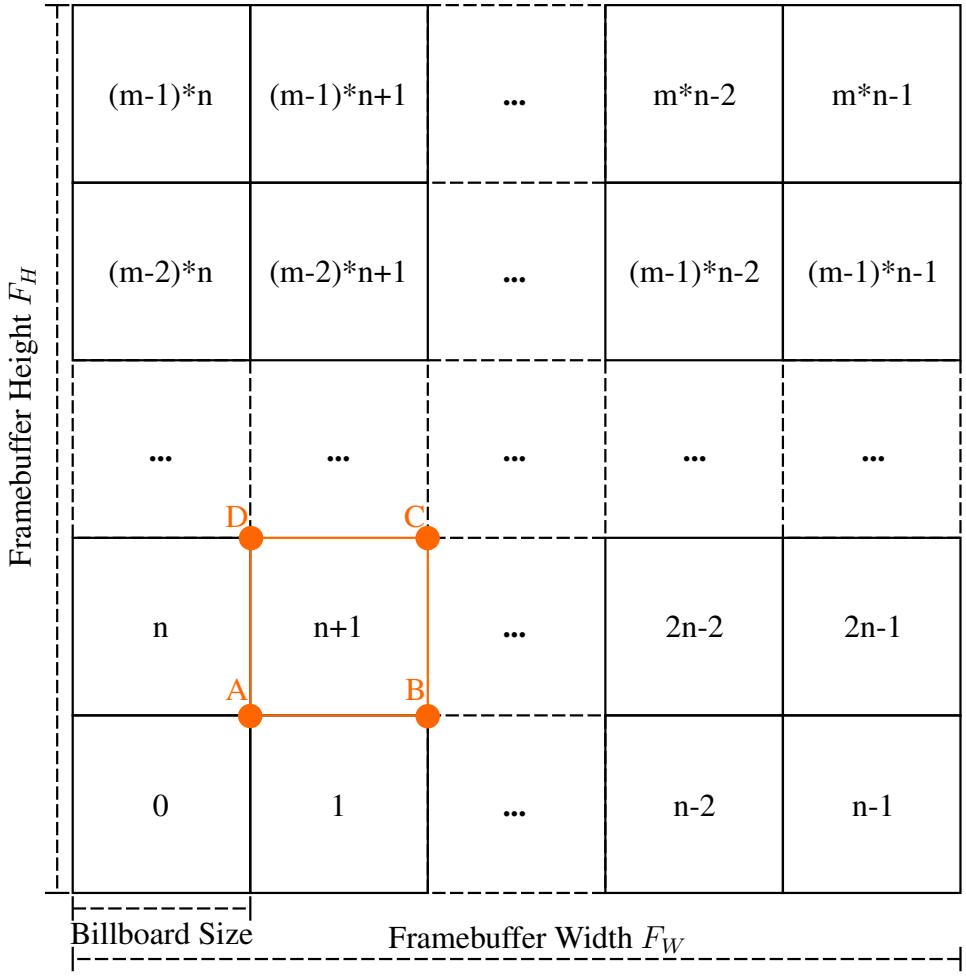

Abbildung 4: *Anordnung der Billboards im Billboard-Framebuffer. Breite und Höhe dieses Framebuffers werden aus der Anzahl an Billboards, die der Framebuffer enthalten soll, berechnet. Das orange hervorgehobene Billboard und dessen Eckpunkte illustrieren, wie Viewport und Texturkoordinaten für ein Billboard berechnet werden (siehe Kapitel 3.4.1). Ein solcher Billboard-Framebuffer existiert für jede der verwendeten Billboardgrößen von 16x16 bis 512x512 Pixel.*

Framebuffer erstellt, wobei die Anzahl an Billboards pro Dimensionslevel separat über eine Konfiguration eingestellt werden kann. Da diese Anzahl angibt, wie viele Billboards einer Größe maximal gleichzeitig vorhanden sein können, ist dies eine wichtige Stellschraube sowohl für Speicherbedarf als auch für Performance und Bildqualität.

Beim Testen hat sich herausgestellt, dass für die getesteten Modelle die Allokation von 512 Billboards pro Dimension einen guten Standardwert darstellt. Für die größte Klasse werden jedoch nur 128 Billboards angelegt, da diese nur selten benötigt wird. Damit sind die Kapazitäten einer Dimension nicht zu schnell ausgelastet, die Billboard-Framebuffer benötigen aber auch nicht übermäßig viel Speicherplatz, da die Billboard-Framebuffer mit diesen Größen zusammen insgesamt ca. 1 GB an Videospeicher belegen. Dieser sollte den meisten Grafikkarten der letzten Jahre zur Verfügung stehen.

Werden für eine Szene nur sehr wenige oder sehr viele Billboards einer bestimmten Größe benötigt, so ist es sinnvoll die Anzahl an verfügbaren Billboards für diese Dimension nach unten bzw. nach oben anzupassen. Dies kann den Speicherverbrauch und die Bildqualität positiv beeinflussen.

Der vom Nutzer für Dimension B_D eingestellte Wert sei bezeichnet als $A(B_D)$. Zuerst wird daraus die Breite berechnet, die $A(B_D)$ Billboards nebeneinander angeordnet einnehmen würden:

$$B_T = B_D * A(B_D)$$

Damit sichergestellt ist, dass der Framebuffer immer eine ganze Anzahl an Billboards sowohl in der Breite als auch der Höhe aufnehmen kann, wird anschließend die maximal nutzbare Breite a bzw. Höhe b eines Framebuffers auf der aktuellen Grafikkarte berechnet. Dazu wird zuerst die maximal unterstützte Breite bzw. Höhe eines Framebuffers von der OpenGL-Implementation abgefragt. Aus diesen Werten wird dann der Logarithmus zur Basis 2 gebildet, welcher anschließend auf die nächste Ganzzahl abgerundet wird und anschließend wieder die Zweierpotenz be-

rechnet:

$$a = 2^{\lfloor \log_2(\text{ GL\_MAX\_FRAMEBUFFER\_WIDTH}) \rfloor},$$
$$b = 2^{\lfloor \log_2(\text{ GL\_MAX\_FRAMEBUFFER\_HEIGHT}) \rfloor}$$

Hierdurch wird die maximal unterstützte Breite bzw. Höhe im Fall, dass diese keine Zweierpotenzen sind, auf die nächstkleinere Zweierpotenz abgerundet.
Daraus können die genutzte Breite und Höhe des Framebuffers berechnet werden als das Minimum aus der Breite der Billboards nebeneinander B_T und der zuvor bestimmten maximalen Breite eines Billboard-Framebuffers a. Reicht die maximale Breite des Framebuffer nicht aus, um alle $A(B_D)$ Billboards nebeneinander unterzubringen, so wird eine neue Reihe begonnen:

$$F_W = \min(B_T, a),$$
$$F_H = \min(B_D * \left\lceil \frac{B_T}{FB_W} \right\rceil, b)$$

Die Anzahl der Reihen in einem Billboard-Framebuffer kann damit berechnet werden als die totale Breite B_T geteilt durch die Breite des Framebuffers F_W. Da für die vom User angegebene Anzahl an Billboards nicht garantiert werden kann, dass diese immer genau eine ganze Anzahl an Reihen füllt, wird dieser Wert anschließend auf die nächste Ganzzahl aufgerundet. Dadurch wird garantiert, dass mindestens $A(B_D)$ Billboards in den Billboard-Framebuffer passen, wobei jedoch eine Ausnahme existiert. Überschreitet die so berechnete Höhe die maximale Höhe b des Framebuffers, so wird stattdessen diese als Höhe verwendet. In diesem Fall ist es also technisch nicht möglich, $A(B_D)$ Billboards in einem Framebuffer unterzubringen.

3.3 Entscheidung zwischen Billboard und normalem Objekt

In jedem Frame muss für jedes Objekt erneut entschieden werden, ob dieses im sichtbaren Bereich liegt. Ist dies nicht der Fall, braucht das Objekt nicht genzeichnet werden. Für sichtbare Objekte muss jedoch außerdem noch entschieden werden, ob diese normal oder als Billboard gezeichnet werden sollen. Hat ein Objekt mehrere

unterschiedliche Instanzen, so wird die Entscheidung noch verkompliziert, da diese nun für jede Instanz separat getroffen werden muss. Außerdem benötigt dann auch jede Instanz ein eigenes Billboard, falls sie vereinfacht gezeichnet werden soll.

Zur Entscheidungsfindung wird jedes Objekt mit einem Score bewertet, welcher im Folgenden als S_O oder *Object Score* bezeichnet wird. Da die Berechnung von S_O in jedem Frame für jedes Objekt (bzw. jede Instanz) stattfinden muss, muss dieser möglichst einfach zu berechnen sein, soll aber trotzdem noch nachvollziehbare und gut aussehende Ergebnisse liefern.

Zuerst wird die Bounding Box des Objektes in NDC-Koordinaten bestimmt. Liegt diese komplett außerhalb des sichtbaren Bereiches, wird S_O auf einen negativen Wert gesetzt und keine weiteren Berechnungen mehr durchgeführt. Dieser Check ist in NDC Koordinaten sehr einfach durchzuführen, da der sichtbare Bereich den Koordinaten im Bereich [-1, 1] in allen drei Achsen entspricht.
Dies fungiert also als eine Art Sichtbarkeitstest durch den nur Objekte innerhalb des sichtbaren Bereiches auch tatsächlich gezeichnet werden. Der Sichtbarkeitstest wird auf diese Weise durchgeführt, da die Bounding Box in NDC-Koordinaten für spätere Berechnungen sowieso bestimmt werden muss.

Für Objekte innerhalb des sichtbaren Bereiches wird S_O jedoch auf 0 oder einen positiven Wert gesetzt, wobei 0 für normal zu zeichnende Objekte, und ein positiver Wert für durch ein Billboard zu ersetzende Objekte verwendet wird. Die Berechnung dieses Wertes wird im Folgenden erklärt.

Zuerst sei die Größe eines Objektes definiert als die Distanz zwischen Minimum und Maximum der Axis Aligned Bounding Box des Objektes in Kamerakoordinaten:

$$G_O = |aabb.max - aabb.min|$$

Auf dieselbe Weise sei auch die Größe der gesamten Szene G_S definiert.

Da die Distanz zur Kamera sehr einfach zu berechnen ist, wird diese als primäre Eigenschaft genutzt. Bestimmt wird die Distanz zur Kamera anhand der z-Koordinate eines Punktes in Kamerakoordinaten. Um sehr große von hinter der Kamera bis in den sichtbaren Bereich reichende Objekte wie bspw. Wände oder eine Skybox ausschließen zu können, wird jedoch nicht nur die Entfernung des Centerpunktes

oder Schwerpunktes des Objektes betrachtet, sondern die Entfernungen von sowohl minimalem Punkt der AABB des Objektes in Kamera-Koordinaten als auch des maximalen Punktes dieser Bounding Box.
Diese Entfernungen werden als d_1 und d_2 bezeichnet und werden zuerst auf ein negatives Vorzeichen überprüft, da die Kamera in OpenGL in die negative z-Richtung guckt. Dadurch wird nochmal sichergestellt, dass das Objekt komplett vor der Kamera liegt. Denn die später durchgeführten Berechnungen bei der Erstellung des Billboards sind für ein hinter der Kamera liegendes Objekt nicht definiert und könnten zu unerwarteten Ergebnissen führen.

Liegen beide Punkte vor der Kamera, werden diese anschließend mit einem auf Basis der gesamten Szenengröße G_S berechneten Schwellwert G_T verglichen (bspw. 20% der Szenentiefe). Diese Szenengröße wird, wie auch die Größe eines Objektes, definiert als die Distanz vom Minimum zum Maximum der Bounding Box der Szene. Aufgrund der negativen z-Werte sichtbarer Objekte wird dieser Schwellwert mit einem negativen Vorzeichen versehen. Zusammen mit der Größe G_O eines Objektes wird aus den Entfernungen d_1 und d_2 der Object Score S_O für dieses Objekt berechnet. Liegt das Objekt näher als G_T an der Kamera, ist der Objekt Score 0. Zusammenfassed wird S_O also wie folgt berechnet:

$$S_O = \begin{cases} \dfrac{(-d_1 - d_2)}{G_O}, & \text{wenn } d_1 < G_T \text{ und } d_2 < G_T, \\ 0, & \text{wenn } d_1 >= G_T \text{ oder } d_2 >= G_T \\ -1, & \text{wenn Objekt nicht im sichtbaren Bereich} \end{cases}$$

Aufgrund der negativen Kamerablickrichtung in OpenGL und dem bereits erfolgten Ausschluss hinter der Kamera liegender Objekte sind die Werte von d_1 und d_2 immer negativ, sodass S_O durch die zusäzlichen Vorzeichen im obersten Fall einen positiven Wertebereich erhält.

Durch diese Heuristik erhalten kleine Objekte mit großer Entfernung zur Kamera hohe Scores und werden deshalb bevorzugt durch Billboards ersetzt.

In Abbildung 5 ist die Berechnung des Scores für sichtbare Objekte noch einmal veranschaulicht.

Besitzt das Objekt bereits ein Billboard aus einem vorherigen Frame, so werden für dieses zusätzlich noch drei weitere Scores berechnet, um zu entschieden, ob ein

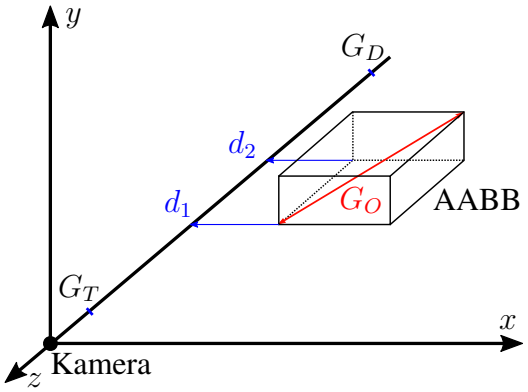

Abbildung 5: *Visualisierung, wie der Billboard Score berechnet wird. Es werden die Tiefeninformationen der AABB des Objektes (d_1, d_2) mit einem Schwellwert G_T verglichen, und außerdem wird noch die Größe G_O des Objektes berücksichtigt, definiert als die Distanz zwischen den beiden Extrempunkten der AABB. G_S ist die gesamte Szenentiefe.*

Update nötig ist (mehr dazu später in Kapitel 3.6).

Besitzt das Objekt hingegen noch kein Billboard, so wird der Score genutzt, um das Objekt in eine Priority-Queue (BCQ, Billboard Create Queue) einzureihen. Für die Objekte in dieser Queue werden später Billboards erstellt. Damit schnelle Rotationen der Kamera nicht zu stark schwankenden Framerates führen, weil plötzlich Hunderte Objekte auf einmal durch Billboards ersetzt werden müssen, ist die Queue jedoch in der Länge beschränkt (bspw. 64 Elemente).
Ist die BCQ bereits voll und wird ein neues Element hinzugefügt, fällt das Element mit dem niedrigsten Score aus der Queue heraus. Für dieses Objekt wird S_O auf 0 zurückgesezt.

Wurde der Object Score für alle Objekte berechnet, so wird daraufhin für jedes Objekt auf Basis dieses Scores entschieden, ob dieses überhaupt gerendert, normal gezeichnet oder durch ein Billboard ersetzt werden soll.
Ist S_O negativ, so wird das Objekt überhaupt nicht gezeichnet, denn es liegt außerhalb des sichtbaren Bereiches.

Hat ein Objekt aus einem vorherigen Frame ein Billboard und wird jedoch aktuell gar nicht oder normal gezeichnet, so wird dessen Billboard aus dem Billboard-Framebuffer entfernt. Dadurch steht der Slot im Framebuffer für andere Billboards wieder zur Verfügung.

Objekte bzw. Instanzen deren Object Score 0 ist, werden jetzt anschließend nor-

mal gerendert, während Objekte mit einem positiven Score durch Billboards ersetzt werden.

Der von Funkhouser und Séquin [FS93] vorgeschlagene Algorithmus zur LOD-Entscheidung basierend auf einer angestrebten Framerate wird nicht eingesetzt, da dies in bestimmten Fällen zu einer sehr schlechten Bildqualität führen kann, wenn bspw. aufgrund einer knappen Frametime ein lange nicht geupdatetes, fehlerhaftes Billboard für ein relativ nahe an der Kamera gelegenes Objekt eingesetzt wird.

3.4 Erstellen eines neuen Billboards

Anschließend werden für alle Objekte in der BCQ neue Billboards erstellt. Die Erstellung eines neuen Billboards erfordert einige Berechnungen, welche in diesem Abschnitt erläutert werden.

Zuvor eine kurze Erklärung zu der im Folgenden verwendeten Notation. Ein hochgestellter Buchstabe gibt das Koordinatensystem eines Punktes an, wobei ^{W} bedeutet, dass der Punkt im Weltkoordinatensytem liegt, ^{V} bezeichnet das Kamerakoordinatensystem und ^{N} meint das NDC-Koordinatensystem. Als Spezialfall existiert außerdem noch ^{X}. Dies bezeichnet einen Punkt im Kamerakoordinatensystem, der jedoch zusätzlich auf der Near-Plane der Kamera liegt.

In diesem Schritt wird jetzt die zuvor bereits für den Sichtbarkeitstest eingesetzte AABB in NDC-Koordinaten wieder benötigt. Von dieser werden das Minimum, bezeichnet als P^N_{min}, und das Maximum, bezeichnet als P^N_{max}, genutzt. Unter Verwendung des aktiven Viewports V können mit diesen beiden Positionen die Dimensionen des Objektes auf dem Bildschirm berechnet werden:

$$x_1 = P^N_{min}.x * \frac{V.w}{2} + \frac{V.w}{2} + V.x, \qquad y_1 = P^N_{min}.y * \frac{V.h}{2} + \frac{V.h}{2} + V.y$$
$$x_2 = P^N_{max}.x * \frac{V.w}{2} + \frac{V.w}{2} + V.x, \qquad y_2 = P^N_{max}.y * \frac{V.h}{2} + \frac{V.h}{2} + V.y$$

x_1 und x_2 geben dabei die minimalen bzw. maximalen x-Koordinaten des Objektes auf dem Bildschirm und y_1 bzw. y_2 die minimale und maximale y-Koordinate des Objektes auf dem Bildschirm an.
Daraus lassen sich Breite w und Höhe h des Objektes auf dem Bildschirm berechnen, wobei jedoch zu beachten ist, dass diese Dimensionen aufgrund der nicht-

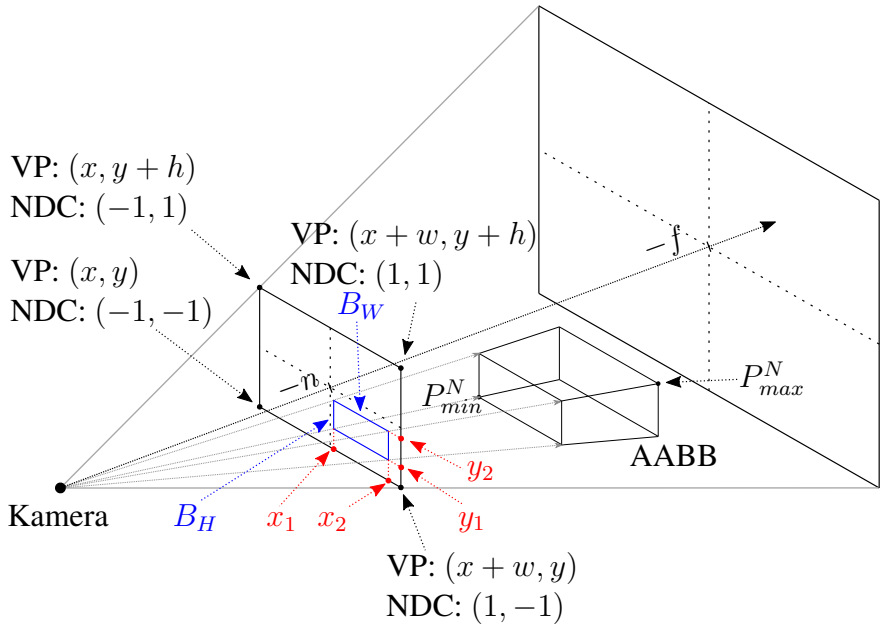

Abbildung 6: *Berechnung der Größe eines Objektes auf dem Bildschirm. Zuerst werden x_1, x_2, y_1, und y_2 berechnet und anschließend die Breite B_W und Höhe B_H des Billboards. VP bezeichnet die Koordinaten des jeweiligen Punktes in Viewport-Koordinaten und NDC die Koordinaten desselben Punktes im NDC-Koordinatensystem.*

optimalen AABB deutlich größer sein können, als das Objekt tatsächlich ist:

$$B_W = x_2 - x_1, \qquad B_H = y_2 - y_1$$

Diese Berechnungen werden in Abbildung 6 visualisiert.

Mithilfe der Größe des Objektes auf dem Bildschirm kann jetzt eine angemessene Framebuffergröße für das Billboard ausgewählt werden, wobei die Größen von 16x16 bis 512x512 Pixeln zur Verfügung stehen. Es wird hier die Dimension gewählt, deren Differenz zur Bildschirmgröße des Billboards minimal ist.
Führt dies zu unerwünschten Effekten, wie einer sichtbar geringen Auflösung vieler Billboards, könnte alternativ die Dimension auch immer aufgerundet werden. Da dieses Problem jedoch mit den getesteten Modellen nicht aufgetreten ist, wird in der Implementation die erste Methode verwendet.

Jetzt wird in Weltkoordinaten der Vektor von der Position der Kamera zum Mittelpunkt der Bounding Box bestimmt. Anschließend werden Minimum und Maximum der NDC-AABB auf die Dimension der Near-Plane der Kamera projiziert, wobei

nur die x- und y-Achsen berücksichtigt werden (da z die Tiefe abbildet, die uns in diesem Zusammenhang nicht interessiert).

Aufgrund der Wertebereiche der Koordinaten eines Punktes auf der Near-Plane in NDC-Koordinaten bzw. Kamerakoordinaten (die Koordinaten der Eckpunkte der Near-Plane in beiden Koordinatensystemen sind in Abb. 7 dargestellt) ist die Umrechnung zwischen diesen sehr einfach, da nur mit den Dimensionen der Near-Plane skaliert werden muss, wobei E_W die Breite und E_H die Höhe bezeichne.

Anschließend werden die auf der Near-Plane liegenden Punkte

$$P_{cc}^X = \begin{pmatrix} o \\ q \\ -n \end{pmatrix}, \qquad P_{tc}^X = \begin{pmatrix} o \\ E_H * y_2 \\ -n \end{pmatrix}, \qquad P_{rc}^X = \begin{pmatrix} E_W * x_2 \\ q \\ -n \end{pmatrix}$$

im Kamerakoordinatensystem konstruiert. o und q werden dabei berechnet als:

$$o = \frac{E_W * (x_1 + x_2)}{2}, \qquad q = \frac{E_H * (y_1 + y_2)}{2}$$

Die tiefergestellten Bezeichnungen $_{cc}$, $_{tc}$ und $_{rc}$ werden im Folgenden noch öfter Verwendung finden und stehen für Center-Center, Top-Center und Right-Center und bezeichnen jeweils den Mittelpunkt des Billboards, den Mittelpunkt der oberen Kante und den Mittelpunkt der rechten Kante des Billboards, wie in Abbildung 7 zu erkennen.

P_{cc}^X stellt den Mittelpunkt der vom Objekt verdeckten Fläche dar, während P_{tc}^X der Mittelpunkt der oberen Kante dieser Fläche und P_{rc}^X der Mittelpunkt der rechten Kante ist.

Diese Punkte sollen jetzt per Strahlprojektion[2] so verschoben werden, dass sie eine Ebene mit derselben Entfernung zur Kamera wie der Mittelpunkt des ersetzten Objektes bilden und außerdem noch an der perspektivisch korrekten Position sind.

Dazu müssen die Vektoren von der Kamera zu den Punkten P_{cc}^X, P_{tc}^X und P_{rc}^X entsprechend verlängert werden. Der Faktor e, um den diese Vektoren verlängert wer-

[2] Verlängerung eines Vektors in dessen Richtung um einen Faktor

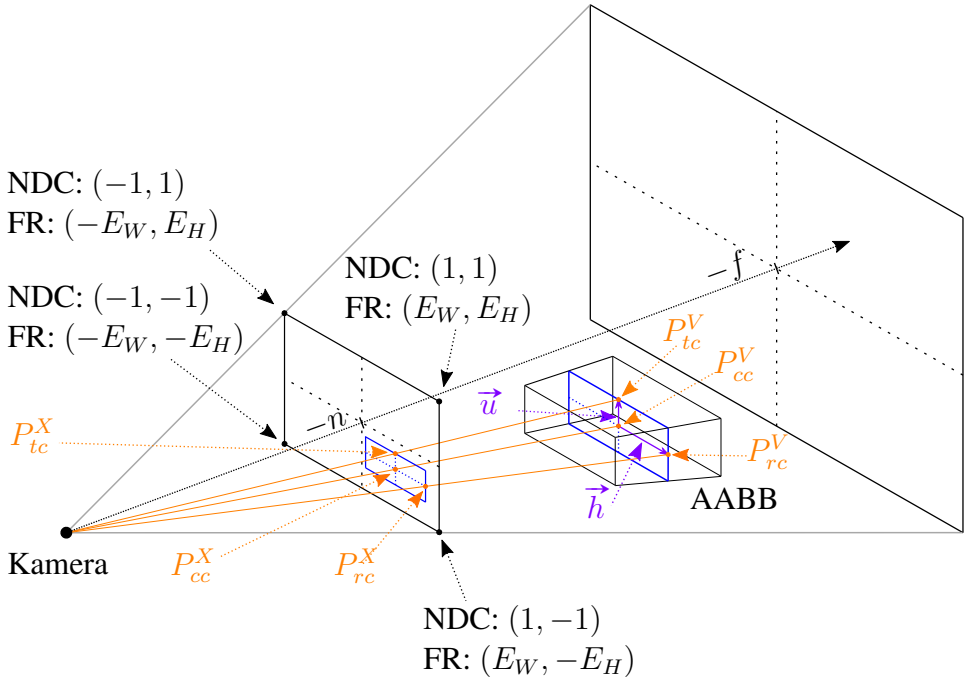

Abbildung 7: *Aus der Größe des Objektes auf der Near-Plane werden jetzt die Punkte P^X_{cc}, P^X_{tc} und P^X_{rc} bestimmt. Durch Strahlprojektion können aus diesen daraufhin die Punkte P^V_{cc}, P^V_{tc}, P^V_{rc} berechnet werden. Aus diesen wiederum können dann die horizontale ($\vec{h}$) und vertikale ($\vec{u}$) Ausrichtung bzw. Größe des Billboards im Kamerakoordinatensystem berechnet werden.*

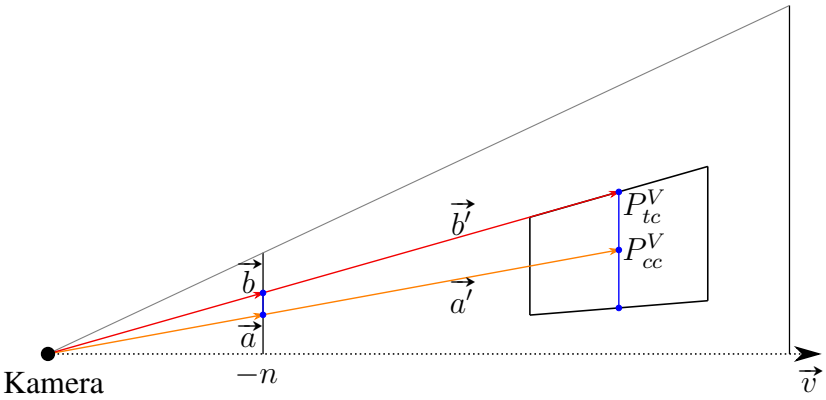

Abbildung 8: *Seitenansicht der Szene. Es sollen die Punkte P^V_{cc} und P^V_{tc} berechnet werden. Dies sind die Punkte auf einer Ebene mit derselben Entfernung zur Kamera wie der Mittelpunkt des Objektes. Da $\vec{a'}$ eine Verlängerung des Vektors $\vec{a}$ ist, kann dieser Vektor also ausgedrückt werden als $\vec{a'} = x * \vec{a}$, wobei x ein Faktor ist. Daraus folgt $P^V_{cc} = \vec{a} + \vec{a'} = (x+1) * \vec{a}$. Der Faktor x kann bestimmt werden als das Verhältnis der Längen der beiden Vektoren $\vec{a}$ und $\vec{a'}$. Damit folgt $P^V_{cc} = (1 + \frac{|\vec{a'}|}{|\vec{a}|}) * \vec{a}$. Laut des Strahlensatzes reicht es, dieses Verhältnis einmal für $\vec{a} = P^X_{cc}$, den Mittelpunkt, zu berechnen und dann auch für die anderen Punkte zu verwenden.*

den müssen, kann berechnet werden als:

$$e = 1 + \frac{d_{cc}}{P^X_{cc}.z}$$

Dies wird in Abbildung 8 erläutert.

Der Wert d_{cc} bezeichnet dabei die Tiefe des Objektmittelpunktes in Kamerakoordinaten, kann also beispielsweise mit den für die Berechnung der Größe eines Billboards bereits genutzten Tiefenwerten d_1 und d_2 berechnet werden als $d_{cc} = \frac{d_1+d_2}{2}$.

Dies liefert die folgenden Punkte, wobei P^V_{cc} der Mittelpunkt des zukünftigen Billboards ist, P^V_{tc} den Mittelpunkt der oberen Kante des Billboards bezeichnet und P^V_{rc} der Mittelpunkt der rechten Kante des Billboards ist und sind auch in Abbildung 7 dargestellt.

$$P^V_{cc} = e * P^X_{cc}, \qquad P^V_{tc} = e * P^X_{tc}, \qquad P^V_{rc} = e * P^X_{rc}$$

Der Mittelpunkt muss bestimmt werden, da dieser in den NDC-Koordinaten durch die perspektivische Verzerrung für am Rand des Kamera-Frustums gelegene Objek-

te vom Mittelpunkt in Weltkoordinaten abweichen kann. Dadurch kann der Weltmittelpunkt nicht als Mittelpunkt des Billboards verwendet werden.

Daraufhin werden aus diesen Positionen die Dimensionen des Billboards in Kamerakoordinaten berechnet:

$$\vec{h} = P_{rc}^V - P_{cc}^V\text{: Horizontale Dimension}$$
$$\vec{u} = P_{tc}^V - P_{cc}^V\text{: Vertikale Dimension}$$

Diese Vektoren haben immer nur eine Bewegungsrichtung, nämlich x für die Horizontale und y für die Vertikale.
Anschließend werden der Mittelpunkt, genauso wie der Vektor $\vec{h}$, in Weltkoordinaten zurücktransformiert und dann mit den Längen $|\vec{u}|$ und $|\vec{h}|$ der beiden Dimensionsvektoren gespeichert. Zusätzlich dazu wird noch die Entfernung des Objektes von der Kamera (d_{cc}) im Kamerakoordinatensystem gespeichert.

Jetzt wird die Projektionsmatrix P_B für das Billboard bestimmt. Dafür werden die zuvor bereits berechneten Punkte P_{min}^X und P_{max}^X verwendet und ein neues Frustum speziell für dieses Billboard bestimmt.

3.4.1 Viewport und Texturkoordinaten

Anschließend wird versucht, im Billboard-Framebuffer der passenden Dimension einen Platz für das Billboard zu reservieren. Ist in diesem kein Platz mehr frei, so werden zuerst die größeren Dimensionen auf einen freien Platz überprüft. Sind auch diese alle voll, werden anschließend noch die kleineren Dimensionen überprüft. Sind alle Billboard-Framebuffer voll, so wird der Erstellprozess abgebrochen.

Falls ein freier Platz gefunden wurde, werden zur späteren Verwendung der Viewport V_B und die Texturkoordinaten T des Billboards im Framebuffer berechnet. In den folgenden Formeln stellt c die Spalte und r die Reihe des Billboards im Framebuffer dar, während F_c bzw. F_r die Anzahl an Spalten bzw. Reihen von Billboards im Billboard-Framebuffer angeben. Die Zwischenergebnisse F_s und F_t geben die

Breite eines Billboards in Texturkoordinaten in s bzw. t-Richtung an.

$$V_B = \begin{pmatrix} c * B_D \\ r * B_D \\ B_D \\ B_D \end{pmatrix}, \qquad F_s = \frac{1}{F_c}, \qquad F_t = \frac{1}{F_r}, \qquad T = \begin{pmatrix} c * F_s \\ r * F_t \\ (c+1) * F_s \\ (r+1) * F_t \end{pmatrix}$$

Beispielhaft können die Eckpunkte des orange markierten Billboards aus Abbildung 4 (Seite 20) damit berechnet werden zu:

$$A = \begin{pmatrix} r * B_D \\ c * B_D \end{pmatrix} = \begin{pmatrix} r * F_s \\ c * F_t \end{pmatrix}, \qquad C = \begin{pmatrix} (r+1) * B_D \\ (c+1) * B_D \end{pmatrix},$$
$$B = \begin{pmatrix} r * B_D \\ (c+1) * B_D \end{pmatrix} = \begin{pmatrix} r * F_s \\ (c+1) * F_t \end{pmatrix}, \qquad D = \begin{pmatrix} r * B_D \\ c * B_D \end{pmatrix}$$

Anschließend kann das Billboard in den Framebuffer gezeichnet werden. Hierfür muss zuerst der Billboard-Framebuffer als aktiver Framebuffer eingestellt werden. Anschließend werden zuerst alte Farb- und Tiefeninformationen gelöscht und die Pixel auf einen Standardwert initialisiert. Die dazu verwendete Clear-Operation des OpenGL-Frameworks tut dies jedoch standardmäßig nur für den gesamten Buffer. Damit würden andere Billboards in dem Billboard-Framebuffer bei jedem Zeichnen eines neuen Billboards gelöscht. Um dies zu verhindern, wird deswegen einer der optionalen Tests am Ende der OpenGL-Pipeline aktiviert. Dieser *Scissor*-Test erlaubt es, den Bereich, in den Draw-Calls im Framebuffer zeichnen können, zu begrenzen. Indem mit diesem Test der aktive Bereich auf den Viewport des Billboards begrenzt wird, können einzelne Billboards gelöscht und neu gezeichnet werden. Anschließend muss noch die Projektionsmatrix P_B eingestellt werden. Jetzt kann mit dem auch vom Deferred Renderer zum Füllen des G-Buffers verwendeten Shader das Billboard in den Framebuffer gezeichnet werden.

Nach Zeichnen des Billboards sollten die vorherige Projektionsmatrix, Viewport und Framebuffer wiederhergestellt werden. Auch der Scissor-Test muss wieder deaktiviert werden.

3.5 Zeichnen der Billboards

Da die neuen Billboards nun fertig sind, können diese jetzt in der Szene gezeichnet werden.

3.5.1 Billboard Buffer

Dazu wird zuerst darauf eingegangen, welche Informationen für ein Billboard gespeichert werden.

Als wichtigste Information sei da der zuvor berechnete Mittelpunkt des Billboards P_{cc}^{V}, allerdings wird dieser in Weltkoordinaten zurücktransformiert. Außerdem werden die Längen $|\vec{h}|$ und $|\vec{u}|$ gespeichert. Es werden nicht die Vektoren selber gespeichert, da dies deutlich mehr Speicherplatz benötigen würde. Aufgrund der Ausrichtung der Billboards an den Achsen des Kamerakoordinatensystems können diese Vektoren aus deren Länge außerdem problemlos rekonstruiert werden.
Außerdem wird die Entfernung des Billboards von der Kamera zum Erstellzeitpunkt gespeichert. Hinzu kommen noch die Bindless Texture Handles der vier Texturen des Billboard-Framebuffers und die Texturkoordinaten des Billboards im Billboard-Framebuffer.

Für die Billboards wird nun ein neues Vertex Array Object erstellt, welches Vertexattribute für all diese Informationen bereitstellt. Dieses wird im Folgenden als Billboard Buffer bezeichnet.

Zusätzlich verwendet der Billboard Buffer einen Indexbuffer, um nicht bei jeder Aktivierung oder Deaktivierung eines Billboards dieses komplett aus dem Buffer löschen zu müssen. Denn dies hätte zur Folge, dass der gesamte Buffer in jedem Frame neu zur GPU hochgeladen werden müsste, was bedeutenden Overhead verursacht und unnötig viel Bandbreite beansprucht. Mit dem Indexbuffer muss immerhin nur dieser in jedem Frame geupdatet werden. Aus diesem Grund speichert das Billboard auf der CPU-Seite außerdem noch den Index, unter dem es im Billboard Buffer gespeichert ist.

Dieser Billboard Buffer ist in Abbildung 9 schematisch dargestellt. Auf der linken Seite ist dabei Frame A dargestellt, in dem alle Billboards, die aktuell im Buffer enthalten sind, aktiv sind (verwendet werden), und rechts Frame B, in dem nur drei

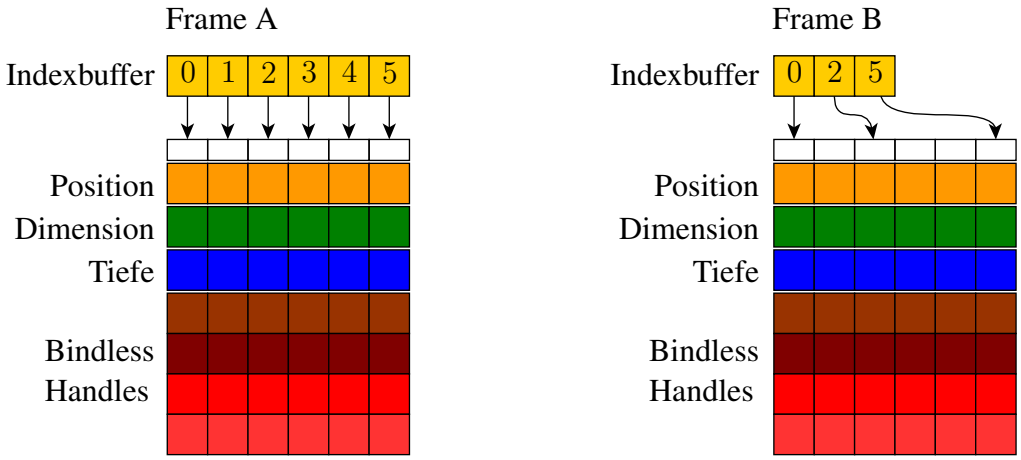

Abbildung 9: *Schematische Darstellung des Billboard Buffers mit den verschiedenen pro Billboard gespeicherten Informationen. Links ist dabei der Zustand in Frame A dargestellt. In diesem Frame sind alle Billboards aktiv, weshalb sie alle im Indexbuffer einen Eintrag besitzen. In Frame B sind jedoch nur die Billboards* 0*,* 2 *und* 5 *aktiv, sodass auch nur diese im Indexbuffer vorhanden sind. Die Informationen der anderen Billboards bleiben zwar in dem Buffer gespeichert, werden aber für diesen Frame nicht verwendet.*

der Billboards noch aktiv sind. Zu beachten ist, dass die Informationen der inaktiven Billboards in Frame B nicht aus dem Buffer gelöscht wurden, sondern nur der Index der nicht verwendeten Billboards aus dem Indexbuffer entfernt wurde.

3.5.2 Billboard Geometry Shader

Im Billboard Buffer hat jedes Billboard einen einzigen Vertex, dementsprechend werden diese als Punktprimitive gezeichnet und mit einem Geometry-Shader die Quads, auf die das Billboard letztendlich gezeichnet wird, rekonstruiert.

Aus den Dimensionen des Billboards $|\vec{h}|$ und $|\vec{u}|$ können die Kantenvektoren des Billboards einfach rekonstruiert werden, da die Billboards an den Achsen des Kamerakoordinatensystems ausgerichtet sind:

$$\vec{i} = |\vec{h}| * \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad \vec{j} = |\vec{u}| * \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

$\vec{i}$ bezeichnet dabei den horizontalen Ausrichtungsvektor und $\vec{j}$ den vertikalen. Zusammen mit dem Mittelpunkt des Billboards ergeben sich die Eckpunkte in Kamerakoordinaten dann als

$$E_{tl} = M - \vec{i} + \vec{j}, \qquad E_{tr} = M + \vec{i} + \vec{j},$$
$$E_{ll} = M - \vec{i} - \vec{j}, \qquad E_{lr} = M + \vec{i} - \vec{j},$$

Diese werden nun vom Geometry-Shader ausgegeben und anschließend werden mit einem Fragment-Shader die Texturen des Billboards auf das Quad aufgetragen. Im Falle des Forward Renderers muss dieser außerdem noch beleuchten. Dieser Schritt fällt für den Deferred Renderer weg, da die Platzierung der Billboards vor dem Lighting-Pass erfolgt, und fast (siehe nächster Absatz) nur eine Kopieroperation zwischen dem G-Buffer des Billboards und dem G-Buffer des Deferred Renderers ist.

3.5.3 Tiefenkorrektur

Für beide Shader ist es jedoch nötig, die Tiefe des Billboards zu korrigieren. Denn in der Tiefentextur des Billboards ist die Tiefe zum Zeitpunkt, zu dem das Billboard erstellt wurde, nicht-linear gespeichert. Bereits minimale Änderungen der Kameraposition würden ohne Korrektur dazu führen, dass die Tiefe des Objektes auf einmal sichtbar fehlerhaft ist (da sich die Entfernung von Objekt zu Kamera geändert hat, nicht jedoch die im Billboard gespeicherte Tiefe).

Dies ist auch der Grund, weshalb die Tiefe bei Erstellung des Billboards (d_{cc}) gespeichert wurde. Zuerst muss der Tiefenwert d_t aus der Textur in eine lineare Skalierung umgerechnet werden:

$$d_l = 2 * \frac{n * f}{f + n - (2 * d_t - 1) * (f - n)}$$

f und n sind die z-Koordinaten der Near- und Far-Plane. Durch Berechnung der Differenz $\Delta d = d_l - d_{cc}$ kann nun mit $d_k = d_a + \Delta d$ die nahezu korrekte Tiefe d_k des Objektes rekonstruiert werden. d_a ist dabei die aktuelle Tiefe des Billboard Mittelpunktes.

Anschließend muss diese lineare Tiefe zur Verwendung in der OpenGL-Pipeline wieder nicht-linear skaliert werden:

$$d_n = -\frac{(f+n) - 2fn}{2(f-n)} * d_k$$

3.6 Entscheidung zum Update eines Billboards

Wie bereits erwähnt, müssen die Billboards manchmal neu gezeichnet werden, da sich beispielsweise die Kameraposition geändert hat, wodurch nun vorher unsichtbare Bereiche des Objektes sichtbar sein müssen.

In diesem Kapitel sollen die verschiedenen Fälle, in denen ein Update durchgeführt werden muss, untersucht werden.

Als erster Indikator, ob ein existierendes Billboard ein Update benötigt, bildet der schon für die Entscheidung zum Erstellen bzw. Zeichnen eines Billboards genutzte Score U_B. Denn es wäre nicht sinnvoll ein Billboard zu updaten, welches aktuell gar nicht verwendet wird. Stattdessen wird der Platz im Billboard-Framebuffer für solch ein nicht mehr benötigtes Billboard freigegeben, der Eintrag im Billboard Buffer bleibt jedoch bestehen.

Natürlich reicht U_B aber nicht zur Entscheidung aus, da sonst jedes existierende, aktuell verwendete Billboard in jedem Frame neu gezeichnet würde.

Basierend auf den bereits von Schaufler ([Sch95]) beobachteten Fällen, in denen ein Update nötig ist, werden drei Heuristiken verwendet. Aufgrund der bei G-Buffer-Billboards hinzugekommenen Tiefen und der Ausrichtung der Billboards an den Achsen des Kamerakoordinatensystems reicht die Verwendung der zwei von Schaufler vorgeschlagenen Heuristiken nicht aus.

3.6.1 Update wegen Änderung der Kamerablickrichtung

Denn auch eine (von Schaufler nicht erfasste) Rotation der Kamerablickrichtung, die zu einer Rotation des Kamerakoordinatensystems führt, verursacht für ein G-Buffer-Billboard einen sichtbaren Fehler, da das Billboard ohne diese Heuristik oft-

mals an der falschen Stelle positioniert wird und dessen Ausrichtung nicht mehr stimmt.

Berechnet wird diese Heuristik, indem der Winkel θ zwischen der alten horizontalen Ausrichtung des Billboards $\vec{h}$ mit einem neuen $\vec{h}'$ verglichen wird. $\vec{h}'$ ist eine für das Billboard mit der aktuellen Kamera neu bestimmte Ausrichtung. Diese Heuristik ist der Grund, weshalb $\vec{h}$ in Weltkoordinaten mit dem Billboard gespeichert wird.

Ab einem bestimmten Fehlerwinkel wird das Billboard nun geupdatet, wobei sich $2.5°$ als sehr guter Wert erwiesen hat, mit einem fast nicht sichtbaren Fehler, und $5°$ noch gute Resultate liefert, also nur hin und wieder zu einem wahrnehmbaren Fehler führt. Je geringer θ gewählt wird, desto genauer bleiben natürlich die Billboards, aber desto öfter müssen die Billboards auch geupdatet werden. Dieser Fall ist in Abbildung 10 oben rechts dargestellt.

3.6.2 Update wegen seitlicher Änderung der Kameraposition

Die beiden folgenden Heuristiken decken sehr ähnliche Fälle wie die von Schaufler entwickelten Heuristiken ab, sind jedoch etwas einfacher.

Die zweite Heuristik überwacht dabei die seitliche Positionsänderung der Kamera, indem der Winkel ϕ zwischen den Vektoren von der Kamera zum Mittelpunkt des Billboards einmal zum Erstellzeitpunkt und ein zweites Mal zum aktuellen Zeitpunkt berechnet wird.

Für diese Heuristik wird der Grenzwert etwas niedriger gewählt, da es ansonsten häufiger zu Artefakten kommt. Ein Wert von $1.0°$ liefert hier sehr gute Ergebnisse mit einem fast nicht wahrnehmbaren Fehler und bis ca. $4°$ ergibt sich noch eine gute Bildqualität. Dieser Fall ist in Abbildung 10 unten links dargestellt.

3.6.3 Update wegen Bewegung in Richtung des Billboards

Mit diesen beiden Heuristiken können jetzt Änderungen der Blickrichtung und der seitlichen Positionsänderung der Kamera detektiert werden. Eine Bewegung der Kamera in Richtung des Vektors $\vec{w}$, der von der Kameraposition zum Mittelpunkt des Objektes zeigt, kann jedoch noch nicht erfasst werden. Denn in diesem Fall bleiben

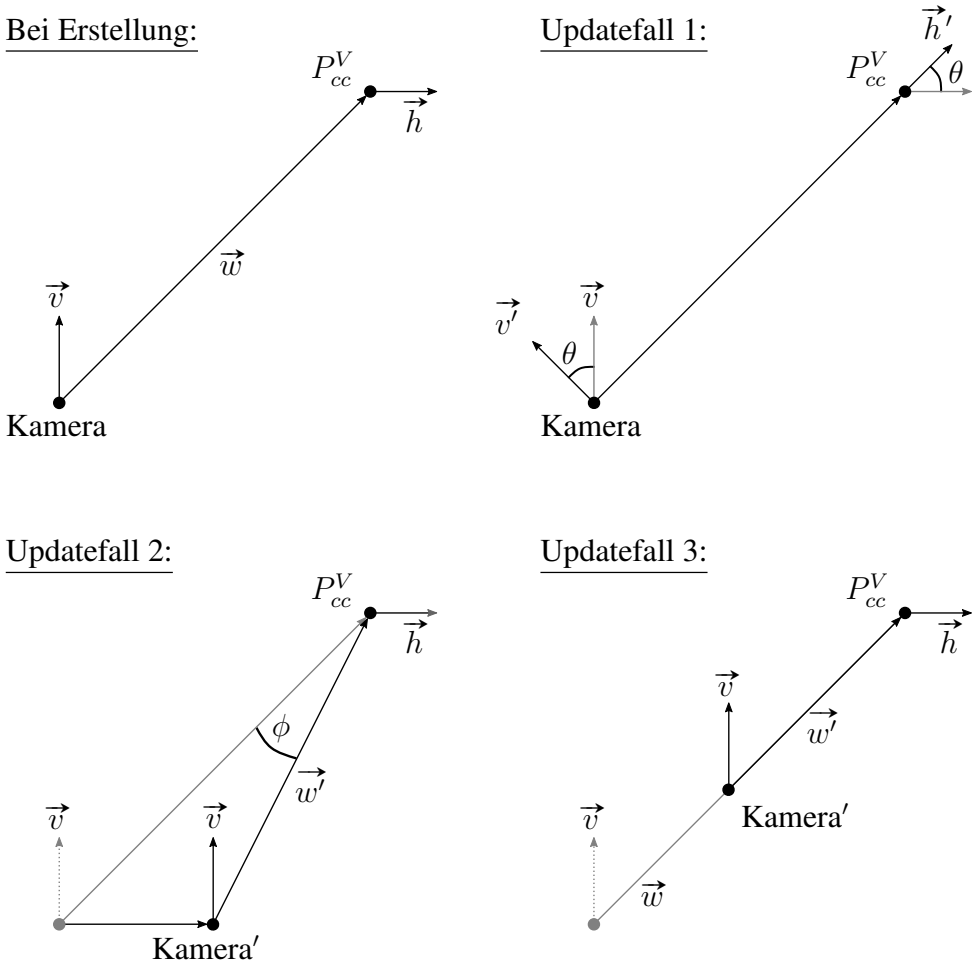

Abbildung 10: *Visualisierung, wie die verschiedenen Scores für das Updaten eines Billboards berechnet werden. Oben links ist der Zustand bei Erstellung des Billboards dargestellt. w ist dabei der Vektor von der Position der Kamera zum Mittelpunkt P_{cc}^V des Billboards.*
Oben rechts ist der erste Updatefall bei Veränderung der Blickrichtung illustriert, wobei θ berechnet wird. Dies ist der Winkel zwischen alter und neuer Blickrichtung bzw. alter und neuer horizontaler Ausrichtung des Billboards.
Unten links ist dann der zweite Fall bei seitlicher Bewegung der Kamera gezeichnet, wobei der Winkel ϕ zwischen altem w und neuem w' berechnet wird.
Unten rechts: Der dritte Updatefall, wenn sich die Kamera auf das Objekt zubewegt oder von diesem entfernt hat. Altes w und neues w' zeigen in dieselbe Richtung, haben jedoch unterschiedliche Längen. Diese Längen werden nun ins Verhältnis gesetzt.

die beiden zuvor berechneten Winkel θ und ϕ beide $0°$.

Dieser Fall entspricht dem zweiten schon von Schaufler erkannten Updategrund, jedoch wird auch hier wieder eine einfachere Heuristik verwendet. Schaufler hat eine Heuristik auf Basis der Größe des Objektes auf dem Bildschirm verwendet; in dieser Implementation werden bloß die Längen der Vektoren $\vec{w}$ und $\vec{w'}$ verglichen, da diese einfacher zu berechnen sind. Diese Vektoren stellen beide den Vektor von der Kamera zum Billboardmittelpunkt P_{cc}^{V} dar, allerdings zu unterschiedlichen Zeitpunkten. Ersterer wird bei Erstellung des Billboards bestimmt, während letzterer zum aktuellen Zeitpunkt berechnet wird.

Da die Längen dieser Vektoren einen unbeschränkten Wertebereich haben, wird aus diesen ein Verhältnis berechnet.

Dadurch wird sowohl die Angabe eines Schwellwertes vereinfacht als auch eine einfachere Gewichtung der Heuristiken ermöglicht. Um diese Gewichtung noch weiter zu vereinfachen, werden die Werte dieser Heuristik außerdem noch in den Wertebereich $[0, 180]$ skaliert, da die beiden anderen Heuristiken bereits Ausgaben in diesem Wertebereich produzieren. Dazu wird zuerst das Verhältnis berechnet und anschließend 1 abgezogen. Vom Ergebnis wird dann nur der absolute Wert berücksichtigt, auf den Wertebereich [0, 1] beschränkt und mit 180 skaliert.

Die komplette Formel zur Berechnung ergibt sich also zu:

$$\psi = 180 * \max(0, \min(1, |\frac{|\vec{w}|}{|\vec{w'}|} - 1|))$$

Ein Schwellwert von 2.5% oder skaliert 4.5 hat sich als guter Wert erwiesen. Diese Heuristik ist in Abbildung 10 unten rechts dargestellt.

Natürlich werden in einer durchschnittlichen Situation vermutlich alle drei Heuristiken gleichzeitig aktiv sein, sodass eine gewichtete Verwendung aller drei berechneten Scores sinnvoll ist.

3.7 Durchführung des Updates

In diesem Kapitel soll erklärt werden, wie der Updateprozess für ein Billboard funktioniert.

3.7.1 Gewichtung der Update-Heuristiken

Wie auch bei der Erstellung existiert auch für das Updaten der Billboards eine Priority-Queue mit maximaler Größe, sodass Billboards, die sehr falsch sind, möglichst schnell geupdatet werden, jedoch auch nicht zu viele Updates in einem Frame durchgeführt werden. Diese Priority-Queue kann jedoch nur einen Score berücksichtigen, weshalb die drei zuvor gebildeten Heuristiken zu einem Wert zusammengefügt werden müssen. Hierfür werden die drei einzelnen Scores mit einer 1:1:1-Gewichtung aufsummiert, weshalb auch die Skalierung der dritten Heuristik in den Wertebereich der anderen Heuristiken stattfindet. Mit diesem Ansatz ist es möglich, dass eine einzelne Heuristik sehr stark ausschlägt und deswegen ein Update anstößt, es kann aber auch ein Update stattfinden, wenn alle drei Heuristiken nur schwach auslösen.

Alternative Heuristiken an dieser Stelle könnten eine aufwändigere Gewichtung beeinhalten oder bspw. immer nur das Maximum der drei Werte berücksichtigen. Da mit der verwendeten Heuristik die Ergebnisse jedoch zufriedenstellend waren, wurden diese Alternativen nicht getestet.

Genauso wie bei der Erstellung von Billboards werden Objekte, die aufgrund eines zu niedrigen Scores aus der Queue herausgefallen sind, normal gezeichnet. Dies geschieht zum Erhalt der Bildqualität, führt jedoch besonders bei starken, schnellen Bewegungen der Kamera temporär zu wenigen aktiven Billboards und damit zu einer geringeren Framerate. Dies kann beispielsweise später in den Performancemessungen mit nur 16 Updates pro Frame beobachtet werden (links in den Abbildungen 14 und 15 auf den Seiten 48 und 49).

Dieser Effekt lässt sich verringern, wenn die maximale Anzahl an erlaubten Billboard-Updates hochgesetzt wird. Allerdings sollte diese natürlich auch nicht zu hoch sein, da sonst zu viele Billboards pro Frame geupdatet werden können, wodurch die Performance ebenfalls negativ beeinflusst wird.

3.7.2 Erneutes Zeichnen

Das Update eines Billboards funktioniert sehr ähnlich wie die Erstellung eines neuen Billboards. Einzig ein Check, ob die bisherige Größe des Billboards noch angemessen ist, kommt hinzu. Hierfür wird zuerst, wie in Kapitel 3.4 (Seite 26) be-

schrieben, die Größe des Objekts auf dem Bildschirm neu berechnet. Unterscheidet sich die bisherige Größe von der neuen Größe, so wird der Platz im Billboard-Framebuffer der alten Größe freigegeben und ein Platz im Billboard-Framebuffer für die neue Dimension angefordert, ansonsten wird der Platz wiederverwendet.

Die restlichen Berechnungen verlaufen genauso wie bereits bei der Erstellung in Kapitel 3.4 beschrieben.

4 Performance

In diesem Kapitel soll die Performance der G-Buffer-Billboards vorgestellt werden, ebenso wie Schwächen des Verfahrens.

Dazu werden zwei verschiedene Szenen und Testfälle verwendet. Einerseits wurden die Auswirkungen von verschiedenen Schwellwerten für die drei Updateheuristiken auf die Bildqualität überprüft, und andererseits wurde die Performance unter Verwendung verschiedener Werte für die Anzahl pro Frame erlaubter Updates gemessen.

4.1 Paris Interior

Zur Verdeutlichung der Artefakte, die durch dieses Verfahren auftreten können, sind der digitalen Version dieser Arbeit Videoaufzeichnungen verschiedener Durchläufe eines Kamerapfades durch den Innenbereich des „Amazon Lumberyard Bistros“ [Lum17] beigelegt.

Diese sind im Unterordner *Video* auf der CD zu finden. Während der Aufzeichnung wurde immer der Deferred Renderer eingesetzt. Die mit „Bistro_X_Y_Z_W“ bezeichneten Dateien stellen dabei die Aufzeichnungen mit Billboards dar, wobei X, Y, Z und W die Schwellwerte für die Erstellung und die drei Updateheuristiken angeben. Die Datei „Bistro_Reference“ stellt eine Referenzaufnahme ohne Billboards dar. Die Dateien „Vespa_Billboards“ und „Vespa_Reference“ zeigen außerdem noch den Kamerapfad durch die zweite verwendete Szene „Vespa und Stühle“, einmal mit aktivierten und einmal mit deaktivierten Billboards. Einzelne Frames aus diesen Aufzeichnungen werden jedoch auch hier gezeigt, da an diesen auch ohne die

Videos die auftretenden Artefakte deutlich werden.

Die Videoaufzeichnungen wurden auf einem System mit einer AMD RX480 mit 8GB VRAM und einem AMD Ryzen 7 1700 mit 32 GB Arbeitspeicher aufgenommen, da hier die Videoaufzeichnung durch eine von AMD bereitgestellte Software ohne Beeinflussung der Performance möglich ist. Die Performancemessungen des Verfahrens wurden jedoch auf einem anderen Computer durchgeführt.
Denn die Grafikkarte wurde in dem AMD-System häufig nicht ausgelastet, da der CPU-Thread die Kommandos nicht schnell genug absenden konnte. Da diese schlechte Performance mit einem zweiten System mit anderer Hardware nicht auftritt, wird dies vermutlich durch eine schlechte Treiberimplementation verursacht.

Dieses andere System ist ein Laptop, ausgestattet mit einer NVIDIA Geforce 940M mit 4GB VRAM und einem Intel Core I7-5500U mit 8GB Arbeitsspeicher. Da dieser jedoch keine Möglichkeit bietet, ohne Performanceverlust Videoaufzeichnungen durchzuführen, wurden beide Systeme eingesetzt.

4.1.1 Billboard Schwellwert

Als erstes sollen die Auswirkungen des Schwellwerts, der für die Entscheidung zum Ersetzen durch ein Billboard verwendet wird, untersucht werden. Dazu wurde der Kamerapfad durch das Bistro mit drei verschiedenen Schwellwerten, 10%, 20% und 40% der Szenentiefe, durchgeführt. Dadurch wurde festgestellt, dass für diese Szene bei 10% die Artefakte der Updates, auch mit sehr geringen Schwellwerten, deutlich wahrnehmbar waren, da viele relativ nahe an der Kamera gelegene Objekte oft geupdatet werden, wodurch häufig *Popping-Artefakte* entstehen. Mit 20% sind die meisten Artefakte bei guter Wahl der Schwellwerte jedoch fast nicht zu erkennen. Dies verbessert sich bei 40% noch weiter, jedoch werden hier nur noch sehr wenige Objekte durch Billboards ersetzt.

4.1.2 Fehler bei Kamerarotationen

Aufgrund sehr gut aussehender Resultate bei Verwendung von 20% der Szenentiefe als Schwellwert zur Entscheidung zwischen normalem Objekt und Billboard wurde dieser anschließend nicht weiter verändert, sondern stattdessen die Schwellwerte der verschiedenen Updateheuristiken untersucht.

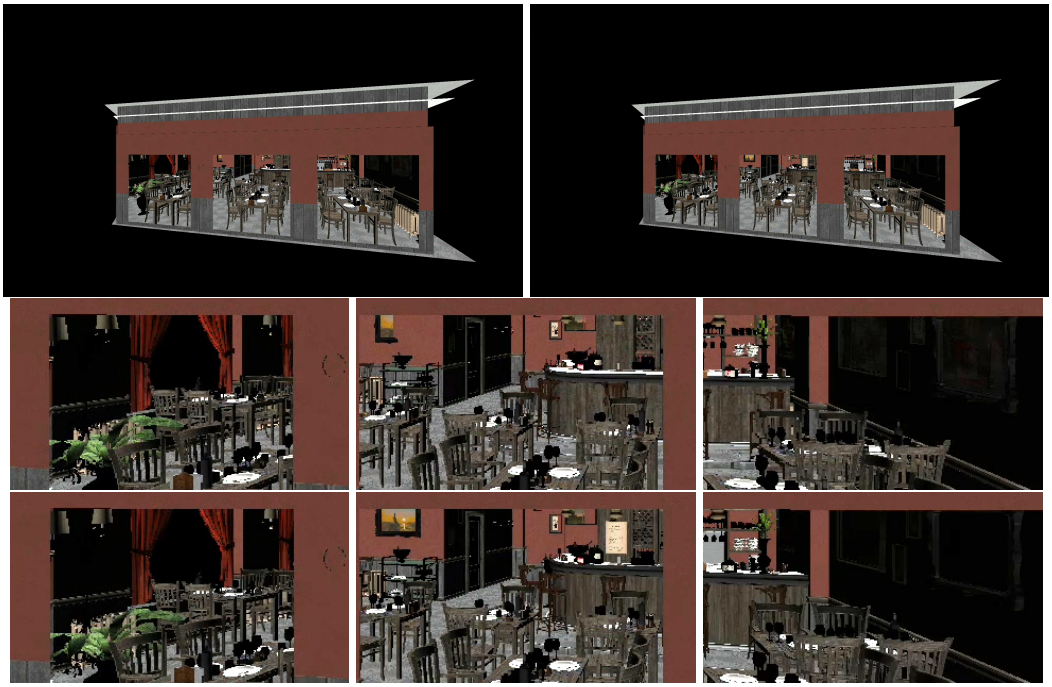

Abbildung 11: *Zwei aufeinanderfolgende Frames aus einem Durchlauf des Kamerapfades durch das Bistro mit den Schwellwerten* 20% *für die Entscheidung Billboard/Objekt,* $10°$ *für den Blickwinkel* θ*,* $5°$ *für* ϕ*, sowie skaliert 15 bzw.* 8.3% *Fehler als Schwellwert für die dritte Heuristik. In diesem sind Artefakte aufgrund eines zu hohen Winkels einer der beiden ersten Heuristiken zu erkennen. Oben links ist dabei der erste Frame, oben rechts der direkt folgende Frame. In der mittleren Zeile sind Detailansichten aus dem ersten Frame und in der dritten Zeile welche aus dem folgenden Frame zu sehen.*

Bei Rotation der Kamera verschiebt ein Billboard seine Position und Ausrichtung, da sich die Achsen des Kamerakoordinatensystems verändern und damit die Ausrichtung des Billboards. Dies resultiert in leicht verschobenen oder verdrehten Objekten, sichtbar bspw. in Abbildung 11.

In den Detailansichten der Abbildung 11 sind verschiedene Objekte dargestellt, die eine falsche Positionierung und Ausrichtung in Frame 1 haben und in Frame 2 aktualisiert werden, sodass diese plötzlich wieder in Position „ploppen“. Links ist dies bspw. der hintere Tisch, mittig sind dies Tresen, Speisekarte und das Bild und rechts die Stühle des hinteren Tisches und der Tresen. Besonders an der Speisekarte und dem Bild ist zu erkennen, dass diese falsche Ausrichtung in bestimmten Fällen auch zu einer fehlerhaften Tiefeninteraktion mit anderen Objekten führen kann.

Zu beachten ist dabei allerdings, dass die Identifikation eines Billboards als fehlerhaft anhand nur eines Frames mit dem bloßen Auge häufig schwierig ist. So sieht der fehlerhafte Frame 1 mit bloßem Auge ohne Referenzbild größtenteils durchaus plausibel aus. Erst die plötzliche Aktualisierung in Frame B und das dadurch verursachte Springen des Objektes lenken die Aufmerksamkeit des Betrachters besonders in der Videoaufnahme auf diese Fehler.

4.1.3 Fehler durch Positionsänderung

Wird für die zweite Heuristik ein schlechter Schwellwert gewählt, kommt es je nach Bewegungsrichtung zu sehr ähnlichen Artefakten. Bei Kamerabewegung in x- oder y-Richtung des Kamerakoordinatensystems sind die Artefakte ebenfalls verdrehte und verschobene Objekte.

Findet jedoch eine Bewegung in Blickrichtung der Kamera statt, so kommt es zu anderen Artefakten. Die durch Billboards ersetzten Objekte scheinen dabei in der Größe zu schrumpfen, sind jedoch am Mittelpunkt des Billboards fixiert. Sobald die Heuristik den Schwellwert überschreitet, „ploppen“ diese wieder in korrekter Größe an die korrekte Position zurück.

Anders als die in Abbildung 11 beobachteten Artefakte sind die durch eine falsche Größe des Billboards verursachten Artefakte deutlich wahrnehmbar (siehe Abbildung 12) und bereits ohne die Detailansichten zu erkennen.

Dieses Artefakt ist sowohl in den Einzelbildern als auch im Video sehr prominent

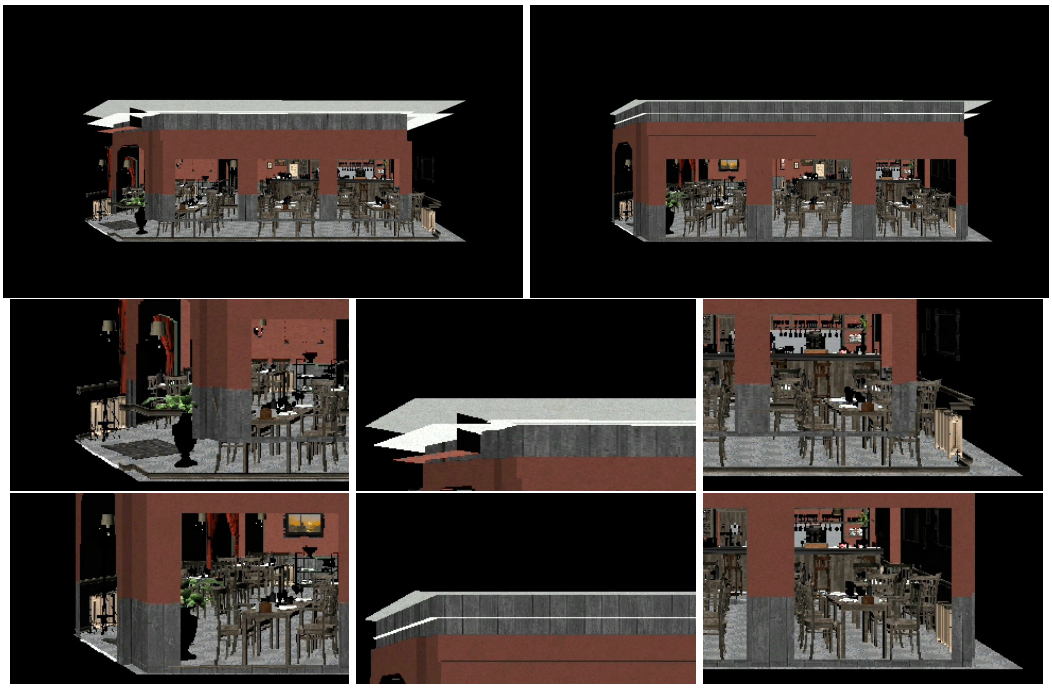

Abbildung 12: *Zwei aufeinanderfolgende Frames aus einem Durchlauf des Kamerapfades durch das Bistro mit den Schwellwerten* 20% *für die Entscheidung Billboard/Objekt,* $10°$ *für den Blickwinkel* θ*,* $5°$ *für Positionswinkel* ϕ*, sowie skaliert 15 bzw.* 8.3% *Fehler als Schwellwert für die dritte Heuristik. In diesem sind Artefakte aufgrund einer fehlerhaften Größe der Billboards zu erkennen. Oben links ist dabei der erste Frame, oben rechts der direkt folgende Frame. In der mittleren Zeile sind Detailansichten aus dem ersten Frame und in der dritten Zeile welche aus dem folgenden Frame zu sehen.*

und wird ebenfalls durch das Springen des Billboards in die korrekte Position in Frame 2 noch verstärkt.

Eine falsche Wahl des Schwellwertes für die dritte Heuristik verursacht ebenfalls diese Artefakte.

4.2 Vespa und Stühle

Die zweite verwendete Szene ist als Stresstest gedacht. Hierbei wird die Zeit, die zum Berechnen eines Frames benötigt wird, gemessen und dabei die maximale Anzahl an Billboard Updates pro Frame variiert.

Für diese Szene werden die Vespa aus dem Außenbereich des „Amazon Lumber-

yard Bistros“ und ein Stuhl aus dem Innenbereich entliehen und per Instancing[3] vielfach gezeichnet.

Die im Folgenden verwendete Testszene besteht aus je 500 Vespas und Stühlen, die zufällig in einer Ebene verteilt werden. Dabei werden die x- und z-Koordinaten, nicht jedoch die y-Koordinate, sowie die Rotationswinkel in allen drei Achsen, zufällig gewählt, sodass die Instanzen auf einer Ebene angeordnet werden.

Für diese Testszene wurde ebenfalls ein Kamerapfad definiert, der auch in zwei Videoaufzeichnungen einmal mit („Vespa_Billboards“) und einmal ohne Billboards („Vespa_Reference“) nachvollzogen werden kann.
Zuerst wurde die Performance des Deferred Renderers mit der eines Forward Renderers verglichen, wobei zuerst keine Billboards aktiviert waren, und dann Durchläufe mit 16, 64 und 256 Billboard Updates pro Frame durchgeführt wurden.

Experimentell wurde dasselbe Setup auch mit je 1000 Instanzen getestet. Es wurde jedoch nicht weiter verwendet, da aufgrund der Anzahl an Objekten die Frametimes in Bereichen mit wenigen aktiven Billboards auf teilweise über 100 Millisekunden ansteigen. Dies ist weit unter den für eine für Echtzeitdarstellung mindestens nötigen 15 FPS. Wie in Abbildung B.4 im Anhang zu erkennen, funktioniert das Verfahren für diese Menge an Objekten jedoch ebenfalls, solange die Anzahl an verwendeten Billboards hoch bleibt.

Die Rohdaten der Performancemessungen sind im Unterordner *Data* auf der CD zu finden und folgen dem Bezeichnungsschema „Modell_Renderer_Updategrenze“. Hierbei gibt die Updategrenze *none* die Aufzeichnung mit deaktivierten Billboards an. Zusätzlich dazu existieren Performancemessungen der „Vespa und Stühle“-Szene mit der höheren Anzahl Instanzen. Diese sind in der Form „Vespa_big_Updategrenze“ bezeichnet, da dies nur mit dem Deferred Renderer aufgezeichnet wurde.

In den folgenden Diagrammen existieren hin und wieder einzelne Frames mit extrem hohen Frametimes, wobei diese zwischen verschiedenen Durchläufen mit derselben Konfiguration in etwa an der gleichen Stelle auftreten. Es ist allerdings nicht klar, wodurch diese Spikes verursacht werden. Da diese jedoch sowohl mit deakti-

[3] Technik, mit der die Geometrie eines Objektes nur einmal gespeichert werden muss und anschließend mit vielen verschiedenen Transformationsmatrizen gezeichnet werden kann. Dadurch kann das Objekt vielfach in der Szene platziert werden, ohne mehr Speicherplatz zu verbrauchen

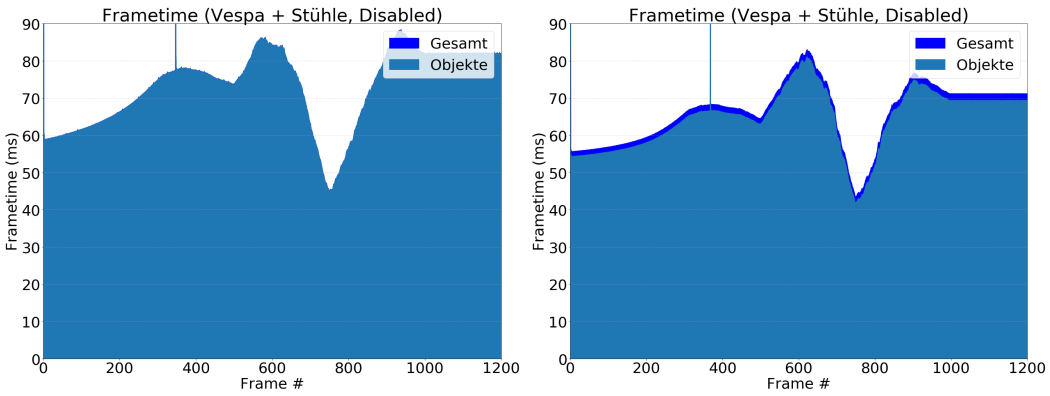

Abbildung 13: *Performance des Forward Renderers (links) und des Deferred Renderers (rechts) mit deaktivierten Billboards. Im helleren Blau ist dabei die Zeit, die zum normalen Rendern der Objekte benötigt wird, und im dunkleren Blau die gesamte Frametime aufgetragen. Wie zu erwarten sind diese Zeiten für den Forward Renderer identisch, während der Deferred Renderer für die Objekte etwas kürzer braucht, dafür jedoch durch den zusätzlichen Framebuffer Overhead verursacht.*

vierten Billboards als auch mit aktivierten Billboards auftreten, werden diese nicht weiter beachtet.

In Abbildung 13 sind dabei die Frametimes während eines Durchlaufs durch diese Szene mit dem Forward Renderer und dem Deferred Renderer gezeigt, wobei Billboards komplett deaktiviert waren. Es werden also alle sichtbaren Objekte normal gezeichnet. Dabei ist zu erkennen, dass Deferred und Forward Renderer in etwa gleich schnell sind und für die Szene zwischen ca. 45 und 85 ms benötigen. Dies ist zurückzuführen auf die Tatsache, dass in beiden Fällen nur mit einer globalen Lichtquelle beleuchtet wird. Nur gegen Ende des Kamerapfades kann der Deferred Renderer einen Performancevorteil von ca. 10 ms erreichen.

Diese Beobachtung bestätigt sich auch bei aktivierten Billboards, wie in den Grafiken im Anhang verifiziert werden kann. In Anhang A sind dabei die Performancemessungen des Bistro-Modells und in Anhang B Performancemessungen der „Vespa ud Stühle“-Szene immer für die Updateraten 16, 64 und 256 jeweils mit dem Deferred und mit dem Forward Renderer dargestellt.

Deshalb wird in Abbildung 14 nur noch der Deferred Renderer betrachtet und stattdessen verschiedene Updateraten miteinander verglichen. Auf der linken Seite wurden pro Frame maximal 16 Updates durchgeführt, während rechts 256 Updates pro Frame erlaubt waren.

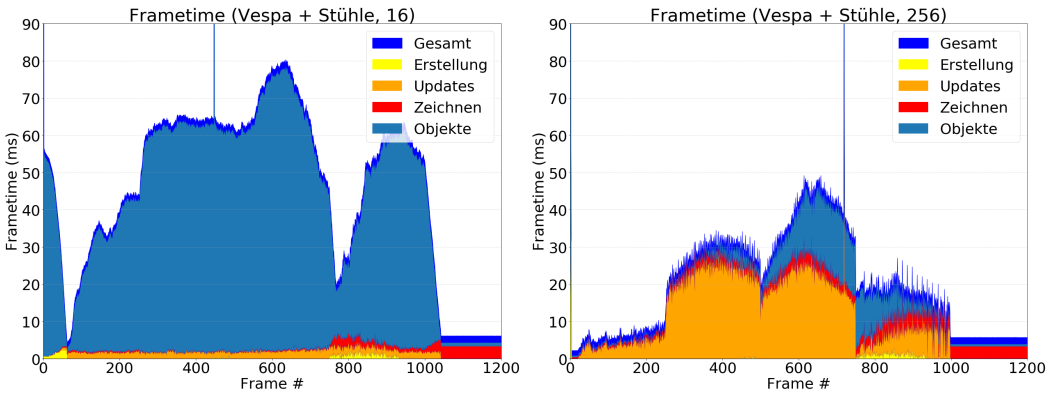

Abbildung 14: *Performance des Deferred Renderers mit maximal 16 Billboard-Updates pro Frame (links) und maximal 256 Billboard Updates (rechts). Jeweils im dunkleren Blau ist die gesamte Frametime, im helleren Blau nur die Zeit für das Zeichnen normaler Objekte, in Gelb die zur Erstellung neuer Billboards benötigte Zeit, in Orange die zum Updaten von Billboards benötigte Zeit und in Rot die Zeit, die zum Zeichnen aller aktiven Billboards benötigt wird, aufgetragen.*

Dabei fällt im Vergleich zu der Performance mit deaktivierten Billboards in bestimmten Bereichen auch mit nur 16 Updates bereits ein enormer Performancegewinn auf. In den ersten 300 Frames kann mit Billboards mit nur 16 Updates ebenfalls bereits eine deutliche Verbesserung der Performance erzielt werden, die Frametimes liegen anstatt zwischen 55 und 60 ms im Bereich von 5 - 55 ms. Dafür schwanken diese allerdings stärker.
Anders sieht es im Bereich von Frame 300 bis ca 1050 aus. In dieser Zeit kann kein nennenswerter Performancevorteil erreicht werden. Bei einem Blick auf die Zahl der aktiven Billboards zu dieser Zeit (in rot links in Abbildung 15) wird deutlich, dass in diesem Bereich die Anzahl an aktiven Billboards sehr gering ist, während die Anzahl an Billboard Updates in diesem Bereich dauerhaft auf dem Maximum von 16 steht. Die Billboards können also nicht schnell genug geupdatet werden, wodurch viele Objekte normal gerendert werden müssen. Ab etwa Frame 1050 bis zum Ende der Aufzeichnung verringert sich die Frametime jedoch um ca. den Faktor 14 von ca. 72 ms ohne Billboards auf 5 ms mit diesen.

Werden 256 Updates pro Frame erlaubt, so verbessert sich die Performance noch deutlicher. Es ergibt sich ein Performancegewinn in allen Frames der Kamerafahrt von, außer in den Frames 600 bis 750, einem Faktor größer 2.
Dies kann erklärt werden durch die fast durchgehend sehr hohe Anzahl aktiver Bill-

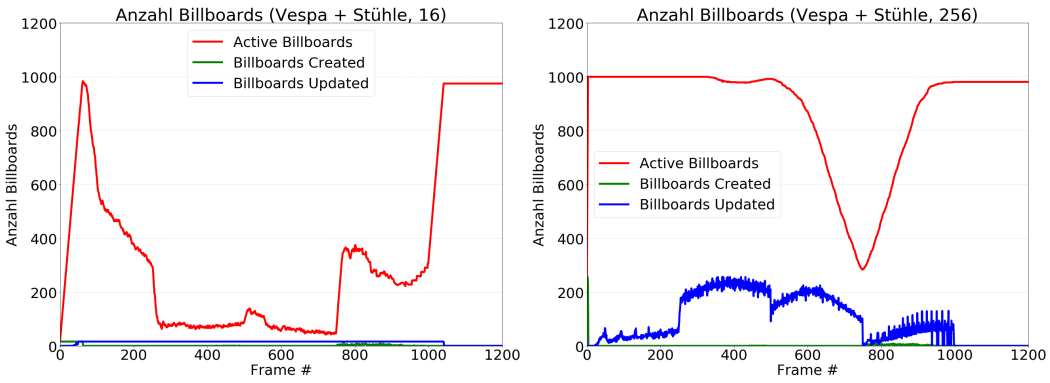

Abbildung 15: *Die Anzahl an aktiven (rot), neu erstellten (grün) und geupdateten Billboards(blau) pro Frame, einmal mit 16 Billboard Updates pro Frame (links) und einmal mit 256 Updates (rechts).*

boards. So sind beispielsweise in den Frames 50 - 350 alle 1000 Objekte der Szene durch ein Billboard ersetzt, wodurch die benötigte Frametime mit weniger als 10 ms sehr gering ist. Anders als mit 16 aktiven Billboards wird die zur Verfügung stehende Kapazität von 256 Billboard Updates pro Frame auch nicht ausgenutzt.

Die geringe Anzahl an verwendeten Billboards um Frame 700 herum kann durch den Kamerapfad erklärt werden. Zu dieser Zeit ist die Kamera sehr nah an den Objekten und viele Objekte liegen außerdem hinter der Kamera. In diesem Fall wird das Objekt nicht gezeichnet, und das zum Objekt gehörende Billboard freigegeben.

In Abbildung 16 kann außerdem noch die Verteilung der Billboards auf die unterschiedlichen Dimensionen betrachtet werden. Diese Verteilungen sehen für verschiedene Updatemaxima genauso wie für die verschiedenen Renderer mit demselben Modell immer sehr ähnlich aus, sodass hier nur die Verteilung für 256 Updates pro Frame für die „Vespa und Stühle“-Szene gezeigt wird (links). Auf der rechten Seite ist diese Verteilung deshalb stattdessen für eine Fahrt durch das zuvor verwendete Lumberyard Bistro dargestellt (ebenfalls mit 256 Updates pro Frame).

Hier lässt sich erkennen, dass besonders das Bistro-Modell sehr viele kleine Billboards verwendet, sodass teilweise über 90% der Billboards in der Dimension 16x16 liegen. Außerdem existieren für beide Modelle Bereiche, in denen die Anzahl aktiver Billboard stark sinkt oder sogar auf 0 fällt. Dies lässt sich in beiden Szenen auf eine nahe an den Objekten befindliche Kamera und eine enge Rotation dieser zurückführen, wodurch viele der Objekte der Szenen hinter der Kamera liegen und somit deren Billboards gelöscht werden.

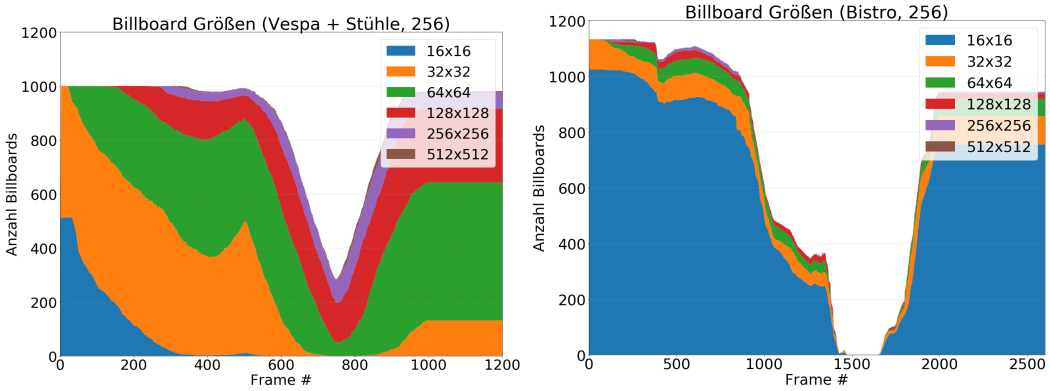

Abbildung 16: *Die Verteilung der erstellten Billboards auf die unterschiedlichen Dimensionen, einmal die „Vespa und Stühle“-Szene (rechts) und einmal für den Innenbereich des Bistros (links). Da diese für verschiedene Updateparameter fast identisch ausssehen, wird stattdessen die Verteilung der aktiven Billboards für die andere Szene, das Bistro gezeigt.*

Jedoch steigt für beide Szenen die Anzahl verwendeter Billboards anschließend wieder, sodass wieder fast alle Objekte der Szene ersetzt werden.

5 Fazit

In diesem Kapitel sollen einige Probleme des Verfahrens erläutert werden. Ebenfalls Erwähnung finden einige bei der Entwicklung aufgetretene Probleme.

Für die meisten Probleme der G-Buffer-Billboards werden außerdem noch Möglichkeiten vorgeschlagen, wie diese umgangen werden können, genauso wie auch allgemeine Vorschläge zur weiteren Verbesserung des Verfahrens gemacht werden.

5.1 Probleme und Ausblick

Zuerst werden die Probleme der G-Buffer-Billboards erläutert und anschließend die bei der Entwicklung aufgetretenen Probleme erklärt.

5.1.1 Probleme der G-Buffer-Billboards

Wie im vorherigen Kapitel gezeigt, kann das Verfahren der G-Buffer-Billboards eine deutliche Verbesserung der Performance ermöglichen, sofern die Anzahl an erlaubten Updates hoch genug angesetzt wird. Bei stillen Szenen ohne Bewegung ist dabei kein Verlust der Bildqualität zu beobachten. Sobald jedoch Bewegung in das Bild kommt, besonders bei Rotationen oder seitlicher Bewegung der Kamera, kommt es schnell zu Fehlern im Bild.

Da die Billboards erst ab einem bestimmten Schwellwert geupdatet werden und gleichzeitig immer am Koordinatensystem der Kamera ausgerichtet sind, scheinen Billboards, die noch als gut eingestuft werden, ein wenig mit der Kamera mitzudrehen oder zu rutschen. Sobald dann für dieses Objekt ein Update ausgeführt wird, kommt es zu einem Popping-Artefakt, da das Objekt auf einmal wieder an der korrekten Stelle gezeichnet wird.

Dieser Popping-Effekt kann durch eine niedrigere Wahl der Schwellwerte zum Updaten zwar weitgehend eliminiert werden, allerdings müssen die Billboards dann öfter aktualisiert werden, was sich negativ auf die Performance auswirkt.
Eine andere Lösung hierfür könnte der Einsatz von *Image-Warping*-Methoden, wie sie von McMillan [McM97] präsentiert wurden, sein. Mit diesen kann ein existierendes Bild aus einer anderen Perspektive betrachtet werden, wodurch die Lebensdauer eines Billboards stark erhöht werden könnte.

Die Popping-Artefakte bei Ausführung eines Updates könnten minimiert werden, indem *Alpha Blending* auf die beiden Versionen des Billboards angewendet wird. Dieses Verfahren wird bspw. in [Lue+03, p. 114f] erklärt. Würde das alte Billboard nach dem Update noch für einige Frames behalten, so könnte dadurch ein weicherer Übergang zwischen neuem und altem Billboard realisiert werden.

Ein anderes Problem der G-Buffer-Billboards tritt auf, wenn mehrere Objekte sehr dicht beieinander liegen oder sich überlappen. Durch Bewegung der Kamera kann es passieren, dass solche Objekte eine falsche Tiefeninteraktion zeigen. Eine Methode dies zu verhindern wäre das sehr häufige Updaten dieser Billboards. Vermutlich die zuverlässigste Methode wäre es jedoch, diese Objekte zu einem Objekt zusammenzufassen, wodurch diese nicht mehr durch mehrere Billboards ersetzt werden, sondern nur noch ein Billboard für alle der zusammengefassten Objekte erstellt

wird.

Ein weiteres, und vielleicht sogar das schwerstwiegende, Problem ist der sehr hohe Speicherbedarf. Für den Deferred Renderer werden für den Framebuffer pro Pixel aktuell 13 Byte benötigt. Da die Billboards dasselbe Layout verwenden, benötigt also auch ein Pixel eines Billboards so viele Bytes, sodass bspw. ein Billboard der Größe 512x512 nur für die generierten Texturen bereits 3,3 Megabyte an Speicherplatz einnimmt. Hier könnte es sinnvoll sein, die Informationen in den Attachments besser zu komprimieren.

Allerdings wurde die Implementation nicht im Hinblick auf den Bedarf an Speicherplatz optimiert, sodass teilweise zur Beschleunigung von Berechnungen zusätzlicher Speicher verwendet wird.

Weiterhin kann die CPU relativ schnell zum Bottleneck für die Performance werden, da die Implementation nur einen Thread verwendet.

5.1.2 Probleme bei der Entwicklung

Während der Entwicklung der Implementation traten eine Reihe von Problemen auf, die von Abstürzen des Programms über Freezes bis zu unvorhergesehenem Programmverhalten und schlechter Performance eine Reihe an unerwünschten Folgen hatten.

Die meisten dieser Probleme traten dabei mit dem zuvor bereits genannten AMD-System auf, auf welchem der Großteil der Entwicklung stattgefunden hat.

Aufgrund einer anscheinend schlechteren Treiberimplementierung[4] ist mit diesem PC trotz einer deutlich leistungsstärkeren Grafikkarte die Performance oftmals schlechter als mit dem Laptop, auf dem die Messungen durchgeführt wurden. Dies liegt vorwiegend daran, dass der OpenGL-Treiber von AMD deutlich mehr Overhead verursacht und auch kein Multi-Threading zu verwenden scheint (anders als der NVIDIA-Treiber auf dem Laptop). Dadurch wird der verwendete CPU-Kern oftmals ausgelastet und zum Bottleneck, sodass die GPU ausgebremst wird.

Außerdem existieren in dem AMD-Treiber einige Bugs, welche eine effiziente Nutzung zusätzlich erschweren. So existiert beispielweise ein Bug mit Bindless Textu-

[4] wie auch eine Vielzahl ähnlicher Reports im Bug-Report Forum von AMD nahelegt. Als Beispiel sei `https://community.amd.com/thread/206176` angeführt.

res seit Ende 2015[5], der dazu führt, dass die Applikation sich aufhängt, sobald mehr als ca. 450 Bindless Textures Resident gesetzt werden, wobei die Größe der Texturen vollkommen irrelevant ist. Für diesen Bug wurde 2016 ein Fix angekündigt, aber anscheinend nie in den Treiber integriert.

Als Workaround müssen Bindless Textures immer in Batches von nicht mehr als 450 Texturen resident gesetzt werden. Anschließend können die Zeichenoperationen mit diesen Texturen ausgeführt werden, um daraufhin die Texturen wieder unresident zu machen. Jetzt kann die nächste Batch an Texturen resident gesetzt werden, wobei zuvor ein (performanceunfreundlicher) Flush durchgeführt werden muss, um die unresident-Operation auszuführen, bevor neue Texturen resident gesetzt werden.

Dieser Flush lässt sich nicht vermeiden, da ansonsten die Anwendung früher oder später abstürzt oder sich aufhängt.

Zusätzlich wurde während der Entwicklung festgestellt, dass das Allokieren einer großen Anzahl an Framebuffern, wenn bereits viele andere Framebuffer existieren, einen enormen Overhead verursacht (die längste hierfür mit der AMD-Grafikkarte beobachtete Zeit waren ca. 20 s). Diese Operationen brauchten zwar auch auf dem anderen PC einige Zeit, jedoch bei weitem nicht so lange (meistens im Bereich von 100 bis 200 ms).

Diese beiden Probleme sind die Hauptursache, weshalb die Billboards einer Dimension in der finalen Version einen Framebuffer teilen, da hierdurch die Anzahl an Framebuffern genauso wie die Anzahl an Bindless Textures massiv gesenkt werden kann. Erst hierdurch konnte die Performance mit der AMD-Grafikkarte so weit gesenkt werden, dass eine Nutzung möglich ist.

Das Profiling zur Findung dieser Probleme hat dabei genauso wie die Entwicklung von Workarounds viel Zeit gekostet.

Außerdem existiert in den neuesten Versionen (ab Version 18.10.1) des AMD-Treibers ein Bug, der die Nutzung von Bindless Textures in einem Fragment Shader mit Discard-Anweisung unmöglich macht, da es in diesem Fall zu einem Absturz des Programms kommt[6]. Dies findet auch in dieser Implementation Anwendung, sodass auf AMD-Grafikkarten aktuell nur ältere als der genannte Treiber eingesetzt werden können, solange der für vermutlich Mitte Dezember angekündigte Fix nicht

[5] Bug-Report im AMD-Forum: `https://community.amd.com/thread/191635`
[6] Bug Report: `https://community.amd.com/thread/232901`

verfügbar ist.

5.2 Verbesserungsmöglichkeiten

Aufgrund der schlechten Unterstützung von Multi-Threading durch OpenGL wird dieses aktuell nicht eingesetzt. Dies führt dazu, dass die CPU schnell zum Bottleneck werden kann.

Moderne Grafik-APIs wie *Vulkan* oder *DirectX 12* erlauben Multi-Threading und unterstützen dafür Command-Buffer, die es erlauben aus verschiedenen Threads Kommandos an die GPU zu senden. Eine sehr informative Präsentation zu Command-Buffern in *Vulkan* ist beispielsweise unter [Wor16] zu finden. Damit könnten bspw. die Billboards asynchron erstellt und geupdatet werden, sodass der CPU-Overhead für diese Operationen aus der Renderloop herausfällt.

Auch die Berechnung der verschiedenen Scores könnte asynchron stattfinden. Dadurch könnte der Renderthread stark entlastet werden, und die Performance verbessert werden.

Eine weitere Idee könnte die Kombination mit herkömmlichen Billboards sein. Denn diese stellen für sehr weit von der Kamera entfernt liegende Objekte auch ohne Tiefeninformationen bereits eine gute Repräsentation dar. Deshalb wäre es möglich sehr weit von der Kamera entfernt liegende Objekte durch normale dynamische Billboards zu ersetzen, während mittelweit von der Kamera entfernte Objekte durch G-Buffer-Billboards ersetzt werden. Dadurch reduziert sich der Speicherbedarf etwas, da für die normalen Billboards keine Tiefentextur benötigt wird.

5.3 Schlusswort

In dieser Arbeit wurden die G-Buffer-Billboards entwickelt und die dazu nötigen Berechnungen erklärt. Hiermit ist es möglich ein Objekt durch eine G-Buffer-Textur zu ersetzen, wodurch die benötigte Frametime deutlich reduziert werden kann.

Auch wenn der Speicherbedarf des Verfahrens relativ hoch ist, ermöglicht dieses einen deutlichen Performancegewinn, weshalb die weitere Erforschung durchaus interessant ist.

Literatur

[Ake+18] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki und S. Hillaire. *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, 2018, S. 1200. ISBN: 978-1-13862-700-0.

[Bla87] E. H. Blake. „A Metric for Computing Adaptive Detail in Animated Scenes Using Object-Oriented Programming“. In: *EG 1987-Technical Papers*. Eurographics Association, 1987. DOI: 10.2312/egtp.19871023.

[CR12a] P. Cozzi und C. Riccio. *OpenGL Insights*. CRC Press, 2012. ISBN: 978-1439893760.

[CR12b] P. Cozzi und C. Riccio. „OpenGL Pipeline Map“. In: *OpenGL Insights*. Hrsg. von P. Cozzi und C. Riccio. CRC Press, 2012. ISBN: 978-1439893760. URL: https://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGL44PipelineMap.pdf.

[Déc+03] X. Décoret, F. Durand, F. X. Sillion und J. Dorsey. „Billboard Clouds for Extreme Model Simplification“. In: *ACM Trans. Graph.* 22.3 (2003), S. 689–696. URL: http://doi.acm.org/10.1145/882262.882326.

[Dec+99] X. Decoret, F. Sillion, G. Schaufler und J. Dorsey. „Multi-layered impostors for accelerated rendering“. In: *Computer Graphics Forum* 18.3 (1999), S. 61–73. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00328.

[Dee+88] M. Deering, S. Winner, B. Schediwy, C. Duffy und N. Hunt. „The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics“. In: *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '88. ACM, 1988, S. 21–30. ISBN: 0-89791-275-6. URL: http://doi.acm.org/10.1145/54852.378468.

[FS93] T. A. Funkhouser und C. H. Séquin. „Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments“. In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '93. Anaheim, CA: ACM, 1993, S. 247–254. ISBN: 0-89791-601-8. URL: http://doi.acm.org/10.1145/166117.166149.

[HG94] P. Heckbert und M. Garland. „Multiresolution Modeling for Fast Rendering“. In: *Proceedings of Graphics Interface '94*. GI '94. Banff, Alberta, Canada: Canadian Human-Computer Communications Society, 1994, S. 43–50. ISBN: 0-9695338-3-7. URL: http://graphicsinterface.org/wp-content/uploads/gi1994-6.pdf.

[JWP05] S. Jeschke, M. Wimmer und W. Purgathofer. „Image-based Representations for Accelerated Rendering of Complex Scenes“. In: *EUROGRAPHICS 2005 State of the Art Reports*. The Eurographics Association und The Image Synthesis Group, 2005, S. 1–20. URL: https://www.cg.tuwien.ac.at/research/publications/2005/jeschke-05-ISTAR/.

[Lue+03] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson und R. Huebner. *Level of Detail for 3D Graphics*. Elsevier Science, 2003. ISBN: 978-1-5586-0838-2. URL: https://books.google.de/books?id=M-gl4aoxQfIC.

[Lum17] A. Lumberyard. *Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)*. 2017. URL: http://developer.nvidia.com/orca/amazon-lumberyard-bistro.

[McG+12] M. McGuire, P. Hennessy, M. Bukowski und B. Osman. „A Reconstruction Filter for Plausible Motion Blur“. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. Costa Mesa, California: ACM, 2012, S. 135–142. ISBN: 978-1-4503-1194-6. URL: http://doi.acm.org/10.1145/2159616.2159639.

[McM97] L. McMillan Jr. „An Image-based Approach to Three-dimensional Computer Graphics“. Diss. 1997.

[Nar14] K. Narkowicz. *Octahedron normal vector encoding*. 2014. URL: https://knarkowicz.wordpress.com/2014/04/16/octahedron-normal-vector-encoding/.

[NL13] M. Nießner und C. Loop. „Analytic Displacement Mapping Using Hardware Tessellation“. In: *ACM Trans. Graph.* 32.3 (2013), 26:1–26:9. URL: http://doi.acm.org/10.1145/2487228.2487234.

[ST90] T. Saito und T. Takahashi. „Comprehensible Rendering of 3-D Shapes“. In: *SIGGRAPH Comput. Graph.* 24.4 (1990), S. 197–206. URL: http://doi.acm.org/10.1145/97880.97901.

[SS95] G. Schaufler und W. Stürzlinger. „Generating Multiple Levels of Detail from Polygonal Geometry Models“. In: *Virtual Environments '95*. Springer Vienna, 1995, S. 33–41. ISBN: 978-3-7091-9433-1.

[Sch95] G. Schaufler. „Dynamically generated Impostors“. In: *GI Workshop on Modeling, Virtual Worlds*. 1995, S. 129–135.

[SS96] G. Schaufler und W. Stürzlinger. „A Three Dimensional Image Cache for Virtual Reality“. In: *Computer Graphics Forum* 15.3 (1996), S. 227–235. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1530227.

[SDB97] F. Sillion, G. Drettakis und B. Bodelet. „Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery“. In: *Computer Graphics Forum* 16.3 (1997), S. C207–C218. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00158.

[Wor16] M. Worcester. *Command Buffers and Pipelines*. 2016. URL: https://www.khronos.org/assets/uploads/developers/library/2016-vulkan-devday-uk/2-Command_buffers_and_pipelines.pdf.

Anhang

A Anhang zum Amazon Lumberyard Bistro

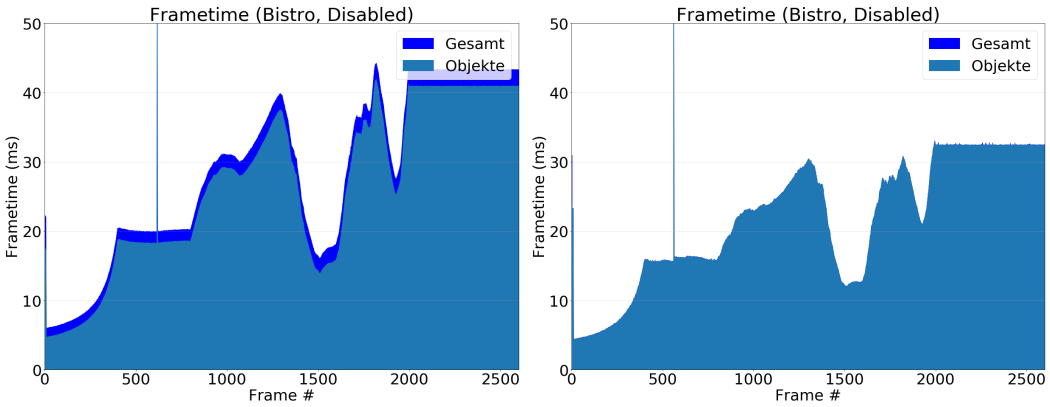

Abbildung A.1: *Performance mit dem Deferred Renderer (links) und Forward Renderer (rechts) durch das Amazon Lumberyard Bistro mit deaktivierten Billboards.*

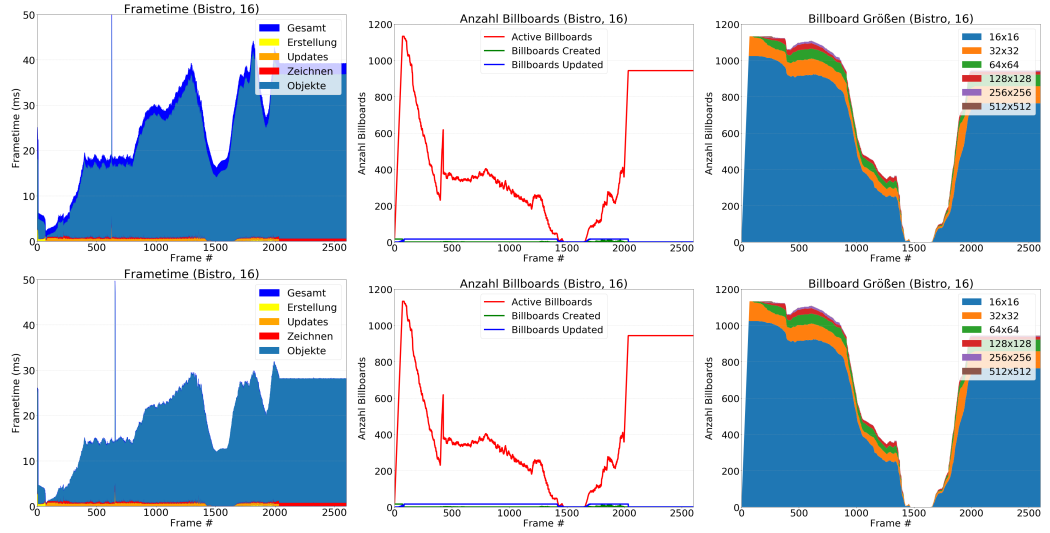

Abbildung A.2: *Performance sowie Billboard Nutzung mit dem Deferred Renderer (obere Zeile) und dem Forward Renderer (untere Zeile) durch das Amazon Lumberyard Bistro mit 16 Billboard Updates.*

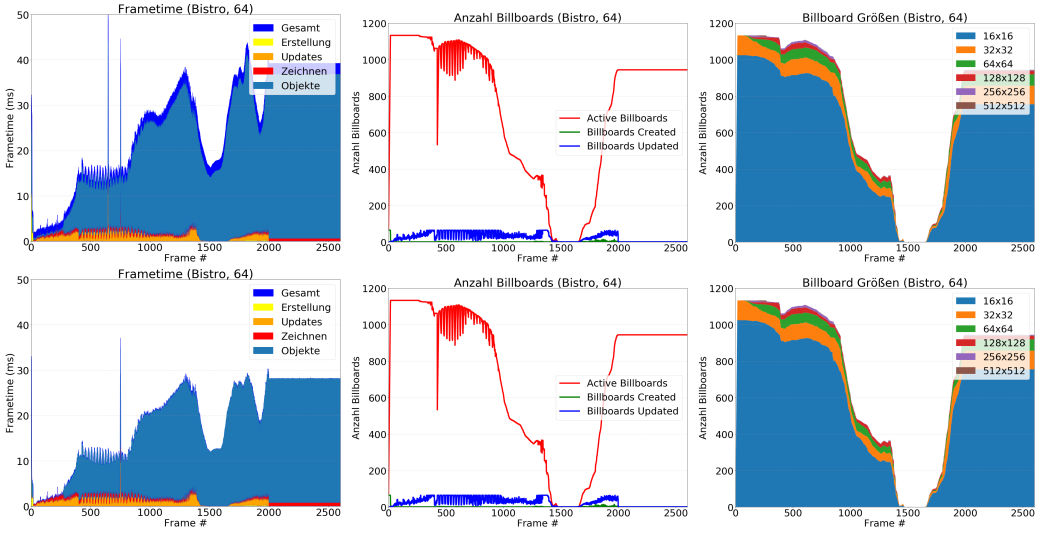

Abbildung A.3: *Performance sowie Billboard Nutzung mit dem Deferred Renderer (obere Zeile) und dem Forward Renderer (untere Zeile) durch das Amazon Lumberyard Bistro mit 64 Billboard Updates.*

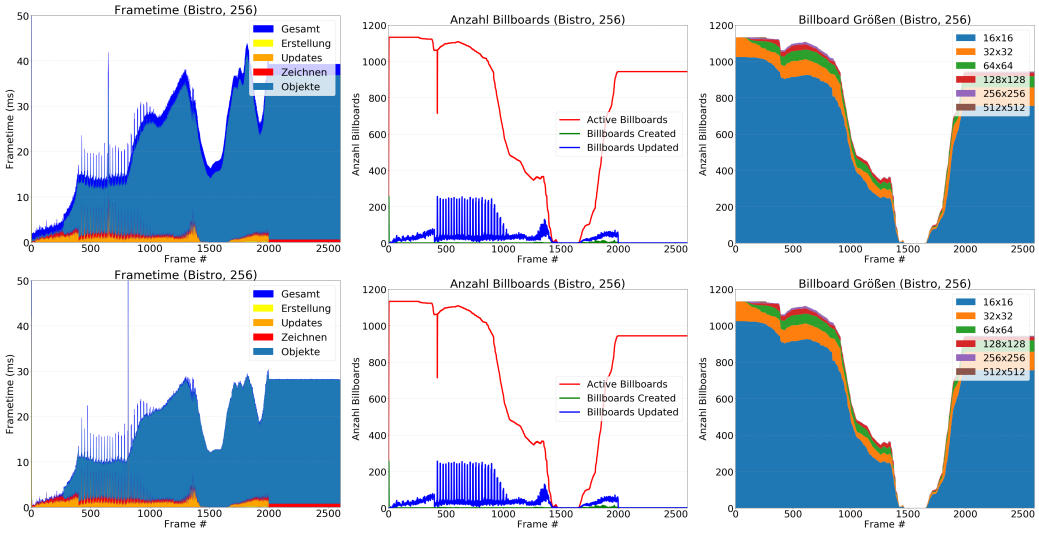

Abbildung A.4: *Performance sowie Billboard Nutzung mit dem Deferred Renderer (obere Zeile) und dem Forward Renderer (untere Zeile) durch das Amazon Lumberyard Bistro mit 256 Billboard Updates.*

B Anhang zu Vespa und Stühle

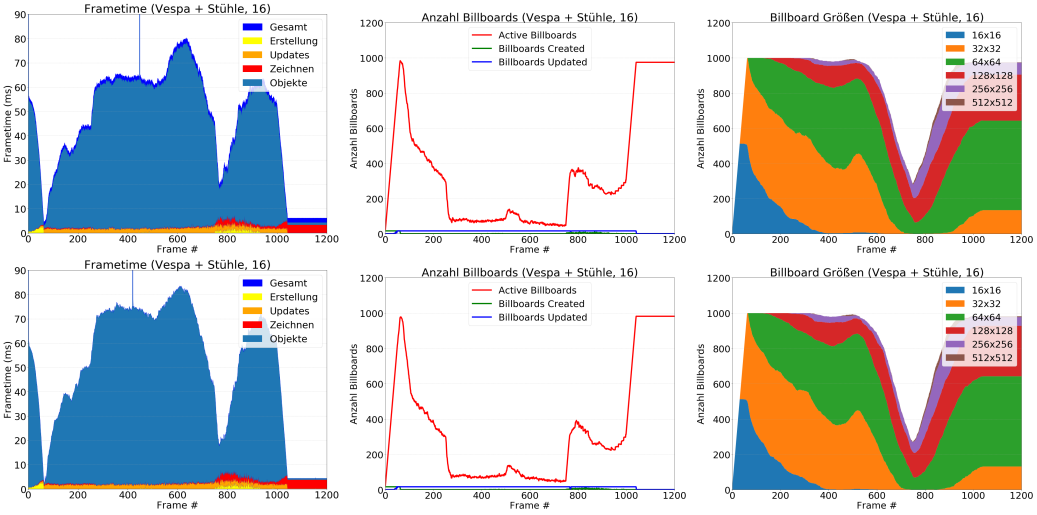

Abbildung B.1: *Performance sowie Billboard Nutzung mit dem Deferred Renderer (obere Zeile) und dem Forward Renderer (untere Zeile) durch die „Vespa und Stühle“-Szene mit 16 Billboard Updates.*

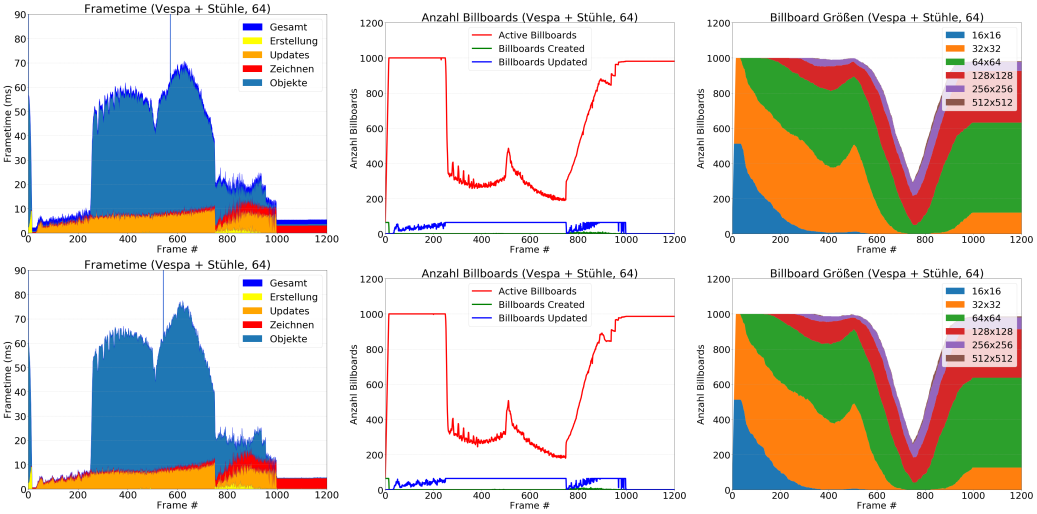

Abbildung B.2: *Performance sowie Billboard Nutzung mit dem Deferred Renderer (obere Zeile) und dem Forward Renderer (untere Zeile) durch die „Vespa und Stühle“-Szene mit 64 Billboard Updates.*

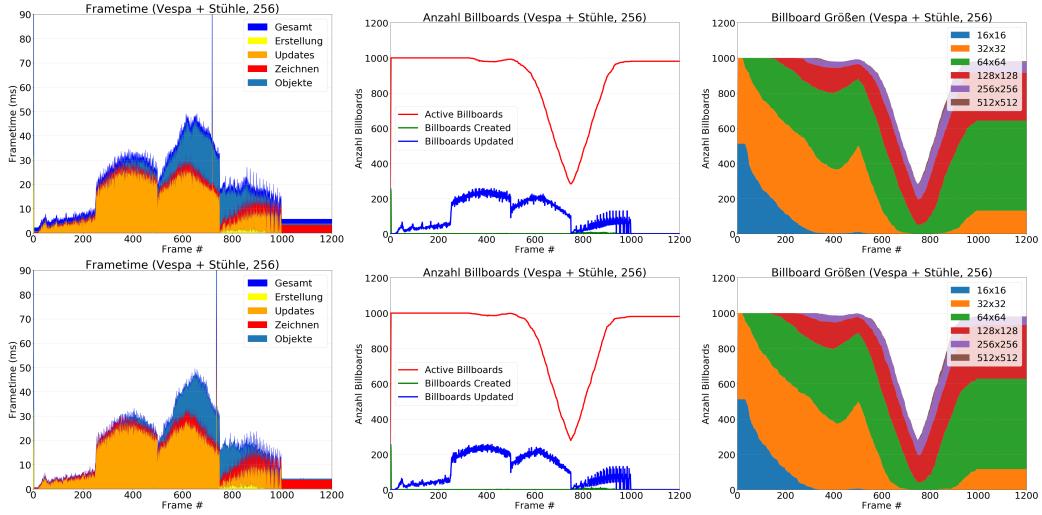

Abbildung B.3: *Performance sowie Billboard Nutzung mit dem Deferred Renderer (obere Zeile) und dem Forward Renderer (untere Zeile) durch die „Vespa und Stühle“-Szene mit 256 Billboard Updates.*

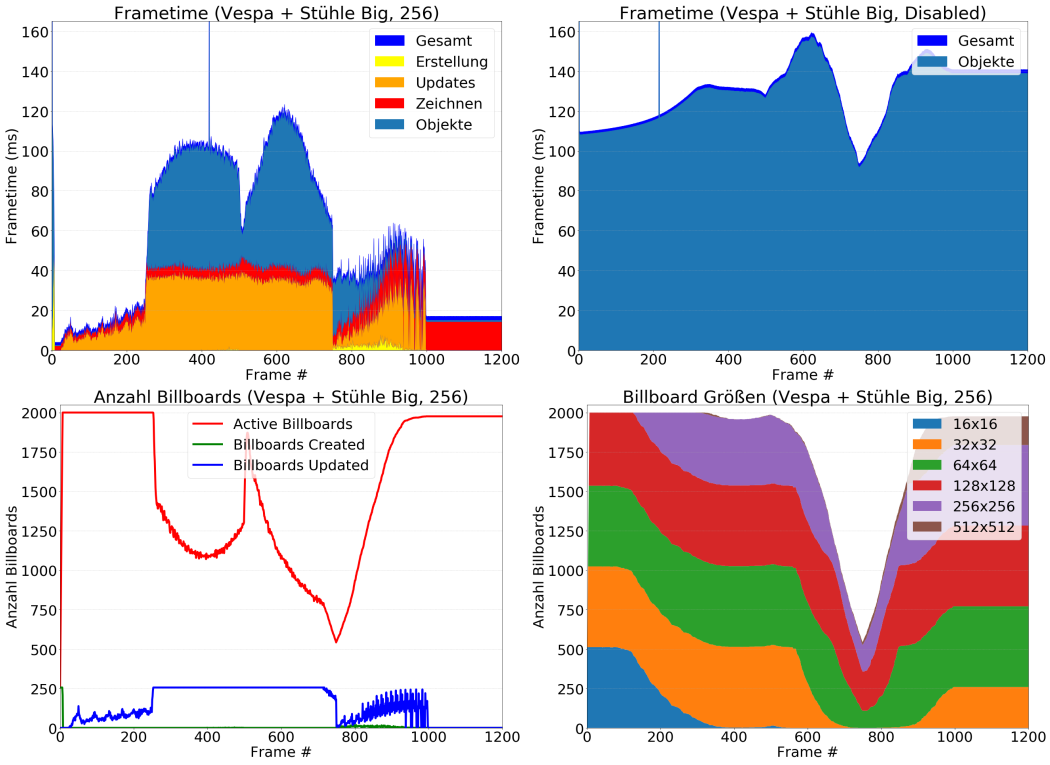

Abbildung B.4: *Performance sowie Billboard Nutzung mit dem Deferred Renderer durch die große „Vespa und Stühle“-Szene mit je 1000 Instanzen mit 256 Billboard Updates (links) sowie ohne Billboards (rechts).*

C Ehrenwörtliche Erklärung

Anhang 1: Ehrenwörtliche Erklärung

E R K L Ä R U N G

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass alle Stellen dieser Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht wurden und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsstelle vorgelegt wurde.

Desweiteren erkläre ich, dass ich mit der öffentlichen Bereitstellung meiner Abschlussarbeit in der Instituts- und/oder Universitätsbibliothek nicht einverstanden bin.
Ich erlaube jedoch ausdrücklich die öffentliche Bereitstellung meiner Abschlussarbeit in einer digitalen Form.

Ort, Datum, Unterschrift