Universität Osnabrück

Fachbereich Mathematik/Informatik

Bachelorarbeit

Dynamische Echtzeit 3D-Darstellung eines Regensystems unter Verwendung paralleler Algorithmen

27. August 2013

Vorgelegt von:

Valentin Bruder
Sigwartstr. 3
72149 Neustetten
vbruder@uni-osnabrueck.de

Gutachter:

Prof. Dr. Oliver Vornberger
Prof. Dr.-Ing. Elke Pulvermüller

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich während meines Studiums und dem Anfertigen dieser Bachelorarbeit unterstützt haben.

Besonders gilt dieser Dank Sascha Kolodzey, der mich während meiner Arbeit betreut und unterstützt hat. Seine Anregungen und Unterstützung bei Problemen waren ungemein hilfreich und haben maßgeblich zum Erreichen meiner Ziele beigetragen. Vielen Dank für die Geduld und die Mühe.

Des Weiteren möchte ich mich bei Frau Prof. Dr.-Ing. Elke Pulvermüller und Herrn Prof. Dr. Oliver Vornberger für die Begutachtung dieser Arbeit bedanken.

Meiner Mutter möchte ich insbesondere für die sprachliche Korrektur der Arbeit danken. Nicht zuletzt gebührt meiner ganzen Familie Dank, die mich während meines Studiums immer unterstützt hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Algorithmenverzeichnis

Dynamische Echtzeit 3D-Darstellung eines Regensystems unter Verwendung paralleler Algorithmen

Valentin Bruder

27. August 2013

Die Darstellung von Wetterphänomenen ist in Echtzeit 3D-Anwendungen ein wichtiges Stilmittel. In dieser Bachelorarbeit wurde ein Framework entwickelt, das die Darstellung und Simulation eines Regensystems umfasst. Dabei werden sowohl die fallenden Regentropfen visualisiert, als auch eine realitätsnahe Simulation von regenbedingten Wasseransammlungen. Einer ansprechenden Darstellung der Wasseroberflächen wurde ebenfalls Aufmerksamkeit geschenkt. Des Weiteren besteht die Möglichkeit für den Anwender, Umgebungseigenschaften wie beispielsweise Regen- und Windstärke zu manipulieren und somit Auswirkungen auf das System sichtbar zu machen. Der Fokus liegt bei dieser Arbeit auf der Echtzeitdarstellung. Um dies zu gewährleisten, werden unter anderem parallele Algorithmen auf der Grafikkarte mittels OpenCL eingesetzt. Die Vorgehensweise, die physikalischen Hintergründe, sowie die verwendeten Techniken, Schnittstellen und Algorithmen werden in dieser Arbeit beschrieben.

1 Einführung

1.1 Motivation

3D-Echtzeitanwendungen werden in der heutigen Zeit, dank leistungsfähiger Hardware, vielfach in verschiedenem Kontext eingesetzt. Das größte Anwendungsgebiet ist dabei nach wie vor das der Videospiele. Aber auch andere Anwendungsfelder gewinnen zunehmend an Beliebtheit. Als Beispiele kann man Simulations- und Konstruktionssoftware aufführen, die unter anderem in technischen Bereichen der Industrie eingesetzt werden oder zu Ausbildungs- und Veranschaulichungszwecken. Die Leistungsfähigkeit moderner Grafikhardware ist durch fortlaufende Weiter- und Neuentwicklungen in den letzten Jahren stark gestiegen. Bei der Entwicklung von 3D-Engines ist deshalb eine Tendenz weg von einer Fokussierung auf pure Geschwindigkeit, hin zu steigendem Realismus in der Darstellung zu beobachten. Eine möglichst realistische Wiedergabe der virtuellen Welt ist also oftmals – heute mehr denn je – wünschenswert. Dazu gehören neben einer flüssigen Bewegungsdarstellung und hochauflösenden Geometrien auch eine glaubwürdige Umweltdarstellung und -repräsentation.

Insbesondere Wetterphänomene können in Freiluftszenen zu diesem Realismus entscheidend beitragen. Regen ist in gemäßigten Klimazonen die am häufigsten vorkommende Niederschlagsform. Der Darstellung dieses wichtigen Phänomens mangelt es jedoch in vielen 3D-Engines, die diese überhaupt bieten, an Realismus. Bedeutend ist ein Einsatz von Regen in 3D-Szenen auch deshalb, weil dieser gezielt als Stilmittel verwendet werden kann. Was bei Filmen bereits seit Jahrzehnten praktiziert wird, kann auch bei Videospielen eingesetzt werden, die häufig als interaktiver Film präsentiert werden. Eine stark verregnete Szene kann beispielsweise eine düstere oder bedrohliche Stimmung erzeugen.

Betrachtet man moderne 3D-Engines wie beispielsweise die Frostbite™ Engine, REDengine oder die UNIGENE™ Engine, so fällt auf, dass zwar oft eine Regendarstellung vorhanden ist, der Regen aber wenig oder gar nicht mit der, den Betrachter umgebenden Landschaft, interagiert (vgl. Abbildung 1). In der Natur findet aber eine solche Interaktion statt, etwa in Form von Pfützenbildung oder kleinen Bächen. Während eine Umgebungsinteraktion in genannten Engines höchstens in einzelnen Szenen durch Scripte realisiert wird, ist zu beobachten, dass insbesondere in den *Open-World-Engines* die Regentropfen nach dem Auftreffen auf den Boden einfach verschwinden. Ein Framework, welches die Darstellung von Regen, sowie dessen Interaktion mit der Umgebung umfasst und dabei auch die entstehenden Wasseroberflächen realitätsnah visualisiert, ist deshalb eine interessante Aufgabe. Eine solche Anwendung und die verwendeten Techniken werden in dieser Arbeit vorgestellt.

Abbildung 1: Screenshot Unigene Valley Benchmark mit Regeneffekt (UNIGENE™)

1.2 Ziele für das Framework

Neben den beiden darstellenden Teilen für den Regen sowie für die Wasseroberflächen, wird die durch den Niederschlag verursachte Wasserbildung und deren Fließverhalten simuliert. Die Umsetzung soll dabei in jedem Fall dem Echtzeitanspruch einer 3D-Anwendung genügen. Das heißt, das Framework soll auf dem Referenzsystem, welches in Abschnitt 4.6 genauer beschrieben wird, zu jeder Zeit mit mindestens 30 Bildern pro Sekunde (FPS) laufen. Um der Echtzeitanforderung gerecht zu werden, wird auf Techniken der Rastergrafik zurückgegriffen. Außerdem wird bei den Simulationsberechnungen weniger Wert auf physikalisch korrekte Berechnungen gelegt, sondern versucht, mit verschiedenen Techniken ein möglichst realistisches Verhalten bei vergleichsweise geringer Rechenlast zu erzeugen. Zusätzlich werden möglichst viele Berechnungen parallel auf Grafikhardware ausgeführt beziehungsweise ausgelagert, um den Hauptprozessor (CPU) zu entlasten. Um dies umzusetzen, wird bei der Implementation auf die Grafikbibliothek OpenGL und die OpenCL API für parallele Programmierung auf heterogenen Systemen zurückgegriffen.

1.3 Überblick

Zu Beginn der Arbeit wird ein Überblick über die, für die Umsetzung des Frameworks, verwendeten Techniken und Schnittstellen gegeben. Hierbei wird insbesondere die verwendete Grafikschnittstelle OpenGL und deren *Rendering Pipeline* näher betrachtet. Im zweiten Abschnitt der Arbeit werden die, für die Darstellung entscheidenden physikalischen Eigenschaften von Regen und Wasser in der Natur beschrieben. Im darauffolgenden Kapitel wird dann näher auf die Implementation des Frameworks eingegangen. Dabei wird zunächst der aktuelle Stand der Technik dargestellt. Der erste Teil widmet sich der Umsetzung der Regendarstellung. Danach wird näher auf die Algorithmen für die Wasseransammlung und das Fließverhalten eingegangen. Als letzter Teil der Implementation wird die Darstellung der Wasseroberflächen näher beschrieben. Nach einer kurzen Erläuterung der restlichen Umweltdarstellung inklusive Nebel, Himmel und Terrain, wird am Ende des Abschnitts eine kurze Performance-Bewertung des Frameworks durchgeführt. Abschließend folgen das Fazit und ein Ausblick. Im Anhang wird ein Glossar bereitgestellt, in dem die verwendeten Fachbegriffe der Grafik- und Parallelprogrammierung kurz erläutert oder definiert werden. Auch verwendete Abkürzungen werden hier aufgeführt.

1.4 Koordinatensystem

Wird in dieser Arbeit von einem Koordinatensystem gesprochen, so handelt es sich – sofern nicht anders angegeben – immer um ein rechtshändiges, kartesisches Koordinatensystem. Die Achsen werden dabei mit x, y und z bezeichnet (siehe Abbildung 2).

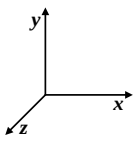

Abbildung 2: Rechtshändiges, kartesisches Koordinatensystem

2 Werkzeuge

2.1 OpenGL

OpenGL (**Open G**raphics **L**ibrary) ist eine sprach- und plattformübergreifende Programmierschnittstelle (API) für die 2D- oder 3D-Darstellung von Objekten in der Computergrafik. Insbesondere dient die API zur Interaktion mit Grafikhardware, wie beispielsweise einer Graphics Processing Unit (GPU). Ursprünglich wurde OpenGL von *Silicon Graphics Inc.* entwickelt und im Jahre 1992 auf den Markt gebracht. Heute wird die Bibliothek von der *Khronos Group* weiterentwickelt und betreut, ein Konsortium zu welchem Branchengrößen wie AMD, NVIDIA, Apple, Google und einige mehr gehören. Aktuell liegt OpenGL in der Version 4.4 vor, diese wurde am 22. Juli 2013 veröffentlicht. [11] [9]

2.1.1 Geometrien

In der 3D-Rastergrafik spielen Geometrien, auch *3D-Models* genannt, eine entscheidende Rolle. Denn aus diesen besteht die Szene, welche dann auf den Bildschirm des Anwenders projiziert wird. Das kleinste Element dieser Geometrien ist dabei immer ein *Vertex* (Knoten). Diese Vertices besitzen mindestens eine Position im Raum, jedoch können sie auch andere Informationen wie beispielsweise eine Farbe enthalten. Folgen dieser Vertices können von OpenGL als sogenannte *Primitives* interpretiert werden. Auf drei, für diese Arbeit relevanten *Primitive* Typen, soll im Folgenden kurz eingegangen werden. [11]

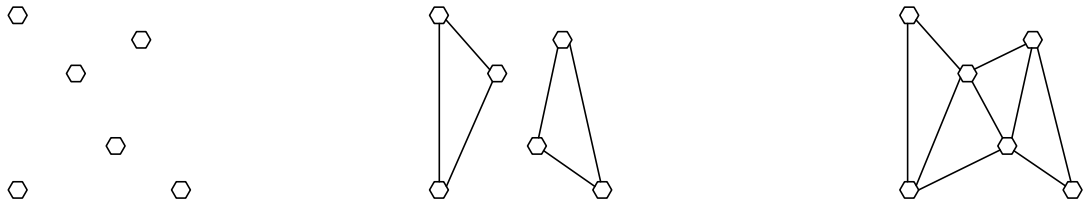

Abbildung 3: OpenGL Primitives: GL_POINTS (links), GL_TRIANGLES (Mitte) und GL_TRIANGLE_STRIP (rechts)

- GL_POINTS
  Vertices können als einzelne Punkte dargestellt werden. Dies geschieht an der jeweiligen Position des Vertex. Eine Verbindung zwischen Vertices findet bei dieser Form nicht statt (vgl. Abbildung 3 links).
- GL_TRIANGLES
  OpenGL gibt aber die Möglichkeit, die Vertices zu verbinden. Hier bietet sich für 3D-Models das Dreieck an, welches das einfachste zweidimensionale Polygon ist. Drei Vertices werden zu einem Dreieck zusammengefasst. Die Dreiecke sind bei diesem Primitive Type unabhängig (siehe Abbildung 3 mitte).
- GL_TRIANGLE_STRIP
  Möchte man die Dreiecke verbinden, um beispielsweise ein zusammenhängendes 3D-

Model zu erhalten, so bietet sich der Primitive Type *Tringle Stip* an. Dieser verbindet die Vertex-Folge fortlaufend zu Dreiecken (vgl. Abbildung 3 rechts). Bei entsprechender Anordnung der Vertex-Folge ergibt sich dann eine geschlossene Geometrie in Form eines *Wire Frame Models*.

2.1.2 Texturen

Um den Detailgrad von Geometrien zu erhöhen, werden in der Computergrafik *Texturen* verwendet. Eine Textur ist ein Bild, das über ein *Wire Frame Model* gelegt wird. In der Regel ist die Auflösung der Textur höher als der Detailgrad der Geometrie. Das heißt, die Anzahl der Pixel des der Textur zugrunde liegenden Bildes, ist höher als die der *Vertices* des Models. Somit wird ein höherer Detailgrad des Objektes suggeriert. Dieses Verfahren wird *Texture Mapping* genannt. Mit Hilfe von Texturkoordinaten zur Ausrichtung und Skalierung werden dabei die Texturen auf die Geometrie gelegt. Mit Hilfe verschiedener Modi kann die Textur dabei auch wiederholt oder gestreckt werden.

2.1.3 Rendering Pipeline

Beim rendern von Objekten geht OpenGL in einer festgelegten, sequenziellen Abfolge vor. Diese wird OpenGL *Rendering Pipeline* genannt. Ein Überblick über diese ist in Abbildung 4 dargestellt.

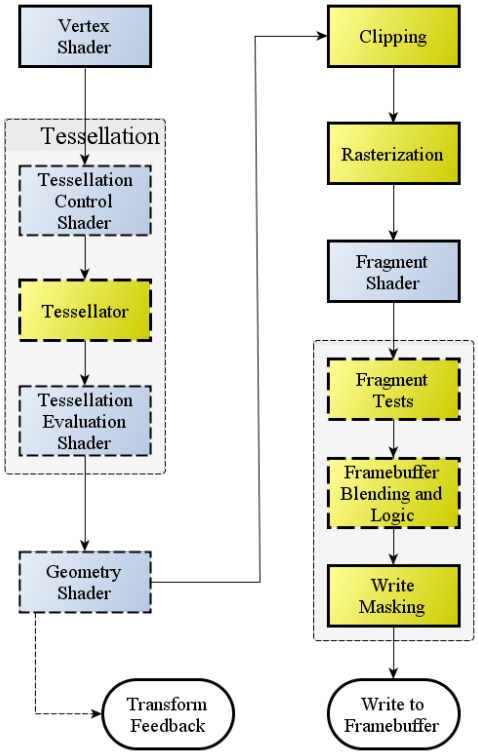

Abbildung 4: OpenGL Rendering Pipeline (Überblick), blaue Boxen repräsentieren programmierbare Shader Stages [12]

Die Pipeline lässt sich grob in zwei Kategorien unterteilen: die festen und die programmierbaren Stufen. Während für die festen Stufen höchstens Flags gesetzt werden können, um die Verarbeitung zu beeinflussen, müssen oder können für die programmierbaren Stufen sogenannte *Shader-Programme* geschrieben werden. Hier gibt es obligatorische (in Abbildung 4 mit geschlossenem Rahmen) und optionale (gestrichelter Rahmen). Shader-Programme werden in einer speziellen Programmiersprache geschrieben. In diesem Framework wurde *GLSL* verwendet, die Open**GL** **S**hading **L**anguage. Die Shader werden dann auf der Grafikhardware in paralleler Form ausgeführt und ermöglichen so eine effiziente Berechnung der für die Rastergrafik benötigten Geometrien und Farben. Die verschiedenen, im Framework verwendeten Shader-Varianten werden in Abschnitt 2.1.5 näher beschrieben. [12]

2.1.4 Koordinatensysteme

In der OpenGL *Rendering Pipeline* werden mehrere Koordinatensysteme verwendet. Zu Beginn der Pipeline liegt eine Geometrie in der Regel in einem eigenen Koordinatensystem, den *Model Coordinates* vor. Mit Hilfe einer entsprechenden Transformationsmatrix wird die Geometrie dann in der Regel im *Vertex Shader* (vgl. Abschnitt 2.1.5) in die sogenannten *World Coordinates* transformiert. Das heißt insbesondere, dass die Geometrien den Ansprüchen des Programmierers nach rotiert, skaliert und translatiert werden.

Um relativ zur Kameraposition angezeigt zu werden, müssen die Objekte noch mit Hilfe der *view*-Matrix transformiert werden. Auch das geschieht in der Regel im Vertex Shader. Die Vertices befinden sich dann in den *View Coordinates*. Von da aus sollten diese dann perspektivisch projiziert werden, um das gewünschte Ausgabebild auf einem zweidimensionalen Ausgabegerät zu erhalten. Dies passiert mit einer Projektionsmatrix. Die Vertices befinden sich nach dieser Transformation in den *Clipping Coordinates*. Nach dem *Clipping* werden sie dann abschließend normalisiert und mittels *Perspective Division* in die *Normalized Device Coordinates* überführt. Die Koordinatensysteme mit Transformationen sind in Abbildung 5 schematisch aufgeführt. [31]

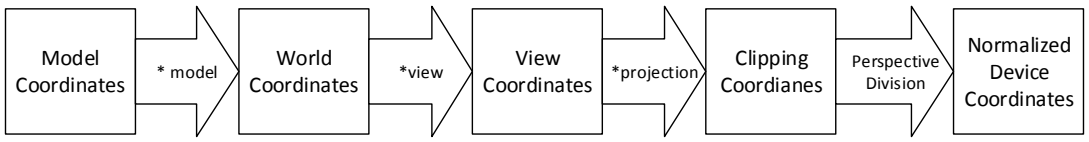

Abbildung 5: Koordinatensysteme der OpenGL Rendering Pipeline

2.1.5 Shader

Shader sind programmierbare Teile der OpenGL Rendering Pipeline (vgl. Abschnitt 2.1.3). In Shader-Programmen wird beschrieben, wie etwas auf eine spezifische Art gezeichnet werden soll, wie zum Beispiel Farbe, Beleuchtung oder Special Effects. Aktuelle Standards unterstützen Vertex, Tesselation, Geometry und Fragment Shader. In meinem

Framework werden alle außer dem Tesselation Shader eingesetzt. Auf die drei verschiedenen, verwendeten Typen wird im Folgenden näher eingegangen.

Vertex Shader Der Vertex Shader ist für die Vertex-Verarbeitung zuständig und muss immer implementiert werden. Alle Vertices der Anwendung werden genau einmal pro Pipeline-Durchlauf unabhängig voneinander und parallel vom Vertex Shader bearbeitet. Pro bearbeitetem Vertex muss auch genau ein Vertex und ein zugehöriger Positionswert ausgegeben werden. Der Vertex Shader wird in der Regel für eine Positionsmanipulation der Geometrien genutzt. Aber auch andere Operationen wie etwa die Berechnung von Texturkoordinaten oder Normalen ist üblich. Ein Beispiel eines Vertex-Shader-Programms ist in Listing 1 aufgeführt. [12]

Als besonders hervorzuheben sind in diesem Shader unter anderem die zusätzlichen Klassifikatoren zu Beginn der Variablendeklarationen in den Zeilen 3 - 10. Mit dem Schlüsselwort in werden Variablen definiert, die die zugehörigen Daten vom Host-Programm pro Vertex erhalten. Mit uniform werden die Variablen bezeichnet, deren Daten für jedes Vertex gleich sind. Diese werden ebenfalls im Host-Programm festgelegt. In diesem Beispiel sind das die Transformationsmatrizen für die Transformation von den Model-Koordinaten in die Weltkoordinaten, sowie die Projektionsmatrix für die Projektion auf ein zweidimensionales Bild, welches dann auf dem Bildschirm angezeigt werden kann. Da diese für jedes Vertex gleich sind, werden sie als uniform übergeben. Als drittes Schlüsselwort vor der Variablendeklaration findet sich noch out. Dies klassifiziert die Variable als einen Ausgabewert. Dieser wird pro Vertex ausgegeben und kann im weiteren Verlauf der Rendering Pipeline verwendet werden.

```
1  #version 330 core
2
3  in vec3 positionMC;
4  in vec2 texCoords;
5
6  out vec2 fragmentTexCoords;
7
8  uniform mat4 model;
9  uniform mat4 proj;
10 uniform mat4 view;
11
12 //simple vertex shader with view projection transformation
13 void main(void)
14 {
15     vec4 positionWC   = model * vec4(positionMC, 1.0f);
16     fragmentTexCoords = texCoords;
17     gl_Position       = proj * view * positionWC;
18 }
```

Listing 1: Einfacher Vertex Shader zur Manipulation einer Himmelgeometrie

Geometry Shader Da der *Tesselation Shader* im Framework nicht zum Einsatz kommt, ist der *Geometry Shader* die nächste programmierbare Stufe der OpenGL Rendering Pipeline, die verwendet wird. Im Allgemeinen ist der Geometry Shader optional. Mit Hilfe eines Geometry Shader Programms können Primitives, die zuvor nach der Vertex Shader Stufe

erstellt wurden weiter verarbeitet werden. Konkret heißt das, dass aus Primitives weitere Primitives erzeugt (*Tesselation*) oder bestehende vernichtet werden können. Pro Vertex in der GL_POINT Primitive Interpretation (siehe 2.1.1) können so beispielsweise drei Vertices erzeugt werden und die Topologie in GL_TRIAGLES geändert werden. Der Geometry Shader wird pro Primitive ausgeführt. Ein konkretes Beispiel für die Verwendung und Implementierung eines Geometry Shaders wird in Abschnitt 4.1.3, Listing 3 gegeben. Als nächste, feste Stufe der Rendering Pipeline folgen dann *Clipping* und emphCulling. Beim Clipping-Schritt werden Primitives außerhalb des sichtbaren Volumens abgeschnitten. Es können auch andere Clipping-Ebenen vom Programmierer definiert werden. Beim Culling werden Rückseiten von Dreiecken je nach Einstellung nicht angezeigt. Im letzten Schritt vor dem Fragment Shader werden die Primitives im *Rasterization*-Schritt gerastert. Durch die Rasterung, die in der gegebenen Reihenfolge der Primitives erfolgt, entstehen dann die sogenannten *Fragments*. [12]

Fragment Shader Die letzte programmierbare Stufe in der Rendering Pipeline ist der *Fragment Shader*. Dieser muss programmiert werden. Hier können die Fragments, welche im Rasterization-Schritt erstellt werden, manipuliert werden. Dies geschieht für jedes Fragment parallel und unabhängig. Die Ausgabe kann unterschiedlich sein. In der Regel enthält sie jedoch mindestens einen Farb- und Tiefenwert, oftmals auch einen Alphawert für das *Blending*. Beim Blending, welches eine feste Stufe der Rendering Pipeline ist, die nach dem Fragment Shader durchgeführt wird, können anhand von unterschiedlichen Kriterien Fragments teiltransparent übereinander gelegt werden. Beispielsweise kann ein Alphawert im Fragment Shader ermittelt werden und anhand diesem dann geblendet werden. In Listing 2 ist ein einfaches Beispiel eines Fragment Shaders aufgeführt. In diesem wird in Zeile 12 die Farbe für das jeweilige Fragment aus einer *Textur* (vgl. Abschnitt 2.1.2) ausgelesen. Für einen Nebeleffekt wird die finale Farbe in Zeile 13 mit einem Grauwert anteilig, je nach eingestelltem Nebelwert, gemischt.

```
1  #version 330 core
2
3  in vec2 fragmentTexCoords;
4
5  out vec4 fragColor;
6
7  uniform sampler2D textureImage;
8  uniform vec3 fogThickness;
9
10 void main(void)
11 {
12      vec4 skyColor = texture(textureImage, fragmentTexCoords);
13      fragColor = mix(skyColor, vec4(0.7), 11.0f * fogThickness.x);
14 }
```

Listing 2: Einfacher Fragment Shader zur Manipulation der Himmelfarben

2.2 Light Weight Java Game Library

Zur Umsetzung einer plattformübergreifenden Funktionalität, wird bei dem Framework die Programmiersprache Java verwendet. Um einfach auf die Bibliotheken OpenGL, OpenCL und OpenAL zugreifen zu können, wird die *Light Weight Java Game Library* (LWJGL) verwendet. Diese stellt im Prinzip einen Wrapper für die genannten Bibliotheken dar. Die Entwickler von LWJGL geben an, verschiedene Designkriterien bei der Programmierung zu verfolgen. Hierzu gehören unter anderem Geschwindigkeit, Einfachheit, Schlankheit und Minimalismus. [14]

Abbildung 6: LWJGL Logo [14]

Durch Konzentration auf die genannten Richtlinien bei der Entwicklung ist meiner Ansicht nach eine leicht zu handhabende, vergleichsweise schnelle und leichtgewichtige Bibliothek entstanden, die sich gut für die Umsetzung des Frameworks eignet. Aktuell (August 2013) liegt LWJGL in der Version 2.9.0 vor.

2.3 OpenAL

Für eine möglichst realistische 3D-Anwendung ist eine passende akustische Untermalung durchaus wünschenswert. OpenAL (**Open A**udio **L**ibrary) ist eine offene Programmierschnittstelle, die zur Erzeugung von Raumklang genutzt werden kann. Entwickelt wurde die plattformübergreifende API von *Creative Technology*. OpenAL soll als Ergänzung zu der Grafikbibliothek OpenGL (vgl. Abschnitt 2.1) gesehen werden [3]. Um einfach Sounddateien zu laden, wird in dem Framework außerdem die *slick-util* Bibliothek verwendet. Diese stellt einige Klassen zum einfachen Laden von Bild- und Sounddateien bereit, und ist für die Zusammenarbeit mit LWJGL ausgelegt. [23]

2.4 OpenCL

OpenCL (**Open C**ompute **L**anguage) ist eine, ursprünglich von *Apple Inc.* entwickelte Bibliothek für parallele Programmierung. Sie ist ein offener Standard und sowohl von Plattform, Betriebssystem als auch Programmiersprachen unabhängig. Diese Eigenschaften machen die API auch zur ersten Wahl für das Framework. Heute wird die Bibliothek, wie auch OpenGL (vgl. Abschnitt 2.1), von der *Khronos Group* betreut. Insbesondere ist OpenCL ein Framework zur Programmierung von Anwendungen, die auf heterogenen Systemen parallel ausgeführt werden können. Dazu gehören Prozessoren (CPUs) wie auch Grafikprozessoren (GPUs). Die Programme für die parallele Ausführung werden als sogenannte *kernel* erstellt, welche in der prozeduralen Programmiersprache *OpenCL C* verfasst werden.

Die Sprache ist an den *C99*-Standard angelehnt, bietet aber einige Erweiterungen speziell für parallele Programmierung. Nicht möglich sind in dieser Sprache architekturbedingt Rekursion, Arrays variabler Länge und dynamische Speicherallokierung.

Abbildung 7: OpenCL Logo [10]

OpenCL basiert auf einer Host-Device Architektur. Diese ist so entworfen, dass es immer genau einen Host geben muss. Dieser muss eine CPU sein und ist für den Kontrollfluss zuständig. Jedem Host können dann mehrere Devices zugewiesen werden. Dabei kann es sich um alle möglichen Hardwareplattformen handeln. Im Regen-Framework wird standardmäßig immer eine GPU als Device verwendet. Die entsprechende Konfiguration mit der Hardware des Referenzsystems ist in Abbildung 8 schematisch aufgeführt. Näheres zum Referenzsystem ist Abschnitt 4.6 zu entnehmen. [32]

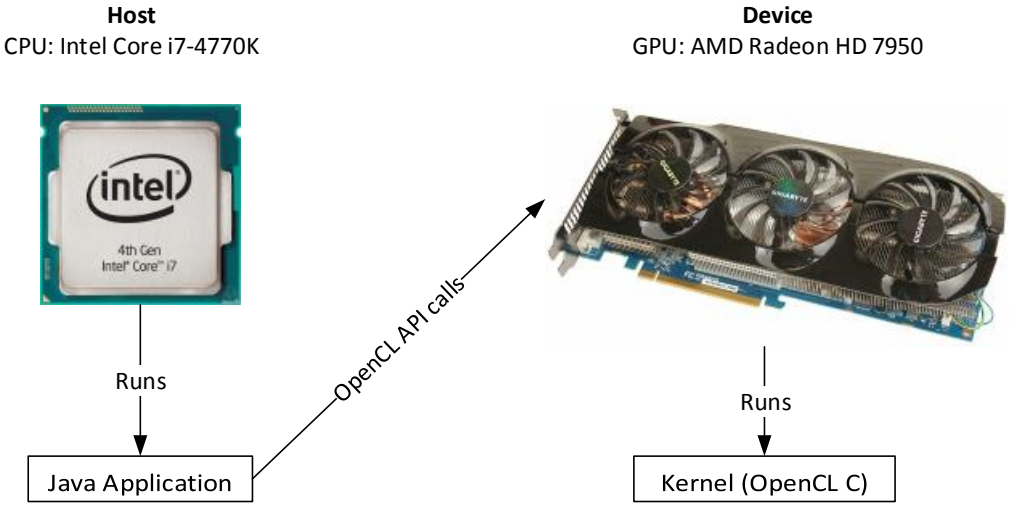

Abbildung 8: OpenCL Host-Device-Architektur mit der Hardware des Referenzsystems

2.5 Grundlegende 3D-Engine

Als Grundlage des Frameworks wird eine, im Zuge der im Sommersemester 2012 an der Universität Osnabrück von *Henning Wenke* gehaltenen Computergrafikvorlesung, bereit-

gestellte, beziehungsweise entwickelte, 3D-Engine verwendet. Diese bietet unter anderem Hilfsklassen zur einfachen Erstellung von Basisgeometrien, Texturen, einer Kamera sowie eine Klasse zur Verwaltung von *Shader*-Programmen. Die Funktionen zum Erstellen des OpenCL *Context* und der Initialisierung von OpenGL werden ebenfalls bereitgestellt. Einige der Klassen oder deren Grundlagen wurden dabei von *Nico Marniok* und *Sascha Kolodzey* entwickelt.

2.6 Hardwareanforderungen

Für die Ausführung des Regen-Frameworks ist Grafikhardware notwendig, die mindestens den OpenGL 3.3 Standard sowie OpenCL 1.1 hardware- und treiberseitig unterstützt. Dies sind alle dedizierten *Radeon™ HD* Grafikkarten der Firma *AMD* ab der Modellreihe 5000, die im Jahr 2009 auf den Markt gebracht wurden. Dedizierte Grafikhardware der Firma *NVIDIA* unterstützt die benötigten Standards ab der *GeForce 400 Series*, deren Markteinführung im Jahr 2010 stattfand. Integrierte Grafikchips der Firma *Intel* unterstützen seit der siebten Generation die Anforderungen. Diese Serie wurde im Jahr 2012 mit der *IvyBridge*-Prozessorarchitektur eingeführt. Eine aktuelle dedizierte Grafikkarte sowie die neueste zugehörige Treiberversion ist für einen flüssigen Betrieb der Anwendung empfehlenswert. Einen Überblick über eine Auswahl an neuerer Consumer-Grafikhardware und die von ihnen unterstützten Standards unter dem Betriebssystem Microsoft Windows, bietet Tabelle 1 (Stand: August 2013). [1] [17] [8] [19]

Tabelle 1: Grafikhardware und unterstützte Standards (Auswahl) [39] [36] [34]

Hersteller und Marke	Modellserie	OpenGL	OpenCL	Einführung
AMD Radeon HD	2000	3.3	1.0	2006-2007
	4000	3.3	1.0	2008
	5000	4.3	1.2	2009
	6000	4.3	1.2	2010
	7000	4.3	1.2	2010
	8000	4.3	1.2	2012
Intel HD Graphics	5^{th} (Ironlake)	2.1	-	2010
	6^{th} (Sandy Bridge)	3.1	-	2011
	7^{th} (Ivy Bridge)	4.0	1.2	2012
	7^{th} (Haswell)	4.0	1.2	2013
NVIDIA GeForce	8 Series	3.3	-	2006
	9 Series	3.3	-	2008
	200 Series	3.3	-	2008-2009
	400 Series	4.4	1.1	2010
	600 Series	4.4	1.1	2012
	700 Series	4.4	1.1	2013

3 Regen und Wasser in der Natur

Für eine möglichst realistische Regen- und Wasserdarstellung ist eine Betrachtung der Phänomene in der Natur unerlässlich. Dieser Abschnitt gibt einen groben Überblick über die physikalischen Eigenschaften von Regenfall und Wasseroberflächen und geht auf einige Eigenschaften dieses Elements beziehungsweise dieser Niederschlagsform ein.

3.1 Regen in der Natur

Regen besteht aus Tropfen flüssigen Wassers. Er entsteht, wenn Wasserdampf in der Atmosphäre kondensiert und dann, bedingt durch die Gravitationskraft, in Richtung Erde fällt. Regenfall ist die häufigste Form von Niederschlag. In Osnabrück beispielsweise regnete es in den Jahren 1991 bis 2010 durchschnittlich mehr als jeden zweiten Tag im Jahr [33]. Oftmals tritt Regen in Kombination mit Wind auf. Dieser verändert dann die Fallrichtung des Regens und sollte deshalb für eine realistische Simulation in die Berechnung der Tropfenfallrichtungen einbezogen werden.

3.1.1 Tropfenform und -größe

Viele Menschen assoziieren mit dem Begriff “Regentropfen” eine Tränenform. Dies entspricht jedoch nicht der Realität. Regentropfen besitzen in der Regel eine Größe zwischen circa 0.1 und sechs Millimetern. Jedoch sind vereinzelt auch größere Tropfen möglich, der größte fotografierte Tropfen besaß einen Durchmesser von zehn Millimetern [20]. Die Form des Regentropfens ist insbesondere von seinem Umfang abhängig. Tropfen mit einem Durchmesser von unter zwei Millimetern besitzen eine annähernd sphärische Form. Je größer der Tropfen ist, desto mehr gleicht seine Form – bedingt durch den beim Fallen entstehenden Luftwiderstand – der einer Kugelschale beziehungsweise eines Fallschirms (vgl. Abbildung 9). Wird der Druck durch den Luftwiderstand an der Unterseite des Tropfen größer als die Oberflächenspannung, so zerfällt der Tropfen in kleinere Teiltropfen. Dies ist in der Regel bei Tropfen, die größer als fünf Millimeter sind, der Fall. Die Grenze hängt jedoch noch von anderen Faktoren, wie der Temperatur, ab. [5] [40]

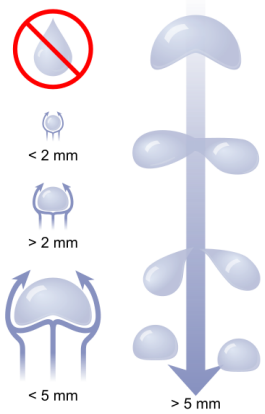

Abbildung 9: Querschnitt eines Regentropfens bei unterschiedlichem Durchmesser (links) und Spaltungsprozess bei großen Tropfen (rechts) [40]

3.1.2 Regenformen

Regen lässt sich anhand seines Entstehungsprozesses in drei Formen klassifizieren [40]:

- Steigungsregen
  Dieser entsteht an Hängen, wenn feuchte Luftmassen über einem Gewässer durch Wind gegen den Hang gedrückt werden. Durch die so entstehende Steigung kühlt die Luft ab und es entsteht Steigungsregen.

- Konvektionsregen
  Diese Form des Regens kommt vor allem in tropischen Regionen oder in der Sommerzeit auch in gemäßigten Zonen vor. Der Regen entsteht durch Konvektionsströme.

- Frontalregen
  Diese Form des Regens entsteht, wenn eine kalte und eine warme Luftfront aufeinandertreffen.

Neben diesen meteorologischen Definitionen gibt es noch andere Klassifikationen, die meist durch Dauer, Intensität und gegebenenfalls andere Faktoren erfolgen. Da leichter Regen und Nieselregen bei Tageslicht meist nur schwer sichtbar sind, beschränke ich mich bei meinem Framework auf die Darstellung von mäßigem und starkem Regen bis hin zu Wolkenbrüchen, einer Form von sehr starkem Regen. In Tabelle 2 aus [13, S. 13] sind die genannten, nach Intensität klassifizierten Regenformen mit durchschnittlichen Werten für Menge, Geschwindigkeit und Anzahl der Tropfen pro Quadratmeter, aufgeführt.

Tabelle 2: Regenformen und -daten [13]

Bezeichnung	Intensität [cm/h]	Menge [mm]	Fallgeschwindigkeit [m/s]	Anzahl Tropfen pro Sekunde pro Meter^2
Nieselregen	0,025	0,96	4,1	151
Leichter Regen	0,102	1,24	4,8	280
Mäßiger Regen	1,381	1,60	5,7	495
Starkregen	1,520	2,05	6,7	495
Exzessiver Regen	4,064	2,40	7,3	818
Wolkenbruch	10,160	2,85	7,9	1216

3.1.3 Fallgeschwindigkeit und Erscheinung

Die Fallgeschwindigkeit der Regentropfen ist primär von ihrer Größe abhängig. Dies kann man mit der höheren Masse bei größerem Durchmesser begründen. Jedoch erhöht sich auch der Luftwiderstand anhand der veränderten Form (vgl. Abbildung 9). Durch ein Kräftegleichgewicht zwischen Gravitations- und Widerstandskräften fallen Regentropfen in der Regel nach einer gewissen Zeit annähernd konstant [41]. In [7] werden Messungen von *Philipp Lenard* beschrieben, der mit Hilfe eines Windtunnels die Geschwindigkeit von Tropfen bestimmte. Nach seinen Messungen erreichen diese größenabhängige Geschwindigkeiten zwischen vier und acht Metern pro Sekunde.

Dies hat zur Folge, dass fallende Regentropfen insbesondere bei stärkerem Regenfall, bedingt durch die Trägheit des Auges, als vertikale Striche wahrgenommen werden. Durch die Reflektierenden und transparenten Eigenschaften von Wasser (vgl. Abschnitt 3.2.2) nehmen die meisten Menschen diese Striche in einer gräulich-transparenten Farbe wahr. Dies ist begründet in der Reflexion des Himmels, welcher in der Regel grau durch die Bewölkung während eines Regenschauers ist. Diese Striche habe ich versucht in meinem Framewok abzubilden, um somit einen realistischen Eindruck des Regens zu vermitteln. Abbildung 10 zeigt eine Szene mit starkem Regen. Die vertikalen Striche sind hier gut erkennbar. Der Effekt lässt sich bei Aufnahmen durch Foto- oder Videoapparate anhand der Blendenzeit und Aufnahmeoptik in einem gewissen Rahmen verstärken oder abschwächen. Da das Phänomen jedoch in Filmen gerne hervorgehoben oder zumindest nicht ausgeblendet wird, und sich Videospiele im Allgemeinen bei der Darstellung an Filmen orientieren, ist dieser Effekt für die Regendarstellung in einer 3D-Anwendung wünschenswert.

Abbildung 10: Aufnahme eines echten Regenschauers, die Regentropfen werden als vertikale Striche wahrgenommen. [41]

3.2 Wasser in der Natur

Das Element Wasser wird schon seit Jahrhunderten von Generationen von Wissenschaftlern untersucht. In diesem Abschnitt werden insbesondere der Versickerungsprozess von Wasser, die sogenannte *Infiltration*, sowie die optische Erscheinung von Wasseroberflächen näher betrachtet. Beides sind Phänomene, die für Darstellung und Berechnung eines realistischen Regensystems unerlässlich sind.

3.2.1 Infiltration

Dringt ein Niederschlag, wie beispielsweise Regen, in den Erdboden ein, so bezeichnet man dies in der Hydrologie als Infiltration. Drei Faktoren spielen bei der Infiltration eine maßgebliche Rolle, die Niederschlagsintensität, die Beschaffenheit des Bodens und die den Boden bedeckende Vegetation. Die Versickerungsrate sinkt mit zunehmender Wassersättigung im Boden. Wenn die Niederschlagsrate die Infiltration übersteigt, so kommt es zu einem Ablauf des Wassers auf der Oberfläche. Das Wasser sammelt sich meistens und fließt dann wie ein Bach in Bereiche mit einem niedrigeren Wasserspiegel. Diese Oberflächenabflüsse sind insbesondere bei starken Regenschauern und wasserundurchlässigen Böden zu beobachten. Abbildung 11 zeigt einen solchen Abfluss. [37]

Abbildung 11: Oberflächenabfluss bei übersättigtem Boden [43]

3.2.2 Erscheinung und Wahrnehmung

Transparenz und Farbe Licht wird im sichtbaren Spektralbereich, also Lichtwellenlängen zwischen ungefähr $380nm$ und $780nm$, vom Medium Wasser kaum absorbiert. Daher erscheint es im Allgemeinen als farblos und transparent. Am geringsten ist die Absorption im unteren sichtbaren Spektralbereich, bei etwa $400 - 440nm$. Dies ist auch der Grund, warum tiefe Gewässer einen leicht bläuliche Färbung aufweisen (Blau besitzt eine Wellenlänge von $420 - 480nm$). Dieser Effekt beginnt mit der Auslöschung von rotem Licht im oberen sichtbaren Bereich, ab circa viereinhalb Metern Tiefe, ist also nur bei relativ tiefen Gewässern überhaupt sichtbar. Ansonsten kann man bei flachen, sauberen Gewässern aufgrund der Transparenz oftmals den Boden gut und farbecht erkennen (vgl. Abbildung 12 links). [35]

Lichtbrechung Ein gut zu beobachtendes Phänomen sind Reflexion und Brechung von Licht an Wasseroberflächen. Diese entstehen beim Übergang von Lichtstrahlen in ein optisch dichteres Medium, da Licht sich im dichteren Medium langsamer ausbreitet. Luft besitzt einen Brechungsindex von näherungsweise 1,0, während Wasser einen von etwa 1,33 besitzt. Lichtstrahlen werden deshalb beim Luft-Wasser-Übergang zum Lot hin gebrochen. Dieses Phänomen wird auch als *Refraction* bezeichnet. Ein sichtbares Resultat dieser Eigenschaft ist die optische Verzerrung von Objekten unter der Wasseroberfläche. Auch Entfernungen, beispielsweise zum Grund eines Gewässers wirken scheinbar kürzer (siehe Abbildung 12 links). [35]

Abbildung 12: Brechung und Transparenz (links) und Reflexion (rechts) an einer Teichoberfläche

Reflexion Ein weiterer Effekt, der mit der Brechung von Licht in Zusammenhang steht, ist die Reflexion des Lichts an der Wasseroberfläche. Diese beträgt bei senkrechtem Einfallswinkel etwa 2%. Je spitzer der Winkel relativ zur Oberfläche jedoch ist, desto mehr reflektiert diese das Umgebungslicht und damit die Umwelt. Bei einem Winkel von wenigen Grad zwischen Wasseroberfläche und Betrachtungsposition ergibt sich eine Reflexion von nahezu 100%. Die das Wasser umgebende Szene erscheint dann perfekt gespiegelt auf der Oberfläche. Der Effekt, der auch als *Fresnel-Effekt* bekannt ist, lässt sich insbesondere auf

ruhigen Gewässer gut beobachten. In Abbildung 12 (rechts) ist dieser auf der Teichoberfläche zu sehen. Man beachte auch, dass bei spitzen Winkeln (Abb. 12 rechts) eine deutlich höhere Reflexion stattfindet als bei einem weitem Winkel (Abb. 12 links) relativ zur Wasseroberfläche. [35]

Kapillarwellen Meist besitzen Gewässer eine unruhige Oberfläche, da Wellen durch Wind und Strömungen entstehen (vgl. Abbildung 11). Bei Regen lassen sich, vor allem auf ansonsten stillen Gewässern, kleine, ringförmige Wellen beobachten, die entstehen wenn Tropfen auf die Wasseroberfläche treffen. Diese werden Kapillarwellen genannt. Je nach Intensität des Regens sind diese, aufgrund der Anzahl an auftreffenden Tropfen pro Quadratmeter und Sekunde, unterschiedlich gut zu erkennen (vgl. Tabelle 2). Abbildung 13 zeigt die von Tropfen verursachten Kapillarwellen auf einer Wasseroberfläche.

Abbildung 13: Kapillarwellen auf einer Wasseroberfläche

4 Techniken und Implementation

4.1 Darstellung des Regens

4.1.1 Stand der Technik

Es gibt verschiedene Methoden in der Computergrafik, die in den letzten Jahren entwickelt wurden, um Regen in 3D-Szenen zu rendern. Diese lassen sich grob in drei Kategorien aufteilen, die in diesem Abschnitt näher betrachtet werden. Die Implementationen beschränken sich teilweise nicht allein auf eine der Techniken, auch verschiedene Kombinationen findet man in der Literatur. Als Überblick des aktuellen Standes der Technik in diesem Bereich, werden einige existierende Umsetzungen der Regendarstellung aus den letzten Jahren und die darin verwendeten Techniken beschrieben. Diese sind am Ende des Unterkapitels in Tabelle 3 vergleichend aufgeführt. Man beachte, dass die dort angegebenen Partikelanzahlen mit der zur Entwicklungszeit aktuellen Hardware erreicht wurden. Mit im Jahr 2013 aktueller Hardware könnten diese demnach unter Umständen deutlich höher ausfallen.

Bildbasierende Techniken Mit diesen Techniken wird versucht, den Regeneffekt in einzelne Bilder einzufügen. Die zu manipulierenden Bilder können aus einer Szene oder einem Video stammen. Um Regen fotorealistisch darzustellen, müssen viele Faktoren beachtet werden, insbesondere die Lichtquellen sowie die Blickrichtung. Bildbasierende Techniken, wie sie beispielsweise in [6] beschrieben werden, versuchen möglichst viele dieser Faktoren physikalisch akkurat umzusetzen. Garg und Nayar, die Autoren des Artikels, benutzen hierfür eine umfassende, selbst erstellte Texturdatenbank mit tausenden Regentexturen für unterschiedliche Lichtquellen und Blickwinkel. Wegen des Umfangs einer solchen Datenbank und der Berechnungen, eignet sich ein vergleichbarer Ansatz in diesem Umfang nur bedingt für interaktive Echtzeitanwendungen. Jedoch werden einige interessante Techniken eingesetzt, die in ähnlicher Form auch in Echtzeitanwendungen verwendet werden. Außerdem gibt es verschiedene Ansätze um einem Video einen Regeneffekt in Post-Processing hinzuzufügen. S. Starik et al. stellten bereits 2003 in [24] eine entsprechende Technik vor. Einen erweiterten und eindrucksvolleren Ansatz beschreiben L. Wang und seine Koautoren in [29]. In ihrem Verfahren analysieren sie Videos von Regenszenen off-line und extrahieren die Regenerscheinung, um sie dann in Echtzeit in anderen Videosequenzen hinzuzufügen. Abbildung 14 zeigt ein solches Bild aus einer Videosequenz, das mit einem Regeneffekt zur Laufzeit versehen wird.

Abbildung 14: Regeneffekt wird zu einer Videosequenz einer Straßenszene hinzugefügt [29]

Scrolling Textures Die grundlegende Idee dieses Ansatzes beruht auf einer sich vertikal bewegenden Textur. Diese deckt in der Regel die ganze Szene ab und wird kontinuierlich in Fallrichtung des Regens gerollt. Eine entsprechende Umsetzung wird in [26] beschrieben. Die Autoren setzen dort mehrere Schichten dieser Texturen ein, um den Regen in verschiedenen Entfernungen zum Betrachter darzustellen und somit einen realistischen Eindruck der Regenszene zu vermitteln. Jedoch benutzen die Autoren auch zusätzlich ein Patrikelsystem, um einige Probleme der Technik zu umgehen. Abbildung 15 zeigt eine Szene aus der in [26] vorgestellten Demoanwendung. Ein Problem bei diesem Ansatz ist, dass die Texturen statisch eingesetzt werden und eine interaktive Kamera demnach, wenn überhaupt, nur mit großem Aufwand umsetzbar ist. Hinzu kommt, dass sich die Texturen nur als Ganzes beeinflussen lassen. Das heißt im Konkreten, dass beispielsweise Windeffekte, die die Fallrichtung der Regentropfen verschieden beeinflussen, sowie dynamische Lichtquellen, die sich in den Regentropfen reflektieren, in Echtzeit nicht mehr umzusetzen sind.

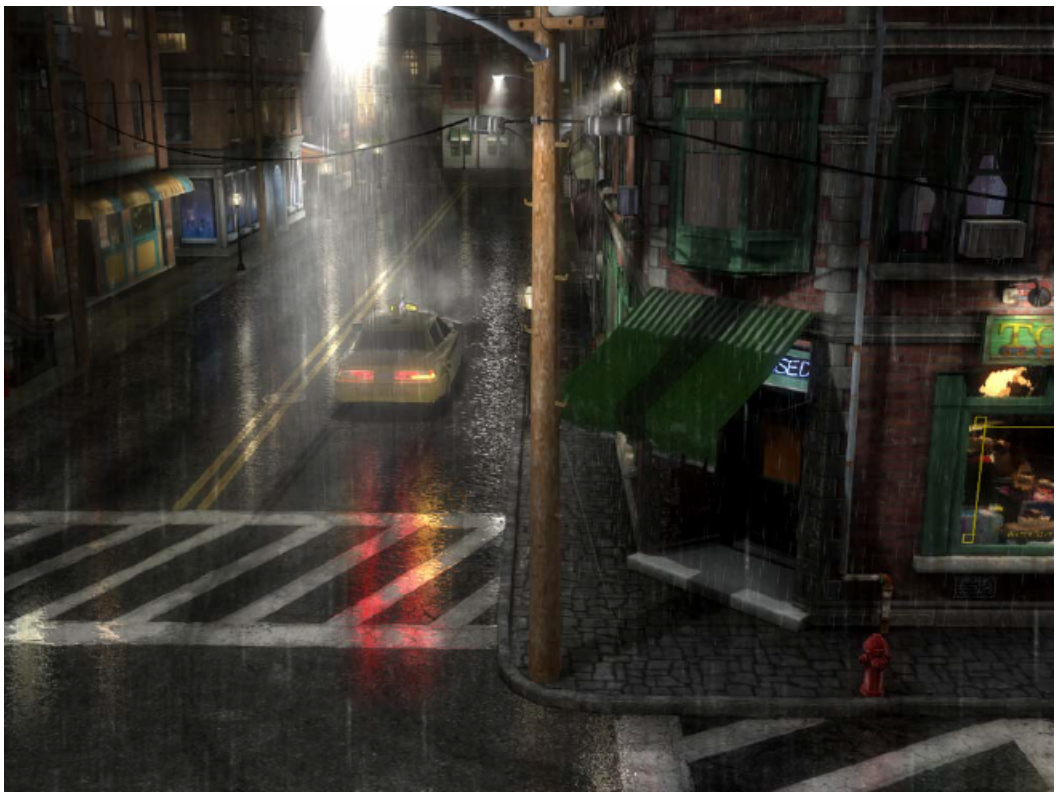

Abbildung 15: Screenshot aus der ToyShop-Demoanwendung mit Scrolling Textures [26]

Ein anderer Ansatz mit Scrolling Textures wird in [30] vorgestellt. Texturen werden von den Autoren auf einen Doppelkegel abgebildet und mittels *hardware texture transformation* skaliert und versetzt. Aber auch bei dieser Technik fehlt eine Umgebungsinteraktion. Die Ergebnisse vermitteln deshalb einen eher unrealistischen Eindruck.

Partikelsysteme Partikelsysteme werden in Echtzeitanwendungen gerne verwendet, um undeutliche oder sich schnell ändernde Phänomene und Effekte darzustellen, da diese in der Regel schwer mit konventionellen Techniken der Rastergrafik umzusetzen sind. Beispiele für solche Effekte sind Feuer, Funken, Rauch und Explosionen aber auch Wetter- und Umweltphänomene wie Wolken, Nebel, Schnee oder Regen. Ein großer Nachteil dieser Technik ist, dass für einen hohen Detailreichtum sehr viele Partikel berechnet und dargestellt werden müssen, worunter die Performance leiden kann. Viele Arbeiten zu dieser Thematik legen deshalb einen Schwerpunkt auf die Optimierung bezüglich der Performance von großen Partikelsystemen.

In [21] präsentieren die Autoren ein solches Partikelsystem. In ihrer Arbeit legen sie viel Wert auf die Darstellung der Regentropfen in Form von Reflexionen und Brechung der Szene. Die reflektierte Szene wird als verformte Texturen auf alle Regentropfen abgebildet und bei der Darstellung auch dynamische Lichtquellen einbezogen. Ein sehr interessanter Ansatz wird von S.Tariq in [25] vorgestellt. Mit den damals neuen Möglichkeiten des *Shader Model 4.0* (unter anderem Einführung des *Geometry Shader*, *Texture Arrays* und *Transform Feedback*) wird eine sehr realistische Regenumgebung geschaffen (siehe Abbildung 16).

Die Regenpartikel werden mit Hilfe des *Geometry Shader* in *Sprites* erweitert und mit einer Textur aus der in [6] vorgestellten Datenbank belegt. Anhand von Betrachterstandpunkt und Lichtquellenposition wird eine passende Textur gewählt. Die in meinem Framework verwendeten Techniken zur Regendarstellung orientieren sich an den von S. Tariqs in [25] präsentierten. Ein ähnlicher Ansatz wird in [18] vorgestellt. Zusätzlich wird dort unter anderem ein umfassendes *Level of Detail*-Modell (LOD) umgesetzt. Außerdem versuchen die Autoren verschiedene Regenformen (vgl. Abschnitt 3.1.2) und deren Übergänge darzustellen.

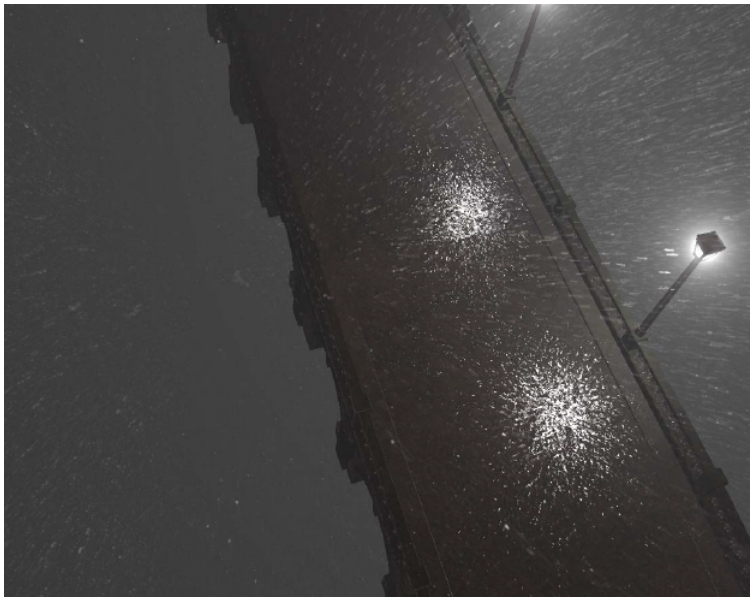

Abbildung 16: Screenshot aus der Anwendung *Rain* (NVIDIA SDK) mit Partikelsystem. [25]

Tabelle 3: Vergleich von Anwendungen zur Regendarstellung [18]

Autoren, Jahr	Technik(en)	Shader	Interakt. Kamera	Reflexion, Brechung	Nebel	Licht-interakt.	Wind	LOD	Sonstiges	Anzahl Partikel
N. Wang, B. Wade Microsoft Corp. 2004 [30]	Scrolling Textures	Vertex	✓	✗	Blur	✗	✗	✗	Schnee	
L. Wang et al. Microsoft Corp. 2006 [29]	Partikel-system	Vertex, Fragment	✓	✗	✓	✓	✓	✗	Spritzer, Kapillarwellen	10 000 - 80 000
N. Tatarchuk ATI Corp. 2006 [26]	Scolling textures, Partikelsys.	Vertex, Fragment	✗	✓	✓	✓	✓	✗	Spritzer, Kapillarwellen	5 000 - 20 000
P. Rousseau et al. Limoges University 2006 [21]	Partikel-system	Vertex, Fragment	✓	✓	✗	✓	✗	✗	Schnee	1 000 - 10 000
S. Tariq NVIDIA Corp. 2007 [25]	Partikel-system	Vertex, Fragment, Geometry	✓	✗	✓	✓	✓	✗	Kapillarwellen	50 000 - 150 000
A. Puig-Centelles et al. Castellón University 2009 [18]	Partikel-system	Vertex, Fragment, Geometry	✓	✗	✓	✗	✓	✓	Regenzonen mit Übergängen	25 000 - 50 000
Mein Ansatz 2013	Partikel-system	Vertex, Fragment, Geometry	✓	✓	✓	✓	✓	✓	Wasseransammlungen, Kapillarwellen	bis 8 Mio.

4.1.2 Partikelsystem

Aufgrund der in Abschnitt 4.1.1 erwähnten Nachteile beim Einsatz von *Scrolling Textures* entschloss ich mich für den Einsatz eines Partikelsystems. Wünschenswert wäre ein System, in dem jeder Regentropfen durch genau einen Partikel repräsentiert wird. Dies hätte den Vorteil, dass man einen exakten Kollisionspunkt für jeden Tropfen mit der Landschaft berechnen könnte und somit entsprechende Interaktionseffekte leicht umzusetzen wären. Zusätzlich könnte derselbe Partikel dann als Wasserpartikel auf oder im Boden "weiterleben" und für die Infiltrations- beziehungsweise Fließberechnungen verwendet werden.

Leider hat diese Art der Umsetzung einen gravierenden Nachteil, der eine Echtzeitumsetzung unmöglich macht. Dieser liegt in der Anzahl der Regentropfen bei einem normalen Regenschauer begründet (vgl. Tabelle 2). Wollte man eine Landschaft mit etwa 100 Quadratmetern Fläche darstellen, so würden bei mäßigem Regen etwa fünf Millionen Partikel pro Sekunde allein auf den Boden auftreffen. Hinzu kämen noch die fallenden sowie die angesammelten Wasserpartikel auf und im Boden. Diese Anzahl von Partikeln ist mit heute erhältlicher Hardware nicht berechenbar beziehungsweise darstellbar. Also bestand die Herausforderung zunächst darin, Techniken zu entwerfen und umzusetzen, die dieses Problem umgehen und dennoch ansehnliche Ergebnisse produzieren. Diese Umsetzung wird im Folgenden beschrieben.

Alle darzustellenden Regentropfen werden zunächst in variabler Anzahl als einfacher *Primitive Type* GL_POINTS erstellt (vgl. Abschnitt 2.1.1). Um die Anzahl und somit den Rechenaufwand klein zu halten, werden Partikel ausschließlich in einem festgelegten Zylinder um die Kamera erstellt. Dies vermittelt für den Anwender den Eindruck, als würde es im ganzen sichtbaren Bereich regnen. Um den Eindruck noch realistischer zu gestalten, werden die Partikel nach einem *Level-of-Detail*-System innerhalb des Zylinders verteilt. Der Zylinder wird dafür in der xz-Ebene in mehrere kreisförmige Flächen um die Kamera unterteilt. Je näher die Fläche am Betrachterstandpunkt liegt, desto mehr Partikel werden in diesem Bereich gestreut. Die Verteilung ist in Abbildung 17 (rechts) veranschaulicht. Auf der linken Seite der Abbildung ist der Zylinder um die Kamera visualisiert, in dem sich die Partikel aufhalten. Dadurch, dass auch Partikel im Halbzylinder hinter der Kameraposition berechnet werden, sind sehr schnelle Kameraschwenks bei der Darstellung kein Problem.

Zu jedem Partikel werden seine initiale Startposition (*seed*), die aktuelle Position, und ein Geschwindigkeitsvektor der auch die Richtung eindeutig angibt, gespeichert. Die Variablenwerte besitzen dabei einfache Genauigkeit. Außerdem werden noch zwei Zufallswerte pro Partikel gespeichert, deren Verwendung in Abschnitt 4.1.4 über Beleuchtung, erläutert wird. Die Position der Partikel wird dann in diskreten Zeitschritten verändert. Die beeinflussenden physikalischen Größen, die in der Simulation abgebildet werden, sind die Gravitationskraft sowie der Wind. Die Auswirkungen der Erdanziehung werden in der y-Komponente des Geschwindigkeitsvektors initial gesetzt. Da Regentropfen mit einer näherungsweise konstanten Geschwindigkeit fallen (vgl. Abschnitt 3.1.3), wird dieser Wert auch nicht verändert. Außerdem werden die x- und z-Werte des Geschwindigkeitsvektors initial auf einen sehr kleinen, zufälligen Wert gesetzt. Dies hat zum Ziel, dass keine Muster zu erkennen sind und dass übermäßiges *Aliasing* verhindert wird.

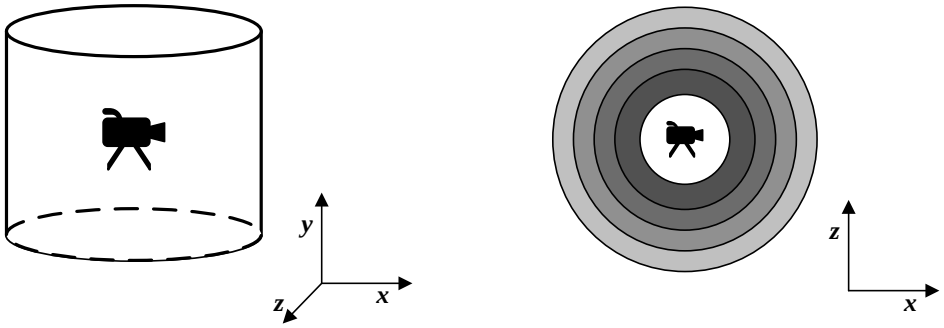

Abbildung 17: Partikelpositionen werden in einem Zylinder um die Kamera gesetzt (links), *Level-of-Detail*-Verteilung der Partikel in der xz-Ebene (rechts).

Die Windrichtung und -stärke beeinflusst im Gegensatz zur Gravitationskraft die x- und z-Komponenten der Geschwindigkeitsvektoren stark. Während die Windrichtung zufällig bei Programmstart gesetzt wird, kann der Anwender die Stärke beeinflussen. Die entsprechenden Berechnungen zur Bewegung der Partikel wird mit einem *OpenCL C kernel* realisiert und damit vollständig parallel auf der Grafikhardware berechnet. Außerdem kann auf den Einsatz von eher umständlichen und fehleranfälligen Techniken wie das in [25] eingesetzte *Transaform Feedback* verzichtet werden. Erreicht ein Partikel das Ende des Zylinders, so wird es entsprechend seines *seed* neu platziert.

Somit entsteht der Eindruck eines kontinuierlichen Regenfalls bei einer festen Anzahl an Partikeln. Bewegt der Anwender die Kamera, so werden die Partikel, die das Ende des Zylinders erreichen, entsprechend des neuen Betrachtungsstandpunktes neu gesetzt. Der Zylinder zieht also gewissermaßen mit der Kamera mit. Dank der relativ hohen Fallgeschwindigkeit von Regentropfen, die in die Simulation übertragen wurde, überzeugt diese Technik bis zu einer moderaten Kamerageschwindigkeit. Ausschließlich bei sehr schnellen beziehungsweise weiten Standpunktwechseln kommt es zu unerwünschten Darstellungsfehlern.

4.1.3 Billboard-Technik

In Abschnitt 3.1.3 wird beschrieben, dass wir Regen auf Grund der Trägheit des Auges als vertikale Linien wahrnehmen. Eine Umsetzung der Tropfen als einfache Punkte mit GL_POINTS ist deshalb für eine realistische Darstellung nicht ausreichend. Um die, durch das Partikelsystem wegen der einfacheren Handhabung verwendeten Punkte dennoch verwenden zu können, bietet sich eine Umsetzung der *Billboard*-Technik mit Hilfe des *Geometry Shaders* an. Die nach Werbetafel an Nordamerikanischen Highways benannte Technik wird eingesetzt, um texturierte *Sprites* abhängig vom Betrachterstandpunkt immer so zu drehen, dass sie orthogonal zur Kamera stehen. Das heißt insbesondere, dass die Vorderseiten der Sprites immer vollständig zu sehen sind. Zwei Screenshots einer Beispielimplementation mit dem Logo der Universität Osnabrück als Textur auf den Sprites ist in Abbildung 18 zu sehen.

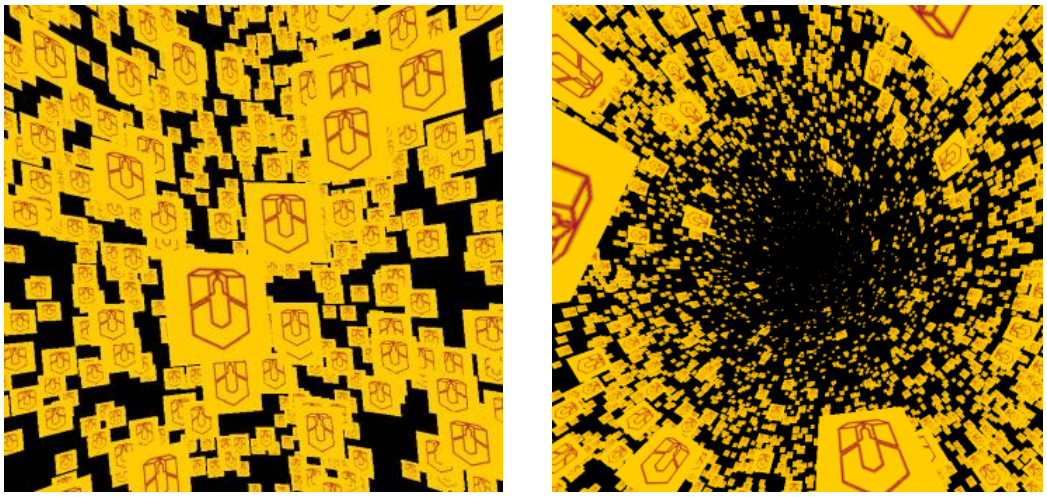

Abbildung 18: Screenshots einer Beispielimplementation der Billboard-Technik. Blickrichtung geradeaus (links) und nach oben (rechts).

Die verwendeten Sprites bestehen jeweils aus zwei Dreiecken. Mit Hilfe des *Geometry Shaders* werden aus jedem als GL_POINTS interpretierten Punkt jeweils vier erstellt. Mit diesen ist es möglich, für jeden Partikel durch die *Primitive*-Form GL_TRIANGLE_STRIP zwei Dreiecke zu erstellen. Diese zwei Dreiecke ergeben dann das für die Technik benötigte Sprite, dessen Vorderseite texturiert werden kann. Die Größe wird anhand der Entfernung dahingehend angepasst, dass Sprites, die näher an der Kamera liegen, dünner sind. Damit sind Tropfen die sich in größerer Entfernung zur Kamera befinden noch gut sichtbar. Zusätzlich versperren solche, die im Sichtfeld sehr weit vorne liegen, nicht die Sicht beziehungsweise erscheinen nicht unnatürlich breit. Die Umsetzung der *Billboard*-Technik mit dem *Geometry Shader* ist in Listing 3 (verkürzt) aufgeführt.

```
1  #version 330 core
2
3  layout (points) in;
4  layout (triangle_strip, max_vertices = 4) out;
5
6  in VertexData
7  {
8      float texArrayID;
9      float randEnlight;
10 } vertex[];
11
12 uniform mat4 viewProj;
13 uniform vec3 eyePosition;
14 uniform vec3 windDir;
15 uniform float dt;
16
17 out vec3 fragmentTexCoords;
18 out float randEnlight;
19 out float texArrayID;
20
```

```
21 // GS for billboard technique (create two triangles from one vertex).
22 void main(void)
23 {
24     //streak size
25     float height = 0.25;
26     float width = height/50.0;
27
28     vec3 pos = gl_in[0].gl_Position.xyz;
29     vec3 toCamera = normalize(eyePosition - pos);
30     vec3 up = vec3(0.0, 1.0, 0.0);
31     vec3 right = cross(toCamera, up) * width * length(eyePosition - pos);
32
33     //bottom left
34     pos -= right;
35     fragmentTexCoords.xy = vec2(0, 0);
36     fragmentTexCoords.z = vertex[0].texArrayID;
37     randEnlight = vertex[0].randEnlight;
38     texArrayID = vertex[0].texArrayID;
39     gl_Position = viewProj * vec4(pos + (windDir*dt), 1.0);
40     EmitVertex();
41
42     //top left
43     pos.y += height;
44     [...]
45     EmitVertex();
46
47     //bottom right
48     pos.y -= height;
49     pos += right;
50     [...]
51     EmitVertex();
52
53     //top right
54     pos.y += height;
55     [...]
56     EmitVertex();
57
58     EndPrimitive();
59 }
```

Listing 3: Geometry Shader zur Umsetzung der Billboard-Technik für Regenpartikel

Im Vergleich zum *Vertex-* oder *Fragment-Shader* (siehe Abschnitt 2.1.3) unterscheidet sich der *Geometry Shader* in den Codezeilen zu Beginn des Programms. In den Zeilen 3 und 4 werden die Ein- und Ausgabe-*Primitives* festgelegt. In diesem Fall werden aus allen eingehenden GL_POINTS jeweils vier Vertices erzeugt und als zwei Dreiecke vom Typ GL_TRIANGLE_STRIP ausgegeben. Die eingehenden Vertex-Daten müssen außerdem als Array behandelt werden (Zeilen 6 - 10). Dies ist vor allem bei *Primitive*-Typen mit mehreren Vertices wichtig. In den Zeilen 28 - 31 wird der Versatzvektor right für die neu zu erzeugenden Vertices berechnet. Dies geschieht mit Hilfe des Kameravektors so, dass das gerade zu bearbeitende Sprite in Richtung des Betrachters ausgerichtet wird. Auch die Größenskalierung anhand der Entfernung findet hier statt. Danach werden die Positionen der vier Vertices mit dem Versatzvektor bestimmt und die Vertices mit den entsprechenden Daten und der Buid-In-Methode EmitVertex() erzeugt. Dem Sprite wird außerdem anhand der Windparameter eine entsprechende Schräglage verliehen. Die Methode EndPrimitive() (Zeile 58) signalisiert, dass das gewünschte *Primitive* – in diesem Fall GL_TRIANGLE_STRIP – abgeschlossen ist. Dies verhindert, dass die einzelnen *Billbord Sprites* untereinander verbunden

werden. Der Vorgang ist in Abbildung 19 veranschaulicht.

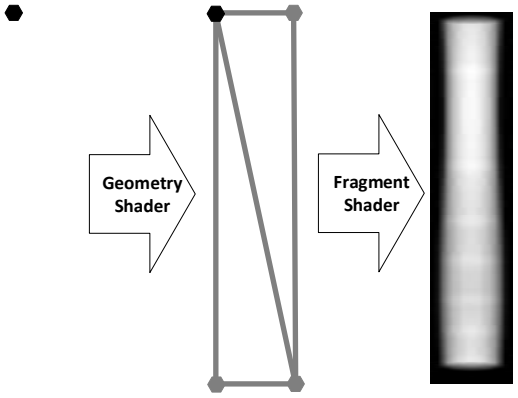

Abbildung 19: Vom Partikel (links) zum *Sprite* (Mitte) zur Regentextur (rechts)

4.1.4 Beleuchtung und Texturen

Für eine realistische Regendarstellung sind ein hoher Detailgrad der Regentropfen sowie insbesondere eine Interaktion der Regentropfen mit einer oder mehreren Lichtquellen wünschenswert. In meinem Framework ist dies mit geschickter Texturierung der Sprites umgesetzt. Kern der Umsetzung bildet dabei die in Abschnitt 4.1.1 erwähnte Arbeit [6]. K. Garg und S. K. Nayar erstellten zu ihrer Arbeit eine umfassende Datenbank mit Tausenden von Regentexturen. Dabei orientierten sie sich an Fotoaufnahmen echter Regentropfen. Praktischerweise sind die in der frei zugänglichen Datenbank gespeicherten Texturen nach Betrachtungsstandpunkt und Lichteinfall geordnet. Dies ermöglicht eine Einbindung der Texturen dergestalt, dass die passende Tropfentextur anhand der Licht-, und Kamerapositionsvektoren herausgesucht werden kann. Obwohl die Autoren die Texturen für die Datenbank berechneten, sind diese fast nicht von echten Aufnahmen zu unterscheiden. Für meine Zwecke sind sie in jedem Fall ausreichend. Abbildung 20 zeigt einige der Texturen im Vergleich zu den Fotografien. Außerdem kann man die Einteilung nach Betrachtungs- und Lichtwinkel sehen.

K. Gargs Datenbank wurde mit dem Ziel, für Regendarstellung in Filmen eingesetzt zu werden, erstellt. Da Effekte in Filmen in der Regel off-line berechnet werden, sind dort die Limitierungen bezüglich Speicher und Rechenkomplexität deutlich niedriger als beim Echtzeiteinsatz. Dem entsprechend wurde die Datenbank entwickelt. Um diese dennoch verwenden zu können, nahm ich einige Einschränkungen in Kauf. Statt aller Texturen, entschloss ich mich nur 370 zu verwenden. Insbesondere wird auf verschiedene Texturen bei Variation des Kamera-Nickwinkels θ_{view} verzichtet (vgl. Abbildung 21). Für den unkomplizierten Einsatz der Texturen wurde die von OpenGL seit Version 3.3 bereitgestellte Datenstruktur GL_TEXTURE_2D_ARRAY eingesetzt. Diese ermöglicht – analog zum Daten-Array – eine Folge von Texturen im Speicher abzulegen, um dann in Shadern schnell auf die einzelnen Texturen zugreifen zu können.

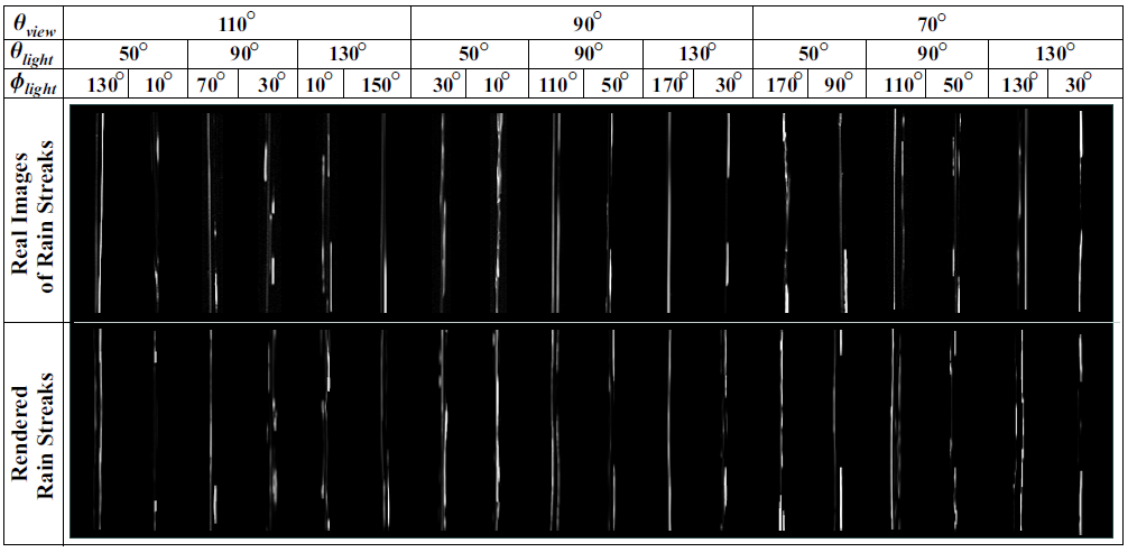

Abbildung 20: Aufnahmen von Regentropfen (oben) und entsprechende Regentexturen aus der Datenbank (unten) [6]

Dafür benötigt man dreidimensionale Texturkoordinaten. Diese werden im *Fragment Shader* anhand des Beleuchtungs- und Kameravektors bestimmt. Die Koordinatensysteme des Partikels und der Kamera sind in Abbildung 21 abgebildet. Entscheidend für die Texturauswahl und somit für die Darstellung sind zwei Winkel. Zum einen ist dies der Nickwinkel der Lichtquelle zum Partikel θ_{light}. Außerdem ist noch der Winkel ϕ_{light} relevant. Dieser befindet sich zwischen der Kameraebene, die durch den Kamerastandpunkt und die Partikelposition eindeutig festgelegt wird, sowie der Lichtebene, die durch die Position der Lichtquelle und die Partikelposition eindeutig festgelegt wird. Neben der Texturauswahl wird so auch der Alphawert ermittelt. Dieser wird für die Transparenz beim *Blending* der Regentexturen verwendet.

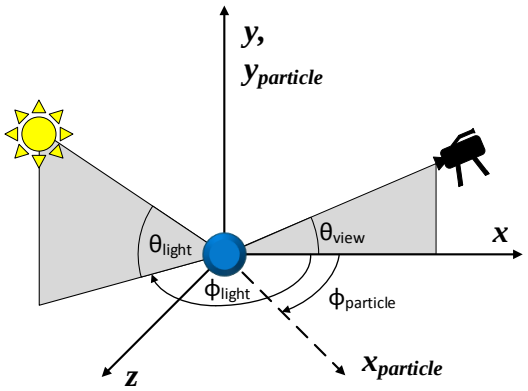

Abbildung 21: Koordinatensysteme (Partikel und Kamera) für die Bestimmung der Regentexturen. [6]

Um eine Variation zu gewährleisten, und somit keine offensichtlichen Muster zu erhalten, wird ein initial gesetzter Zufallswert zur Auswahl von einer aus acht Texturen verwendet. Ein weiterer Zufallswert ist für die Beleuchtung weniger einzelner Regentropfen verantwortlich. So wird sichergestellt, dass unter gewissen Winkeln keine vollständige Transparenz aller Texturen auftritt. Ist die gewünschte Textur geladen, so wird diese auf das *Sprite* abgebildet und anhand des Alphawertes in die Szene geblendet. Die beschriebenen Berechnungen orientieren sich am HLSL-Shader von S. Tariq aus [25]. Abbildung 22 zeigt Screenshots der finalen Regendarstellung in meinem Framework aus verschiedenen Perspektiven und bei verschiedenen Regen- und Windintensitäten. Man beachte insbesondere die Transparenz und Beleuchtung unter verschiedenen Winkeln zur Lichtquelle.

Abbildung 22: Screenshots der Regendarstellung im Framework: unterschiedliche Regen- und Windintensität (oben), verschiedene Blickwinkel bei gleicher Intensität (unten)

4.2 Wasseransammlungen und Fließverhalten

Bei einem Regenschauer sind, bei entsprechendem Untergrund, oftmals Wasseransammlungen zu beobachten. Diese können in Form von Pfützen und kleinen Bächen, bis hin zu teichartigen Gewässern auftreten. Bei felsigem Untergrund tritt dies besonders schnell auf, bei lang anhaltendem oder starkem Regen können aber auch andere Böden übersättigen. Dann findet ein Oberflächenabfluss statt (vgl. Abschnitt 3.2.1). Auch diese Phänomene sollen in meinem Framework nicht fehlen, da sie maßgeblich zum Realitätseindruck beitragen. Ein Regenschauer, dessen Regentropfen einfach im Untergrund verschwinden, ist meiner Ansicht nach nur bedingt glaubwürdig. Für die Umsetzung lohnt es sich, bestehende Techniken zur realistischen Wassersimulation in Echtzeit-3D-Anwendungen zu betrachten. Dies wird im ersten Teilkapitel dieses Abschnitts getan. Danach wird näher auf meine konkrete Umsetzung eingegangen. Am Ende dieses Kapitels findet sich Abbildung 30, eine Übersichtsgrafik, in der der komplette Algorithmus für die Berechnung des Wasserbildungs und -fließverhaltens grafisch zusammengefasst ist.

4.2.1 Stand der Technik

Mit der Wasserdarstellung in Echtzeitanwendungen wurde sich in der Vergangenheit viel befasst. Die angewendeten Techniken lassen sich grob in drei Typen unterteilen: prozedurale Techniken, Partikelsysteme und *Height Field Fluids*. Diese werden im Folgenden inklusive relevanter Arbeiten beziehungsweise Implementationen kurz vorgestellt.

Prozedurale Techniken *Prozedurale Modellierung* in der Computergrafik ist ein Oberbegriff für Techniken, mit denen 3D-Models und Texturen nach gewissen Regeln erstellt werden können. Diese werden oftmals eingesetzt, um aus einer verhältnismäßig kleinen Regelbasis sehr große Szenen zu generieren. Beispiele für den Einsatz prozeduraler Techniken sind die Generierung von Pflanzen, Landschaften und Architekturen. [38]

Abbildung 23: Prozedurale Wasserdarstellung in der OGRE-Demo. [28]

Besagte Techniken können auch angewandt werden, um Wasser und Wellen zu erzeugen. Hierfür eignen sich insbesondere weite, unbegrenzte Oberflächen wie Ozeane oder sehr große Seen. Ein Welleneffekt kann dann, wie in [27] beschrieben, prozedural animiert werden. Eine Umsetzung aus der OGRE-Demo wird in Abbildung 23 gezeigt. Da in meinem Framework Ozeane nicht abgedeckt werden und ich mich auf kleine Gewässer und Wasseransammlungen beschränke, sind diese Techniken für meine Anwendung eher uninteressant.

Partikelsysteme Unter diese Rubrik fallen alle Techniken, die zur Wasserberechnung eine Menge an Partikeln verwenden. Der Einsatz eines Partikelsystems lässt sich leicht nachvollziehen. Denn den Partikeln können physikalische Eigenschaften, wie sie Wasser oder andere Flüssigkeiten besitzen verliehen werden. Beispielsweise können für jedes Partikel Masse, Geschwindigkeit und Viskosität gespeichert werden. Mit Hilfe entsprechender Formeln lässt sich dann eine Interaktion zwischen den Partikeln realisieren. Meist wird als Grundlage *Smoothed Partiles Hydrodynamics* (SPH) verwendet. Dies ist eine numerische Methode, um hydrodynamische Gleichungen zu lösen. Als Lagrange-Methode ist sie relativ einfach zu implementieren. Jedoch steigt bei diesen Techniken die Komplexität mit hohem Realitätsgrad oder einer großen Anzahl an Partikeln schnell an. Deshalb wird bei vielen bisher implementierten Demoanwendungen auf eine feste Anzahl von Partikeln in einem beschränkten Bereich gesetzt oder auf den Echtzeitanspruch gänzlich verzichtet.

Eine dieser Techniken, die echtzeitfähig ist, wird in [15] beschrieben. Ein Screenshot der im Zuge der Arbeit von den Autoren M. Macklin und M. Müller entwickelten Demoanwendung ist in Abbildung 24 (links) abgebildet. In derselben Abbildung sind außerdem die dieser Simulation zugrundeliegenden Partikel zu sehen (rechts). Man beachte, dass sich die Partikel, deren Anzahl feste 128 000 beträgt, hier in einem abgeschlossenen Quader befinden. Nur innerhalb dieser Grenzen interagieren die Partikel. Das simulierte Fließverhalten des Wassers in der Anwendung ist sehr realistisch und kaum mehr von echtem zu unterscheiden. Durch die steigende Komplexität bei hoher Partikelanzahl, sind diese Techniken in einer Echtzeitanwendung nach dem aktuellen Stand jedoch nur für kleine, lokale Wasseransammlungen brauchbar.

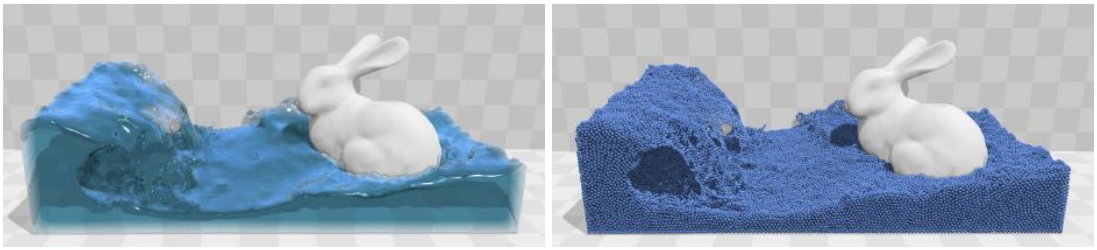

Abbildung 24: Screenshot einer Fluidsimulation auf Partikelbasis (links), zugrundeliegende Simulationspartikel (rechts). [15]

Height Field Fluids Eine, aufgrund moderater Hardwareansprüche gerne verwendete Technik zur Darstellung kleiner Gewässer wie Teiche, Flüsse und Bäche, ist *Height Field Fluids* (HFF). Wie in [16] beschrieben ist die Grundidee dieser Technik, das dreidimensionale Problem der Wassersimulation auf ein zweidimensionales zu vereinfachen. Dadurch fallen die Berechnungen einfacher aus, das heißt der Ressourcenbedarf sinkt. Hierfür wird die als kontinuierlich angesehene Wasseroberfläche zunächst diskretisiert. Für jedes so entstehende Feld in der xz-Ebene wird dann genau ein Skalar für die Wasserhöhe an der entsprechenden Stelle gespeichert. Man erhält also – Analog zur einer Höhenprofilkarte zur Terraingenerierung (vgl. Abschnitt 4.5) – eine Wasserhöhenprofilkarte.

Hält man zusätzlich noch eine Geschwindigkeit pro Feld vor, lässt sich relativ einfach eine simple Wellenanimation umsetzen. Auch eine Objektinteraktion ist mit diesem Verfahren umsetzbar. Das Wasser muss dafür von den Feldern in die das Objekt gelegt wird, an die Nachbarfelder ohne Objekt verteilt werden. Aufgrund des einzelnen Wertes für die Höhe des Wassers sind bei dieser Technik keine brechenden Wellen möglich. Abbildung 25 zeigt einen Screenshot einer HFF-Demoanwendung. Aufgrund der verhältnismäßig geringen Ressourcenansprüche mit dennoch realistischen Ergebnissen ist dieser Ansatz für die Anforderungen meines Frameworks besonders interessant.

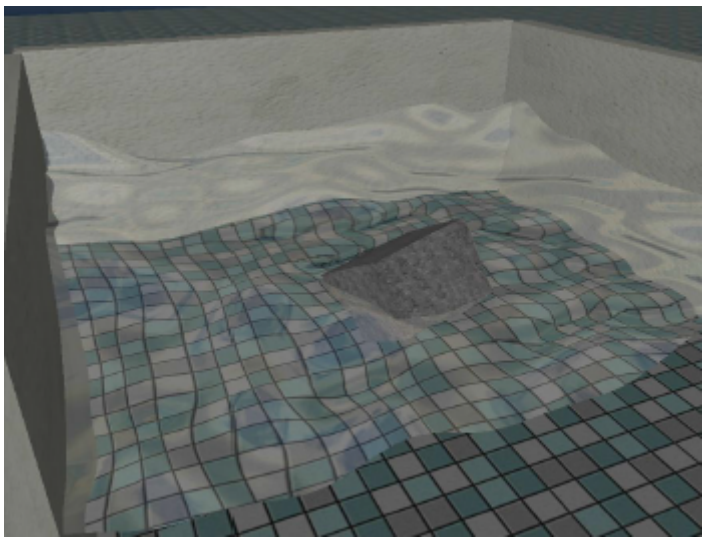

Abbildung 25: Screenshot einer *Hight-Field-Fluids*-Implementation mit Wellengang und Objekt. [16]

4.2.2 Datenstruktur: Diskrete Wasserhöhenprofilkarte

Um die Komplexität der Berechnungen zu vereinfachen, entschloss ich mich nach Vorbild des *Hight-Field-Fluids*-Verfahrens (vgl. Abschnitt 4.2.1) eine diskrete Wasserhöhenprofilkarte als grundlegende Datenstruktur zu verwenden. Das Wasserverhalten kann dann für alle Felder parallel berechnet werden. Dafür wird das verwendete Terrain in ein gleichmäßiges Gitter in der xz-Ebene aufgeteilt. In meinem Framework kommt dabei ein 512^2 großes *Grid* zum Einsatz. Dies ist aber prinzipiell beliebig skalierbar, solange die Hardware nicht überfordert wird. Für das verwendete Terrain ist die Auflösung meiner Meinung nach ausreichend, um einen realistischen Eindruck des Wasser zu vermitteln. Für alle Felder der Wasserkarte werden drei Koordinaten u, v und h gespeichert. Die u- und v-Werte geben die Position der Felder in der Karte an, der h-Wert repräsentiert den Wasserstand an der entsprechenden Stelle. Dieser besitzt initial für alle Felder denselben Wert. Außerdem wird pro Feld eine Geschwindigkeit gespeichert, die für die Wellenausbreitung genutzt wird. In Abbildung 26 ist die Diskretisierung einer kontinuierlichen Wasserfläche schematisch dargestellt.

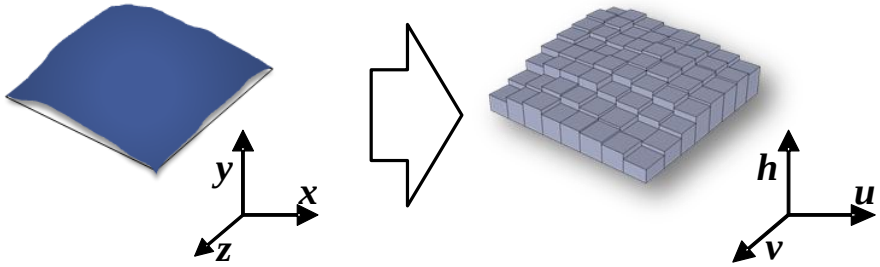

Abbildung 26: Eine kontinuierliche Wasseroberfläche wird diskretisiert.

4.2.3 Regenwasser und Versickerung

Da Regen in der Regel etwa gleichverteilt fällt, wird zunächst jedes Feld der Wasserprofilkarte um denselben Wert, der sich an der aktuellen Regenstärke orientiert, erhöht. Anschließend wird die Versickerung simuliert. Dafür kommt eine *Attribute Map* zum Einsatz, dies ist eine Bilddatei mit derselben Auflösung wie die Wasserkarte. In jedem Pixel ist in dieser als Grauwert die Dichte des Bodens, also die Permeabilität an dieser Stelle, kodiert. Es handelt sich also um eine *Skalar Map*. Eine solche *Attribute Map* lässt sich beispielsweise aus der zugehörigen Terraintextur generieren. Gegebenenfalls muss man dann mehrere Werte interpolieren, um die Auflösung an die der Wasserprofilkarte anzupassen. Mit einem Lookup in der *Attribute Map* pro Feld kann so parallel der Infiltrationsfaktor bestimmt und die Versickerung berechnet werden. Ein Beispiel einer solchen *Attribute Map* mit zugehöriger Textur des Terrains zeigt Abbildung 27.

Abbildung 27: Beispiele für eine Attribute Map (links), eine Höhenprofilkarte (Mitte) und die zugehörige Terraintextur (rechts).

4.2.4 Tangentiales Fließverhalten

Hat sich das Wasser nun auf der Oberfläche angesammelt, so soll es nach Vorbild der Natur in Richtung des (lokal) niedrigsten Punktes fließen. Um der Realität möglichst nahe zu kommen, soll das Wasser außerdem an steilen Gefällen schneller abfließen als in Bereichen des Geländes mit niedriger Steigung. Der Gradient ist also entscheidend für die Abfließgeschwindigkeit. Dafür wird die Steigung jedes, der Felder der Wasserprofilkarte entsprechenden, Feldes benötigt. Im dreidimensionalen Raum gibt die Tangente die Steigung eines Punktes an. Diese erhält man durch Differenzierung der entsprechenden Koordinaten.

Mit Hilfe eines parallel angewandten *Sobel-Algorithmus* wird die benötigte Ableitung der Höhendaten näherungsweise berechnet. Der Sobel-Operator, der unter anderem in der Bildverarbeitung zur Kantenerkennung eingesetzt wird, ist ein diskreter Differentialoperator, mit dem eine Näherung des Gradienten der Bildintensität bestimmt werden kann. [42]

Zur Durchführung wurden von mir zwei 3x3 Sobel-Filter mit den Höhendaten des Terrains, die als Textur in einer dreidimensionalen Höhenprofilkarte (*Height Map*) kodiert wurden, gefaltet. Die Höhenprofilkarte ist eine *Skalar Map*, ein Beispiel ist in Abbildung 27 abgebildet. Ein Ausschnitt des OpenCL-*Kernel*, in dem die Berechnung (Zeilen 18 und 19) stattfindet, ist in Listing 4 aufgeführt. Die berechneten Tangenten werden anschließend in den Zeilen 21 und 23 skaliert und so gedreht, dass sie alle in negative y-Richtung zeigen. An den Randstellen werden die Tangenten auf Null gesetzt.

```
1  // use Sobel-Filter to calculate gradient map from height data
2  //
3  // kernels:
4  //
5  // |+1  0  -1|           |+1  +2  +1|
6  // |+2  0  -2|    and    | 0   0   0|
7  // |+1  0  -1|           |-1  -2  -1|
8  //
9  float top       = height[(abs(id-rowlen))       % N];
```

```
10  float down      = height[(abs(id+rowlen))       % N];
11  float right     = height[(id+1)                 % N];
12  float left      = height[(abs(id-1))            % N];
13  float topleft   = height[(id-rowlen - 1)        % N];
14  float topright  = height[(abs(id - rowlen + 1)) % N];
15  float downright = height[(id + rowlen + 1)      % N];
16  float downleft  = height[(abs(id + rowlen - 1)) % N];
17
18  float dx = border ? 0.0 : topleft - topright + 2*left - 2*right + downleft - ↩
        downright;
19  float dz = border ? 0.0 : topleft + 2*top + topright - downleft - 2*down - ↩
        downright;
20
21  float sharpness = -0.01f;
22
23  float dy = sharpness * sqrt(dx*dx + dz*dz);
24
25  float4 tangent = (float4)(dx, dy, dz, 1.0f);
```

Listing 4: Sobel-Filter zur Tangentenberechnung

Mit Hilfe der Tangenten kann das Fließverhalten von angesammeltem Wasser berechnet werden. Die Länge der Tangente wird dabei als Faktor für die Menge an abfließendem Wasser benutzt. Das heißt, je steiler der Punkt in der Landschaft, desto mehr Wasser fließt in das niedrigere Feld. Des weiteren wird anhand der Richtung der Tangente das benachbarte Feld ausgewählt, in welchem der Wasserstand erhöht werden soll. Dies geschieht in der Moore-Nachbarschaft (Achter-Nachbarschaft). In Abbildung 28 ist der in Felder unterteilte Querschnitt eines Terrains abgebildet (links). Beispielhaft sind in einem Feld die Normale $\vec{n}$ und die dazu orthogonale Tangente $\vec{t}$ angezeichnet. Außerdem ist ein Problem dieses Algorithmus verdeutlicht. Das Wasser sammelt sich bei längerer Laufzeit übermäßig in (lokalen-) Minima an. Es benötigt also einen weiteren Algorithmus, um das Wasser von diesen Ansammlungen in Minima auf die Umgebung zu verteilen. In Abbildung 28 (rechts) ist außerdem ein Feld mit Moore-Nachbarschaft im dreidimensionalen beispielhaft mit entsprechender Tangente dargestellt.

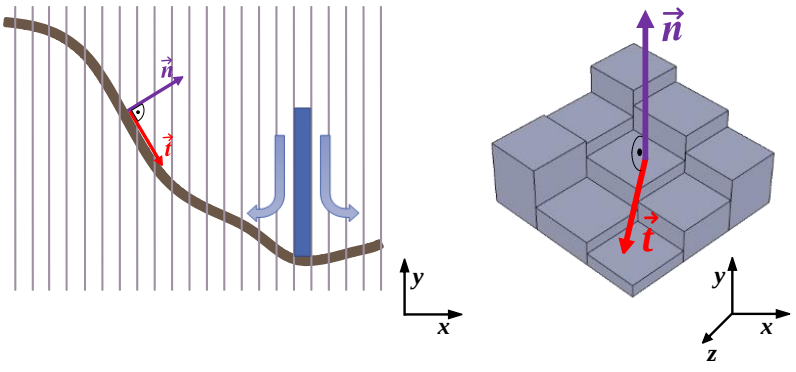

Abbildung 28: Wasser wechselt Felder anhand der Tangente. Es bilden sich Ansammlungen in Minima, die verteilt werden müssen.

4.2.5 Wasserverteilung

Die durch den tangentialen Fluss entstehenden Wasseransammlungen können an den lokalen Minima schnell sehr groß werden. Da aus den Höhenwerten der Wasserprofilkarte die Wasseroberfläche generiert werden soll, sollten für einen realistischen Eindruck die Maxima auf der Wasserprofilkarte auf die umliegenden Felder verteilt werden. Hierfür eignet sich eine abgewandelte Form des in Abschnitt 4.2.1 vorgestellten *Hight-Field-Fluids*-Algorithmus aus [16].

In einem *kernel* wird parallel für jedes Feld die Von-Neumann-Nachbarschaft (Vierer-Nachbarschaft) betrachtet. Zunächst wird ermittelt, welche dieser Nachbarn einen Höhenwert, addiert mit dem Wasserwert besitzen, der entweder niedriger oder in einer Epsilon-Umgebung um den eigenen Wert liegt. Nur diese Felder gehen in die Berechnung mit ein. Diese Abgrenzung dient dazu, dass sich Wasseransammlungen keine Anhebungen hinauf bewegen. Die Werte der Nachbarn werden addiert. Danach wird der eigene Wert des Feldes von dieser Summe so oft abgezogen, wie Nachbarn in die Berechnung eingeflossen sind. Das Ergebnis wird noch mit einer Konstanten c skaliert. Mit dieser lässt sich so die Ausbreitugsgeschwindigkeit beeinflussen. Das Resultat wird dann entsprechend der vergangenen Zeit seit der letzten Berechnung auf die Geschwindigkeit addiert. Abschließend wird der neue Wasserwert durch die Addition des alten mit der Geschwindigkeit berechnet. Algorithmus 1 zeigt den beschriebenen Algorithmus in Pseudocode.

Data: $ownValue$: Die Höhenwerte addiert mit den Wasserwerten
$neighborValue$: Die Höhenwerte addiert mit den Wasserwerten der Felder in der Von-Neumann-Nachbarschaft
$velocity$: Geschwindigkeitswert
dt: Die vergangene Zeit seit der letzten Berechnung
Result: neuer Wasserwert
foreach *field* **in parallel do**
  $sum \leftarrow 0$;
  $cnt \leftarrow 0$;
  foreach *neighbor* **do**
    if $neighborValue < ownValue + epsilon$ **then**
      $sum \leftarrow sum + neighborValue$;
      $cnt \leftarrow cnt + 1$;
    end
  end
  $f \leftarrow c * (sum - cnt * ownValue)$;
  $velocity \leftarrow velociy + f * dt$;
  $ownValue \leftarrow ownValue + velocity * dt$;
end

Algorithmus 1: Wasserverteilung

Aufgrund des Algorithmus bilden sich kleine Wellen in Ausbreitungsrichtung. Dieser Vorgang ist in Abbildung 29 zweidimensional dargestellt. Besitzt ein Feld einen hohen Wasserwert aufgrund des tangentialen Fließverhaltens, so gibt es diesen hohen Wert teilweise an

die Nachbarfelder ab. Dieser setzt sich von da an wieder an die entsprechenden Nachbarn fort und so weiter. Daraus resultiert eine Wellenerscheinung.

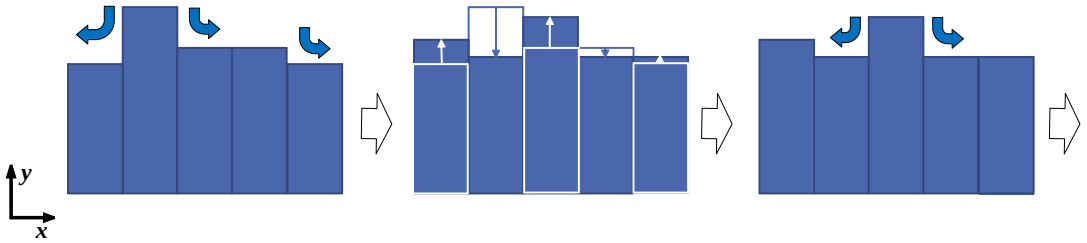

Abbildung 29: Zweidimensionale Darstellung der Wellenbildung durch den Wasserverteilungsalgorithmus.

Um die Wellen nicht zu groß werden zu lassen und die Wasseroberflächen im Allgemeinen realistischer darzustellen, findet abschließend noch eine Glättung der Werte mit Hilfe eines Gauß-Algorithmus statt. Dieser läuft ähnlich wie der Sobel-Algorithmus, der in Abschnitt 4.2.4 beschrieben wird. Statt des Sobel-Filters wird hier jedoch ein Gauß-Filter für die Faltung angewendet. Dieser besteht aus Werten, die nach der Gauß'schen Normalverteilung gewählt sind. Die Filterkernelgröße sowie die Werte können dabei variabel an die Laufzeit angepasst werden. Die Faltung wird auf die Werte der Wasserhöhenprofilkarte angewendet. Somit erhält man abschließend geglättete Wasseroberflächen. Einen Überblick über den Algorithmus zur Wasserbildung und -verteilung bietet Abbildung 30.

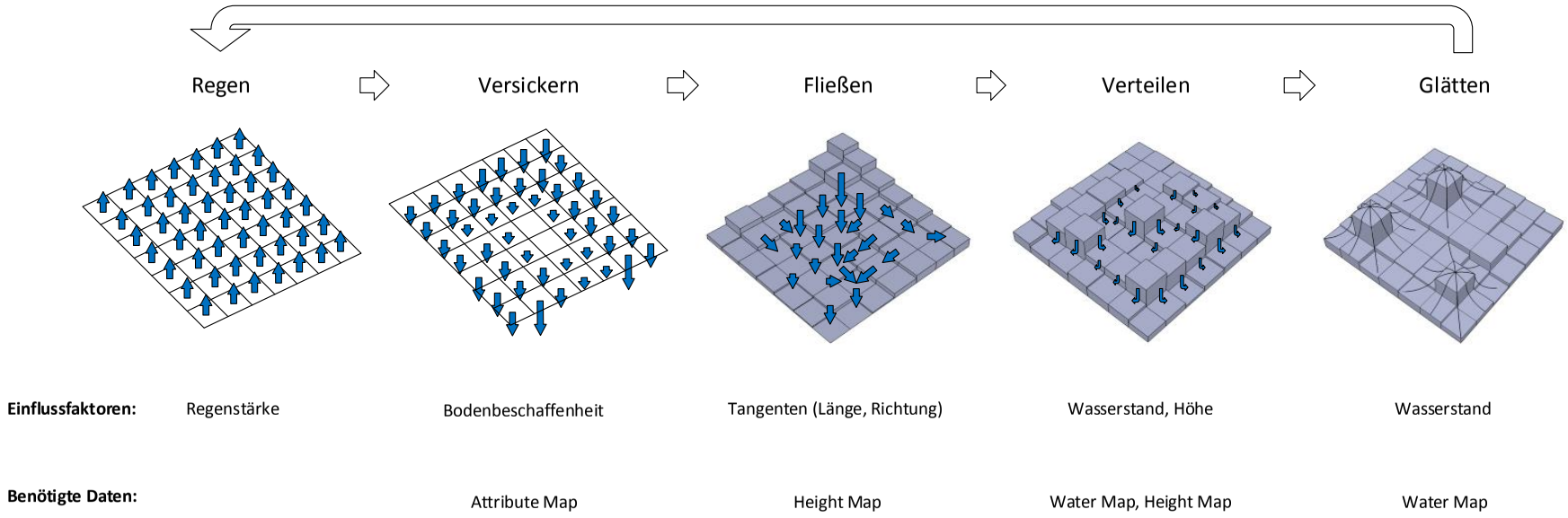

Abbildung 30: Übersicht über die Algorithmen zur Wasserbildung und dem Fließverhalten.

4.3 Darstellung der Wasseroberflächen

Zur Darstellung von Wasseroberflächen existieren unzählige Arbeiten und Implementationen. Die meisten umfassen dabei Reflexionen, Brechung, Transparenz und kleine Wellen (vgl. Abschnitt 3.2.2). Die Techniken basieren jedoch in der Regel auf der Annahme, dass nur eine oder wenige planare Wasserflächen existieren. Da die Wasseroberflächen in meinem Framework als komplexe Geometrie in Form eines GL_TRIANGLE_STRIP-Models aus der Wasserprofilkarte erstellt werden, müssen einige der bekannten Techniken abgeändert werden. Teilweise lassen sich diese gar nicht ohne sehr großen Aufwand beziehungsweise erhebliche Ressurcenanforderungen umsetzen. So wurde auf die Umsetzung der Lichtbrechung verzichtet. Diese ist ohne große optische Einbußen zu vernachlässigen, da in meinem Framework nur relativ flache Wasseransammlungen vorkommen, bei denen auch in natura keine große Verzerrungen durch die Brechung sichtbar wären. Konzentriert wurde sich statt dessen auf Reflexionen, Transparenz und die Darstellung von Kapillarwellen. Auf die Umsetzung wird im Folgenden näher eingegangen.

4.3.1 Reflexion und Transparenz

Ein typischer Ansatz für die Umsetzung von Reflexionen auf Wasseroberflächen ist die Szene in einem ersten Renderpass gespiegelt zu rendern. Hierbei wird die *view-Matrix* (vgl. Abschnitt 2.1.4) gedreht und eine *Clip Plane* erstellt, die der Wasseroberflächenebene entspricht. Es entsteht dann eine Textur, die die Szene oberhalb des Wasserspiegels gedreht zeigt. Diese kann man dann in einem zweiten Renderpass auf der Oberfläche abbilden und erhält somit die gewünschte Reflexion. [4]

Leider ist dieser Ansatz in meiner Anwendung nur schwer umzusetzen, da keine planare Wasseroberfläche existiert. Dies erweist sich vor allem beim Erstellen der *Clip Plane* als sehr hinderlich. Nun bestünde als Workaround die Möglichkeit, im *Fragment Shader* die *Fragments* unter der Wassergeometrie zu verwerfen, die Berechnung ist dabei jedoch relativ aufwendig. Deshalb entschloss ich mich, alternativ *Environment Mapping* umzusetzen. Dabei wird ausschließlich die Himmelgeometrie gespiegelt.

Da die Himmelsdarstellung, wie in Abschnitt 4.5 beschrieben, im Framework als *Sky Box* umgesetzt wurde, muss für den Erhalt der zu reflektierenden Pixel in der Kubustextur nachgeschaut werden. Ausschlaggebend hierfür ist der Reflexionsvektor an der Normalen der *Vertices* der Wassergeometrie. *GLSL* stellt hierfür die Funktion reflect(V, N) bereit, die aus dem *view*-Vektor $\vec{V}$ (der Vektor zwischen der Kameraposition und der Position des Vertex in Weltkoordinaten) und der Normale $\vec{N}$ an der Vertex-Position einen 3D-Vektor berechnet, mit dem der Texturzugriff auf die *Cube Map* stattfinden kann. In Abbildung 32 (links) sind die drei relevanten Vektoren im zweidimensionalen dargestellt. Ein Screenshot der Implementierung einer Oberfläche mit Totalreflexion mittels *Environment Mapping* zeigt Abbildung 31.

Wie in Abschnitt 3.2.2 beschrieben, hängt der Reflexionsfaktor respektive die Transparenz von Wasseroberflächen vom Betrachtungswinkel ab. Das Gesetz von *Fresnel* beschriebt dieses Phänomen: Je spitzer der Winkel relativ zur Wasseroberfläche, desto höher ist die

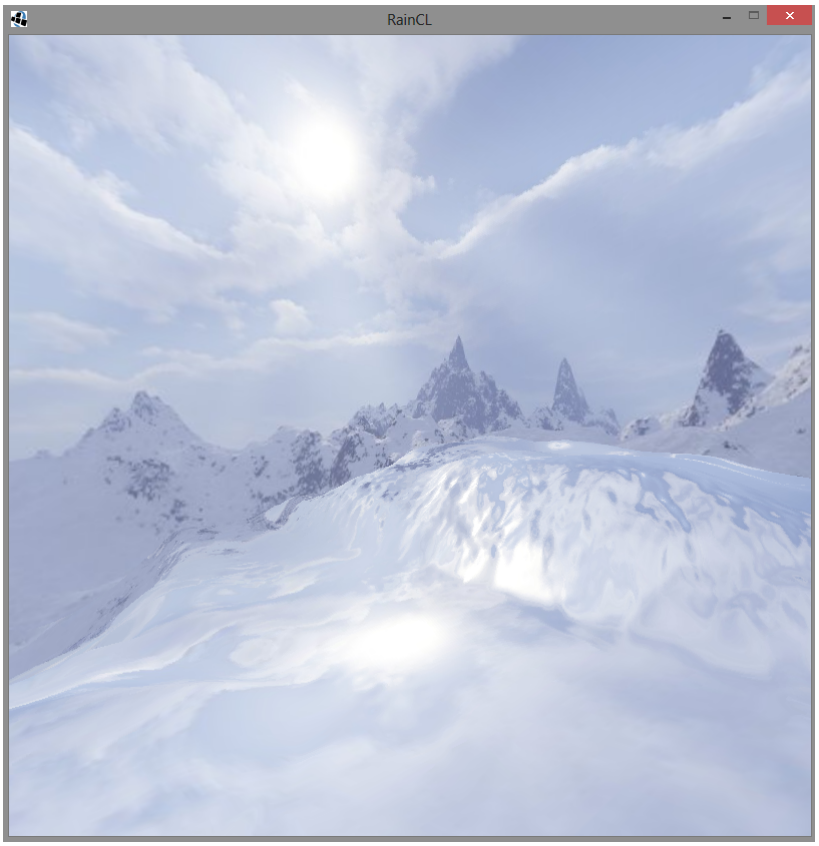

Abbildung 31: Screenshot: Reflexionen mit *Environment Mapping*.

Reflexion. Dieser Zusammenhang wurde mit Hilfe des Kreuzproduktes der normalisierten *view*- und Normalenvektoren umgesetzt. Abbildung 32 (rechts) zeigt eine schematische Darstellung des Zusammenhangs.

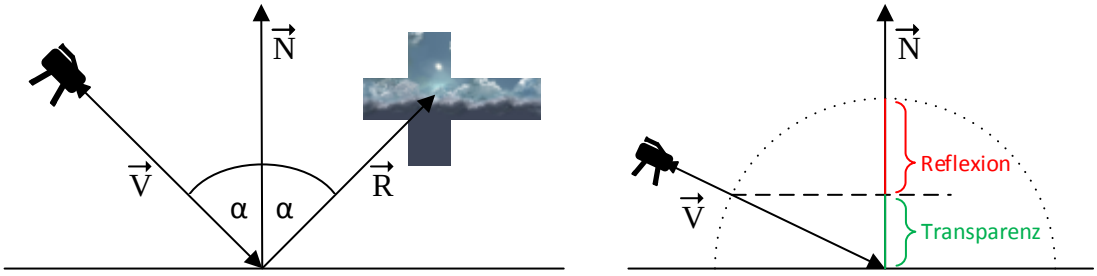

Abbildung 32: Schematische Darstellung der Reflexions- (links) und Transparenzberechnung (rechts), *Sky-Box*-Bilder von [22] .

4.3.2 Kapillarwellen mit *Bump Mapping*

Treffen Regentropfen auf eine Wasseroberfläche, so bilden sich in der Regel kleine Kapillarwellen auf dieser (vgl. Abschnitt 3.2.2). Auch dieses Phänomen verbessert den Realismusgrad und sollte deshalb in einem Programm mit Regendarstellung nicht fehlen. Für die Umsetzung kommt die *Post-Processing*-Technik *Bump Mapping* zum Einsatz.

Bump Mapping ist eine Technik, die es ermöglicht den Detailgrad von Objekten zu verbessern, ohne die Komplexität der zugrunde liegenden Geometrie zu erhöhen. Mit Hilfe zweier Texturen wird bei der Beleuchtung eine Oberflächenschattierung erzeugt, die Unebenheiten simuliert, die die Geometrie nicht bietet. Somit kann der Detaileindruck maßgeblich gesteigert werden, ohne die Renderzeit unnötig zu verlängern beziehungsweise die Hardware übermäßig zu beanspruchen.

Die Technik wurde in meiner Anwendung mittels zweier GL_TEXTURE_2D_ARRAY umgesetzt. In einem der Arrays sind 16 *Skalar-Maps*, sogenannte *Bump Maps* gespeichert, die sich durch ein in Grauwerten kodiertes Höhenprofil auszeichnen. Diese ähneln den in Abschnitt 4.2.4 beschriebenen *Height Maps*. Im zweiten Array sind gleichviele *Normal Maps* gespeichert. Dies sind *Vektor-Maps*, in denen ein Normalenvektor für jeden Pixel anhand der Farbwerte kodiert gespeichert ist. Die Rot-, Grün- und Blauwerte der Pixelfarben entsprechen dabei den x-, y- und z-Werten der Normalen. Beispiele der *Bump* und *Normal Maps* sind in Abbildung 33 dargestellt.

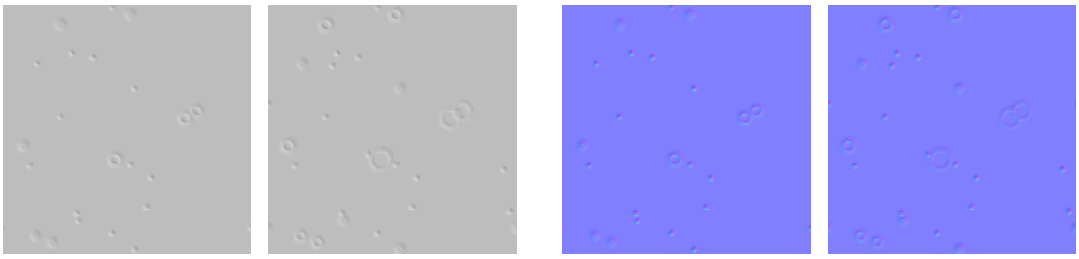

Abbildung 33: Beispiele für zwei konsekutive *Bump Maps* (links) und die zugehörigen *Normal Maps* (rechts).

Ein Ausschnitt der Umsetzung des *Bump Mappings* im *Fragment Shader* für die Wasserdarstellung ist in Listing 5 aufgeführt. Zunächst werden die Texturen in gleicher Form vielfach über die Terraingröße wiederholt abgebildet (Zeile 5). Dies geschieht durch entsprechende Skalierung der Texturkoordinaten und dem Einsatz von GL_REPEAT. So kann trotz relativ kleiner Texturen, wie in Abbildung 33 gezeigt, eine hohe Kapillarwellendichte generiert werden. Im nächsten Schritt wird dann die Normale aus der entsprechenden Textur geladen (Zeile 10). Dann wird die *Fragment*-Position in Zeile 11 anhand des Wertes aus der *Bump Map* entlang der geladenen Normalen verschoben. Bei der folgenden Beleuchtungsberechnung entsteht so der gewünschte plastische Effekt (Zeilen 13 - 16).

```
1  [...]
2  void main(void)
3  {
4    //repeat texture multiple times over terrain
5    vec2 texCoordsRepeat = texCoords * 16.0;
6    //calculate surface color (reflection and fresnel)
7    [...]
8
9    //calculate bump mapping
10   vec3 normal = normalize(texture2DArray(rainNormalTex, vec3(texCoordsRepeat, ↩
         circle)).xyz);
11   vec3 position = positionWC.xyz + (1.5 * texture2DArray(rainBumpTex, vec3(↩
         texCoordsRepeat, circle)).r * normal);
12
13   vec3 vPosLight = (position - lightPos);
14
15   fragColor.rgb = surfaceColor.rgb * max(dot(normalize(vPosLight), normal), 0);
16   fragColor.a = cubeCoords.q;
17 }
```

Listing 5: Fragment Shader zur Umsetzung der Bump-Mapping-Technik für Kapillarwellen

Für eine Animation der Kapillarwellen wird durch die 16 Texturen im Array gewechselt. Dabei hängt die Wechselgeschwindigkeit von der Regenstärke ab. Somit entsteht der Eindruck von mehr Wellen bei stärkerem Regen. Abbildung 34 zeigt Screenshots der finalen Wasserdarstellung im Framework.

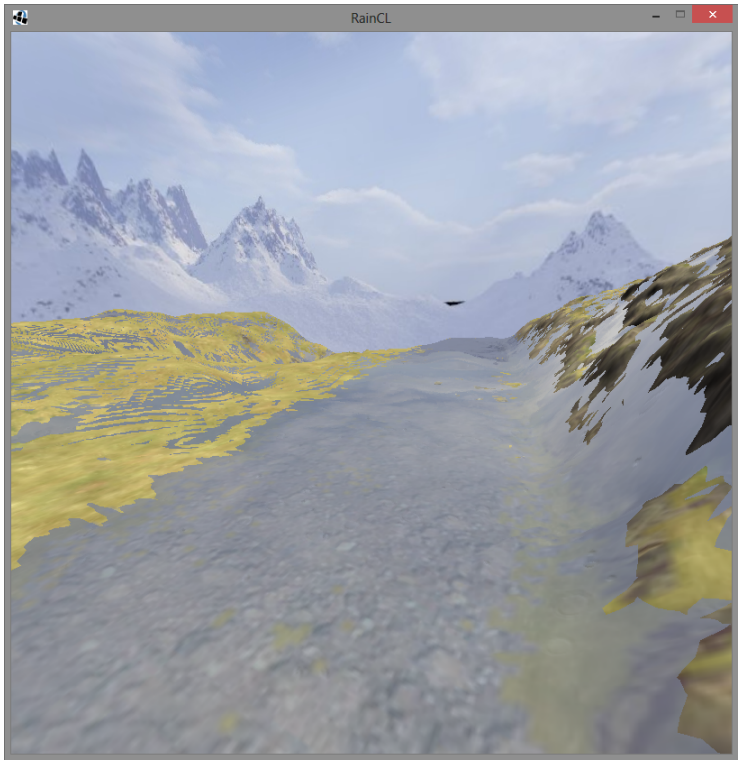

Abbildung 34: Screenshot der Wasserdarstellung im Framework

4.4 Nebel

Bei starkem Regenfall ist oftmals eine Nebel- beziehungsweise Dunstbildung zu beobachten. Auch dies sollte deshalb in dem Framework umgesetzt werden. Dafür werden zwei Techniken verwendet. Zunächst werden die *Fragment*-Farben der Terraingeometrie abhängig von der Kameraposition mit einem Grauwert gemischt. Dies geschieht im *Fragment Shader*, ein weiter entferntes Fragment bekommt einen höheren Grauanteil. Der Verlauf entspricht dabei einer Exponentialfunktion. Dadurch entsteht der Eindruck von einem Dunstschleier vor dem Terrain, je weiter der Betrachter von einem Punkt entfernt ist, desto undeutlicher wird dieser. In ähnlicher Weise wird auch die Himmelsgeometrie mit einem Grauton versehen. Die Funktion lässt sich theoretisch auf beliebig viele andere Geometrien ausweiten und somit einer ganzen Szene der Eindruck von Nebel verleihen.

Auch wenn die oben beschriebene Technik gute Ergebnisse liefert, fehlt es dem Nebel optisch an Tiefe. Es scheint nämlich, als würde dieser nur auf dem Terrain aufliegen, auch eine Bewegung von Dunstschwaden beispielsweise ist nicht möglich. Um einen volumetrischen Eindruck des Nebels zu schaffen, setze ich die in Abschnitt 4.1.3 beschriebene Billboard-Technik ein. Sehr große Billboards bewegen sich langsam hintereinander über den Bildschirm in einer Ebene, die parallel zur xz-Ebene verläuft. Auf die Billboards wird eine stark transparente Nebeltextur gelegt, die sich mit jedem Frame ändert und somit den Eindruck von sich bewegenden Dunstschwaden vermittelt. Die benötigten Texturen sind in einem GL_TEXTURE_2D_ARRAY abgelegt, um einen schnellen Zugriff zu ermöglichen (vgl. Abschnitt 4.1.4). Der Anwender besitzt die Möglichkeit die Nebelstärke zur Laufzeit anzupassen. Screenshots der Ergebnisse sind in Abbildung 35 abgebildet.

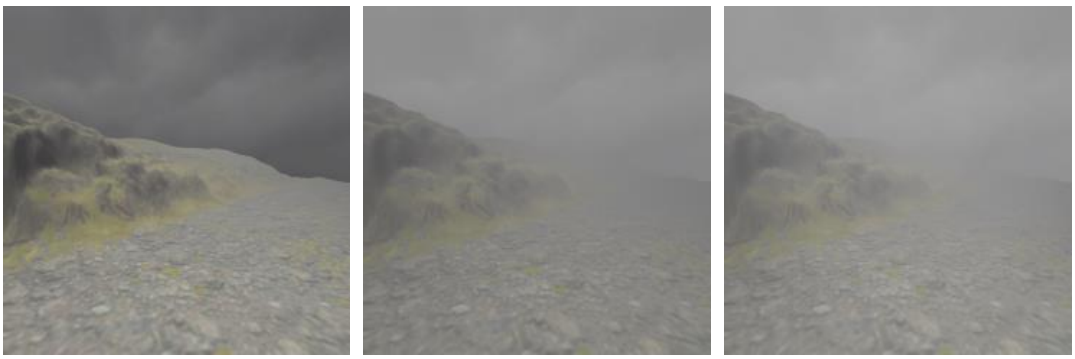

Abbildung 35: Nebeleffekt: schwacher Nebel (links), starker Nebel (Mitte), starker Nebel mit volumetrischem Nebeleffekt (rechts)

4.5 Terrain und Himmel

Bei einem Regensystem darf natürlich auch eine adäquate Umgebung nicht fehlen. Da dies in meiner Arbeit jedoch eher zweitrangig ist, entschloss ich mich dazu ein einfaches Freilandterrain sowie einen Himmel mit der, für Regen typischen, Bewölkung darzustellen.

Das Terrain wurde dabei mit Hilfe einer Höhenprofilkarte generiert. In jedem Pixel dieser sogenannten *Hight Map* ist ein Höhenwert für das Terrain gespeichert. Bei der Generierung der Geometrie werden diese Werte ausgelesen und mit Hilfe eines GL_TRIANGLE_STRIP zu einem Modell verbunden. Mit entsprechender Skalierung und einer Textur wurde die Terraingeometrie auf die gewünschten Bedürfnisse angepasst. Die Terraindaten wurden mit Hilfe des Tools *L3DT* [2] erstellt.

Der Himmel wurde mit einer *Sky Box* realisiert. Bei dieser Technik werden sechs zusammenhängende Texturen mit Hilfe von *Cube Mapping* an die Innenseiten eines Würfels projiziert. Die eigentliche Szene wird dann im Inneren des Würfels dargestellt, in dem sich auch der Betrachter befindet. Es entsteht somit die Illusion eines weit entfernten Himmels. Bewegt sich der Betrachter in der xz-Ebene, so wird die *Sky Box* mit bewegt, damit der Anwender nicht aus dem Himmel "hinausläuft" und der Eindruck eines konstant entfernten Horizontes entsteht.

4.6 Performance-Betrachtung

In diesem Abschnitt soll die Performance des Frameworks untersucht werden. Dies geschieht auf zwei Systemen, einem aktuellen von 2013 (*System 1*), welches das Referenzsystem ist, und einem älteren (*System 2*), mit Hardware aus den Jahren 2008 - 2010. Die relevanten Hardwarekomponenten der Systeme sind in Tabelle 4 aufgeführt.

Tabelle 4: Relevante Hardwarekomponenten der Testsysteme

	System 1 (2013)	
Grafikkarte	AMD Radeon HD 7950 @ 1 GHz	3GB GDDR5 RAM @ 1250 MHz
Prozessor	Intel Core i7-4770K @ 3,5 GHz	4 Cores / 8 Threads
Arbeitsspeicher	Corsair DDR3 @ 1600 MHz	16 GB
Solid State Drive	Samsung SSD 840 Pro	256 GB

	System 2 (2008 - 2010)	
Grafikkarte	NVIDIA GeForce 450 GTS @ 783 MHz	1 GB GDDR5 RAM @ 902 MHz
Prozessor	Intel Core2Duo E8400 @ 3,0 GHz	2 Cores / 2 Threads
Arbeitsspeicher	DDR3 @ 800 MHz	4 GB
Festplatte	7200 rpm	160 GB

Um die Performance zu messen wurden alle optischen Features aktiviert und ausschließlich die Regenstärke, also die Partikelanzahl der Regentropfen, variiert. Die Auflösung wurde

auf $800 * 800$ Pixel festgelegt und die Algorithmen zur Berechnung des Wasserverhaltens liefen konstant mit 262 144 Feldern. Die Regenpartikelanzahl wurde binär erhöht, verdoppelt sich also mit jeder Stufe. Der Startwert lag bei 2^{14} (16 384), der Endwert bei 2^{24} (etwa 16,8 Millionen) Partikeln. Die durchschnittlichen Frameraten beider Systeme bei der entsprechenden Partikelanzahl sind in Tabelle 5 aufgeführt. Zu beachten ist, dass eine *Vertikale Synchronisation* bei 60 Bildern pro Sekunde (FPS) stattfindet, dieser Wert also niemals überschritten wird.

Tabelle 5: Performancevergleich des Frameworks auf zwei Systemen

Partikelanzahl	*System 1* (2013) [FPS]	*System 2* (2008 - 2010) [FPS]
2^{14}	60	41-53
2^{15}	60	40-52
2^{16}	60	39-51
2^{17}	60	39-48
2^{18}	60	35-45
2^{19}	60	32-38
2^{20}	60	28-30
2^{21}	60	19-21
2^{22}	53	Absturz
2^{23}	32	Absturz
2^{24}	18	Absturz

Wie Tabelle 5 zu entnehmen ist, läuft das Framework mit einer annehmbaren Regen- und Umgebungsdarstellung selbst auf älteren Systemen problemlos. Dies ist höchstwahrscheinlich auf die weitgehende Parallelisierung der Berechnungen und deren Auslagerung auf die Grafikhardware zurückzuführen. Lediglich bei sehr hoher Partikelanzahl wird hier der Speicher zum limitierenden Faktor. Beim aktuellen System sind selbst noch acht Millionen Partikel in flüssiger Darstellung möglich, was optisch keinerlei Mehrwert zu ein oder zwei Millionen Partikeln bringt. Dies zeigt vor allem, dass noch viele Ressourcen vorhanden sind. Somit ist eine Umsetzung des Regensystems beziehungsweise Teilen davon in modernen 3D-Engines durchaus denk- und umsetzbar. Insbesondere als zuschaltbares Feature für Anwender mit potenter Hardware könnte so der Realitätsgrad gesteigert werden.

5 Fazit und Ausblick

Ziel der Arbeit war es, Algorithmen zur Darstellung und Simulation eines Regensystem in Echtzeit zu entwerfen. Dieses Ziel konnte erfolgreich umgesetzt werden. Die entwickelten Algorithmen ermöglichen die dreidimensionale Darstellung von Regen verschiedener Intensität, umfassen Wasseransammlungen und deren Fließverhalten auf dem Boden sowie die Darstellung der Wasseroberflächen. Meine entwickelte Anwendung fasst diese Algorithmen in einem Framework zusammen.

Für die Regendarstellung wurde auf moderne Techniken der Grafikprogrammierung zurückgegriffen. Mit Hilfe eines Partikelsystems und der *Billboard*-Technik wird eine realitätsnahe Umsetzung der Regentropfen erreicht. Durch eine entsprechende Anordnung und *Level-of-Detail*-Verteilung der Partikel, wird eine hohe Performance beim Einsatz moderner Grafikhardware garantiert. Anhand der Kamera- und Lichtposition werden Regentexturen aus einer Datenbank auf die *Billboard Sprites* abgebildet und somit eine Lichtinteraktion ermöglicht. Ein volumetrischer Nebeleffekt dient dazu, den Realismuseindruck zu steigern.

Mit Hilfe der OpenCL-Schnittstelle wurden parallele Algorithmen für die Wasserbildung und das Fließverhalten entworfen und umgesetzt. Als grundlegende Datenstruktur kommt dabei eine diskrete Höhenprofilkarte zum Einsatz, die den Wasserstand repräsentiert. Diese ermöglicht auch die Parallelisierung all dieser Algorithmen, die aufgrund von Performancevorteilen auf Grafikhardware ausgeführt werden. Nach Änderung des Wasserstandes anhand der Regenstärke und der Bodenbeschaffenheit, fließt das Wasser in niedrigere Bereiche. Die Tangenten des Terrains dienen hierbei als Richtungs- und Stärkeindikator. Anschließend wird das angesammelte Wasser in angrenzende Bereiche verteilt und wirkt somit der Bildung von unrealistischen Wassermaxima entgegen. Abschließend findet eine Glättung der Wasserwerte mit Hilfe eines Gauß-Filters statt.

Für die Darstellung der Wasseroberflächen kommen klassische Techniken zum Einsatz. Mit *Environment Mapping* wird der Himmel auf den Oberflächen gespiegelt. Der Betrachterwinkel wird nach dem Vorbild der Natur zur Berechnung der Transparenzerscheinung des Wassers verwendet. Außerdem werden kleine Kapillarwellen, die beim Auftreffen von Regentropfen auf eine Wasseroberfläche entstehen, mit Hilfe von *Bump Mapping* umgesetzt.

Verbesserungen und Erweiterungen sind, neben einer Optimierung der Fließalgorithmen, insbesondere im Bereich der Wasserdarstellung denkbar. Hier wäre eine umfassende Spiegelung der gesamten Umgebung inklusive der Geometrien wünschenswert. Außerdem wirken die Übergänge zwischen Wasser und dem Boden sehr künstlich. Auch bei den Fließanimationen an steilen Oberflächen besteht noch die Möglichkeit optischer Verbesserungen. *Volume Ray Casting* als alternativer Ansatz zur Wasserdarstellung ist für künftige Arbeiten zu der Thematik sicherlich eine genauere Betrachtung wert.

Abschließend kann gesagt werden, dass sich die entwickelten Techniken und Algorithmen grundlegend für die Darstellung eines Regensystems in Echtzeit 3D-Anwendungen eignen.

Sie können als Erweiterung moderner Engines in Betracht gezogen werden, um den Eindruck von realistischen Freilandszenen zu verbessern.

Abbildung 36: Screenshot des finalen Frameworks mit Regen, Nebel und Wasser

6 Literatur

[1] AMD: *AMD Radeon Grafikkarten für Desktop PCs*. http://www.amd.com/de/products/desktop/graphics/Pages/desktop-graphics.aspx. – zuletzt aufgerufen am 01.08.2013

[2] Bundysoft: *L3DT*. http://www.bundysoft.com/L3DT/. – zuletzt aufgerufen am 17.08.2013

[3] Creative Technology: *OpenAL*. http://www.openal.org/. – zuletzt aufgerufen am 31.07.2013

[4] Forno, Reto D.: Realtime Water-Rendering. (2006). http://keepcoding.ch/kcdev/tutorials/Realtime%20Water%20Rendering.pdf. – zuletzt aufgerufen am 17.08.2013

[5] Fraser, Alistair B.: *Bad Rain*. http://www.ems.psu.edu/~fraser/Bad/BadRain.html. – zuletzt aufgerufen am 02.08.2013

[6] Garg, Kshitiz ; Nayar, Shree K.: Photorealistic Rendering of Rain Streaks. (2006). http://www1.cs.columbia.edu/CAVE/projects/rain_ren/rain_ren.php

[7] Heidorn, Keith C.: *Weather People and History: Philipp Lenard*. http://www.islandnet.com/~see/weather/history/lenard.htm. – zuletzt aufgerufen am 03.08.2013

[8] Intel: *Intel Product Information*. http://ark.intel.com/. – zuletzt aufgerufen am 01.08.2013

[9] Khronos Group: *Khronos Group*. http://www.khronos.org/. – zuletzt aufgerufen am 31.07.2013

[10] Khronos Group: *OpenCL*. http://www.khronos.org/opencl/. – zuletzt aufgerufen am 31.07.2013

[11] Khronos Group: *OpenGL*. http://www.opengl.org/about/. – zuletzt aufgerufen am 31.07.2013

[12] Khronos Group: *Rendering Pipeline Overview*. http://www.opengl.org/wiki/Rendering_Pipeline_Overview. – zuletzt aufgerufen am 31.07.2013

[13] Lull, H.W.: *Soil Compaction on Forest and Range Lands*. Forest Service, U.S. Department of Agriculture, 1959 (Miscellaneous publication (United States. Dept. of Agriculture)). http://books.google.de/books?id=OSsuAAAAYAAJ

[14] lwjgl.org: *Light Weight Java Game Library*. http://lwjgl.org/. – zuletzt aufgerufen am 31.07.2013

[15] Macklin, Miles ; Müller, Matthias: Position Based Fluids. (2013). http://www.matthiasmueller.info/publications/pbf_sig_preprint.pdf

[16] Müller, Matthias: Fast Water Simulation for Games Using Height Fields. (2008). http://gdcvault.com/play/203/Fast-Water-Simulation-for-Games

[17] NVIDIA: *GeForce-Grafikprozessoren*. http://www.nvidia.de/object/geforce_family_de.html. – zuletzt aufgerufen am 01.08.2013

[18] Puig-Centelles, Anna ; Ripolles, Oscar ; Chover, Miguel: Creation and control of rain in virtual environments. In: *Visual Computing*, Springer, 2009, S. 1037–1052

[19] Riccio, Christophe: *g-trunc*. http://www.g-truc.net/. – zuletzt aufgerufen am 01.08.2013

[20] Rincon, Paul: Monster raindrops delight experts. (2004). http://news.bbc.co.uk/2/hi/science/nature/3898305.stm. – zuletzt aufgerufen am 02.08.2013

[21] Rousseau, Pierre ; Jolivet, Vincent ; Ghazanfarpour, Djamchid: Realistic real-time rain rendering. In: *Computers and Graphics 30*, 2006, S. 507–518

[22] Skoglund, Jockum: *Sky Texture*. http://www.zfight.com/misc/images/textures/envmaps/miramar_large.jpg. – zuletzt aufgerufen am 17.08.2013

[23] Slick2D: *Slick-Util*. http://slick.ninjacave.com/slick-util/. – zuletzt aufgerufen am 31.07.2013

[24] Starik, Sonia ; Werman, Michael: Simulation of Rain in Videos. (2003)

[25] Tariq, Sarah: Rain. (2007). http://developer.download.nvidia.com/SDK/10/direct3d/Source/rain/doc/RainSDKWhitePaper.pdf

[26] Tatarchuk, Natalya: ATI: ToyShop. (2006). http://developer.amd.com/wordpress/media/2012/10/ToyShop-Eurographics_AnimationFestival.pdf

[27] Thürey, Nils ; Müller-Fischer, Matthias ; Schirm, Simon ; Gross, Markus: Real-time Breaking Waves for Shallow Water Simulations. In: *PG '07 Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, IEEE Computer Society, 2007, 39-46

[28] Torus Knot Software Ltd.: *OGRE 3D Demo*. http://www.ogre3d.org/download/demos. – zuletzt aufgerufen am 06.08.2013

[29] Wang, Lifeng ; Lin, Zhouchen ; Fang, Tian ; Yang, Xu ; Yu, Xuan ; Kang, Sing B.: Real-Time Rendering of Realistic Rain. (2006). http://www.research.microsoft.com

[30] Wang, Niniane ; Wade, Bretton: Rendering Falling Rain and Snow. (2004). http://www.research.microsoft.com

[31] Wenke, Henning: *Folien zur Vorlesung "Computergrafik" an der Universität Osnabrück*. 2012

[32] Wenke, Henning: *Folien zur Vorlesung "Parallele Algorithmen mit OpenCL" an der Universität Osnabrück*. 2013

[33] WETTERDIEST.DE: *Klima - Station Münster/Osnabrück Flughafen*. http://www.wetterdienst.de/Deutschlandwetter/Osnabrueck/Klima/. – zuletzt aufgerufen am 03.08.2013

[34] WIKIPEDIA FOUNDATION INC.: *Comparison of Intel graphics processing units*. http://en.wikipedia.org/wiki/Comparison_of_Intel_graphics_processing_units. – zuletzt aufgerufen am 04.08.2013

[35] WIKIPEDIA FOUNDATION INC.: *Eigenschaften des Wassers - Optische Eigenschaften*. http://de.wikipedia.org/wiki/Eigenschaften_des_Wassers#Optische_Eigenschaften. – zuletzt aufgerufen am 03.08.2013

[36] WIKIPEDIA FOUNDATION INC.: *Geforce*. http://en.wikipedia.org/wiki/Geforce. – zuletzt aufgerufen am 04.08.2013

[37] WIKIPEDIA FOUNDATION INC.: *Infiltration (Hydrologie)*. https://de.wikipedia.org/wiki/Infiltration_(Hydrogeologie). – zuletzt aufgerufen am 03.08.2013

[38] WIKIPEDIA FOUNDATION INC.: *Procedural Modeling*. https://en.wikipedia.org/wiki/Procedural_modeling. – zuletzt aufgerufen am 06.08.2013

[39] WIKIPEDIA FOUNDATION INC.: *Radeon*. http://en.wikipedia.org/wiki/Radeon#Processor_generations. – zuletzt aufgerufen am 04.08.2013

[40] WIKIPEDIA FOUNDATION INC.: *Rain*. http://en.wikipedia.org/wiki/Rain. – zuletzt aufgerufen am 02.08.2013

[41] WIKIPEDIA FOUNDATION INC.: *Regen*. http://de.wikipedia.org/wiki/Regen. – zuletzt aufgerufen am 03.08.2013

[42] WIKIPEDIA FOUNDATION INC.: *Sobel Operator*. https://en.wikipedia.org/wiki/Sobel_operator. – zuletzt aufgerufen am 07.08.2013

[43] WIKIPEDIA FOUNDATION INC.: *Surface runoff*. http://en.wikipedia.org/wiki/Surface_runoff. – zuletzt aufgerufen am 03.08.2013

A Anhang

A.1 Glossar und Abkürzungsverzeichnis

3D-Echtzeitanwendung	Anwendung für die Darstellung einer 3D-Szene, die scheinbar flüssig abläuft.
3D-Engine	System für die Erstellung und Entwicklung von Videospielen.
Aliasing	Treppchenbildung: durch die diskrete Darstellung mit Pixeln auf Bildschirmen kann bei geraden Kanten eine Treppchenbildung auftreten.
API	Application Programming Interface: Programmierschnittstelle.
Blending	Fester Schritt in der ↑ *Rendering Pipeline*. ↑ *Fragments* werden nach bestimmten Kriterien (tansparent) überblendet.
Clipping	Fester Schritt in der ↑ *Rendering Pipeline*. ↑ *Primitive* werden außerhalb des ↑ *View Volume* bzw. der ↑ *Clip Plane* abgeschnitten.
Clip Plane	Benutzerdefinierte, zweidimensionale Ebene, an der ↑ *Clipping* stattfindet.
CPU	Central Processing Unit: Hauptprozessor.
Cube Map	Sechsseitige Textur die auf einen Würfel abgebildet werden kann. Wird u. a. zur Himmelsdarstellung verwendet (↑ *Sky Box*).
Culling	Fester Schritt in der ↑ *Rendering Pipeline*. Rückseiten von Dreiecken werden nicht angezeigt.
Context (OpenCL)	Ein Container, der alle zusammengehörenden OpenCL-Objekte kapselt. Ist das zentrale Element einer OpenCL Anwendung.
Device	Teil der OpenCL Architektur. Führt ↑ *Kernel* aus. Es sind verschiedene GPU-Varianten als Device möglich. Es können mehrere Devices in einem ↑ *Context* existieren.
DirectX	Von Microsoft entwickelte API zur Multimediaprogrammierung. Funktionsumfang ähnelt dem von OpenGL.
FPS	Frames Per Second: Bilder pro Sekunde.

Fragment	Die notwendigen Daten für das Zeichnen eines Pixels.
Fragment Shader	↑ *Shader* zur Manipulation von ↑ *Fragments*.
Geometry Shader	↑ *Shader* um ↑ *Primitives* zu manipulieren und zu erzeugen.
GPU	Graphics Processing Unit: Grafikprozessor.
GLSL	OpenGL Shading Language: Eine auf C basierende Sprache zur Shader-Programmierung.
HLSL	High Level Shading Language: Eine von Microsoft entwickelte Sprache zur Shader-Programmierung unter ↑ *DirectX*.
Host	Ist Teil der OpenCL Architektur und immer genau eine ↑ *CPU*. Ist zuständig für die Koordination der Komponenten.
Kernel	↑ *OpenCL C* Programm, das parallel auf dem ↑ *Device* ausgeführt wird.
Level of Detail (LOD)	Oberbegriff für Techniken, bei denen versucht wird durch den Einsatz verschiedener Detailstufen die Performance zu steigern. Bspw. weniger Details bei größerer Entfernung.
OpenCL C	Sprache zur Programmierung von ↑ *kernels*. Angelehnt an ISO C99, enthält Erweiterungen für parallele Programmierung.
Primitive(s)	Die einfachsten geometrischen Objekte, die von OpenGL verwaltet werden können.
Open World Engine	↑ 3*D-Engine*, die auf die Darstellung von offenen Welten optimiert ist.
Rasterization	Fester Schritt in der ↑ *Rendering Pipeline*. ↑ *Primitives* werden in ↑ *Fragments* gerastert.
Rendering	Die Konvertierung von ↑ *3D-Wire-Frame-Modellen* in ein 2D-Bild mit Fotorealistischen Effekten mittels Hard- und/oder Software.
Rendering Pipeline	Sequentielle Abfolge von Schritten um eine 2D-Rasterrepräsentation einer 3D-Szene zu erstellen.

Skalar Map	Eine ↑ *Textur*, die in jedem Pixel einen Skalar kodiert. In der Regel ist dies ein Grauwert oder es wird nur der rote Farbkanal genutzt.
Sky Box	Möglichkeit der Himmelsdarstellung als Texturen auf der Innenseite eines Würfels unter Einsatz von ↑ *Cube Mapping*.
Sprite	Ein Grafikobjekt, das vor den restlichen Geometrien eingeblendet wird. Wird i.d.R. mit einer ↑ *Textur* versehen und ggf. geblendet.
Tesselation	Parkettierung: Wird benutzt um detailarme Oberflächen in detailreiche ↑ *Primitives* zu konvertieren.
Tesselation Shader	↑ *Shader* um ↑ *Tesselation* umzusetzen.
Transform Feedback	Optionaler Schritt in der ↑ *Rendering Pipeline* um Daten nach dem ↑ *Geometry Shader* abzugreifen und ggf. die Pipeline von neuem durchlaufen zu lassen.
Vector Map	Eine ↑ *Textur*, die in jedem Pixel einen Vektor kodiert. Dabei entsprechen die Farbwerte (Rot, Grün und Blau) den Vektorkomponenten (x, y und z).
Vertex	Datenstruktur, die einen Punkt im 2D- oder 3D-Raum repräsentiert. Enthält oft zusätzliche Informationen abseits der Ortskoordinaten.
Vertex Shader	↑ *Shader* zur Manipulation von ↑ *Vertices*.
Vertikale Sync.	Vertikale Synchronisation: Technik, die eine Aktualisierung der Bilddaten verhindert während der Bildschirm das Bild aufbaut.
View Volume	Sichtbares Volumen in der Rastergrafik, in welchem eine 3D-Szene ↑ *gerendert* wird.
Wire Frame Model	Repräsentation eines 3D-Objektes, dass aus Knoten und Kanten besteht. Als grundlegendes Polygon wird in der Regel ein Dreieck verwendet.

A.2 Steuerungsanleitung zum Framework *RainCL*

Das *RainCL* genannte Framework findet sich inklusive Quellcode auf der beiliegenden CD-ROM. Zum Starten muss die Datei RainCL.jar ausgeführt werden, eine aktuelle *Java Runtime Environment* ist dafür erforderlich. Der aktuelle Quellcode kann außerdem von meiner GitHub Projektseite heruntergeladen werden:

`https://github.com/theVall/RainCL`

Nach Starten des Programms, kann der Benutzer den Betrachterstandpunkt mit Hilfe der WASD-Tasten, analog zu einem First-Person-Shooter, verändern. Mit der Mausposition wird dabei die Richtung angegeben. Die Leer- und C-Taste dienen dazu, die Kameraposition zu erhöhen beziehungsweise zu erniedrigen. Mit gedrückter Shift-Taste bei der Steuerung erhöht man die Geschwindigkeit der Kamerapositionsveränderung.

Mit der M-Taste kann zur Laufzeit ein grafisches Menü aufgerufen werden. In diesem werden oben die Anzahl der Regenpartikel sowie die Bilder pro Sekunde (FPS) angezeigt. Im Reiter *Environment* besteht für den Anwender die Möglichkeit, mit Checkboxen und Schiebereglern die Umgebung direkt zu beeinflussen und verschiedene Effekte aus- oder einzublenden. Unter *System Info* werden Informationen zur verwendeten Grafikhardware, Treiberversion und unterstützten Standards gegeben. Einen Screenshot des Menüs zeigt Abbildung 37.

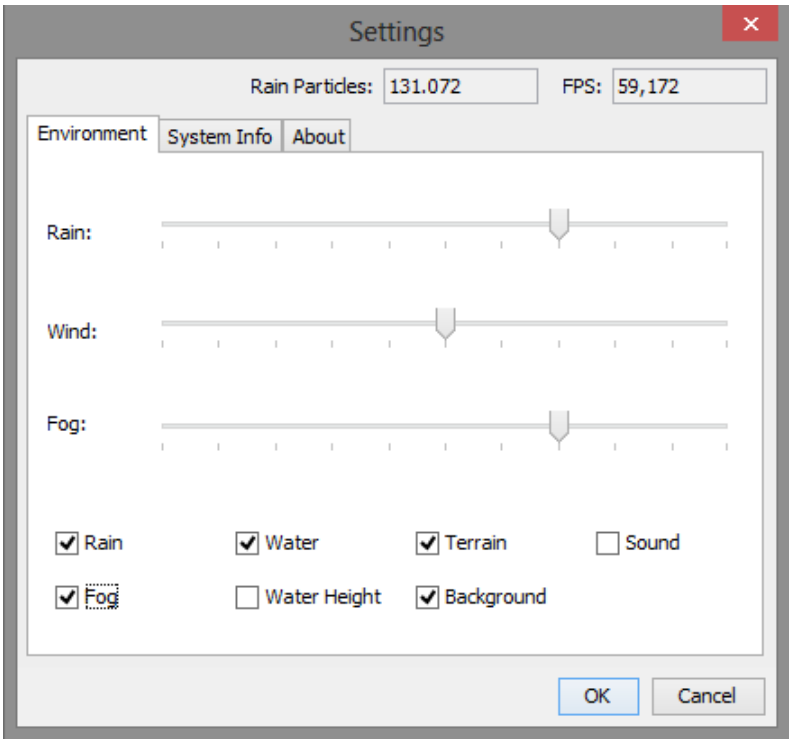

Abbildung 37: Screenshot des grafischen Menüs; Aufruf mit der M-Taste

A.3 Abstract (English)

Dynamic Realtime 3D-Rendering of a Rain System using Parallel Algorithms

Rendering weather phenomena in realtime 3D-applications is an important stylistic device. In this Bachelor thesis a framework is presented which covers rendering and simulation of a rain system. Visualisation of falling rain streaks is a part as well as a realistic simulation of water puddles that form due to the rainfall. Convincing rendering of water surfaces also plays a central role. In addition, the user shall be able to manipulate the environment in form of rain and wind intensity and therefore visualize the impact on the system. The focus of the presented work lies in realtime capability. To ensure that, parallel algorithms that run on graphics hardware are used with the help of the OpenCL API. My approach with the used techniques, interfaces and algorithms as well as the physical background is presented in this thesis.

A.4 Erklärung zur selbstständigen Abfassung der Bachelorarbeit

Ich versichere, dass ich die eingereichte Bachelorarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als den von mir angegebenen Hilfsmitteln und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen, habe ich kenntlich gemacht.

.....
Ort, Datum, Unterschrift