GPU-basierte Simulation von Schneedeformationen mittels Shader Model 5

Thomas Berger

DIPLOMARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2011

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 24. September 2011

Thomas Berger

Inhaltsverzeichnis

Kurzfassung

Schnee in Echtzeit-Anwendungen ist ein relativ vernachlässigtes Thema. Gerade in Spielen wird meist nur über Texturen die Optik von Schnee simuliert, obwohl eine entsprechende Simulation das Spielerlebnis enorm bereichern würde. In dieser Arbeit wird eine Methode präsentiert, wie dank der Rechenleistung moderner GPUs, Objekt–Schnee Interaktionen simuliert werden können, bei denen Schnee einerseits komprimiert und andererseits verteilt wird. Dabei wird intensiv auf Shader Model 5 Funktionalitäten wie Compute Shader und Tessellierung zurückgegriffen.

Die Simulation ist stark abstrahiert und arbeitet hauptsächlich mit Höhenwerten (Height Maps). Zuerst wird durch Finden von Überschneidungen zwischen Schnee und Objekt die verdrängte Schneemenge ermittelt. Diese wird anschließend teilweise komprimiert und zum Teil entlang der ermittelter Kontur rund um das Objekt verteilt. Die Verteilung ist ungleich und hängt hauptsächlich von der Bewegungsrichtung des Objektes ab. Am Schluss wird noch sichergestellt, dass keine zu steilen Hügel in der Szene sind indem diese notfalls abgeflacht werden.

Die Simulation wird zu Beginn theoretisch vorgestellt. Anschließend wird die praktische Umsetzung etwas genauer beschrieben, inklusive aufgetretener Probleme und Herausforderungen. Dabei wird nicht nur auf die simulationsspezifischen Eigenheiten eingegangen, sondern auch auf die allgemeine Funktionsweise von Compute Shadern und Tessellierung, wodurch die Stärken und Möglichkeiten des Shader Models 5 aufgezeigt werden sollen.

Abstract

Snow in real-time applications is a fairly neglected topic. In many games, only textures are used to simulate the optics of snow, even though a snow simulation would enrich the gaming experience tremendously. In this thesis, a method is introduced to simulate snow—object interactions using the power of modern GPUs, resulting in deformed and displaced snow. This method will be using Shader Model 5.0 functionality such as Compute Shader and Tessellation extensively.

The very abstract simulation mainly uses height information (height maps) as input data. At first, the displaced amount of snow is determined using overlaps between snow and objects. Afterwards, this displaced snow is partly compressed and partly distributed along a calculated contour around the object. The distribution is uneven and depends mainly on the moving direction of the object. Finally, the simulation checks the scene for too steep hills and flattens them if necessary.

The theory behind the simulation is introduced at the beginning of this thesis. Afterwards, the practical realization is shown in detail, including the problems and challenges which occurred during the implementation. Not only simulation-specific details but general usage of Compute Shader and tessellation are described, which should show the strengths and possibilities of the Shader Model 5.0.

Kapitel 1

Einleitung

1.1 Problemstellung

Realismus ist einer der wichtigsten Faktoren in der Spieleentwicklung. Das zeigt sich besonders gut in den Bereichen Grafik und Physik, in denen regelmäßig neue Engines auf den Markt kommen, die ihre Vorgänger in Sachen Leistung und Qualität bei Weitem übertreffen. So ist es auch nicht sonderlich überraschend, dass viele Spieleentwickler die Grafik als Hauptfeature ihres Spieles anpreisen [27]. Daher ist es verwunderlich, dass es in den letzten Jahren im Bereich Schneesimulation keine nennenswerten Verbesserungen gegeben hat. Es haben zwar diverse Entwickler versucht, die grafische Darstellung etwas realistischer zu gestalten (Abbildung 1.1), dennoch beschränken sie sich aber meist auf einfache Texturen, bei denen eventuell noch Fußabdrücke zu sehen sind, welche dann aber nach einigen Sekunden wieder verschwinden.

Eine physikalische Auswirkung des Schnees gibt es kaum in einem Spiel, wobei sich das bei einigen durchaus anbieten würde. Eine Beeinflussung des Balles bei einem Fußball-Videospiel würde beispielsweise zu einem komplett neuen Spielgefühl führen, welches dem Spieler mehr Abwechslung und Rea-

Abbildung 1.1: Vergleich der Schneedarstellung zwischen Madden NFL 08 und Madden NFL 09. Quelle: [11]

lismus bietet.

1.2 Motivation

Für die Simulation von Schnee gibt es mehrere Ansätze, die aber meist komplett auf der CPU ablaufen. Manchmal finden die Berechnungen parallel auf mehreren Systemen statt, um einen entsprechenden Performance-Gewinn zu erzielen (genaueres folgt in Kapitel 2). Diese parallelen Abläufe bieten sich besonders für moderne GPUs an, da vor allem in den letzten Jahren GPGPU[1] verstärkt eingesetzt wird, was vor allem von der hohen Parallelisierbarkeit der Grafikkarten Prozessoren lebt. Diese Entwicklung ist soweit gegangen, dass mit der Einführung von DirectX 11 und damit verbunden dem Shader Model 5 eine eigene Shader-Kategorie für allgemeine Berechnungen auf der GPU hinzugekommen ist – Compute Shader. Da diese die enorme Rechenpower von GPUs mit paralleler Ausführbarkeit kombinieren, bietet es sich an, die Schneesimulation auf die Grafikkarte auszulagern.

1.3 Zielsetzung und Aufbau der Arbeit

Ziel dieser Arbeit ist es, eine Möglichkeit zur Simulation von Schneedeformationen auf der GPU zu zeigen. Dabei sollen ein oder mehrere Objekte mit dem Schnee interagieren können. Wenn diese mit dem Schnee in Verbindung kommen, sollen die Auswirkungen (Abdruck im Schnee, Schneeverdrängung, Verlangsamung des Objektes, etc.) erkennbar sein. Abbildung 1.2 zeigt ein Beispielszenario der Simulation. Bei der Umsetzung wird nur in zweiter Linie auf Performance geachtet, Hauptaugenmerk liegt in der grundsätzlichen Umsetzung, wobei exzessiv auf die Shader Model 5 Funktionalität (Compute Shader und Tessellierung) zurückgegriffen wird.

Nach dem Einführungskapitel, in dem sich auch eine rudimentäre Modellbeschreibung befindet, werden themenverwandte Arbeiten vorgestellt. Anschließend werden das verwendete Modell und die damit verbundene Methode zur Schneesimulation (unabhängig jeglicher Implementierungsdetails) theoretisch vorgestellt. Im darauf folgenden Kapitel wird dann gezeigt, wie diese Methode auf der GPU umgesetzt wurde, inklusive aufgetauchter Probleme und entsprechender Lösungsansätze. Zu guter Letzt werden noch die Ergebnisse präsentiert und es wird ein Ausblick gegeben, wie die Simulation verbessert werden kann.

[1] General Purpose Computation on Graphics Processing Unit (kurz GPGPU, vom Englischen für Allzweck-Berechnung auf Grafikprozessoreinheit(en)) bezeichnet die Verwendung eines Grafikprozessors für Berechnungen über seinen ursprünglichen Aufgabenbereich hinaus. Bei parallelen Algorithmen kann so eine enorme Geschwindigkeitssteigerung im Vergleich zum Hauptprozessor erzielt werden. Quelle: [12]

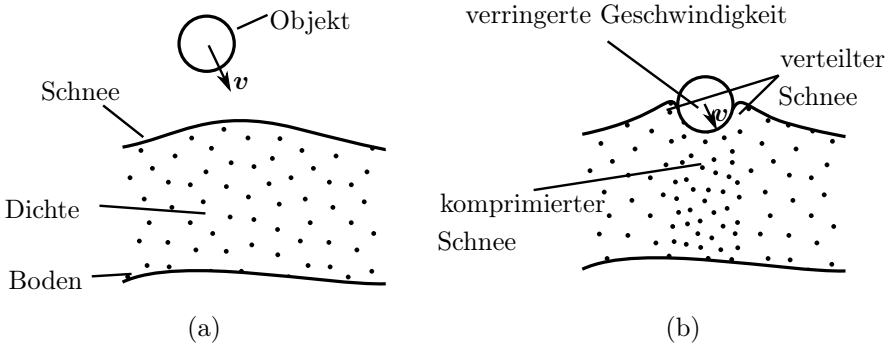


Abbildung 1.2: Beispiel der Simulation. In der Szene existiert ein Objekt, dass sich mit Geschwindigkeit $\boldsymbol{v}$ bewegt (a). Kommt es zu einer Kollision mit dem Schnee, wird die Geschwindigkeit des Objektes verringert und der verdrängte Schnee wird komprimiert (höhere Dichte ρ) bzw. verteilt (Hügelbildung) (b).

1.4 Modell

Das verwendete Modell spiegelt keine korrekte physikalische Simulation dar und soll lediglich ein grafisch und physikalisch „akzeptables“ Ergebnis liefern, wodurch die Berechnung und die verwendeten Werte stark vereinfacht sind.

1.4.1 Physikalisch korrekte Simulation

Bei einer korrekten Simulation von Schnee gibt es sehr viele Faktoren zu berücksichtigen. Die Komplexität der Materie Schnee zeigt sich bereits in der Unzahl verschiedener Themenbereiche, die in der Literatur bereits behandelt worden sind. Hierbei gibt es beispielsweise Werke, die sich lediglich mit Schneefall beschäftigen. Andere wiederum setzen sich mit der grundsätzlichen Mechanik von Schnee auseinander, wobei der Fokus wie beispielsweise in [6] oft auf der Simulation elasto-plastischer Materialien liegt und wieder andere beschäftigen sich rein mit Lawinen. In Abschnitt 2.1.1 findet sich ein Modell, mit dem das Schneeverhalten simuliert werden kann.

Ein wichtiger Punkt, der bei einer korrekten Simulation berücksichtigt werden muss, ist jener der verschiedenen Schneearten, welche sich hauptsächlich über die Schneekristalle definieren. Abbildung 1.3 zeigt verschiedene Arten von Schneekristallen, die abhängig von der Temperatur entstehen. Masse, Dichte und Verbundenheit sind nur wenige der Eigenschaften, die sich daraus ergeben und zu sehr unterschiedlichen Verhalten führen. Pulverschnee beispielsweise verhält sich wesentlich anders als fester feuchter Schnee.

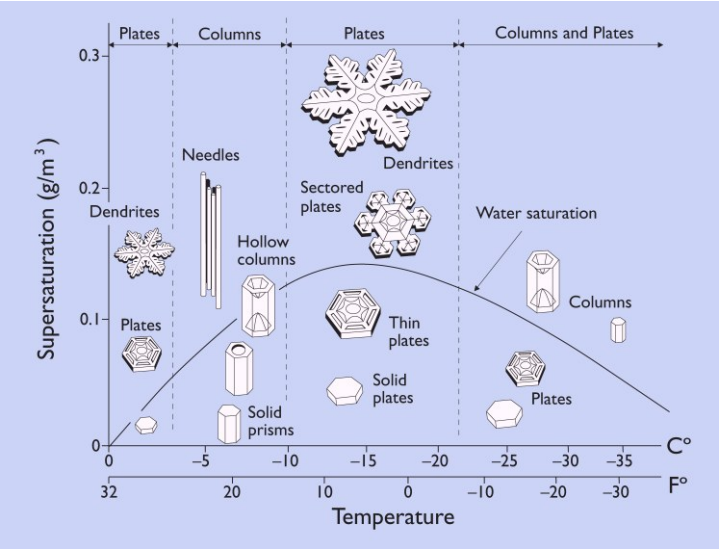


Abbildung 1.3: Übersicht über verschiedene Arten von Schneekristallen, die je nach Temperatur entstehen und zu unterschiedlichen physikalischen Eigenschaften führen. Quelle: [13]

1.4.2 Abstrahiertes Modell

In dieser Arbeit wird auf ein stark abstrahiertes Modell zurückgegriffen. Schnee definiert sich lediglich durch seine Höhe S und seiner Dichte ρ, wobei diese positionsabhängig sind. Das Modell sieht auch einen Boden vor, der allerdings lediglich mit seiner Höhe B zur Simulation beiträgt. Des weiteren können mehrere Objekte in der Simulation vorkommen, wobei von diesen die Höhe der Unterkante U bekannt sein muss. Zusätzlich besitzt jedes Objekt eine Geschwindigkeit $v = |\boldsymbol{v}|$ und eine Masse m. Die Positionen $x/\boldsymbol{x}$, von denen S, ρ, B und U abhängen, sind diskret. In einer Simulation in 2D entspricht x einem Skalar, in 3D ist $\boldsymbol{x}$ ein 2-dimensionaler Vektor. Die Werte für S, ρ, B und U selbst können beliebige Werte größer oder gleich 0 annehmen. Abbildung 1.4 zeigt ein Beispiel, wie eine bestimmte Situation im Modell abgebildet wird.

Die Simulation an sich erfolgt immer für einen konkreten Zustand, wobei zwischen zwei Simulationsschritten beliebig viel Zeit vergehen kann, in der sich die einzelnen Objekte abhängig ihrer Geschwindigkeit bewegen. Das kann zu Überschneidungen zwischen Schnee und Objekten führen, die im Zuge der Simulation wieder berichtigt werden, was dem eigentlichen Hauptteil der Simulation entspricht. Die Differenz

$$D(\boldsymbol{x}) = B(\boldsymbol{x}) + S(\boldsymbol{x}) - U(\boldsymbol{x}) \tag{1.1}$$

ist sowohl für die Schneeverschiebung als auch für die Verlangsamung des

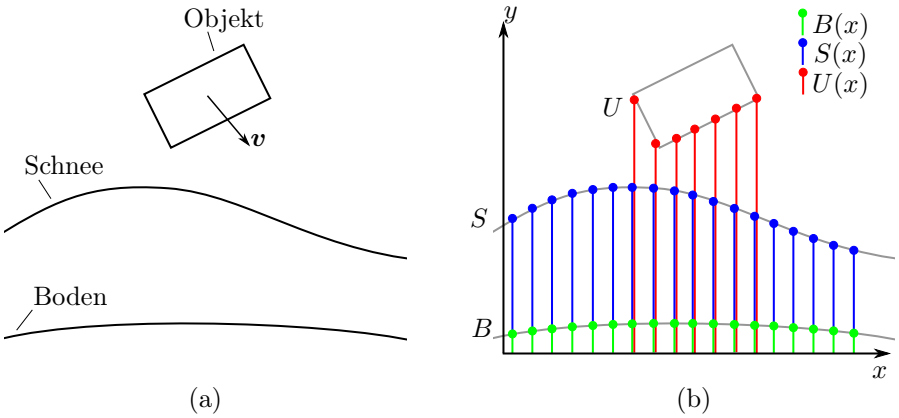


Abbildung 1.4: Eine Szene aus der Realität (a) wird im Modell abgebildet (b). Dabei wird für jede diskrete Position x die Bodenhöhe B, die Schneehöhe S, die Schneedichte ρ (die aus Übersichtsgründen nicht eingezeichnet ist) und die Höhe der Objekt-Unterkante U ermittelt.

Objektes ausschlaggebend. Nachdem D komprimiert bzw. verteilt wurde und sich entsprechend auf die Geschwindigkeit des Objektes ausgewirkt hat (Gegenkraft) ist ein Simulationsschritt abgeschlossen. Eine genauere Beschreibung des Modells befindet sich in Abschnitt 3.1.

1.5 GPU

Moderne Grafikkarten zeichnen sich vor allem durch ihre hohe (parallele) Rechenpower aus, da die GPUs im Vergleich zu den CPUs ein vielfaches an Leistung erbringen. In [26] findet sich beispielsweise eine Tabelle mit Projekten, bei denen durch den Umstieg der Berechnung von CPU auf GPU die Performance um mehr als das 100-fache verbessert werden konnte. Dieser Unterschied zeigt sich auch bei den Floating Point Operations per Second (FLOPS), bei denen der 6-Kern Prozessor i7 980 XE von Intel mit 109 GigaFLOPS[2] deutlich hinter der AMD Radeon HD 6990 mit 1,37 TeraFLOPS[3] hinterher hinkt.

1.5.1 Architektur

- Um zu veranschaulichen, woher diese Leistung kommt, wird hier als Beispiel die Cayman Architektur vorgestellt, die bei den Radeon HD 6900 Grafikkarten zum Einsatz kommt (Abbildung 1.5). Diese Architektur verwendet zwei

[2] http://en.wikipedia.org/wiki/FLOPS

[3] http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/hd-6990/Pages/amd-radeon-hd-6990-overview.aspx#3

Grafik Engines, welche grundsätzliche Aufgaben wie beispielsweise Rasterisierung übernehmen. Pro Takt können durch die beiden Grafik Engines zwei dieser Aufgaben gleichzeitig erfolgen, was zu erheblichen Performance-Gewinnen gegenüber der alten Architektur mit nur einer Grafik-Engines führt. Des weiteren werden bei der Cayman Variante 24 SIMD[4]-Engines eingesetzt, wobei jede für sich 64 ALUs[5] beinhaltet, was in Summe 1536 Shader Processors (SPs) ausmacht, die allesamt gleichzeitig arbeiten können [30].

Die Thread Gruppen der Compute Shader, welche für die Ausführung von Code auf der GPU zuständig sind (genaueres in Abschnitt 1.6.2), werden auf den SIMDs ausgeführt, wobei sich der Speicher, den sich die Threads einer Gruppe teilen (`groupshared`-Memory), im „Local Data Share Memory“ befindet [23]. Da es aus Performance Gründen keinen Sinn macht, Paging wie es beispielsweise beim Arbeitsspeicher üblich ist, für diesen Speicher umzusetzen, ist dieser gemeinsame Speicher limitiert. Bereits beim Kompilieren der Compute Shader wird daher sichergestellt, dass die als `groupshared` gekennzeichneten Variablen 32 kB nicht übersteigen. Der Zugriff auf den „Local Data Share Memory“ ist um einiges schneller, als der Zugriff auf Texturen und UAVs (siehe Abschnitt 1.6.2), wodurch es sich oft anbietet, zuerst die Daten in den geteilten Gruppenspeicher zu laden, dort zu bearbeiten und anschließend wieder entsprechend zurückzuschreiben.

1.5.2 Render Pipeline

Um die Hardware entsprechend ausnützen zu können, gibt es die Möglichkeit einzelne Bereiche davon mittels Programmen – Shadern – zu steuern. Dabei gibt es seit DirectX 11 im Prinzip zwei verschiedene Pipelines (siehe Abbildung 1.6). Programmierbar sind dabei Vertex Shader, Hull Shader, Domain Shader, Geometry Shader und Pixel Shader bei der Grafikpipeline und Compute Shader bei der alternativen Pipeline. Der Vertex Shader dient zur Positionierung von Vertices (anhand von Position des Objektes und Kamera), Domain und Hull Shader sind neu bei DirectX 11 um die Tessellierung zu steuern, Geometry Shader ermöglichen es neue Geometrie zu erzeugen und der Pixel Shader ist für die einzelnen Pixel der Ausgabe zuständig. Compute Shader werden für GPGPU eingesetzt, womit es unnötig ist, andere Bereiche der Grafik Pipeline zu durchlaufen, wenn lediglich allgemeine Berechnungen durchgeführt werden sollen. Genaueres zu den neuen Shadern in DirectX 11 findet sich in Abschnitt 1.6.2.

[4]SIMD – Single Instruction, Multiple Data. Dabei wird derselbe Befehl auf unterschiedliche Daten angewendet, was vor allem bei parallelisierbaren Aufgaben gut einsetzbar ist.

[5]Eine arithmetisch-logische Einheit (englisch arithmetic logic unit, daher oft abgekürzt ALU) ist ein elektronisches Rechenwerk, welches in Prozessoren zum Einsatz kommt. Quelle: [14]

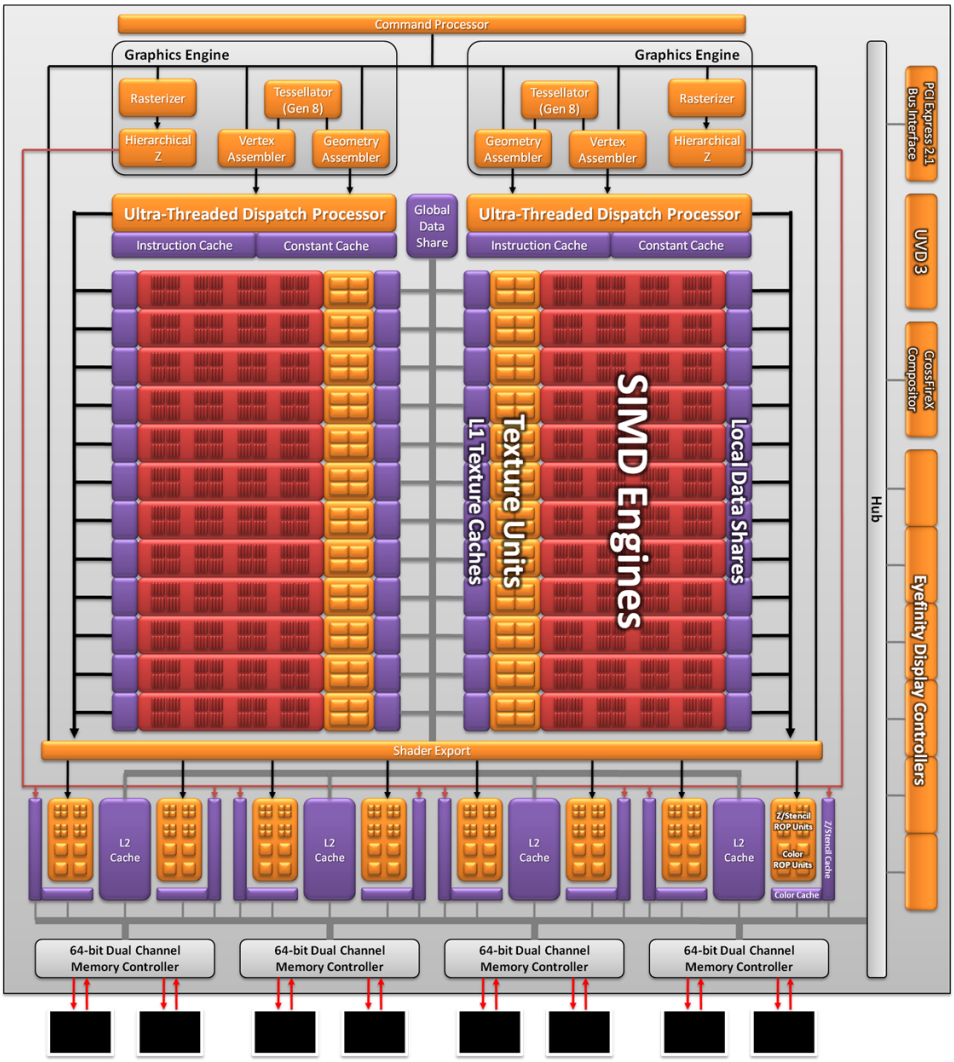


Abbildung 1.5: Cayman Architektur der Radeon HD 6900 Grafikkarten-Serie. Quelle: [15]

1.6 Shader Models

Bezüglich Shader Models gibt es keine einheitliche Definition. Im Prinzip legt ein Shader Model aber fest, welche Funktionalitäten von den einzelnen Shadern unterstützt werden. Eine Version eines Shader Models ist größtenteils abwärts kompatibel, wobei seit Version 4, die Features des ersten Shader Models nicht mehr unterstützt werden. Unterstützt eine Hardware also beispielsweise Shader Model 4.0, dann können auch Funktionalitäten der Shader Models 2 und 3 verwendet werden.

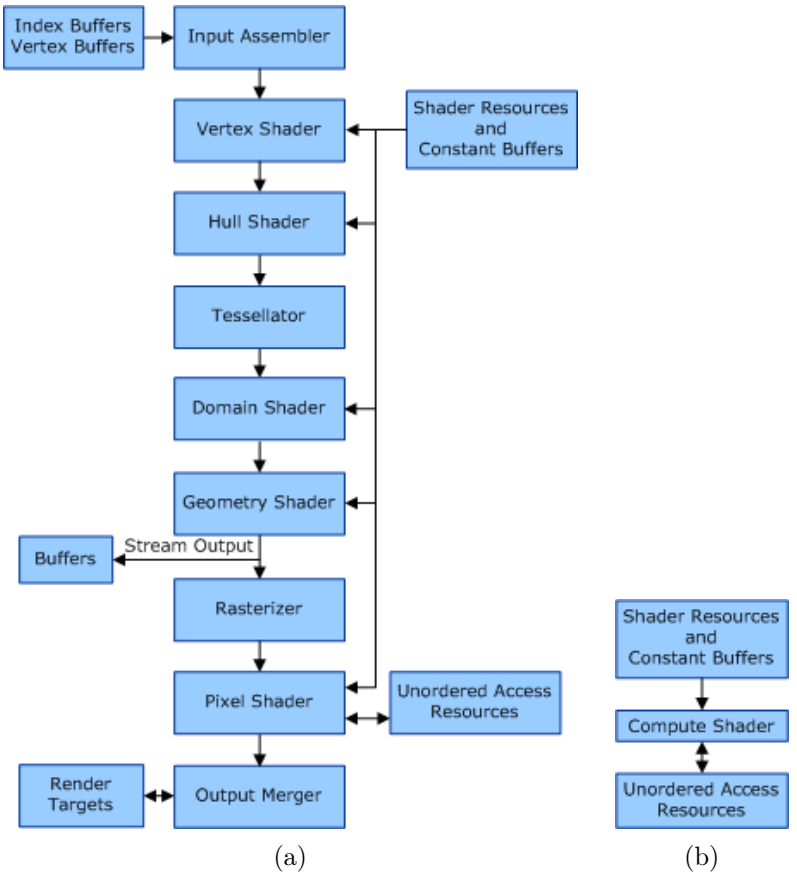


Abbildung 1.6: Pipelines von DirectX 11. Die Grafik Pipeline ist ähnlich der von DirectX 10, lediglich erweitert um die 3 Abschnitte für die Tessellierung: Hull Shader, Tessellator und Domain Shader (a). Zusätzlich gibt es bei DirectX eine eigene Pipeline für Compute Shader, welche losgelöst von der grafischen Pipeline existiert (b). Quelle: [16]

1.6.1 Geschichte

Die Version 1.1 des Shader Models wurde 2001 mit DirectX 8 eingeführt, wodurch auch die ersten Shader möglich waren, mit denen der Rendervorgang auf der GPU beeinflusst werden konnte (wenn auch nur sehr limitiert). 2003 wurde dann mit DirectX 9.0a das Shader Model 2.0 eingeführt, welches nun auch Schleifen in den Shadern ermöglichte. Mit DirectX 9.0c und dem damit verbundenen Shader Model 3.0 wurden if/else Verzweigungen realisierbar. Sowohl PlayStation 3 als auch XBox 360 unterstützen das Shader Model 3.0. 2006 wurden mit dem Shader Model 4 (DirectX 10) Geometry Shader eingeführt, mit denen es möglich ist, auf der GPU neue Geometrie zu erstellen. Das aktuellste Shader Model, dessen Funktionalitäten in dieser Arbeit intensiv genutzt werden, kam 2009 mit DirectX 11. Auf dessen Neuerungen

wird im nachfolgenden Abschnitt näher eingegangen [17, 25].

1.6.2 Shader Model 5

Mit DirectX 11 ist auch das Shader Model 5 eingeführt worden, welches im Vergleich zum Vorgänger wesentliche Neuerungen gebracht hat. Die Wichtigsten davon, welche in dieser Arbeit Verwendung finden, sind Tessellierung und Compute Shader, wobei hierfür auch neue Datenstrukturen eingeführt wurden.

Unordered Access Views

Eine Unordered Access View (UAV) ist eine Shader Resource, auf die an beliebige Stellen geschrieben bzw. von der aus beliebiger Stelle gelesen werden kann. Dieser View liegt meist ein Buffer (z.B. RWStructuredBuffer) zugrunde, wodurch ein Shader über die UAV die Werte des Buffers verändern kann. Gesetzt werden sie über `OMSetRenderTargetsAndUnorderedAccessViews()` für Pixel Shader, bzw. `CSSetUnorderedAccessViews()` für Compute Shader.

Structured Buffer

Structured Buffer sind vergleichbar mit Arrays von Objekten. In HLSL[6] wird zunächst ein `struct` definiert, welches anschließend als Datenstruktur für den Buffer herangezogen wird. Wie dieser Buffer verändert werden kann, hängt davon ab, wie er an den Shader gebunden wird. UAVs ermöglichen beispielsweise Lese- und Schreibeoperationen (RWStructuredBuffer), herkömmliche Shader Resource Views (SRVs) lediglich Leseoperationen (StructuredBuffer). Programm 1.1 zeigt ein Beispiel für die Verwendung von Strutured Buffers.

Append/Consume Buffer

Ein Append Buffer ermöglicht das Anfügen neuer Elemente mittels `Append()` an das Ende des Buffers. Wird dieser Buffer dann bei einem anderen Shader als Consume Buffer gebunden, kann mittels `Consume()` das letzte Element wieder geholt werden. Auf diese Weise ist es möglich, eine Art Stack/Liste umzusetzen. Allerdings ist es nicht zwingend erforderlich den Buffer mittels `Consume()` auszulesen, da dieser Buffer auch als Structured Buffer gebunden werden kann und somit mittels normaler Indizierung darauf zugegriffen werden kann.

[6] HLSL (High Level Shading Language) ist die Sprache, die für die Erstellung von Shadern in DirectX verwendet wird.

```
1 struct MyStruct
2 {
3    float4 vValue1;
4    uint   uBitField;
5 };
6
7 StructuredBuffer< MyStruct >   MyInputBuffer;  // SRV
8 RWStructuredBuffer< MyStruct > MyOutputBuffer; // UAV
9
10 float4 MyPS( PSINPUT input ) : COLOR
11 {
12    MyStruct StructElement;
13    StructElement = MyInputBuffer[ input.index ]; // Read from SRV
14    MyOutputBuffer[ input.index ] = StructElement; // Write to UAV
15
16    // Rest of code ...
17 }
```

Programm 1.1: Beispiel aus [31], welches die Verwendung unterschiedlicher Structured Buffer zeigt.

Compute Shader

Compute Shader sind eines der Hauptfeatures des Shader Models 5. Dabei handelt es sich um einen separaten Renderschritt (losgelöst von der normalen Renderpipeline) der lediglich für GPGPU verwendet wird. Hierbei wird auf eine riesige Anzahl an Threads zurückgegriffen, welche in mehrere Gruppen zusammengefasst werden. Threads innerhalb einer Gruppe können gemeinsam auf einen gleichen Speicherbereich am Chip zugreifen (der jedoch mit 32KB pro Gruppe limitiert ist), was wesentlich schneller ist, als der Zugriff auf globale Speicher (z.B. Texturen, UAVs, etc.).

Jeder Compute Shader verfügt über ein 3-dimensionales Array an Gruppen, wobei jede Gruppe wieder aus einem 3-dimensionalen Array an Threads besteht. Auf diese Weise lassen sich parallele Aufgaben relativ gut aufsplitten. Allerdings gilt es wie bei gewöhnlichen multi-threading Programmierung die Concurrency Problematik der „zeitgleichen“ Schreib-/Lese-Zugriffe. Hierfür existieren aber im Shader Model 5 entsprechende atomare Operationen wie beispielsweise `InterlockedAdd()` oder `InterlockedExchange()`, die zwar zulasten der Performance gehen, jedoch für ein funktionierendes Programm unverzichtbar sind.

Der grundsätzliche Aufbau eines Compute Shaders ist in Programm 1.2 gezeigt. Darin ist zu sehen, dass die Anzahl der Threads pro Gruppe bei der Shader Methode selbst definiert ist. Die Anzahl der Gruppen wird beim Aufruf, welcher in der Form

```
1 m_deviveContex->CSSetShader( m_shader, NULL, 0 );
2 m_deviveContex->Dispatch( m_groupsX, m_groupsY, 1 );
```

```
 1 struct BufferStruct
 2 {
 3    uint4 color;
 4 };
 5
 6 RWStructuredBuffer<BufferStruct> g_OutBuff;
 7
 8 [numthreads( 4, 4, 1 )] // number of threads per group 4 x 4 x 1
 9 void main( uint3 threadIDInGroup : SV_GroupThreadID, uint3 groupID :
       SV_GroupID )
10 {
11    int index = ( groupID.y * THREAD_GROUP_SIZE_Y + threadIDInGroup.y ) *
        THREAD_GROUPS_X * THREAD_GROUP_SIZE_X +
12                ( groupID.x * THREAD_GROUP_SIZE_X + threadIDInGroup.x );
13    g_OutBuff[ index ].color = float4( 1, 0, 0, 1 );
14 }
```

Programm 1.2: Beispiel eines (sinnfreien) Compute Shaders, bei dem sich jeder Thread ausgehend von seiner Gruppen ID (`groupID`) und der ID innerhalb der Gruppe (`threadIDInGroup`) einen Index errechnet, an dessen Stelle im Buffer dann die Farbe rot gespeichert wird. Die Anzahl der Threadgruppen wird beim Aufruf des Shaders aus dem Programm angegeben.

erfolgt, angegeben. Zuerst muss also der Compute Shader gesetzt werden und anschließend kann er mit `Dispatch()` gestartet werden, wobei diese Funktion die Anzahl der Gruppen mitübergeben bekommt.

Tessellierung

Neben den Compute Shadern, die zweite große Neuerung im Shader Model 5 ist die Tessellierung. Dabei werden Polygone, die an die GPU geschickt werden, von dieser weiter trianguliert, wodurch ein viel höher aufgelöstes Mesh entsteht. In Kombination mit Displacement Mapping lassen sich so trotz grundsätzlich eher niedrig aufgelöster Meshes hohe Detailgrade ohne nennenswerte Performance-Verluste erzielen, da hauptsächlich der CPU-GPU Transfer zulasten der Performance geht. In Abbildung 1.7 wird ein Beispiel für den Einsatz von Tessellierung in Kombination mit Displacement Mapping gezeigt.

Die Tessellierung selbst erfolgt über die drei neuen Abschnitte in der Renderpipeline: Hull Shader, Tessellator und Domain Shader, wobei bis auf den Tessellator beide programmierbar sind.

Hull Shader Der Hull Shader bekommt vom Vertex Shader die einzelnen Positionen der Vertices und besteht aus zwei Teilen, der Patch Constant Function und dem Hull Shader an sich. Erstere ist für die Ermittlung jener Daten zuständig, die für einen gesamten Patch (z.B. Quad, Triangle) gleich

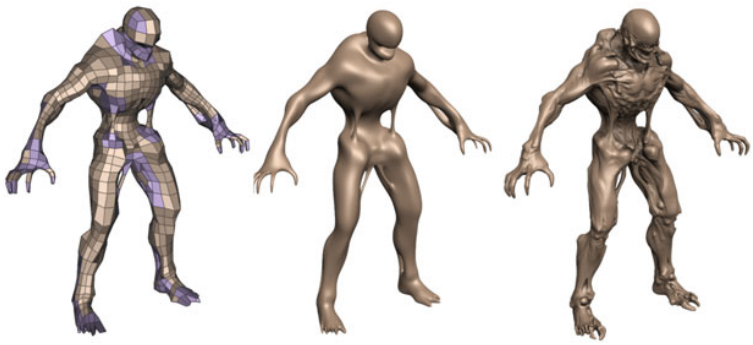


Abbildung 1.7: Ein niedrig aufgelöstes Objekt (links) wird tesseliert, wodurch ein hoch aufgelöstes Mesh erzeugt wird (Mitte). Auf dieses Mesh kann dank der vielen Polygone Displacement Mapping angewendet werden (rechts). Grafik von Kenneth Scott, id Software 2008. Quelle: [18]

sind, wobei die Funktion nur einmal pro Patch aufgerufen wird. Zwingend erforderlich dabei sind die Tessellierungsfaktoren für die einzelnen Kanten bzw. für die innere Tessellierung. Zusätzlich können jedoch beliebige Informationen (z.B. Tangenteninformationen) ermittelt werden, die jedoch vom Tessellator selbst nicht berücksichtigt werden. Der Hull Shader an sich wird pro erzeugten Kontrollpunkt aufgerufen, wobei ihm außer den kompletten Patch-Informationen auch die ID des Kontrollpunktes zur Verfügung steht. Auf diese Weise können beispielsweise Kontrollpunkte für Bézier-Kurven berechnet werden, welche später im Domain Shader weiterverwendet werden.

Tessellator Der Tessellator ist im Gegensatz zum Hull und Domain Shader nicht programmierbar. Er generiert aus dem Patch und den vom Hull Shader ermittelten Tessellierungsfaktoren neue Dreiecke (bis zu 8192 [32]), welche er zusammen mit den Kontrollpunkten vom Hull Shader an den Domain Shader weiterleitet.

Domain Shader Der Domain Shader wird für jeden neu erzeugten Vertex aufgerufen, wobei ihm die vom Hull Shader ermittelten Kontrollpunkte übergeben werden. Aufgabe des Domain Shaders ist es nun, die neu erzeugten Vertices, entsprechend der Kontrollpunkte, zu platzieren. Hierfür bekommt jeder Vertex die UVW Koordinaten übergeben, die die Position des Vertices im ursprünglichen Patch widerspiegeln.

Kapitel 2

Verwandte Arbeiten

2.1 Schneedarstellung/-simulation

Im Bereich Schneesimulation/-darstellunge gibt es einige unterschiedliche Thematiken, die in der Literatur behandelt werden. Die Arbeiten diesbezüglich reichen von realistisch fallenden Schneeflocken, über liegen bleibenden/schmelzenden Schnee bis hin zur realistischen Schneedarstellung an sich. Es gibt auch einige Werke, die sich mit der allgemeinen Verformung von Materialien beschäftigen, was natürlich auch teilweise auf Schnee anwendbar ist.

2.1.1 Schneephysik

Bezüglich Schneephysik gibt es einige Werke, die versuchen die komplexen Eigenschaften zu beschreiben und zusammenzufassen. Diese beschäftigen sich mit einer recht großen Anzahl unterschiedlicher Teilbereiche.

Allgemeine Literatur zur Schneesimulation

Eines der größeren Forschungsgebiete im Bereich Schnee ist jenes der Lawinenforschung. In [7] wird beispielsweise gezielt auf die Bruch-Mechanik von Schnee eingegangen, die für Lawinen entscheidend ist. Im Gegensatz dazu gibt es auch Werke, die sich auf die grundsätzlichen Mechaniken und die damit verbundenen physikalischen Eigenschaften von Schnee konzentrieren. In [28] wird beispielsweise der Aufbau von Schneeflocken (zu deren Entstehung es ebenfalls eigene Werke gibt) und die daraus resultierende grundlegende Physik bezüglich Dichte, Temperatur und Flüssigkeitsgehalt von Schnee beschrieben. Eine weitere Beschreibung von grundlegenden Schneemechaniken, u.a. mit Bezug auf die viskoelastischen[1] Eigenschaften von Schnee gibt es in [6]. In [9] wird eine Möglichkeit gezeigt, wie u.a. Viskoelastizität auf

[1] Als Viskoelastizität bezeichnet man die zeit-, temperatur- und frequenzabhängige Elastizität von Schmelzen oder Festkörpern [19].

3D-Meshes anwendet werden kann. Eine etwas allgemeinere Abhandlung bezüglich der Eigenschaften von elasto-plastischen[2] Stoffen, findet sich in [3].

Rheologische Modelle zur Schneesimulation

In der Rheologie[3] gibt es einen Teilbereich für viskoelastische Stoffe, welchem auch Schnee zugeordnet werden kann. Hierbei wird die Elastizität mittels Feder(n) und die Viskosität (Zähflüssigkeit) mittels newtonschen Dämpfungsglied(ern) simuliert. Abbildung 2.1 veranschaulicht die Zusammenhänge von Druck σ und Verformung ϵ.

Bei einer Feder existiert eine Verformung

$$\epsilon = \frac{\sigma}{E} \tag{2.1}$$

nur so lange auch tatsächlich ein Druck σ ausgeübt wird, wobei der Elastizitätsmodul[4] E ausschlaggebend ist, wie stark sich dieser auswirkt. Mit einer Feder können somit elastische Stoffe simuliert werden, da diese anschließend wieder in ihren Ursprungszustand zurückkehren. Bei zähflüssigen Stoffen wird auf ein newtonschen Dämpfungsglied zurückgegriffen, da sich bei diesem die Verformung

$$\frac{\mathrm{d}\epsilon}{\mathrm{d}t} = \frac{\sigma}{\eta} \tag{2.2}$$

nur unter bestehendem Druck verändert und Änderungen somit auch nach dessen Einwirken bestehen bleiben. η gibt an, wie zähflüssig der Stoff ist. Um viskoelastische Stoffe simulieren zu können, gibt es mehrere Modelle, die Federn und newtonsche Dämpfungsglieder kombinieren, um das gewünschte Verhalten zu erreichen. In Abbildung 2.2 werden die drei meistvertretenen Modelle gezeigt.

Mit Hilfe des Maxwell Models, bei dem eine Feder und ein newtonsches Dämpfungsglied in Serie geschaltet sind, ist es möglich, das grundsätzliche Verhalten von viskoelastischen Stoffen zu simulieren, wobei die Verformung durch das newtonsche Dämpfungsglied linear ist (Abbildung 2.3(b)), was im Vergleich zum tatsächlichen viskoelastischen Verhalten (Abbildung 2.1(d)) der Hauptunterschied ist. Das Kelvin-Voigt Model hingegen ermöglicht durch

[2] Elasto-plastische Materialien weisen eine Kombination aus elastischen (nehmen die Ursprungsform nach Verformung wieder ein) und plastischen (behalten die verformte Form bei) Eigenschaften auf.

[3] Die Rheologie ist die Wissenschaft, die sich mit dem Verformungs- und Fließverhalten von Materie beschäftigt. Die Rheologie umfasst daher Teilgebiete der Elastizitätstheorie, der Plastizitätstheorie und der Strömungslehre (nichtnewtonsche Flüssigkeiten) [20].

[4] Der Elastizitätsmodul ist ein Materialkennwert aus der Werkstofftechnik, der den Zusammenhang zwischen Spannung und Dehnung bei der Verformung eines festen Körpers bei linear elastischem Verhalten beschreibt. Der Betrag des Elastizitätsmoduls ist umso größer, je mehr Widerstand ein Material seiner Verformung entgegensetzt. Quelle: http://de.wikipedia.org/wiki/Elastizitätsmodul

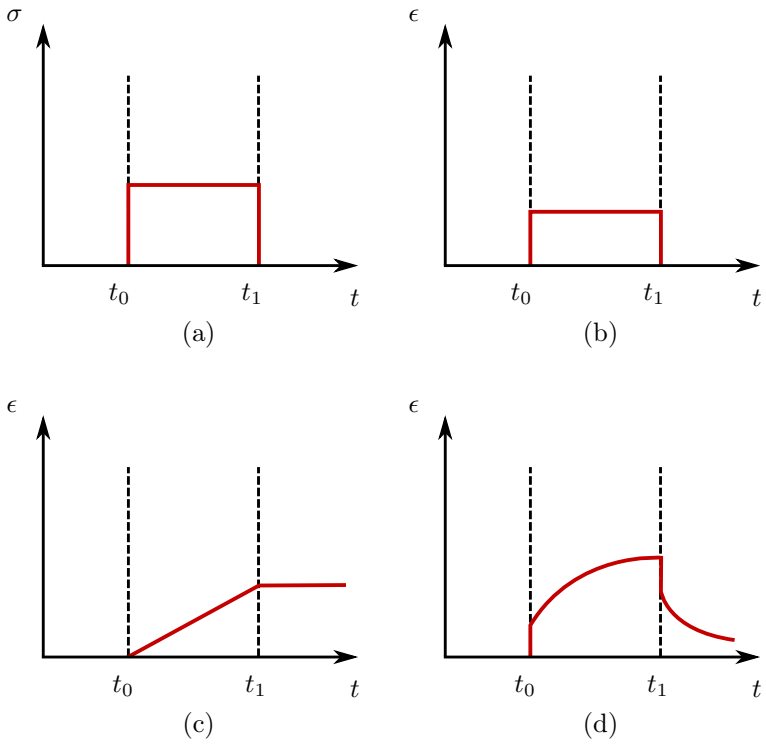


Abbildung 2.1: Solange ein Druck σ (a) exisitert, führt das bei einer Feder zu einer Verformung ϵ (b). Damit kann elastisches Verhalten simuliert werden. Die Verformung eines newtonsches Dämpfungsgliedes verändert sich nur unter existierendem Druck und bleibt anschließend erhalten (c), was zähflüssigen Stoffen entspricht. (d) zeigt das tatsächliche Verhalten von viskoelastischen Stoffen.

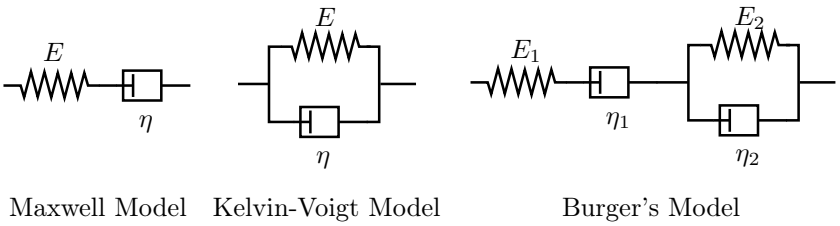


Abbildung 2.2: Verschiedene Modelle unter der Verwendung von Federn und newtonschen Dämpfungsgliedern um viskoelastisches Verhalten zu simulieren. Burger's Model besteht aus einer seriellen Kombination aus Maxwell Model und Kelvin-Voigt Model.

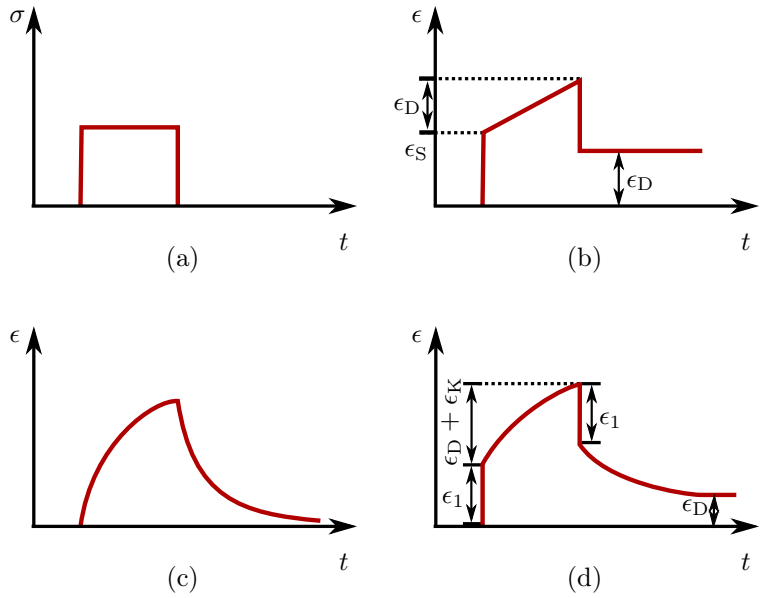


Abbildung 2.3: Ergebnisse der einzelnen Modelle unter Einwirkung eines bestimmten Drucks (a). Das Maxwell Model liefert eine gute Annäherung an das Verhalten viskoelastischer Stoffe, wobei der viskose Anteil der Deformation linear ist (b). Eine bessere Methode für den viskosen Anteil bietet das Kelvin-Voigt Model (c). Eine Kombination dieser beiden Modelle (Burger's Model) bietet eine gute Annäherung an viskoelastisches Verhalten (d).

das parallele Kombinieren von Feder und newtonschen Dämpfungsglied das Simulieren von verzögerter Elastizität (Abbildung 2.3(c)). Hierbei fehlt allerdings die sofortige Deformation. Um viskoelastisches Verhalten zu erreichen, werden Maxwell Model und Kelvin-Voigt seriell kombiniert, was schließlich Burger's Model entspricht. Das dabei entstehende Ergebnis (Abbildung 2.3(d)) ist eine recht gute Annäherung an das tatsächliche Verhalten.

Auch wenn diese Modelle eine recht gute Beschreibung für viskoelastische Deformationen darstellen, sind sie gerade für Schnee nur bedingt geeignet. Das Problem dabei ist, dass die Viskosität und die Elastizität von Schnee im Bezug zur Temperatur und Dichte stark nichtlinear sind [28]. Ein weiteres großes Problem stellt die große Komprimierbarkeit dar, durch die sich Schnee von den meisten anderen Stoffen unterscheidet. Der Elastizitätsmodul variiert bereits um drei Größenordnungen im Bereich der verschiedenen Dichten, in denen sich Schnee üblicherweise befindet [28].

(a) (b)

Abbildung 2.4: Ergebnisse aus [4], in denen gezeigt wird, wie aus einer gewöhnlichen Landschaftsszene (a) eine Winterlandschaft (b) wird. Dabei wird für jede Position ermittelt, wie viel Schnee sich dort befindet, woraus anschließend ein eigenes Mesh für den Schnee generiert wird.

2.1.2 Mesh Generierung

Paul Fearing hat sich in seiner Dissertation [4] mit der Darstellung von gefallenem Schnee beschäftigt (Abbildung 2.4). In dieser Arbeit setzt er sich auch mit den physikalischen Eigenschaften von Schnee auseinander, beginnend bei Schneekristallen bis hin zur Entstehung von Lawinen.

Bei der Umsetzung des gefallenen Schnees wird als erstes ermittelt, wie viel Schnee sich an einer Position befindet. Das passiert, indem von jeder Position aus überprüft wird, ob freie Sicht auf den Himmel herrscht. Falls nicht, wird errechnet, wie groß die Chance ist, dass sich dennoch Schnee dort befindet (z.B. durch Wind). Ausgehend von den ermittelten Werte wird anschließend ein 3D Mesh generiert. Diese Methode eignet sich nur bedingt für die Ermittlung von Spuren im Schnee, da sie grundsätzlich für statische Szenen entwickelt wurde, in denen lediglich gefallener Schnee simuliert wird.

2.1.3 Partikel-basierte Simulation

Eine weitere Möglichkeit der Simulation von Schnee (Sand, Schlamm, usw.) besteht in der Verwendung von Partikeln. Eine Möglichkeit hierfür zeigen die Autoren Bell, Yu und Mucha in [2]. Hierbei wird eine große Anzahl an Partikeln (mit physikalischen Eigenschaften) simuliert. Auf diese Weise lässt sich beispielsweise Sand recht gut simulieren, aber auch bei Flüssigkeiten wird oft auf eine partikel-basierte Simulation zurückgegriffen. Vorteile dieser Methode liegen in den realistischen Ergebnissen und der Möglichkeit, für einzelne Partikel unterschiedliche Eigenschaften zu definieren (was sich beispielsweise für verschiedene Schneeschichten eignen würde). Großer Nachteil ist jedoch der große Rechenaufwand. Die Simulation einer Lawine, wie in Abbildung 2.5 gezeigt, benötigte pro Bild bereits über 26 Minuten [2]. Bei

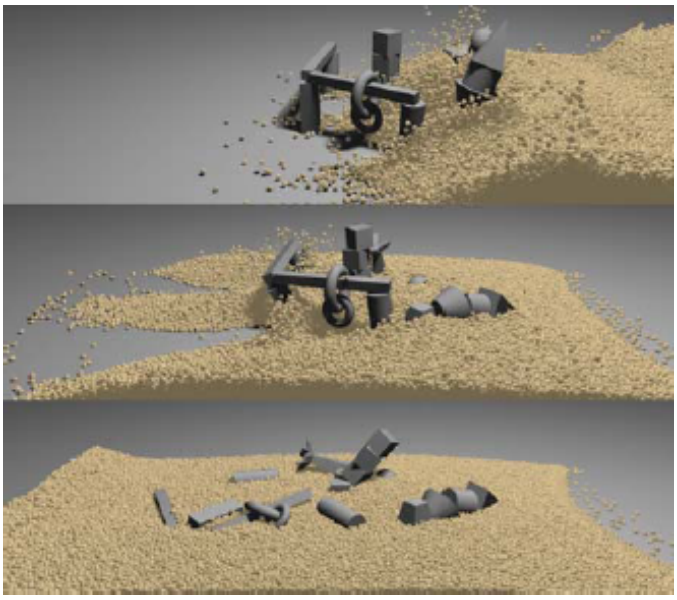


Abbildung 2.5: Partikel-basierte Simulation, in der eine Lawine auf eine Szene mit beweglichen Objekten losgelassenw wird. Quelle: [2]

einigen Methoden, wie zum Beispiel in [8], werden allerdings Partikelsysteme ergänzend eingesetzt, um die optische Qualität des Ergebnisses aufzubessern (um beispielsweise wegfliegenden Schnee zu simulieren).

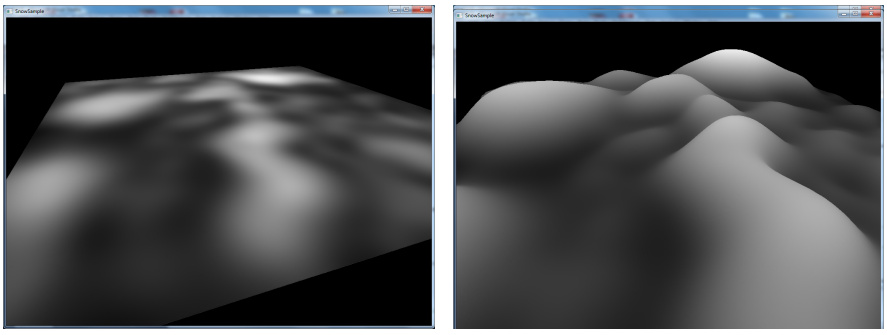
2.1.4 Height Map Simulation

Eine weitere Möglichkeit der Simulation, die im Zuge dieser Arbeit ebenfalls zum Einsatz kommt, besteht darin, mit Höheninformationen (Height Maps) zu arbeiten. Hierbei wird die Schneehöhe für jede Position gespeichert (z.B. in einem Graustufenbild). Beim Darstellen des Schnees wird dann jeder Vertex einer Plane in der Höhe entsprechend den Höheninformationen der Height Map verändert (siehe Abbildung 2.6). Verformungen im Schnee können dann einfach mittels Verändern der Höheninformationen erreicht werden, da sich das Mesh somit automatisch darauf anpasst.

Aufbauend auf diese Methode wird in [10] und [1] gezeigt, wie man mittels Depth Renderings Reifenspuren in schlammigen Untergrund realisieren kann. Hierfür wird die Differenz der Tiefenwerte zwischen Boden und Fahrzeug ermittelt und schlussendlich auf die ursprüngliche Height Map angewendet, auf diese Weise können Spuren relativ schnell umgesetzt werden. Der Vorgang wird in Abbildung 2.7 gezeigt.

Ebenfalls mit einer Height Map realisiert ATI bei einem SDK Beispiel[5] eine Szene, in der in Echtzeit Spuren im Schnee umgesetzt werden (Abbildung

[5] http://www.lynxengine.com/old-site/ati.htm



(a) (b)

Abbildung 2.6: Die Heightmap, die auf eine Plane gemappt wurde (a), führt zu einer entsprechenden Verformung (b). Dabei ist recht schön ersichtlich, dass je heller eine Stelle in der Height Map ist, desto höher ist die Position des Vertex im Mesh.

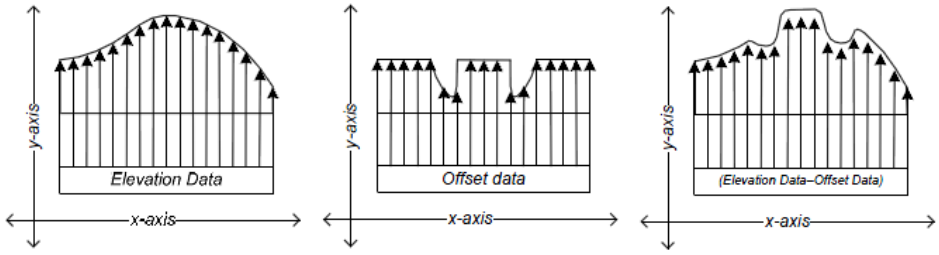


Abbildung 2.7: Reifenspuren im Terrain. Zu den Höheninformationen des Bodens (y-Achse) werden die Differenzwerte von Boden und Fahrzeug addiert, wodurch sich eine Verformung in Spurenform ergibt. Quelle: [1]

2.8). Dabei werden die endgültigen Höhenwerte (Minimum aus Bodenhöhe und Objekthöhe) in eine Textur gerendert, wobei die Kollissionserkennung ebenfalls texturbasiert erfolgt. Ausgehend von dieser Textur werden die Werte in den Vertexbuffer gerendert, was dazu führt, dass die Verformungen im Mesh ersichtlich sind.

Ausgehend von den ermittelten Differenzwerten (die die verdrängte Masse widerspiegeln) wird in [8] gezeigt, wie man eine grafische Simulation verschiedener Materialien wie Sand, Schlamm und Schnee erreichen kann. Hierfür werden fünf Parameter verwendet (u.a. Flüssigkeitsanteil und Komprimierungsgrad) die bestimmen, wie die verdrängte Masse verteilt/komprimiert wird. Je nach Komprimierungsgrad „verschwindet“ ein gewisser Anteil einfach. Der Rest wird zuerst rund um das Objekt (das die Verdrängung verursacht) verteilt. Anschließend werden die entstandenen Säulen so auf die umliegenden Felder verteilt, dass die entstandenen Hänge eine vorab defi-

Abbildung 2.8: Render to Vertexbuffer Beispiel aus dem ATI SDK. Quelle: [29]

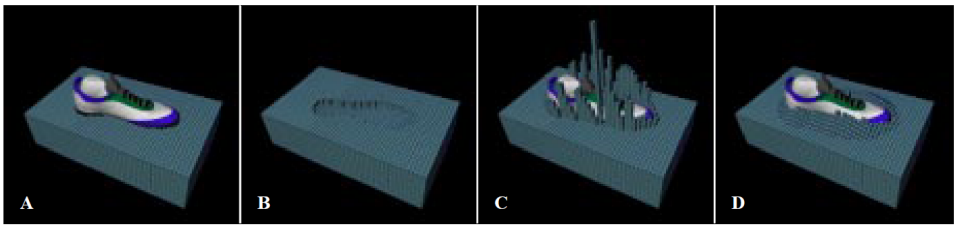


Abbildung 2.9: Die Differenz aus Boden und Objekt (a) wird ermittelt und entsprechend zum Boden addiert (b). Die verdrängte Masse wird an den Rand des Objektes verschoben (c) und anschließend verteilt (d). Quelle: [8]

nierte Maximalsteilheit nicht übersteigen. Hierfür wird auf die Methode aus [5] zurückgegriffen, in der immer angrenzende Bereiche auf ihre Höhenunterschiede überprüft werden. Wird eine Position gefunden, die eine zu hohe Differenz zu dessen Nachbarn aufweist, wird Material von der zu hohen Stelle auf dessen Nachbarfelder verteilt. Der genaue Ablauf ist in Abbildung 2.9 visualisiert.

2.2 Shader Model 5

Bezüglich Shader Model 5 gibt es relativ wenig Literatur. Ein Großteil der Informationen zu Tesselierung[6] und Compute Shadern[7], den größten Neuerungen des Shader Model 5, finden sich in einzelnen Tutorials, die im Internet zur Verfügung stehen oder mit dem DirectX SDK[8] mitinstalliert werden. Es gibt auch eine Onlinedokumentation [21], wobei diese hauptsächlich eine Auflistung der einzelnen Methoden liefert, deren Beschreibungen des öfteren nicht sehr hilfreich sind.

2.2.1 Tessellierung

In [32] wird das Shader Model 5 vorgestellt wobei der Schwerpunkt der Arbeit auf der Tessellierung liegt. Es wird die grundlegende Shaderarchitektur erklärt und einige Code-Beispiele veranschaulichen die Verwendung der hardwareseitigen Tessellierung.

Komplexere Beispiele zum Thema Tessellierung finden sich in den mitgelieferten Samples von DirectX. Im „DetailTessellation11 Sample“ werden beispielsweise verschiedene Arten von Tessellierungsalgorithmen vorgestellt, wie Screen-space adaptive Tessellation, Distance-adaptive Tessellation und Density-based Tessellation. Screen-space adaptive Tesselation unterteilt ein Polygon stärker, je mehr Platz es auf dem Bildschirm einnimmt, Distance-adaptive Tessellation legt den Tesselierungsfaktor anhand der Entfernung des Polygons zur Kamera fest und density-based Tessellation basiert auf einer Deformation Texture (Height Map) anhand derer ermittelt wird, wie stark sich ein Vertex von seinen Nachbaren unterscheidet und wie stark dementsprechend tesseliert werden muss. Diese Methoden können auch miteinander kombiniert werden. Nachdem die density-based Tessellation auf einer Height Map basiert, lässt sie sich ausgesrochen gut mit der Height Map Simulation (siehe Abschnitt 2.1.4) kombinieren. Ein Vergleich der verschiedenen Algorithmen wird in Abbildung 2.10 gezeigt.

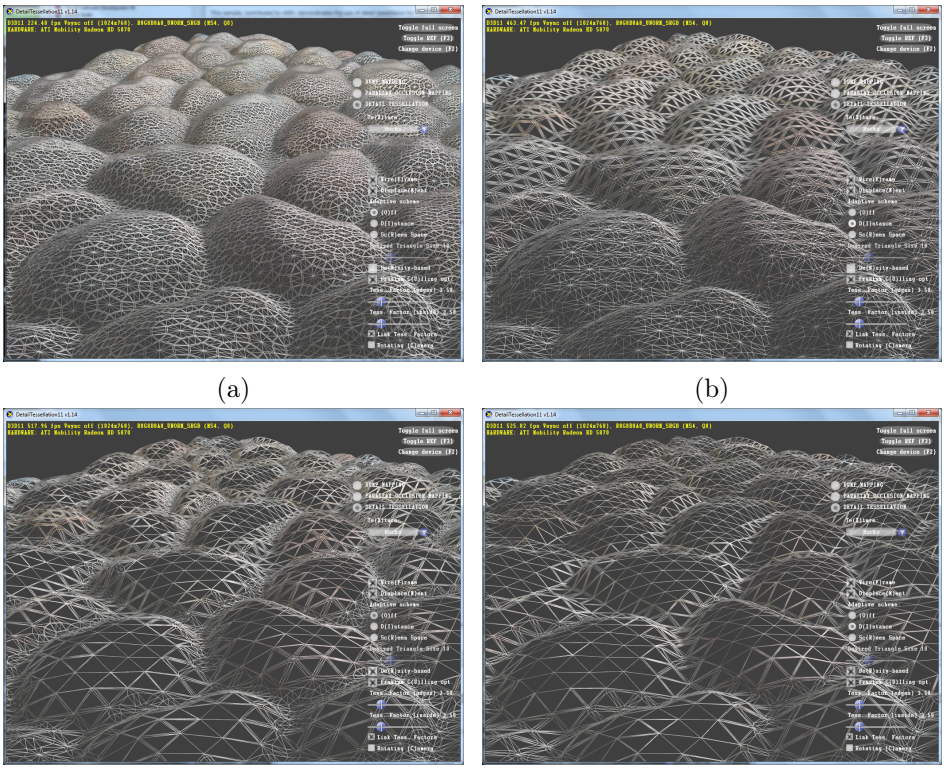
2.2.2 Compute Shader

Auch im Bereich Compute Shader gibt es DirectX Samples, angefangen von einfachen Tutorial Beispielen bis hin zu „komplexen“ Partikelsimulationen. Zu DirectCompute (Überbegriff unter den auch Compute Shader fallen) findet man im Internet allgemeine Informationen, eine wirklich ausführliche Dokumentation gibt es aber zum jetzigen Zeitpunkt noch nicht. Aus diesem Grund wird oft auf die Präsentationen von Boyd [24] und Thibieroz [31]

[6] Hardwareseitiges Zerlegen eines Polygons in mehrere Dreiecke.

[7] Shader, die nicht Teil der Graphics-Pipeline sind, sondern allgemeine Berechnungen durchführen. Sie sind vor allem bei parallelisierbaren Aufgaben sehr effektiv.

[8] http://msdn.microsoft.com/en-us/directx/



(a) (b)

(c) (d)

Abbildung 2.10: Screenshots vom DirectX 11 Sample „DetailTessellation“. Eine gleichmäßige Tessellierung führt zu einem unnötig hoch aufgelösten Mesh (a), was durch distance-adaptive Tessellation (b) oder density-based Tessellation (c) verbessert werden kann. Eine Kombination dieser beiden Methoden führt zum effizientesten Ergebnis (d).

verwiesen. Diese veranschaulichen die Grundlagen von Compute Shadern relativ gut und liefern auch kleine Code-Beispiele. Einen Überblick über die Hardwarearchitektur und damit verbundene Optimierungsmöglichkeiten für Compute Shader zeigt Bill Bilodeau in [23].

Kapitel 3

Allgemeine Herangehensweise

3.1 Modellbeschreibung

Wie in Abschnitt 1.4 bereits kurz beschrieben, gibt es in der Simulation einen schneebedeckten Boden und beliebig viele Objekte. Sollten diese Objekte mit dem Schnee in Berührung kommen, führt das zu Verformungen. Nachdem es sich bei dem verwendeten Modell um eine starke Abstraktion der Realität handelt, gibt es einige Annahmen und Limitierungen.

Boden

Der Boden ist die Grundfläche, auf der sich der Schnee befindet. Er muss nicht zwingend eben sein und dient lediglich dazu ein Terrain (oder andere Untergründe) in der Simulation berücksichtigen zu können, wobei hierbei lediglich die diskreten Höheninformationen $B(x)$ verwendet werden. Diese Werte werden als fix angenommen, wodurch sie auch von der Simulation unangetastet bleiben. Im Prinzip trägt $B(x)$ nur zur Berechnung der Gesamthöhe des Schnees in der Form

$$H(x) = B(x) + S(x) \tag{3.1}$$

bei. Der Boden hat auch keinerlei physikalische Eigenschaften, die von der Simulation berücksichtigt werden, wodurch er sich auch bezüglich Kollisionen nicht auf Objekte auswirkt (hierfür muss eine externe Physik-Simulation verwendet werden).

Schnee

Schnee kann sich nur auf dem Boden befinden, nicht auf Objekten. Er ist (bis auf die Geschwindigkeit der Objekte) das Einzige, das in der Simulation verändert wird. Für jede diskrete Position x sind die Höhe $S(x)$ und die Dichte $\rho(x)$ des Schnees bekannt. Die Höhe wird zur Ermittlung von

Tabelle 3.1: Dichtevergleich unterschiedlicher Schneearten [22].

Dichte ρ in kg/m^3	Bezeichnung
30...50	trockener, lockerer Neuschnee
50...100	gebundener Neuschnee
100...200	stark gebundener Neuschnee
200...400	trockener Altschnee
300...500	feuchtnasser Altschnee
150...300	Schwimmschnee
500...800	mehrjähriger Firn
800...900	Eis

Überschneidungen zwischen Schnee und Objekten benötigt. Über das Verändern dieser Höhenwerte werden die Verformungen im Schnee umgesetzt. Die Dichte wird benötigt, um die Gegenkraft zu ermitteln, die auf ein Objekt wirkt, wenn es mit dem Schnee in Verbindung kommt. Außerdem ist sie wesentlicher Bestandteil bei der Verschiebung und Komprimierung des Schnees. Für die Berechnung des Verteilungs- und des Komprimierungsfaktors (Abschnitt 3.5.1) muss die minimale und maximale Dichte ($\rho_{\min}/\rho_{\max}$) angegeben werden. Auf diese Weise kann das Verhalten des Schnees in der Simulation beeinflusst werden. In Tabelle 3.1 werden die Dichtebereiche verschiedener Schneearten gezeigt.

Kommt es durch die Position eines Objektes zu einer Überschneidung und somit Verformung des Schnees, wird die verdrängte Schneemenge einerseits verteilt und andererseits komprimiert. Zu welchen Teilen das passiert hängt von der Dichte an der entsprechenden Position ab. Eine hohe Dichte führt zu einer stärkeren Verdrängung, eine geringe Dichte hingegen zu einer verstärkten Komprimierung. Eine große Einschränkung der Simulation liegt in der Limitierung auf eine Schneeart. Das ist allerdings erforderlich um die Simulation möglichst einfach zu halten. Die Beschreibung der verwendeten Schneeart erfolgt über die Dichte ($\rho_{\min}$ und $\rho_{\max}$) und die maximale Steilheit $\phi_{\max}$ der Schneehügel.

Objekte

In der Simulation können beliebig viele Objekte vorkommen, die sowohl fix positioniert als auch frei beweglich sein können. Sollten bewegliche Objekte mit dem Schnee in Kontakt kommen, wirkt sich die Verdrängung des Schnees auf die Geschwindigkeit der Objekte aus. Die Gegenkraft durch den Widerstand wird aus der verdrängten Schneemenge berechnet und wirkt der Bewegungsrichtung entgegen. Diese führt somit zu einer Verlangsamung und in dessen Folge zum Stillstand, wobei die Masse des Objektes eine entscheidende Rolle spielt (eine Eisenkugel verhält sich anders als ein Luftballon).

Um zu ermitteln, ob ein Objekt mit dem Schnee in Verbindung gekom-

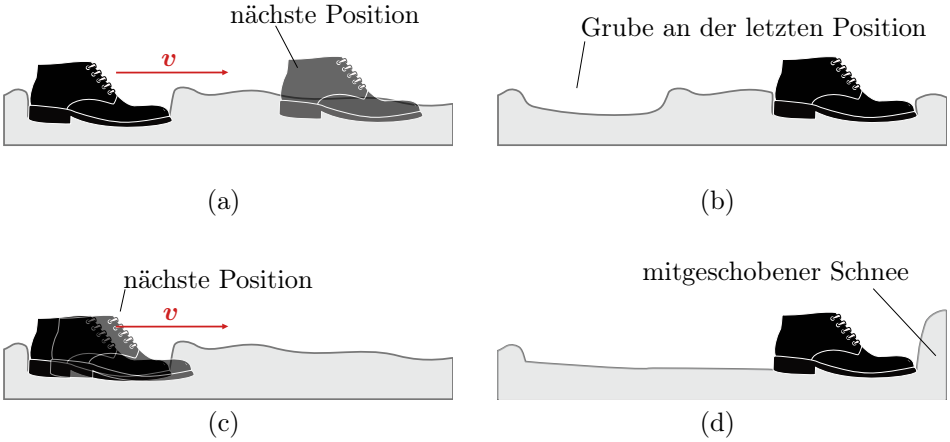


Abbildung 3.1: Zeit zwischen Simulationsschritten. Die Objekte bewegen sich mit einer Geschwindigkeit v, wobei die Position im nächsten Simulationsschritt abhängig von der vergangen Zeit ist (a, c). Ist diese Zeitdifferenz zu groß, bleibt bei der ursprünglichen Position eine Grube zurück und der Schnee wird an der neuen Position entsprechend der Form des Objektes deformiert (b). Bei einer ausreichend kurzen Zeitdifferenz entsteht der Eindruck einer kontinuierlichen Bewegung, da der Schnee immer mitgeschoben wird (d).

men ist, wird die Höhe der Unterkante des Objektes $U(x)$ benötigt. Sollten mehrere Objekte in der Szene sein $(O_1 \dots O_n)$, besitzt jedes eigene Höheninformationen $(U_1(x) \dots U_n(x))$. Für die Simulation wird die tatsächliche Höheninformation

$$U(x) = \min_i U_i(x) \tag{3.2}$$

verwendet, wobei zu berücksichtigen ist, dass $U(x) = \infty$, wenn sich kein Objekt an der Position x befindet.

3.1.1 Simulation

Bei der Simulation werden immer Momentaufnahmen der Szene abgearbeitet, es werden keine Vorgänge berücksichtigt. Um dennoch den Eindruck einer kontinuierlichen Bewegung zu erhalten, müssen die Zeiten zwischen zwei Simulationsschritten relativ klein sein (abhängig von der Geschwindigkeit, mit der sich die Objekte bewegen). Abbildung 3.1 zeigt Ergebnisse, wie sie durch passende bzw. zu lange Simulationszeiten entstehen können.

Jeder dieser Simulationsschritte besteht im Wesentlichen aus zwei Teilen: Ermitteln und Auflösen (Komprimieren/Verteilen) von Überschneidungen. Dabei ist zu berücksichtigen, dass der gesamte Vorgang sehr vereinfacht/abstrahiert ist (z.B. Diskretisierung, Simplifizieren von Schnee-Eigenschaften, nur ein Höhen-/Dichtewert pro Position, etc.). Durch diese Vereinfachun-

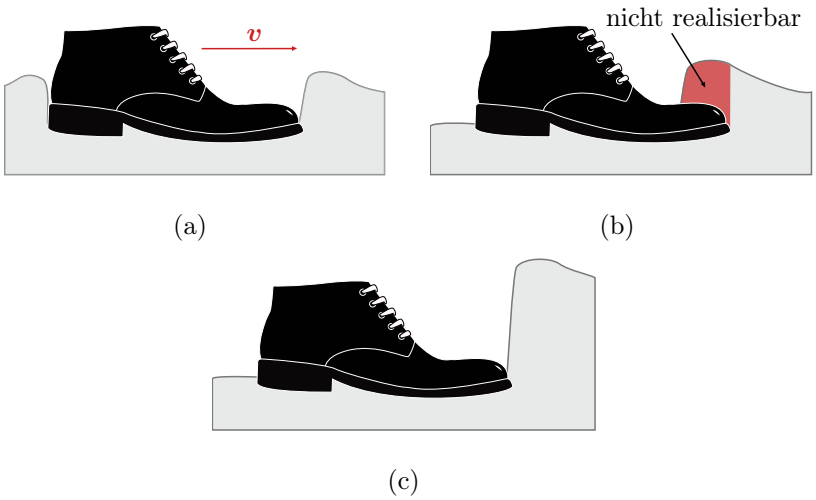


Abbildung 3.2: „Höhlen“. Das Objekt befindet sich im Schnee und bewegt sich nach rechts (a), was dazu führen würde, dass Schnee auf das Objekt fällt (b). Dieser Fall wird jedoch nicht unterstützt, da pro Position nur ein Wert für die Schneehöhe gespeichert werden kann. In diesem Fall wird die komplette Schneemenge einfach mitgeschoben bzw. komprimiert (c).

gen gibt es einige Szenarien, die nicht von der Simulation abgedeckt werden können.

3.1.2 Limitierungen

Eine Limitierung durch das abstrahierte Modell besteht darin, dass pro Position nur ein Höhenwert und ein Dichtewert für den Schnee vorhanden sind. Dadurch wird die Simulation wesentlich vereinfacht, allerdings können so keine verschiedenen Schneeschichten bzw. „Höhlen“ (Abbildung 3.2) dargestellt werden.

Ein weiterer Punkt, der bei dieser Herangehensweise nicht berücksichtigt wird, ist jener, dass sich Verschiebungen im Schnee auf Objekte auswirken können. Ein Schneepflug würde beispielsweise andere Objekte beeinflussen/-verschieben, wenn er Schnee vor sich herschiebt. Bei der in dieser Arbeit verwendeten Methode würde das lediglich zu der Bildung einer hohen Schneesäule führen.

3.2 Ablauf der Simulation

Für die Simulation wird auf eine Height Map basierte Lösung zurückgegriffen (siehe Abschnitt 2.1.4), wobei hier vor allem [8] als Grundlage dient. In dieser Arbeit wird bereits eine Möglichkeit für die Schneeverdrängung gezeigt.

Der Großteil der Simulation funktioniert mittels Höheninformationen, die in den einzelnen Schritten sowohl für die Kollissionserkennung/-behandlung als auch für die Verteilung des Schnees verwendet werden.

Die komplette Deformationsberechnung kann in mehrere Abschnitte eingeteilt werden:

1. Ermitteln der Überschneidung zwischen Objekt und Schnee,
2. Berechnen der Widerstände des Objektes,
3. Komprimieren und Verteilen der verdrängten Schneemasse und
4. Berichtigen zu steiler Hänge.

Diese Abschnitte werden bei jedem Simulationsschritt der Reihe nach durchgearbeitet, wobei die Schritte 1–3 für jedes Objekt einzeln durchgeführt werden müssen. Bei der Berichtigung zu steiler Hänge reicht es aus, die ganze Szene auf einmal abzuarbeiten. Auf diese Weise werden die Objekte abgebremst und der Schnee wird entsprechend der Form der Objekte verdrängt/komprimiert.

3.3 Ermitteln der Überschneidung

Zur Berechnung von Überschneidungen werden die Höhe der Objektunterkante U und die Höhe der Schneeoberfläche

$$H(x) = B(x) + S(x) \tag{3.3}$$

benötigt. Aus diesen kann die Überschneidung

$$D(x) = \max\big(0, H(x) - U(x)\big) \tag{3.4}$$

errechnet werden. Positive Ergebnisse kennzeichnen dabei Überschneidungen, die später entsprechend aufgelöst werden müssen (Komprimieren/Verteilen). Negative Ergebnisse zeigen hingegen, dass sich der Schnee unterhalb des Objektes befindet. Nachdem nur Ergebnisse interessant sind, die zeigen, dass sich ein Objekt im Schnee befindet, müssen alle negativen Werte eliminiert werden. Abbildung 3.3 zeigt den Vorgang.

Das Ergebnis dieses Simulationsschrittes sind Differenzinformationen für alle Positionen (D), welches in Abbildung 3.3(d) gezeigt wird. Diese Information ist Grundlage für die Berechnung der Widerstände und das Verteilen des Schnees.

3.4 Berechnen der Widerstände

Für die Berechnung des Widerstandes, der auf ein Objekt wird, wird die verdrängte Schneemenge benötigt. Für die generelle Berechnung der resultierenden Gegenkraft wird der statische Auftrieb

$$\boldsymbol{F} = \rho \cdot V \cdot \boldsymbol{g} \tag{3.5}$$

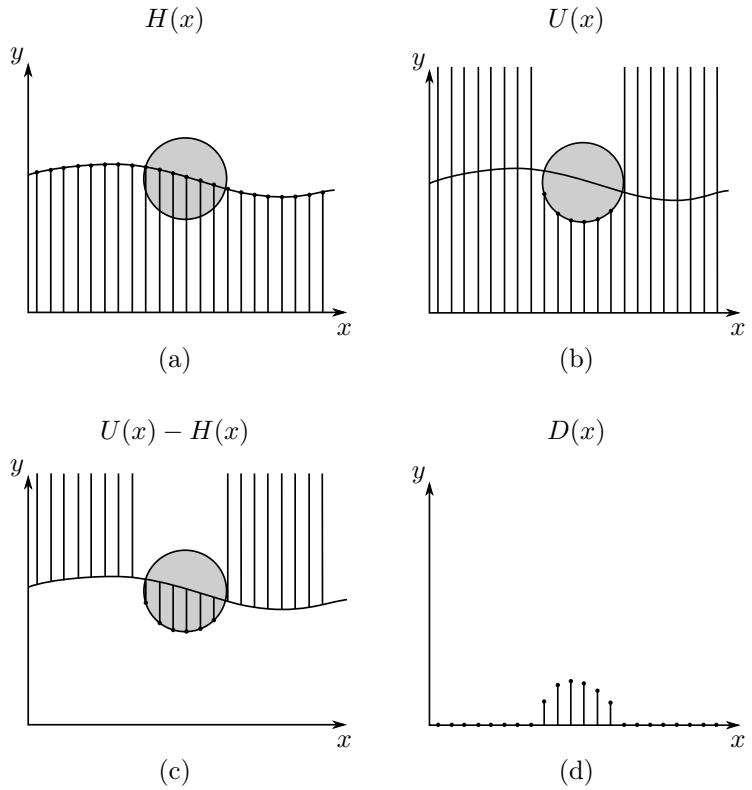


Abbildung 3.3: Ermitteln von Überschneidungen. Die Differenz aus der Schneehöhe (a) und der Objekthöhe(b) zeigt die Überschneidungen zwischen Objekt und Schnee (c), wobei für das finale Ergebnis alle Differenzen eliminiert werden müssen, bei denen die Objekthöhe größer ist als die Schneehöhe (d).

verwendet. Die Dichte $\rho = m/V$ ist dabei bereits in den Positionsinformationen gespeichert. Das Volumen V ergibt sich aus der verdrängten Schneemenge und $\boldsymbol{g}$ entspricht der Erdbeschleunigung und ist somit konstant. Diese Berechnung ist zwar nicht physikalisch korrekt, da sie an sich für flüssige Stoffe gilt und Schnee visko-elastisch ist, sie ist aber ein einfacher Weg um ein physikalisch plausibles Ergebnis zu erzielen.

3.4.1 Abbremsen des Objektes

Die ermittelte Kraft wirkt der Bewegungsrichtung des Objektes entgegen, wodurch es zu einer Abbremsung kommt. Die Komprimierung des Schnees durch das Objekt führt zu einer erhöhten Dichte, wodurch eine immer kleiner werdende verdrängte Schneemenge ausreicht, um das Objekt entsprechend zu verlangsamen bzw. zum Stillstand zu bringen. Um das langsamere Abbremsen eines schweren Objektes im Gegensatz zu einem leichten kümmert

sich die (externe) Physiksimulation, die die kinetische Energie $T = \frac{1}{2}mv^2$ mitberücksichtigt ($v = |\boldsymbol{v}|$). Diese wird durch die Widerstände so lange verringert, bis sie 0 ist, was bei einem schweren Objekt entsprechend länger dauert als bei einem gleich schnellen leichten Objekt.

3.4.2 Berechnen der Widerstände aus den Differenzwerten

Neben der Stärke der Gegenkraft, ist die Position, an der diese wirkt, ein entscheidender Punkt. Fällt beispielsweise ein Würfel senkrecht auf eine ebene Schneefläche, wird lediglich die Geschwindigkeit verlangsamt, kollidiert allerdings nur ein bestimmter Teil davon, führt das zu einer Rotation. Wichtig ist dies auch bei waagrechten Bewegungen, da auch diese Verdrängung zu einer Abbremsung des Körpers führt. Abbildung 3.4 zeigt diese drei Beispiele.

Die Berechnung der Gegenkraft an sich erfolgt für das gesamte Objekt auf einmal, wobei zuvor die verdrängte Masse

$$m = \sum_{x=x_{\min}}^{x_{\max}} D(x) \cdot \rho(x). \tag{3.6}$$

ermittelt werden muss. $x_{\min}$ und $x_{\max}$ kennzeichnen die Grenzen des Objektes, da lediglich an jenen Stellen eine Überprüfung stattfinden muss, an denen sich das Objekt auch befindet. Zusätzlich muss auch errechnet werden, an welcher Position p die Gegenkraft wirkt. Hierfür wird der Durchschnitt aller Position (gewichtet mit der verdrängten Schneemenge) ermittelt. Zu beachten ist allerdings, dass die Kraft immer von unten auf das Objekt wirkt. Für die Position x_0 an der die Kraft wirkt, ergibt sich also

$$x_0 = \frac{1}{m} \cdot \sum_{x=x_{\min}}^{x_{\max}} x \cdot D(x) \cdot \rho(x). \tag{3.7}$$

Eine grafische Repräsentation findet sich in Abbildung 3.5. Die Berechnung im 3D-Raum erfolgt ähnlich, es muss lediglich die gesamte Grundfläche des Objektes bei der Berechnung berücksichtigt werden (2 Dimensionen statt einer). Für die Begrenzung wird eine Bounding Box verwendet. Die Gegenkraft

$$\boldsymbol{F} = m \cdot (-\boldsymbol{v}) \tag{3.8}$$

ergibt sich aus der errechneten verdrängten Masse und der Geschwindigkeit des Körpers, wobei die Kraft entgegen der Bewegung des Körpers wirkt. Eine Anwendung von $\boldsymbol{F}$ an der Position x_0 auf das Objekt, führt zu dem Ergebnis, dass das Objekt auf den verdrängten Schnee reagiert.

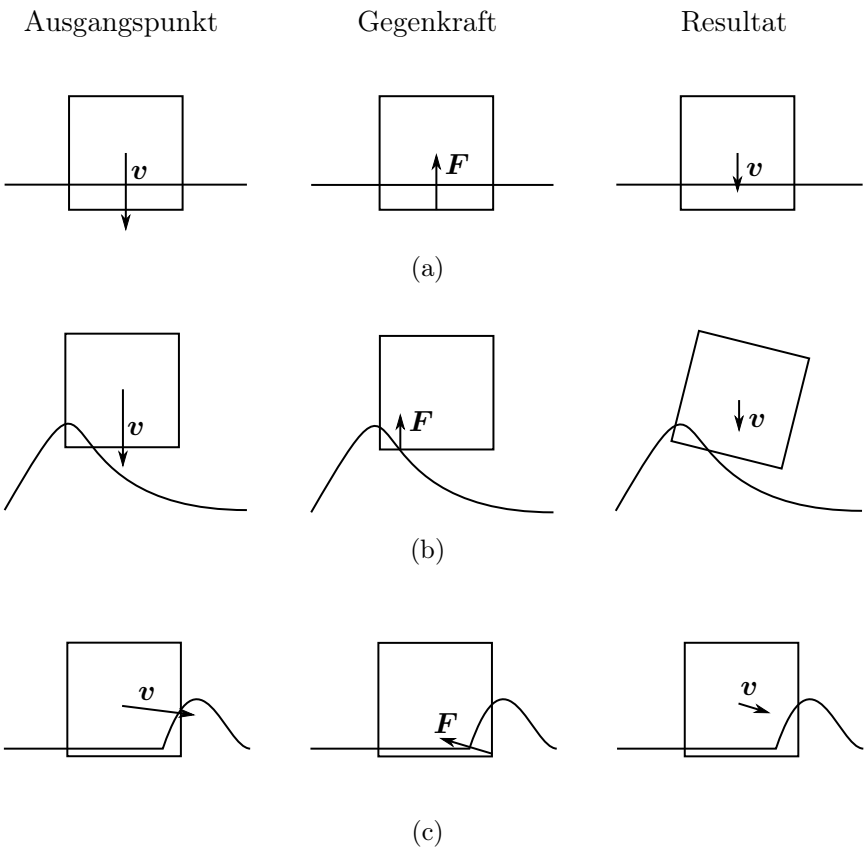


Abbildung 3.4: Kollissionsabhängige Widerstände. Je nach Bewegungsrichtung wirken die Kräfte unterschiedlich auf den Körper. Bei einer senkrechten Bewegung mit gleichmäßigen Verdrängung wirkt die Kraft überall gleich stark entgegen, wodurch lediglich eine Abbremsung erfolgt(a). Einseitige Kollissionen führen zu einer Rotationsbewegung(b). Beinahe waagrechte Bewegungen wirken ebenfalls lediglich der Bewegungsrichtung entgegen(c).

3.5 Komprimieren des Schnees

3.5.1 Ermitteln des Komprimierungs-/Verteilungsfaktors

Je nach verdrängter Schneemenge und deren Dichte, wird der Schnee entsprechend verteilt und komprimiert. Hierfür werden zwei vordefinierte Dichtewerte $\rho_{\min}$ und $\rho_{\max}$ benötigt. Diese geben an, um welche Art Schnee es sich handelt. Für trockenen Neuschnee beispielsweise liegt die Dichte zwischen 30 und 50 kg pro m^3. Eine umfangreichere Auflistung findet sich in Tabelle 3.1 im Einleitungskapitel, wobei diese Werte lediglich als Richtwerte dienen. $\rho_{\min}$ und $\rho_{\max}$ können je nach gewünschtem Schneeverhalten beliebig verändert werden.

Der Dichtewert kann an jeder Position unterschiedlich sein und ist maß-

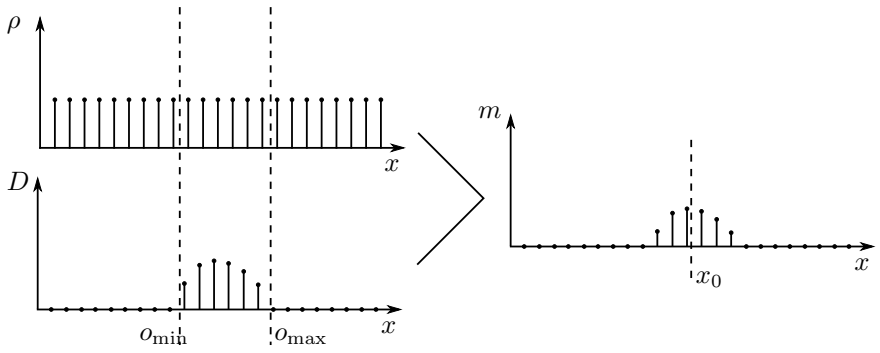


Abbildung 3.5: Positionsermittlung der Krafteinwirkung. Aus dem Produkt der Dichtewerte und der Werte für die verdrängte Masse ergeben sich die gewichteten Positionswerte. Aus deren Durchschnitt kann dann die gesuchte Position ermittelt werden.

geblich dafür verantwortlich, zu welchem Anteil verschoben bzw. komprimiert wird. Je näher sich ein Dichtewert an $\rho_{\max}$ annähert, desto mehr Schnee wird verschoben. Ist die Dichte noch am unteren Limit, wird stärker komprimiert. Somit ergibt sich für den Verteilungsfaktor

$$W(x) = \frac{\rho(x) - \rho_{\min}}{\rho_{\max} - \rho_{\min}} \tag{3.9}$$

und entsprechend für den Komprimierungsfaktor

$$Q(x) = 1 - W(x). \tag{3.10}$$

3.5.2 Komprimiervorgang

Wie stark sich die Dichte an einer bestimmten Position ändert, hängt vom Komprimierungsfaktor und damit verbunden von der zu komprimierenden Schneemenge ab. Zusätzlich wird auch die Schneemenge benötigt, die sich zuvor auf dieser Position befunden hat. Für den neuen Dichtewert ergibt sich also

$$\rho'(x) = \frac{S(x)}{S(x) - D(x) \cdot Q(x)} \cdot \rho(x). \tag{3.11}$$

Diese Komprimierung findet nur an jenen Stellen statt, an denen eine Überschneidung zwischen Objekt und Schnee stattgefunden hat. Im Prinzip müsste sich auch die Verteilung des Schnees auf die Dichte auswirken. Dieser Fall wird in diesem Modell allerdings nicht berücksichtigt. Anstatt den Schnee zusammenzuschieben und dabei zu komprimieren, wird einfach die zu verteilende Schneemenge entsprechend der Dichte zur Schneehöhe der Nachbarfelder addiert. Genaueres findet sich in Abschnitt 3.10.

(a) (b) (c)

Abbildung 3.6: Verteilvorgang. Zuerst wird die verdrängte Schneemenge ermittelt (a), welche anschließend an den Rand des Objektes verschoben wird. Dabei entstehen Säulen (b), deren Schnee in einem späteren Simulationsschritt auf die benachbarten Positionen verteilt wird, wodurch abgeflachte Hänge entstehen (c).

3.6 Verteilvorgang

Ziel des Verteilvorgangs ist es, den Schnee, der sich mit dem Objekt überschneidet, an den Rand des Objekts zu verschieben. Auf diese Weise kommt es zu einer Säulenbildung, wie es in Abbildung 3.6 gezeigt wird. Diese Säulen werden in einem späteren Simulationsschritt abgeflacht. Es gibt mehrere Arten wie diese Verschiebung umgesetzt werden kann, wobei in dieser Arbeit zwei Ansätze beschrieben werden.

Der erste verwendet Distance Transforms[1], wobei es sich dabei um einen iterativen Ansatz handelt. Dabei wird bei jeder Iteration der Schnee von einem Feld zu all den Nachbarfeldern verschoben, die sich näher am Rand befinden. Dieser Vorgang wird so lange wiederholt, bis sich der komplette verdrängte Schnee außerhalb des Objektes befindet.

Beim zweiten Ansatz wird zuerst die Kontur der Überschneidung zwischen Objekt und Schnee ermittelt. Anschließend wird der verdrängte Schnee entlang dieser Kontur aufgeteilt. Der Vorteil dieser Methode im Gegensatz zur Distance Transform Variante ist, dass das Ermitteln der Kontur und im Anschluss das Verteilen keine iterativen Prozesse sind und somit pro Simulationsschritt nur einmal durchgeführt werden müssen.

Bei beiden Ansätzen gilt es jedoch die Bewegungsrichtung des Objektes zu beachten, da eine rein senkrechte Bewegung zu einer anderen Verschiebung führt als eine hauptsächlich waagrechte. Abbildungen 3.6 und 3.7 zeigen Beispiele für gleichmäßige bzw. ungleichmäßige Verschiebung. Um die Verschiebung abhängig von der Bewegungsrichtung des Objektes realisieren zu können, werden entsprechen Richtungsfaktoren ermittelt. Diese können sowohl für die Distance Transform Variante, als auch für die Kontur Variante

[1]Ein Distance Transform entspricht einem „Bild“, in dem die einzelnen Pixelwerte der Entfernung zum nächsten interessanten Pixel entsprechen (z.B. der Abstand zum nächsten Hintergrundpixel). In dieser Arbeit werden Distance Transforms verwendet, um bei Überschneidungen zwischen Objekt und Schnee, die Entfernung zum nächsten nicht von der Überschneidung betroffenen Pixel (Rand des Objektes) zu ermitteln.

(a) (b) (c)

Abbildung 3.7: Verteilvorgang mit Richtung. Im Gegensatz zu der gleichmäßigen Verteilung bei einer rein vertikalen Bewegung wie in Abbildung 3.6, führt eine teils horizontale Bewegung des Objektes (a) zu einer ungleichmäßigen Verteilung (b,c).

eingesetzt werden.

3.7 Berechnen der Richtungsfaktoren

Die Verdrängung des Schnees erfolgt nicht immer gleichmäßig, da sie von der Bewegungsrichtung der Objekte abhängig ist. Um den Schnee entsprechend verschieben zu können, werden Richtungsfaktoren ermittelt, die angeben, wie wahrscheinlich ein Schnee in die entsprechende Richtung verschoben wird. Bei der Verwendung eines Distance Transforms wie in [8] kann mit den Richtungsfaktoren beispielsweise ein gewichtetes Distance Transform erstellt werden.

3.7.1 Berechnung in 2D

Die Berechnung der beiden Richtungsfaktoren (links/rechts) erfolgt zweigeteilt. Dabei werden zuerst Richtungs- und Restanteil ermittelt. Ersterer gibt an, wie sehr die Richtung, für die der Faktor ermittelt wird, der Bewegungsrichtung des Objektes entspricht. Der Linksanteil ist zum Beispiel bei einer Bewegung $\boldsymbol{v} = (-4, 1)^T$ höher als bei $\boldsymbol{v} = (-4, 4)^T$. Der Restanteil ist zuständig um die vertikale Bewegung des Objektes zu berücksichtigen. Je höher der Anteil der vertikalen Bewegung ist, desto gleichmäßiger ist auch die Verteilung. Diese Gleichmäßigkeit wird durch eine Aufteilung des Restanteils auf beide Richtungen erreicht. Der Richtungsanteil

$$d = \hat{\boldsymbol{v}} \cdot \boldsymbol{v}_x \tag{3.12}$$

wird aus dem innerem Produkt des normalisierten Richtungsvektors $\hat{\boldsymbol{v}}$ mit sich selbst, jedoch ohne y-Komponente,

$$\boldsymbol{v}_x = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \hat{\boldsymbol{v}} = \begin{pmatrix} \Delta_x \\ 0 \end{pmatrix}, \tag{3.13}$$

ermittelt, wodurch sich für den Restanteil

$$r = 1 - d \tag{3.14}$$

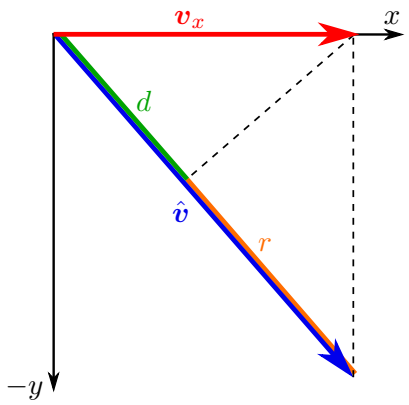


Abbildung 3.8: Ermitteln des Skalierungsfaktors. Das Skalarprodukt aus dem normalisierten Richtungsvektor $\boldsymbol{v}$ ($\hat{\boldsymbol{v}}$) mit sich selbst, jedoch ohne y-Komponente, ($\boldsymbol{v}_{\mathrm{x}}$) ergibt den absoluten Richtungsanteil d. Der verbleibende Restanteil r wird auf alle Richtungen aufgeteilt.

ergibt. In Abbildung 3.8 wird diese Berechnung visualisiert. Im 2D Raum kann eine Verteilung nur in 2 Richtungen stattfinden (links/rechts). Die beiden Richtungsfaktoren dafür,

$$s_{\mathrm{L}} = \begin{cases} \frac{r}{2} + (1-r) & \Delta_x < 0 \\ \frac{r}{2} & \Delta_x \geq 0 \end{cases} \tag{3.15}$$

und

$$s_{\mathrm{R}} = \begin{cases} \frac{r}{2} + (1-r) & \Delta_x > 0, \\ \frac{r}{2} & \Delta_x \leq 0, \end{cases} \tag{3.16}$$

geben an, wie sehr die Richtung der Bewegungsrichtung des Objektes entspricht. Der Wert 1 entspricht dabei einer optimalen und 0 überhaupt keiner Übereinstimmung. $1/s_{\mathrm{L}}$ bzw $1/s_{\mathrm{R}}$ können beispielsweise für die Gewichtung eines Distance Transform verwendet werden. Im Falle einer waagrechten Bewegung in gegengesetzter Richtung würde das zu einer Division durch 0 führen, was bedeutet, dass überhaupt kein Schnee in diese Richtung verschoben wird. In Anhang A.1 werden Beispiele gezeigt, wie sich die Bewegungsrichtung des Objektes auf ein Distance Transform auswirkt.

3.7.2 Berechnung in 3D

Im 3D Raum ist die Berechnung ein wenig komplizierter, da die Verschiebung nicht mehr nur nach links und rechts passiert, sondern auf einer Ebene, wodurch es im Prinzip unendlich viele Richtungen gibt. Die Unterteilung in

Richtungs- und Restanteil erfolgt allerdings ähnlich wie bei der 2D-Variante. Der Restanteil

$$r = 1 - \hat{\boldsymbol{v}} \cdot \boldsymbol{v}_{xz} \tag{3.17}$$

wird auch in 3D aus dem Skalarprodukt zwischen normalisierten Richtungsvektor $\hat{\boldsymbol{v}}$ mit sich selbst, jedoch ohne y-Anteil,

$$\boldsymbol{v}_{xz} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \hat{\boldsymbol{v}} = \begin{pmatrix} \Delta_x \\ 0 \\ \Delta_z \end{pmatrix}, \tag{3.18}$$

ermittelt. Der Richtungsanteil hängt in 3D jedoch von der Nähe zur Optimalrichtung ab (Abbildung 3.9). Auch hier bietet sich der Einsatz des Skalarproduktes an. Projiziert man $\boldsymbol{v}_{\mathrm{xz}}$ auf die Zielrichtung $\boldsymbol{w}$, erhält man als Ergebnis den Wert der Übereinstimmung. Je größer das entsprechende Ergebnis ist, desto besser ist die Übereinstimmung der beiden Richtungen. Wichtig ist dabei allerdings, dass die beiden Vektoren normalisiert sind. Für den Richtungsfaktor im 3D-Raum ergibt sich also

$$s_{\mathrm{w}} = \max(0, \hat{\boldsymbol{v}}_{\mathrm{xz}} \cdot \hat{\boldsymbol{w}}) + \frac{r}{k}. \tag{3.19}$$

Alle Richtungen, die $\boldsymbol{v}_{\mathrm{xz}}$ entegegenwirken ($\hat{\boldsymbol{v}}_{\mathrm{xz}} \cdot \hat{\boldsymbol{w}} \leq 0$), werden eliminiert. Anschließend wird noch der gewichtete Restanteil addiert, um die Bewegung in y-Richtung zu berücksichtigen, die zu einer gleichmäßigen Verteilung des Schnees rund um das Objekt führt. k kann je nach gewünschten Verhalten gewählt werden (muss jedoch immer größer oder gleich 1 sein), je größer der Wert, desto weniger stark wirkt sich der vertikale Anteil der Bewegung auf die Verschiebung aus. (Bei der Umsetzung wurde ein Wert von 8 gewählt.)

3.8 Verteilen mittels Distance Transform

Eine Möglichkeit, wie diese Richtungsfaktoren bei der Verschiebung verwendet werden können, besteht in gewichteten Distance Transforms. Ausgehend von der Gleichung 3.19 können nun beispielsweise die Matrizen für den Chamfer Algorithmus (Berechnung des Distance Transforms) ermittelt werden. Hierfür wird für die 4 Richtungen (links/recht/oben/unten) die entsprechenden Faktoren errechnet, wobei für k in diesem Fall 4 gewählt wird, da der Restanteil auf 4 Richtungen aufgeteilt werden soll. Die Diagonalen ergeben sich aus diesen Werten, je nachdem welche Art der Vorgehensweise gewählt wird (z.B. euklidische Entfernung oder Manhatten-Distance). In Anhang A.2 werden Beispiele für die Berechnung gezeigt.

3.8.1 Verteilvorgang

Beim Verteilen wird der Schnee mit Hilfe des zuvor berechneten Distance Transforms an den Rand des Objektes verschoben. Dabei handelt es sich um

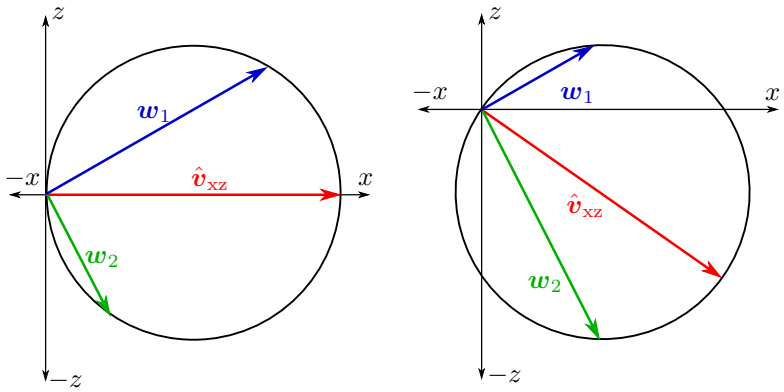


Abbildung 3.9: Berechnung des Richtungsanteils. Je näher sich eine Zielrichtung $(\boldsymbol{w}_1, \boldsymbol{w}_2)$ an der Bewegungsrichtung des Objektes $\hat{\boldsymbol{v}}_{\text{xz}}$ befindet, desto größer ist der Richtungsanteil. Die Länge des Vektors vom Koordinatenurpsrung zum Schnittpunkt mit dem Kreis, entspricht dem Richtungsanteil. Alle Richtungen, die sich um mehr als 90 Grad von der Bewegungsrichtung des Objektes unterscheiden, haben einen Richtungsanteil von 0.

einen iterativen Vorgang, bei dem immer vom höchsten noch nicht behandelten Wert im Distance Transform der verdrängte Schnee an der entsprechenden Stelle auf alle Nachbarfelder mit einem geringeren Wert aufgeteilt wird. Die Komprimierung des Schnees basiert jedoch auf der ursprünglichen Schneemenge, die neu hinzugekommene wird lediglich so lange weiter verschoben, bis sie an den Rand des Objektes gelangt. In Anhang A.3 wird ein Beispiel für den Vorgang gezeigt.

3.8.2 Probleme

Bei der Verschiebung mittels Distance Transform gibt es jedoch einige Probleme, die auch beim Beispiel in Anhang (Anhang A.3) ersichtlich sind. Die resultierende Verschiebung spiegelt nicht das Ergebnis wider, das bei einer Bewegung $\boldsymbol{v} = (2, -1, 3)^T$ zu erwarten wäre. Vor allem im Bereich rechts oben, welcher relativ gut der Bewegungsrichtung des Objektes entspricht, sollten größere Werte vorkommen. Eine Ursache hierfür ist das Verwenden einer 4er Nachbarschaft (keine Verschiebung entlang der Diagonalen). Doch selbst bei einer 8er Nachbarschaft wird kein zufriedenstellendes Ergebnis erreicht. Abbildung 3.10 zeigt das Ergebnis des Beispiels aus dem Anhang mit 8er Nachbarschaft. Auch hier sind die Werte im rechten oberen Bereich zu klein.

Die Werteverteilung zeigt das Problem. Entlang der Horizontalen bzw. Vertikalen sind die Werte weit höher ist als entlang der Diagonalen. Der Grund hierfür liegt darin, dass beim Distance Transform jedes mal, wenn eine Verschiebung entlang einer Diagonalen vorkommt, gleichzeitig auch eine

0,4	1,4	2,4	2,8	2,1	1,1	0,3
0,8	0,0	0,0	0,0	0,0	0,0	0,7
1,2	0,0	0,0	0,0	0,0	0,0	1,4
1,0	0,0	0,0	0,0	0,0	0,0	1,8
0,7	0,0	0,0	0,0	0,0	0,0	1,9
0,4	0,0	0,0	0,0	0,0	0,0	1,1
0,1	0,3	0,4	0,6	0,8	0,7	0,4

Abbildung 3.10: Ergebnis des Verteilvorganges für $\boldsymbol{v} = (2, -1, 3)^{\mathrm{T}}$ mit 8er-Nachbarschaft.

Verschiebung nach links/recht bzw. oben/unten stattfindet. Umgekehrt ist das nicht der Fall. Da der Schnee immer gleichmäßig aufgeteilt wird, führt das bei diagonalen Verschiebungen verhältnismäßig zu geringeren Mengen.

Die relativ häufig vorkommenden Speicherzugriffe (bei der Implementierung), die durch den iterativen Prozess entstehen, sind ein weiterer Nachteil dieser Methode. Diese Zugriffe sind auf der GPU recht kostbar. Aus diesem Grund wurde auch die Kontur-basierte Variante umgesetzt.

3.9 Verteilen mittels Kontur

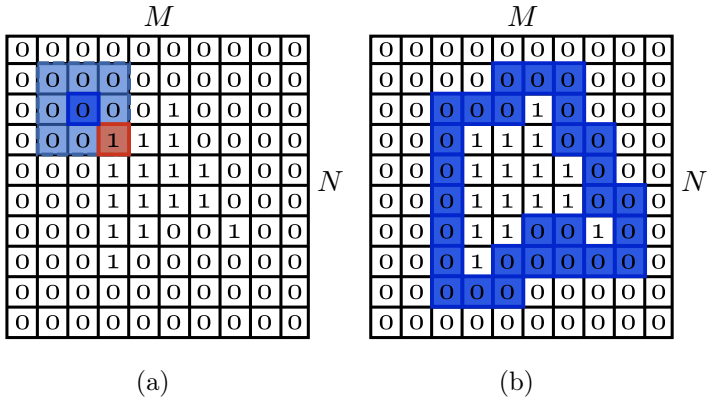
Aufgrund diverser Probleme bei der Verschiebung mittels Distance Transforms (siehe Abschnitt 3.8.2) wird als Alternative eine Kontur-basierte Lösung verwendet. Auf diese Weise bleibt die restliche Vorgehensweise identisch. Die Ermittlung der Kontur basiert auf den Differenzinformationen, die die Überschneidungen kennzeichnen. Da diese Informationen in diskreter Form vorliegen, muss lediglich jedes Feld, das keine Differenz aufweist, mit seinen Nachbarfeldern verglichen werden. Befindet sich darunter zumindest eines mit Differenz, handelt es sich beim aktuellen Feld um ein Konturfeld. Für die Kontur K ergibt sich also

$$K = \{(x, y) \,|D(x, y) = 0 \ \wedge \ L(x, y) \neq 0\}\,, \tag{3.20}$$

wobei

$$L(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} D(x + i, y + i). \tag{3.21}$$

L lieferte dabei die Gesamtdifferenz der 8er-Nachbarschaft. Da D lediglich positive Werte beinhaltet bedeutet $L(x, y) \neq 0$, dass zumindest ein Feld eine Differenz aufweist. Die Unterscheidung zwischen innerer und äußerer Kontur ist in diesem Fall komplett obsolet. Abbildung 3.11 zeigt ein Beispiel für das Ermitteln der Kontur.



(a) (b)

Abbildung 3.11: Für jedes Feld in D, das keine Überschneidung aufweist, werden die 8 Nachbarn auf ein Feld mit einem Wert größer 0 durchsucht (a). Existiert ein solches, wird das aktuelle Feld als Konturfeld gespeichert. Wurde der Vorgang für alle Felder durchgeführt, sind alle Konturen gefunden (b).

Ist die Kontur gefunden, muss der verdrängte Schnee darauf verteilt werden. Hierbei kommen wieder die in Abschnitt 3.7 ermittelten Richtungsfaktoren zum Einsatz. Um die dafür benötigte Richtung zu ermitteln, wird auf einen Kantendetektions-Filter, genauer gesagt den Sobel-Operator, zurückgegriffen. Für die Zielrichtung $\boldsymbol{w}$ ergibt sich

$$\boldsymbol{w} = \begin{pmatrix} w_\mathrm{x} \\ 0 \\ w_\mathrm{y} \end{pmatrix}, \tag{3.22}$$

wobei

$$w_\mathrm{x} = \boldsymbol{S_\mathrm{x}} * D = \begin{bmatrix} 1 & 0 & \text{-}1 \\ 2 & 0 & \text{-}2 \\ 1 & 0 & \text{-}1 \end{bmatrix} * D, \tag{3.23}$$

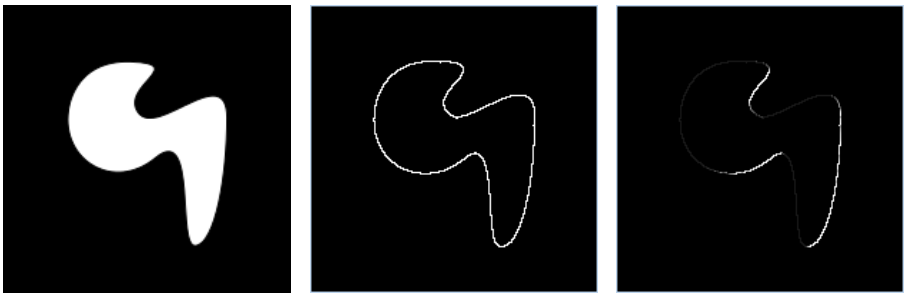
$$w_\mathrm{y} = \boldsymbol{S_\mathrm{y}} * D = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ \text{-}1 & \text{-}2 & \text{-}1 \end{bmatrix} * D. \tag{3.24}$$

Um nun die Schneemenge zu ermitteln, die zu dem Konturfeld addiert wird, muss zuerst die Gesamtmenge des verdrängten Schnees

$$s_\mathrm{T} = \sum_{u=0}^{M} \sum_{v=0}^{N} D(u,v) \tag{3.25}$$

ermittelt werden. Anschließend muss die Gesamtgewichtung der Kontur

$$k_\mathrm{T} = \sum_{i=0}^{n} s_w\big(\boldsymbol{w}(n)\big) \tag{3.26}$$



Differenz D $\quad \boldsymbol{v} = (0, -1, 0)^T$ $\quad \boldsymbol{v} = (3, -3, 2)^T$

Abbildung 3.12: Ergebnisse der Kontur-basierten Variante. Bei einer rein vertikalen Bewegung erfolgt die Verteilung des Schnees gleichmäßig rund um das Objekt (mitte). Bei einer teilweise waagrechten Bewegung wird der Großteil des Schnees in Bewegungsrichtung verschoben (rechts).

berechnet werden, die sich aus der Summe der Richtungsfaktoren zusammensetzt. Die Richtungen

$$\boldsymbol{w}(n) = \begin{pmatrix} w_x\big(K(n)\big) \\ 0 \\ w_y\big(K(n)\big) \end{pmatrix} \tag{3.27}$$

ergeben sich aus den mittels Kantenfilter errechneten Werten für die Konturfelder. Somit kann ermittelt werden, wie viel Schnee auf dem entsprechenden Feld liegt,

$$S'\big(K(n)\big) = S\big(K(n)\big) + s_\mathrm{T} \cdot \frac{s_w(\boldsymbol{w}(n))}{k_\mathrm{T}}. \tag{3.28}$$

Der Vorteil bei dieser Lösung ist, dass die Ermittlung der Kontur und viele der damit verbundenen Berechnungen (z.B. Berechnung von k_T und s_T) auf einmal durchgeführt werden können. Das Verteilen des Schnees entlang der Kontur ist auch wesentlich effizienter als das Verteilen bei der Distance Transform Methode, da jedes Konturfeld dafür nur einmal behandelt werden muss. Die Ergebnisse, wie in Abbildung 3.12 gezeigt, spiegeln ebenfalls die gewünschten Resultate recht gut wider. Nachteil dieser Methode ist, dass der verdrängte Schnee entlang der kompletten Kontur verteilt wird, wenn z.B. im rechten unteren Bereich $1\mathrm{m}^3$ Schnee verdrängt wird und links oben $5\mathrm{m}^3$ führt das zur selben Verteilung wie wenn die verdrängten Mengen vertauscht wären. Bei der Distance Transform Methode würde das berücksichtigt werden.

3.10 Berichtigen zu steiler Hänge

Nachdem der Schnee immer an die nächste freie Stelle verschoben wird, bilden sich rund um das Objekte hohe Säulen/Wände, die entsprechend ab-

geflacht, sprich auf die anliegenden Felder verteilt werden müssen. Für deren Beseitigung wird auf die Methode in [5] zurückgegriffen. Dabei wird die Schneehöhe jedes Feldes mit der der anliegenden Felder verglichen und die Höhendifferenz ermittelt. Ist diese zu groß, wird Schnee von der höheren Stelle auf die anliegenden (niedrigeren) geschoben. Dieser Vorgang wird so lange wiederholt, bis keine zu großen Unterschiede mehr existieren.

Bei der Überprüfung der Schneehöhen, wird immer ein Feld a mit seinem Nachbarn b verglichen. Sollte die Differenz höher als ein festgelegter Maximalwert $\phi_{\max}$ sein, wird Schnee von den höheren auf die niedrigeren Felder verteilt. Die zu verschiebende Schneemenge hängt vom Unterschied

$$d_{\mathrm{n}}(a,b) = \max(H(a) - H(b) - \phi_{\max}, 0) \tag{3.29}$$

zwischen den Feldern a und b, der Summe der zu großen Unterschiede von einem Feld zu seinen Nachbarn

$$c(a) = \sum_{i=a_{\mathrm{x}-1}}^{a_{\mathrm{x}}+1} \sum_{j=a_{\mathrm{y}-1}}^{a_{\mathrm{y}}+1} d_{\mathrm{n}}\left(a, \{i,j\}\right), \tag{3.30}$$

der Anzahl der Nachbarfelder mit zu hoher Differenz ($d_{\mathrm{n}}(a,b) > 0$) n und der Anzahl an niedrigeren Nachbarn ($H(a) > H(b)$) m ab. Wie die Gleichung 3.29 zeigt, entspricht die zu verteilende Menge nur jenem Schnee, der die maximale Differenz übersteigt. Die zu verschiebende Schneemenge von a nach b

$$s(a,b) = \begin{cases} c(a)/n/m & \text{für } H(a) > H(b), \\ 0 & \text{für } H(a) \leq H(b), \end{cases} \tag{3.31}$$

errechnet sich aus der durchschnittlichen Schneemenge, die die maximale Differenz übersteigt. Diese wird gleichmäßig auf alle niedrigeren Nachbarfelder verteilt.

Ein Punkt, der bei dieser Berechnung nicht beachtet wird, ist die Dichte. Eine Berücksichtigung dieser bei der Berechnung von d könnte die Qualität der Simulation durchaus verbessern. Für die hier verwendete Vorgehensweise, wird lediglich sichergestellt, dass sich beim Verschieben die Gesamtmenge des Schnees nicht verändert. Die neue Schneehöhe an der Position b

$$S'(b) = S(b) + \frac{s(a) \cdot \rho(a)}{\rho(b)} \tag{3.32}$$

wird ausgehend von den Dichtewerten der beiden Felder a und b berechnet. Für die Höhe bei a ergibt sich

$$S'(a) = S(a) - \frac{c(a)}{n}. \tag{3.33}$$

Das Ergebnis dieses Verschiebe-Vorgangs wird in Abbildung 3.13 gezeigt, wobei als Ausgangslage die Konturen der Abbildung 3.12 verwendet wurden.

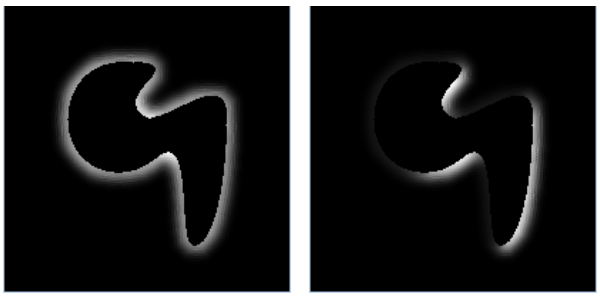


Abbildung 3.13: Ergebnisse nach dem Abflachen der zu steilen Hänge für die Ergebnisse aus Abbildung 3.12.

Kapitel 4

Implementierung

4.1 Programmaufbau

4.1.1 Architektur

Bei der Implementierung wird auf C++ in Verbindung mit DirectX 11 zurückgegriffen. Für die Architektur wird ein komponentenbasiertes System eingesetzt, mit der die Physik (`PhysicsBody`) von der grafischen Repräsentation (`Renderable`) entkoppelt wird. Die `PhysicsBody`- bzw. die `Renderable`-Objekte werden in entsprechenden Managern verwaltet, die sich um das Hinzufügen, Updaten und Entfernen der Komponenten kümmern. Somit ist es möglich, jedem Rendervorgang (`RenderPass`) alle zu rendernden Objekte zu übergeben. Diese `RenderPass`-Klassen können dann auf die Objekte unterschiedlich reagieren, was bei der Simulation notwendig ist (Schnee unterscheidet sich von den Objekten). Eine grobe Übersicht über die verwendeten Klassen findet sich im Klassendiagramm in Abbildung 4.1.

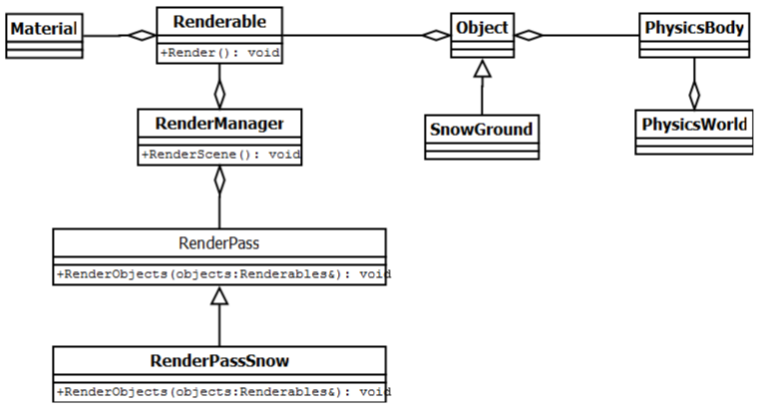


Abbildung 4.1: Klassendiagramm der Hauptbestandteile des Programmes.

4.1.2 Objekte

Bei der Umsetzung gibt es einige verschiedene Objekte, die zum Teil für die Datenspeicherung und zum Teil für die Simulation selbst zuständig sind. Einige der hier verwendeten Begriffe werden im Einführungskapitel in Abschnitt 1.6.2 erklärt.

Schneeboden

Die Schneeboden-Klasse `SnowGround` besitzt zwei Unordered Access Views (UAVs), die die Werte für die Schneehöhe und die Dichte verwalten. Diese beiden UAVs können mittels HeightMap-Texturen initialisiert werden, wodurch die Ausgangssituation der Simulation recht einfach definiert werden kann. Ein weiterer Bestandteil der `SnowGround`-Klasse ist das Material zum Rendern des Bodens, welches die Shader für die Tessellierung beinhaltet.

Objekt

In der Simulation kann es beliebig viele Objekte geben, die sich allesamt über ein `Renderable` und einen `PhysicsBody` definieren. Diese Objekte können jede beliebige Form annehmen, die entweder im Code definiert wird oder über .o Files eingelesen werden kann. Dabei muss sichergestellt werden, dass die grafische und physikalische Repräsentationen übereinstimmen. Die Positionierung/Steuerung kann entweder über Benutzereingaben direkt erfolgen oder über die Simulation einer externen Physiksimulation.

`Renderable`

Die `Renderable`-Klasse ist für die grafische Repräsentation zuständig. Diese besteht einerseits aus den geometrischen Informationen (InputLayout, Vertexbuffer und Indexbuffer) und andererseits aus Materialeigenschaften (Shader die für das Rendern verwendet werden, Texturen, etc.). Die Initialisierung der geometischen Informationen kann entweder über entsprechende abgeleitete Klassen im Code definiert werden oder über das Einlesen von .o Files erfolgen.

Alle erzeugten `Renderable`-Objekte werden im `RenderManager` verwaltet. Dieser ist auch für das Abarbeiten der einzelnen `RenderPass`-Objekte zuständig, die ihrerseits beim Rendern immer alle aktiven `Renderable`-Objekte übergeben bekommen. Auf diese Weise können relativ einfach individuelle Renderpasses erstellt werden, die beispielsweise für das Rendern einer Shadow Map verwendet werden können oder wie im Falle dieser Arbeit für eine Simulation.

`PhysicsBody`

Die `PhysicsBody`-Klasse ist für die physikalische Repräsentation und Steuerung der Objekte zuständig. Für die in dieser Arbeit beschriebene Schneesimulation ist an sich nur jene Funktionalität wichtig, dass auf das Objekt eine Kraft einwirken kann (Gegenkraft durch den verdrängten Schnee)und dass die Geschwindigkeit des Objektes abgefragt werden kann. Da bei der Schneesimulation in dieser Arbeit, lediglich Schnee/Objekt Interaktionen behandelt werden, bietet sich eine externe Physikengine ohnehin für die restlichen Berechnungen (Kollissionen, Schwerkraft, etc.) an.

`RenderPassSnow`

Diese Klasse ist das eigentliche Herzstück der Simulation. Wie bei den anderen Renderpässen werden alle `Renderable`-Objekte übergeben. Aus denen müssen zuerst alle relevanten Objekte wie Boden und bewegliche Objekte ermittelt werden. `Renderable`-Objekte wie Overlays können zum Beispiel ignoriert werden, da sie sich nicht auf die Simulation auswirken. Sind alle benötigten Objekte gefunden, wird die eigentliche Simulation durchgeführt (Überschneidungen ermitteln, Widerstände berechnen, usw.).

4.1.3 Implementierungsdetails

Bei der Umsetzung wurden einige Annahmen getroffen bzw. sind diverse Workarounds notwendig, um die verwendeten Algorithmen auf der GPU zum Laufen zu bringen.

Höhenbereiche

Bei den Höhenwerten gilt es zu beachten, dass diese im Bereich von 0 bis 1 sind, wobei 0 die Minimalhöhe bedeutet und 1 entspricht der maximalen Höhe $s_{\max}$, welche vorab definiert werden muss. Diese gibt an, wie hoch der Schnee (inkl. Bodenhöhe) maximal sein kann. Dieser Wert kann zwar im Prinzip beliebig gewählt werden, da er lediglich als Skalierungsfaktor für die Höhenwerte dient, es gilt jedoch zu berücksichtigen, dass für den gesamten Bereich nur eine bestimmte Anzahl an Bits (je nachdem wie genau die Höhen gespeichert werden) zur Verfügung steht. Werden pro Position beispielsweise nur 8 Bits verwendet (256 verschiedene Werte) können für $s_{\max} = 4$ (Meter) nur Höhenuterschiede von rund 1,5 cm dargestellt werden. Diese Einschränkung der Höhe führt ebenfalls dazu, dass nicht alle Objekte in der Simulation berücksichtigt werden müssen, da Objekte die sich über der maximalen Schneehöhe befinden, unmöglich mit dem Schnee in Kontakt kommen können. Eine Höhe kleiner 0 sollte für Objekte ohnehin unmöglich sein.

Buffer für Schneehöhe und Dichte

Die Buffer für die Schneehöhe und -dichte werden im `SnowGround`-Objekt angelegt und initialisiert. Die Größe der Buffer kann beliebig gewählt werden, es ist jedoch zu beachten, dass diese in Kombination mit dem Kamerasetup (siehe Abschnitt 4.1.3) maßgeblich die Qualität des Ergebnisses und die Performance beeinflussen. Je größer der Buffer, desto genauer ist auch die Unterteilung des Simulationsbereichs. Ein größerer Buffer verursacht aber auch einen weit höheren Rechenaufwand, wodurch ein passender Kompromiss zwischen Qualität und Performance gefunden werden muss.

Skalierung der Bufferwerte

Als Datentyp für die Buffer wird `unsigned int` verwendet. Die Gründe hierfür liegen einerseits in der erhöhten Performance beim Rechnen mit Variablen vom Typ`uint` auf der GPU und anderseits bei der Limitierung der atomaren Operationen, nicht mit Gleitkommavariablen rechnen zu können. Da diese atomaren Operationen aber des öfteren in den Compute Shadern zum Einsatz kommen, dürfen die zugrunde liegenden Werte keine Gleitkommavariablen sein. Das hat natürlich zur Folge, dass die Höhen-/Differenzwerte skaliert werden müssen, da beispielsweise die ermittelten Höhenwerte (siehe Gleichung 4.4) im Bereich zwischen 0 und 1 sind. Für diese Skalierung wird s_f, was bei der Implementierung der Konstante `INT_TO_FLOAT_FACTOR` entspricht, verwendet.

Simulationsbereich

Der Simulationsbereich hängt von der Kameraposition und -projektion ab, die verwendet werden, um die Höhe der Unterkanten der Objekte U zu ermitteln (siehe Abschnitt 4.5.2). Die Position bestimmt das Zentrum der Simulation und die Dimensionen der orthographischen Projektion legen fest, wie viele Einheiten in die entsprechenden Richtungen in der Simulation berücksichtigt werden. Wichtig ist dabei allerdings, dass die verwendeten Buffer (Schneehöhe, -dichte, etc.) der Größe des Rendertargets und dem Verhältnis der Dimensionen der Projektion entsprechen. Sind die Buffer beispielsweise quadratisch sollte auch die Projektion sowohl in x- als auch in z-Richtung gleich groß sein. Die Buffer und die Projektion bestimmen auch sehr stark das Ergebnis der Simulation. Wird beispielsweise ein $1\mathrm{m}^2$ Feld von einem 100 mal 100 Einheiten großen Buffer abgedeckt, kann jeder cm^2 dargestellt werden, wird der selbe Buffer jedoch für ein $100\mathrm{m}^2$ Feld verwendet, sinkt die Genauigkeit schon enorm.

Kopier-Kamera

An manchen Stellen im Programm ist es erforderlich, Informationen von einem Buffer in eine Textur (oder umgekehrt) zu schreiben. Da in diesen Fällen auch des öfteren noch diverse Berechnungen hinzukommen, gibt es eine „Kopier-Kamera“ die es ermöglicht, einen Pixel Shader aufzurufen, bei dem jeder Pixel einem Wert im Buffer entspricht. Wichtig ist dabei, dass die Größe des Rendertargets genau den Ausmaßen des entsprechenden Buffers entspricht. Die Kamera ist so positioniert, dass eine Hilfsplane kamerafüllend gerendert wird, wodurch aus den Texturkoordinaten die Position im Buffer ermittelt werden kann. Der Up-Vektor der Kamera ist dabei für die Interpretation der Texturkoordinaten entscheidend, da dieser im Prinzip regelt, wo oben/unten/links/rechts ist. Für die Ermittlung des Buffer Index wird die Funktion `GetIndexFromTexCoord()` verwendet.

```
1 uint GetIndexFromTexCoord( float2 texCoord )
2 {
3   uint row = ( SNOW_TEXTURE_WIDTH - 1 ) - floor( texCoord.y * (
      SNOW_TEXTURE_WIDTH - 1 ) );
4   uint col = floor( texCoord.x * ( SNOW_TEXTURE_WIDTH - 1 ) );
5   return row * SNOW_TEXTURE_WIDTH + col;
6 }
```

4.2 Compute Shader Verwendung

Für einige der Aufgaben, die im Laufe der Simulation anfallen, werden Compute Shader eingesetzt. Die Grundsätzliche Verwendung, wurde bereits kurz im Einleitungskapitel in Abschnitt 1.6.2 beschrieben. Oft wird beim Einsatz von Compute Shadern ein kompletter Buffer abgearbeitet. Hierfür wird immer das gleiche Grundkonstrukt verwendet, welches den Buffer entsprechend der Thread-Gruppen und Threads innerhalb der Gruppen unterteilt, sodass jeder Thread einen kleinen Bereich zum Abarbeiten hat. In Abbildung 4.2 wird ein Beispiel für die Aufteilung eines Buffers auf 4 Gruppen zu je 64 Threads veranschaulicht, wobei die Farben den einzelnen Gruppen entsprechen und die Abstufungen den einzelnen Threads.

Um einen Buffer aufteilen zu können, müssen dessen Dimensionen M und N bekannt sein, die Anzahl der Thread-Gruppen in x- und y-Dimension (G_x und G_y) und die Anzahl der Threads innerhalb einer Gruppe in x- und y-Dimension (T_x und T_y). Aus diesen Informationen kann die Anzahl der Felder, die jeder Thread entlang der einzelnen Achsen bearbeiten muss,

$$T_\mathrm{w} = \frac{M}{T_\mathrm{x} \cdot G_\mathrm{x}} \quad \text{und} \quad T_\mathrm{h} = \frac{N}{T_\mathrm{y} \cdot G_\mathrm{y}}, \tag{4.1}$$

errechnet werden. Als letztes fehlt noch die Startposition, an denen ein Thread seine Arbeit beginnen muss. Hierfür bekommt er die ID der Gruppe

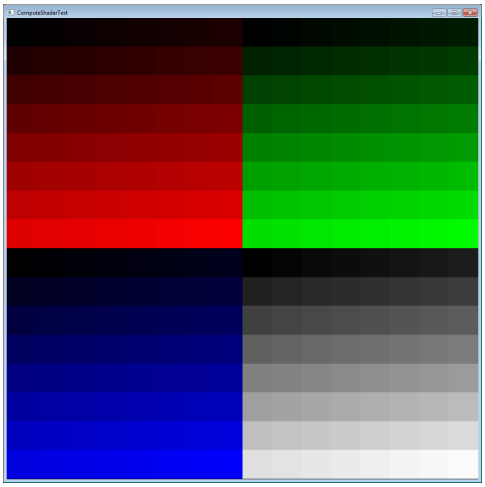


Abbildung 4.2: Aufteilung eines Buffers auf die Threads bzw. Thread-Gruppen. Die Farben kennzeichnen die vier Gruppen und die Abstufungen die 64 Threads innerhalb jeder Gruppe.

G_I und die ID innerhalb der Gruppe T_I mitübergeben. Diese bestehen beide aus 3 `uint` Werten für die einzelnen Dimensionen. Aus ihnen können dann die Startwerte

$$S_\mathrm{x} = (G_\mathrm{I}.x \cdot G_\mathrm{x} + T_\mathrm{I}.x) \cdot T_\mathrm{w}, \tag{4.2}$$
$$S_\mathrm{y} = (G_\mathrm{I}.y \cdot G_\mathrm{y} + T_\mathrm{I}.y) \cdot T_\mathrm{h}, \tag{4.3}$$

ermittelt werden. Im Algorithmus 4.1 wird gezeigt, wie der Ablauf im Allgemeinen funktioniert. Eine einfache Umsetzung davon, findet sich in Anhand B.1. Dieses Beispiel führt zur Ausgabe, die in Abbildung 4.2 zu sehen ist.

4.3 Initialisierung

Sowohl vor dem Start der Simulation, als auch zwischen den einzelnen Simulationsschritten müssen einige Vorkehrungen getroffen bzw. Änderungen vorgenommen werden.

4.3.1 Schneeboden

Um die gewünschte Ausgangssituation zu erreichen, müssen zu Beginn alle Objekte initialisiert werden, dazu gehört auch der Schneeboden. Dessen beide Buffer für die Schneehöhe und -dichte werden zu Beginn über eine Textur initialisiert. Hierfür wird ein Graustufenbild verwendet, wobei wichtig ist, dass diese Werte (zwischen 0 und 1) entsprechend dem Skalierungsfaktor s_f (Abschnitt 4.1.3) auf `unsigned int` Variablen umgewandelt werden. Um das

Algorithmus 4.1: Grundkonstrukt eines Compute Shaders zur Abarbeitung eines Buffers I.

```
1:  ComputeShaderFunction(I, M, N, G_x, G_y, T_x, T_y, G_I, T_I)
        Hierbei handelt es sich um das Grundkonstrukt eines Compute Shaders, der zur Bearbeitung eines Buffers eingesetzt wird.
2:      T_w ← M / (T_x · G_x)
3:      T_h ← N / (T_y · G_y)
4:      S_x ← (G_I.x · G_x + T_I.x) · T_w
5:      S_y ← (G_I.y · G_y + T_I.y) · T_h
6:      for i = 0 to T_w do
7:          for j = 0 to T_h do
8:              DoSomethingWith(I(S_x + i, S_y + j))
9:          end for
10:     end for
11: end
```

zu bewerkstelligen, wird mittels Kopier-Kamera die UAV des Buffers und die Textur an einen Pixel Shader übergeben, welcher anschließend die Werte der Textur entsprechend umrechnet und in den `RWStructuredBuffer` der UAV schreibt. Dieser Pixel Shader ist denkbar einfach:

```
1 float PShaderCopyHeightFromTexture( PS_INPUT input ) : SV_Target
2 {
3   g_OutBuff[ GetIndexFromTexCoord( input.texCoord ) ].value =
4             g_Texture.Sample( g_SampleLinear, 1 - ( input.texCoord ) ).x
5             * INT_TO_FLOAT_FACTOR;
6 }
```

Ein weiterer Punkt, der bei der Initialisierung des Schnees wichtig ist, betrifft die Auflösung des Meshes für den Schnee. Ein höher aufgelöstes Mesh ermöglicht in der Regel mehr Details, wobei somit auch der langsame CPU–GPU Transfer stärker beansprucht wird. Außerdem gilt es zu berücksichtigen, dass das Mesh später noch auf der Hardware tesselliert wird, wodurch eine Auflösung in Buffer-Größe unnötig ist und die Performance lediglich drücken würde. Beim Testbeispiel wird eine Plane verwendet, die aus 100 Vertices (10 mal 10) besteht, welche nach der Tessellierung auch für einen Buffer mit 1024 mal 1024 Elementen ausreichend war.

4.3.2 Zwischen Simulationsschritten

Einige Werte müssen am Beginn jedes Simulationsschrittes initialisiert werden. Hierzu gehören die diversen Rendertargets inklusive derer Tiefenbuffer, welche beispielsweise beim Ermitteln der Unterkanten der Objekte verwendet werden (siehe Abschnitt 4.5.2).

```
1 //neu initialisieren des Tiefenbuffers
```

```
2 DirectX::GetInstance().GetDeviceContext()->ClearDepthStencilView(
      m_pDepthStencilView, D3D10_CLEAR_DEPTH, 1.0, 0 );
3
4 //clearen des RenderTargets
5 DirectX::GetInstance().GetDeviceContext()->ClearRenderTargetView(
      m_pObjectsDepthRenderTarget, D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f ) );
```

Des Weiteren müssen auch die UAVs, in denen die Differenzwerte und die Unterkanten der Objekte gespeichert werden, neu initialisiert werden. Das hat den Grund, dass sich bei diesen nur jene Felder ändern, an deren Positionen sich Objekte befinden. Nachdem sich diese bewegen, sind die betroffenen Felder natürlich in jedem Simulationsschritt unterschiedlich.

```
1 UINT values[ 4 ];
2 // clearen des Differenzbuffers
3 memset( values, 0, sizeof( UINT ) * 4 );
4 DirectX::GetInstance().GetDeviceContext()->ClearUnorderedAccessViewUint(
      m_pDifferenceUAV->GetUnorderedAccessView(), values );
5
6 // clearen des Buffers für die Unterkanten der Objekte
7 memset( values, -1, sizeof( UINT ) * 4 );
8 DirectX::GetInstance().GetDeviceContext()->ClearUnorderedAccessViewUint(
      m_pObjectHeightUAV->GetUnorderedAccessView(), values );
```

Wie dieser Code zeigt, müssen die Werte des Differenzbuffers auf 0 gesetzt werden (standardmäßig keine Differenz) und die Werte für den Buffer der Unterkanten mit dem Maximalwert, was am Einfachsten mit `UINT` $= -1$ erreicht wird. Das hat den Grund, da an den Positionen, an denen sich kein Objekt befindet, $U = \infty$ ist, wobei natürlich der höchste mögliche Wert dem am Nächsten kommt.

4.4 Renderpass

Der Renderpass, welcher für die Simulation des Schnees zuständig ist, wird in jedem Renderdurchlauf einmal aufgerufen und bekommt dabei alle aktiven `Renderable` Objekte mitübergeben. Die Klasse `RenderPassSnowObjects` ist demnach das Herzstück der Simulation.

4.4.1 Bestandteile der Klasse

Die Klasse beinhaltet einige Membervariablen, die für die Simulation benötigt werden. Dazu gehören

- Constant Buffer[1] für Boundingbox und Geschwindigkeit des Objektes,
- Compute Shader für die Berechnungen,
- UAVs zur Datenspeicherung und

[1] Constant Buffers sind Buffer, die an den Shader übergeben werden, wobei der Zugriff lediglich lesend erfolgen kann.

- Elemente der Kopier-Kamera.

Der Constant Buffer für die Boundingbox sorgt dafür, dass der Simulationsbereich eingeschränkt werden kann (da manche Operationen nur dort ausgeführt werden müssen, an denen sich tatsächlich ein Objekt befindet). Der Buffer für die Geschwindigkeit wird für die Berechnung der Richtungsfaktoren benötigt. Die vorhandenen Compute Shader (z.B. zur Ermittlung der Kontur) werden für die einzelnen Simulationsschritte benötigt (deren Beschreibung erfolgt in den entsprechenden Abschnitten). Zu den UAVs gehört jene, die zur Berechnung der Kontur verwendet wird und die beiden zur Speicherung der Objekthöhe und der Differenz.

4.4.2 `RenderObjects`–Methode

Diese Methode ist die Schnittstelle zum restlichen Programm, wobei sie alle `Renderable`-Objekte R mitübergeben bekommt. Sollte das Objekt für den Schneeboden G noch nicht ermittelt worden sein, muss es zu Beginn der Funktion passieren, da dieser natürlich gesondert von den restlichen Objekten behandelt werden muss. Anschließend kann die Simulation beginnen. Zuerst werden die nötigen Initialisierungen durchgeführt. Anschließend werden für jedes Objekt folgende Schritte durchgeführt:

- Rendern der Objekttiefe inkl. Berechnung der Differenz,
- Ermitteln der Widerstände und
- Verschieben des Schnees an den Rand des Objektes.

Diese Schritte müssen natürlich nur durchgeführt werden, wenn sich das Objekt tatsächlich im Simulationsbereich befindet. Wurden diese Schritte für alle Objekte durchgeführt, wird anschließend für die gesamte Szene die Berichtigung der Hügel durchgeführt. Hierfür werden jedoch die Positionen aller Objekte benötigt, welche sich aber automatisch im entsprechenden Buffer befinden, da ja die UAV nicht bei jedem Objekt zurückgesetzt wird, sondern immer nur am Beginn der Methode. Der genaue Ablauf ist im Algorithmus 4.2 gezeigt.

Simulationsbereich

In der Szene können sich Objekte befinden, die außerhalb des Simulationsbereichs sind (Abbildung 4.3). Um die Simulation zu beschleunigen und nicht unnötig Berechnungen durchzuführen, können diese Objekte vernachlässigt werden. Hierfür reicht eine Überprüfung aus, ob sich die Boundingbox des Objektes innerhalb des Simulationsbereichs befindet, welcher sich durch die orthographische Projektion der Kamera und der maximalen Schneehöhe $s_{\max}$ definiert.

Algorithmus 4.2: Hauptmethode der Simulation.

```
1:  RenderObjects(R)
2:      G ← FindGInR(R)
3:      if G = null then
4:          return                ▷ kein Objekt für den Schneeboden gefunden
5:      end if
6:      for all r in R do
7:          if IsInSimulationArea( r ) then
8:              CalculateObjectsDepth( r, G )
9:              CalculateContour( G )
10:             MoveSnowToBorder( G )
11:         end if
12:     end for
13:     FlattenHills( G )
14: end
```

4.5 Ermitteln der Überschneidung

Der erste Schritt in der Simulation ist das Ermitteln der Überschneidungen zwischen dem Schnee und den Objekten. Hierfür wird die Höhe der Schneeoberfläche H und die Höhe der Objektunterkanten U benötigt.

4.5.1 Scheehöhe

Die eigentliche Höhe des Schnees H ist durch das `SnowGround`-Objekt bekannt, da dieses einen Buffer inklusive UAV für die Schneehöhe besitzt. Da bei der Umsetzung in diesem Fall kein Boden existiert, entspricht H gleich S.

4.5.2 Objekthöhe

Die Ermittlung von U gestaltet sich etwas komplizierter. Die Objekte können beliebige Formen besitzen, wodurch auch die Höhe der Unterkanten stark variieren kann. Eine Variante zur Berechnung wird beim Rendern einer Szene eigentlich schon automatisch durchgeführt – Mitführen einer Depth Map, die auch für den Depth Test verwendet wird.

Verwenden des z-Buffers

Für den Depth Test wird der Tiefenwert mitgespeichert, der am nächsten bei der Kamera ist, denn alle Objekte, die sich dahinter befinden, müssen nicht mehr gerendert werden. Hierfür wird in der Regel ein z-Buffer eingesetzt, welcher jedoch im Gegensatz zu einem w-Buffer nicht linear ist. Normalerweise sind bei Renderings nahe Objekte meist größer und wichtiger als kleine

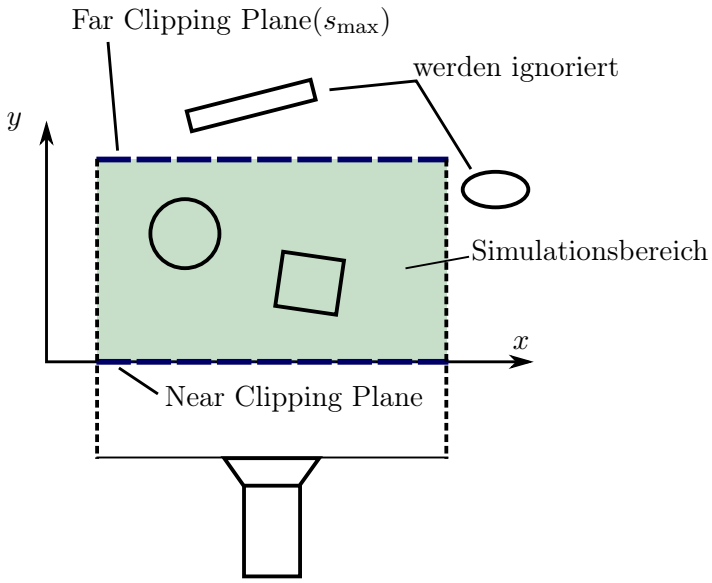


Abbildung 4.3: Kamerasetup bei der Höhenberechnung. Die orthografische Projektion der Kamera definiert den Simulationsbereich. Objekte, die sich außerhalb befinden, werden bei der Simulation nicht berücksichtigt.

Objekte weit weg. Daher wird beim z-Buffer dafür gesorgt, dass in der Nähe der Near Clipping Plane eine genauere Unterscheidung erfolgt als in der Nähe der Far Clipping Plane. Nachdem bei der Simulation aber alle Entfernungen gleich wichtig sind, ist eine lineare Verteilung zu bevorzugen.

Auch wenn sich der z-Buffer selbst also nicht für die Berechnung der Differenz eignet, führt er dennoch dazu, dass nur die „Unterkanten“ der sichtbaren Objekte gerendert werden. Wird im Pixel Shader nun dafür gesorgt, dass nicht die geshadeten Werte zurückgegeben werden, sondern einfach der Abstand zur Kamera, ist schlussendlich U im Rendering gespeichert.

Berechnen der Objekthöhe

Um U ermitteln zu können, muss sich die Kamera unter der Szene befinden und so ausgerichtet sein, dass sie entlang der y-Achse (nach oben) schaut (Abbildung 4.3). Des Weiteren muss eine orthographische Projektion verwendet werden, da keine perspektivischen Verzerrungen vorkommen dürfen. Diese Projektion gibt auch an, welcher Bereich für die Schneesimulation verwendet wird, bzw. wie genau Verformungen dargestellt werden können. Wird beispielsweise eine Projektion gewählt, die entlang der x und z Dimensionen die Länge 1 hat (entspricht 1m^2) wird beispielsweise bei einem 1000 mal 1000 Felder Großen Buffer jeder Millimeter abgebildet.

Für die Höhenberechnung sind die Far und Near Clipping Planes der Kameraprojektion ausschlaggebend, da diese den Bereich definieren, in dem sich Objekte befinden müssen, um von der Berechnung berücksichtigt zu werden. Die Höhe

$$U = \frac{z \cdot (Z_{\mathrm{F}} - Z_{\mathrm{N}})}{s_{\max}} \tag{4.4}$$

ergibt sich aus dem z-Wert des Objektes in Abhängigkeit zu den Positionen der Near Clipping Plane Z_{N} und der Far Clipping Plane Z_{F}. z entspricht im diesem Fall nicht der z-Position des Objektes im Welt-Koordinatensystem, sondern in dem der Kamera. Nachdem diese Berechnung jedoch im Pixel Shader durchgeführt wird, sollten die vorhandenen Koordinaten bereits im Kamera-Koordinatensystem sein. Um den verfügbaren Wertebereich möglichst effizient auszunutzen und Umrechnungen zur Schneehöhe zu vermeiden, bietet es sich an, die Near-Clipping Plane auf die Nullebene/Höhe des Bodens zu setzen und die Far Clipping Plane auf $s_{\max}$. Um nun die Berechnung für die gesamte Fläche durchführen zu können, wird die Höhenberechnung in den Pixelshader verlagert.

Ein weiterer Punkt, der berücksichtigt werden muss, liegt in der Größe des Buffers, in den die Ergebnisse geschrieben werden. Um später die Differenzen aus Schneehöhe und Objekthöhe berechnen zu können, muss sichergestellt werden, dass gleich viele Werte vorhanden sind. Das wird dadurch erreicht, dass die Größe des Buffers für die Objekthöhen jenem der Schneehöhen entspricht und auch das Rendertarget die entsprechenden Ausmaße besitzt. Beim Buffer gilt es außerdem darauf zu achten, dass dieser korrekt initialisiert wird, da der Pixelshader (in dem die Berechnung erfolgt) nur für jene Positionen ausgeführt wird, an denen sich auch tatsächlich Objekte befinden. Ein Befüllen des Buffers mit dem Maximalwert ((UINT)−1) führt dazu, dass an den Positionen ohne Objekt die Höhe $s_{\max}$ angenommen wird. In Abschnitt 4.3.2 finden sich der Code und zusätzliche Informationen dazu.

4.5.3 Berechnen und Speichern der Differenz

Die Differenzberechnung selbst ist grundsätzlich recht einfach, jedoch gilt es dafür zu sorgen, dass der Differenzbuffer gleich groß ist, wie die Buffer für die Objekthöhe und die Höhe der Schneeoberfläche. Der Wert für die Differenz

$$D = \max(0, H - U) \tag{4.5}$$

errechnet sich aus der Differenz aus der Höhe der Schneeoberfläche und Objekthöhe, wobei nur positive Ergebnisse interessant sind (da nur sie eine Überschneidung zwischen Objekt und Schnee kennzeichnen). Da bei der Implementierung jedoch Variablen vom Typ `uint` verwendet werden, muss statt der max Anweisung überprüft werden, ob die Höhe der Schneeoberfläche größer als die des Objektes ist, da max nur negative Ergebnisse verhindern

würde, die es jedoch bei `uint` Variablen nicht gibt. Somit ergibt sich für die Differenz

$$D = \begin{cases} 0 & \text{bei} H \leq U \\ H - U & \text{andererseits.} \end{cases} \tag{4.6}$$

Bei D muss außerdem dafür gesorgt werden, dass es skaliert um s_f gespeichert wird, da nur `unsigned int` Werte gespeichert werden. Da aber sowohl H als auch U bereits skalierte Werte beinhalten sollten, reicht dennoch eine einfache Subtraktion aus. Ein weiteres kleines Detail, das berücksichtigt werden muss, ist jenes, dass der Depth Test per default nach dem Pixel Shader durchgeführt wird, was dazu führen würde, dass dennoch in den Buffer geschrieben wird, obwohl das entsprechende Objekt eigentlich gar nicht gerendert wird. Um das zu umgehen, gibt es die Möglichkeit, den Pixel Shader mit der Anweisung `[earlydepthstencil]` zu versehen, womit der Depth Test bereits vorher ausgeführt wird.

Nachdem die Berechnung von D genauso wie die Berechnung von U für jedes Feld im Buffer erfolgen muss, können diese beiden direkt hintereinander im selben Pixel Shader durchgeführt werden. Im Algorithmus 4.3 wird die Vorgehensweise gezeigt und die genaue Implementierung dieses Pixel Shaders findet sich in Anhang B.2.

Algorithmus 4.3: Berechnen der Objekhöhe U und Differenz D

1: CalculateObjectsDepth(z, x, y, H, U, D)
   Diese Funktion berechnet für einen Tiefenwert z an der Position (x, y) die Differenz und Objekthöhe.
   H ... Buffer für Schneehöhe
   U ... Buffer für Objekthöhe
   D ... Buffer für Differenzen
2: $U(x, y) \leftarrow \frac{z \cdot (Z_\mathrm{F} - Z_\mathrm{N})}{s_\mathrm{max}}$
3: **if** $H(x, y) > U(x, y)$ **then**
4: $D(x, y) \leftarrow H(x, y) - U(x, y)$
5: **else**
6: $D(x, y) \leftarrow 0$
7: **end if**
8: **end**

4.6 Berechnung der Widerstände

Ausgehend vom Differenzbuffer können die Widerstände berechnet werden, die auf ein Objekt wirken. Um die Stärke und Position der Gegenkraft zu ermitteln, muss für jede Position, an der eine Überschneidung stattfindet, die Masse des verdrängten Schnees ermittelt werden. Je mehr Schnee an einer Position verdrängt wird, desto stärker wird diese gewichtet. Nachdem

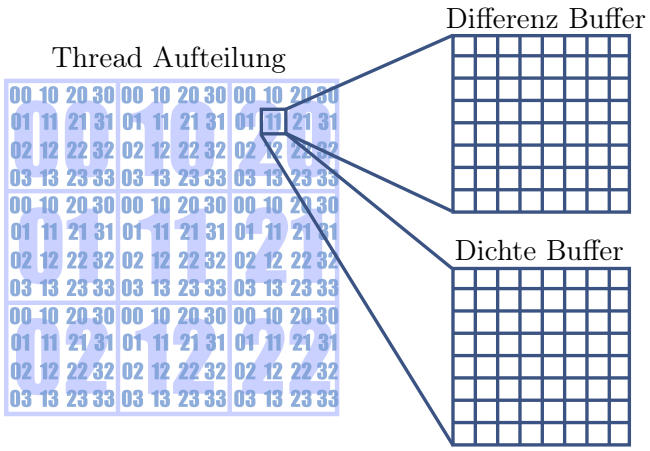


Abbildung 4.4: Jeder Thread kümmert sich um einen eigenen Bereich, der durch dessen Gruppen ID und ID innerhalb der Gruppe bestimmt wird. Für diesen Bereich holt er sich die benötigten Werte aus dem Differenz Buffer und dem Dichte Buffer und führt die entsprechenden Berechnungen durch.

die Berechnung der Masse auf jeder Position unabhängig erfolgen kann, bietet es sich hier an, diese parallel mittels Compute Shader auf der GPU zu berechnen. Hierbei wird der Bereich gleich zwischen den einzelnen Threads aufgeteilt, wobei jeder für seinen Bereich die Ergebnisse ermittelt und anschließend zu einem entsprechenden Buffer addiert. Die einzige Stelle, an der eine Concurrency Problematik auftreten kann, ist jene beim addieren, wobei hier die Funktion `InterlockedAdd()` verwendet werden kann.

Eine Möglichkeit, wie die Berechnung aufgeteilt werden könnte, wird in Abbildung 4.4 gezeigt. Dabei wird wieder auf das Basiskonstrukt aus Abschnitt 4.2 aufgebaut. Jeder Thread muss für jedes seiner Felder, die Masse der verdrängten Schneemenge ermitteln und die Position entsprechend der Masse gewichtet addieren. Hat ein Thread seinen Bereich abgearbeitet, müssen die entsprechenden Werte zu einem globalen Buffer addiert werden, welcher schlussendlich vom C++ Code ausgelesen und verwendet werden kann. Beim Addieren muss dafür gesorgt werden, dass keine Probleme durch zeitgleiche Schreibezugriffe auftreten können, wodurch die atomare Funktion `InterlockedAdd()` verwendet werden sollte. Die Implementierung des Compute Shaders findet sich im Anhang B.3.

Durch die durchschnittliche Position und die Masse kann wie in Abschnitt 3.4 die Gegenkraft berechnet werden. Diese muss dann nur an der entsprechenden Position auf das Objekt wirken, wofür die meisten Physikengines die Funktionalität zur Verfügung stellen.

4.7 Komprimieren des Schnees

Das Komprimieren des Schnees erfolgt mittels der Differenz- und Dichteinformationen. Hierfür werden alle Werte des Differenzbuffers, die größer als 0 sind und somit eine Verdrängung bedeuten, entsprechend der Dichte auf der jeweiligen Position, komprimiert und „verschoben“. Die Verschiebung an sich erfolgt über die Kontur (siehe nächsten Abschnitt), daher müssen die Werte lediglich aufsummiert werden. Die Komprimierung passiert wie in Abschnitt 3.5.1 beschrieben über den Komprimierungsfaktor. Da dessen Berechnung für jede Position eigenständig gemacht werden kann, bietet sich auch für die Komprimierung an, diese mittels Compute Shader umzusetzen. Nachdem hier keinerlei Concurrency-Probleme auftreten, können auch die Einträge im Dichtebuffer direkt aktualisiert werden.

4.8 Verteilen des Schnees

Um den verdrängten Schnee verteilen zu können, muss zuerst dessen Menge berechnet werden. Anschließend muss die Kontur ermittelt werden, auf die der Schnee entsprechend der Bewegung des Objektes verteilt wird.

4.8.1 Ermitteln der verdrängten Schneemenge

Hierfür muss der Differenzbuffer nach positiven Werten durchsucht werden. Jener Anteil der nicht komprimiert wird, muss aufsummiert werden. Da dies von der Dichte an den jeweiligen Positionen abhängt, muss sichergestellt werden, dass diese Berechnung vor dem Komprimieren stattfindet. Um diese Problematik zu umgehen, bietet es sich an, sowohl das Komprimieren als auch die Berechnung der Schneemenge in einem Schritt zu erledigen. Hierbei gilt es allerdings zu beachten, dass bei einer Umsetzung mittels Compute Shader mehrere Threads gleichzeitig auf denselben Speicher schreiben wollen. Nachdem dieser Zugriff immer schreibend erfolgt, reicht der Einsatz der `InterlockedAdd()`-Methode aus, um diese Problematik zu umgehen.

4.8.2 Ermitteln der Kontur

Das Ermitteln der Kontur kann ebenfalls im gleichen Simulationsschritt erfolgen, wie das Ermitteln der zu verschiebenden Schneemenge und das Komprimieren. Auch hier sind die Werte im Differenzbuffer ausschlaggebend, wobei im Gegensatz zu den anderen beiden Schritten auch die Werte der Nachbarfelder benötigt werden. Diese Zugriffe sind jedoch alle lesend, wodurch keine Concurrency Problematik auftritt. Es bietet sich jedoch an, nicht direkt auf den Buffer mit den Differenzinformationen zuzugreifen, sondern auf einen gruppeninternen (`groupshared`) Speicher, da diese Werte relativ häufig gebraucht werden.

Wie Algorithmus 4.4 zeigt, muss, bevor irgendwelche Berechnungen stattfinden, der Gruppeninterne Speicher D', wie in Algorithmus 4.5 gezeigt, befüllt werden. Hierfür wird im Prinzip wieder das Basiskonstrukt aus Abschnitt 4.2 eingesetzt. Dabei ist zu beachten, dass alle Felder, die von Threads einer Gruppe bearbeitet werden, auch die Nachbarfelder benötigen. Hierfür werden die Dimensionen von D' entsprechend auf $T_\mathrm{w} \cdot T_\mathrm{x} + 2$ und $T_\mathrm{h} \cdot T_\mathrm{y} + 2$ gesetzt. Beim Lesen kommt es bei den Randfeldern somit dazu, dass Werte benötigt werden, die eigentlich nicht existieren (außerhalb des Buffers). In diesen Fällen wird einfach der Wert 0 angenommen, der keine Überschneidung kennzeichnet und sich somit in keinster Weise auf die Berechnung auswirkt. Nachdem D' befüllt wurde, müssen alle Threads wieder synchronisiert werden, damit keiner davon bereits mit der Berechnung anfängt, obwohl noch nicht alle benötigten Daten vorhanden sind.

Bei der eigentlichen Berechnung wird von den Threads einer Gruppe jedes Feld in D', welches nicht am Rand ist, abgearbeitet. In Algorithmus 4.6 ist zu sehen, dass für jedes Feld überprüft wird, ob es sich um ein Konturfeld handelt oder nicht (Algorithmus 4.7). Ist das Feld ein Konturfeld, werden die Position, die Richtung der Kontur (Algorithmus 4.8) und die Gewichtung basierend auf der Richtung (Algorithmus 4.9) gespeichert und in die Kontur aufgenommen. Dabei handelt es sich bei der Umsetzung um einen `Append-/ConsumeBuffer`, in den alle Konturfelder gespeichert werden. Gleichzeitig können auch die Aufsummierungen für S_T und k_T erfolgen.

Die tatsächliche Implementierung findet sich in Anhang B.4. Dabei werden die vorgestellten Algorithmen umgesetzt, wobei kleiner Abweichungen zugunsten der Performance vorkommen.

Algorithmus 4.4: Struktur der `main`-Funktion im Shader, zur Ermittlung der Kontur.

```
1: CalculateContour(G_I, T_I)
2:     LoadValues(G_I, T_I)
3:     SynchronizeThreads()        ▷ GroupMemoryBarrierWithGroupSync()
4:     CalculateContourWithThread(G_I, T_I)
5: end
```

4.8.3 Verteilen des Schnees

Für das Verteilen des Schnees wird die gesamte verdrängten Schneemenge s_T und die Summe der Richtungsfaktoren entlang der Kontur k_T benötigt, welche im vorherigen Simulationsschritt ermittelt wurden. Nun muss lediglich für jeden Eintrag im `Append-/Consumebuffer` der Richtungsfaktor ermittelt werden. Die resultierende Schneemenge ergibt sich aus dessem Anteil zum Gesamtrichtungsfaktor. Im Algorithmus 4.10 wird der Vorgang gezeigt.

Nachdem für jedes Konturfeld die neue Schneemenge separat ermittelt

Algorithmus 4.5: Laden der benötigten Differenzwerte aus dem Buffer D in die gruppeninterne Variable D'. Gleichzeitig wird auch die Schneehöhe H entsprechend der Differenz D verringert. $K, D, H, M, N, G_\mathrm{x}, G_\mathrm{y}, T_\mathrm{x}, T_\mathrm{y}, T_\mathrm{w}$ und T_h sind als globale Variablen verfügbar.

```
1:  LoadValues(G_I, T_I)
        Lädt jene Differenzwerte von D in D', die für die aktuelle Gruppe
        benötigt werden. Da die Nachbarfelder ebenfalls benötigt werden,
        wird am Rand jeweils 1 Feld zusätzlich geladen.
2:      S_x ← (G_I.x · G_x + T_I.x) · T_w
3:      S_y ← (G_I.y · G_y + T_I.y) · T_h
4:      o_x ← 0                                  ▷ Offset in x für Nachbarfeld
5:      if T_I.x = 0 ∨ T_I.x = T_x − 1 then
6:          o_x ← 1
7:      end if
8:      o_y ← 0                                  ▷ Offset in y für Nachbarfeld
9:      if T_I.y = 0 ∨ T_I.y = T_y − 1 then
10:         o_y ← 1
11:     end if
12:     i ← 0
13:     if T_I.y = 0 then i ← −1
14:     end if
15:     j ← 0
16:     if T_I.x = 0 then j ← −1
17:     end if
18:     for i to (T_h + o_y) do
19:         for j to (T_w + o_x) do
20:             if S_y + i < 0 ∨ S_y + i ≥ N ∨ S_x + j < 0 ∨ S_x + j ∨ M then
21:                 D'(T_I.y · T_h + i + 1, T_I.x · T_w + j + 1) ← 0
22:             else
23:                 D'(T_I.y · T_h + i + 1, T_I.x · T_w + j + 1) ← D(S_x + j, S_y + i)
24:                 if i ≥ 0 ∧ i < T_h ∧ j ≥ 0 ∧ j < T_w then      ▷ kein
                        Nachbarfeld
25:                     H(S_x + j, S_y + i) ← H(S_x + j, S_y + i) − D(S_x + j, S_y + i)
26:                 end if
27:             end if
28:         end for
29:     end for
30: end
```

wird, bietet es sich auch in diesem Fall an, die Berechnung mittels Compute Shader durchzuführen. Hierbei übernimmt jeder Thread eine gewisse Anzahl an Konturfeldern (abhängig von der Anzahl im Buffer). Diese Anzahl muss jedoch zuvor ermittelt werden, da es vom Compute Shader aus

Algorithmus 4.6: Ermitteln der Kontur K, der Gesamtschneemenge s_T und der Konturgewichtung k_T. Hierfür werden die Differenzwerte aus D' verwendet und die Geschwindigkeit des Objektes $\boldsymbol{v}$ welche ebenfalls als globale Variable vorhanden ist.

```
1:  CalculateContourWithThread(G_I, T_I)
        Überprüft alle Felder des Threads auf Konturfelder und fügt sie gegebenenfalls der Konturliste K hinzu. Dabei besteht jeder Eintrag (k) von K aus der Position des Feldes, der Konturrichtung und deren Gewichtung.
2:      S_x ← (G_I.x · G_x + T_I.x) · T_w
3:      S_y ← (G_I.y · G_y + T_I.y) · T_h
4:      for i = 0 to T_h do
5:          for j = 0 to T_w do
6:              s_T ← s_T + D'(S_y + i + 1, S_x + j + 1) · W(S_x + j + 1, S_y + i + 1)
7:              if IsContour(S_y + i, S_x + j) then
8:                  k.dx ← GetDirection(S_y + i, S_x + j).x        ▷ x-Richtung
9:                  k.dz ← GetDirection(S_y + i, S_x + j).z        ▷ z-Richtung
10:                 k.x ← (G_I.x · G_x + T_I.x) · T_w + j
11:                 k.y ← (G_I.y · G_y + T_I.y) · T_h + i
12:                 k.s_w ← GetDirectionFactor( k.dx, k.dz, v );
13:                 k_T ← k_T + k.s_w                  ▷ Concurrency Problematik!
14:                 K.append(k)
15:             end if
16:         end for
17:     end for
18: end
```

Algorithmus 4.7: Überprüfen, ob es sich bei einem Feld um ein Konturfeld handelt.

```
1:  IsContour(y, x)
2:      if D'(y + 1, x + 1) > 0 then
3:          return false
4:      end if
5:      for i = 0 to 2 do
6:          for j = 0 to 2 do
7:              if D'(y + i, x + j) > 0 then
8:                  return true
9:              end if
10:         end for
11:     end for
12:     return false
13: end
```

Algorithmus 4.8: Ermitteln der Konturrichtung, durch Anwenden eines Kantenfilters.

```
1:  GetDirection(i, j)
2:      d ← (0, 0)
3:      d ← d + D'(i, j) · (−1, 1)
4:      d ← d + D'(i, j + 1) · (0, 1)
5:      d ← d + D'(i, j + 2) · (1, 1)
6:      d ← d + D'(i + 1, j) · (−1, 0)
7:      d ← d + D'(i + 1, j + 2) · (1, 0)
8:      d ← d + D'(i + 2, j) · (−1, −1)
9:      d ← d + D'(i + 2, j + 1) · (0, −1)
10:     d ← d + D'(i + 2, j + 2) · (1, −1)
11:     d ← d̂
12:     return d
13: end
```

Algorithmus 4.9: Ermitteln der Konturrichtung, durch Anwenden eines Kantenfilters.

```
1:  GetDirectionFactor(x, z, v)
2:      w ← (x, 0, z)
3:      w ← ŵ
4:      v ← v̂
5:      r ← 1 − v · (v.x, 0, v.z)
6:      return max(0, v · w) + r/K
7:  end
```

Algorithmus 4.10: Verteilen des Schnees entlang der Kontur.

```
1:  MoveSnowToBorder(K)
        n ... Nummer an Iterationen pro Thread
2:      for i = 0 to n do
3:          k ← K.consume()
4:          if k.s_w > 0 then
5:              S(k.x, k.y) ← S(k.x, k.y) + s_T · k.s_w / k_T
6:          end if
7:      end for
8:  end
```

keinen Zugriff auf die Länge des Buffers gibt (lediglich die maximale Buffergröße kann abgefragt werden). Dieses Auslesen der Buffergröße gestaltet sich als äußerst kompliziert und unperformant. Zuerst muss ein Buffer erstellt werden welcher anschließend mittels `CopyStructureCount()`-Methode befüllt wird. Dieser Buffer muss danach für Leseoperationen gemappt werden

damit schlussendlich der Wert ausgelesen werden kann. Dieser Wert kann dann an den Shader weitergegeben werden. Der Code hierfür findet sich im Anhang B.5. Da dieser Transfer von der GPU zur CPU ziemlich kostspielig ist (da die GPU zu der Zeit keine anderen Berechnungen durchführen kann), wird bei der tatsächlichen Umsetzung nicht die korrekte Anzahl ermittelt. Stattdessen holt sich jeder Thread gleich oft einen Wert aus dem Buffer, überprüft danach allerdings, ob es sich um einen gültigen Wert handelt. Das funktioniert am Einfachsten, indem abgefragt wird, ob der Richtungsfaktor des Konturfeldes größer 0 ist.

```
1 uint iterations = 100;
2
3 for( uint i = 0; i < iterations ; ++i ) {
4   BufferStructContour bsc = g_Contour.Consume();
5   if( bsc.sw > 0 ) {
6     ...
7   }
8 }
```

4.8.4 Berichtigen der Hügel

Das Berichtigen der Hügel ist der performance-lastigste Teil der Simulation. Grund dafür ist, dass es sich dabei um einen iterativen Prozess handelt, welcher einige Male ausgeführt werden muss, bevor ein gültiger Zustand erreicht wird. Die Anzahl der Iterationen und der Aufwand pro Iteration hängen dabei maßgeblich von der Größe des Buffers für die Schneehöhe ab. Bei jeder Iteration wird jedes Element des Höhen-Buffers mit seinen acht Nachbarn verglichen. Dabei wird sichergestellt, dass, falls die maximale Differenz $\phi_{\max}$ überschritten wird, Schnee von dem Feld auf die niedrigeren Nachbarfelder verteilt wird. Dieser Vorgang muss so lange wiederholt werden, bis nicht mehr verschoben werden musste. Eine genauere Beschreibung des Vorganges findet sich in Abschnitt 3.10.

Allgemeine Umsetzung

Auch bei dieser Aufgabe wird auf Compute Shader zurückgegriffen, indem sich wieder jeder Thread einem bestimmten Bereich im Buffer widmet. Es gilt dabei allerdings zu berücksichtigen, dass sich die einzelnen Threads gegenseitig beeinflussen, da der Schnee von einem Element im Buffer auf dessen Nachbarfelder verschoben wird (welche von anderen Threads bearbeitet werden können). Der selbe Fall kann natürlich auch gruppenübergreifend passieren, wodurch es notwendig wird, einen eigenen Buffer für die Ergebnisse zu verwenden. Somit kann der Vergleich immer mittels der Werte des Höhenbuffers erfolgen und die Verschiebungen rein am temporären Buffer. Auf diese Weise existiert immer ein konsistenter Datenzustand.

Vorbereitung

Nachdem bei dieser Methode relativ viele Lesezugriffe erfolgen, bietet es sich an, die Werte für die Höheninformationen von Schnee bzw. Objekten in `groupshared` Variablen einzulesen. Um Überprüfungen der Randpunkte zu minimieren, werden diese gruppeninternen Variablen mit einem zusätzlichen Feld am Rand eingelesen. Für die äußersten Randpixel, für die keine Werte in den Höhenbuffern existieren, wird einfach ein Wert höher der Maximalhöhe verwendet, da somit nie auf diese Felder Schnee verschoben werden kann. Nachdem insgesamt drei `groupshared` Variablen benötigt werden (zwei für die Eingangswerte und eine für die Resultate) muss darauf geachtet werden, dass der Speicher pro Gruppe die 32 Kilobyte nicht übersteigt. Je nach Buffergröße muss daher eventuell die Anzahl der Thread-Gruppen erhöht werden (die Anzahl der Threads pro Gruppe ändert natürlich nichts an dem Problem). Die Umsetzung ist in Anhang B.6 zu sehen.

Verschieben

Beim Verschieben iteriert jeder Thread über seine ihm zugeteilten Felder. Für jedes von ihnen überprüft er die Höhenunterschiede zu seinen Nachbarfeldern und ermittelt sich gegebenenfalls die Menge an Schnee, die verschoben werden muss. Anschließend wird diese Menge gleichmäßig auf alle umliegenden Felder verteilt, die niedriger sind, als das aktuelle Feld. Algorithmus 4.11 zeigt den Ablauf genauer.

Wie dieser Algorithmus zeigt, wird der Höhenbuffer für das Ergebnis mit 0 Initialisiert und erst später mit Werten befüllt. Dabei muss darauf geachtet werden, dass nicht zwei Threads gleichzeitig den Ergebnisbuffer verändern, wobei sich hier wieder die `InterlockedAdd`-Methode anbietet. Eine Änderung am Algorithmus, die das Ergebnis etwas verbessert, ist jene, dass nur jene Nachbarfelder berücksichtigt werden, auf denen kein Objekt im Weg ist. Ansonsten kann es passieren, dass der Schnee in ein Objekt verschoben wird (was erst im nächsten Simulationsschritt wieder beseitigt werden würde). Hierfür müsste die Überprüfung in den Zeilen 7 und 19 wie folgt lauten:

$$\textbf{if } H(x,y) > H(x+i,y+j) \wedge H(x,y) < U(x+i,y+j) \textbf{ then}$$

4.9 Rendern des Schnees

Das Rendern vom Schneeboden kann im Prinzip relativ einfach umgesetzt werden. Es wird eine Plane mit einer großen Anzahl an Vertices dargestellt, wobei die y-Position jedes Vertex von dem Wert im Höhenbuffer an der entsprechenden Stelle bestimmt wird. Eine allerdings weit effizientere Möglichkeit besteht im Einsatz von Tessellierung. Außerdem sollten auch die

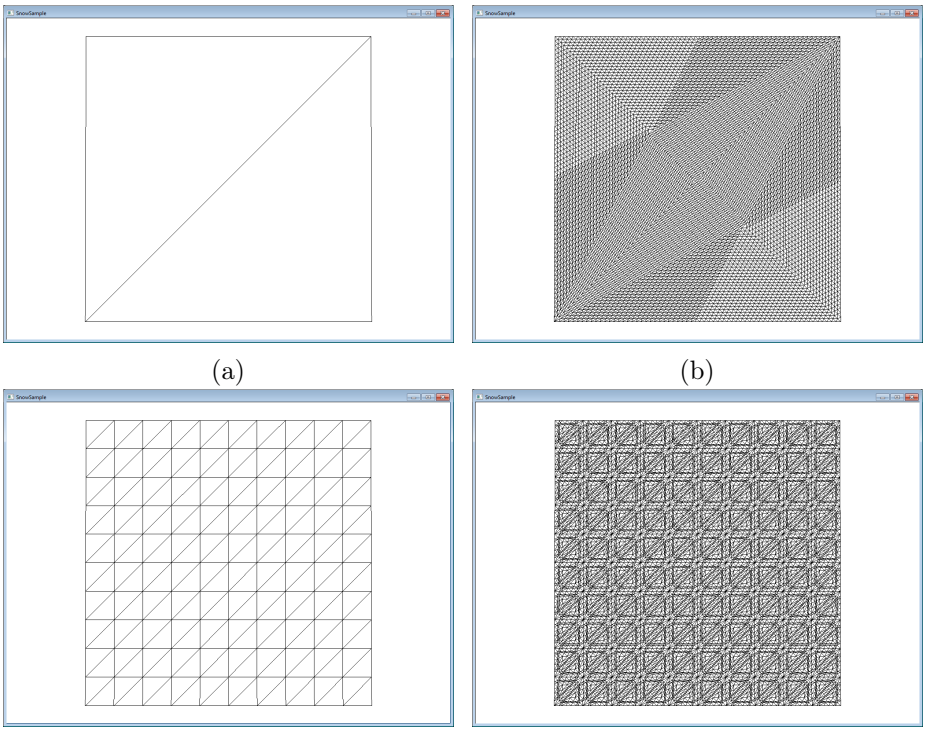
Algorithmus 4.11: Verschieben des Schnees

```
1:  MoveSnow(x, y, H, H')  ▷ H ... aktueller Höhenbuffer  ▷ H' ... neuer Höhenbuffer
2:      n ← 0                                   ▷ Anzahl der kleineren Nachbarn
3:      m ← 0          ▷ Anzahl der Nachbarn, deren Höhenunterschied das Maximum übersteigt
4:      s ← 0                                   ▷ Menge an zu verschiebenden Schnee
5:      for i = −1 to 1 do
6:          for j = −1 to 1 do
7:              if H(x, y) > H(x + i, y + j) then
8:                  n ← n + 1
9:                  if H(x, y) − φ_max > H(x + i, y + j) then
10:                     m ← m + 1
11:                     s ← s + H(x, y) − φ_max − H(x + i, y + j)
12:                 end if
13:             end if
14:         end for
15:     end for
16:     t ← (s/m)/n                             ▷ Zu verschiebender Schnee pro Nachbar
17:     for i = −1 to 1 do
18:         for j = −1 to 1 do
19:             if H(x, y) > H(x + i, y + j) then
20:                 H'(x + i, y + j) ← H'(x + i, y + j) + t
21:             end if
22:         end for
23:     end for
24:     H'(x, y) ← H'(x, y) + H(x, y) − s
25: end
```

Vertex-Normalen den Schneehöhen angepasst werden, da ohne entsprechendes Shading das Ergebnis optisch ziemlich schlecht ist.

4.9.1 Mesh

Für das Test-Mesh wird eine Plane verwendet, die eine Seitenlänge von 10 Einheiten/Metern besitzt. Wird Tessellierung verwendet, spielt die Auflösung des Meshes eine eher untergeordnete Rolle. In Abbildung 4.5 wird gezeigt, mit welch geringen Auflösungen bereits relativ hoch aufgelöste Meshes erzeugt werden können. Wie viele Vertices im Endeffekt gebraucht werden, hängt von der Buffergröße ab, es macht beispielsweise relativ wenig Sinn eine Plane zu erzeugen, die rund 1000 Vertices pro Seitenlänge hat, wenn der Buffer nur 100 Werte besitzt. Umgekehrt gehen Werte im Buffer verloren, wenn die Auflösung zu gering gewählt wird. Für das Testprogramm hat sich



(a) (b)

(c) (d)

Abbildung 4.5: Tessellierung des Meshes. Bereits ein 1 mal 1 großes Mesh (a) kann relativ fein unterteilt werden (b). Bei einem 10 mal 10 Mesh (c) reicht bereits ein viel kleinerer Tessellierungsfaktor aus, um ein entsprechend hoch aufgelöstes Mesh zu erhalten (d).

eine Plane mit ursprünglich 10 mal 10 Vertices bewährt, da damit die Tessellierung recht gut gesteuert werden kann und die 256 mal 256 Bufferwerte recht gut ausgenützt werden können.

Eine weitere Kleinigkeit, die beim Mesh berücksichtigt werden sollte, sind Texturkoordinaten. Auf diese Weise kann dem Boden eine Textur zugewiesen werden. Das würde ebenfalls die Möglichkeit eröffnen, bei deformierten Schnee eine andere Textur zu verwenden, wodurch das optische Ergebnis verbessert werden könnte.

4.9.2 Tessellierung

Wie bereits im vorherigen Abschnitt kurz gezeigt, bietet Tessellierung die Möglichkeit, die Auflösung des Meshes auf der GPU zu erhöhen. Diesen Prozess auf die Grafikkarte auszulagern ist um einiges performanter, als ein bereits hoch aufgelöstes Mesh von der CPU zur GPU zu schicken. Im Einführungskapitel in Abschnitt 1.6.2 wurde bereits auf die Tessellierungs-

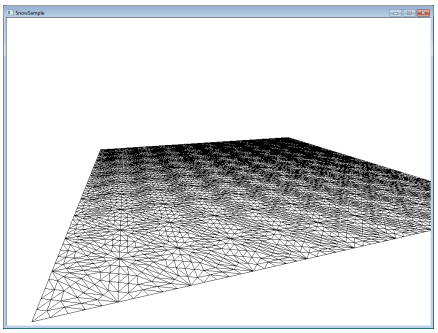


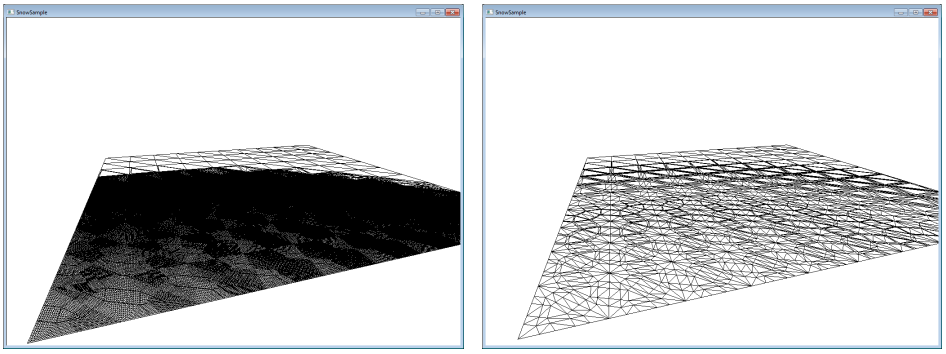
Abbildung 4.6: Konstante Tessellierung wirkt sich überall gleich auf das Mesh aus. In dieser Abbildung wird ein konstanter Tessellierungsfaktor von 5 verwendet.

Funktionalität des Shader Models 5 und die damit verbundenen neuen Abschnitte der Renderpipeline eingegangen.

Eine Möglichkeit, Tessellierung einzusetzen, besteht darin, einen fixen Tessellierungsfaktor für die gesamte Szene festzulegen. Im Anhang B.7 ist der Code des entsprechenden Schnee-Shaders zu finden. Dieser ist eine etwas abgeänderte Variante des Beispielcodes aus dem „DetailTessellation11“ Projekt der DirectX Beispiele. Das Ergebnis, wie sich dieser Algorithmus auf die Bodenplane auswirkt, wird in Abbildung 4.6 gezeigt. Wie dieses Beispiel sehr schön veranschaulicht, wird gerade im hinteren Bereich des Meshes, der verhältnismäßig wenig Platz am Bildschirm einnimmt, unnötig viel Geometrie erzeugt. Um das zu verhindern, gibt es relativ viele Alternativen, eine davon, „Distance Adaptive Tessellation“ (entfernungsabhängige Tessellierung) wird im nachfolgenden Abschnitt näher vorgestellt.

4.9.3 Distance Adaptive Tessellation

Bei der Distance Adaptive Tessellation wird der Tessellierungsfaktor von der Entfernung der Geometrie zur Kamera bestimmt. Je näher sie sich bei der Kamera befindet, desto stärker wird tesselliert. Der Code bleibt dabei im Vergleich zum Beispiel mit der konstanten Tessellierung beinahe unverändert, lediglich der konstante Teil des Hull Shaders wird leicht modifiziert, da in ihm die Berechnung der Tessellierungsfaktoren passiert. Wo bei der konstanten Tessellierung die Werte für die Faktoren der einzeln Kanten noch direkt gesetzt wurden, wird bei der entfernungsabhängigen Variante für jede Kante ein eigener Faktor ermittelt. Es ist wichtig, dass jede Kante ihren eigenen Faktor ermittelt und nicht für den gesamten Patch (hier Triangle) den gleichen verwendet. Ansonsten könnte es passieren, dass zwei gleiche Kanten, die zu unterschiedlichen Patches gehören, verschieden stark tesselliert werden, was zu Löchern im Mesh führen kann.



$t_{\max} = 25,\ d_{\min} = 20,\ r = 2$ $t_{\max} = 5,\ d_{\min} = 5,\ r = 20$

Abbildung 4.7: Beispiele für Distance Adaptive Tessellation

Der Faktor für die entfernungsabhängige Tessellierung einer Kante

$$t_{\mathrm{f}} = 1 - \min(1, \max(0, d/r)), \tag{4.7}$$

wobei

$$d = |\big((\boldsymbol{p}_1 + \boldsymbol{p}_2) \cdot 0,5\big) - \boldsymbol{p}_{\mathrm{c}}| - d_{\min} \tag{4.8}$$

ermittelt sich aus den Anfangs- und Endpunkten $\boldsymbol{p}_1$ und $\boldsymbol{p}_2$ der Kante, der Position der Kamera $\boldsymbol{p}_{\mathrm{c}}$, der minimalen Entfernung, an der bereits maximal tesselliert werden soll $d_{\min}$ und dem Bereich r, welcher angibt, wie weit ab $d_{\min}$ der Faktor interpoliert werden soll. Dabei ist zu berücksichtigen, dass der Wert für t_{f} immer im Bereich zwischen 0 und 1 liegt, wodurch dieser noch mit dem maximal erwünschten Tessellierungsfaktor $t_{\max}$ multipliziert werden muss, um die gewünschte Tessellierung zu erhalten. Der entsprechende Code wird im Anhang B.8 gezeigt, verschiedene Ergebnisse, abhängig der gewählten Werte für die Berechnung von t_{f} finden sich in Abbildung 4.7.

Ein weiteres kleines Detail, dass aber große Auswirkungen haben kann, ist die Definition der „Primitive Topology“ im C++ Code. Diese Unterscheidet sich nämlich in Abhängigkeit des Einsatzes von Tessellierung. Wird die falsche Topologie verwendet, führt das nicht nur zu einem nicht funktionierenden Programm sondern kann das ganze System zum freezen[2] bringen. Wenn `D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST` beispielsweise ohne Tessellierung verwendet wird, muss die Topologie für das Primitiv mit Tessellierung als `D3D11_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST` angegeben werden.

[2]Das komplette System reagiert nicht mehr, was nur über einen erzwungenen Neustart behoben werden kann.

4.9.4 Vertex Normalen

Eine reine Positionierung der einzelnen Vertices reicht zwar aus um das Mesh zu verformen, jedoch ist das Ergebnis ohne entsprechendes Shading nicht wirklich ansehnlich. Um ein „korrektes“ Shading zu erreichen, müssen die Vertexnormalen entsprechend verändert werden. Hierfür gibt es einige verschiedene Möglichkeiten. Eine besteht beispielsweise darin, die Face Normalen der angrenzenden Faces zu ermitteln und anschließend den Durchschnitt zu bilden. Da die Position der Vertices allerdings ursprünglich nicht bekannt ist, da diese erst vom Tessellator erzeugt werden, stehen die benötigten Informationen im Domain Shader (in dem die Normalen berechnet werden) jedoch nicht zur Verfügung. Aus diesem Grund wird eine Variante verwendet, die rein auf den Höheninformationen basiert. Dabei wird für die aktuelle Stelle im Höhen Buffer H die Normale aus den Differenzen der umliegenden Felder errechnet. Aus den Differenzen werden zwei Vektoren gebildet,

$$\boldsymbol{d}_{\mathrm{x}}(x,y) = \begin{pmatrix} 2 \\ H(x+1,y) - H(x-1,y) \\ 0 \end{pmatrix} \tag{4.9}$$

und

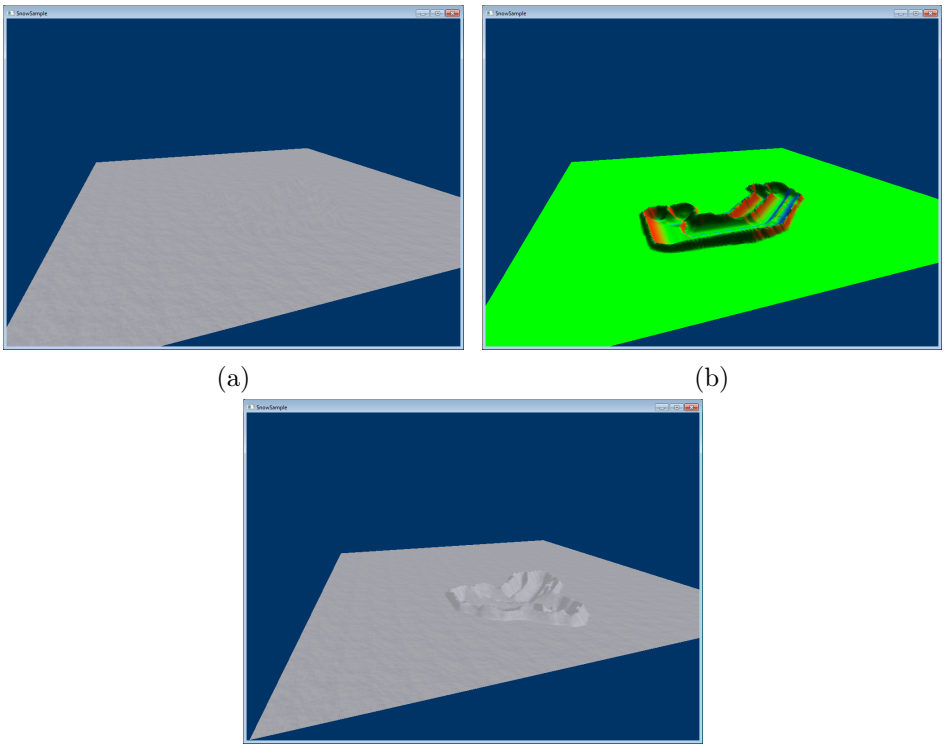
$$\boldsymbol{d}_{\mathrm{z}}(x,y) = \begin{pmatrix} 0 \\ H(x,y-1) - H(x,y+1) \\ 2 \end{pmatrix}, \tag{4.10}$$

welche den Höhenunterschied entlang der x-Achse (d_{x}) und entlang der z-Achse (d_{z}) kennzeichnen. Die Vertexnormale

$$\boldsymbol{n}(x,y) = \boldsymbol{d}_{\mathrm{z}}(x,y) \times \boldsymbol{d}_{\mathrm{x}}(x,y) \tag{4.11}$$

wird schlussendlich aus dem Kreuzprodukt dieser beiden Vektoren gebildet. Im Code sieht das dann wie folgt aus:

```
1 float3 GetNormal( float2 position ) {
2   float dx = g_txSnowDeformation.SampleLevel( g_SampleLinear, float2(
      position.x + ( 1.0f / SNOW_TEXTURE_WIDTH ), position.y ), 0 ).x
3             - g_txSnowDeformation.SampleLevel( g_SampleLinear, float2(
      position.x - ( 1.0f / SNOW_TEXTURE_WIDTH ), position.y ), 0 ).x;
4   float dz = g_txSnowDeformation.SampleLevel( g_SampleLinear, float2(
      position.x, position.y - ( 1.0f / SNOW_TEXTURE_WIDTH ) ), 0 ).x
5             - g_txSnowDeformation.SampleLevel( g_SampleLinear, float2(
      position.x, position.y + ( 1.0f / SNOW_TEXTURE_WIDTH ) ), 0 ).x;
6
7   float3 a = float3( 2.0f / SNOW_TEXTURE_WIDTH, dx, 0 );
8   float3 b = float3( 0, dz, 2.0f / SNOW_TEXTURE_WIDTH );
9   float3 n = cross( b, a );
10
11  return normalize( n );
12 }
```



(a) (b)

(c)

Abbildung 4.8: Werden die Vertex Normalen nicht den Verformungen angepasst, sind diese kaum zu erkennen (a). Werden diese aus den Höheninformationen errechnet (visualiert in (b)), sind die Verformungen durch das richtige Shading gut sichtbar (c).

Nachdem hier die Höhe nicht aus dem Buffer direkt, sondern aus einer Textur (die den Buffer zugrundeliegend hat) ermittelt wird, müssen die Entfernungen dementsprechend auf die Buffergröße normalisiert werden, was mit einer Division durch `SNOW_TEXTURE_WIDTH` erreicht wird. In Abbildung 4.8 werden die Ergebnisse gezeigt, die durch die Berechnung der Vertex Normalen erreicht werden.

Kapitel 5

Ergebnisse

5.1 Visuelle Ergebnisse

In Abbildung 5.1 werden Ergebnisse der Simulation gezeigt. Bei diesen Abbildungen wurde ein 256 mal 256 Elemente großer Buffer verwendet, der auf eine Fläche von 100 m^2 aufgeteilt wird. Als maximale Steigung der Hügel ($\phi_{\max}$) wurde 60 cm pro Meter gewählt.

5.2 Performance

Das verwendete System ist ein Notebook mit Intel Core i7 CPU mit 4 Kernen zu je 1,73 GHz, 6GB Ram und einer ATI Mobility Radeon 5850 mit 1 GB RAM. Zur Framerate Messung wurde Fraps[1] verwendet, zur Performance Messung des C++ Codes wurde auf den Shiny Profiler[2] gesetzt . Als Bildschirmauflösung der Anwendung wurde 1024 mal 768 verwendet. Bei einer Buffergröße von 256 mal 256 wurde eine durchschnittliche Framerate von 110 Frames pro Sekunde erzielt, wobei ohne den Verteilungsvorgang noch über 500 Frames Durchschnitt sind. Bei einer Buffergröße von 512 mal 512 fällt die Framerate bereits auf knapp über 30 Frames mit und auf rund 280 Frames ohne Verteilvorgang. Beide diese Beispiele zeigen, dass das größte Potential bezüglich Performance beim Verteilvorgang liegt.

5.3 Probleme

Bei den Ergebnissen werden auch ein paar Probleme durch die abstrahierte Simulation relativ schnell aufgezeigt. Vor allem jenes, dass sich kein Schnee auf dem Objekt befinden kann. In Abbildung 5.2 wird das Ergebnis gezeigt, das entsteht, wenn sich ein Objekt senkrecht zu tief in den Schnee „bohrt“.

[1] http://www.fraps.com/

[2] http://sourceforge.net/projects/shinyprofiler/

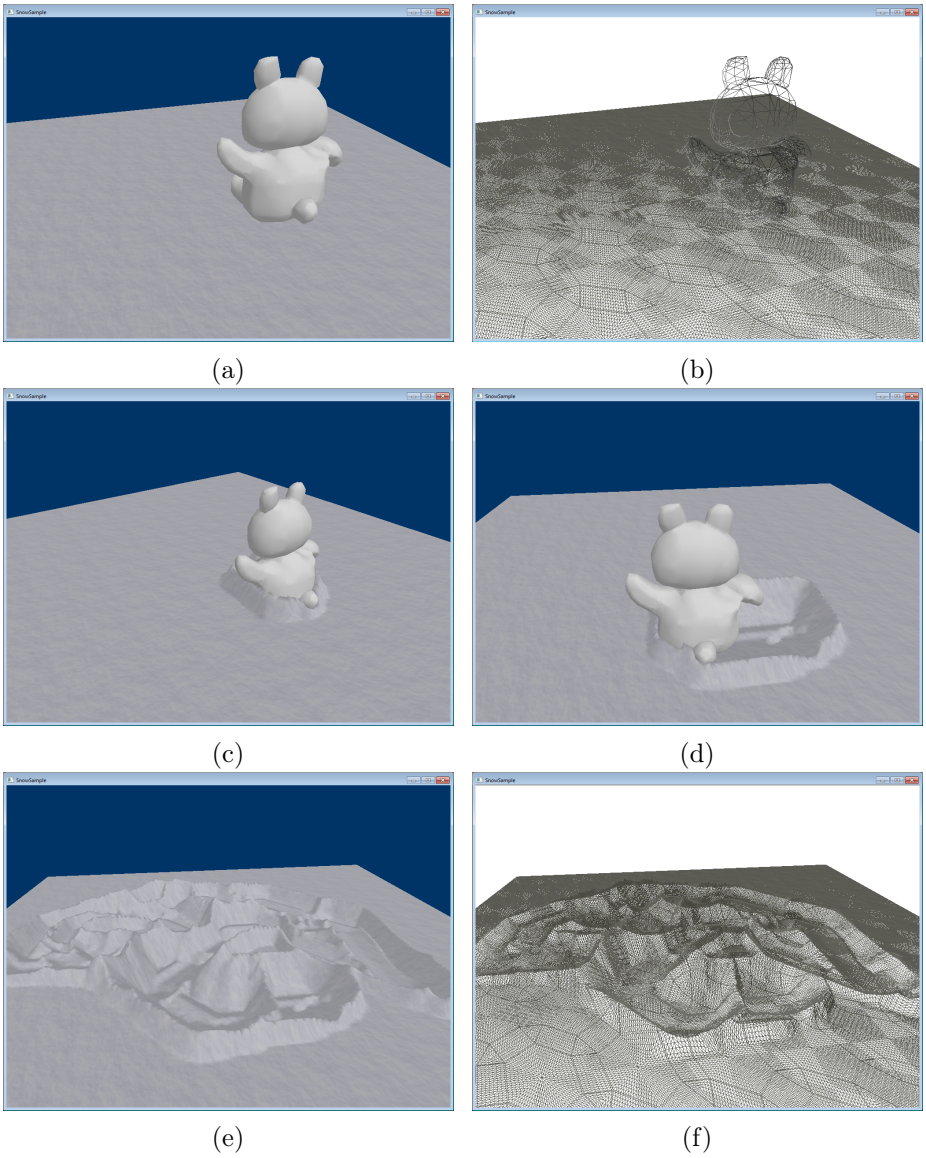


Abbildung 5.1: Ergebnisse der Simulation. Die ursprüngliche Szene besteht lediglich aus einer flachen Schneefläche und einem Objekt (a, b). In (c) wird das Ergebnis einer vertikalen Verdrängung gezeigt, bei der der Schnee gleichmäßig rund um das Objekt verteilt wird. Bei einer horizontalen Bewegung wird der Schnee hauptsächlich in Bewegungsrichtung verschoben (d). (e) und (f) zeigen das Ergebnis, nach einigen Verformungen durch das Objekt.

Das führt auch gleich zum nächsten etwas unschönen Nebeneffekt, dass diese Hänge relativ hoch sind und erst abgeflacht werden, wenn sich das Objekt wieder wegbewegt. Das hat den Grund, dass nur an Positionen Schnee ver-

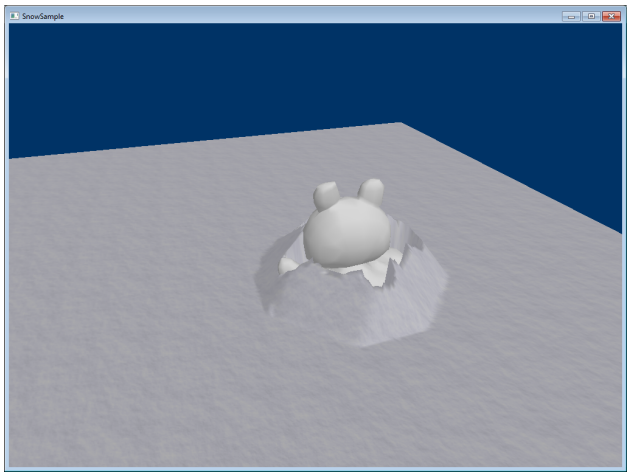


Abbildung 5.2: Die Limitierung, dass sich kein Schnee auf einem Objekt befinden kann, führt relativ schnell zu problematischen Ergebnissen

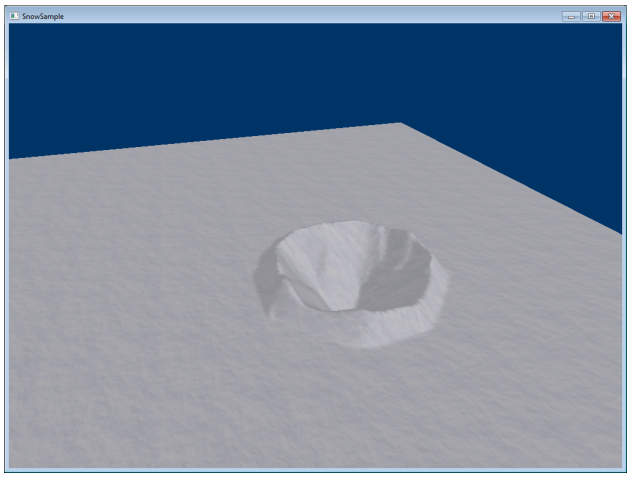


Abbildung 5.3: Die Hänge sind nach Verschwinden des Objektes (das diese verursacht hat) bei weitem nicht mehr so hoch wie vorher (vgl. Abbildung 5.2).

schoben wird, an denen kein Objekt im Weg ist. Abbildung 5.3 zeigt, wie die Hügel aus Abbildung 5.2 aussehen, nachdem das Objekt weg ist. Wie zu erkennen ist, sind diese bei weitem niedriger als mit dem Objekt.

Kapitel 6

Diskussion und Ausblick

6.1 Diskussion

Die Ergebnisse der Simulation zeigen, dass eine grundsätzliche Deformation von Schnee in einer sehr abstrahierten Variante in Echtzeit möglich ist. Die Qualität leidet etwas an der abstrahierten Vorgehensweise, bei der einige Limitierungen in Kauf genommen werden müssen. Die Methode eignet sich also vermutlich nicht für Anwendungefälle, bei denen eine realitätsgetreue Umsetzung erwünscht ist (z.B. realistische Animationsfilme). Wenn der Schnee nur als ergänzendes optisches Mittel eingesetzt wird (z.B. in Sport-Videospielen), sollten die Ergebnisse der Simulation aber definitiv ausreichend sein. Gerade bei Spielen, die in recht limitierten Arealen stattfinden (z.B. Beat ´em ups) würde sich diese Methode aber anbieten, da sie relativ einfach akzeptable Ergebnisse liefert.

6.1.1 Möglichkeiten

Die starke Abstraktion bringt aber nicht nur Nachteile mit sich. Da die Simulation jetzt nicht perfekt für Schnee angepasst wurde, ist es möglich, diese auch für andere Materialien zu verwenden. Vor allem Sand würde sich anbieten, da hierfür nur die Dichtewerte $\rho_{\min}$ und ρmax entsprechend dicht aneinander liegen müssten (damit möglichst wenig Komprimierung erfolgt). Außerdem müsste $\phi_{\max}$ entsprechend niedriger gewählt werden, da Sandhügel üblicherweise nicht so steil sein können wie Schneehügel. In Abbildung 6.1 wird ein Beispiel gezeigt, wie das ausschauen könnte.

6.1.2 Verbesserungen

Sowohl bei der grafischen Darstellung als auch bei der Simulation selbst können einige Verbesserungen vorgenommen werden.

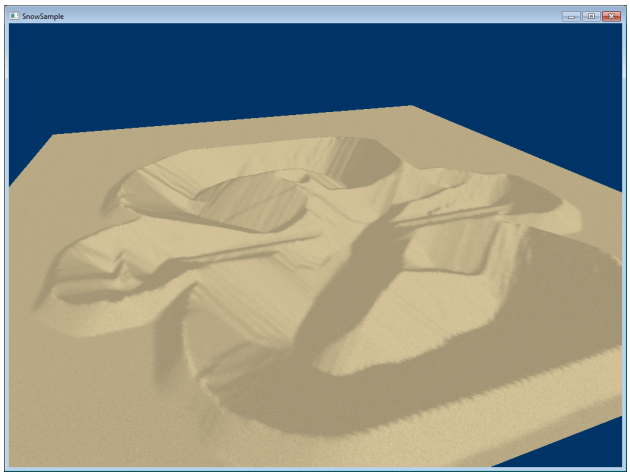


Abbildung 6.1: Die in dieser Arbeit vorgestellte Methode, kann auch für andere Untergründe, wie Sand verwendet werden.

Grafik

Hierbei ist vor allem die Schneedarstellung an sich verbesserungswürdig. Gerade im Bereich der Verformungen würde es sich anbieten, mit einer alternativen Textur zu arbeiten, die nicht eine schöne ebene Schneefläche widerspiegelt. Außerdem würde das optische Ergebnis verbessert werden, wenn diverse Unebenheiten automatisch eingefügt werden würden (z.B. von herausgebrochenen Schneebrocken oder von kleinen Schneeteilen die etwas weggeschleudert werden). Wie auch bereits in [8] vorgeschlagen wurde, würde sich hier vermutlich der Einsatz von Partikelsystemen anbieten, um etwaige wegfliegende Schneeteile zu visualisieren.

Aber auch das Shading an sich ist stark verbesserungswürdig. Hier wurde ja lediglich eine Textur abhängig der Vertexnormale zur Lichtrichtung mehr oder weniger stark erhellt. Hierbei gäbe es natürlich einiges, das zu einem besseren Endergebenis beitragen würde, beispielsweise kleine Lichtpunkte die durch Reflektionen des Lichts durch die Schneekristalle entstehen. Außerdem unterscheiden sich Schneearten voneinander (z.B. feuchter Schnee ist anders als festgefrorener harter Schnee). Die Erzeugung eines entsprechenden Shaders würde also definitiv eine entsprechende Verbesserung mit sich bringen.

Simulation

Die Simulation ist wie bereits mehrfach erwähnt sehr abstrahiert, wodurch zwar eine echtzteitfähige Lösung möglich ist, die Qualität allerdings des öfteren zu wünschen übrig lässt. Gerade die Limitierung, dass sich kein Schnee auf einem Objekt befinden kann, führt relativ bald zu Problemen. Vor allem

wenn ein Objekt eine große Schneemenge vor sich her schiebt, die bereits größer als das Objekt selbst ist, fällt das negativ auf. Es wäre natürlich auch vorteilhaft, wenn fixe Objekte in der Szene, mit Schnee bedeckt sein könnten. Ein Haus mit einer dicken Schneeschicht, die ebenfalls auf Objekte reagiert, würde definitiv die Simulation bereichern.

Eine weitere Verbesserung betrifft die Hügel. Bei diesen würde es sich eventuell anbieten, zwischen denen zu unterscheiden, die vom Objekt direkt durch die Berührung des Objektes entstanden sind (Abdruck) und jenen, die durch die Verteilung des Schnees entstanden sind, da zweitere meist nicht so steil sind. Eine Möglichkeit das zu erreichen, was auch den allgemeinen Charakter der Simulation verbessern würde, wäre es, die Dichte bei der Steilheit miteinzubeziehen. Je komprimierter eine Stelle ist, desto steiler kann sie sein.

Große Flächen sind ebenfalls ein Punkt, der in dieser Arbeit nicht behandelt worden sind. Die praktische Umsetzung dieser Arbeit verwendet eine relativ kleine Schneefläche (10 mal 10 Meter). Bei einem Fußballspiel müssten das Beispielsweise schon eher 100 mal 100 Meter sein. Das mit einem einzigen Buffer zu realisieren ist unrealistisch, da dieser viel zu groß sein und somit viel zu viel Speicher auf der Grafikkarte in Anspruch nehmen würde. Hierfür müssten also vermutlich mehrere Buffer kombiniert werden, wobei eine entsprechende Routine zur Behandlung der Übergänge zwischen den Buffern gefunden werden müsste.

Performance

Wie bereits in Abschnitt 5.2 angesprochen, ist die performance-lastigste Stelle das Verschieben des Schnees. Hierbei gäbe es vermutlich Algorithmen, die eine performantere Umsetzung gewährleisten würden. Alternativ dazu könnte auch der Bereich des Buffers begrenzt werden, auf den die Verschiebung angewendet wird. Bei der in dieser Arbeit vorgestellten Lösung wird der komplette Buffer in jedem Simulationsschritt mehrmals abgearbeitet. Diese Entscheidung wurde getroffen, da somit beliebig viele Objekte jederzeit mit dem Boden interagieren können und die Performance beinahe unverändert bleibt. Bei einer Limitierung müssten jene Bereiche ermittelt werden, die von einer Verschiebung betroffen sind. Diese Variante würde sich vor allem anbieten, wenn der Simulationsbereich um einiges größer wäre, als die Objekte.

6.2 Ausblick

Das Shader Model 5 wird vermutlich noch länger aktuell sein, da nach wie vor viele Spiele mit DirectX 9.0c (Shader Model 3.0) entwickelt werden und auch die aktuellen Konsolen am Markt nur die Version 3 unterstützen. Hierbei wird es interessant sein, zu sehen, in welche Richtung sich die Nachfolgekonsolen von Sony und Microsoft entwickeln, da vor allem die Spieleindustrie da-

für sorgt, dass neue Grafiktechnologien massentauglich werden. Im Vergleich zum Shader Model 4.0, dass hauptsächlich Geometry Shader eingeführt hat, sind die Funktionalitäten der Version 5 vermutlich universeller einsetzbar, wodurch eine größere Unterstützung in Zukunft recht wahrscheinlich sein dürfte.

Anhang A

Schneeverschiebung – Beispiel

A.1 2D Distance Transform

In Abbildung A.1 werden Beispiele für die Errechnung eines Distance Transforms ausgehend von verschiedenen Bewegungsrichtungen der Objekte gezeigt.

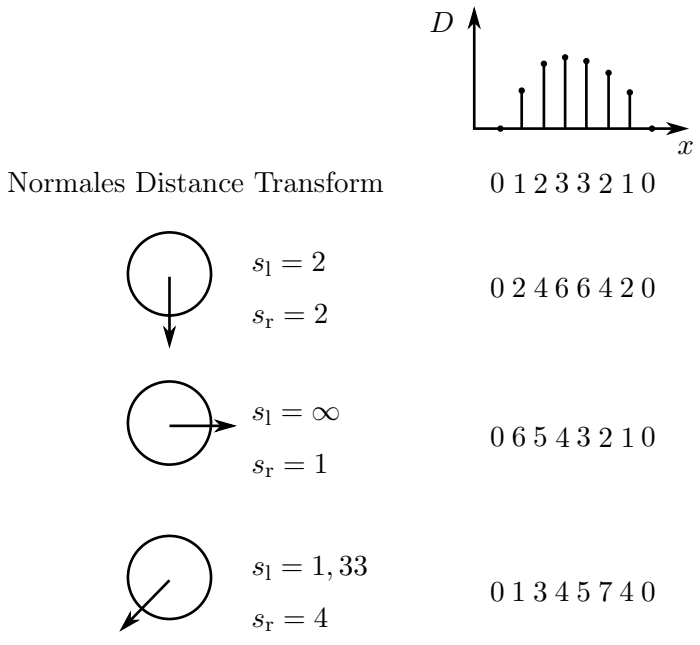


Abbildung A.1: Distance Transform Berechnung anhand der Bewegungsrichtung des Objektes.

A.2 3D Distance Transform

In diesem Beispiel wird die Errechnung eines gewichteten Distance Transforms mittels der in Abschnitt 3.7 beschriebenen Richungsfaktoren gezeigt.

A.2.1 Ausgangssituation

Ausgangssituation ist eine Überschneidung eines Würfels mit einer ebenen Schneefläche, was sich durch die Differenz D zeigt:

$$D = \begin{array}{|c|c|c|c|c|c|c|}\hline 0&0&0&0&0&0&0\\\hline 0&1&1&1&1&1&0\\\hline 0&1&1&1&1&1&0\\\hline 0&1&1&1&1&1&0\\\hline 0&1&1&1&1&1&0\\\hline 0&1&1&1&1&1&0\\\hline 0&0&0&0&0&0&0\\\hline\end{array} \tag{A.1}$$

Einsen kennzeichnen eine Überschneidung, während Nullen in diesem Beispiel die ersten Felder kennzeichnen, die nicht mit dem Objekt in Berührung gekommen sind. Ziel ist es nun, von jedem Feld mit einer Überschneidung die Entfernung zu einem Feld ohne zu ermitteln, wobei nicht der Wert selbst entscheidend ist, sondern das Verhältnis zu den Nachbarfeldern (es ist nur nötig zu wissen, welches Feld näher am Rand ist). Für die Berechnung des Distance Transforms wird der Chamfer Algorithmus verwendet, wobei für die Eckpunkte die euklidische Entfernung berechnet wird.

A.2.2 Beispiel 1

Die Bewegung des Objektes ist mit $\mathbf{v} = (0, -1, 0)^T$ eine rein senkrechte, wodurch die Richtungsfaktoren rein aus dem aufgeteilten Restanteil bestehen:

$$r = 1, \tag{A.2}$$

$$s_{\mathrm{L}} = s_{\mathrm{R}} = s_{\mathrm{D}} = s_{\mathrm{U}} = \frac{1}{4} = 0{,}25. \tag{A.3}$$

Für k wurde in diesem Fall 4 gewählt. Die Werte für die Chamfer Algorithmus Matrizen ergeben sich aus der Division von 1 durch diese Faktoren. Bei einer euklidischen Entfernung ergeben sich folgende Werte:

$$M^R = \begin{pmatrix} \sqrt{32} & 4 & \sqrt{32} \\ 4 & . & . \\ . & . & . \end{pmatrix}, \tag{A.4}$$

$$M^L = \begin{pmatrix} . & . & . \\ . & . & 4 \\ \sqrt{32} & 4 & \sqrt{32} \end{pmatrix}. \tag{A.5}$$

Erstellt man mit diesen Matrizen nun das Distance Transform, ergibt sich ein gleichmäßige Verteilung:

0	0	0	0	0	0	0
0	4	4	4	4	4	0
0	4	8	8	8	4	0
0	4	8	12	8	4	0
0	4	8	8	8	4	0
0	4	4	4	4	4	0
0	0	0	0	0	0	0

.

A.2.3 Beispiel 2

Die Bewegung des Objektes ist mit $\mathbf{v} = (2, -1, 0)^T$ eine Kombination aus einer senkrechten Bewegung und einer nach rechts, wodurch sich die Richtungsfaktoren entsprechend aus dem Richtungsanteil und dem aufgeteilten Restanteil ($k = 4$) zusammensetzen,

$$r = \frac{1}{5} \tag{A.6}$$

$$s_{\mathrm{L}} = (\frac{1}{20} + 0 \cdot \frac{4}{5}) = 0{,}05 \tag{A.7}$$

$$s_{\mathrm{R}} = (\frac{1}{20} + 1 \cdot \frac{4}{5}) = 0{,}77 \tag{A.8}$$

$$s_{\mathrm{U}} = (\frac{1}{20} + 0 \cdot \frac{4}{5}) = 0{,}05 \tag{A.9}$$

$$s_{\mathrm{D}} = (\frac{1}{20} + 0 \cdot \frac{4}{5}) = 0{,}05. \tag{A.10}$$

Bei diesen Faktoren ist bereits ersichtlich, dass hauptsächlich eine Verschiebung nach rechts erfolgt. Da die anderen Richtungen sich alle um zumindest 90° von der Optimalrichtung unterscheiden, haben diese den Richtungsfaktor 0. Für den Chamfer Algorithmus ergeben sich (auch hier mittels einer Division von 1 durch Richtungsfaktor und einer euklidischen Entfernung) folgende Matrizen:

$$M^R = \begin{pmatrix} 28 & 20 & 20{,}04 \\ 20 & . & . \\ . & . & . \end{pmatrix} \tag{A.11}$$

$$M^L = \begin{pmatrix} . & . & . \\ . & . & 1{,}3 \\ 28 & 20 & 20{,}04 \end{pmatrix}. \tag{A.12}$$

Erstellt man mit diesen Matrizen nun das Distance Transform, ist auch hier das links/rechts Gefälle eindeutig erkennbar,

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	6,5	5,2	3,9	2,6	1,3	0,0
0,0	6,5	5,2	3,9	2,6	1,3	0,0
0,0	6,5	5,2	3,9	2,6	1,3	0,0
0,0	6,5	5,2	3,9	2,6	1,3	0,0
0,0	6,5	5,2	3,9	2,6	1,3	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

.

A.2.4 Beispiel 3

Die Bewegung des Objektes ist mit $\mathbf{v} = (2, -1, 3)^T$, lediglich eine um eine z-Bewegung erweiterte Variante zum zweiten Beispiel. Auch hier setzen sich die Richtungsfaktoren aus dem Richtungsanteil und dem aufgeteilten Restanteil zusammen,

$$r = \frac{1}{14} \tag{A.13}$$

$$s_{\mathrm{L}} = 1/(\frac{1}{56} + 0 \cdot \frac{13}{14}) = 0{,}018 \tag{A.14}$$

$$s_{\mathrm{R}} = 1/(\frac{1}{56} + 0{,}53 \cdot \frac{13}{14}) = 0{,}15 \tag{A.15}$$

$$s_{\mathrm{U}} = 1/(\frac{1}{56} + 0{,}8 \cdot \frac{13}{14}) = 0{,}76 \tag{A.16}$$

$$s_{\mathrm{D}} = 1/(\frac{1}{56} + 0 \cdot \frac{13}{14}) = 0{,}018. \tag{A.17}$$

Bei diesen Faktoren ist bereits ersichtlich, dass eine Verschiebung entlang der Bewegungsrichtung des Objektes eindeutig wahrscheinlicher ist, als eine entgegengesetzte. Für den Chamfer Algorithmus ergeben sich folgende Matrizen:

$$M^R = \begin{pmatrix} 3{,}49 & 1{,}29 & 1{,}99 \\ 3{,}24 & . & . \\ . & . & . \end{pmatrix} \tag{A.18}$$

$$M^L = \begin{pmatrix} . & . & . \\ . & . & 1{,}52 \\ 6{,}13 & 5{,}21 & 5{,}42 \end{pmatrix}. \tag{A.19}$$

Erstellt man mit diesen Matrizen nun das Distance Transform, ist auch hier das Gefälle zugunsten der Bewegungsrichtung eindeutig erkennbar,

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,3	1,3	1,3	1,3	1,3	0,0
0,0	2,6	2,6	2,6	2,6	1,9	0,0
0,0	3,9	3,9	3,9	3,9	1,9	0,0
0,0	5,2	5,2	5,2	3,9	1,9	0,0
0,0	6,6	6,6	5,8	3,9	1,9	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

.

A.3 Verschiebung mittels Distance Transform

Nachfolgend wird ein Beispiel für die Verchiebung mittels Distance Transform gezeigt, welches durch eine Bewegungsrichtung von $\mathbf{v} = (2, -1, 3)^T$ entstanden ist. Dabei wird in jeder Iteration Schnee vom höchsten noch nicht behandelten Feld auf die benachbarten (kleineren) Felder verschoben. Bei diesem Beispiel wird auf eine 4er Nachbarschaft zurückgegriffen, wodurch entlang der Diagonalen kein Schnee verschoben wird.

Distance Transform

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,3	1,3	1,3	1,3	1,3	0,0
0,0	2,6	2,6	2,6	2,6	1,9	0,0
0,0	3,9	3,9	3,9	3,9	1,9	0,0
0,0	5,2	5,2	5,2	3,9	1,9	0,0
0,0	6,6	6,6	5,8	3,9	1,9	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

Zu verteilende Schneemenge

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

Iteration 1

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,3	1,3	1,3	1,3	1,3	0,0
0,0	2,6	2,6	2,6	2,6	1,9	0,0
0,0	3,9	3,9	3,9	3,9	1,9	0,0
0,0	5,2	5,2	5,2	3,9	1,9	0,0
0,0	x	x	5,8	3,9	1,9	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,3	1,3	1,0	1,0	1,0	0,0
0,3	0,0	0,0	1,3	1,0	1,0	0,0
0,0	0,3	0,3	0,0	0,0	0,0	0,0

Iteration 2

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,3	1,3	1,3	1,3	1,3	0,0
0,0	2,6	2,6	2,6	2,6	1,9	0,0
0,0	3,9	3,9	3,9	3,9	1,9	0,0
0,0	5,2	5,2	5,2	3,9	1,9	0,0
0,0	x	x	x	3,9	1,9	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,3	1,3	1,4	1,0	1,0	0,0
0,3	0,0	0,0	0,0	1,4	1,0	0,0
0,0	0,3	0,3	0,4	0,0	0,0	0,0

Iteration 3

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,3	1,3	1,3	1,3	1,3	0,0
0,0	2,6	2,6	2,6	2,6	1,9	0,0
0,0	3,9	3,9	3,9	3,9	1,9	0,0
0,0	x	x	x	3,9	1,9	0,0
0,0	x	x	x	3,9	1,9	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,7	2,3	1,7	1,0	1,0	0,0
0,7	0,0	0,0	0,0	1,7	1,0	0,0
0,3	0,0	0,0	0,0	1,4	1,0	0,0
0,0	0,3	0,3	0,4	0,0	0,0	0,0

Iteration 4

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,3	1,3	1,3	1,3	1,3	0,0
0,0	2,6	2,6	2,6	2,6	1,9	0,0
0,0	x	x	x	3,9	1,9	0,0
0,0	x	x	x	3,9	1,9	0,0
0,0	x	x	x	3,9	1,9	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,8	3,3	1,9	1,0	1,0	0,0
0,8	0,0	0,0	0,0	1,9	1,0	0,0
0,7	0,0	0,0	0,0	1,7	1,0	0,0
0,3	0,0	0,0	0,0	1,4	1,0	0,0
0,0	0,3	0,3	0,4	0,0	0,0	0,0

Iteration 5

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,3	1,3	1,3	1,3	1,3	0,0
0,0	2,6	2,6	2,6	2,6	1,9	0,0
0,0	x	x	x	x	1,9	0,0
0,0	x	x	x	x	1,9	0,0
0,0	x	x	x	x	1,9	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,0	1,0	1,0	1,0	1,0	0,0
0,0	1,8	3,3	1,9	1,9	1,0	0,0
0,8	0,0	0,0	0,0	0,0	1,9	0,0
0,7	0,0	0,0	0,0	0,0	2,7	0,0
0,3	0,0	0,0	0,0	0,0	1,7	0,0
0,0	0,3	0,3	0,4	0,7	0,0	0,0

Iteration 6

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,3	1,3	1,3	1,3	1,3	0,0
0,0	x	x	x	x	1,9	0,0
0,0	x	x	x	x	1,9	0,0
0,0	x	x	x	x	1,9	0,0
0,0	x	x	x	x	1,9	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,9	4,3	2,9	2,0	1,0	0,0
0,9	0,0	0,0	0,0	0,0	2,0	0,0
0,8	0,0	0,0	0,0	0,0	1,9	0,0
0,7	0,0	0,0	0,0	0,0	2,7	0,0
0,3	0,0	0,0	0,0	0,0	1,7	0,0
0,0	0,3	0,3	0,4	0,7	0,0	0,0

Iteration 7

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,3	1,3	1,3	1,3	1,3	0,0
0,0	x	x	x	x	x	0,0
0,0	x	x	x	x	x	0,0
0,0	x	x	x	x	x	0,0
0,0	x	x	x	x	x	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	1,9	4,3	2,9	2,0	2,0	0,0
0,9	0,0	0,0	0,0	0,0	0,0	1,0
0,8	0,0	0,0	0,0	0,0	0,0	1,9
0,7	0,0	0,0	0,0	0,0	0,0	2,7
0,3	0,0	0,0	0,0	0,0	0,0	0,9
0,0	0,3	0,3	0,4	0,7	0,9	0,0

Iteration 8

0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	x	x	x	x	x	0,0
0,0	x	x	x	x	x	0,0
0,0	x	x	x	x	x	0,0
0,0	x	x	x	x	x	0,0
0,0	x	x	x	x	x	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0

0,0	1,0	4,3	2,9	2,0	1,0	0,0
1,0	0,0	0,0	0,0	0,0	0,0	1,0
0,9	0,0	0,0	0,0	0,0	0,0	1,0
0,8	0,0	0,0	0,0	0,0	0,0	1,9
0,7	0,0	0,0	0,0	0,0	0,0	2,7
0,3	0,0	0,0	0,0	0,0	0,0	0,9
0,0	0,3	0,3	0,4	0,7	0,9	0,0

Anhang B

Code

B.1 Compute Shader Aufbau

```
1  static const uint BUFFER_WIDTH = 1024;
2
3  static const uint THREAD_GROUP_SIZE_X = 8;
4  static const uint THREAD_GROUP_SIZE_Y = 8;
5  static const uint THREAD_GROUPS_X = 2;
6  static const uint THREAD_GROUPS_Y = 2;
7
8  static const uint THREAD_WIDTH = BUFFER_WIDTH / ( THREAD_GROUP_SIZE_X *
       THREAD_GROUPS_X );
9  static const uint THREAD_HEIGHT = BUFFER_WIDTH / (THREAD_GROUP_SIZE_Y *
       THREAD_GROUPS_Y );
10
11 struct BufferStructColor {
12   float4 color;
13 };
14 RWStructuredBuffer<BufferStructColor> g_OutBuff : register( u0 );
15
16 float GetValue( uint3 threadIDInGroup ) {
17   return ( ( 256 / ( THREAD_GROUP_SIZE_X * THREAD_GROUP_SIZE_Y ) ) * ( (
        threadIDInGroup.y * THREAD_GROUP_SIZE_X ) + threadIDInGroup.x ) ) /
        256.0f;
18 }
19
20 [numthreads( THREAD_GROUP_SIZE_X, THREAD_GROUP_SIZE_Y, 1 )]
21 void main( uint3 groupID : SV_GroupID, uint3 threadIDInGroup :
       SV_GroupThreadID ) {
22   uint startX = ( groupID.x * THREAD_GROUP_SIZE_X + threadIDInGroup.x )
       * THREAD_WIDTH;
23   uint startY = ( groupID.y * THREAD_GROUP_SIZE_Y + threadIDInGroup.y )
       * THREAD_HEIGHT;
24
25   float value = GetValue( threadIDInGroup );
26   for( uint i = 0; i < THREAD_WIDTH; ++i ) {
27     for( uint j = 0; j < THREAD_HEIGHT; ++j ) {
28       float4 color;
```

```
29        color.r = groupID.x == 0 && groupID.y == 0 ? value : 0;
30        color.g = groupID.x == 1 && groupID.y == 0 ? value : 0;
31        color.b = groupID.x == 0 && groupID.y == 1 ? value : 0;
32        color.a = 1;
33
34        if( groupID.x == 1 && groupID.y == 1 ){
35          color = float4( value, value, value, 1 );
36        }
37
38        g_OutBuff[ ( startY + j ) * BUFFER_WIDTH + startX + i ].color =
        color;
39      }
40    }
41 }
```

B.2 Differenz Pixel Shader

```
1  [earlydepthstencil]
2  float PShader( PS_INPUT input) : SV_Target
3  {
4    uint objectHeight = ( input.Pos.z * ( Z_FAR - Z_NEAR ) /
       SNOW_HEIGHT_FACTOR ) * INT_TO_FLOAT_FACTOR;
5    uint snowHeight = g_buffSnowHeight.Load( input.Pos.y *
       SNOW_TEXTURE_WIDTH + SNOW_TEXTURE_WIDTH - ( input.Pos.x +
       SNOW_TEXTURE_WIDTH / 2 ) );
6
7    // /2 is neccessary because otherwise the deformation
8    // would start in the middle of the plane
9    // no max( 0, snowHeight - objectHeight ) because
10   // uints can't get negative -> very large value
11   g_DiffBuff[ input.Pos.y * SNOW_TEXTURE_WIDTH + SNOW_TEXTURE_WIDTH - (
       input.Pos.x + SNOW_TEXTURE_WIDTH / 2 ) ].difference = snowHeight >
       objectHeight ? snowHeight - objectHeight : 0;
12   g_ObjHeightBuff[ input.Pos.y * SNOW_TEXTURE_WIDTH + SNOW_TEXTURE_WIDTH
        - ( input.Pos.x + SNOW_TEXTURE_WIDTH / 2 ) ].difference =
       objectHeight;
13
14   return max( 0, snowHeight - objectHeight );
15 }
```

B.3 Berechnung der Masse

```
1  // ...
2  static const uint THREAD_WIDTH = SNOW_TEXTURE_WIDTH / (
       THREAD_GROUP_SIZE_X * THREAD_GROUPS_X );
3  static const uint THREAD_HEIGHT = SNOW_TEXTURE_WIDTH / (
       THREAD_GROUP_SIZE_Y * THREAD_GROUPS_Y );
4  //...
5  [numthreads( THREAD_GROUP_SIZE_X, THREAD_GROUP_SIZE_Y, 1 )]
6  void main( uint3 id : SV_GroupID, uint3 threadIDInGroup :
       SV_GroupThreadID )
7  {
```

```
 8   int x = ( id.x * THREAD_GROUP_SIZE_X + threadIDInGroup.x ) *
       THREAD_WIDTH;
 9   int y = ( id.y * THREAD_GROUP_SIZE_Y + threadIDInGroup.y ) *
       THREAD_HEIGHT;
10   float mass = 0;
11   float2 position = 0;
12
13   for( uint i = 0; i < THREAD_WIDTH; ++i )
14   {
15     for( uint j = 0; j < THREAD_HEIGHT; ++j )
16     {
17       float tmpMass = g_DensityBuf[ ( y + j ) * SNOW_TEXTURE_WIDTH + x +
        i ].density * g_DiffBuf[ ( y + j ) * SNOW_TEXTURE_WIDTH + x + i ].
       difference;
18       mass += tmpMass;
19       position += float2( x + i, y + j ) * tmpMass;
20     }
21   }
22
23   uint notNeeded;
24   InterlockedAdd( g_OutBuff[ 0 ].mass, (uint) mass, notNeeded );
25   InterlockedAdd( g_OutBuff[ 0 ].positionX, (uint) position.x, notNeeded
        );
26   InterlockedAdd( g_OutBuff[ 0 ].positionY, (uint) position.y, notNeeded
        );
27 }
```

B.4 Ermitteln der Kontur

```
 1 static const float K = 8.0f;
 2 static const uint THREAD_GROUP_SIZE_X = 8;
 3 static const uint THREAD_GROUP_SIZE_Y = 8;
 4 static const uint THREAD_GROUPS_X = 16;
 5 static const uint THREAD_GROUPS_Y = 16;
 6
 7 static const uint THREAD_WIDTH = SNOW_TEXTURE_WIDTH / (
       THREAD_GROUP_SIZE_X * THREAD_GROUPS_X );
 8 static const uint THREAD_HEIGHT = SNOW_TEXTURE_WIDTH / (
       THREAD_GROUP_SIZE_Y * THREAD_GROUPS_Y );
 9
10 groupshared uint area[ THREAD_GROUP_SIZE_Y * THREAD_HEIGHT + 2 ][
       THREAD_GROUP_SIZE_X * THREAD_WIDTH + 2 ];
11
12 cbuffer ObjectBuffer : register( b0 ) {
13   float3 objectVelocity;
14   float  unused_b0;
15 }
16
17 struct BufferStructContour {
18   uint  x;
19   uint  y;
20   float nx;
21   float ny;
```

```
22 };
23 AppendStructuredBuffer<BufferStructContour> g_OutBuff : register( u0 );
24
25 struct BufferStructDifference {
26   uint difference;
27 };
28 RWStructuredBuffer<BufferStructDifference> g_DiffBuff : register( u1 );
29
30
31 struct BufferSnowTotal {
32   uint snowTotal;
33   uint contourFactorTotal;
34 };
35 RWStructuredBuffer< BufferSnowTotal > g_TotalBuff : register( u2 );
36
37 struct BufferSnowHeight {
38   uint height;
39 };
40 RWStructuredBuffer< BufferSnowHeight > g_SnowHeightBuff : register( u3 )
       ;
41
42 struct BufferSnowDensity {
43   uint density;
44 };
45
46 RWStructuredBuffer< BufferSnowDensity > g_SnowDensityBuff : register( u4
       );
47
48 float GetRest( float3 v ) {
49     float3 v2 = float3( v.x, 0, v.z );
50     return 1.0f - dot( v, v2 );
51 }
52
53 float GetDirectionFactor( float3 v, float3 v2 ) {
54   v = normalize( v );
55   v2 = normalize( v2 );
56   float rest = GetRest( v );
57   return max( 0.0f, dot( v, v2 ) ) + rest / K;
58 }
59
60 void LoadValues( uint3 id, uint3 threadIDInGroup ) {
61   int x = ( id.x * THREAD_GROUP_SIZE_X + threadIDInGroup.x ) *
       THREAD_WIDTH;
62   int y = ( id.y * THREAD_GROUP_SIZE_Y + threadIDInGroup.y ) *
       THREAD_HEIGHT;
63
64   uint offsetx = ( threadIDInGroup.x == 0 || threadIDInGroup.x ==
       THREAD_GROUP_SIZE_X - 1 ) ? 1 : 0;
65   uint offsety = ( threadIDInGroup.y == 0 || threadIDInGroup.y ==
       THREAD_GROUP_SIZE_Y - 1 ) ? 1 : 0;
66
67   for( int i = (threadIDInGroup.y == 0 ) ? -1 : 0; i < (int) (
       THREAD_HEIGHT + offsety ); ++i ) {
```

```
68      for( int j = (threadIDInGroup.x == 0 ) ? -1 : 0; j < (int)(
        THREAD_WIDTH + offsetx ); ++j ) {
69        if( y + i < 0 || y + i >= (int)SNOW_TEXTURE_WIDTH || x + j < 0 ||
        x + j >= (int)SNOW_TEXTURE_WIDTH ) {
70          area[ threadIDInGroup.y * THREAD_HEIGHT + i + 1 ][
        threadIDInGroup.x * THREAD_WIDTH + j + 1 ] = 0;
71        } else {
72          uint difference = g_DiffBuff[ ( y + i ) * SNOW_TEXTURE_WIDTH + x
         + j ].difference;
73          area[ threadIDInGroup.y * THREAD_HEIGHT + i + 1 ][
        threadIDInGroup.x * THREAD_WIDTH + j + 1 ] = difference;
74
75          // to not add the difference multiple times
76          if( i >= 0 && i < (int) THREAD_HEIGHT && j >= 0 && j < (int)
        THREAD_WIDTH ) {
77            if( difference > 0 ) {
78              uint val = g_SnowHeightBuff[ ( y + i ) * SNOW_TEXTURE_WIDTH
        + x + j ].height;
79              g_SnowHeightBuff[ ( y + i ) * SNOW_TEXTURE_WIDTH + x + j ].
        height -= min(val, difference );
80            }
81          }
82        }
83      }
84    }
85  }
86
87  bool IsContour( int y, int x ) {
88    uint snow = area[ y + 1 ][ x + 1 ];
89    if( snow > 0 ) {
90      float compressionFactor = 0.15f;//GetCompressionFactor( density );
91
92      float outParam;
93      InterlockedAdd( g_TotalBuff[ 0 ].snowTotal, snow * ( 1 -
        compressionFactor ), outParam );
94      return false;
95    }
96    if( area[ y ][ x ] > 0 ) {
97      return true;
98    }
99    if( area[ y + 1 ][ x ] > 0 ) {
100     return true;
101   }
102   if( area[ y + 2 ][ x ] > 0 ) {
103     return true;
104   }
105   if( area[ y ][ x + 1 ] > 0 ) {
106     return true;
107   }
108   if( area[ y + 2 ][ x + 1 ] > 0 ) {
109     return true;
110   }
111   if( area[ y ][ x + 2 ] > 0 ) {
112     return true;
```

```
113   }
114   if( area[ y + 1 ][ x + 2 ] > 0 ) {
115     return true;
116   }
117   if( area[ y + 2 ][ x + 2 ] > 0 ) {
118     return true;
119   }
120
121   return false;
122 }
123
124 BufferStructContour GetBufferStruct( uint i, uint j ) {
125   BufferStructContour bsc;
126   bsc.x = j;
127   bsc.y = i;
128
129   i += 1;
130   j += 1;
131
132   float2 v = float2( 0, 0 );
133   v += area[ i - 1 ][ j - 1 ] * float2( -1,  1 );
134   v += area[ i - 1 ][ j     ] * float2(  0,  1 );
135   v += area[ i - 1 ][ j + 1 ] * float2(  1,  1 );
136   v += area[ i     ][ j - 1 ] * float2( -1,  0 );
137   v += area[ i     ][ j + 1 ] * float2(  1,  0 );
138   v += area[ i + 1 ][ j - 1 ] * float2( -1, -1 );
139   v += area[ i + 1 ][ j     ] * float2(  0, -1 );
140   v += area[ i + 1 ][ j + 1 ] * float2(  1, -1 );
141
142   normalize( v );
143   bsc.nx = v.x;
144   bsc.ny = v.y;
145
146   return bsc;
147 }
148
149 void CalculateContour( uint3 id, uint3 threadIDInGroup ) {
150   uint startx = threadIDInGroup.x * THREAD_WIDTH;
151   uint starty = threadIDInGroup.y * THREAD_HEIGHT;
152
153   for( uint i = 0; i < THREAD_HEIGHT; ++i ) {
154     for( uint j = 0; j < THREAD_WIDTH; ++j ) {
155       if( IsContour( starty + i, startx + j ) ) {
156         BufferStructContour bsc = GetBufferStruct( starty + i, startx +
       j );
157         uint outParam;
158         InterlockedAdd( g_TotalBuff[ 0 ].contourFactorTotal, (uint)(
       GetDirectionFactor( objectVelocity, float3( bsc.nx, 0, bsc.ny ),
       false ) * INT_TO_FLOAT_FACTOR ), outParam );
159
160         bsc.x = ( id.x * THREAD_GROUP_SIZE_X + threadIDInGroup.x ) *
       THREAD_WIDTH + j;
161         bsc.y = ( id.y * THREAD_GROUP_SIZE_Y + threadIDInGroup.y ) *
       THREAD_HEIGHT + i;
```

```
162
163         g_OutBuff.Append( bsc );
164       }
165     }
166   }
167 }
168
169 [numthreads( THREAD_GROUP_SIZE_X, THREAD_GROUP_SIZE_Y, 1 )]
170 void main( uint3 id : SV_GroupID, uint3 threadIDInGroup :
        SV_GroupThreadID ) {
171   LoadValues( id, threadIDInGroup );
172   GroupMemoryBarrierWithGroupSync();
173   CalculateContour( id, threadIDInGroup);
174 }
```

B.5 Ermitteln der Elemente im Append-Buffer

```
 1 ID3D11Buffer* m_pDataBuffer;
 2 D3D11_BUFFER_DESC desc;
 3 ZeroMemory( &desc,
 4 sizeof(desc) );
 5 m_pContourComputeShader->GetUnorderedAccessView()->GetBuffer()->GetDesc(
        &desc );
 6 desc.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
 7 desc.Usage = D3D11_USAGE_STAGING;
 8 desc.BindFlags = 0;
 9 desc.MiscFlags = 0;
10 DirectX::GetInstance().GetDevice()->CreateBuffer( &desc, NULL, &
       m_pDataBuffer );
11
12 DirectX::GetInstance().GetDeviceContext()->CopyStructureCount(
       m_pDataBuffer, 0, m_pContourComputeShader->GetUnorderedAccessView()
       ->GetUnorderedAccessView() );
13
14 D3D11_MAPPED_SUBRESOURCE resource;
15 DirectX::GetInstance().GetDeviceContext()->Map( m_pDataBuffer, 0,
       D3D11_MAP_READ, 0, &resource );
16 DirectX::GetInstance().GetDeviceContext()->Unmap( m_pDataBuffer, 0 );
17
18
19 ContourElementsConstantBuffer cb;
20 memset( &cb, 0, sizeof( ContourElementsConstantBuffer ) );
21 cb.contourElements = * ( (unsigned int*)resource.pData );
22 DirectX::GetInstance().GetDeviceContext()->UpdateSubresource(
       m_pContourElementsConstantBuffer, 0, NULL, &cb, 0, 0 );
23 m_pDataBuffer->Release();
```

B.6 Vorbereitung zum Beseitigen der Hügel

```
1 static const uint THREAD_WIDTH = SNOW_TEXTURE_WIDTH / (
      THREAD_GROUP_SIZE_X * THREAD_GROUPS_X );
2 static const uint THREAD_HEIGHT = SNOW_TEXTURE_WIDTH / (
      THREAD_GROUP_SIZE_Y * THREAD_GROUPS_Y );
```

```
3 groupshared uint area[ THREAD_GROUP_SIZE_Y * THREAD_HEIGHT + 2 ][
      THREAD_GROUP_SIZE_X * THREAD_WIDTH + 2 ];
4 groupshared uint resultArea[ THREAD_GROUP_SIZE_Y * THREAD_HEIGHT + 2 ][
      THREAD_GROUP_SIZE_X * THREAD_WIDTH + 2 ];
5 groupshared uint objHeightArea[ THREAD_GROUP_SIZE_Y * THREAD_HEIGHT + 2
      ][ THREAD_GROUP_SIZE_X * THREAD_WIDTH + 2 ];
6
7 // ...
8
9 void LoadValues( uint3 id, uint3 threadIDInGroup ) {
10   int x =(id.x*THREAD_GROUP_SIZE_X + threadIDInGroup.x) * THREAD_WIDTH;
11   int y =(id.y*THREAD_GROUP_SIZE_Y + threadIDInGroup.y) * THREAD_HEIGHT;
12   int offsetx = ( threadIDInGroup.x == THREAD_GROUP_SIZE_X - 1 )? 1 : 0;
13   int offsety = ( threadIDInGroup.y == THREAD_GROUP_SIZE_Y - 1 )? 1 : 0;
14
15   for( int i = (threadIDInGroup.y == 0 ) ? -1 : 0; i < (int)
       THREAD_HEIGHT + offsety; ++i ) {
16     for( int j = (threadIDInGroup.x == 0 ) ? -1 : 0; j < (int)
       THREAD_WIDTH + offsetx; ++j ) {
17       resultArea[ threadIDInGroup.y * THREAD_HEIGHT + i + 1 ][
       threadIDInGroup.x * THREAD_WIDTH + j + 1 ] = 0;
18
19       if( y + i < 0 || y + i >= (int)SNOW_TEXTURE_WIDTH || x + j < 0 ||
       x + j >= (int)SNOW_TEXTURE_WIDTH ) {
20         area[ threadIDInGroup.y * THREAD_HEIGHT + i + 1 ][
       threadIDInGroup.x * THREAD_WIDTH + j + 1 ] = 10000000;
21         objHeightArea[ threadIDInGroup.y * THREAD_HEIGHT + i + 1 ][
       threadIDInGroup.x * THREAD_WIDTH + j + 1 ] = 10000000;
22       } else {
23         objHeightArea[ threadIDInGroup.y * THREAD_HEIGHT + i + 1 ][
       threadIDInGroup.x * THREAD_WIDTH + j + 1 ] = g_ObjHeightBuff[ ( y +
       i ) * SNOW_TEXTURE_WIDTH + x + j  ].height;
24         area[ threadIDInGroup.y * THREAD_HEIGHT + i + 1 ][
       threadIDInGroup.x * THREAD_WIDTH + j + 1 ] = g_SourceSnowHeightBuff[
        ( y + i ) * SNOW_TEXTURE_WIDTH + x + j ].height;
25       }
26     }
27   }
28 }
```

B.7 Konstante Tessellierung

B.7.1 `struct`-Definitionen

```
1 struct VS_RenderSceneInput {
2     float3 f3Position   : POSITION;
3     float3 f3Normal     : NORMAL;
4     float2 f2TexCoord   : TEXCOORD;
5 };
6
7 struct HS_Input {
8     float3 f3Position   : POSITION;
9     float3 f3Normal     : NORMAL;
```

```
10     float2 f2TexCoord   : TEXCOORD;
11 };
12
13 struct HS_ConstantOutput {
14     float fTessFactor[3]    : SV_TessFactor;
15     float fInsideTessFactor : SV_InsideTessFactor;
16 };
17
18 struct HS_ControlPointOutput {
19     float3 f3Position    : POSITION;
20     float3 f3Normal      : NORMAL;
21     float2 f2TexCoord    : TEXCOORD;
22 };
23
24 struct PS_RenderSceneInput {
25     float4 f4Position   : SV_Position;
26     float2 f2TexCoord   : TEXCOORD0;
27     float4 f4Diffuse    : COLOR0;
28     float4 f4PosLight   : TEXCOORD1;
29 };
30
31 struct PS_RenderOutput {
32     float4 f4Color      : SV_Target0;
33 };
```

B.7.2 Vertex Shader

```
1 HS_Input VS_RenderSceneWithTessellation( VS_RenderSceneInput I ) {
2     HS_Input O;
3     O.f3Position = (float3) mul( float4( I.f3Position, 1.0f ),
      g_f4x4World );
4     O.f3Normal = normalize( mul( I.f3Normal, (float3x3)g_f4x4World ) );
5     O.f2TexCoord = I.f2TexCoord;
6     return O;
7 }
```

B.7.3 Hull Shader

```
1 HS_ConstantOutput HS_PNTrianglesConstant( InputPatch<HS_Input, 3> I ) {
2     HS_ConstantOutput O = (HS_ConstantOutput)0;
3     O.fTessFactor[0] = O.fTessFactor[1] = O.fTessFactor[2] =
      g_TessellationFactorEdge;
4     O.fInsideTessFactor = ( O.fTessFactor[0] + O.fTessFactor[1] + O.
      fTessFactor[2] ) / 3.0f;
5     return O;
6 }
7
8 [domain("tri")]
9 [partitioning("fractional_odd")]
10 [outputtopology("triangle_cw")]
11 [patchconstantfunc("HS_PNTrianglesConstant")]
12 [outputcontrolpoints(3)]
13 HS_ControlPointOutput HS_PNTriangles( InputPatch<HS_Input, 3> I, uint
       uCPID : SV_OutputControlPointID ) {
```

```
14     HS_ControlPointOutput O = (HS_ControlPointOutput)0;
15     O.f3Position = I[uCPID].f3Position;
16     O.f3Normal = I[uCPID].f3Normal;
17     O.f2TexCoord = I[uCPID].f2TexCoord;
18     return O;
19 }
```

B.7.4 Domain Shader

```
 1 [domain("tri")]
 2 PS_RenderSceneInput DS_PNTriangles( HS_ConstantOutput HSConstantData,
       const OutputPatch<HS_ControlPointOutput, 3> input, float3
       f3BarycentricCoords : SV_DomainLocation ) {
 3     PS_RenderSceneInput O = (PS_RenderSceneInput)0;
 4     float3 f3Position = f3BarycentricCoords.x * input[ 0 ].f3Position +
 5                         f3BarycentricCoords.y * input[ 1 ].f3Position +
 6                         f3BarycentricCoords.z * input[ 2 ].f3Position;
 7     O.f2TexCoord = f3BarycentricCoords.x * input[ 0 ].f2TexCoord +
 8                    f3BarycentricCoords.y * input[ 1 ].f2TexCoord +
 9                    f3BarycentricCoords.z * input[ 2 ].f2TexCoord;
10     float3 f3Normal = f3BarycentricCoords.x * input[ 0 ].f3Normal +
11                       f3BarycentricCoords.y * input[ 1 ].f3Normal +
12                       f3BarycentricCoords.z * input[ 2 ].f3Normal;
13     f3Normal = normalize( f3Normal );
14     O.f4Diffuse.rgb = (g_f4MaterialDiffuseColor * g_f4LightColor * max
       (0, dot( f3Normal, g_f4LightDir.xyz )) + g_f4MaterialAmbientColor ).
       xyz;
15     O.f4Diffuse.a = 1.0f;
16
17     float deformation = g_txSnowDeformation.SampleLevel( g_SampleLinear,
        O.f2TexCoord, 0 ).x;
18     f3Position.y += deformation * g_maxSnowDeformation;
19
20     O.f4Position = mul( float4( f3Position.xyz, 1.0 ),
       g_f4x4ViewProjection );
21     O.f4PosLight = mul( float4( f3Position.xyz, 1.0 ),
       g_f4x4LightViewProjection );
22
23     return O;
24 }
```

B.7.5 Pixel Shader

```
1 PS_RenderOutput PS_RenderSceneTextured( PS_RenderSceneInput I )
2 {
3     PS_RenderOutput O;
4     O.f4Color = g_f4MaterialAmbientColor * g_txDiffuse.Sample(
      g_SampleLinear, I.f2TexCoord );
5     return O;
6 }
```

B.8 Entfernungs-abhängige Tessellierung

```
1 float GetDistanceAdaptiveScaleFactor( float3 f3Eye, float3
      f3EdgePosition0, float3 f3EdgePosition1, float fMinDistance, float
      fRange ) {
2     float3 f3MidPoint = ( f3EdgePosition0 + f3EdgePosition1 ) * 0.5f;
3     float fDistance = distance( f3MidPoint, f3Eye ) - fMinDistance;
4     float fScale = 1.0f - saturate( fDistance / fRange );
5     return fScale;
6 }
7
8 HS_ConstantOutput HS_PNTrianglesConstant( InputPatch<HS_Input, 3> I )
9 {
10    HS_ConstantOutput O = (HS_ConstantOutput)0;
11    O.fTessFactor[ 0 ] = 1 + GetDistanceAdaptiveScaleFactor( g_f4Eye.xyz
      , I[1].f3Position, I[2].f3Position, g_MinDistanceForMaxTessellation,
       g_TesselationDistance ) * g_TessellationFactorEdge;
12    O.fTessFactor[ 1 ] = 1 + GetDistanceAdaptiveScaleFactor( g_f4Eye.xyz
      , I[2].f3Position, I[0].f3Position, g_MinDistanceForMaxTessellation,
       g_TesselationDistance ) * g_TessellationFactorEdge;
13    O.fTessFactor[ 2 ] = 1 + GetDistanceAdaptiveScaleFactor( g_f4Eye.xyz
      , I[0].f3Position, I[1].f3Position, g_MinDistanceForMaxTessellation,
       g_TesselationDistance ) * g_TessellationFactorEdge;
14    O.fInsideTessFactor = ( O.fTessFactor[0] + O.fTessFactor[1] + O.
      fTessFactor[2] ) / 3.0f;
15
16    return O;
17 }
```

Anhang C

Inhalt der CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

C.1 Diplomarbeit

Pfad: /

Diplomarbeit.pdf	diese Diplomarbeit

C.2 Quellcode

Pfad: /Quellcode/

SnowSimulation/	Quellcode des Haupt-Projektes
SnowSimulation/bin/ .	ausführbare exe-Datei (DX11 benötigt)
AlgorithmTests/	Quellcode diverser Hilfsprojekte zum Testen von Algorithmen

C.3 Bilder

Pfad: /Bilder/

*.png	Bilder der Simulation

C.4 Quellen

Pfad: /Quellen/

WebQuellen/	gespeicherte Web-Quellen
Sonstige/	sonstige Quellen

Anhang D

Symbole

ρ	Dichte
ρ'	neue Dichte
$\rho_{\min}, \rho_{\max}$	minimale und maximale Dichte in der Simulation
$\phi_{\max}$	maximaler Höhenunterschied zweier Nachbarfelder
ϵ	Verformung einer Feder/eines Dämpfungsglieds
σ	Druck
η	Wert für die Zähflüssigkeit eines Stoffes
a, b	Felder in einem Buffer
B	Bodenhöhe
c	Gesamtdifferenz eines Feldes zu seinen niedrigeren Nachbarn
d	Richtungsanteil
$d_{\min}$	minimale Entfernung für maximale Tessellierung
d_{n}	Unterschied zweier Nachbarfelder
$\boldsymbol{d}_{\mathrm{x}}, \boldsymbol{d}_{\mathrm{z}}$	Höhendifferenz entlang x- bzw z-Richtung
D	Differenzinformationen
D'	gruppeninterner Speicher für Differenzen
E	Elastizitätsmodul
$\boldsymbol{F}$	Gegenkraft durch Widerstand
$\boldsymbol{g}$	Erdbeschleunigung
G	Objekt für den Boden
G_{I}	Gruppenindex eines Threads
$G_{\mathrm{x}}, G_{\mathrm{y}}$	Anzahl der Thread-Gruppen
H	Gesamthöhe
i, j	Hilfs-/Zählvariablen
I	beliebiger Buffer
k	Gewichtung des Restanteils in 3D
k_{T}	Gesamtgewichtung (Richtungsfaktoren) der Kontur
K	Kontur
L	Hilfsfunktion zum Ermitteln der Kontur
n, m	Anzahl der niedrigeren Nachbarfelder

m	Masse
$\boldsymbol{n}$	Vertexnormale
M, N	Dimensionen der Buffer (D, S, H, B, ρ, usw.)
O	Objekt
$\boldsymbol{p}_1, \boldsymbol{p}_2$	Anfangs- und Endpunkt einer Kante
$\boldsymbol{p}_\mathrm{c}$	Kameraposition
Q	Komprimierungsfaktor
r	Restanteil
r_t	Entfernung ab d_min, in der interpoliert wird
R	Listen von `Renderable`-Objekten
s	zu verschiebende Schneemenge
s_f	Faktor für die `float`–`uint` Umwandlung
$s_\mathrm{L}, s_\mathrm{R}$	Richtungsfaktoren in 2D
s_T	Gasamte verdränge Schneemenge
s_w	Richtungsfaktor in 3D
S	Schneehöhe
$S_\mathrm{x}, S_\mathrm{y}$	Startpunkte eines Threads beim Abarbeiten eines Buffers
$\boldsymbol{S}_\mathrm{x}, \boldsymbol{S}_\mathrm{y}$	Kantenfilter-Matrizen
t	Schnee pro Nachbar
t_f	entfernungsabhängige Tessellierung
t_max	maximaler Tessellierungsfaktor
T	kinetische Energie
T_I	Thread ID innerhalb der Gruppe
$T_\mathrm{w}, T_\mathrm{h}$	Breite und Höhe des abzuarbeitenden Bereichs eines Threads
$T_\mathrm{x}, T_\mathrm{y}$	Anzahl der Threads pro Gruppe
U	Höhe der Objektunterkanten
$\boldsymbol{v}$	Geschwindigkeit des Objektes
$\boldsymbol{v}_\mathrm{x}, \boldsymbol{v}_\mathrm{xz}$	$\boldsymbol{v}$ ohne y-Anteil
v	$\lvert\boldsymbol{v}\rvert$ – Länge des Richtungsvektors
V	Volumen
w	Zielrichtung für Richtungsfaktor
W	Verteilungsfaktor
$x, \boldsymbol{x}$	Position im Buffer in 2D (Skalar) bzw. 3D (Vektor)
x_0	Position an der die Gegenkraft wirkt
x, y, z	Positionen
$Z_\mathrm{F}, Z_\mathrm{N}$	Far und Near Clipping Plane

Quellenverzeichnis

Literatur

[1] Anthony S. Aquilio. „A framework for dynamic terrain with application in off-road ground vehicle simulations“. Diss. Atlanta, GA, USA: Georgia State University, 2006.

[2] Nathan Bell, Yizhou Yu und Peter J. Mucha. „Particle-based simulation of granular materials“. In: *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. (Los Angeles, California). SCA '05. New York, NY, USA: ACM, 2005, S. 77–86.

[3] John P. Carter, John Robert Booker und Edward Hughson Davis. „Finite deformation of an elasto-plastic soil“. In: *International Journal for Numerical and Analytical Methods in Geomechanics* 1.1 (1977), S. 25–43.

[4] Paul Fearing. „Computer modelling of fallen snow“. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. SIGGRAPH'00. New York: ACM, 2000, S. 37–46.

[5] Xin Li und J. Michael Moshell. „Modeling soil: realtime dynamic models for soil slippage and manipulation“. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '93. Anaheim, CA: ACM, 1993, S. 361–368.

[6] Malcolm Mellor. „A Review of Basic Snow Mechanics“. In: *Proceedings of the Grindelwald Symposium on Snow Mechanics*. Grindelwald: International Association of Hydrological Sciences (IAHS), 1975, S. 251–291.

[7] Christian Siegrist. „Measurement of Fracture Mechanical Properties of Snow and Application to Dry Snow Slab Avalanche Release“. Diss. University of Bern, 2006.

[8] Robert W. Sumner, James F. O'Brien und Jessica K. Hodgins. „Animating Sand, Mud, and Snow“. In: *Computer Graphics Forum* 18.1 (1999), S. 17–26.

[9] Demetri Terzopoulos und Kurt Fleischer. „Modeling inelastic deformation: viscolelasticity, plasticity, fracture“. In: *SIGGRAPH Comput. Graph.* 22 (4 Juni 1988), S. 269–278.

[10] Dong Wang u. a. „Real-Time GPU-Based Visualization of Tile Tracks in Dynamic Terrain“. In: *2009 International Conference on Computational Intelligence and Software Engineering*. Wuhan, China, 2009, S. 1–4.

Online-Quellen

[11] URL: http://hondajvx.files.wordpress.com/2008/04/maddennfl_snow_improvement.jpg.

[12] URL: http://de.wikipedia.org/wiki/GPGPU.

[13] URL: http://www.its.caltech.edu/~atomic/snowcrystals/primer/morphologydiagram.jpg.

[14] URL: http://de.wikipedia.org/wiki/Arithmetisch-logische_Einheit.

[15] URL: http://images.anandtech.com/doci/4061/Cayman%20block%20diagram.png.

[16] URL: http://msdn.microsoft.com/en-us/library/ff569022%28v=vs.85%29.aspx.

[17] URL: http://de.wikipedia.org/wiki/DirectX.

[18] URL: http://www.nvidia.com/object/tessellation.html.

[19] URL: http://de.wikipedia.org/wiki/Viskoelastizität.

[20] URL: http://de.wikipedia.org/wiki/Rheologie.

[21] URL: http://msdn.microsoft.com/en-us/library/ff471356%28v=VS.85%29.aspx.

[22] URL: http://de.wikipedia.org/wiki/Schnee\#Dichte.

[23] Bill Bilodeau. *Efficient Compute Shader Programming*. 2011. URL: http://developer.amd.com/gpu_assets/Efficient%20Compute%20Shader%20Programming.pps.

[24] Chas. Boyd. *DirectX 11 Compute Shader*. 2008. URL: http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf.

[25] Wil Harris. 2005. URL: http://www.bit-tech.net/hardware/2005/07/25/guide_to_shaders/1.

[26] Andy Keane. 2010. URL: http://blogs.nvidia.com/2010/06/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel/.

[27] Gek Siong Low. *Understanding Realism in Computer Games through Phenomenology*. 2001. URL: http://xenon.stanford.edu/~geksiong/papers/cs378/cs378paper.htm.

[28] Martina Píková u. a. *An Introduction to the Basic Properties and Mechanics of Snow*. 2008. URL: http://czu.kbx.cz/BOKU/SNOW%20AND%20AVALANCHE/Our%20prezentation/written/02_Basics_of_snow_mechanics.pdf.

[29] *R2VB dynamic terrain deformation*. 2005. URL: http://www.lynxengine.com/old-site/atisdk/R2VB-Footprint.pdf.

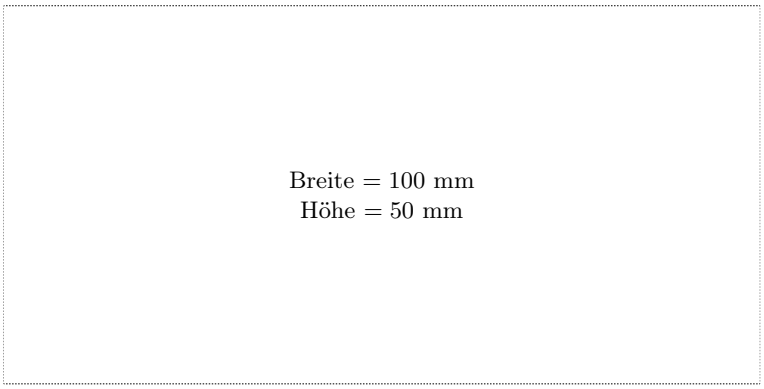
[30] Ryan Shrout. 2010. URL: http://www.pcper.com/reviews/Graphics-Cards/AMD-Radeon-HD-6970-and-6950-Review-Cayman-Architecture-Revealed.

[31] Nick Thibieroz. *Shader Model 5.0 and Compute Shader*. 2009. URL: http://developer.amd.com/gpu_assets/Shader%20Model%205-0%20and%20Compute%20Shader.pps.

[32] Alexandre Valdetaro u. a. *Understanding Shader Model 5.0 with DirectX 11*. 2010. URL: http://xtunt.com/wp-content/uploads/2010/11/Understanding_Shader_Model_5-format_rev4_web.pdf.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —