UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

COMPUTERVISUALISTIK

Darstellung von Sand durch Partikelsimulation

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Kai Ludwig

Betreuer: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im April 2007

Erklärung

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☒	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☒	☐

.....
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung

1.1 Ziel der Arbeit

Ziel der vorliegenden Studienarbeit war die Darstellung von Sand. Dabei wurde der Schwerpunkt weniger auf realitätsgetreue Visualisierung gelegt, sondern es wurde primär versucht, den Eindruck von fließendem Sand zu vermitteln.

Dieser sollte durch die Simulation von Fließverhalten und Aufschüttung des Sandes erreicht werden.

Modelliert werden die einzelnen Körner mithilfe eines Partikelsystems. Da die Simulation von Sand sehr aufwändig ist, sollten für diese Studienarbeit effiziente Datenstrukturen und Algorithmen für die Verwaltung der Sandkörner entwickelt werden. Die Kollisionserkennung ist bei derartigen Datenmengen ebenfalls sehr zeitraubend. Deshalb sollten auch hierfür geeignete Algorithmen erstellt werden.

Um die Ergebnisse der Arbeit zu demonstrieren, sollte eine entsprechende, graphisch ansprechende Beispielanwendung implementiert werden.

1.2 Charakter der Arbeit

Natürlich war es ein wichtiges Ziel der Arbeit, die gestellten Aufgaben zu lösen und die Anforderungen zu erfüllen. Doch hatte die Studienarbeit für mich persönlich noch weitere Ziele.

Zum einen war es eine Herausforderung, mich mit der Organisation eines Projekts in derartiger Größenordnung auseinanderzusetzen, und dabei das Gelernte aus Vorlesungen wie *Softwaretechnik* und *Programmierung* in der Praxis anzuwenden.

Weiter war die Studienarbeit natürlich auch eine ideale Gelegenheit, mich tiefer in die Programmiersprache *C++* einzuarbeiten und auch meine Programmierkenntnisse im Allgemeinen weiter auszubauen.

Etwas, was mir ebenfalls sehr am Herzen lag und einen Großteil des Charakters der Arbeit bestimmt, war es, mehrere Dinge einfach mal *auszuprobieren*. Es ist wichtig hierbei zu erwähnen, dass die Studienarbeit quasi ohne Zuhilfenahme von Literatur entstanden ist. Sämtliche Algorithmen wurden von mir selbst speziell für diese Arbeit entwickelt. Natürlich taten sich dadurch einige Hürden auf, doch konnte ich die Probleme alle entweder lösen oder umgehen.

Während der Recherche zu Beginn der Arbeit bin ich auf das Paper *Particle Based Simulation Of Granular Materials*[BYM05] gestoßen, dessen Inhalt genau auf meine Anforderungen und Aufgabenstellungen abgezielt zu sein schien. Auch die Bilder der Resultate waren äußerst ansprechend. Doch die Tatsache, dass hier die Rechenzeiten im Bereich von 3 bis 26 Minuten pro Frame lagen, führte dazu, dass ich beschloss, Einbüßen hinsichtlich Qualität und physikalischer Korrektheit hinzunehmen und dafür die Simulation in Echtzeit zu berechnen, um wie bereits erwähnt den Eindruck von fließendem deformierbaren Sand zu erwecken. Hierfür hatte ich mir dann auch vorgenommen, auf Literatur zu verzichten und mir die Algorithmen lieber selbst auszudenken.

Die Studienarbeit hat also einen starken experimentellen Charakter, der sich auch an vielen Stellen noch zeigen wird.

1.3 Bestandteile der Arbeit

Im Laufe der Studienarbeit wurden all diese Teilaufgaben gelöst und implementiert.

- Es wurde eine einfache Simulation für das Fließverhalten von Sand entwickelt. Hierzu gehört, dass sich der Sand entsprechend der Schwerkraft nach unten bewegt. Dazu kommt eine Kollisionserkennung, die, in Wechselwirkung mit der Fließbewegung, dazu führt, dass der Sand aufgeschüttet wird.
- Es wurde die Möglichkeit implementiert, dass 3D-Modelle aus Dateien eingelesen werden können. Zusätzlich wurde ein Algorithmus entwickelt, mit dem diese 3D-Modelle effizient mit Sand gefüllt werden, wodurch quasi Sandskulpturen dargestellt werden können.
- Diese beiden Punkte ergeben zusammen, dass Sanduhren von praktisch beliebiger Gestalt erstellt und simuliert werden können, in denen der Sand nun wirklich von oben nach unten fließt und sich dann im unteren Teil der Form aufschüttet.
- Es ist alternativ auch möglich, dass Sandskulpturen eingelesen und dargestellt werden, die dann in Sand zerfließen, als ob man eine sie stützende Hülle plötzlich entfernen würde.
- Es wurde auch integriert, dass man mit Sand malen kann. Hierzu können einfach mit der Maus Linien und Kurven gezeichnet werden, wobei dann Sandspuren den Bewegungen folgen, die sich natürlich auch wieder aufschütten, falls man mehrfach mit der Maus über dieselbe Stelle zeichnet.
- Zunächst über unzählige kryptische Tastaturkürzel gesteuert wurde im Sinne einer intuitiveren Steuerbarkeit auch eine GUI[1] erstellt, mit der nun die Kontrolle über das Programm behalten werden kann.

[1] Graphical User Interface (dt. graphische Benutzer-Oberfläche)

2 Grundlagen

2.1 Partikelsysteme

Unter *Partikelsystemen* versteht man insbesondere in der Computergraphik eine Technik, stark dynamische Objekte zu simulieren. Hierbei wird meist eine große Anzahl geometrischer Primitive und deren zeitliche und räumliche Bewegung verwaltet.

Meistens werden sie zur Simulation von natürlichen Phänomenen verwendet, wie Feuer, Rauch oder Wasser, wobei sich äußerst realistische Ergebnisse erzielen lassen. Abgesehen von wissenschaftlichen Simulationen werden sie auch in der Film- oder Spielebranche immer öfters zur Generierung spektakulärer Effekte eingesetzt.

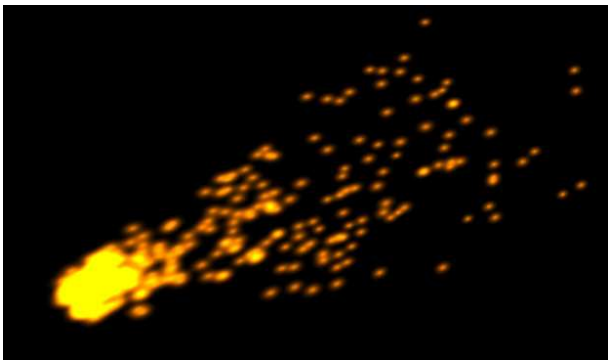


Abbildung 1: In *Cinema4D* mit einem Partikelsystem erstellter Kometenschweif

Partikel

Die simulierten Objekte werden dabei von *Partikeln* repräsentiert, die dann durch ihr Zusammenwirken wie eine größere dynamische Einheit. Diese besitzen verschiedene Eigenschaften, um sie zu beschreiben. Grundlegend sind dies ihre Position, ihre (gerichtete) Geschwindigkeit sowie ein gewisses „Alter“. Oft kommen noch weitere Attribute wie Farbe, Größe oder Form dazu. Bei jedem Simulationsschritt wird dabei ihr Alter erhöht und ihre Position und Geschwindigkeit an die neue Situation angepasst. Über diese Lebensdauer werden oft auch die anderen Attribute verändert, um so das Aussehen der Partikel zu kontrollieren. In vielen Partikelsystemen werden die Teilchen mit der Zeit zum Beispiel kleiner, bis sie dann letztendlich ganz verschwinden. Überschreitet die Lebenszeit eines Partikels einen gewissen Schwellwert, so „stirbt“es. Meistens wird dann vom *Emitter* dafür allerdings ein neues erzeugt, so dass die Anzahl konstant bleibt.

Emitter

Von einem *Emitter* werden die Partikel ausgesandt, weswegen mindestens einer im System vorhanden sein sollte. Der Emitter wird frei in der Welt platziert und

gibt damit die Startposition der Partikel, sowie deren initiale Richtung an. Meistens sind sie rechteckige oder kugelförmig. Prinzipiell ist aber jede Form denkbar. Für eine gleichmäßig Verteilung bei der „Geburt“der Partikel muss lediglich die Emissionsfunktion entsprechend angepasst werden.

Ein großer Vorteil der Partikelsysteme ist ihre an sich sehr einfache aber dennoch flexible Handhabung. Als Partikel kann nahezu jedes beliebige Objekt dienen, wenn auch im Regefall meistens kleine Kugeln verwendet werden. In Abb. 1 wurden beispielsweise Lichtquellen in die Welt versendet. Des Weiteren können sie von jeder Art von externen Kräften beeinflusst werden, wie zum Beispiel Erdanziehung oder Wind. Durch deren Hinzufügen oder Weglassen kann das System genau kontrolliert und auf die entsprechenden Bedürfnisse der jeweiligen Anwendung angepasst werden.

Auch durch die Bestimmung der Partikelanzahl sowie der Auswirkung des Alters auf die Attribute der Teilchen kann weiter Einfluss auf das System genommen werden.

2.2 Sand

Auf den ersten Blick betrachtet, scheinen sich Partikelsysteme gut zur Simulation von Sand zu eignen, da auch dieser aus vielen Teilchen besteht. Ein Problem dabei ist aber, dass die Sandkörner untereinander, im Gegensatz zu beispielsweise Feuerpartikeln, in Wechselwirkung treten und kollidieren, wodurch erst zur Aufschüttung eines Sandberges kommt.

Sand ist an sich eine physikalisch komplexe Materie, da sie sowohl das Verhalten von Festkörpern als auch von Flüssigkeit zu haben scheint. Auf den ersten Blick scheint das Rieseln in einer Sanduhr nichts anderes zu sein als das Ausgießen einer Flüssigkeit. Der Unterschied liegt jedoch darin, dass im Falle einer Flüssigkeit die Menge beim Austritt von der Höhe der Flüssigkeitssäule über der Öffnung abhängt. Bei Sand hingegen rieselt immer die gleiche Menge pro Zeiteinheit, weswegen sich Sand überhaupt so gut zur Zeitmessung eignet.

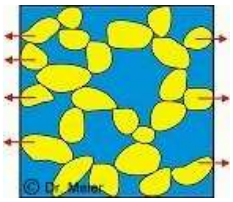


Abbildung 2: Druckverteilung in sandgefüllten Gefäßen[San]

Die besondere Art der Kraftübertragung ist der Grund für diese konstante Fließ-Geschwindigkeit. Sie erfolgt über die Berührungspunkte zwischen den Körnern, wodurch diese Kontakte eine Art Netzwerk bilden, das den Druck auf die Seitenwände leitet. Hierdurch werden Sandschichten, die weiter unten liegen, vom Gewicht des darüber liegenden Sandes entlastet. Damit bleibt der mittlere Druck über der Sanduhrverengung auch bei verändernder Sandhöhe konstant.

Die Simulation solch komplexer Materien ist allerdings sehr zeitraubend, was auch [BYM05, S. 8] belegt. Für diese Studienarbeit wurden diese Kräfte weitgehend außer Acht gelassen. Zwar wurde zu Beginn noch versucht, alles physikalisch korrekt zu berechnen, was aber doch letztendlich als zu aufwändig angesehen wurde. Deshalb wurde dann im Folgenden ein komplett anderer Ansatz gewählt.

3 Technik

3.1 Erste Gehversuche

3.1.1 Ohne Kollision der Partikel

Um ein einfaches Partikelsystem zu implementieren bedarf es nicht viel. So konnte schon sehr früh im Rahmen der Arbeit eine erste Version erstellt werden. Jedes Partikel wurde dabei noch als eigenes Objekt einer Klasse *Particle* angelegt. Diese enthielten jeweils wiederum Objekte vom Typ *Vector3D*, um die Position des Partikels, dessen (gerichtete) Geschwindigkeit und die darauf wirkende Kraft modellieren zu können. Dargestellt wurden die Partikel als einfache Kugeln.

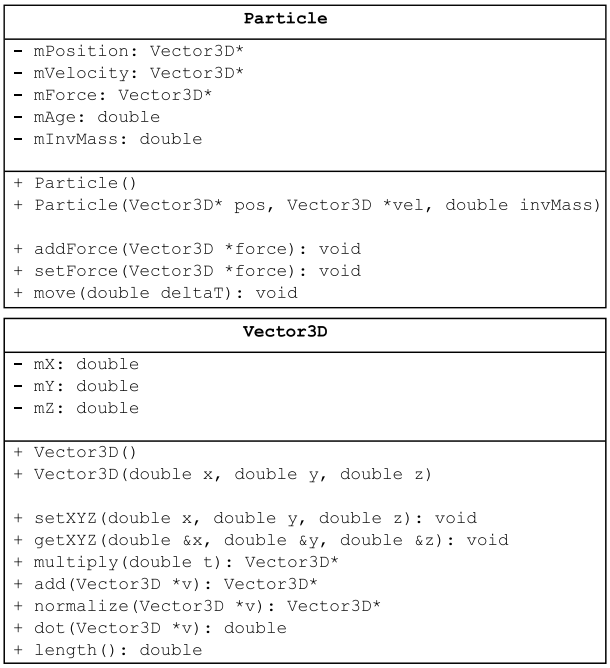


Abbildung 3: Einblicke in die Klassen Particle und Vector3D

Um die Simulation noch interessanter zu gestalten wurde noch ein Objekt eingefügt, mit dem die Partikel kollidieren konnten - die Möglichkeit der Kollision der Partikel untereinander wurde in diesem einfachen System noch nicht implementiert. Dieses Objekt war eine aus zwei Dreiecken bestehende, schräg im Raum liegende rechteckige Fläche.

Um die Bewegung der Partikel zu berechnen werden in jedem Simulationsdurchlauf die folgenden Schritte ausgeführt.

Auf Tastendruck können neue Partikel erzeugt werden, die nun ebenfalls auf die Fläche fielen. Hierzu wurden einfach neue Partikel-Objekte generiert, ihre Geschwindigkeit mit $\vec{v} = (0, 0, 0)$ und ihre Position mit zufälligen Werten innerhalb eines gewissen Bereichs oberhalb der Platte initialisiert. Natürlich wird das System dabei mit wachsender Anzahl an Partikeln zunehmend langsamer, da in jedem Durchlauf die Berechnung für jedes Partikel neu durchgeführt werden muss.

```
simulation(delta_t) {
  for every active particle p do
    p.force = gravity;
    p.force += airFriction;
    for every triangle t do
      p.collision(t);
    end do
    p.move(delta_t);
  end do
}

collision(triangle t) {
  if(there would be a collision)
    reflect(p.velocity);
    multiply(p.velocity, groundAsorption);
  end if
}
```

Abbildung 4: Simulationsablauf

Da in diesem einfachen System eine Kollision der Partikel untereinander noch nicht vorgesehen war, konnte eine simple Optimierung durchgeführt werden, so dass auch bei hoher Anzahl der Partikel flüssige Simulationen erreicht werden können.

Für die Verwaltung der Partikel wird die Datenstruktur *std::vector* verwendet. Für die Beschleunigung werden statt wie bisher einer Liste dieses Typs nun zwei verwendet, *particlesActive* und *particlesInactive*. Wenn ein Partikel neu erzeugt wird, wird es zunächst der aktiven Liste hinzugefügt. Unterschreitet die y-Koordinate der Partikelposition einen gewissen Schwellwert, der außerhalb des sichtbaren Bereichs liegt, wird es von der Liste gestrichen. Weitere Simulationsschritte wären für dieses Objekt unnötig, da man es nicht mehr sehen kann. Zudem wird jedes Partikel der aktiven Liste, dessen Geschwindigkeit $\vec{v}$ betragsmäßig eine gewisse Grenze nahe 0 unterschreitet, von *particlesActive* gelöscht und dafür auf der inaktiven Liste hinzugefügt.

Die Optimierung liegt nun wie in Abb. 4 angedeutet darin, dass die Simulationsberechnungen nur noch für *aktive* Partikel durchgeführt werden. Da es keine Kollision zwischen den einzelnen Objekten gibt, werden in diesem einfachen System Partikel, die einmal durch Reibung auf der Platte zur Ruhe gekommen sind, nicht mehr von ihrer Position fortbewegt und weitere Simulationsschritte müssen deshalb hierfür auch nicht betrachtet werden.

Ob eine Kollision zwischen Kugel und Dreieck stattgefunden hat, wird wie in Abb. 5 durch folgenden Zusammenhang errechnet.

Wäre die Länge des Vektors $\vec{d}$, also der lotrechte Abstand von Partikel zu Fläche, im nächsten Simulationsschritt kleiner als der Radius der Kugel, hat eine Kollision stattgefunden. Mit $\vec{v_{out}} = \vec{v_{in}} - 2 \cdot \vec{n} \cdot (\vec{n} \circ \vec{v_{in}})$ wird dann der an der Dreiecksnormale reflektierte Geschwindigkeitsvektor des Partikels berechnet. Multi-

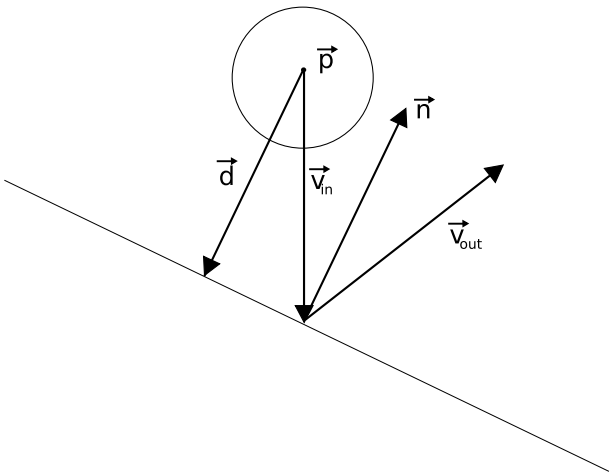


Abbildung 5: Kollision zwischen Kugel und Fläche

pliziert mit einem gewissen Absorptionsfaktor μ (in der Implementation: 0.5) für die Fläche ist dies nun die neue Geschwindigkeit für das Partikel nach der Kollision.

Dies ist eine sehr einfache Kollisionsabfrage und behandelt streng genommen nicht den Fall einer Kollision zwischen Kugel und Dreieck sondern lediglich zwischen Kugel und Ebene. Ob die Kollision auch wirklich mit dem Dreieck selbst und nicht nur mit der Ebene auf der es liegt stattgefunden hat, wird überprüft, indem man testet, ob der senkrecht auf die Ebene projizierte Mittelpunkt des Partikels (in Abb. 5: $\vec{p} + \vec{d}$) innerhalb der Grenzen des Dreiecks liegt. Dadurch wird hier auch erst ermöglicht, dass Partikel an der Seite am Rand der Fläche herunterfallen können.

Die Luftreibung wird simuliert, indem eine Kraft auf das Partikel draufgerechnet wird, die der negativen Geschwindigkeit multipliziert mit einem gewissen Reibungsfaktor η (in der Implementation: 0.05) entspricht, also $\vec{f_{new}} = \vec{f_{old}} - \eta \cdot \vec{v}$.

Als letztes werden die Partikel dann mit der Euler-Integration[Mül04, Folie 35] bewegt. Die Geschwindigkeit ändert sich mit $\vec{v_{new}} = \vec{v_{old}} + \vec{f} \cdot \Delta t \cdot \frac{1}{m}$ und die neue Position wird mit $\vec{p_{new}} = \vec{p_{old}} + \vec{v_{old}} \cdot \Delta t$ berechnet.

Das Zwischenergebnis bis hierhin ist in Abb. 6 dargestellt.

3.1.2 Mit Kollision der Partikel

Das vorliegende System konnte ein sehr einfaches Fließverhalten an sich schon gut simulieren. Doch schien es mehr den Anschein von Wasser als von Sand zu erzeugen, da noch keine Aufschüttung der Partikel stattfand. Mithilfe von [BYM05] wurde nun versucht, die Kollision der Partikel zu berechnen.

Bezogen auf Abb. 7 wird die Kraft auf das Partikel an der Position $\vec{x_1}$ nach

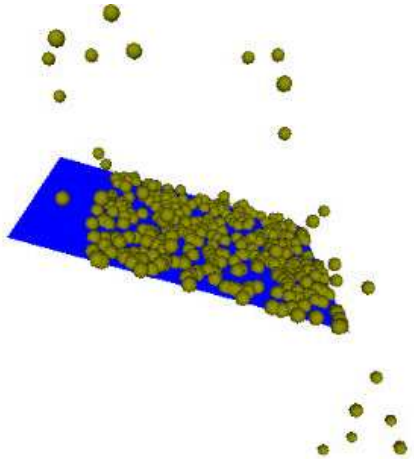


Abbildung 6: Erstes Partikelsystem in Aktion

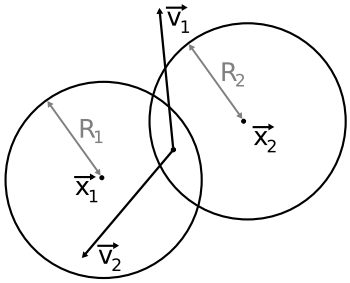


Abbildung 7: Kollision zwischen zwei Partikeln

[BYM05, S.3f] wie folgt berechnet.

Die Überlappung ξ der Partikel sowie die normierte Verbindungslinie $\vec{N}$ der Partikelzentren sind definiert als $\xi = max(0, R_1 + R_2 - \|\vec{x_2} - \vec{x_1}\|)$ und $\vec{N} = \frac{\vec{x_2} - \vec{x_1}}{\|\vec{x_2} - \vec{x_1}\|}$. Die relative Geschwindigkeit am Kontaktpunkt der beiden Partikel ist $\vec{V} = \vec{v_1} - \vec{v_2}$ und der Anteil der relativen Geschwindigkeit in Normalenrichtung beträgt $\dot{\xi} = \vec{V} \circ \vec{N}$. Dadurch lässt sich dann die tangentiale Geschwindigkeit $\vec{V_t} = \vec{V} - \dot{\xi}\vec{N}$ berechnen.

Die Kraft wird nach [BYM05] mit $\vec{F_n} = f_n \vec{N}$ berechnet, wobei $f_n + k_d \xi^\alpha \dot{\xi} + k_r \xi^\beta = 0$. Dies ist eine recht allgemeine Form einer Gleichung für Normalkräfte. Hierbei ist k_d als Dämpfungsfaktor zu verstehen, womit die Zerstreuung des Partikels bei einer Kollision kontrolliert werden kann, und k_r als Elastizitätskoeffizient, mit dem dessen Steifheit oder Festigkeit gesteuert wird. Im einfachsten Fall lassen sich $\alpha = 0$ und $\beta = 1$ wählen, so dass man $f_n + k_d \dot{\xi} + k_r \xi = 0$ erhält, was den Kräften bei einer einfachen gedämpften Federschwingung entspricht. Die Faktoren k_r und k_d lassen sich nun leicht so einstellen, dass die Simulation zu gegebenen, experimentell ermittelten Daten passt, oder um gewünschte Ergebnisse zu erzeugen.

Die so ermittelte Kraft wird auf das Partikel an der Position $\vec{x_1}$ in Abb. 7 angewandt. Eine betragsmäßig gleiche, aber entgegen gerichtete Kraft wirkt auf das Partikel an der Stelle $\vec{x_2}$.

Um nun eine momentane Situation zu simulieren, muss jedes Partikel mit jedem anderen Partikel auf Kollision getestet werden und falls eine auftritt müssen die resultierenden Kräfte berechnet werden. Durch die aufwändige Kollisionssuche entspricht dies einem $O(n^2)$-Problem.

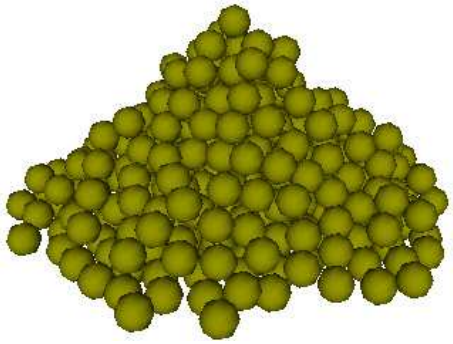


Abbildung 8: Partikelsystem mit Kollision

In dem System, dass zu diesem Zeitpunkt für die Studienarbeit implementiert worden war, fing die Simulation dadurch schon mit relativ wenigen Partikeln an zu ruckeln.

Auch der Versuch, die Anzahl der Kollisionsabfragen mit einem OcTree[2] zu reduzieren, scheiterte. Zwar wurde die Simulation zunächst deutlich flüssiger, allerdings konnte bei weiter wachsender Anzahl der Partikel auch der OcTree nicht mehr helfen, was natürlich direkt wieder mit dem n^2-Aufwand zusammenhängt.

3.2 Diskrete Zellen

Die Simulation der einzelnen Sandkörner ist so zwar weitgehend physikalisch korrekt, aber die Berechnungen, insbesondere die Kollisionsabfragen und -behandlungen, sind trotz Optimierungsversuchen immer noch zu teuer. Da es einer der Ansprüche dieser Arbeit war, das Fließverhalten möglichst in Echtzeit darzustellen, wurde dieser Weg nicht weiter verfolgt, sondern stattdessen ein komplett anderer Ansatz umgesetzt.

Die Idee war es, sich vom Partikelsystem an sich zu lösen und den Partikeln nur noch diskrete Werte im Raum als Positionen zu erlauben, wodurch Nachbarschaftsinformationen sehr leicht errechnet werden können. Nimmt man noch vereinfachend an, dass ein Sandpartikel nicht springen kann, sondern direkt an einem bestehenden Sandhügel herunter rutscht, wenn es auf ihn fällt, lässt sich auch der Aufwand einer Kollisionsbehandlung auf ein Minimum reduzieren.

Die dem Algorithmus für das Fallen und Rutschen der Sandkörner zugrunde liegende Struktur ist in Abb. 9 visualisiert. Der Raum ist hierbei in diskrete Zel-

[2] Raumaufteilende, beschleunigende Datenstruktur

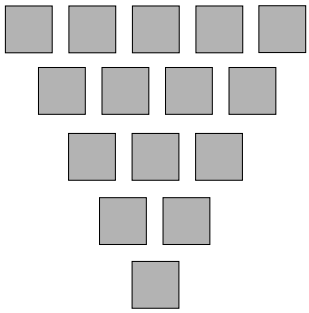


Abbildung 9: Erster Entwurf der Zellen-Struktur

len unterteilt, von denen jede jeweils maximal ein Partikel aufnehmen kann. Im hier abgebildeten 2D-Entwurf hat jede der Zellen zwei Vorgänger (oder 0, in der obersten Reihe) und ein oder zwei Nachfolger (oder 0, in der untersten Reihe).

Fällt eines der Sandkörner, so wird per Zufall ein Vorfahre des nun freien Knotens gewählt, dessen Partikel nachrutscht. Hierdurch wird entsprechend in einer Schicht höher eine Zelle frei, für die der Algorithmus wiederum rekursiv aufgerufen wird.

Die nachrutschenden Partikel springen dabei nicht schlagartig von ihrer aktuellen Lage zu ihrer Zielposition, sondern bewegen sich kontinuierlich in Richtung ihrer Zielzelle. Dies ist genauer im Abschnitt 3.3.1 erklärt.

Ein einfaches Beispiel ist in Abb. 10 dargestellt. Die jeweils aktive Zelle ist hierbei mit *x* markiert.

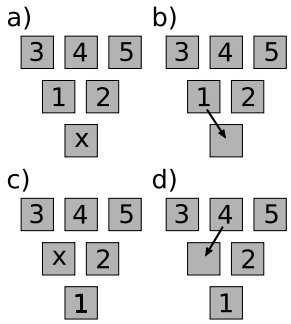


Abbildung 10: Ablauf des Fallens und Nachrutschens im ersten Entwurf

Dieser Entwurf lässt sich auch leicht in den 3-dimensionalen Fall übertragen. Hierfür muss lediglich jeder Zelle die richtige Position zugewiesen werden und sie muss Informationen über ihre entsprechenden Vorgänger und Nachfolger haben.

Überarbeitung des ersten Entwurfs

Ein Nachteil des Entwurfs, der dabei auch in einer 3-dimensionalen Umsetzung bleiben würde, ist die Tatsache, dass die Partikel nicht gerade nach unten fallen

würden, sondern sich eher entlang einer durch die Zellen-Struktur bedingte Zick-Zack-Bahn bewegen würden.

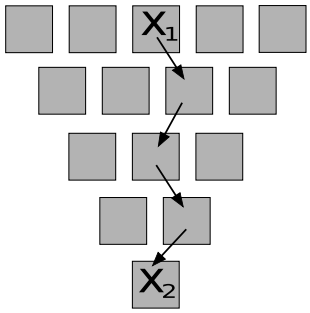


Abbildung 11: Problem des Zick-Zack-Verlaufs beim eigentlich freien Fall

Um dies zu umgehen, wurde der Entwurf an dieser Stelle überdacht und die Anordnung der Zellen überarbeitet. In dieser neuen Version wurden die Knoten der Struktur nicht mehr versetzt sondern direkt übereinander positioniert. In dieser neuen Fassung (Abb. 12) lassen sich auch frei fallende Sandkörner gut modellieren. Eine weitere Strukturierung, wie beispielsweise eine Sanduhröffnung, ist hier nicht mehr nötig, da ihre Form auf eine andere Weise gegeben wird (siehe Abschnitt 3.4).

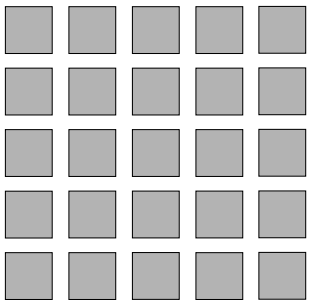


Abbildung 12: Überarbeitete Version der Zellenstruktur

Ein weiterer Vorteil, den diese neue Version bietet, ist die Tatsache, dass auch die Nachbarschaftsinformationen, also Vorgänger und Nachfolger jedes Sandkorns, sehr leicht bestimmt werden können.

Überarbeitung des Algorithmus

Allerdings lässt sich das soweit vorgestellte Modell des Fallens und Nachrutschens nicht auf die gesamte Sanduhr anwenden, sondern lediglich auf die *obere* Hälfte, in der die Sandkörner sich noch nicht neu aufschütten.

Daher wurde auch der Algorithmus noch einmal überarbeitet. Es werden nun nur noch zwei einfache Fälle unterschieden: das *Fallen* und das *Rutschen* der Partikel (Abb. 13). Auf beide Abläufe und auch darauf, wie und wann sie abgearbeitet werden, wird im Abschnitt 3.3 detaillierter eingegangen.

```
fall(cellNr) {
  if isEmpty(directSuccessor)
    content(directSuccessor) = content(cellNr);
    content(cellNr) = EMPTY;
  end if
}

slip(cellNr) {
  if isEmpty(directSuccessor)
    fall(cellNr);
    return;
  end if

  if(anySuccessorIsEmpty)
    target = randomEmptySuccessor(cellNr);
    content(target) = content(cellNr);
    content(cellNr) = EMPTY;
  end if
}
```

Abbildung 13: Ablauf des Fallens und Rutschens von einer Zelle in eine andere

Dieser neue Algorithmus lässt sich nun auch einfach auf den unteren Teil der Sanduhr anwenden. Es muss lediglich noch der Übergang zwischen dem oberen und dem unterem Teil der Sanduhr behandelt werden. Auch dies wird im Abschnitt 3.3.2 genau erklärt.

Jittering[3]

Durch die diskrete Struktur der Zellen, tritt das Problem auf, dass Partikel zu geordnet zum liegen kommen. Wenn man beim Positionswechsel eines Partikels dessen Zielposition gegenüber der Position der Zielzelle leicht verschiebt, wirkt die Struktur nicht so regelmäßig und wirkt optisch ansprechender.

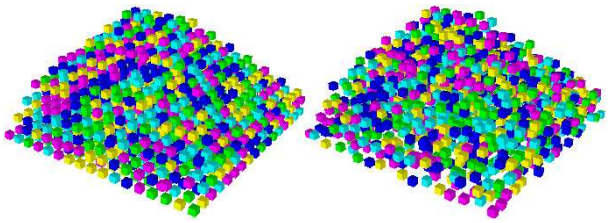


Abbildung 14: links: ohne Jittering, rechts: mit Jittering

[3]Jittering (dt. Flackern, Zittern); Technik zur Verschleierung von eigentlich regelmäßigen Strukturen

3.3 Aufbau des Systems

Das System wird von drei Klassen dominiert, die den Hauptteil der Simulation übernehmen - *ParticleManager*, *ParticleSystem* und *Hourglass*.

Die Klasse ParticleManager regelt dabei die Steuerung der Partikel und wählt ihren jeweiligen Weg durch die Sanduhr. Das ParticleSystem sorgt für kontinuierliche Bewegung der Sandkörner von einer Zelle des ParticleManagers zur nächsten. Die Klasse Hourglass verwaltet hierbei die Objekte der beiden Klassen sowie weitere (Hilfs-)Objekte. Hier werden beispielsweise die Objekte erzeugt, das nächste Fallen der Partikel angestoßen, die Sandkörner auf Wunsch wieder zurückgesetzt oder das Zeichnen der Sanduhr geregelt. Näheres zu den Aufgaben der Klasse ist im Abschnitt 3.3.3 beschrieben.

3.3.1 ParticleManager

Wie bereits erwähnt, sorgt die Klasse *ParticleManager*(Abb. 15) für die Verwaltung der einzelnen Partikel. Sie hält hierzu Informationen über direkte Nachbarn, sowie Nachfolger und Vorgänger bereit.

ParticleManager
- mSizeX, mSizeY, mSizeZ: int - mNum: int - mNumPres, mNumPosts, mNumNears: int - mPres, mPosts, mNears: int* - mPositions: int* - mParticles: int* - mActive: bool*
+ ParticleManager() + create(int shape, int sizeX, int sizeY, int sizeZ, int fillWith): void + fallDown(int actual): void + slipDown(int actual): void + pushInto(int target, int actual): void + transition(ParticleManger *lower): void + insert(int what, int where): bool + refreshParticleSystem(int where): void + form(Shape *shape): void

Abbildung 15: Auszug aus der Klasse ParticleManager

Die Indizes der Vorgängerzellen werden für alle Zellen hintereinander im Array *mPres* gespeichert. Die Anzahl der Vorgänger legt die Variable *mNumPres* fest. Jeder der *mNumPres* Verweise auf die Vorgänger der Zelle ist durch einen Wert ≥ 0 gegeben. Der Wert -1 bedeutet, dass diese Zelle an der entsprechenden Stelle keinen Vorgänger hat, was zum Beispiel am Rand der Sanduhr häufig auftritt. Analoges gilt auch für Nachfolger und direkte Nachbarn und deren jeweilige Variablenbenennung.

Jede Zelle enthält den entsprechenden Index des zugeordneten Partikels des ParticleSystem-Objekts (*mParticles*), also einen Wert ≥ 0. Sonderfälle hierfür werden durch negative Konstanten dargestellt, wobei *PARTICLE_EMPTY = -1* dafür steht, dass die Zelle leer ist, also kein Partikel enthält, und *PARTICLE_INVALID = -2* bedeutet, dass diese Zelle nicht mit einem Partikel gefüllt werden darf. Letzteres wird in Abschnitt 3.4 noch einmal zusammen mit Beispielen für solche ungültigen Zellen genauer erklärt.

Die Klasse bietet nun verschiedene Methoden zur Steuerung der Simulation an. So wird mit dem Methoden *fallDown* und *slipDown* das Fallen bzw. Rutschen des in der angegebenen Zelle enthaltenen Partikels gestartet. Die Funktion *pushInto* verschiebt den Inhalt von einer Zelle in die nächste. Abbildung 16 stellt die Algorithmen noch einmal dar und zeigt auch die zu beachtenden Randbedingungen, die in Abbildung 13 der Einfachheit halber vernachlässigt wurden.

Erzeugt wird der ParticleManager indem im Konstruktor die Methode *create* aufgerufen wird. Sie erwartet als Parameter zunächst die Angabe, ob es sich um den oberen oder den unteren Manager[4] handelt, der hier konstruiert werden soll. Dies ist für die Anordnung der Zellen wichtig, da die erste Reihe stets die y-Koordinate 0 haben soll. Dadurch überlappen sich die beiden Manager genau in einer Reihe, was sich für den Übergang zwischen ihnen als nützlich erweisen wird.

Die nächsten drei Parameter stellen die Größenangabe dar, also wie viele Zellen in x-, y- und z-Richtung jeweils angelegt werden sollen. Der letzte Parameter gibt an, mit welchem Wert die Zellen beim Anlegen gefüllt werden sollen. Erlaubt sind hier wieder die Konstanten *PARTICLE_EMPTY* und *PARTICLE_INVALID*.

Weitere Bemerkungen zu den Größenangaben sowie Erläuterungen, warum es sinnvoll sein kann, die gesamten Zellen des Managers bei der Initialisierung als ungültig (*PARTICLE_INVALID*) zu deklarieren, befinden sich ebenfalls in Abschnitt 3.4.

In der Methode *create* wird zunächst der Speicher der Arrays für die Positionen der Zellen, die Indizes der jeweiligen Partikel und für die Indizes der jeweiligen Vorgänger-, Nachfolger und Nachbarnzellen alloziiert. Daraufhin werden die Zellen verbunden, indem die Indizes der benachbarten Boxen berechnet werden. Durch die simple quaderförmige Anordnung ist dies auch recht einfach zu bewerkstelligen.

Die Vorgängerzellen p_0 bis p_8 einer Zelle n in Abb. 17, die direkt unter p_0 liegt, werden beispielsweise bestimmt, indem man auf den Index von n bestimmte Werte addiert. So kann p_0 errechnet werden, indem man n um die Größe einer Schicht, und damit das Produkt von Breite und Tiefe des ParticleManagers, erhöht, also $p_0 = n + sizeX \cdot sizeZ$. Analog lassen sich die anderen Werte der Vorgänger berechnen. Nach demselben Prinzip werden auch die Indizes aller Nachfolger und aller direkter Nachbarn bestimmt.

Damit ist es bereits möglich, den ParticleManager für die Verwaltung der Sandkörner zu erstellen und den Sand sowohl im oberen als auch unteren Teil der Sanduhr fließen zu lassen. Es muss jedoch noch der Übergang von der einen Hälfte in die andere geregelt werden. Hierfür steht die Methode *transition* zur Verfügung,

[4]Die Erklärung, warum zwei Manager benutzt werden, ist in Abschnitt 3.4 nachzulesen.

```
fall(cell) {
  if(isEmpty(cell) or isInvalid(cell)) return;
  if(directSuccessor = -1) return;
  if(isFull(directSuccessor)) return;
  if(isInvalid(directSuccessor)) return;

  pushInto(directSuccessor, cell);
}

slip(cell) {
  if(isEmpty(cell) or isInvalid(cell)) return;
  if(directSuccessor != -1 and isEmpty(directSuccessor))
    fall(cellNr);
    return;
  endif

  anySuccessorIsEmpty = false;
  for every successor s
    if(s != -1 and isEmpty(s)) anySuccessorIsEmpty = true;
  end for

  if(anySuccessorIsEmpty)
    target = randomEmptySuccessor(cellNr);
    pushInto(target, cellNr);
  end if
}

pushInto(target, actual) {
  content(target) = content(actual);
  velocity(target) = position(target) - position(actual);
  content(actual) = EMPTY;
}
```

Abbildung 16: Ablauf des Fallens und Rutschens, mit Randbedingungen

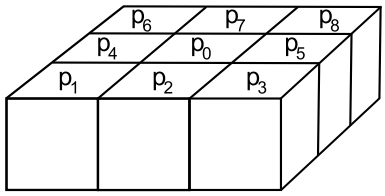
die als Parameter einen Zeiger auf das ParticleManager-Objekt erwartet, dem die Partikel übergeben werden sollen.

Hierbei wird, wie in Abb. 18 zu sehen, intern auf die Methode *insert* zugegriffen, die im unteren Teil der Sanduhr die Randbedingungen prüft und die entsprechende Zelle mit dem Partikelindex belegt.

3.3.2 ParticleSystem

Die Klasse *ParticleSystem*(Abb. 19) speichert die tatsächlichen Positionen der einzelnen Partikel und sorgt so auch dafür, dass sich die Sandkörner kontinuierlich und nicht sprungartig zwischen den einzelnen Zellen bewegen.

Wie in der Methode *pushInto* der Klasse ParticleManager (Abb. 16) schon angedeutet, wird dies erreicht, indem die Differenz der Positionen von Zielzelle und aktueller Zelle als eine Art Geschwindigkeit eines Partikels verwendet wird. In




```
p0 = n + sizeX·sizeZ
p1 = n + sizeX·sizeZ - sizeX - 1
p2 = n + sizeX·sizeZ - sizeX
p3 = n + sizeX·sizeZ - sizeX + 1
p4 = n + sizeX·sizeZ - 1
p5 = n + sizeX·sizeZ + 1
p6 = n + sizeX·sizeZ + sizeX - 1
p7 = n + sizeX·sizeZ + sizeX
p8 = n + sizeX·sizeZ + sizeX + 1
```

Abbildung 17: Bestimmung der Vorgängerzellen; Zelle n sei dabei direkt unter der Zelle p_0

der Methode *updatePos* wird die Position des Partikels entsprechend dieser Geschwindigkeit angepasst. Es wird dabei auch ein Parameter, im Bereich $0 \leq \Delta t \leq 1$ übergeben, mit dem gesteuert wird, wie weit sich das Partikel entsprechend dieser „Geschwindigkeit"bewegen soll, also $\vec{p_{new}} = \vec{p_{old}} + \Delta t \cdot \vec{v}$. Wie der Wert Δt bestimmt wird und wann diese Aktualisierung der Postion stattfindet, wird im Abschnitt 3.3.3 beschrieben.

Mit der Methode *zeroVelocity* lässt sich die Geschwindigkeit aller Partikel wieder auf 0 setzen, was dafür sorgt, dass die Sandkörner nicht einfach durch ihre Zielzelle durchfallen, sondern auch anhalten, wenn sie sie erreicht haben. Erst bei einem erneutes Aufrufen der Methoden *fallDown* und *slipDown* werden sie sich wieder weiter bewegen.

3.3.3 Hourglass

Die Klasse *Hourglass*(Abb. 20) verwaltet die Objekte der Klassen *ParticleManager* und *ParticleSystem*. Sie regelt auch den zeitlichen Ablauf der Ereignisse. Im Hauptprogramm wird eine Instanz dieser Klasse angelegt, über die der gesamte Zugriff auf die Simulation erfolgt.

Hier wird auch die Methode für das Zeichnen der Sandkörner bereitgestellt. Auch das Zurücksetzen der Sandkörner wird in dieser Klasse ermöglicht.

Der zeitliche Ablauf der Ereignisse wird geregelt, indem diskrete Zeitschritte mitgezählt werden. Hierzu wird bei jedem Aufruf der *update*-Methode der Wert *mAllSteps* erhöht. Bei jedem *mEventSteps*-ten Schritt werden die Sandkörner zu einer neuen Bewegung angeregt. Dazwischen bewegen sie sich kontinuierlich mit ihrer in Abschnitt 3.3.2 beschriebenen „Geschwindigkeit"weiter. Mit dem Wert *mEventSteps* wird also quasi die Geschwindigkeit des gesamten Sandflusses geregelt.

```
transition(ParticleManager lower) {
  for every cell c in the lowest row
    p = content(c);
    if(p >= 0 and lower.insert(p, c))
      content(c) = EMPTY;
    end if
  end for
}

insert(what, where) {
  if(where != 1 and content(where) = EMPTY)
    content(where) = what;
    return true;
  end if
  return false;
}
```

Abbildung 18: Übergang zwischen den ParticleManager-Objekten

Mit der *update*-Methode (Abb. 21) wird das nächste Fallen bzw. Rutschen der Partikel angestoßen. Dazu wird zunächst bei jedem Partikel die Geschwindigkeit auf 0 gesetzt. Hiermit wird erreicht, dass Sandkörner, die sich nun nicht mehr weiter bewegen würden, weil sie z.B. auf dem Boden angekommen sind, auch wirklich liegen bleiben. Dann wird für jede Zelle der beiden Manager die in Abschnitt 3.3.1 vorgestellte *fallDown*- und *slipDown*-Methode aufgerufen, die, falls hierbei bestimmt wird, dass eine Bewegung gestartet werden soll, der Zelle bzw. dem ihr aktuell zugeordneten Partikel eine neue „Geschwindigkeit" zuweist. Danach wird der Übergang zwischen den beiden Managern ausgeführt. Da sie sich, wie ebenfalls weiter oben bereits beschrieben, genau in einer Reihe überlappen, können die Positionen der betroffenen Sandkörner einfach beibehalten werden und lediglich die entsprechenden Indizes werden aus den Zellen des oberen Managers gestrichen und dafür in die des unteren eingetragen.

Danach wird, unabhängig davon ob eine neue Partikelbewegung ausgelöst wurde, die Position der einzelnen Sandkörner aktualisiert, indem, wie in Abschnitt 3.3.2 erklärt, ihre „Geschwindigkeit" auf ihre momentane Lage aufaddiert wird. Der Faktor Δt, der hierbei verwendet wird, wird mit $\Delta t = \frac{1}{eventSteps}$ berechnet. Dies hat zur Folge, dass nach exakt *eventSteps* Zeitschritten, das Partikel in der gewünschten Zielzelle angekommen ist[5].

Die Klasse enthält auch eine Membervariable *mSize* die Einfluss auf die tatsächlich dargestellte Größe der Sanduhr hat. Sämtliche Größenangaben, wie sie im ParticleManager vorhanden sind, werden für die Darstellung erst mit diesem Faktor multipliziert. Der Wert *mSize* gibt damit also die Größe einer Zelle an. Hier-

[5]Begründung hierzu: $\vec{v} = \vec{p}_{target} - \vec{p}_{old}$ und $\vec{p}_{new} = \vec{p}_{old} + \Delta t \cdot \vec{v}$. Dies wiederholt angewendet ergibt $\vec{p}_{new} = \vec{p}_{old} + \Delta t \cdot \vec{v} + ... + \Delta t \cdot \vec{v}$, wobei $\Delta t = \frac{1}{eventSteps}$ und der additive Faktor ($\Delta t \cdot \vec{v}$) *eventSteps* mal auftacht. Damit gilt also $\vec{p}_{new} = \vec{p}_{old} + eventSteps \cdot (\frac{1}{eventSteps} \cdot \vec{v}) = \vec{p}_{old} + \frac{eventSteps}{eventSteps} \cdot (\vec{p}_{target} - \vec{p}_{old}) = \vec{p}_{old} + \vec{p}_{target} - \vec{p}_{old} = \vec{p}_{target}$. Also ist das Partikel tatsächlich nach genau *eventSteps* Schritten an der gewünschten Zielposition angelangt.

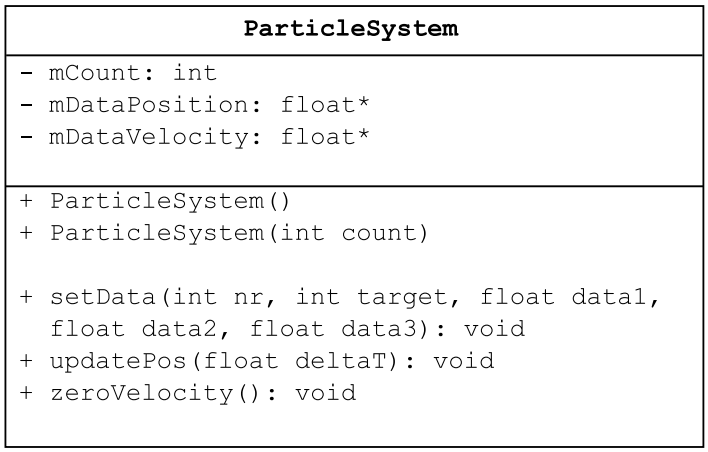


Abbildung 19: Auszug aus der Klasse ParticleSystem

mit lässt sich beispielsweise leicht einstellen, dass eine Sanduhr, deren ParticleManager viele Zellen enthält, genauso groß, wenn auch besser aufgelöst, dargestellt wird wie eine, die nicht so viele Zellen verwaltet. Durch die Kombination dieses Größenfaktors mit der Einstellung der Zellenanzahl lässt sich die Simulation auf verschiedenen Systemen an die gegebenen Frameraten anpassen.

Der Wert *mMode* gibt an, ob die Sanduhr lediglich eine leere Leinwand darstellt, auf der mit Sand gemalt werden kann (siehe hierzu auch Abschnitt 3.5), oder ob sie nach einer Form gestaltet wurde (näheres hierzu im Abschnitt 3.4). Die Variable entspricht also einer der beiden Konstanten *MODE_PAINT* oder *MODE_SHAPE*. Im letzteren Fall ist noch zu unterscheiden, ob wirklich eine feste Hülle vorhanden ist, die der Sand lediglich nach unten verlassen kann, was also wirklich einer Sanduhr entsprechen würde, oder ob die Form nur die initiale Positionen der Partikel bestimmt, woraufhin sie sich danach aber frei bewegen können, was eher einer Sandskulptur entspricht, die plötzlich zerfließt. Dies wird in der Variablen *mShapeMode* mit Hilfe der Konstanten *SHAPE_FIX* bzw. *SHAPE_LOSE* festgehalten.

Diese beiden Variablen, insbesondere *mMode*, helfen dabei zu entscheiden, was beim Aufruf der Methode *reset* geschehen soll. Zunächst wird in jedem Fall, die Geschwindigkeit aller Partikel auf 0 zurückgesetzt. Sollte es sich bei der Sanduhr in der aktuellen Simulation um eine Leinwand handeln, werden einfach beide ParticleManager wieder geleert[6]. Sollten die Positionen der Sandkörner durch eine Form gegeben sein, unabhängig davon, ob die Hülle fest ist oder lediglich initial formgebend war, wird der untere Manager geleert und sämtliche Zellen in der oberen Sanduhrhälfte, die innerhalb der Form liegen, werden mit Sand gefüllt[7].

[6]Hierzu wird jeder Partikelindex des Managers auf den Wert *PARTICLE_EMPTY* gesetzt.

[7]Näheres hierzu im Abschnitt 3.4.

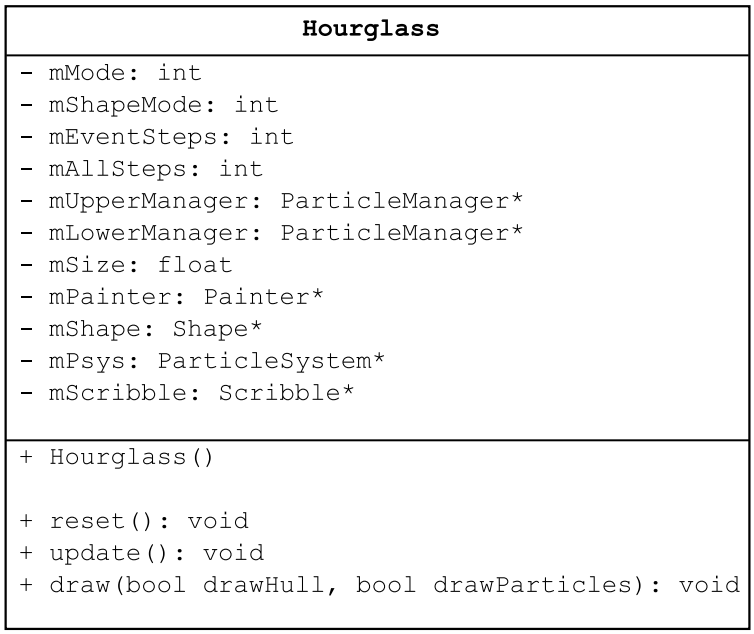


Abbildung 20: Auszug aus der Klasse Hourglass

3.4 Formen

Eine wichtige Anwendung der Algorithmen ist natürlich eine Sanduhr. Bevor sie hieran allerdings gezeigt werden können, ist zunächst sowohl das Erstellen wie auch das Füllen einer gewissen Form notwendig.

Für die vorliegende Arbeit ist ein Verfahren entwickelt worden, diese Formen aus einem eingelesenen 3D-Modell zu erschaffen[8]. So ist es möglich, sich ein beliebiges Modell in einem 3D-Programm zu erstellen und dieses dann in der hier implementierten Anwendung als Sandskulptur zerfallen zu lassen oder es als Sanduhr zu verwenden. Hierbei ist natürlich anzumerken, dass sich einfache Modelle besser eignen als solche, die hochdetailliert modelliert wurden, da die Positionen der Zellen im Partikelmanager ja nur diskrete Werte annehmen.

Die Funktionalität des Einlesens wird von der Klasse *Shape* (Abb. 22) bereitgestellt. Beim Aufruf des Konstruktors werden die Dreiecksdaten aus der angegebenen Datei eingelesen und als Zeiger auf Objekte vom Typ *Triangle* in der Datenstruktur *std::vector* abgelegt. Dem Konstruktor werden auch noch drei Parameter (*sizeX*, *sizeY* und *sizeZ*) als Größenangaben übergeben, die als maximale Ausdehnung in die jeweilige Richtung interpretiert werden. Hierfür werden noch während des Einlesens der Datei die minimalen und maximalen Werte der Eckpunktkoordinaten jeder der drei Koordinatenachsen gespeichert und stets aktualisiert. Aus den Differenzen von Maxima und Minima kann so die tatsächliche Ausdehnung der eingelesen Form in jede Richtung errechnet werden. Als Skalierungsfaktoren s_x, s_y und s_z werden nun die Quotienten aus gewünschter und

[8]Hierfür wurde ein Konverter geschrieben, der aus *Wavefront*-Dateien (.obj) die Dreiecksdaten extrahiert. Siehe hierzu auch Abschnitt 6.1

```
update() {
  allSteps = allSteps + 1;

  if(allSteps % eventSteps = 0)
    particleSystem.zeroVelocity();

    for(every cell c in the upperManager)
      c.fallDown();
      c.slipDown();
    end for

    for(every cell c in the lowerManager)
      c.fallDown();
      c.slipDown();
    end for

    upperManager.transition(lowerManager);
  end if

  particleSystem.updatePos(1 / eventSteps);
}
```

Abbildung 21: Update, regelt den Sandfluss

tatsächlicher maximalen Breite, Höhe und Tiefe berechnet, also beispielsweise $s_x = \frac{sizeX}{max_X - min_X}$. Die Eckpunktkoordinaten aller eingelesen Dreiecke werden nun nochmal mit diesen Faktoren multipliziert. Anschließend werden die y-Werte aller Eckpunkte um den nach der Skalierung erneut berechneten Wert $|min_y|$ erhöht, so dass kein Dreieck in den negativen y-Bereich ragt. Dadurch wird die Form komplett in den oberen ParticleManager passen.

Die Größenangaben der ParticleManager richten sich dabei auch tatsächlich nach den hier übergegebenen Werten. Damit sich der Sand frei bewegen und aufschütten kann, werden die Manager dabei größer als die Form angelegt. Insbesondere der untere Manager sollte dabei zusätzlich auch noch höher sein, als die Form, damit der Sand auch ausreichend Platz zum Fallen hat, was ansprechender aussieht.[9]

Zellen innerhalb der Form

Nun liegen alle Dreiecksdaten vor und es gilt nun, als nächstes die Punkte innerhalb der Form zu finden. Diese Punkte sind analog zu den Zellen in den ParticleManagern angeordnet und entsprechend durchnummeriert. Es wird mit der Variablen *mInPoints* Speicher für $(max_x - min_x + 1) \cdot (max_y - min_y + 1) \cdot (max_z - min_z + 1)$ *bool*-Werte alloziiert, in denen für jeden Punkt festgehalten wird, ob er innerhalb der Form liegt oder nicht. Um dies zu bestimmen, wird nun für jeden y-Wert im Bereich $min_y \leq y \leq max_y$ und jeden z-Wert im Bereich $min_z \leq z \leq$

[9] In der vorliegenden Implementation sind beide Manager doppelt so breit und tief wie die Form, und der untere auch noch doppelt so hoch.

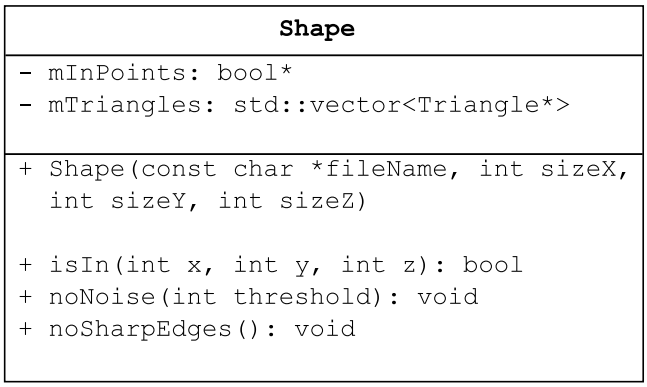


Abbildung 22: Auszug aus der Klasse Shape

max_z ein Strahl in positiver x-Richtung verschossen, der mit jedem Dreieck auf Schnittpunkte untersucht wird. Hierbei werden all diese gefundenen Punkte als Eintritts- bzw. Austrittspunkte der Form deklariert, je nachdem ob der Startpunkt des Strahles, also (min_x, y, z), vor oder hinter dem jeweils untersuchten Dreieck liegt.[10] Alle gefundenen Eintritts- und Austrittspunkte werden getrennt nach ihrer x-Koordinaten sortiert. Dadurch lassen sich Paare von zugehörigen Eintritts- und Austrittspunkten bestimmen. Für jeden x-Wert im Bereich $min_x \leq x \leq max_x$ wird nun getestet, ob er zwischen einem solchen Paar von Schnittpunkten liegt. Ist dies der Fall kann der entsprechende *mInPoints*-Wert auf *true* gesetzt werden, ansonsten auf *false*.

Durch eine einfache Koordinatenumrechnung unter Berücksichtigung der min- und max-Werte der Form kann nun für jeden Punkt im Raum bestimmt werden, ob er innerhalb oder außerhalb der Form liegt. Dies ist genau die Funktionalität, die die Methode *isIn* anbietet.

Entfernen von unbeabsichtigt gefüllten Zellen

Durch die Diskretisierung der Werte kann es vorkommen, dass Punkte als innerhalb der Form deklariert werden, obwohl sie es eigentlich nicht sind. Hierfür wird in der Klasse die Methode *noNoise* bereitgestellt, die derartige Fehler beseitigt. Hierzu wird jeder der Punkte in *mInPoints* mit dem Wert *true* sowie seine direkten Nachbarn betrachtet. Sollten lediglich höchstens so viele der direkten Nachbarn ebenfalls als innerhalb angenommen sein wie der der Methode übergebene Schwellwertsparameter, so wird auch der Wert für den untersuchten Punkt selbst auf *false* gesetzt.

Sollte sich beim Durchlaufen aller Punkte einer der Werte ändern, so wird im Anschluss immer wieder ein neuer Durchlauf gestartet, bis sich nichts mehr an den Werten ändert.

[10] Dies kann leicht getestet werden, indem man einen Verbindungsvektor $\vec{v}$ von einem beliebigen Eckpunkt des Dreiecks zum zu untersuchenden Punkt legt, und das Skalarprodukt von diesem Vektor mit der Dreiecksnormalen $\vec{n}$ bildet. Ist $\vec{v} \circ \vec{n} < 0$, dann zeigen $\vec{v}$ und $\vec{n}$ in verschiedene Halbräume und der Punkt liegt entsprechend hinter dem Dreieck.

Entfernung scharfer Kanten

Ebenfalls durch derartige Diskretisierungs-Ungenauigkeiten oder unvorsichtiges Modellieren kann es auch passieren, dass in der Form scharfe Kanten entstehen. In Abb. 23 kann man dies gut sehen. Hier würden die Sandkörner in den mit *x* markierten Zellen niemals weiterrutschen, da sie entweder gar keinen Nachfolger haben oder wiederum nur solche, die niemals leer werden können.

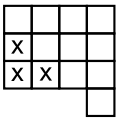


Abbildung 23: Beispiel für scharfe Kanten

Dadurch kommt es, dass am Ende noch Sandkörner im oberen Teil der Sanduhr liegen bleiben können, was weniger ansprechend wirkt, als wenn sich die gesamte obere Hälfte leeren würde. Dem kann mit der Methode *noSharpEdges* abgeholfen werden. Hier wird wieder jeder der Punkte in *mInPoints* mit dem Wert *true* betrachtet, wobei von unten nach oben durchgegangen wird. Hat der Punkt mindestens einen Nachfolger, der ebenfalls den Wert *true* hat, so ändert sich nichts. Hat er jedoch keinen solchen Nachfolger, so wird der Punkt als Teil einer scharfen Kante angesehen und sein Wert wird auf *false* gesetzt.

Auch bei dieser Methode wird nach dem Durchlauf ein neuer gestartet, falls sich mindestens einer der Werte geändert hat. Erst, wenn sich bei einem Aufruf nichts mehr ändert, endet die Methode. Angewandt auf das Beispiel aus Abb. 23 hat das zur Folge, dass eben die mit *x* markierten Punkte als nicht zur Form gehörig deklariert werden.

Sanduhr/Sandskulptur

Wie in Abschnitt 3.3.1 angerissen, gibt es prinzipiell drei Kategorien von Werten für das Partikel in jeder der Zellen des Managers. Ein Wert ≥ 0 gibt den Index des Partikels an, der Wert -1, also die Kontante *PARTICLE_EMPTY* bedeutet, dass die Zelle leer ist, und der Wert -2, der der Konstanten *PARTICLE_INVALID* entspricht, steht dafür, dass die Zelle nicht mit Partikeln gefüllt werden darf. Hiermit kann leicht zwischen einer Sanduhr und einer Sandskulptur unterschieden werden.

Soll die geladene Form als Skulptur interpretiert werden, so wird jede Zelle initial mit dem Wert *PARTICLE_EMPTY* belegt, da es für den Sand und seine Bewegung keine Einschränkungen gibt. Nach dem Initialisieren werden alle Punkte, die sich innerhalb der Form befinden, mit einem Partikelindex ≥ 0 gefüllt.

Falls mit der Form jedoch eine Sanduhr dargestellt werden soll, wird jede Zelle initial mit dem Wert *PARTICLE_INVALID* gefüllt. Dadurch wird gewährleistet, dass kein Sandkorn jemals an eine Stelle außerhalb der Form gelangen kann. Auch hier werden nach dem Initialisieren alle Punkte innerhalb der Form mit einem nichtnegativen Wert versehen.

Beim Zurücksetzen der Sanduhr, also beim Aufruf der *reset*-Methode aus der Hourglass-Klasse, wird erneut jede Zelle in der Form mit einem Indexwert belegt. Da die Ungültigkeitswerte im Modus *SHAPE_FIX* (siehe Abschnitt 3.3.3) erhalten bleiben, muss hier auch nichts geändert werden. Im Modus *SHAPE_LOSE* muss der obere ParticleManager vor dem erneuten Füllen erst komplett geleert, also alle Zellen auf *PARTICLE_EMPTY* gesetzt werden.

Zu beachten ist, dass diese Überlegungen lediglich für den oberen ParticleManager gelten. Der untere Teil sollte stets mit $PARTICLE\_EMPTY$ initialisiert werden. Diese Unterscheidung ist auch der Grund, warum zwei getrennte Manager benutzt werden.

Hindernisse

Mit dieser Technik, Zellen, die nie gefüllt werden dürfen, als *invalid* zu markieren, können auch leicht Hindernisse realisiert werden, auf die der Sand zwar drauf geschüttet werden aber nicht in sie eindringen kann.

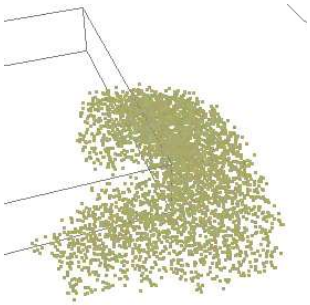


Abbildung 24: Hindernis; Sand liegt darauf und daneben, aber nicht darin

Als Hindernisse wurden einfache Quader gewählt, die sich durch minimale und maximale Koordinaten in jede der drei Richtungen leicht beschreiben lassen. Hierzu wurde eine neue Klasse *Obstacle* hinzgefügt, die lediglich diese Koordinaten verwaltet und eine Funktion zum Zeichnen des Hindernisses anbietet.

Der Klasse ParticleManager wurde eine neue Membervariable vom Typ *std::vector* für die Verwaltung der Obstacle-Objekte, sowie eine Methode *addObstacle* zu deren Erzeugung hinzugefügt. Die anzugebenden Koordinaten beziehen sich dabei auf das ParticleManager-Koordinatensystem und liegen dementsprechend im Bereich von 0 bis zur entsprechenden Größenangabe (z.B. $0 \leq x \leq sizeX$).

3.5 Malen

Als eine weitere Anwendung, in der der rieselnde Sand gut zur Geltung kommt, wurde die Möglichkeit implementiert, mit Sand zu „malen“. Hierfür soll es möglich sein, mit der Maus auf einer Art Leinwand Linien und Formen zu zeichnen, auf die dann Sand fällt. Natürlich soll sich der Sand auch hier wieder aufhäufen, falls der Mauszeiger mehrfach über die selbe Stelle bewegt wird.

Eine Herausforderung stellt hierbei das Umrechnen der 2D-Bildschirmkoordinaten des Mauszeigers in das 3D-Koordinatensystem, in dem der Sand später liegen soll. Hierfür wurde ein effizientes Verfahren implementiert. Des Weiteren

soll es möglich sein, ein Bild als Vorlage zu nehmen, auf dem gezeichnet wird. Hierfür muss auch die Liniendicke einstellbar sein, sowie die Option bestehen, bereits gesetzte Linien wieder zu entfernen.

Picking

Es gibt verschiedene Ansätze, um zu bestimmen, welche 3D-Koordinate mit der Maus getroffen wurde. Eine Möglichkeit wäre es von den Mauskoordinaten aus einen Strahl in den Raum zu schicken, der mit jedem Objekt und jeder Fläche auf Schnittpunkte getestet wird.

Eine weitere gängige Option ist es, bei einem Mausklick die gesamte Szene ohne Beleuchtung und Antialiasing neu zu zeichnen, wobei für jedes Objekt eine eindeutige Farbe verwendet wird. Das so gerenderte Bild wird aber nicht auf dem Bildschirm dargestellt sondern hieraus wird an der Stelle der Mauskoordinaten die Farbe ausgelesen und daraus Rückschlüsse auf das angeklickte Objekt gezogen. Ein solcher Ansatz wurde auch in der vorliegenden Anwendung verwendet.

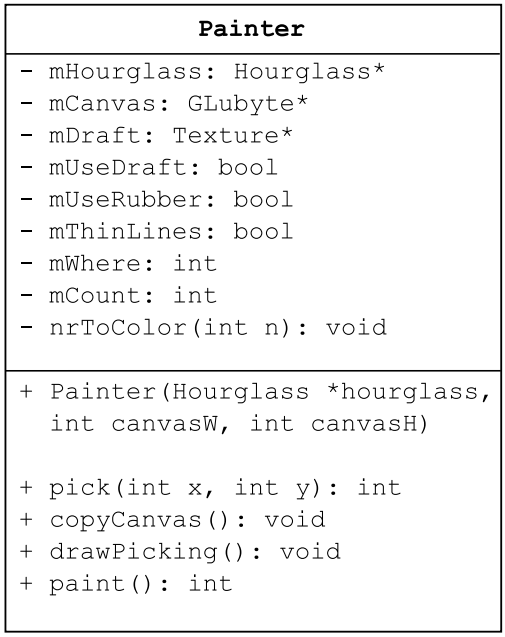


Abbildung 25: Auszüge aus der Klasse Painter

Die Klasse *Painter* (Abb. 25) fasst alle nötigen Methoden und Variablen zusammen. Bereits im Konstruktor wird ihr ein Zeiger auf das zugehörige Hourglass-Objekt übergeben, von dem sie die nötigen Informationen über die ParticleManager-Objekte erhält.

Der Wert *mWhere* entspricht, einer der beiden Konstanten *PAINT_TOP* oder *PAINT_BOTTOM*. Hiermit wird festgelegt, wie die mit der Maus anvisierte Stelle interpretiert werden soll - im ersteren Fall wird damit bestimmt, von wo aus der Sand rieseln soll, im letzteren zeigt die Maus dorthin, wo der Sand hin rieseln soll. Je nach Blickwinkel kann eine dieser Einstellungen besser sein als die andere.

Der Zeiger auf die Textur *mDraft* weist auf eine Textur hin, die als Vorlage genutzt werden kann. Hierauf wird später noch genauer eingegangen.

Gezeichnet wird nun immer dann, wenn die Maus bei gedrückter Maustaste bewegt wird. Hierzu werden Zellen des ParticleManagers wie oben beschrieben farbig codiert im Hintergrund neu gezeichnet - bei der Einstellung *PAINT_TOP* wird hierzu die oberste Zeile des oberen Managers verwendet, bei *PAINT_BOTTOM* hingegen, werden die Zellen der untersten Zeile des unteren Managers gezeichnet. Auch im zweiten Fall beginnt der Sand jedoch in der entsprechenden Zelle der obersten Zeile des oberen Managers zu rieseln. Die Codierung übernimmt hierbei die Methode *nrToColor* indem sie die übergebene Nummer der jeweiligen Zelle in die RGB-Farbe $(nr\ div 255, nr\ mod 255, 127)$[11] Hiermit sind also etwa 256^2 verschiedene Zellen codierbar, was einer xz-Auflösung der ParticleManager von 256×256 entsprechen würde.

Da es sehr ineffizient wäre, bei jeder Mausbewegung die Codierungsinformationen neu zu zeichnen, wurde diese Idee ein klein wenig abgewandelt. Während des Malvorgangs wird sich der Blickwinkel nicht ändern, deswegen ist es möglich, die farblich codierten Zellen beim Betätigen der Maustaste zu zeichnen, und die Farbwerte aller Pixel des Fensters in einem Array zu speichern (*mCanvas*), auf das nun bei einer Bewegung zugegriffen wird. Zum Kopieren der Farbwerte steht hierzu die Methode *copyCanvas* und für das Auslesen einer gegebenen Position die Funktion *pick* zur Verfügung.

Um zu entscheiden, welcher Partikelindex in die selektierte Zelle eingetragen werden soll, wird jeder Picking-Vorgang mit der Methode *paint* mitgezählt. Sie gibt auch den aktuell gezählten Wert zurück, mit dem die Zelle nun gefüllt werden kann.

Vorlage

Wie bereits erwähnt ist es möglich, ein Bild als Vorlage einzulesen. Wenn kein solches Bild geladen wird, wird eine Außenlinie um die im Hintergrund gezeichneten Zellen des ParticleManagers gezogen, um die Fläche für die Mausbewegung anzuzeigen, und so die Navigation deutlich zu erleichtern. Sollte eine Vorlage eingelesen sein, wird statt dieser Außenlinie das entsprechende Bild so angezeigt, dass es genau dieselbe Fläche einnimmt, wozu es falls nötig gestaucht oder gestreckt wird. Bei der Einstellung *PAINT_TOP* wird die Vorlage halbtransparent gezeichnet, um die Sicht auf den liegenden Sand möglichst nicht zu behindern.

Liniendicke

Es kann vorkommen, dass man sich für das Zeichnen unterschiedliche Linienstärken wünscht. Wenn man nämlich mit der Maus Linien zeichnet, werden fast immer, insbesondere bei langsamer Bewegung des Mauszeigers, mehrere Positionsänderung über ein und derselben Zelle registriert. So kommt es, dass mehre-

[11] Die Division im rot-Kanal erfolgt hierbei ganzzahlig. Der Wert 127 im blau-Kanal stellt lediglich eine zusätzliche Sicherheitskontrolle dar, damit die ausgelesene Farbe eindeutig wieder in die Nummer der Zelle zurückgerechnet werden kann, ohne dass Fehler durch Antialiasing aufgetreten sind. Prinzipiell ist der Wert hier an dieser Stelle beliebig setzbar, sollte sich lediglich von der Hintergrundfarbe des Fensters unterscheiden.

re Partikel in die selben Zelle geschüttet werden, die sich beim Ankommen am Boden sofort aufhäufen und dabei zur Seite wegrutschen, wodurch automatisch eher dickere Linien entstehen. Dünne Linien sind also nur mit schnellen Mausbewegungen zu zeichnen, wobei es dann aber schwierig wird, präzise zu zielen.

Abbildung 26: Unterschiedliche Liniendicke bei gleicher Geschwindigkeit der Mausbewegung

Deshalb wurde die Möglichkeit implementiert, dass auch dünnere Linien mit normalen Bewegungen mit der Maus erzielt werden können (gesteuert mit der Variablen *mThinLines*). Sollte diese Option aktiviert sein, wird beim Picking die aktuell gewählte Zelle gespeichert, und nur wenn sich eine selektierte Zelle von dieser letzten gespeicherten Position unterscheidet, wird ein Sandkorn an die entsprechende Stelle zum Fall freigegeben und die Information zur letztgewählten Zelle aktualisiert.

Radieren

Um beim Zeichnen noch flexibler vorgehen zu können, wurde noch die Möglichkeit des „Radierens“hinzugefügt. Dies mag zwar eine ungewöhnliche Metapher sein, die nicht wirklich zu Sand zu passen scheint, bietet sich dafür aber im Sinne des „Zeichnens “hervorragend an. Dabei wird zunächst das Picking genau wie beim Zeichnen durchgeführt, doch statt Partikel hinzuzufügen, wird über die Vorgängerrelation die unterste Zelle über der gewählten im unteren ParticleManager gewählt, die Sand enthält, und deren Partikelindex auf den Wert *PARTICLE_EMPTY* gesetzt. Fährt man also hier mit der Maus über einen Sandhügel kann man so seine oberste Schicht abtragen. Dabei ist darauf zu achten, dass die Zelle nur dann überschrieben wird, wenn sie mit einem Wert ≥ 0 gefüllt ist, da ansonsten versehentlich kleine Hindernisse mit wegradiert werden könnten (siehe Abschnitt 3.4, S. 25).

3.6 GUI

Um dem Benutzer unzählige Tastaturkürzel zu ersparen und dadurch die Bedienung des Programms erheblich zu erleichtern, wurde eine graphische Benutzeroberfläche mit implementiert. Auch hierbei wurde kein bestehendes Modell wie beispielsweise *Qt*[12] oder *wxWidgets*[13] verwendet, sondern ein eigenes Konzept entworfen.

Für das vorliegende Programm werden als Bedienelemente lediglich einfache Buttons benötigt, um Funktionen an- bzw. auszuschalten oder zwischen bestimmten Zuständen hin und her zu schalten. Deswegen genügt es hier, die Schaltflächen

[12] www.trolltech.com/products/qt
[13] www.wxwidgets.org

lediglich als einfache mit Texturen belegte Rechtecke auf den Bildschirm zu zeichnen.

Die Umsetzung der Oberfläche befindet sich hauptsächlich in den Klassen *Button* und *Gui*.

Button

Button
- mLeft, mTop, mWidth, mHeight: float - mTexture: Texture* - mCallback: int - mFlag: int
+ Button(float l, float t, float w, float h, char *texFile, int callback, int flag) + draw(): void + click(float x, float y): int + release(float x, float y): bool + isFlagged(int flag): bool

Abbildung 27: Auszug aus der Klasse Button

Für die Darstellung der Gui wird die Projektionsmatrix in OpenGL auf orthographisch gestellt, und dem Fenster in x- und y-Richtung der Koordinatenbereich [0, 1] zugewiesen. Für die Angaben der Lage und der Größe eines Schaltelements werden entsprechend vier Werte benötigt. Des Weiteren enthält die Klasse *Button* einen Zeiger auf ein Texturobjekt und einen Wert, der den Button eindeutig identifiziert (*mCallback*), mit dessen Hilfe bei einer Benutzerinteraktion entschieden wird, welche Funktionalität nun durch das Anklicken ausgelöst werden soll. Die Button-Objekte enthalten auch noch einen Flag-Wert, um zusammengehörige Schaltflächen zu Gruppen zusammenfügen zu können. Doch dazu später mehr.

Die Methode *draw* zeichnet den Button, wie bereits erwähnt als einfaches *Quad* an die entsprechenden Koordinaten in der gesetzten Größe in das Anwendungsfenster. Beim Aufruf von *click* wird zunächst überprüft, ob die beiden übergebenen Koordinaten, die ebenfalls schon in den Wertebereich [0, 1] umgerechnet wurden, im Bereich des Buttons liegen. Falls dies der Fall ist, wird der callback-Wert der Schaltfläche zurückgegeben, ansonsten -1. Analog wird in der Methode *release* ebenfalls getestet, ob sich der Mauszeiger wirklich auf dem Button befindet, und entsprechend true oder false zurückgegeben.

Später wurde die Klasse noch um zwei Variablen erweitert. Zum einen wurde ihr ein zweiter Zeiger auf ein *Texture*-Objekt hinzugefügt, zum anderen kam noch eine *bool*-Variable hinzu, mit der sich steuern lässt, welche der beiden Texturen benutzt werden soll. Hierdurch ist es nun möglich beim Klicken eines Buttons

dessen Textur zu wechseln, um leichter und schneller zwischen den beiden zugehörigen Zuständen unterscheiden zu können. Ein Beispiel ist der Schalter, mit dem sich die ganze Simulation an- bzw. ausschalten lässt und der im entsprechenden Zustand einen grünen bzw. roten Knopf darstellt.

Gui

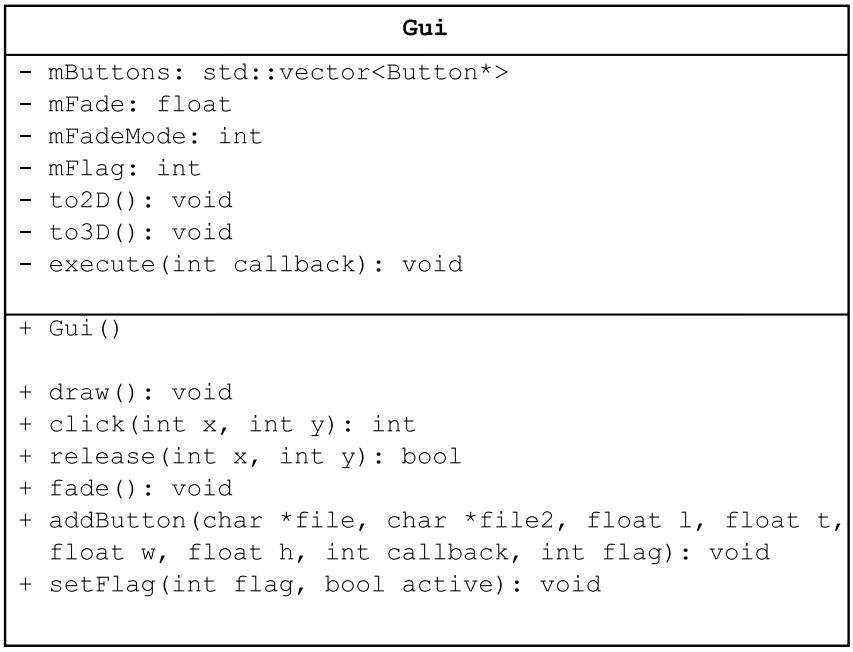


Abbildung 28: Auszug aus der Klasse Gui

In der Klasse *Gui* werden diese Schaltflächen nun verwaltet (*mButtons*). Neben den in Abb. 28 gezeigten Variablen und Methoden enthäht sie zudem noch Zeiger auf verschiedene Objekte, auf die die GUI Einfluss nehmen muss, wie beispielsweise das *Hourglass*- oder das *Painter*-Objekt.

Über die Methode *fade* lässt sich die Oberfläche anzeigen oder verbergen. Der Wert *mFadeMode* steuert dabei die Richtung des Fadens, also ob sie gerade ein- oder ausgeblendet werden soll.

Über die Methode *addButton* werden neue Buttons hinzugefügt. Sie bekommt als Übergabeparameter alle Werte, die benötigt werden, ein neues Button-Objekt anzulegen und es dem Vector *mButtons* hinzuzufügen.

Die Variable *mFlag* regelt die weiter oben bereits erwähnte Gruppierung der Schaltfächen. Jeder wird ein solcher Flag-Wert zugeteilt. Ergibt dann die „Veroderung“ davon mit *mFlag* den Wert 0, bedeutet das, dass der Button gerade nicht sichtbar ist und wird entsprechend bei der Darstellung sowie der Auswertung der *click*-Methode übergangen. Diese übergibt dabei einfach die Koordinaten der Maus an alle gültigen Schaltflächen, wobei der zurückgegebene Wert, der wie oben beschrieben dem Callback des Buttons entspricht, weiterverarbeitet wird.[14]

[14]Diese Technik wurde in der Implementierung am Schluss nicht verwendet, da die Ober-

Hierzu wird die Methode *execute* aufgerufen, die die eigentliche Arbeit verrichtet. Über mehrere Konstanten (Abb.29), von denen jedem Button eine als Callbackwert übergeben wurde, wird in einer einfachen *switch*-Verzweigung geregelt, was beim Klicken auf die jeweilige Schaltfläche passieren soll.

```
static const int CALLBACK_GLOBAL_EXIT;
static const int CALLBACK_GLOBAL_SIMULATION;
static const int CALLBACK_GLOBAL_CLEAR;
static const int CALLBACK_GLOBAL_BGBW;
static const int CALLBACK_GLOBAL_FILL_PLUS;
                    ...
static const int CALLBACK_PAINTER_SWITCH_THIN;
static const int CALLBACK_PAINTER_DRAFT;
                    ...
```

Abbildung 29: Auszüge der Callback-Konstanten

Zusätzliche Steuerungsmöglichkeiten

- Über die Leertaste lässt sich die GUI jederzeit ein- bzw. ausblenden.
- Mit der „#"-Taste kann die aktuelle Anzahl gefüllter Zellen auf der Konsole ausgegeben werden.
- Mit der „F"-Taste kann die aktuelle Frame auf der Konsole ausgegeben werden.
- Mit der „B"-Taste wird ein einfacher Benchmark-Test gestartet und dessen Ergebnis in die Datei „benchmark.txt"abgespeichert.
- Mit der „Esc"-Taste kann das Programm jederzeit beendet werden.
- Bei gedrückter „Alt"-Taste kann die Kamera gesteuert werden.
    - Linke Maustaste oder Pfeiltasten: Kamera drehen
    - Mittlere Maustaste: Kamera bewegen
    - Rechte Maustaste oder +/- Tasten: Zoomen
    - 1: Perspektivische Ansicht
    - 2: Frontalansicht
    - 3: Ansicht von oben

3.7 GUI-Elemente

In diesem Abschnitt werden die Bedienelemente der Benutzeroberfläche vorgestellt.

---

fläche am Ende weniger Elemente enthält als ursprünglich vermutet. Die Variablen und Methoden hierfür wurden dennoch nicht entfernt, falls sie für spätere Versionen doch noch benötigt werden.

Kontrolle

Schaltet die Simulation an bzw. aus.

Standard: Simulation ist ausgeschaltet

Je nach Modus (vgl. Abschnitt 3.3.3) wird hier die Zeichenfläche geleert, die Skulptur zurückgesetzt oder die untere Hälfte der Sanduhr geleert und die obere wieder gefüllt.

Mit diesem Button wird das Programm verlassen.

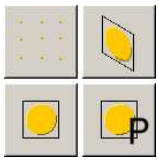
Darstellung

Es werden mehr Partikel für die Darstellung der Sandkörner eingesetzt. Mehr dazu im Abschnitt 3.8.

Es werden weniger Partikel für die Darstellung der Sandkörner eingesetzt. Mehr dazu im Abschnitt 3.8.

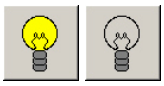
Hier kann die Hintergrundfarbe des Fensters zwischen schwarz und weiß gewechselt werden.

Standard: weiß



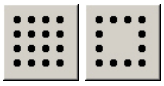
Wechselt den Darstellungsmodus der Sandkörner zwischen „Punkte“, „(ungerichtete) Sprites“, „Billboards“ und „Pseudo-Billboards“. Mehr dazu jeweils im Abschnitt 3.8.

Standard: Punkte



Hier lässt sich die Beleuchtung über die geschätzten Normalen ein- und ausschalten. Mehr dazu im Abschnitt 3.8.

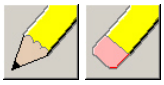
Standard: keine Beleuchtung



Bestimmt, ob die inneren Partikel mitgezeichnet werden oder nicht.

Standard: mit inneren Partikeln

Zeichnen

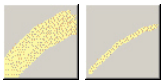


Hier lässt sich umschalten, ob gezeichnet oder radiert werden soll. Mehr dazu im Abschnitt 3.5.

Standard: zeichnen

Mit dieser Schaltfläche wird bestimmt, ob der Mauszeiger beim Zeichnen die Position der Quelle oder des Ziels der fallenden Sandkörner angibt.

Standard: Quelle



Gibt an, ob dicke Linien oder dünne Linien gezeichnet werden sollen. Weitere Erklärungen hierzu befinden sich im Abschnitt 3.5

Standard: dick

Blendet eine Vorlage zum Zeichnen ein bzw. aus. Näheres hierzu ebenfalls im Abschnitt 3.5.

Standard: keine Vorlage

3.8 Visualisierung

Die Algorithmen und Überlegungen zur Simulation des Sandflusses sowie zur Unterscheidung zwischen Sanduhr und Sandskulptur wurden nun ausführlich vorgestellt. Doch bisher wurde noch gar nicht auf die Darstellung des Sandes an sich eingegangen. Im Laufe der Entstehung der vorliegenden Arbeit wurden verschiedene Ansätze ausprobiert, die sich alle in Qualität und Effizienz unterschieden.

Wie in Abschnitt 3.1.1 bereits erwähnt und in Abbildung 6 auch dargestellt, wurden die Sandkörner zu Beginn noch als ganze Kugeln visualisiert, was aber durch die viele Geometrie schnell sehr teuer wird. Doch war es eines der wichtigsten Ziele dieser Arbeit, die Simulation möglichst in Echtzeit darstellen zu können.

Punkte

Deutlich schneller können von der Grafikkarte einzelne Punkte gezeichnet werden, weshalb auch lange dieser Ansatz weiter verfolgt wurde. Jedoch sind sie an sich sehr klein und wenn man pro Zelle der Partikelmanager nur einen Punkt für die Darstellung nutzen würde, würden sehr große Lücken zwischen ihnen auftauchen, was den Eindruck eines kompletten Sandberges enorm stören würde. Dies könnte umgangen werden, indem man die Zellen der Manager sehr klein hält und ihre Anzahl entsprechend erhöht. Jedoch wird das System bei zu vielen Zellen, in denen sich nichts bewegt schnell sehr ineffizient. Eine Alternative wäre es also, pro Zelle nicht nur einen sondern gleich mehrere Punkte darzustellen. Zwar werden dabei immer noch Lücken entstehen aber nicht mehr so große wie zuvor. Beispiele für diese Art der Visualisierung sind die Abbildungen 24 und 26.

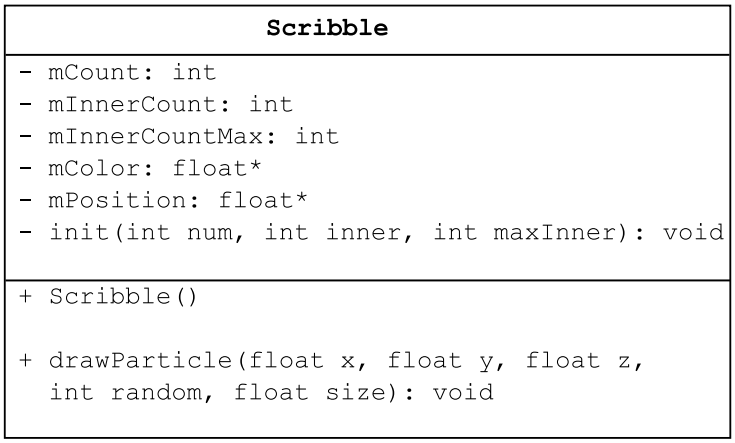


Abbildung 30: Auszüge aus der Klasse Scribble

Damit nicht jede Zelle gleich aussieht wurde die Klasse *Scribble*(Abb. 30) zu deren Verwaltung hinzugefügt. Es werden schon bei der Erstellung des Objekts *Hourglass* verschiedene „Muster“ erzeugt. Die Anzahl wird dabei über die Variable *mCount* gesteuert. Der Wert *mInnerCount* gibt an, wieviele Partikel in jede Zelle gezeichnet werden sollen und wird durch *mInnerCountMax* begrenzt.

Jedes Muster enthält Informationen über die Platzierung der Sandkörner innerhalb einer Zelle sowie über ihre Farbgebung (*mPosition* und *mColor*). Die Positionsangaben liegen im Bereich [-1, 1] und die Farbwerte repräsentieren einen hellen gelb-Ton der für jede Zelle leicht abweicht, damit nicht alle Sandkörner dieselbe Farbe haben.[15] Da jede gefüllte Zelle einen eindeutigen Partikelindex ≥ 0 besitzen, kann dieser Wert verwendet werden um ihr eines der Zufallsmuster zuzuweisen. Hierdurch wird vermieden, dass in jedem Frame für jede Zelle per Zufall ein Muster ausgewählt werden würde, was zu starkem Flackern führen würde sowie zu Performanceeinbrüchen, da jedes mal unzählige Zufallswerte erzeugt werden müssten.

[15] In der vorliegenden Implementation liegen diese Farbwerte im RGB-Bereich (220 - 240, 220 - 240, 150 - 190).

Die Methode *drawParticle* kümmert sich dann um das Zeichnen der Punkte. Es werden entsprechend der Größe *mInnerCount* viele Punkte gezeichnet, bei denen eben eines der zufälligen Muster für die Positionen und eines für die Farbe der einzelnen Sandkörner gewählt wird. Dazu wird der Methode ein Wert (*random*) übergeben, der die Auswahl des Musters bestimmt.[16] Da jedes Muster *mInnerCountMax* viele Angaben enthält, werden, falls *mInnerCount* $<$ *mInnerCountMax*, entsprechend die ersten *mInnerCount* Werte für Position und Farbe gewählt. Als endgültige Positionsangabe, wo die Punkte gesetzt werden, wird zu den der Methode übergebenen Koordinaten die jeweilige Position aus dem gewählten Muster addiert. Das Ergebnis wird dann noch mit dem Größenparameter *size* multipliziert, der wieder dem Größenwert der Klasse *Hourglass* (siehe auch Abschnitt 3.3.3) entspricht.

Sprites

Während der Sand rieselt, sieht die oben beschriebene Darstellungsweise zwar sehr gut aus, doch wenn die Partikel liegen, sind noch große Lücken zu erkennen. Erhöht man die Anzahl der Sandkörner pro Zelle oder vergrößert die Punktgröße mit der die Partikel gezeichnet werden ein wenig, kann man diesen Umstand zwar verbessern, dennoch sehen die Sandberge nicht plastisch aus.

Als nächster Ansatz wurde versucht, die Sandkörner nicht mehr durch Punkte einheitlicher Farbe darstellen, sondern auf Sprites zurückzugreifen. Hierbei handelt es sich um eine Technik, die jedes Partikel mit Hilfe eines kleinen mit einer Textur belegten Quadrats darstellt. In der vorliegenden Implementation wurden insbesondere Billboards[17] eingesetzt. Es besteht zwar die Möglichkeit, derartige so genannte „Pointsprites“direkt mit Hilfe der Grafikkarte umzusetzen, doch sehen die Partikel dann in jeder Entfernung, wie auch schon bei der oben beschriebenen Verwendung der Punkte an sich, gleich groß aus, was insbesondere beim Zoomen äußerst störend wirkt.

Deshalb wurde die notwendige Matrix, mit der die aktuelle Modelview-Matrix multipliziert werden muss, um die durch den gewünschten Punkt gehende, senkrecht zum Blickvektor liegende Ebene zu erhalten, „manuell“aus den Kamerakoordinaten berechnet, die sich wiederum aus der Modelview-Matrix auslesen und in Weltkoordinaten umrechnen lassen.

Sei $MV = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix}$ die aktuelle Modelview-Matrix[Mül05, Folie 6], dann lässt sich die Kameraposition ablesen als $\vec{C_{at}} = \begin{pmatrix} m_{14} \\ m_{24} \\ m_{34} \end{pmatrix}$, muss allerdings zunächst noch in das Weltkoordinatensystem zurückgerechnet werden.

[16] Es wird das Muster mit dem Index *random mod mCount* gewählt.

[17] Sprites, die sich automatisch zur Kamera ausrichten und sich so immer in der Ebene senkrecht zum Blickvektor befinden

Hierzu wird der Rotationsteil von MV transponiert[18] und $\vec{C_{at}}$ mit der so modifizierten Matrix multipliziert. $\vec{C_{at}}' = MV' \cdot \vec{C_{at}}$ ist dementsprechend die Position der Kamera in Weltkoordinaten. Um einen beliebigen Punkt $\vec{p} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ in das Billboardkoordinatensystem umzurechnen, wird wie oben erwähnt eine neue Matrix B aufgestellt, die an MV, also die aktuelle Modelview-Matrix dranmultipliziert wird. Zunächst wird der Blickvektor $\vec{l}$ von der Kamera zu diesem Punkt benötigt, um dann mit zwei aufgestellten Hilfsvektoren $\vec{r}$ und $\vec{u}$[19] die Matrix zu bestimmen. Die Billboardmatrix lautet dann $B = \begin{pmatrix} r_x & u_x & l_x & p_x \\ r_y & u_y & l_y & p_y \\ r_z & u_z & l_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Diese Berechnungen müssen für jedes Sandkorn neu durchgeführt werden, was zusammen mit der Tatsache, dass nun jeweils texturierte Vierecke statt lediglich Punkte gezeichnet werden, zu starken Performanceeinbrüchen führt. Allerdings werden auch die Lücken zwischen den Partikeln kleiner und die Sandflächen wirken durch die Schattierungen auf der Spritetextur ansprechender. Ein wirklich überzeugender Eindruck von 3-Dimensionalität kann aber auch weiterhin ohne Bewegen der Kamera nicht erreicht werden.

„Pseudo"-Billboards

Für diese Arbeit wurde ein zusätzlicher Ansatz entwickelt, der zwar den optischen Eindruck nicht verbessert aber die Darstellung deutlich beschleunigt.[20] Das ständige Zwischenspeichern und Neusetzen der jeweils aktuellen Modelview-Matrix sowie die aufwändigen Matrizen-Berechnungen sind wie bereits erwähnt sehr zeitraubend. Im Folgenden wird daher ein Verfahren vorgestellt, das mit erheblich weniger Matrixoperationen auskommt.

Da die dargestellten Partikel sehr klein sind, fällt es nicht weiter auf, wenn nicht alle genau zur Kamera ausgerichtet sind. Es reicht, wenn sie senkrecht zum allgemeinen Blickvektor der Kamera gedreht sind. In Abb. 31 ist der Unterschied dargestellt. Durch die eigentlich große Entfernung der Kamera zu den Partikeln, ist der Winkelunterschied tatsächlich vernachlässigbar.

Dazu genügt es, an den Sprites dieselben Rotationen wie an der Kamera durchzuführen. Jede Position der Kamera lässt sich dabei, bis auf einen konstanten Faktor um den sie in Blickrichtung gesehen nach vorne oder hinten bewegt wird, durch zwei Rotationen um jeweils eine Achse beschreiben. Die Kamera wird zunächst um einen gewissen Winkel α um die y-Achse und anschließend um einen gewissen Winkel β um die (rotierte) x-Achse gedreht (Abb. 32).

---

[18] $MV' = \begin{pmatrix} m_{11} & m_{21} & m_{31} & m_{14} \\ m_{12} & m_{22} & m_{32} & m_{24} \\ m_{13} & m_{23} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix}$

[19] $\vec{l} = \frac{\vec{p} - \vec{C_{at}}'}{|\vec{p} - \vec{C_{at}}'|}$, $\vec{r} = \vec{C_{up}} \times \vec{l}$, $\vec{u} = \vec{l} \times \vec{r}$ — *l* steht hierbei für *look*, *r* für *right* und *u* für *up*

[20] Zeitmessungen und durch Zahlen belegte Vergleiche sind in Abschnitt 4.3 zu finden.

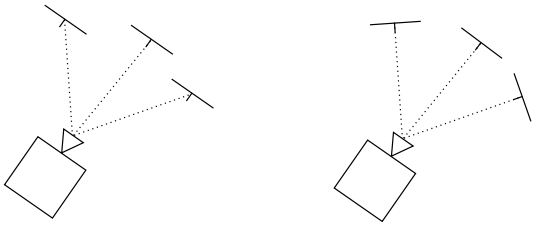


Abbildung 31: Unterschied zwischen „Pseudo“-Billboards(links) und echten Billboards(rechts)

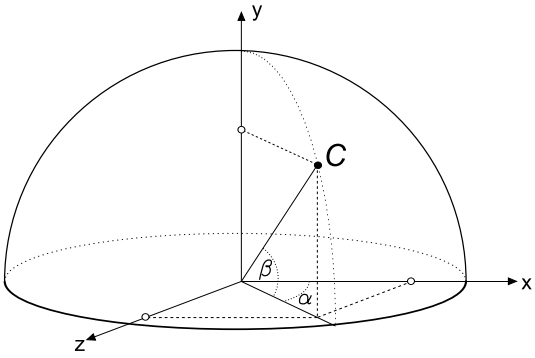


Abbildung 32: Positionsbeschreibung der Kamera durch zwei Rotationen

Die Vektoren $\vec{v_1} = \begin{pmatrix} -1 \\ -1 \\ 0 \end{pmatrix}$, $\vec{v_2} = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}$, $\vec{v_3} = \begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix}$ und $\vec{v_4} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$ die für das Zeichnen des Billboards im Koordinatenursprung verwendet wurden müssen zunächst noch mit einer gewissen Rotationsmatrix M multipliziert werden. M ist dabei die Multiplikation, also die hintereinander geschaltete Anwendung der Rotationsmatrizen um die y- bzw. x-Achse.

$$M = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$\Rightarrow M = \begin{pmatrix} \cos\alpha & \sin\alpha \cdot \sin\beta & \sin\alpha \cdot \cos\beta \\ 0 & \cos\beta & -\sin\beta \\ -\sin\alpha & \cos\alpha \cdot \sin\beta & \cos\alpha \cdot \cos\beta \end{pmatrix}$$

Diese Matrix wird nun auf die oben beschriebenen vier Vektoren angewandt und die Ergebnisse $\vec{b_1}$ bis $\vec{b_4}$ abgespeichert. Das Billboard wird nun wie gewohnt für jedes Partikel an dessen jeweilige Position $\vec{p}$ gezeichnet. Dabei werden jetzt aber nicht mehr die Eckpositionen $\vec{p} + \vec{v_i}$, mit einer Billboardmatrix multipliziert, verwendet, sondern stattdessen die Koordinaten $\vec{p} + \vec{b_i}$, die dann auch schon in absoluten Weltkoordinaten vorliegen.

Die Vektoren $\vec{b_i}$, deren Berechnung bei dieser Methode am meisten Zeit benötigt, müssen dabei lediglich einmal pro Frame aufgestellt werden, weshalb diese

Technik gegenüber den oben beschriebenen regulären Billboards deutlich schneller ist.

Dabei muss lediglich beachtet werden, dass die Kamera stets auf den Koordinatenurpsrung gerichtet sein muss, da sonst die Berechnungen der Orientierung nicht mehr passen und die Billboards zu einer falschen Ebene hingedreht werden.

Normalen-Schätzung

Um den Eindruck von Plastizität zu erwecken, werden Normalen für die Oberflächen benötigt, die dann in der Beleuchtung verwendet werden. Beim vorliegenden System gibt es jedoch keine wirklichen Oberflächen sondern nur eine große Anzahl an Partikeln. Es musste also ein Verfahren entwickelt werden, die Normalen an den Seiten der Sandberge zu schätzen.

Hierfür wird in jeder Spalte des Partikelmanagers die aktuelle Höhe der darin befindlichen Sandsäule berechnet und diese mit der ihrer Nachbarspalten verglichen.

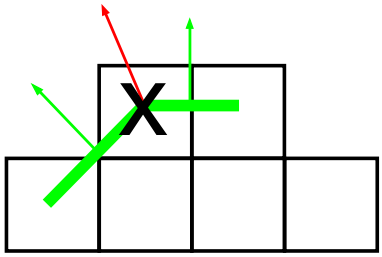


Abbildung 33: Normalen-Schätzung

Die Verbindung der betrachteten Zelle zu jeder Nachbarzelle kann als kleine Oberfläche gesehen werden, für die die Normale mit dem Höhenunterschied sehr leicht berechnet werden kann, insbesondere da der Höhenunterschied aufgrund der Implementierung bei einem ruhenden Sandberg vom Betrag her nur ≤ 1 sein kann. In Abb. 33 ist dies für den 2D-Fall angedeutet, lässt sich aber auch leicht in den 3-dimensionalen Raum übertragen, so dass jeder Punkt acht „Nachbarnormalen“besitzt, für die die man eine einfache Tabelle aufstellen kann (Ausschnitt: siehe unten).

Δx	-1		
Δz	-1	0	1
n_x	$\frac{\sqrt{3}}{3}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}}{3}$
n_y	$\frac{\sqrt{3}}{3}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}}{3}$
n_z	$\frac{\sqrt{3}}{3}$	0	$-\frac{\sqrt{3}}{3}$

Auszug aus der Normalen-Tabelle (hier: $hoehe_{nachbar} > hoehe_{betrachtet}$)

Diese Normalen (Abb. 33, grün) werden aufaddiert und komponentenweise durch die Anzahl geteilt, um so eine gemittelte Normale zu erhalten (Abb. 33, rot). Für jede Spalte wurde so eine Normale geschätzt, die nun für jede Zelle darin gilt. Da man normalerweise nur die oberste Schicht sieht, und die Normale ja eben

hierfür eigentlich berechnet wurde, ist dies auch nicht weiter schlimm.

Als Lichtquelle wurde ein einfaches weißes Pointlight verwendet, dass immer ein klein wenig unter der Kamera positioniert wird, um wirklich aus jeder Blickrichtung beleuchten zu können.

Durch die Matrizenmultiplikationen im Billboard-Modus stimmen die Normalen hier nicht mehr, wodurch die Partikel hier falsch beleuchtet werden. Dies ist ein Fehler der bis zum Abschluss der Arbeit leider nicht mehr behoben werden konnte.[21] Es muss auch angemerkt werden, dass dieser Ansatz bei Gefäßwänden nicht angewendet werden kann, da man hier auch Partikel sieht, die unterhalb der obersten Schicht liegen, für die die Normalen dann nicht mehr stimmen. Auch dies konnte bis zum Abschluss der Arbeit nicht mehr ausgebessert werden.

Zwar ist der Eindruck von Plastizität nun deutlich höher, dafür leidet die Framerate allerdings recht deutlich unter der eingeschalteten Beleuchtung.[22]. Um dem entgegen zu wirken wurde in die Benutzeroberfläche die Option eingebaut, die Beleuchtung ein- und auszuschalten. Des Weiteren besteht nun auch die Möglichkeit, die inneren Partikel bei der Darstellung auszuschließen.

[21] Er ist jedoch insofern verschmerzbar, da die Beleuchtung bei den Pseudo-Billboards funktioniert, diese den richtigen Billboards rein optisch gleichen und dabei sogar schneller sind.

[22] Genaue Messungen hierzu im Abschnitt 4.3

4 Ergebnisse

4.1 Zusammenfassung

Wie in Abschnitt 1.3 aufgelistet, gab es für diese Studienarbeit verschiedene Teilaufgaben, die es zu lösen galt. Im Folgenden werden die Lösungswege noch einmal zusammengefasst. Anschließend wird es eine Sammlung von Abbildungen geben, die auch den Verlauf der Arbeit rekapitulieren.

Zu Beginn der Arbeit wurde noch versucht, die Simulation physikalisch korrekt zu berechnen (3.1.1), was aber schon durch den hohen Aufwand für die Kollisionsberechnung ($O(n^2)$) sehr zeitaufwändig wurde.

Durch die Entwicklung eines „Partikel-Managers“(3.2) zur Verwaltung der Partikel sowie eines abstrakten Algorithmus für fallende und rutschende Partikel konnte die Simulation extrem vereinfacht und das Problem dadurch weitgehend gelöst werden.

Als zentrale Schnittstelle wurde die Klasse *Hourglass* (3.3.3) entwickelt. Sie verwaltet alle nötigen Objekte und Methode, die zur Erstellung, Simulation sowie Darstellung des Systems nötig sind und ist damit das einzige im Hauptprogramm notwendigerweise bekannte Objekt.

Des Weiteren wurde die Möglichkeit implementiert, 3D-Modelle direkt aus externen Dateien einzulesen und diese als Formen zu verwenden (3.4). Hierzu wurde ein effizienter Algorithmus entwickelt, der bestimmt, welche Zellen des Managers innerhalb und welche außerhalb der Form liegen. Durch eine geschickte Codierung war es auch sehr einfach möglich diese Hülle entweder strikt zu verstehen, was einer Sanduhr glich, oder auch als lediglich initial formgebend, was mehr einer zerfallenden Sandskulptur entspräche.

Als zusätzliche Demonstration des Sandflusses wurde die Option hinzugefügt, mit dem Sand zu „malen“(3.5), wobei eine Sandspur dem Mauszeiger folgt. Die Markierung ungültiger Zellen, wie sie schon bei den Sanduhren zum Einsatz kam, konnte hier wieder genutzt werden, um einfache Hindernisse einzubauen, auf denen der Sand liegen blieb. Andere „Gimmicks“, wie Radieren, unterschiedliche Liniendicke und das Einblenden von Vorlagen, wurden von der Datenstruktur derart leicht ermöglicht, dass sie ebenfalls mitimplementiert wurden.

Um die Bedienung der Anwendung zu vereinfachen, wurde auch eine GUI eingebaut (3.6), die wiederum auf einem extra für diese Arbeit entwickelten System basiert. Im Abschnitt 3.7 sind alle Bedienelemente mit ihrer Funktion aufgelistet.

Auch wenn es nie primäres Ziel der Arbeit war, den Sand realistisch darzustellen, wurde zuletzt auch noch versucht, den Sand ansprechend zu visualisieren (3.8). Hierfür wurden verschiedene Verfahren entwickelt und miteinander auf deren Performance hin verglichen (4.3).

4.2 Bilder

Erste Gehversuche

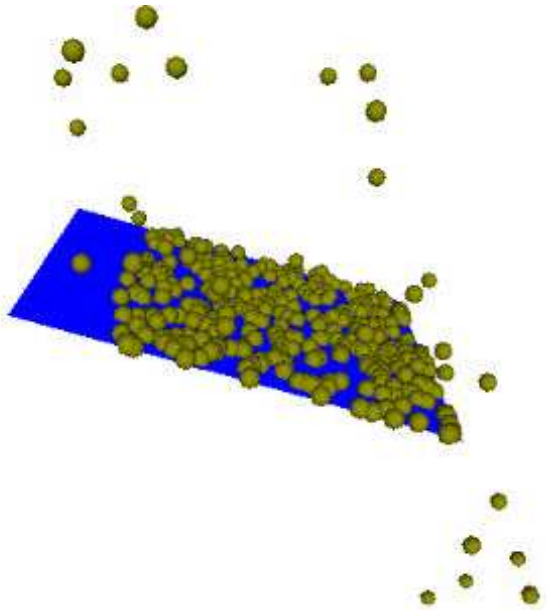


Abbildung 34: Erstes Partikelsystem

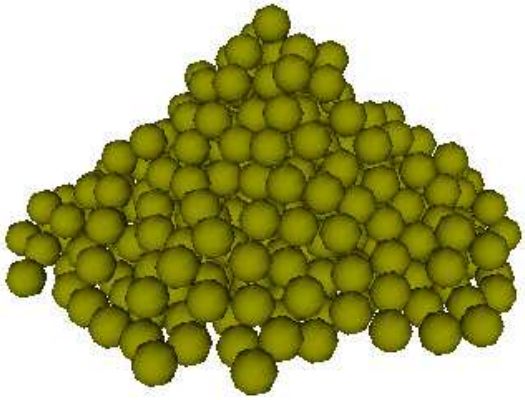


Abbildung 35: Partikelsystem mit Kollision zwischen den Partikeln

Erste Tricks

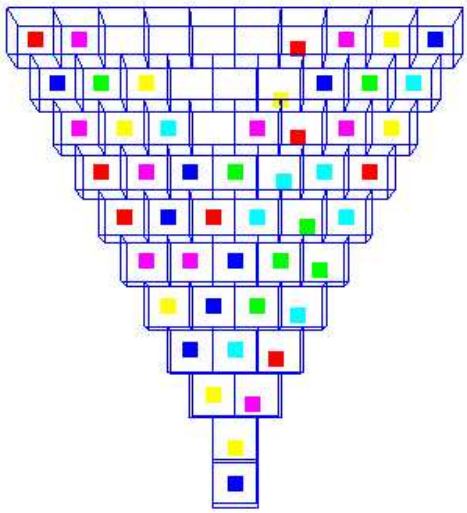


Abbildung 36: Erste Version des oberen Partikelmanagers

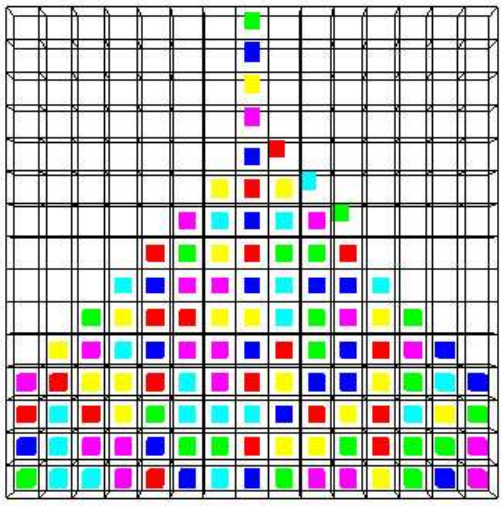


Abbildung 37: Erste Version des unteren Partikelmanagers

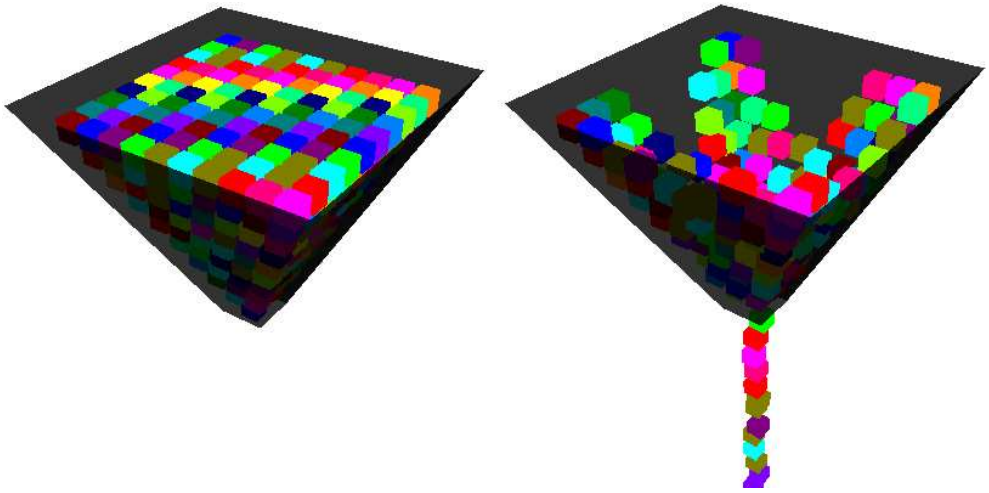


Abbildung 38: 3D-Version des oberen Partikelmanagers

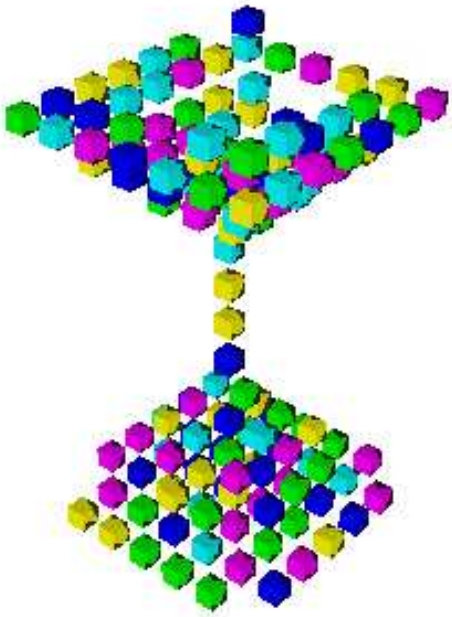


Abbildung 39: Vereinheitlichter Partikelmanager; für beide Hälften einsetzbar

Erste Formen

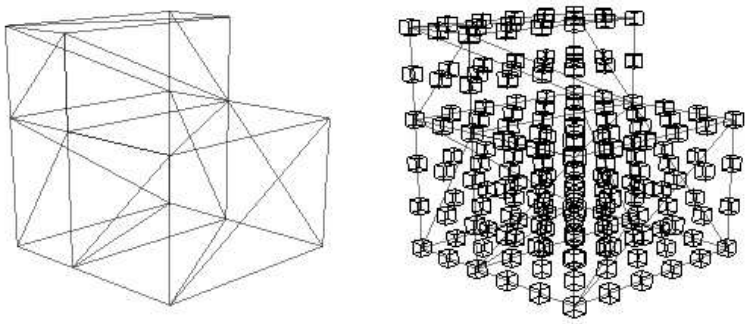


Abbildung 40: Demonstration des Füllalgorithmus

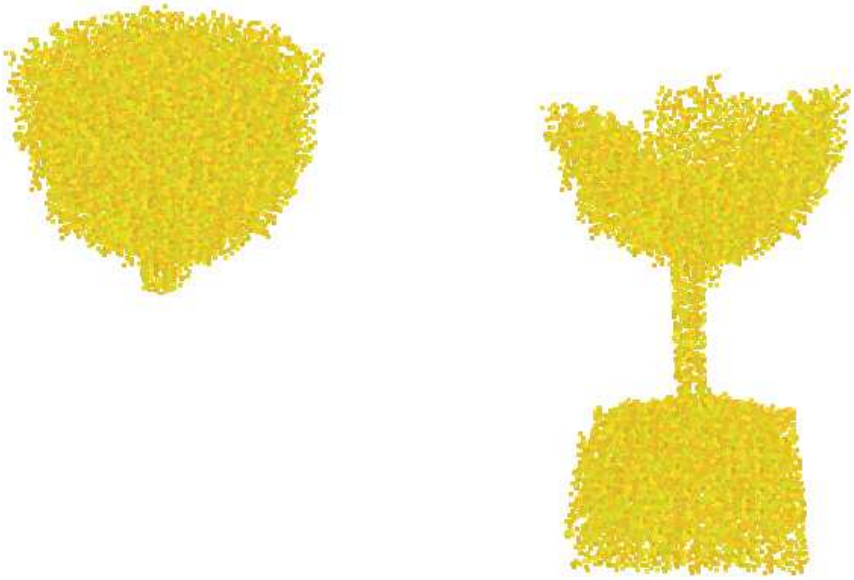


Abbildung 41: Erstes aus externer Datei geladenes 3D-Objekt in Aktion

Finale Version

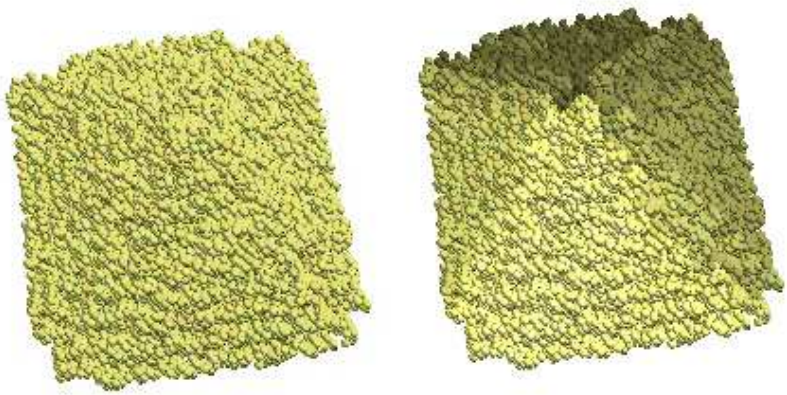


Abbildung 42: Gegenüberstellung ohne und mit Beleuchtung

In Abb. 42 wird deutlich, dass ohne Beleuchtung kein wirklicher Eindruck von Plastizität entstehen kann (links). Erst bei eingeschalteter Beleuchtung ist zu erkennen, dass es sich hier um einen Sandhügel handelt (rechts).

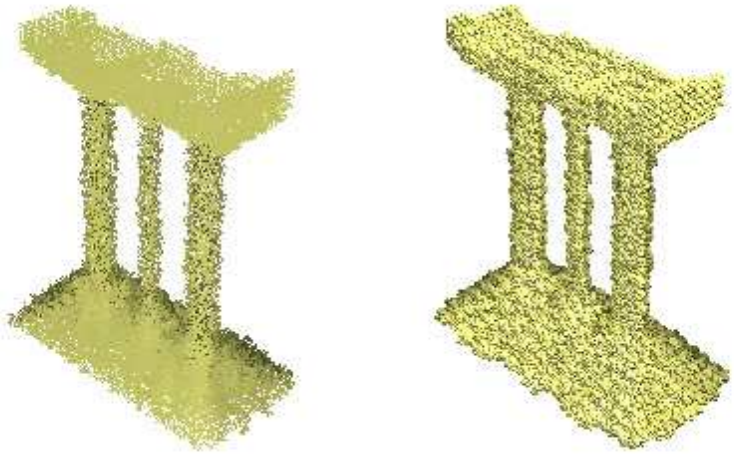


Abbildung 43: Sanduhrmodell (vgl. Abb. 50); mit Punkten und Pseudo-Billboards

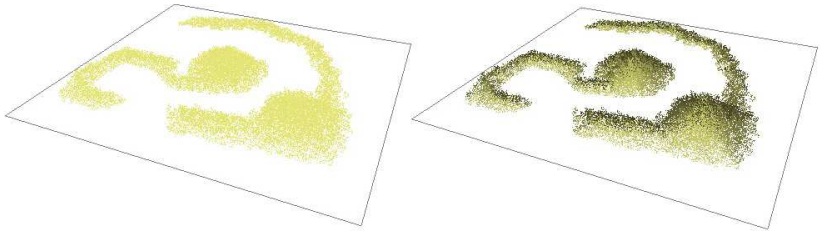


Abbildung 44: Darstellung des Sandes mit Punkten

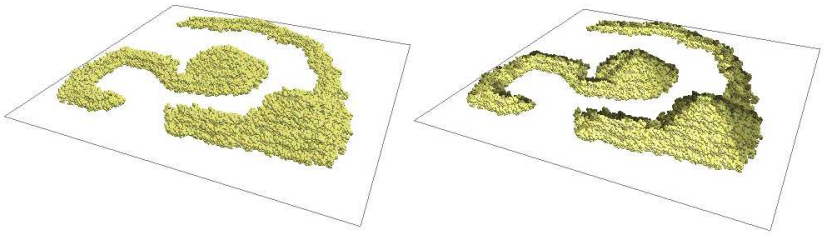


Abbildung 45: Darstellung des Sandes mit Pseudo-Billboards (siehe 3.8)

Abbildung 46: Sandgemälde mit Vorlage

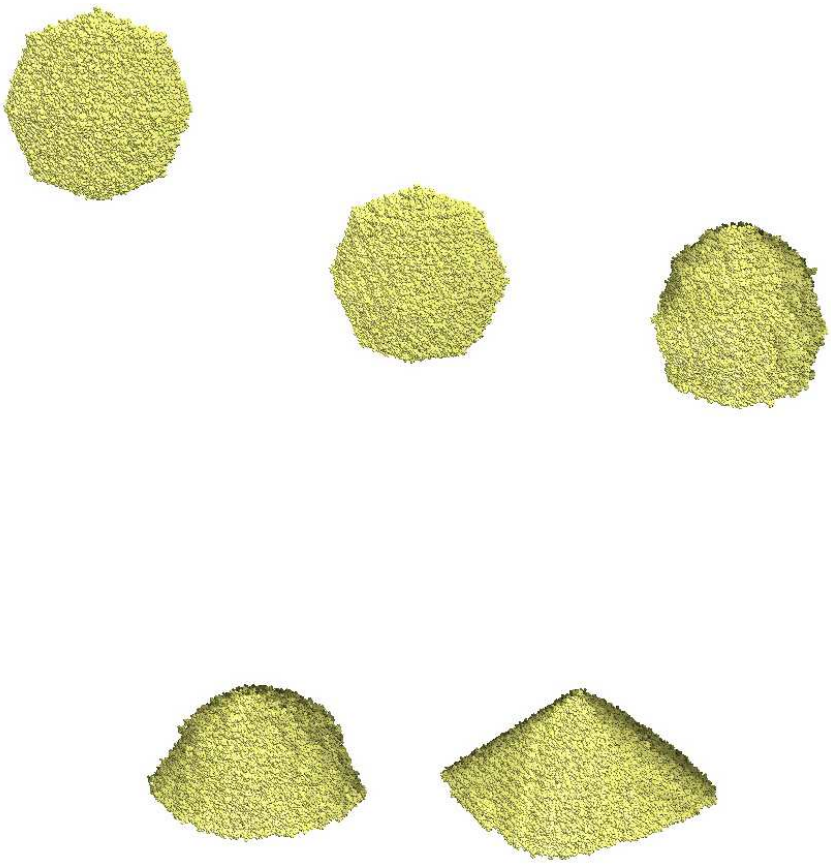


Abbildung 47: Zerfallende Sandskulptur (Ball)

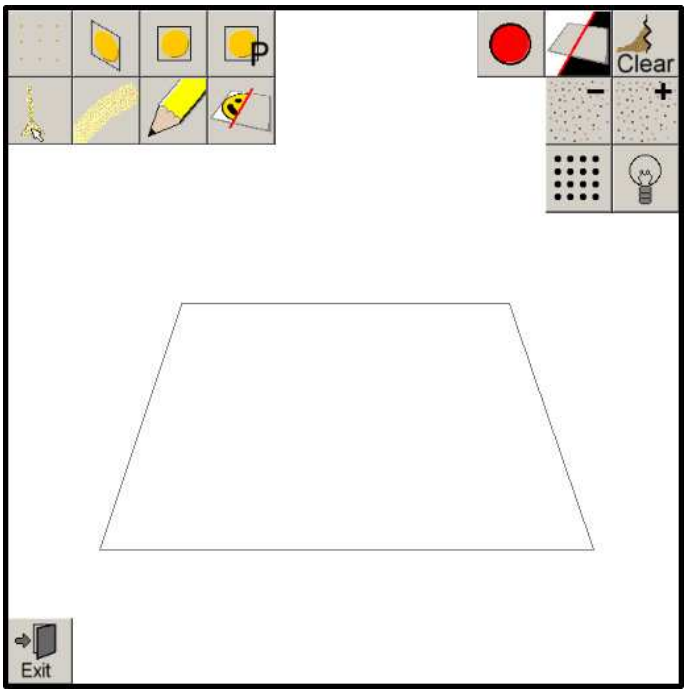


Abbildung 48: Benutzeroberfläche

Nebenprodukte

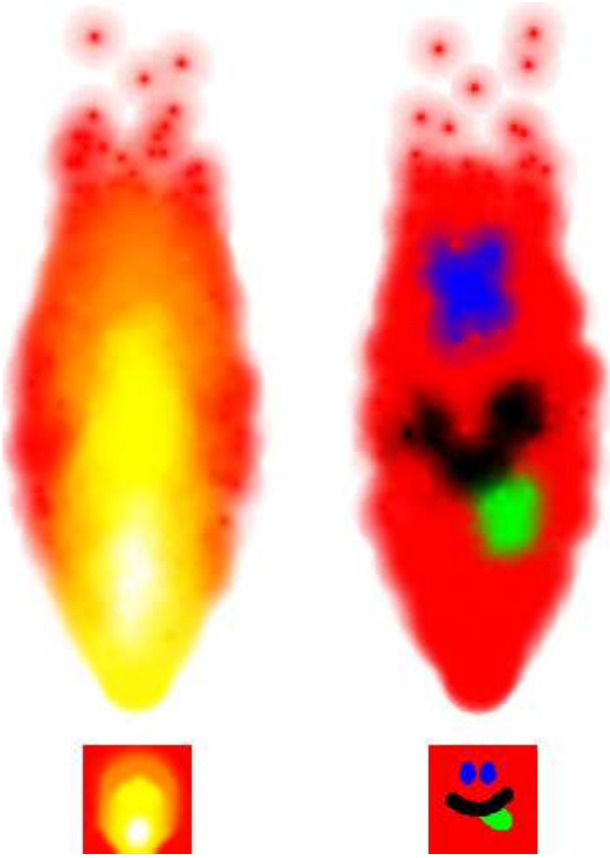


Abbildung 49: Beispielflammen mit den zugehörigen LookUp-Maps

4.3 Zeitmessungen

Da Performanz ein wichtiges Teilziel dieser Arbeit war, wurden, um die verschiedenen Modi vergleichen zu können, einige Tests mit einem Partikelmanager der Größe $60 \times 10 \times 60$ durchgeführt, deren Ergebnisse hier zu finden sind.

Gestetet wurde auf folgendem System:

- AthlonXP 3000+
- 1 GB RAM
- GeForce 6800GT

Malen

Ohne Beleuchtung:

Sim	Gui	Modus	PpZ	gZ	Licht	innen	FPS
an	an	P	20	500	aus	ja	**224**
an	an	BB	20	500	aus	ja	**78**
an	an	PBB	20	500	aus	ja	**141**
an	an	P	20	2500	aus	ja	**142**
an	an	BB	20	2500	aus	ja	**21**
an	an	PBB	20	2500	aus	ja	**52**

Erklärung zu den Spalten:

Sim	Simulation
Modus	P (Punkte), BB (Billboards), PBB (Pseudo-Billboards)
PpZ	Partikel pro Zelle
gZ	Gefüllte Zellen
innen	„Innere“ Zellen werden gezeichnet
FPS	Frames pro Sekunde

Mit Beleuchtung:

Sim	Gui	Modus	PpZ	gZ	Licht	innen	FPS
an	an	P	20	500	an	ja	**179**
an	an	BB	20	500	an	ja	**67**
an	an	PBB	20	500	an	ja	**123**
an	an	P	20	2500	an	ja	**110**
an	an	BB	20	2500	an	ja	**18**
an	an	PBB	20	2500	an	ja	**46**
aus	aus	P	20	2500	an	ja	**259**
aus	aus	BB	20	2500	an	ja	**20**
aus	aus	PBB	20	2500	an	ja	**59**

Sanduhr

Mit inneren Zellen:

Sim	Gui	Modus	PpZ	gZ	Licht	innen	FPS
an	an	P	1	3482	aus	ja	**312**
an	an	BB	1	3482	aus	ja	**159**
an	an	PBB	1	3482	aus	ja	**236**
an	an	P	20	3482	aus	ja	**149**
an	an	BB	20	3482	aus	ja	**16**
an	an	PBB	20	3482	aus	ja	**41**
an	an	P	20	3482	an	ja	**121**
an	an	BB	20	3482	an	ja	**13**
an	an	PBB	20	3482	an	ja	**38**

Ohne innere Zellen:

Sim	Gui	Modus	PpZ	gZ	Licht	innen	FPS
an	an	P	1	3482	aus	nein	**341**
an	an	BB	1	3482	aus	nein	**274**
an	an	PBB	1	3482	aus	nein	**314**
an	an	P	20	3482	aus	nein	**271**
an	an	BB	20	3482	aus	nein	**29**
an	an	PBB	20	3482	aus	nein	**98**
an	an	P	20	3482	an	nein	**225**
an	an	BB	20	3482	an	nein	**24**
an	an	PBB	20	3482	an	nein	**90**

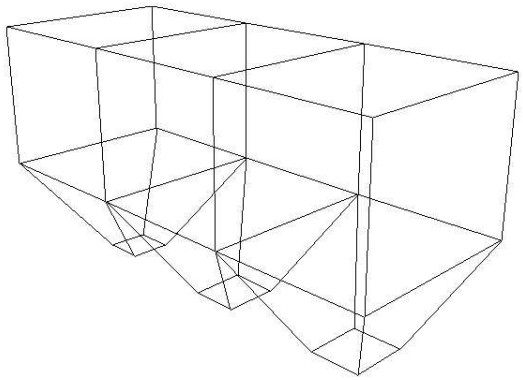


Abbildung 50: Für die Zeitmessung verwendetes Sanduhr-Modell

5 Fazit und Ausblick

Zusammenfassend lässt sich sagen, dass alle in Abschnitt 1.3 aufgelisteten Teilziele erreicht wurden. Die Entscheidung auf Literatur weitgehend zu verzichten, hatte sowohl Vor- als auch Nachteile. Zwar wurde so für die Lösung von zunächst anscheinend kleinen Problemen teilweise recht zeitraubend, auf der anderen Seite war dafür das Erfolgserlebnis jeweils um so größer. Doch gerade der Anspruch, die Simulation möglichst in Echtzeit zu berechnen schränkte die in Frage kommende Literatur ohnehin stark ein.

Die Erstellung eines Projekts dieses Größenumfangs war eine große aber auch interessante Herausforderung. Die Aufteilung in einfache, losgelöste Klassen und in solche, die mehrere der Kleineren verwalten, ist dabei recht gut gelungen. Beim Bearbeiten kam es jedoch mehrfach vor, dass der Code nicht ausführlich genug kommentiert worden war oder im Nachhinein unverständlich erschien, so dass das gesamte Projekt neu aufgesetzt wurde. Hierbei konnte immer auf das inzwischen erworbene Wissen zurückgegriffen werden, so dass bei der Neuerstellung immer auch aus softwaretechnischer Sicht Verbesserungen eingebaut werden konnten und veraltete Methoden herausgenommen wurden. Die vorliegende, finale Version stellt hierbei bereits die vierte „Generation“ dar.

Dabei kam ein System heraus, mit dem mit sehr einfachen Algorithmen optisch überzeugende Simulationen für das Fließverhalten von Sand dargestellt werden können, auch wenn diese praktisch keinerlei physikalische Plausibilität mehr besitzen. Es wurden zur Demonstration des Systems auch einfache Beispielanwendungen wie die Sanduhren und das „Malen“ mit Sand implementiert. Es wurde dabei auch darauf geachtet, dem Anwender bei den Einstellungen viel Freiraum zu lassen, so dass er das System bequem an seine Wünsche anpassen kann.

Die Frameraten variierten stark bei den verschiedenen Einstellungen. Im Punkt-Modus können die Sandkörner zwar sehr schnell dargestellt werden, jedoch bleiben hierbei recht große Lücken dazwischen. Die Billboards konnten dieses Problem gut beheben, verringern aber auch deutlich die Performance. Die Pseudo-Billboards boten praktisch gleiche Darstellungsqualiät jedoch bei höherer Geschwindigkeit und sind deshalb vorzuziehen. Lässt man die inneren Zellen bei der Darstellung weg, kann die Performance noch einmal enorm verbessert werden, insbesondere beim Zeichnen der (Pseudo-)Billboards wurde die Framerate hiermit erhöht.

Die Beleuchtung und damit auch die Berechnung bzw. Schätzung der Normalen verringern zwar noch einmal die Geschwindigkeit, sind aber notwendig, um einen Tiefeneindruck des Sandes zu erhalten und Struktur zu erkennen.

Durch die flexible Handhabung der Einstellungen kann immer der gewünschte Mittelweg zwischen Darstellungsqualität und -geschwindigkeit gewählt werden.

Als eine weitere denkbare Anwendung könnte man sich einen Bagger vorstellen, der durch den Sand fährt und ihn dabei vor sich herschiebt, während die Berge zu seiner Seite hinter ihm wieder in den Graben rutschen, den er durch seine Fahrt im Sand hinterlässt.

Verschiedene Attribute sind noch fest im Code integriert. Das betrifft sowohl die Simulation, wie zum Beispiel die Gitterauflösung, als auch die Darstellung, wie zum Beispiel Punkt- bzw. Spritegröße. Als Erweiterung könnte man auch die-

se Werte in die Benutzeroberfläche mit aufnehmen. Dadurch wird sie zwar sehr voll, doch die Möglichkeit, die Elemente mit Hilfe von Flags zu gruppieren, um dann bequem mehrere auf einmal ein- bzw. auszublenden, ist bereits eingebaut (siehe auch 3.6).

Auch die Auswahl des Modells ist noch fest. Ein Dialog zur Auswahl einer Datei ist im vorliegenden GUI-Konzept nicht vorgesehen und müsste entsprechend extra eingebaut werden. Sogar die Auswahl, ob man eine Sanduhr, eine Sandskulptur oder doch eine leere Fläche zum Malen erstellen möchte, wird noch im Code geregelt und erfordert bei einem Wechsel derzeit noch eine Neukompilierung des Programms.

Die visuelle Darstellung an sich ist zwar zufrieden stellend aber noch suboptimal. Unter den Versuchen, die Partikel schöner darzustellen, litt jeweils deutlich die Geschwindigkeit des Systems. In Verbindung mit der GPU lässt sich das Ganze bestimmt noch ohne große Performanceeinbußen etwas verbessern. Des Weiteren müsste auch die Normalenschätzung insbesondere bei den Billboards und den Gefäßwänden noch einmal überarbeitet werden.

Bei der Entstehung des Programms konnte dabei natürlich auch tiefer in die genutzte Entwicklungsumgebung *Microsoft Visual Studio .NET 2003* eingestiegen werden. Zu Beginn wurde noch mit *Bloodshed Dev-C++* gearbeitet, das sich für ein Projekt dieser Größenordnung aber schnell als unhandlich herausstellte.

Zudem konnte bei der Anfertigung der Ausarbeitung ein besserer Eindruck des Softwarepakets LaTeXgewonnen werden, das sich beim Schreiben und Formatieren eines solch umfangreichen Dokuments als praktisch unverzichtbare Hilfe erwies.

Neben den eigentlichen Teilen dieser Arbeit wurden auch Ideen entwickelt und umgesetzt, die primär nichts mit Sandsimulation zu tun haben, aber dennoch ihre Erwähnung finden sollten. Ihnen wurde der Abschnitt „Nebenprodukte“(6) gewidmet.

6 Nebenprodukte

Während der Entstehung dieser Arbeit wurden immer wieder Versionen und Ideen kreiert und dann doch wieder verworfen. Doch dabei kamen neben kleineren Anwendungen auch Programme heraus, die aufgrund ihres Aufwands eigentlich doch erwähnt werden sollten, auch wenn sie letztendlich nichts mehr mit dem eigentlichen Endprodukt zu tun haben.

6.1 Konverter

Für die Formen der Sanduhren und -skulpturen wurde hauptsächlich das 3D-Modellierungsprogramm *Cinema4D* verwendet. Hierbei wurden die Dateien im *Wavefront (*.obj)*-Format exportiert. Dieses wurde gewählt, da es recht einfach aufgebaut und auch vom Menschen gelesen werden kann.

Auch bei den Modellen war es im Sinne der vorliegenden Arbeit ein Ziel, keine vorgegebenen Loader zu verwenden, sondern auch hierfür die Algorithmen neu zu schreiben.

Zwar lassen sich auch mit der Programmiersprache *C++*, die in dieser Arbeit hauptsächlich verwendet wurde, externe Dateien öffnen und einlesen, jedoch ist die String-Behandlung relativ umständlich. Deutlich einfacher geht dies beispielsweise mit der Sprache *Visual Basic*. Deswegen wurde hiermit ein Konverter geschrieben, der die Informationen der **.obj*-Datei so anordnet und in eine neue Datei speichert, dass sie auch von *C++* aus sehr leicht eingelesen werden können. Dieses neu entstandene Dateiformat wurde **.tri* genannt.

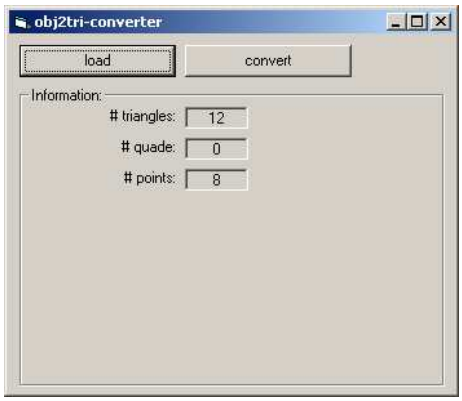


Abbildung 51: Konverter „obj2tri“

In der Variante des **.obj*-Formats, wie es von *Cinema4D* exportiert wird, wird zunächst ein Kommentar ausgegeben. Dann werden alle auftauchenden Punkte mit ihren durch Leerzeichen getrennten xyz-Koordinaten zeilenweise aufgelistet. Danach folgen alle Texturkoordinaten. Zum Schluss kommen die Informationen über alle Flächen, bestehend aus den Indizes der jeweiligen zu verbindenden Punkte in Kombination mit dem zugehörigen Index der Texturkoordinaten.

Der Konverter „obj2tri“ liest die Informationen ein und speichert sie dann folgendermaßen ab. Zu Beginn wird eine Kommentarzeile über die Art der Flächen

```
# WaveFront *.obj file

v -129.691895 -100 56.391602
v -129.691895 100 56.391602
.
.
.

vt 1 1 0
vt 0 1 0
.
.
.

f 1/2 2/6 4/11
f 3/8 4/12 6/15
.
.
.
```

```
# triangles
12
// triangle[1]
-129.691895
-100
56.391602
-129.691895
100
56.391602
56.391602
100
129.691895
// triangle[2]
56.391602
-100
129.691895
.
.
.
```

Abbildung 52: Gegenüberstellung der beiden Formate; links **.obj*, rechts **.tri*

ausgegeben, ob es sich um Dreiecke oder Vierecke handelt.[23] Danach folgt die Anzahl der Dreiecke. Getrennt durch je eine Kommentarzeile zur Übersicht werden nun nacheinander alle Flächen aufgelistet, wobei statt Indizes direkt die jeweiligen Koordinaten der Eckpunkte gespeichert werden. Dabei wird pro Zeile nur eine Angabe ausgegeben. Dies erhöht natürlich den Speicherbedarf, was aber bei Betrachten der Tatsache, dass der Aufwand beim Einlesen aus *C++* heraus minimiert wird, in Kauf genommen werden kann. Texturkoordinaten werden für den eigentlichen Zweck der **.tri*-Dateien nicht mehr benötigt und brauchen auch nicht mehr mitgespeichert werden.

Beispielhafte Auszüge aus den beiden Dateiformaten sind in Abbildung 52 dargestellt.

6.2 Feuer

Immer wieder werden Algorithmen beschrieben, die die Simulation von Feuer bewirken sollen. Im Folgenden wird ein Ansatz vorgestellt, der sowohl die Implementation von Feuer leicht ermöglicht als auch die Farbgebung einfach und gleichzeitig flexibel hält. Wie die meisten anderen Algorithmen basiert auch dieser auf Partikelsimulation.

Die Variable *mP_num* repräsentiert die Anzahl der verwendeten Partikel. Die Größen *mP_pos*, *mP_col* und *mP_age* geben Position, Farbe und Alter jedes Parti-

[23] In dieser Arbeit wurde prinzipiell nur mit Dreiecken gearbeitet. Der Konverter kann aber für spätere Zwecke auch Vierecke verarbeiten.

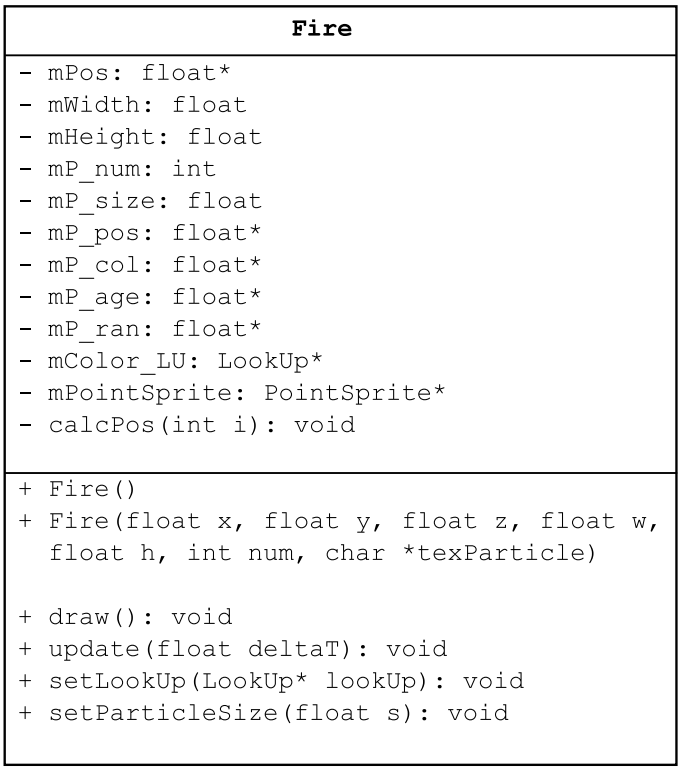


Abbildung 53: Auszug aus der Klasse Fire

kels an. In der Methode *draw*, die das Feuer zeichnen soll, wird zunächst in den Pointsprite-Modus geschaltet.[24] Danach wird für dieses Partikel die entsprechende Farbe gesetzt und ein Punkt, also ein Sprite, an die aus *mP_pos* gegebene Stelle gesetzt.

Die Bewegung der Partikel wird in der Methode *update* gesteuert. Hier wird für jedes Partikel der übergebene Wert *deltaT* zu dessen Alter aufaddiert. Die neue Position wird dann in der Methode *calcPos* berechnet. Zum Schluss wird dem Partikel noch seine zu dieser neuen Position zugehörige Farbe berechnet.

Position der Partikel

Die Idee war es, die Form der Flamme durch eine einfache quadratische Funktion zu modellieren (Abb. 54).

Jedes Partikel folgt bei seinem Flug einer solchen festen Bahn. Der Parameter *r*, der die „Höhe“ der Funktion angibt, also später die „Breite“ der Flugbahn, wird dabei für jedes Partikel mit abgespeichert (*mP_ran*[25]) und wird initialisiert als ein Zufallswert im Bereich [-w, w], wobei *w* für die Breite des Feuers steht (*mWidth*).

[24] Da die Texturen für die Partikel hier halbtransparent sind und Größenunterschiede bei verschiedenen Entfernungen keine wirkliche Rolle spielen, wurde hier im Gegensatz zu Abschnitt 3.8 die Pointsprites-Implementierung der Grafikkarte verwendet.

[25] *ran* steht dabei als Kürzel für *range*

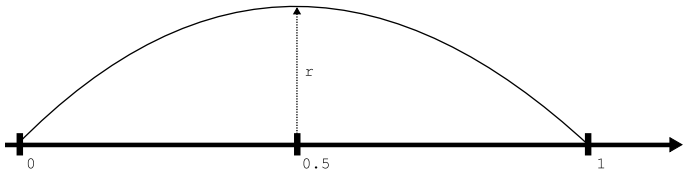


Abbildung 54: Modellierungsfunktion der Flammenform

Betrachtet man die Modellierungsfunktion im mathematisch üblichen Sinne[26], also wie in Abb. 54 dargestellt, so kann man als *x*-Parameter das Alter des Partikels nehmen, das von 0.0 bis 1.0 läuft. Die Funktion soll quadratisch sein, also die Form $f(x) = a \cdot x^2 + b \cdot x + c$ besitzen.

Als bekannt werden dabei die folgenden Stellen gesehen:

$f(0.0) = 0$

$f(0.5) = r$

$f(1.0) = 0$

Eingesetzt ergibt dies die folgenden Gleichungen:

$c = 0$

$\frac{a}{4} + \frac{b}{2} + c = r$

$a + b + c = 0$

Setzt man $c = 0$ in die anderen Gleichungen ein, so bleibt:

$\frac{a}{4} + \frac{b}{2} = r$

$a + b = 0$

Löst man $a + b = 0$ nach b auf, also $b = -a$ und setzt dies in die andere Gleichung ein, so erhält man $\frac{a}{4} - \frac{a}{2} = -\frac{a}{4} = r \Leftrightarrow a = -4 \cdot r$ und damit auch $b = 4 \cdot r$.

Die fertige Funktion lautet also $y = f(x) = -4 \cdot r \cdot x^2 + 4 \cdot r \cdot x$ bzw. angewandt als Flugbahn $x = -4 \cdot r \cdot age^2 + 4 \cdot r \cdot age$. Dies kann auch für negative r verwendet werden. Nimmt man an, die Partikel bewegen sich gleichmäßig, kann als y-Koordinate vereinfacht angenommen werden, dass $y = age \cdot h$, wobei h die Höhe der Flamme, also *mHeight* darstellt.

Dies wird alles in der Methode *calcPos* berechnet[27]

[26] Als Flugbahn muss man sich diese Abbildung um 90° gegen den Uhrzeigersinn gedreht vorstellen, damit die Flammen auch wirklich von unten noch oben steigen.

[27] Die z-Koordinate der Partikel wurde hier einfach stets auf 0 gesetzt. Es wäre möglich auch hierfür dieselbe Flugbahn-Funktion zu verwenden, um dreidimensionale Flammen zu erzeugen.

Farbe der Partikel

Auch die Farbgebung der Partikel wurde sehr einfach gehalten. Hierzu wird dem Flammenobjekt eine Textur als LookUp-Map zur Verfügung gestellt. Ausgelesen wird eine Farbe, in dem in der *update*-Methode nach der Neuberechnung der Position eines Partikels seine Koordinaten, normiert auf den x-Bereich [-1, 1] und den y-Bereich [0, 1], an das LookUp-Objekt übergeben werden, das diese Werte auf die Bereiche [0, textureWidth] und [0, textureHeight] zurückrechnet und dann an der entsprechenden Stelle die Farbe aus der Textur ausliest und diese zurückgibt. Hiermit wird dann die Pointsprite-Textur für die Darstellung eingefärbt.

Abbildung 55: Beispiele für LookUp-Maps der Flammen

Wie in Abb. 55 zu sehen ist, ist jede beliebige Farbgebung für die Flammen denkbar. So kann neben normalem Feuer (links) auch beispielsweise blaues Feuer (mitte) dargestellt werden. Es können sogar ganz andere Sachen in einer Flamme dargestellt werden, wie zum Beispiel ein Smilie (rechts).

Sterben der Partikel

Wenn das Alter eines Partikels einen gewissen Grenzwert überschreitet, z.B. 0.8, kann es passieren, dass das Partikel als zu alt gekennzeichnet wird und daher verschwindet. Hierzu wird im *update*-Vorgang für jedes Partikel, dessen Alter jenseits des Schwellwerts liegt mit einem gewissen „Test“ entschieden, ob es ausgeblendet werden soll oder nicht. Dieser Test stellt ein einfaches Zufallsexperiment mit einer Wahrscheinlichkeit von zehn Prozent dar. Schlägt der Test fehlt, also in 90% aller Fälle, so wird das Partikel regulär weiterbehandelt. Sollte er jedoch klappen, so wird das Alter des Partikels auf 0 zurückgesetzt und seine Reichweite (*mP_ran*) neu berechnet.

Es scheint zunächst so, dass die Partikel auch sehr spät noch recht lange leben würden. Dies ist aber der nicht der Fall. Aufgrund der Tatsache, dass dieser Test in jedem Frame durchgeführt wird, werden die Partikel recht schnell zurückgesetzt.

Des Weiteren wird bei der Positionsberechnung für jedes Partikel mit einem Alter größer als dem Schwellwert die Neuberechnung der x-Koordinate ignoriert und das Partikel fliegt ab sofort nur noch senkrecht nach oben. Dadurch kommt der Effekt hinzu, dass die Flamme nach oben hin ein wenig „ausfranst“, was einen optisch ansprechenderen Eindruck bietet.

Auch obwohl die Feuersimulation zu den Nebenprodukten zählt sind dennoch Bilder davon im Ergebnisse-Teil im Abschnitt 4.2 abgebildet.

Literatur

[BYM05] BELL, Nathan ; YU, Yizhou ; MUCHA, Peter J.: Particle-Based Simulation of Granular Materials. In: *ACM SIGGRAPH Symposium on Computer Animation 2005* (2005)

[Mül04] MÜLLER, Stefan: *Animation und Simulation, Dynamik von Massepunkten*. Version: 2004. http://geri.uni-koblenz.de/ws0405/ansimfolien/06_dynamik_6.pdf

[Mül05] MÜLLER, Stefan: *Computergraphik 1, OpenGL Transformationen*. Version: 2005. http://geri.uni-koblenz.de/ss05/cg1folien/09_trafo_ogl_6.pdf

[San] *Sand-Abc*. http://www.sand-abc.de/sandphysik/sandphysik1.htm, Abruf: 10. Apr. 2007