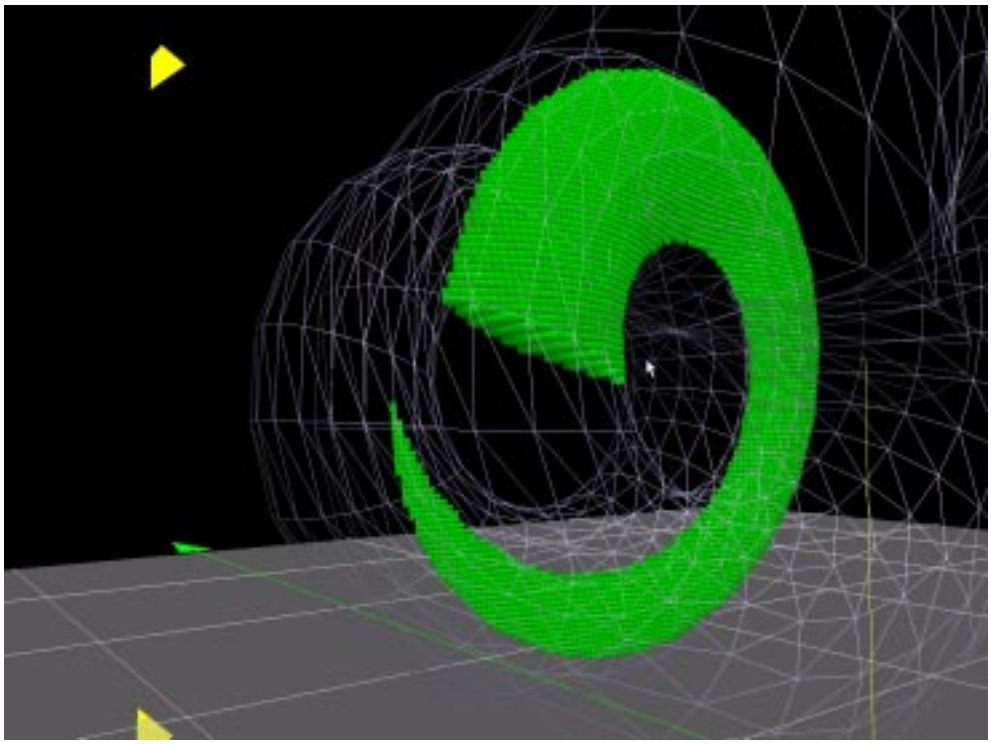
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl computer graphics lab

Detektion von Selbstkollisionen deformierbarer Objekte



Semesterarbeit
Bernhard Wymann
Sommersemester 2003

Betreuer: Bruno Heidelberger

Prof. Dr. Markus Gross
Computer Graphics Lab
ETH Zurich

Zusammenfassung

Diese Semesterarbeit präsentiert einen effizienten Algorithmus für die Selbstkollisionserkennung deformierbarer Objekte. Das auf Selbstkollisionen zu überprüfende Objekt muss geschlossen sein (wasserdicht) und alle Flächennormalen müssen konsistent gegen aussen zeigen. Der Algorithmus liefert als Ergebnis eine diskretisierte Repräsentation des Schnittvolumens der Selbstkollision. Die Auflösung der Repräsentation beeinflusst die Qualität des Schnittvolumens und ist vom Benutzer frei wählbar.

Der Algorithmus benötigt keine aufwändigen Datenstrukturen, wie zum Beispiel hierarchische Bounding Boxen oder andere Datenstrukturen zur Raumunterteilung. Dies ist wichtig, weil diese Datenstrukturen für ein deformierbares Objekt fortlaufend aktualisiert oder neu aufgebaut werden müssten.

Der Algorithmus basiert auf sogenannten Layered Depth Images. Es werden zwei Implementationen diskutiert, eine die von moderner Graphikhardware beschleunigt wird, und eine die vollständig auf der CPU berechnet wird.

Abschliessend werden die Ergebnisse der Performancemessungen präsentiert und der Algorithmus diskutiert.

Abstract

This semester thesis presents an efficient algorithm for selfcollission detection of deformable objects. The object must have a closed surface (watertight) and all face normals have to point outside. The algorithm returns a discrete representation of the selfintersection volume. The resolution of the representation affects the quality of the intersecion volume and can be choosen by the user.

The algorithm does not require datastructures like hierarchical bounding boxes or other spatial data structures. This is important because those datastructures must become updated all the time for deformable objects.

The algorithm is based on Layered Depth Images. This thesis discusses two implementations. The first implementation works only on the host processor, the second makes use of modern graphics hardware.

The thesis shows finally the results of the performance measurements and the analysis of the algorithm.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl computer graphics lab

Semesterarbeit für Herrn Bernhard Wymann

Detektion von Selbstkollisionen deformierbarer Objekte

Einleitung

Algorithmen zur Kollisionsdetektion sind üblicherweise in der Lage, Kollisionen zwischen verschiedenen Objekten zu erkennen. Bei deformierbaren Objekten existiert das zusätzliche Problem, dass ein Objekt mit sich selbst kollidieren kann. Dies ist insbesondere bei der Simulation deformierbarer Kleidungsstücke oder bei medizinischen Anwendungen der Fall.

Ziel

Aufbauend auf einem neuen Verfahren zur Kollisionsdetektion zwischen verschiedenen deformierbaren Objekten soll eine Erweiterung entwickelt werden, die das Erkennen von Selbstkollisionen ermöglicht. Es sollen zwei Varianten des Verfahrens implementiert werden, wobei eine Variante durch aktuelle Grafikhardware beschleunigt wird.

Aufgabenstellung

Folgende Aufgaben sollen bearbeitet werden:

- Einarbeitung in das Gebiet der (Selbst-)Kollisionsdetektion und des Software-Rendering, sowie in die bestehende Testumgebung.
- Entwurf eines geeigneten Verfahrens zur Erkennung von Selbstkollisionen.
- Implementierung der beiden Varianten des Verfahrens und Integration in die bestehende Testumgebung.
- Entwicklung von verschiedenen Testszenarien zur Überprüfung der Funktionalität und Robustheit des Verfahrens.
- Optimierung des Software-Renderers auf hohe Performance, speziell ausgerichtet auf das neu entwickelte Verfahren zur Kollisiondetektion.

Bemerkungen

Ein schriftlicher Bericht und eine mündliche Präsentation schliessen die Arbeit ab. Die Semesterarbeit wird von Bruno Heidelberger und Dr.-Ing. Matthias Teschner betreut.

Ausgabe: 24.3.2003
Abgabe: 2.5.2003

1

Einleitung

1.1 Motivation

Algorithmen zur Kollisionsdetektion sind üblicherweise in der Lage, Kollisionen zwischen verschiedenen Objekten zu erkennen. Bei deformierbaren Objekten existiert das zusätzliche Problem, dass ein Objekt mit sich selber kollidieren kann. Dies ist insbesondere bei der Simulation deformierbarer Kleidungsstücke oder bei medizinischen Anwendungen der Fall.

1.2 Ziel

Das Ziel der Semesterarbeit war, aufbauend auf einem Verfahren zur Kollisionsdetektion zwischen verschiedenen deformierbaren Objekten, eine Erweiterung zu entwickeln, die das Erkennen von Selbstkollisionen ermöglicht. Das Verfahren sollte in zwei Varianten implementiert werden, wobei eine Variante mit Grafikhardware unterstützt und die Andere vollständig auf dem Prozessor berechnet wird. Für die zweite Variante stand ein Softwarerenderer zur Verfügung, der für die Selbstkollisionserkennung weiter auf Performance optimiert werden sollte. Die Implementation des Verfahrens sollte schliesslich in eine schon vorhandene Testumgebung integriert werden.

1.3 Übersicht

In Kapitel zwei werden einige Grundlagen erläutert. Es wird erklärt was Layered Depth Images sind und weitere wichtige Begriffe werden eingeführt. In Kapitel drei wird die Methode zur Detektion von Selbstkollisionen mit Layered Depth Images präsentiert. Im vierten Kapitel werden die konkreten Implementationen diskutiert. Das fünfte Kapitel widmet sich der Präsentation der Resultate. Im letzten Kapitel wird die Arbeit, mögliche Erweiterungen, Probleme, sowie mein persönlicher Eindruck der Arbeit präsentiert.

2

Grundlagen

In diesem Kapitel werden kurz die Grundlagen wie der Tiefen-Puffer, Layered Depth Images (LDI), Culling und der Stencil-Puffer besprochen und auf weitergehende Informationsquellen verwiesen.

2.1 Layered Depth Images

Wenn ein dreidimensionales Objekt mit einem üblichen Renderer auf einem zweidimensionalen Bildschirm (x und y Koordinaten) visualisiert wird, muss für jeden Pixel auch die Entfernung zur Bildebene berechnet werden (die z-Koordinate). Dies ist notwendig, um festzustellen, ob ein neu berechneter Pixel sichtbar ist oder von einem bereits gezeichneten Pixel verdeckt wird. Dazu wird ein sogenannter Tiefen-Puffer (z-Buffer, Depth-Buffer) verwendet, der pro Pixel einen Tiefenwert (z-Koordinate) speichern kann.

Um das Objekt zu renderen wird zuerst der Tiefen-Puffer mit Distanz unendlich initalisiert. Nun wird das Objekt gerendert, dabei werden Pixel mit x, y und z Koordinaten berechnet. Der Pixel (x, y, z) wird nur dann gezeichnet, wenn der z-Wert des Pixels kleiner ist als der korrespondierende z-Wert im Tiefen-Puffer (dieser Vergleich wird z-Test genannt). Beim Zeichnen des Pixels wird der z-Wert des Pixels in den Tiefen-Buffer gespeichert und somit der alte z-Wert überschrieben. Zu erwähnen ist noch die Möglichkeit, einen anderen z-Test zu verwenden, so dass zum Beispiel der z-Wert immer überschrieben wird. OpenGL bietet verschiedene z-Tests an, weitergehende Informationen findet man in [2].

Die Layered Depth Image Datenstruktur (LDI) speichert nun die ganze Tiefeninformation die pro Pixel anfällt, nicht nur den kleinsten Wert wie der Tiefen-Puffer. Wenn zum Beispiel die Pixel (x, y ,z_1), (x, y, z_2) und (x, y, z_3) gerendert werden, speichert die LDI-Datenstruktur an Position (x, y) alle aufgetretenen z-Werte (z_1, z_2, z_3) ab. Zusammengefasst ist ein LDI also eine Datenstruktur mit einer diskreten (x, y) Auflösung, die pro (x, y) Koordinate alle gerenderten z-Werte abspeichert.

2.2 Front- und Backfaces

Unter Frontface versteht man der Kamera zugewandte Flächen, unter Backface der Kamera abgewandte Flächen. Da wir nur Dreiecke verwenden, ist die Normale jeder Fläche genau defi-

niert. Eine Fläche ist der Kamera zugewandt, wenn die Flächennormale, projiziert auf einen Strahl der von der Kamera zur Wurzel der Normalen führt, in Richtung der Kamera zeigt.

2.3 Culling

In diesem Kontext heisst Culling verwerfen. Wenn wir also von Backface-Culling sprechen bedeutet das, dass die Backfaces sobald sie als solche erkannt wurden, verworfen und somit nicht gerendert werden.

2.4 Stencil-Puffer

Dieser Abschnitt bezieht sich auf den Stencil-Puffer von OpenGL. Dieser Puffer ist in der Regel acht Bit tief. Manipuliert wird er durch das Rendern von üblichen Graphikprimitiven und die Konfiguration mittels *glStencilOp*. Basierend auf dem Inhalt des Stencil-Puffers kann dann das Rendering des Color- und Tiefen-Puffers beeinflusst werden. Was genau passieren soll, wird mit *glStencilFunc* konfiguriert. Detaillierte informationen zum OpenGL Stencil-Puffer finden sich in [2].

3

Methode

Dieses Kapitel beschreibt die Funktionsweise des Verfahrens zur Selbstkollisionserkennung. Zuerst wird eine Übersicht des Algorithmus präsentiert, danach wird genauer auf die einzelnen Schritte eingegangen.

3.1 Übersicht

3.1.1 Eingabe

Gegeben ist ein abgeschlossenes dreidimensionales Objekt als Dreiecksmesh. Zusätzlich ist es notwendig, dass alle Normalen der Dreiecke konsistent gegen aussen zeigen. Schliesslich wird noch eine Methode zur Generierung der Layered Depth Images benötigt.

3.1.2 Ausgabe

Der Algorithmus berechnet eine LDI-Datenstruktur, die das Schnittvolumen der Selbstkollision repräsentiert.

3.1.3 Algorithmus

Der verwendete Algorithmus unterteilt sich in drei Schritte, siehe auch Abb. 3.1.

Im ersten Schritt wird die Bounding Box des Objekts neu berechnet. Dies ist notwendig, da das Objekt deformierbar ist und sich die Grösse und Form verändern. Da wir noch nicht wissen, ob und wo das Objekt mit sich selbst kollidiert, muss die Bounding Box das ganze Objekt beinhalten (es reicht nicht, nur einen Ausschnitt zu betrachten). Konkret werden an dem Welt-Koordinatensystem ausgerichtete Bounding Boxen (AABB, Axis-Aligned Bounding Box) verwendet, da diese besonders schnell berechnet werden können.

Der zweite Schritt generiert nun zwei Layered Depth Images ausgehend von einer ausgewählten Seite der Bounding Box. Es wird je ein LDI für die Frontfaces (Front-LDI) und für die Backfaces (Back-LDI) generiert. Wenn wir einem Strahl ausgehend von der ausgewählten Seite in Richtung des Objektes folgen, stellen wir fest, dass alle Eintrittspunkte in ein Volumen auf Frontfaces liegen und alle Austrittspunkte auf Backfaces. Eine Selbstkollision ist genau dann vorhanden, wenn die Anzahl der dem Strahl entlang passierten Frontfaces minus die Anzahl passierter Backfaces grösser als eins ist. Man kann sich das als Volumenzähler vorstellen, der beim Eindringen in ein Volumen um eins inkrementiert und beim Verlassen eines Volumens

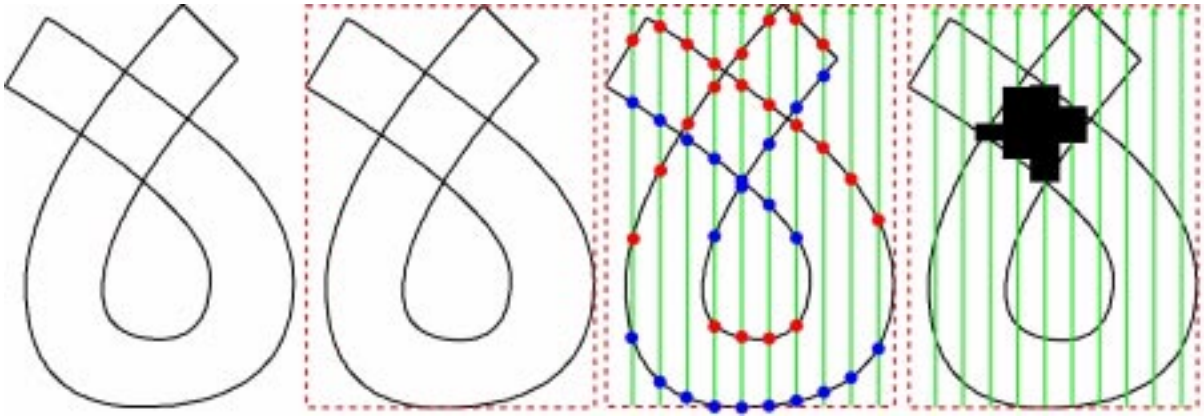


Abbildung 3.1: Von links nach rechts: Das Objekt, Objekt mit Bounding Box, Objekt mit markierten LDI z-Werten (blaue Punkte in Front-LDI, rote Punkte in Back-LDI), Objekt mit eingezeichnetem Selbstkollisions-LDI (schwarz).

um eins dekrementiert wird. Wenn wir uns an einem Ort befinden, wo mehr als ein Volumen liegt, dann muss eine Selbstkollision vorliegen (siehe Kapitel 3.4).

Im letzten Schritt wird aus den beiden Layered Depth Images ein neues Layered Depth Image berechnet, welches die Selbstkollisions-Schnittvolumen repräsentiert

3.2 Berechnung der Bounding Box

Wir benötigen eine Bounding Box, um die begrenzte (x, y) Auflösung der Layered Depth Images optimal auszunutzen. Wir verwenden Axis-Aligned Bounding Boxes. AABB sind am Welt-Koordinatensystem ausgerichtet, somit entfällt die Berechnung der Hauptachsen des zu untersuchenden Objektes. Für die Berechnung der AABB müssen alle Knoten des Objektes genau einmal untersucht werden, um die maximalen und minimalen (x, y, z) Koordinaten zu finden. Da das Objekt in einer Simulation mit diskreten Zeitschritten deformiert wird, muss die Bounding Box genau einmal pro Zeitschritt neu berechnet werden.

3.3 Generierung der Layered Depth Images

Die hier vorgestellte Methode ist im wesentlichen identisch zu der Methode gezeigt in [1]. Der Vollständigkeit halber wird sie hier trotzdem präsentiert.

Bevor die Layered Depth Images berechnet werden, wird eine Seite der Bounding Box als Bildebene ausgewählt. Wir wählen eine der Seiten, die von den beiden kürzeren Kanten aufgespannt wird, somit hat die längste Kante die höchste Auflösung, nämlich die, die der Tiefen-Puffer liefert.

Nun können die Darstellungs-Parameter für das Rendern der Layered Depth Images bestimmt werden. Wir wählen orthographische Projektion und initialisieren das View-Frustum so, dass es der Bounding Box entspricht, wobei die oben erwähnte Bildebene als Near-Plane verwendet wird. Durch diese Parametrisierung wird sichergestellt, dass im LDI an der Position (x, y) alle z-Werte gespeichert werden, die vom zur Bildebene senkrecht an Position (x, y) stehenden Strahl geschnitten werden. Die Genauigkeit des berechneten Selbstkollisions-Schnittvolumens hängt von der (x, y) Auflösung der Layered Depth Images ab, welche vom Benutzer frei gewählt werden kann.

Der weitere Verlauf ist abhängig davon, ob wir den Hardware- oder Software-Renderer verwenden. Zuerst wird die Methode mit dem Hardware-Renderer erläutert.

Der Hardware-Renderer berechnet den Front- und Back-LDI separat. Für die Generierung des Front-LDI wird Backface-Culling eingeschaltet, so dass nur die z-Werte der Frontfaces generiert werden, für den Back-LDI wird analog dazu Frontface-Culling eingeschaltet. Wenn das Objekt die Tiefenkomplexität n_{max} besitzt, so haben die beiden Layered Depth Images Tiefenkomplexität $n_{max}/2$, da bei geschlossenen Körpern pro Frontface auch ein schliessendes Backface existieren muss (siehe Kapitel 3.4). Der erste Rendering-Durchgang des Front-LDI generiert nun den ersten LDI-Layer und berechnet die Tiefenkomplexität $n_{max}/2$. Der Tiefen-Test ist dabei ausgeschaltet und Backface-Culling eingeschaltet. Zusätzlich ist der Stencil-Test eingeschaltet. Der Stencil-Test ist so konfiguriert, dass das erste Fragment pro Pixel gezeichnet wird und dabei der dazugehörige Stencil-Wert inkrementiert wird. Wenn weitere Fragmente auf diesen Pixel fallen, so werden diese vom Stencil-Test verworfen und der Stencil-Wert inkrementiert. Nach dem ersten Rendering-Durchgang enthält der Tiefen-Puffer den ersten Layer der z-Werte und der Stencil-Puffer die Tiefenkomplexität von jedem Pixel. Nun kann die Tiefenkomplexität $n_{max}/2$ einfach gefunden werden, denn sie entspricht dem maximalen Wert im Stencil-Buffer.

Wenn $n_{max}/2 > 1$ ist, sind weitere Rendering-Durchgänge nötig, um die z-Werte für die Layer 2 bis $n_{max}/2$ zu generieren. Für den i-ten Rendering-Durchgang wird der Stencil-Test so konfiguriert, dass die ersten i Fragmente den Test bestehen und somit gezeichnet werden. Da der Tiefen-Test ausgeschaltet ist, enthält der Tiefen-Puffer nach dem i-ten Rendering Durchgang den i-ten LDI-Layer. Der Back-LDI wird mit der gleichen Methode generiert, der einzige Unterschied ist die Verwendung von Frontface-Culling anstatt von Backface-Culling.

Da die Oberfläche des Objektes durch Dreiecke beschrieben wird, können die zu rendernden Fragmente in beliebiger Reihenfolge auftreten. Der Stencil-Test pickt sich für den i-ten LDI einfach das i-te gerenderte Fragment, unabhängig vom effektiven z-Wert. Deshalb müssen nach dem Generieren der LDI-Layer die z-Werte pro Pixel zusätzlich noch sortiert werden, siehe Abb. 3.2. Jeder LDI-Layer wird als zweidimensionales Array abgespeichert. Der Front- und Back-LDI besitzen jeweils $n_{max}/2$ solche Layer. Da nicht alle Pixel die volle Tiefenkomplexität besitzen müssen, werden nur die gültigen z-Werte entsprechend der Tiefenkomplexität sortiert.

Die Generierung der Layered Depth Images mit dem Software-Renderer ist anders implementiert. Die vorhandene Information im Software-Renderer ermöglicht es, den Front- und Back-LDI zusammen in einem Durchgang zu generieren. Wenn der Pixel (x, y, z) gerendert wird, weiss der Software-Renderer, ob er zu einem Front- oder Backface gehört, und kann den z-Wert in den entsprechenden LDI einfügen. Diese Operation beinhaltet das Inkrementieren der Tiefenkomplexität für den Pixel (x, y) und das Abspeichern des z-Wertes in den entsprechenden Layer. Auch bei dieser Variante müssen nach dem Generieren der Layer im Front- und Back-LDI die z-Werte pro Pixel sortiert werden.

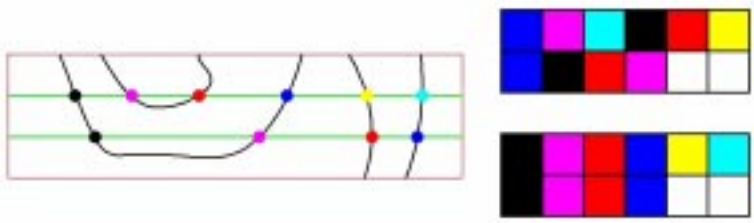


Abbildung 3.2: Die z-Werte werden in einer beliebigen Reihenfolge in die LDI's gerendert (oben rechts) und danach sortiert (unten rechts).

Zu erwähnen ist noch ein Spezialfall (siehe Abb. 3.3), nämlich wenn bei der Generierung des Layered Depth Images genau ein Knoten oder eine Kante getroffen wird. Da der Knoten oder die Kante zu allen angrenzenden Dreiecken gehört, wird der Knoten oder die Kante zweimal gerendert, also je einmal für das Frontface und das Backface. Somit sind die Layered Depth Images konsistent.

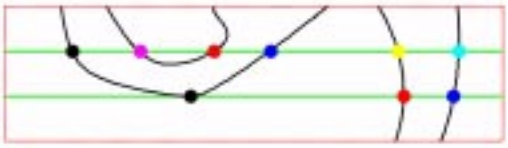


Abbildung 3.3: Der untere Strahl trifft direkt auf einen Knoten (schwarz).

3.4 Kollisionserkennung

Um Kollisionen zu erkennen, werden nun der Front- und Back-LDI zu einem neuen LDI kombiniert, welcher das Selbstkollisionsvolumen repräsentiert. Dieser LDI enthält pro Pixel (x, y) alle Paare von z-Werten, die die Selbstkollision beschreiben. Enthält beispielsweise der Pixel (x, y) die z-Werte $z_1 < z_2 < z_3 < z_4$, bedeutet das eine Selbstkollision im Bereich z_1 bis z_2, und z_3 bis z_4.

Die Kombination zum Selbstkollisions-LDI funktioniert mittels eines Raumbelegungszählers. Dieser definiert an jeder Position des Raumes (x, y, z) wieviele Volumen sich dort überlappen. Da wir nur ein Objekt untersuchen, ist eine Selbstkollision überall dort im Raum vorhanden, wo der Raumbelegungszähler grösser als eins ist. Dieser Zähler kann nun entlang von einem Strahl, ausgehend von der Near-Plane, durch Inkrementieren beim Passieren eines Frontfaces und Dekrementieren beim Passieren eines Backfaces berechnet werden.

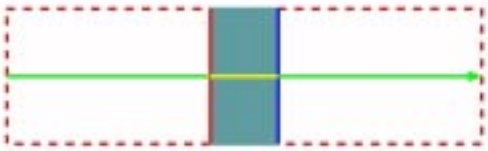


Abbildung 3.4: Strahl durch ein Objekt ohne Selbstkollision.

Abb. 3.4 zeigt einen Strahl durch ein einfaches Objekt. Die Near-Plane ist links, und wir folgen dem Strahl von links nach rechts. Zuerst ist der Raumbelegungszähler null, da wir uns ausserhalb des Objektes befinden. Wenn wir das Fronface (rot) passieren, dringen wir mit dem Strahl in den Körper ein und erhöhen den Raumbelegungszähler um eins, wir befinden uns nun in einem Volumen. Wir folgen dem Strahl weiter und verlassen das Volumen über das Backface (blau). Hier wird der Raumbelegungszähler um eins erniedrigt. Wir stellen fest, dass für ein einfaches Objekt ohne Selbstkollisionen, welches den oben genannten Voraussetzungen genügt, der Raumbelegungszähler so bestimmt werden kann.

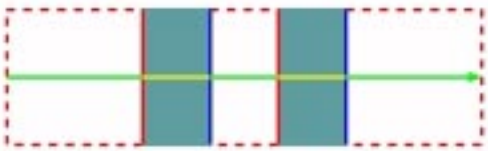


Abbildung 3.5: Strahl durch ein Objekt welches zweimal geschnitten wird.

Abb. 3.5 zeigt einen Strahl durch ein Objekt das mehrfach vom Strahl geschnitten wird. Wenn wir wie oben dem Strahl folgen, stellen wir fest, das der Raumbelegungszähler immer noch korrekt berechnet wird.

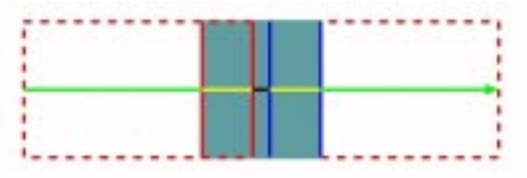


Abbildung 3.6: Ausschnitt aus einem Objekt mit einfacher Selbstkollision.

Abb. 3.6 zeigt einen Strahl durch ein Objekt mit einer Selbstkollision. Entlang dem Strahl wird nun der Raumbelegungszähler beim ersten Frontface inkrementiert, beim Passieren des zweiten Frontfaces noch einmal. Da der Raumbelegunszähler nun gleich zwei ist (grösser als eins), bedeutet das, dass wir eine Selbstkollision gefunden haben. Beim Passieren des Backfaces wird der Zähler wieder dekrementiert. Diese einfache Selbstkollision (Raumbelegungszähler gleich zwei) kann mit dieser Methode immer bestimmt werden, egal wie kompliziert das Objekt und die einfache Selbstkollision auch sein mag, weil wir immer einen Strahl durch das Objekt schicken können. Entlang von diesem Strahl sind dann nur noch die beobachteten z-Werte interessant. Eine weitere Beobachtung ist, dass das Selbstkollisionsvolumen einer einfachen Selbstkollision für sich selber betrachtet wieder ein Objekt ist, dass den oben genannten Anforderungen gerecht wird.

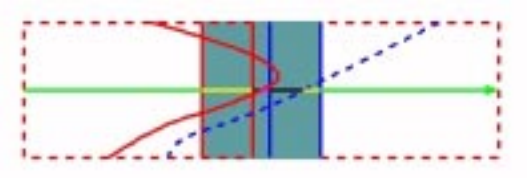


Abbildung 3.7: Mehrfache Selbstkollision durch hinzufügen eines beliebigen Frontfaces.

Abb. 3.7 illustriert, was bei einer beliebigen vorhandenen Konfiguration passiert, wenn wir das Objekt so deformieren, dass ein zusätzliches Frontface von dem Strahl geschnitten wird. Wenn das Objekt den Anforderungen entspricht, ist es zwingend notwendig, das zu diesem Frontface auch ein Backface gehört. Wir können wieder dem Strahl folgen und stellen fest, dass der Raumbelegungszähler korrekt funktioniert.

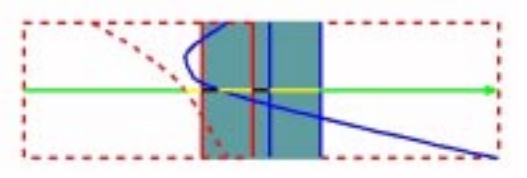


Abbildung 3.8: Mehrfache Selbstkollision durch hinzufügen eines beliebigen Backfaces.

Abb. 3.8 illustriert das Einfügen eines beliebigen Backfaces durch die Deformation des Objektes. Wenn das Objekt den Anforderungen genügt, muss es zu diesem Backface auch ein korrespondierendes Frontface geben, der Raumbelegunszähler bleibt konsistent.

3.5 Generierung des Selbstkollisions-LDI

Die Generierung des Selbstkollisions-LDI umfasst die folgenden Schritte. Betrachtet wird nur ein Pixel (x, y), da das Verfahren für jeden Pixel gleich und unabhängig von den anderen Pixeln ist. Der Einfachheit halber stellen wir uns die z-Werte des Front- und Back-LDI für den Pixel (x, y) vereinigt in einer sortierten Liste aufgereiht vor, wobei wir immer noch wissen, welche z-Werte von welchem LDI stammen. Diese Liste ist nun exakt eine Darstellung der Schnittpunkte der vorher verwendeten Strahlen. In den Selbstkollisions-LDI Pixel (x,y) werden nun die z-Werte eingefügt, bei denen der Raumbelegunszähler von eins auf zwei erhöht wird, und die z-Werte bei denen der Raumbelegunszähler von zwei auf eins fällt.

Die vorher erwähnte Liste ist nur ein Denkmodell, sie wird zu keinem Zeitpunkt explizit generiert. Mit dem Front- und Back-LDI kann unter Zuhilfenahme zweier Indizes und einer Vergleichsoperation der selbe Effekt erzielt werden.

4

Implementation

Dieses Kapitel diskutiert zuerst die Implementationen zur Generierung der Layered Depth Images. Danach wird auf die Implementation der Kollisionsdetektion eingegangen.

4.1 LDI Generierung mit Hardware-Unterstützung

Dieser Abschnitt beschreibt die Implementation in OpenGL. Die Layered Depth Images werden in mehreren Rendering-Durchgängen generiert. Pro Durchgang wird ein Layer des LDI generiert. Diese Methode ist im wesentlichen identisch zu der Methode präsentiert in [1].

Wenn nur das Selbstkollisionsvolumen von Interesse ist, kann das Rendern des Color-Puffers ausgeschaltet werden. Wie in Kapitel 3.3 erwähnt wollen wir, wenn immer ein Fragment gezeichnet wird, dessen z-Wert im Tiefen-Puffer abspeichern. Es müssen auch die Modelview- und die Projektions-Matrizen basierend auf den Daten der Bounding Box und der gewählten Near-Plane initialisiert werden. Zusätzlich müssen wir noch Backface- oder Frontface-Culling einschalten, abhängig davon, ob wir den Front- oder Back-LDI generieren wollen. Der Pseudocode sieht folgendermassen aus:

```
// Rendering Setup
glEnable(GL_STENCIL_TEST);
glEnable(GL_CULL_FACE);
if (renderFrontLDI) {
      glCullFace(GL_BACK);
} else {
      glCullFace(GL_FRONT);
}
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_ALWAYS);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
Setzen der Projektions- und Modelview-Matrizen

// Der erste Rendering-Durchgang berechnet die maximale
// Tiefenkomplexititität und den ersten Layer.
glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_GREATER, 1, 0xff);
glStencilOp(GL_INCR, GL_INCR, GL_INCR);
Rendern des Objektes;
```

```
Tiefenkomplexitäten := Stencil_Puffer;
n := MAX(Tiefenkomplexitäten);
layer[1] := Tiefen_Puffer;
// In den folgenden Durchgängen ist die Tiefenkomplexität schon
// bekannt und muss somit nicht mehr neu berechnet werden.
i := 2;
while (i <= n) {
      glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
      glStencilFunc(GL_GREATER, n, 0xff);
      glStencilOp(GL_KEEP, GL_INCR, GL_INCR);
      Rendern des Objektes;
      layer[i] := Tiefen_Puffer;
      i := i + 1;
}
```

Wie man sieht, muss das gesamte Objekt in jedem Rendering-Durchgang (ein Durchgang pro Layer) neu gerendert werden und der Tiefen-Puffer vom Speicher der Graphikkarte in den normalen Hauptspeicher zurückkopiert werden. Der Stencil-Puffer kann gemeinsam mit dem Tiefen-Puffer zurückkopiert werden, da üblicherweise die z- und Stencil-Werte paarweise in einem 32 Bit Wort liegen (24 Bit z- und 8 Bit Stencil-Wert). Dieser Kopiervorgang ist ein wesentlicher Faktor für die Performance des Verfahrens.

Deshalb wurde von den Autoren von [1] eine Implementation getestet, die zuerst alle (ausser dem ersten) Layer in den Speicher der Graphikkarte rendert und danach alle Layer gemeinsam kopiert. Der Geschwindigkeitszuwachs war jedoch nicht wesentlich. Eine weitere Variante zur Steigerung der Performance bietet auch die Verwendung eines zweiten Tiefen-Puffers, der heute auf vielen Graphikkarten vorhanden ist zur Unterstützung von Shadow-Mapping. Somit können zwei Layer in einem Durchgang generiert werden. Das wurde auch von den Autoren von [1] getestet und wegen zu geringer Performance wieder verworfen.

Damit die Geometrie des Objektes nicht für jeden Layer erneut in die Graphikkarte übertragen werden muss, könnte man Display-Listen verwenden. Die Generierung dieser Display-Listen ist jedoch zu langsam. Da das Objekt deformierbar ist, muss die Display-Liste in jedem Zeitschritt neu erstellt werden. Eine weitere Möglichkeit die Geometrie nur einmal pro Zeitschritt zu übertragen wäre die Verwendung der ARB_vertex_buffer_object Erweiterung, die es zum Beispiel ermöglicht ein Vertex Array in den Graphikspeicher zu Laden. Dies wurde noch nicht getestet.

4.2 LDI Generierung in Software

Für die Generierung der Layered Depth Images stand ein Software-Renderer zur Verfügung. Dieser wurde leicht modifiziert, um die Generierung des Front- und Back-LDI in einem Durchgang zu ermöglichen. Dazu musste die Information, ob ein Front- oder Backface abgearbeitet wird, bis zur Methode, die den z-Wert in den LDI einfügt, heruntergereicht werden. Dies ist nötig, damit der z-Wert in den richtigen LDI eingefügt werden kann.

Mit Front- und Backface-Culling wären zwei Rendering-Durchgänge zur Generierung des Front- und Back-LDI nötig, mit dieser Verbesserung reicht nun ein einziger Durchgang. Weiterhin wurde der Softwarerenderer daraufhin untersucht, ob noch mehr Verbesserungen möglich wären. Leider wurden keine Verbesserungsmöglichkeiten gefunden, die keinen Einfluss auf die Flexibilität des Renderers gehabt hätten. Somit wurden keine weiteren Änderungen durchgeführt. Der Code wurde auf unnötige Kopiervorgänge, unnötige Indirektion, Cachefeindlichkeit (ungeschickt angeordnete Daten, die viele Cache Misses verursachen) und ungün-

stige Ausdrücke untersucht (zum Beispiel a/2 +b/2, anstatt (a+b)/2). Nicht untersucht wurde die Berechnung von konstanten Ausdrücken in Schleifen und die mehrfache Berechnung von gleichen Teilausdrücken, da die Compiler diese erkennen und selber Optimieren können.

4.3 Kollisionsdetektion

Dieser Abschnitt beschreibt die Implementation der Selbstkollisionserkennung in Pseudocode. Die Eingabe besteht aus dem Front- und dem Back-LDI (beide sortiert), die Ausgabe ist ein LDI, der das Selbstkollisionsvolumen repräsentiert. Betrachtet wird nur die Generierung von einem Selbstkollisions-LDI Pixel (x, y), da die Verarbeitung unabhängig von den anderen Pixeln ist. Weiter unten in diesem Abschitt ist ein Pseudocode-Listing enthalten.

Die Variable incount enthält den vorher definierten Raumbelegungszähler, zfront und zback enthalten die aktuellen z-Werte des Front- und Back-LDI, mit denen gerade gearbeitet wird. Der Algorithmus arbeitet maximal solange, bis alle Werte vom Back-LDI verarbeitet sind. Die Idee dabei ist, dass der letzte z-Wert, den wir sehen, zu einem Backface gehören muss, da das Objekt laut den Voraussetzungen geschlossen ist. Dadurch terminiert der Algorithmus garantiert, auch wenn die Anforderungen für das Objekt nicht erfüllt sind.

Wenn zfront kleiner als zback ist, müssen wir zusätzlich testen, ob noch z-Werte im Front-LDI vorhanden sind (wir zählen auch den aktuellen Wert zu dieser Menge). Wenn noch z-Werte im Front-LDI vorhanden sind, bedeutet das, dass wir in ein zusätzliches Volumen über ein Frontface eindringen. Demzufolge wird der Raumbelegungszähler erhöht, und gegebenenfalls der z-Wert in den Selbstkollisions-LDI abgespeichert. Nun können wir den aktellen zfront Wert aus der Menge der zu verarbeitenden z-Werte im Front-LDI entfernen, und falls noch z-Werte vorhanden sind, den nächsten zum aktuellen zfront Wert machen.

Wenn zfront kleiner als zback ist und keine Werte mehr im Front-LDI vorhanden sind, bedeutet das, dass wir in keine zusätzlichen Volumen mehr eindringen können. Daraus folgt, dass wir bei dem aktuellen Übergang nur ein Backface passieren können. Demzufolge wird der Raumbelegungszähler dekrementiert. Gegebenenfalls wird der zback Wert in den Selbstkollisions-LDI abgespeichert und die Verarbeitung des aktuellen Pixels abgebrochen. Dies ist möglich, da nur noch ein z-Wert vom letzten Backface zur Verfügung steht, und dieser trägt nichts mehr zu einer Selbstkollision bei. Falls die Verarbeitung nicht abgebrochen wird, wird der aktuelle zback Wert aus der Menge der zu verarbeitenden z-Werte der Back-LDI entfernt. Falls noch ein z-Wert im Back-LDI vorhanden ist, wird dieser zum aktuellen zback Wert gemacht.

Wenn zfront grösser oder gleich zback ist können wir daraus schliessen, dass wir als nächstes ein Volumen verlassen und später wieder in ein Volumen eindringen werden. Deshalb muss nicht getestet werden, ob noch Front-LDI Werte zu verarbeiten sind. Der Verlauf ist analog zum oberen Abschnitt, nur mit dem Unterschied, dass wir nicht abbrechen können, da wir später wieder in mindestens ein Volumen eindringen werden und weitere Selbstkollisionen nicht auszuschliessen sind.

Hier folgt das Pseudocode-Listing des beschriebenen Algorithmus.

```
Für alle x, für alle y
incount = 0
zfront = erster z Wert von front LDI
zback = erster Wert von back LDI
      solange nicht alle back LDI Werte verarbeitet
            wenn (zfront < zback)
                  wenn noch front LDI Werte vorhanden
                        incount++
```

```
                wenn (incount == 2)
                      zfront in Kollisions-LDI abspeichern
                nächsten front LDI z Wert nach zfront falls noch
                Werte vorhanden sind
          andernfalls
                incount--
                wenn  (incount == 1)
                      zback in Kollisions LDI abspeichern
                      Abbruch
                nächsten back LDI z Wert nach zback falls noch
                Werte vorhanden sind
    wenn (zfront >= zback)
          incount--
          wenn (incount == 1)
                zback in Kollisions-LDI abspeichern
          nächsten back LDI z Wert nach zback falls noch Werte
          vorhanden sind
```

5

Resultate

In diesem Kapitel werden die Resultate der Performance-Tests präsentiert. Dazu wird zuerst das Testsystem spezifiziert und das Test-Setup beschrieben. Danach werden die Resultate analysiert.

5.1 Testsystem

Alle hier beschriebenen Experimente wurden auf einem 800MHz Pentium III mit 512 MB Hauptspeicher durchgeführt. Die verwendete Graphikkarte ist eine Nvidia GeForce 3 mit 64 MB Speicher in einem AGP 2x Slot. Das verwendete Testsystem läuft mit Linux, Xfree86 4.2.1 und dem Nvidia Treiber 1.0-4349. Alle Sourcen wurden mit gcc 2.95.3 übersetzt und mit -O2 -march=i686 optimiert.

5.2 Testobjekte und Parameter

Da es sich als schwierig erweist, bei Selbstkollisionen von einem beliebigen Objekt den Überblick zu behalten oder eine vernünftige Metrik einzuführen, werden als Testobjekte Stapel von parallelen quadratischen Ebenen verwendet, siehe Abb. 5.1. Wenn n Ebenen in der Versuchsanordnung vorhanden sind, besitzen die ersten n/2 Ebenen, die sich näher an der Near-Plane befinden, eine Normale die in Richtung der Near-Plane zeigt, und die anderen n/2 Ebenen eine Normale in entgegengesetzter Richtung. Dies entspricht einer Selbstkollision mit Raumbelegungszähler n/2-1 in der Mitte, da wir zuerst n/2-Mal in das Objekt eindringen und es dann wieder n/2-Mal verlassen. Damit dieses Objekt von der gewünschten Seite her gerendert wird, muss man die Geometrie entsprechend wählen (die längste Achse der Bounding Box steht senkrecht auf der gewählten Near-Plane).

Diese Objekte können leicht generiert werden, bei einer gegebenen Anzahl von uniform angeordneten Knoten konvergiert die Anzahl der Dreiecke pro Ebene für viele Knoten gegen 2 Mal die Anzahl der Knoten pro Ebene.

Es wurden zwei Reihen von Experimenten durchgeführt. In der ersten Versuchsreihe wurde die Anzahl der Knoten im Objekt ungefähr konstant gehalten (~10000) und die Anzahl der Ebenen variiert. In der zweiten Versuchsreihe wurde die Anzahl der Ebenen konstant gehalten (acht) und die Anzahl der Knoten variiert. Ungefähr bezieht sich dabei darauf, dass sich exakt 10000 Knoten nicht wie gewünscht auf n Ebenen verteilen lassen. Die Abweichung ist jedoch

vernachlässigbar. Die Auflösung der Layered Depth Images war bei allen Experimenten auf 128 mal 128 Pixel eingestellt.

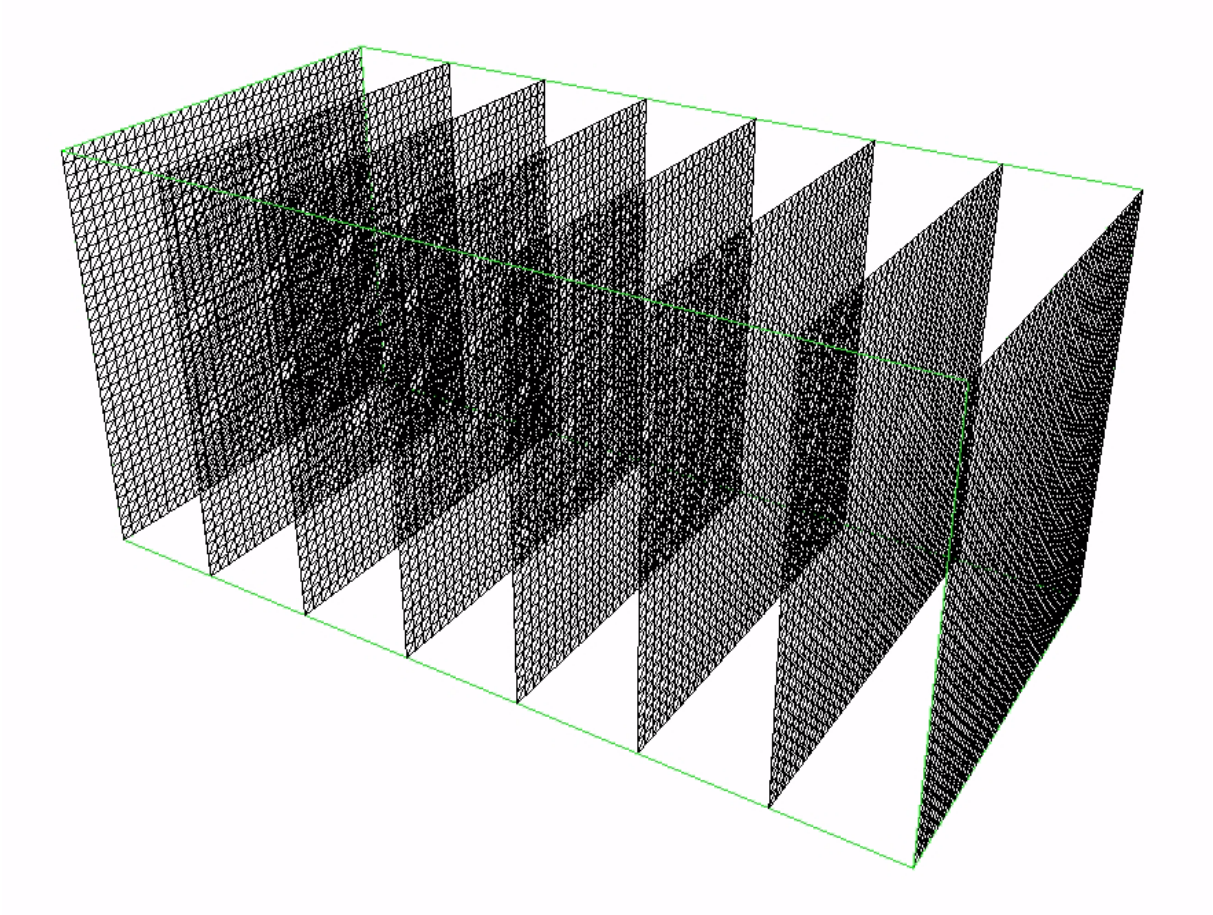


Abbildung 5.1: Testobjekt mit acht Ebenen, Selbstkollisionstiefe ist drei.

In den Experimenten wurde die benötigte Zeit zur Generierung des Front- und Back-LDI, sowie die Zeit zur Generierung des Selbstkollisions-LDI untersucht. Gemessen wurden die Zeiten unter Verwendung des Hardware-Renderers, des beschriebenen optimierten Softwarerenderers, sowie mit einem zu Vergleichszwecken erstellten Softwarerenderer, der jeweils einen Durchgang für den Front- und den Back-LDI braucht (zur Erinnerung, die optimierte Variante generiert den Front- und Back-LDI gemeinsam in einem Durchgang).

5.3 Messresultate und Analyse

Abb. 5.2 zeigt die Zeit die zum Rendern des Front- und Back-LDI benötigt wurde als Funktion der Selbstkollisionstiefe mit einer konstanten Anzahl Knoten. Die Selbstkollisionstiefe entspricht dabei dem Raumbelegungszähler. Null entspricht keiner Selbstkollision (zwei Ebenen, einmal eindringen und wieder verlassen), 1 einer einfachen Selbstkollision (vier Ebenen), und so weiter. Auf der y-Achse sind die gemessenen Zeiten in Millisekunden aufgetragen.

Als erstes stellen wir fest, dass der Softwarerenderer mit diesem Testsystem bei allen Messungen langsamer ist als der Hardwarerenderer. Da die Performance vom Softwarerenderer sehr von der CPU abhängig ist, könnte je nach verwendeter Hardware das Ergebnis auch zugunsten des Softwarerenderers ausfallen. Weiterhin ist offensichtlich, dass der optimierte Softwarerenderer mit dieser Konfiguration kaum Vorteile bietet. Wenn wir uns die Optimierung betrachten wird klar, dass diese effektiv nur dann hilft, wenn im Verhältnis viel Geometrie und wenige

Pixel gerendert werden. Wenn wir viele Pixel aus wenig Geometriedaten rendern, ist es im Verhältnis nicht teuer, sich die Geometriedaten zweimal anzusehen, zu cullen und zu rendern, da jedes Dreieck in jeder Variante genau einmal rasterisiert werden muss.

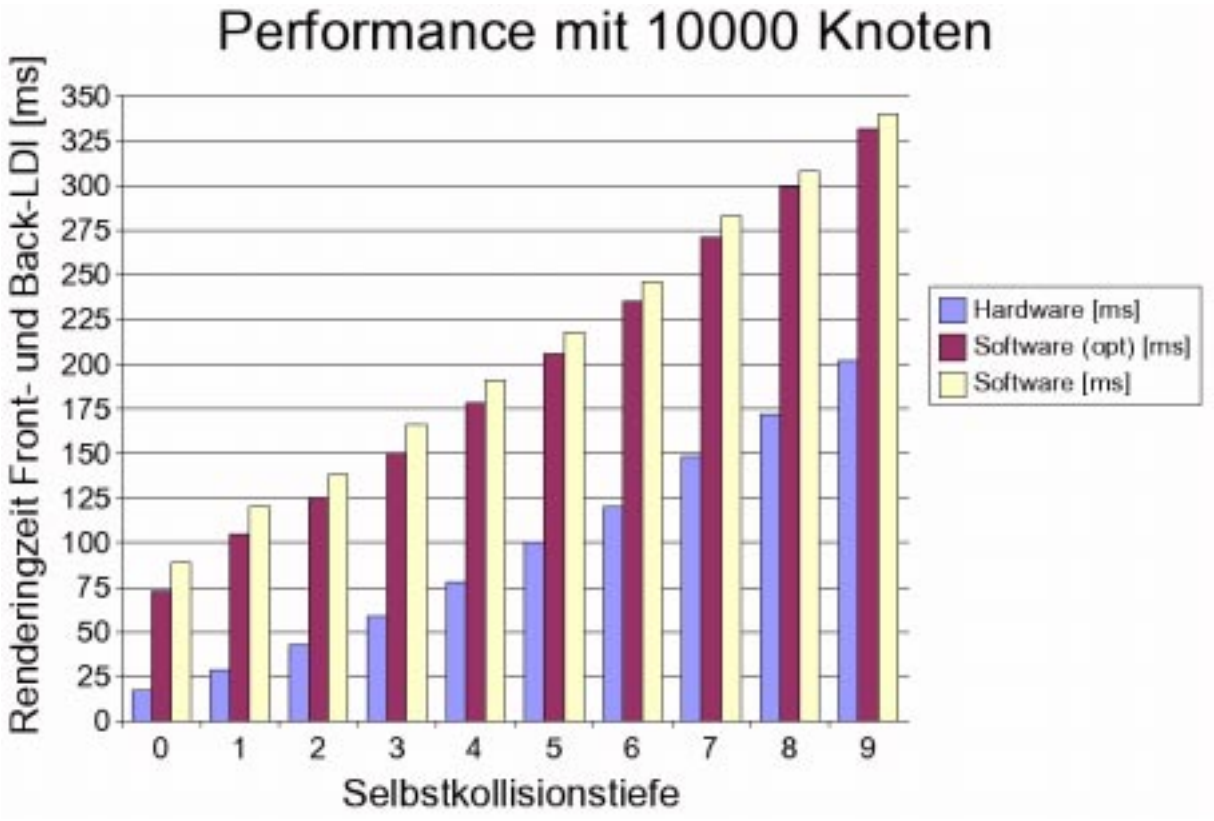


Abbildung 5.2: Renderingzeiten Front- und Back-LDI in Abhängigkeit der Selbstkollisionstiefe mit einer konstanten Anzahl Knoten.

Zusätzlich stellen wir fest, dass die Zeiten scheinbar ein wenig stärker als linear mit der Selbstkollisionstiefe wachsen. Der lineare Anteil beim Hardware-Renderer erklärt sich durch das zu transferierende Datenvolumen beim Zurücklesen der Layer in den Hauptspeicher und beim Softwarerenderer durch die linear wachsende Anzahl der zu rendernden Pixel. Der nichtlineare Anteil könnte vom Sortieren der LDI's verursacht werden. Die Messreihe deutet auch darauf hin, das es einen konstanten Anteil gibt. Dieser konnte mit dem Füllen der CPU-Caches erklärt werden (die Suche nach der maximalen Tiefe der LDI's kostet nur 128 mal 128 Operationen und ist nicht relevant).

Für die oben dargestellten Zeiten zur Generierung der LDI's wurde auch die Zeit gemessen, die benötigt wird, um aus dem Front- und Back-LDI den Selbstkollisions-LDI zu generieren. Diese Werte wachsen nahezu linear von 2.5 ms bis 23 ms, wie es zu erwarten war.

Abb. 5.3 zeigt die Zeit die zum Rendern des Front- und Back-LDI benötigt wurde als Funktion der Anzahl Knoten. Die Selbstkollisionstiefe ist dabei Konstant gleich drei (acht Ebenen).

Da die Generierung des Selbstkollisionsvolumens abhängig ist von der Anzahl der Layer und deren Auflösung, erwartet man für alle Messungen die gleiche Zeit. Das wurde durch die Messungen bestätigt (immer 8 ms).

Diese Messungen bestätigen die Vermutung, dass der optimierte Softwarerenderer vor allem dann Vorteile bringt, wenn im Verhältnis viel Geometrie und wenige Pixel zu verarbeiten sind.

Für den Softwarerenderer wächst die Zeit nun praktisch linear mit der Anzahl der Knoten. Dies ist darauf zurückzuführen, dass für jede Anzahl Knoten genau gleich viele Layer mit der gleichen Auflösung und sogar identischen z-Werten sortiert werden müssen. Daraus folgt, dass die Zeit zum Sortieren als Konstante in die Laufzeit einfliesst.

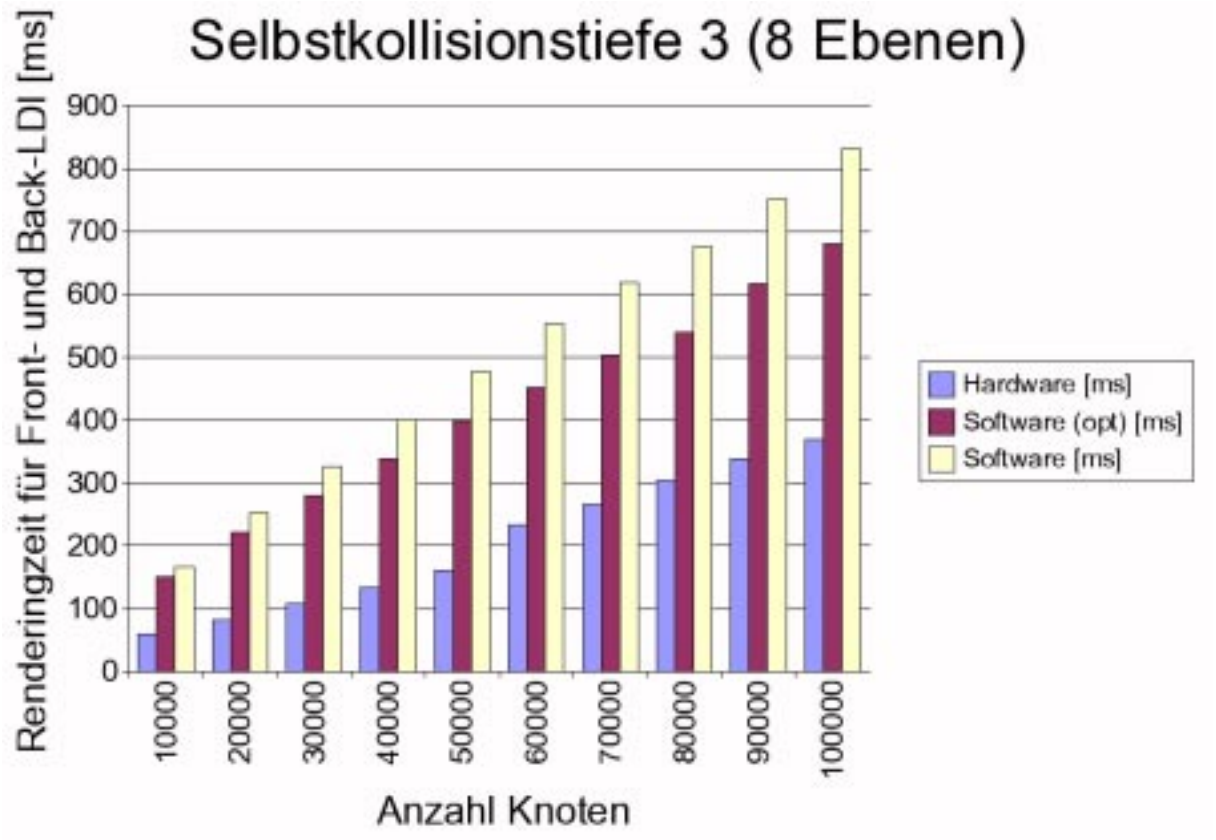


Abbildung 5.3: Renderingzeiten Front- und Back-LDI in Abhängigkeit der Anzahl Knoten mit einer konstanten Selbstkollisionstiefe.

Beim Hardware-Renderer wächst die Zeit auch linear bis 50000 Knoten, macht dann einen kleinen Sprung, und wächst danach scheinbar linear weiter. Für diesen Sprung habe ich keine Erklärung. Ich vermute, dass die Ursache dafür in der verwendeten Hardware liegt. Zum Beispiel ist es denkbar, dass gewisse Daten nicht mehr in einen Cache passen. Es wäre auch interessant diese Messung mit einer anderen Hardware-Konfiguration zu wiederholen.

5.4 Testumgebung

Der beschriebene Algorithmus wurde für diese Tests in ein bestehendes Framework integriert. Screenshots finden sich in Anhang A 5.

6

Bewertung und Ausblick

6.1 Erreichte Ziele

Das Ziel, eine Erweiterung zur effizienten Selbstkollisionserkennung zu entwickeln, basierend auf dem Kollisionsdetektionsalgorithmus für deformierbare Objekte beschrieben in [1], wurde erreicht. Wie gefordert wurden zwei Lösungen implementiert: Eine die die Vorteile moderner Graphikhardware ausnutzt, und eine, die vollständig auf der CPU ausgeführt wird. Der Algorithmus wurde wie verlangt in die bestehende Testumgebung integriert, ohne die bestehende Funktionalität zu beeinträchtigen. Es wurden verschiedene Testkörper erstellt und die nötige Funktionalität in die Testumgebung integriert, um Selbstkollisionen zu generieren und zu testen. Für die Performancemessungen wurden spezielle Objekte generiert (siehe Kapitel 5.2), um aussagekräftige Resultate zu erhalten.

Obwohl der Softwarerenderer optimiert wurde, konnte keine signifikante Performanceverbesserung erzielt werden (die gemessene Verbesserung ist mindestens 4% bis maximal 22%). Da die Implementation jedoch auf einem Lehrbuch beruht, ist das nicht wirklich erstaunlich.

Der entwickelte Algorithmus ist robust, solange die Objekte abgeschlossen (wasserdicht) sind und alle Flächennormalen konsistent gegen aussen zeigen. Wenn diese Bedingungen verletzt werden, terminiert der Algorithmus trotzdem garantiert. Das Resultat wird dann jedoch nicht die Selbstkollision repräsentieren.

6.2 Einschränkungen und mögliche Erweiterungen

6.2.1 Deformationen

Zuerst möchte ich auf das in Abb. 6.1 illustrierte Experiment eingehen. Wir nehmen ein Ellipsoid, das den Anforderungen des Algorithmus gerecht wird. Nun deformieren wir die Vorderseite in Richtung der Hinterseite, und zwar solange, bis ein Teil der Vorderseite hinter der Hinterseite liegt. Das führt dazu, dass die Normalen gerade verkehrt herum gerichtet sind.

Das Experiment soll zeigen, dass wir mit ungeschickter Simulation ausgehend von einem korrekten Objekt leicht ein Objekt generieren können, welches den Anforderungen nicht mehr genügt. Physikalisch gibt es meines Wissens kein Objekt, das tatsächlich so deformiert werden könnte. Das Experiment zeigt auch, dass selbst wenn die Normalen “automagisch” korrigiert würden, wir keine Selbstkollision erkennen, was eigentlich auch der Intuition entspricht.

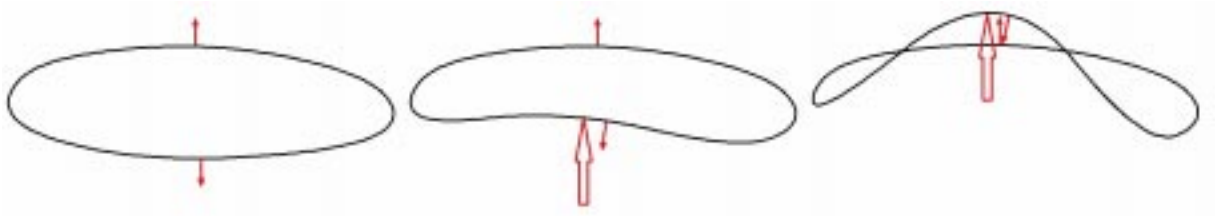


Abbildung 6.1: Deformation welche die nach aussen gerichteten Normalen umkehrt.

Mit diesem Experiment wurde gezeigt, dass gewisse Deformationen Begriffe wie innen und aussen unklar erscheinen lassen. Die pysikalische Simulation muss solche Situationen unbedingt vermeiden, da es im Nachhinein nicht mehr klar ist, wie sich das Objekt weiter deformieren soll. In der Realität würde das Objekt wohl in mehrere Objekte zerfallen oder ein Loch würde entstehen.

6.2.2 Dünne Objekte

Bei der Beschreibung des Algorithmus wurde erklärt, dass die Auflösung der Layered Depth Images vom Benutzer wählbar ist. Es ist offensichtlich, dass wenn relativ zur Auflösung der LDI's sehr dünne Objekte verwendet werden, Selbstkollisionen übersehen werden können. Das Problem ist sehr gut sichtbar in Abb. 6.2. Wenn die LDI's horizontal generiert werden, bleibt die Selbstkollision unerkannt, wenn sie vertikal generiert werden, wird der blaue Bereich erkannt.

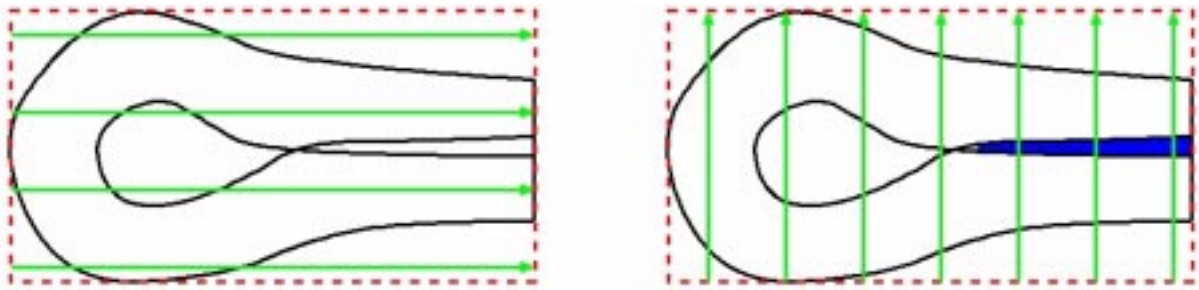


Abbildung 6.2: Ein relativ zur Auflösung der LDI's dünnes Objekt. Abhängig von der Rendering-Richtung wird die Selbstkollision nicht erkannt.

In diesem Beispiel schafft eine andere Richtung zur Generierung der LDI's Abhilfe, dies funktioniert jedoch nicht immer.

6.2.3 Selbstkollisionstiefe

Zum Testen des Algorithmus wurden vielfache Selbstkollisionen generiert, mit bis zu 10 überlappenden Volumen. Der Algorithmus funktioniert in diesen Situationen fehlerfrei, trotzdem dürfen diese Fälle meiner Meinung nach in einer Simulation nie auftreten. Der Grund dafür ist, dass eine gute Chance besteht, dass die Kräfte in einer so komplexen Situation ein stabiles System bilden, und somit die Selbstkollision nicht aufgelöst werden kann. Die typische Selbstkollision wird also zwei überlappende Volumen besitzen.

6.2.4 Software-Renderer

Der Software-Renderer könnte mit der Verwendung von SIMD Instruktionen (Single Instruction Multiple Data wie zum Beispiel SSE, 3D Now, Altivec, ...) beschleunigt werden. Dies würde sich nun besonders anbieten, da der gcc 3.3 [3] dafür eine Spracherweiterung

besitzt, die automatisch auf diese Einheiten zurückgreift, wenn diese Vorhanden sind. Der Nachteil wäre natürlich die Inkompatibilität mit andern Compilern.

6.2.5 Hardware-Renderer

Eine Verbesserung der Performance wäre vielleicht möglich unter der Verwendung der ARB_vertex_buffer_object [4] Erweiterung oder von diversen herstellerspezifischen Erweiterungen, die es ermöglichen, die Geometrie des Objektes nur einmal für alle Layer in die Graphikkarte zu transferieren. Es ist eventuell auch in naher Zukunft schon möglich, die ganze Deformation sowie die Selbstkollisionserkennung komplett auf der GPU durchzuführen.

6.2.6 Kollisionsantwort

Die Generierung des Selbstkollisionsvolumens ist nur ein erster Schritt. Das Ziel ist in einer Simulation sinnvoll auf Selbstkollisionen zu reagieren. Dazu müssen wahrscheinlich die Penetrationstiefe sowie der nächste Punkt auf der Oberfläche berechnet werden.

Falls dazu die Normalen benötigt würden, die zu den z-Werten im LDI gehören, könnten diese einfach mit folgender Methode bestimmt werden. Zusätzlich zum Tiefen-Puffer müsste auch der Color-Puffer gerendert werden. Jedes gerenderte Dreieck würde dabei so eingefärbt, das die Farbe als Index für das Dreieck verwendet werden kann. Da nun jeder gerenderte Pixel als Farbe den Index seines Dreiecks hat, kann jederzeit auf die Normale zugegriffen werden. Dies wäre natürlich nicht gratis, da der Color-Puffer gerendert und zurückgelesen werden muss.

6.3 Persönlicher Eindruck

Die Thematik der Kollisionserkennung mit Layered Depth Images finde ich sehr Interessant, besonders weil das Verfahren für mich komplett neu war. Der Fortschritt war zu Beginn etwas langsam, da ich mich mit den Sourcen der Demoumgebung und des Software-Renderers vertraut machen musste. Vor allem der Software-Renderer ist meiner Meinung nach etwas schwer zu lesen, da zum Teil schon die Dateinamen etwas irreführend sind.

Besonders interessant war die Generierung der Layered Depth Images mit dem Stencil-Puffer, der Entwurf und die Implementation des Algorithmus zur Selbstkollisionserkennung.

Etwas ernüchternd war die Analyse des Software-Renderers, da ich hier keine wesentlichen Verbesserungen anbringen konnte. Insgeheim hatte ich das eigentlich erwartet, da der Software-Renderer gemäss einem Lehrbuch implementiert wurde. Obwohl diese Analyse ziemlich aufwändig war, gibt es dazu leider nichts zu dokumentieren.

A

Anhang

A.1 Referenzen

[1] B. Heidelberger, M. Teschner, M. Gross:
Volumetric Collision Detection for Deformable Objects
Technical Report No. 395, Institute of Scientific Computing, ETH Zurich, April 2003.

[2] M. Woo, J. Neider, T. Davis, D. Shreiner:
The OpenGL Programming Guide, Third Edition, ISBN-0-201-60458-2.

[3] GNU Compiler Collection der Free Software Foundation
http://gcc.gnu.org

[4] http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt

A.2 Deformierbare Objekte für die Demoumgebung

Damit die Deformationsroutinen zur Generierung von Selbstkollisionen der Testumgebung die Testobjekte nicht zerreissen, müssen diese eine bestimmte Struktur aufweisen. Geeignete Testobjekte können mit cylinder_twist.cpp aus dem tools Verzeichnis der CD generiert werden. Dazu ist es nötig die Parameter im Quellcode oder gegebenenfalls den Code selber anzupassen und das Tool neu zu kompilieren.

A.3 Starten der Demoumgebung

Zum Starten der Demoumgebung kann im Verzeichnis collisioncontrol/demo das Script rundemo.sh verwendet werden. Es werden die in die Umgebung zu ladenden Objekte als Kommandozeilenparameter übergeben, zum Beispiel:

```
$ ./rundemo.sh ../../models/fish.off ../../models/spiral2.off
```

A.4 Tastenbelegung der Demoumgebung

Tabelle A.1: Tastenbelegungen.

Taste	Beschreibung
q oder <Esc>	Verlassen der Demoumgebung
<Space>	Anhalten der Animation
a	Kamera nach links bewegen
d	Kamera nach rechts bewegen
w	Kamera vorwärts bewegen
s	Kamera rückwärts bewegen
r	Kamera nach oben bewegen
f	Kamera nach unten bewegen
c	Kollisionserkennung ein/aus
v	Selbstkollisionserkennung ein/aus
n	Normalen berechnen ein/aus
m	Visualisierungsmodus der Modelle (unsichtbar, wireframe, solid, fancy)
b	Bounding Box ein/aus
l	LDI Generierung ein/aus
t	LDI Visualisierungsmodus (solid, layers, points)
p	Deformation (none, twist, wobble, indexed1, indexed2)
[,]	Einstellen der Amplitude
/	Softwarerenderer ein/aus
<linke Maustaste>	Objekt selektieren
<Shift> und <linke Maustaste>	Mehrere Objekte selektieren
<mittlere Maustaste>	Objekte verschieben
<rechte Maustaste>	Kameraorientierung ändern

A.5 Galerie

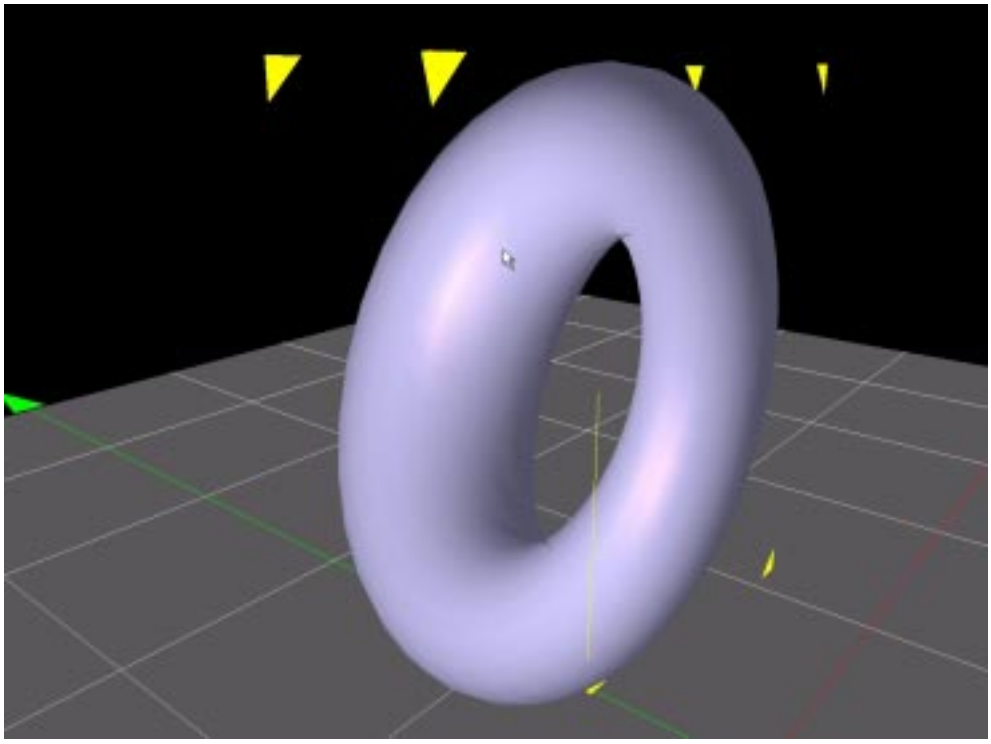


Abbildung A.1: Ein Torus in der Demoumgebung.

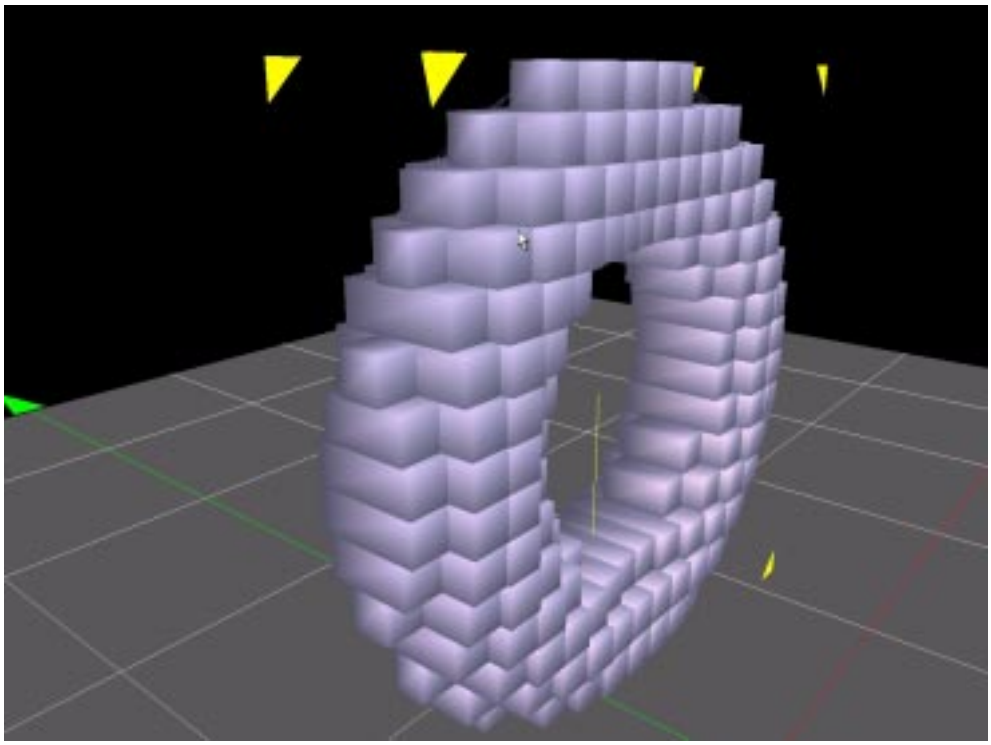


Abbildung A.2: LDI des Torus mit geringer Auflösung.

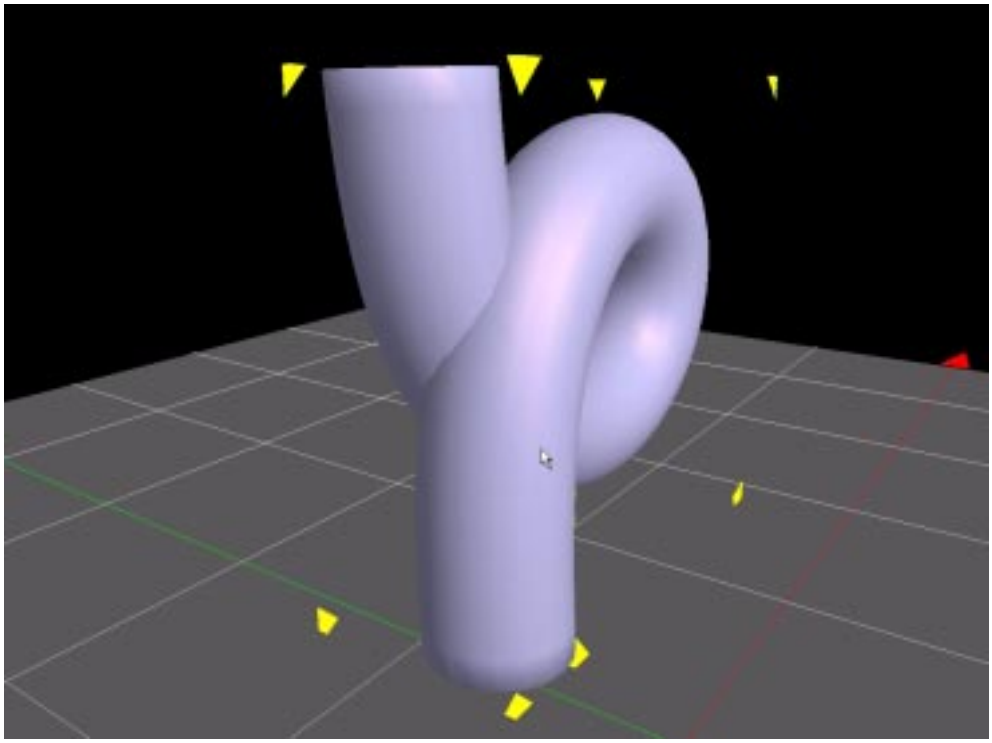


Abbildung A.3: Objekt mit einfacher Selbstkollision.

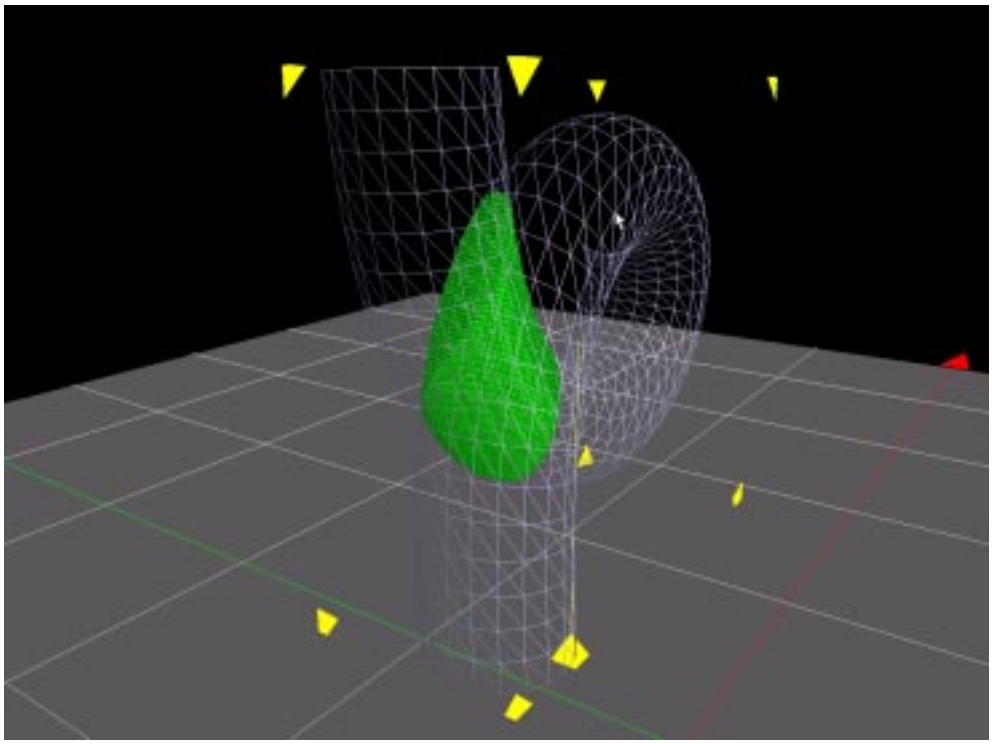


Abbildung A.4: Selbstkollisions-LDI des Objektes.

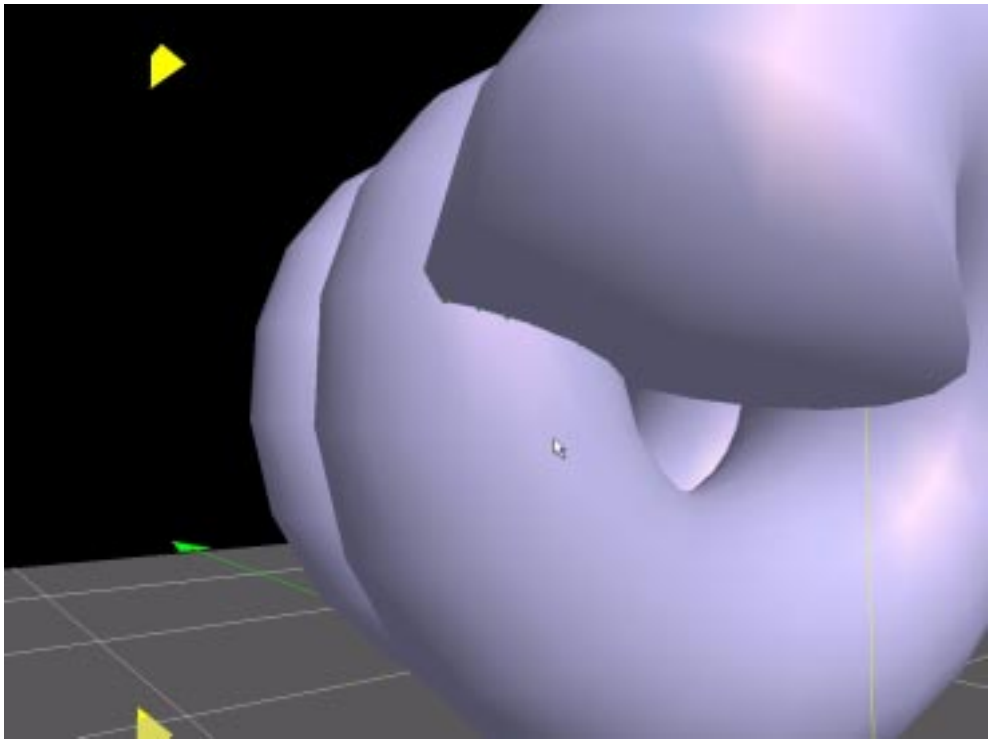


Abbildung A.5: Sich gegen hinten verjüngende Spiralfeder mit Selbstkollision.

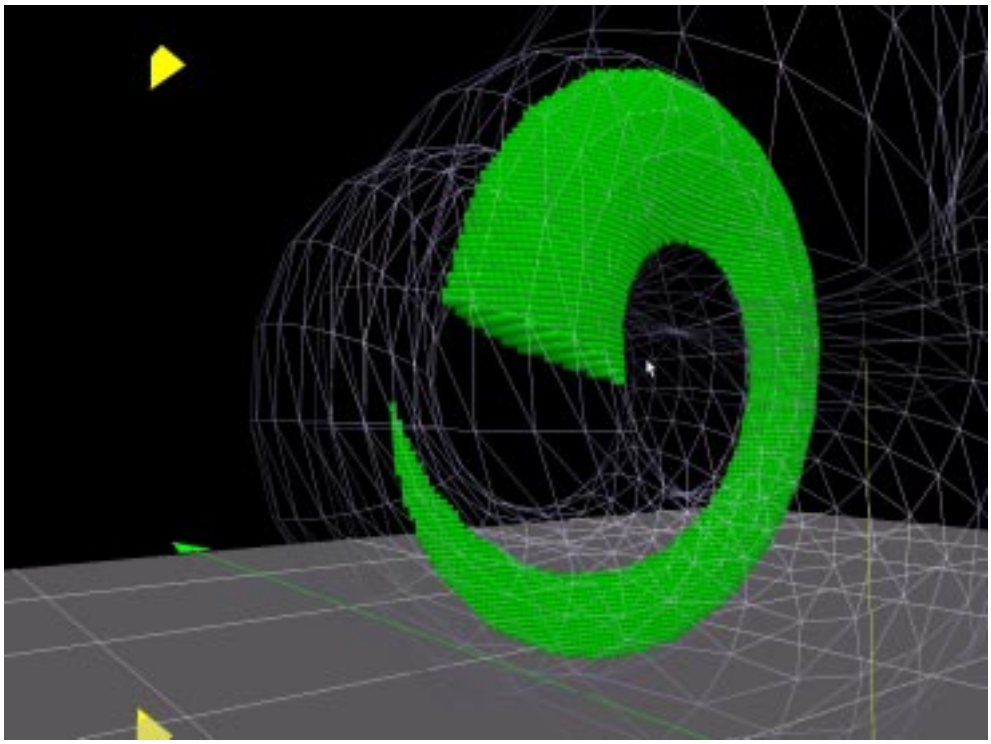


Abbildung A.6: Selbstkollisions-LDI der Spiralfeder.

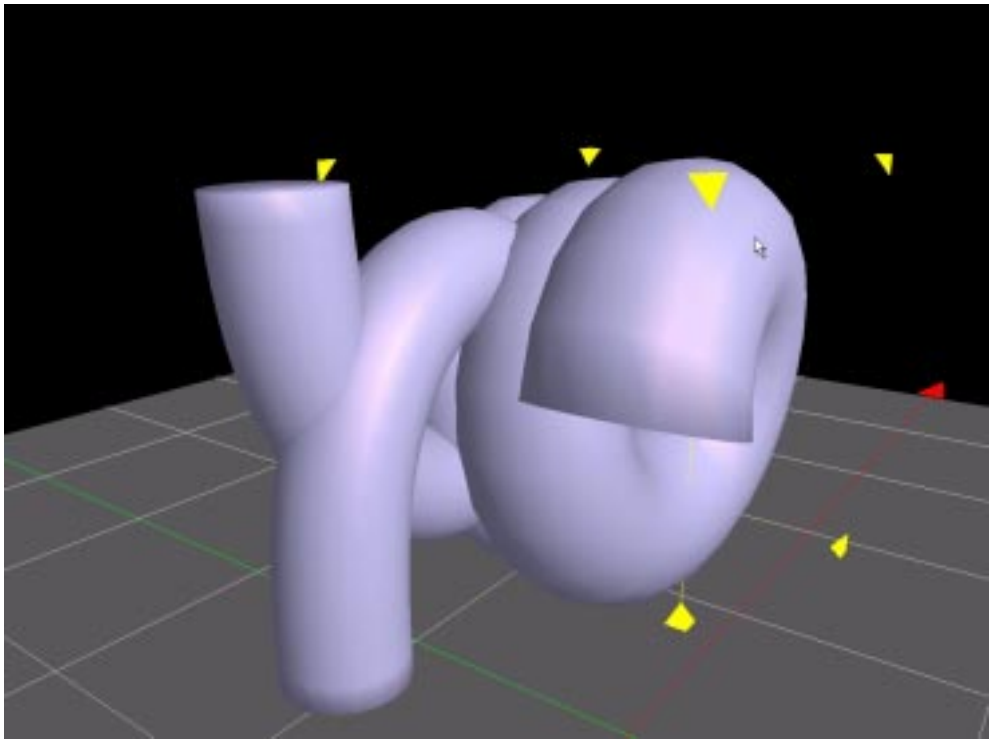


Abbildung A.7: Kollisionen und Selbstkollisionen zwischen zwei Objekten.

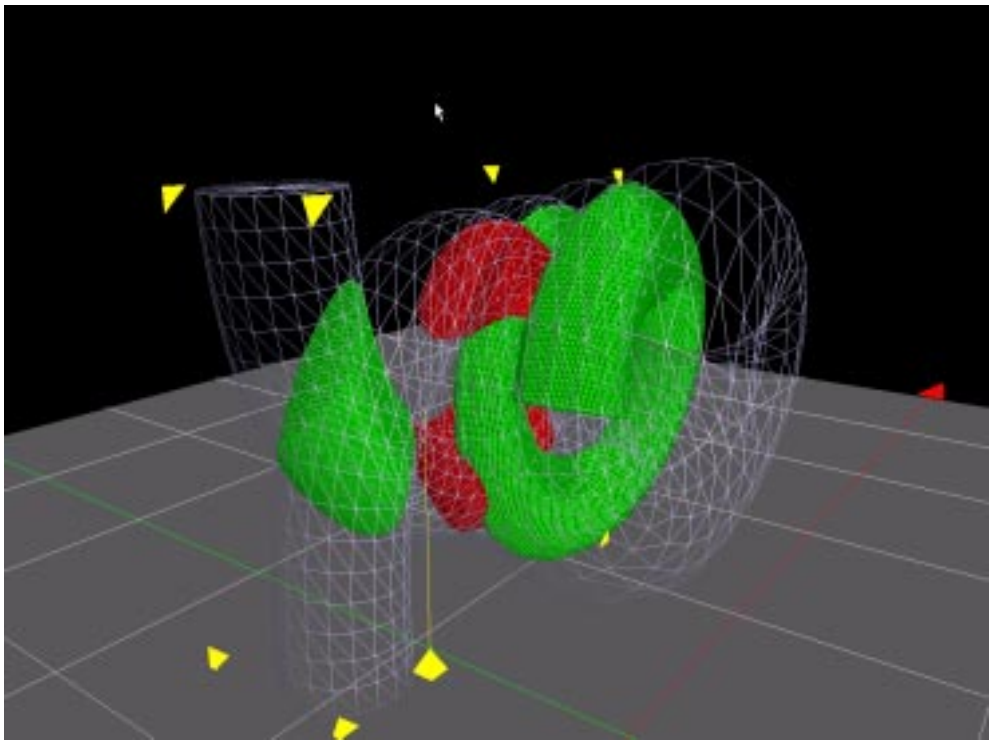


Abbildung A.8: Der Kollisions- (in rot) und die Selbstkollisions-LDI's (grün) der Situation.