Chirurgische Simulationen in VR mit Position Based Dynamics und Game Engines

Maximilian Legnar

Master-Thesis

Studiengang Angewandte Informatik
Fakultät für Mathematik und Informatik

Institut für Wissenschaftliches Rechnen (IWR)
Ruprecht-Karls-Universität Heidelberg

01.11.2020 - 30.04.2021

Betreuer
Prof. Dr. Jürgen Hesser
Prof. Dr. XXX

Legnar, Maximilian:
Chirurgische Simulationen in VR mit Position Based Dynamics und Game Engines / Maximilian Legnar. –
Master-Thesis, Heidelberg: Ruprecht-Karls-Universität Heidelberg, 2021. 20 Seiten.

Legnar, Maximilian:
Surgery simulations in VR with Position Based Dynamics and game engines / Maximilian Legnar. –
Master-Thesis, Heidelberg: Ruprecht Karl University of Heidelberg, 2021. 20 pages.

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit ist in gleicher oder vergleichbarer Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Universität Heidelberg öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Heidelberg, 01.11.2020 - 30.04.2021

Maximilian Legnar

Abstract

Chirurgische Simulationen in VR mit Position Based Dynamics und Game Engines

Ziel: Potential von PBD für serious games für chirurgischen Bereich untersuchen. Kann mit handelsüblichen VR Geräten realistische Operations-Szenarien nachgebildet werden?

NVIDIA Flex wird mit Spiele Engine (Unreal Engine) um Funktionalitäten erweitert, die für Serious Games benötigt werden, ohne Flex-Bibliothek zu verändern.

(Brücke zwischen PnysX und Flex hergestellt...)

Use Case in dieser Arbeit: Werkzeuge können in VR mit virtuellen Händen aufgehoben und benutzt werden. Mit Werkzeugen können soft bodies manipuliert werden und virtuelle Operationen durchgeführt werden. Werkzeuge: Nadel und Pinzette.

Nadel-Gewebe-Interaktionen untersucht. Wie kann man das mit dem gegebenen Modell gut erreichen?

Surgery simulations in VR with Position Based Dynamics and game engines

... Abstract in English kommt hier hin...

Inhaltsverzeichnis

Abkürzungsverzeichnis

PBD Position Based Dynamics

FPS Frames Per Second

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

teste quellen ref: [Sur, 2017]

1.1 Motivation und Problemstellung

1.2 Einordnung und Abgrenzung

1.3 Aufbau dieser Arbeit

Kapitel 2

Grundlagen

zuerst Anforderungen klären, dann theoretische Grundlagen und benutzte Software Werkzeuge erläutern...

2.1 Anforderungen

Folgende Anforderungen werden an das zu entwickelnde System und dieser Arbeit gestellt: ...

Interaktion und Echtzeit

...

Performance

Simulationsgenauigkeit

Muss nicht so gut sein.

2.2 Chirurgische Simulatoren

Was gibts da? was wird da benutzt? meist partikel basiert, technischen stand klären...

2.3 Simulationstechniken für chirurgische Simulatoren

2.3.1 Position Based Dynamics

spätestens hier Position Based Dynamics (PBD) einführen.

2.3.2 Erweiterungen von PBD

HPBD

Unified Particle Dynamics und NVIDIA Flex

zu erwähnen über [Macklin et al., 2014] und Flex:

Partikel sind über constraints miteinander verbunden.

Unified solver gut weil es einfacher zu handeln ist. Nur ein solver muss optimiert werden und vor allem weil dann objekte unterschiedlicher Art (soft, rigid, fluid) miteinander interagieren (collision constraint gillt für alle) können, aut of the box, in Echtzeit.

single particle radius sorgt für effizentere kollisionserkennen (wieso? nochmal nachlesen!)

Constaints: Distance(cloth) shape-matching(rigid, plastic/elastic deformation, siehe paper fig 5) density(fluid) volume(inflatables), friction(wichtiges constraint für nadel, flex is stolz auf friction constraint, siehe sand-teepot-demos), different collisions...

solver: PBD-solver, [Macklin et al., 2014] constraint-projektion als optimization problem, siehe kap 4.2, Gleichung (7). Bei jedem sim-step PBD versucht die Distanz zur Constraint-Verteilung (constraint manifold) zu minimieren unter berücksichtigung der partikel massen. Das tolle an dieser optimization sichtweise ist, das wir den PBD solver mit anderen solvern veergleichen können. So erkennt man das implizit euler recht ähnlich zu PBD ist. Unterschied: euler macht die minimization with respect zur initialen condition. bei BPD with respect to last iteration.

Parallelisierung mit gauss-jacobi solver (statt gaus-seidel aus PBD, der sequenziel arbeitet), der convergiert dann allerdings häufig nicht, z.b. wenn 2 partikel über identisches constraint verbunden sind. lösung: constraints paralell lösen, constraint

deltas für jedes partikel summieren und durch n=anzahl constraints teilen. Convergiert gut.

Kapitel 3

NVIDIA FleX und Unreal Engine 4 für Serius Games

In diesem Kapitel wird die verwendete Unreal Engine 4 mit FleX-integration evaluiert und wenn nötig verbessert. Die Evaluation dient dazu, einen Eindruck davon zu bekommen, wie gut die Software für die Entwicklung von Serious Games im medizinischen Bereich geeignet ist und ob noch fehlende Funktionalitäten hinzugefügt werden müssen.

Wenn eine fehlende Funktionalität entdeckt wird, die für die Entwicklung von Serius Games unverzichtbar ist, wird direkt präsentiert, wie die fehlende Funktionalität implementiert wurde.

In dieser Arbeit wurden fehlende Funktionalitäten nur durch hinzufügen neuer Softwareelemente realisiert, nicht durch Veränderung der zugrundeliegenden Unreal Integration. Dadurch können neue Versionen der Unreal Integration problemlos verwendet werden, durch einfaches Austauschen. So soll die Qualitätsanforderung *Modularität* eingehalten werden (siehe Kapitel **XXX**).

Über Unreal Engine 4 mit Flex integration

Für diese Arbeit wurde die Game Engine *Unreal Engine 4.19* mit NVIDIA FleX Integration verwendet und getestet (siehe [UE4, 2017]). Diese spezielle Version der Unreal Engine 4 beinhält nicht nur viele nützliche Elemente einer klassischen Game Engine, sondern auch eine umfangreiche Integration von NVIDIA Flex 1.2.

So bietet die Unreal Integration einen sehr schnellen und einfachen Einstieg in die Nutzung von NVIDIA Flex, weil hier bereits viele Funktionalitäten implementiert sind, die für die Nutzung von Flex benötigt werden. Polygon-Modelle (im FBX-Format) können unkompliziert importiert und in Soft Bodies, Rigid Bodies oder Cloth-Assets konvertiert werden, indem die Polygon-Modelle in Partikelmodelle

umgewandelt werden. Eine Flex-Simulation ist schnell aufgesetzt und alle Simulationsparameter lassen sich komfortabel in einer übersichtlichen Tabelle einstellen. Unterschiedliche Flex-Simulationsinstanzen können als Objekte (*FlexContainer*) vorbereitet und abgespeichert werden. Auch Fluidsimulationen können schnell und einfach aufgesetzt werden.

Unreal Engine 4.19 mit Flex-Integration steht als Open Source Variante zur Verfügung und kann von github bezogen und weiterentwickelt werden (github.com/NvPhysX/UnrealEngine/tree/FleX-4.19.2). Es ist erlaubt, den Source Code nach den eigenen Bedürfnissen anzupassen und zu erweitern, wobei man stets an die Endnutzer Lizenzbedingungen der Unreal Engine 4 gebunden ist (siehe unrealengine.com/en-US/eula/publishing).

3.1 Performance

NVIDIA Flex zeichnet sich laut diversen Quellen ([Fle, 2017], [Macklin et al., 2014], HIER NOCH MEHR REIN!) durch eine besonders gute Performance aus, wobei in [Fle, 2017] genauer darauf eingegangen wird, welchen Einfluss unterschiedliche Größen auf die Geschwindigkeit des Flex-Solvers haben.

So wirkt sich die Partikelanzahl weniger stark auf die Performance aus, wie der Partikelradius oder die Anzahl und komplexität der verwendeten Constraints [Fle, 2017]. Befinden sich beispielsweise nur Soft Bodies in einer Szene, muss der Solver deutlich weniger Constraints lösen, als in einer Szene, in der sich Soft Bodies, Flüssigkeiten und Stoffe befindet.

Im Kapitel **??** wollen wir untersuchen, ob die Performance von Flex auch zusammen in einer komplexen Game Engine zufriedenstellend ist.

Test Hardware

Für diese Arbeit und für alle durchgeführten Messungen und Tests stand die Hardware aus Tabelle 3.1 zur Verfügung.

Tabelle 3.1: Die verwendete Hardware für diese Arbeit.

CPU	Intel Core i7-9700K
GPU	NVIDIA GTX 1070 - 8GB GDDR5
RAM	4 x 8GB DDR4

Die verwendete Grafikkarte gehört zur vorletzten Generation der NVIDIA Grafikkarten und ist somit weniger aktuell und nicht übermäßig stark.

Test Szenario

Für die Anwendungsfälle dieser Arbeit sind vor allem Soft Bodies interessant. Daher wird nun eine Szene simuliert, in der sich ausschließlich Soft Bodies befinden (siehe Abbildung 3.1).

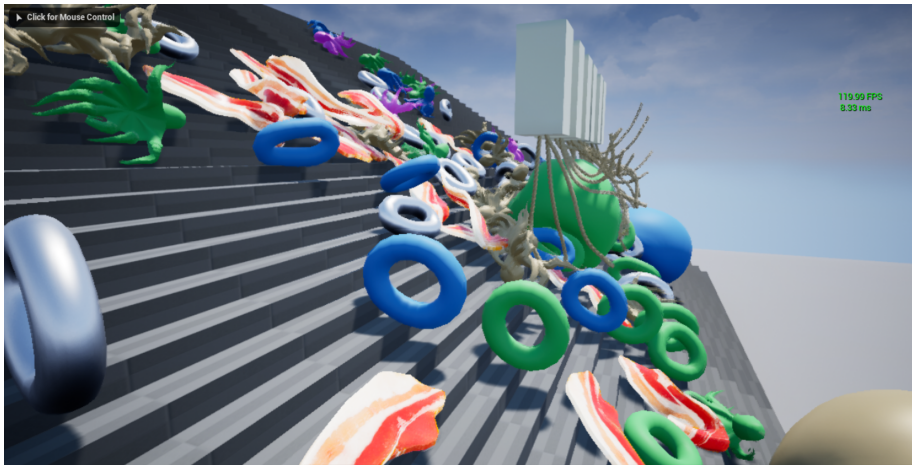

Abbildung 3.1: In der Testszene befinden sich mehrere unterschiedliche Soft Bodies, die eine Treppe hinunter rollen.

Die simulierten Körper kollidierten und verformten sich dabei stets augenscheinlich realistisch.

Folgenden Simulationsparameter wurden eingesetzt (siehe Abbildung 3.2):

Für alle Messungen wurde der Simulationsparameter *Sleep Threshold* auf einen negativen Wert gesetzt, um zu verhindern, dass zu langsame Partikel von der Simulation ausgeschlossen werden.

Alle Messungen wurden mithilfe des Befehls *stat flex* ermittelt. In [UE4, 2017] unter gameworksdocs.nvidia.com/FleX/1.2/ue4_docs/FLEXUe4_Debug.html kann nachgeschlagen werden, was die gemessenen und gezeigten Attribute bedeuten.

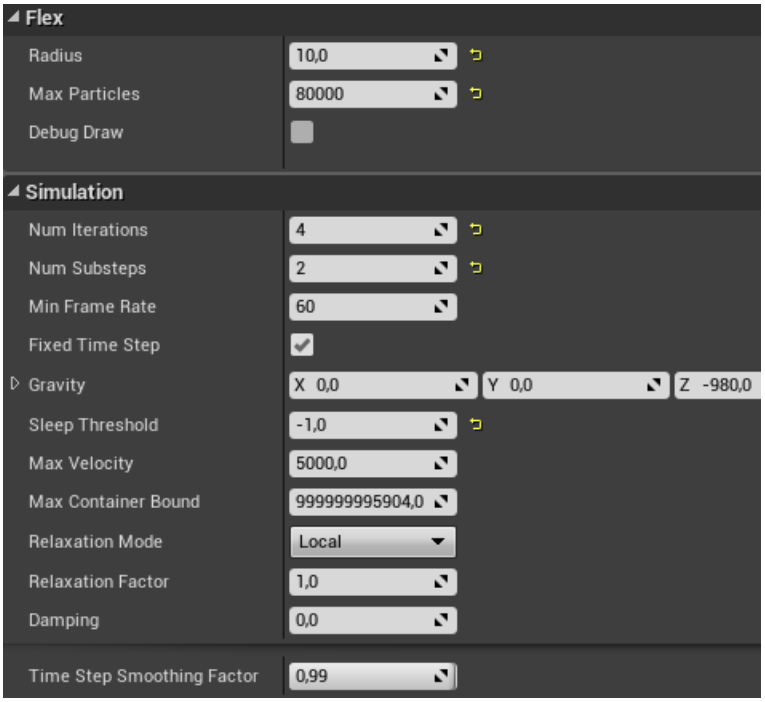

Abbildung 3.2: Die verwendeten Simulationsparamer für die erste Testreihe.

Messergebnisse

In Abbildung 3.3 sind die Messdaten zu sehen, die sich ergaben.

Zu den Messdaten sei gesagt, dass *Solve Sync Time* mithilfe der CPU gemessen wurde. *Solve Sync Time* repräsentiert also nicht nur die Zeit, die der Flex-Solver benötigt, sondern zusätzlich die CPU-GPU Synchronisierungszeit. Dennoch ist dies ein gutes Maß für die Performance von NVIDIA Flex.

Zur gemessenen Performance: Die Messungen zeigen eindeutig, dass NVIDIA Flex besonders gut für Echtzeitanwendungen geeignet ist. Selbst bei 70000 simulierten Partikeln (134 Soft Bodies) benötigt der Flex Solver durchschnittlich nur 1 ms zum Lösen aller Constraints, wobei er maximal 2,5 ms benötigt, was immer noch im akzeptablen Bereich liegt.

In Abbildung 3.3 zeigt die Linie *SumAvg* an, wie lange der gesamte Simulationsprozess im Durchschnitt gedauert hat. Der gesamte Simulationsprozess ergibt sich aus den Zeiten der folgenden Arbeitsschritte, summiert (Abbildung 3.4):

Dank der beeindruckenden Geschwindigkeit des Flex-Solvers konnten alle Tests mit 120 Frames Per Second (FPS) gerendert werden, bei einer Auflösung von ca. 1080p. So ist noch ausreichend Luft für größere Auflösungen, die vor allem im für

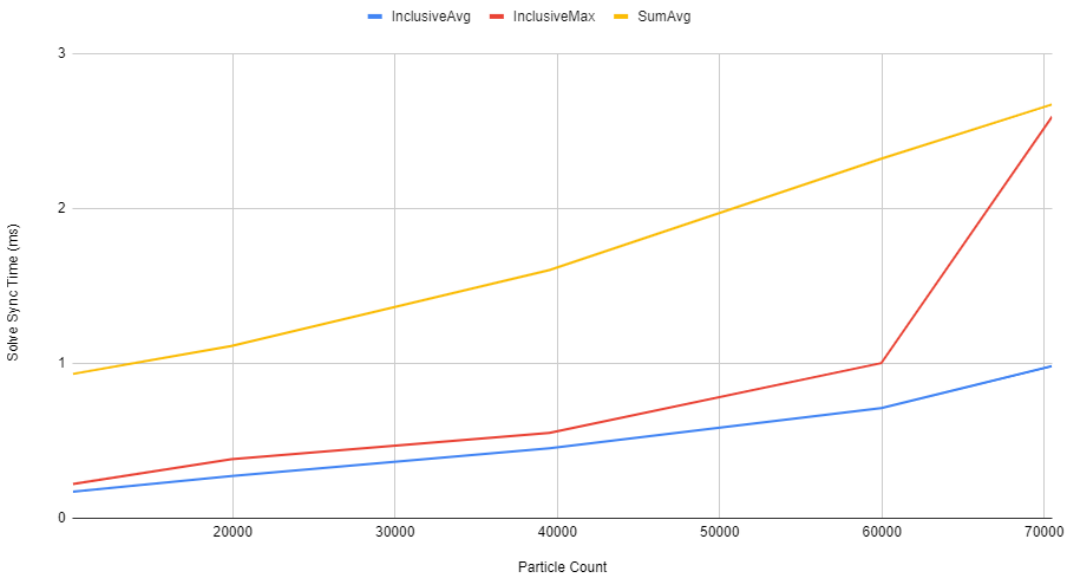

Abbildung 3.3: TEMP!!! Performance Messungen bei unterschiedlichen Partikelanzahlen mit einem Partikelradius von 10cm. Die durchschnittliche Solver Zeit steigt flach und linear mit der Partikelanzahl an.

VR-Anwendungen benötigt werden. So wie in [Fle, 2017] bereits erwähnt, hat die Anzahl der simulierten Partikel einen geringen Einfluss auf die Geschwindigkeit des Solvers.

3.2 Funktionalitäten

In der Einleitung von [UE4, 2017] wird bereits darauf hingewiesen, dass die Unreal Integration noch nicht weit genug entwickelt ist, um Gameplay-beeinflussende Physikszenarios zu erstellen. In diesem Kapitel wird genauer untersucht, welche Funktionalitäten die Unreal Integration bereits bietet und welche noch fehlen.

3.2.1 Synchronisation mit Flex Simulation

3.2.2 Transformationsmanipulationen

Game Engines verfügen in der Regel über eine umfangreiche Bibliothek für Transformationsberechnungen. So können Punkte oder Richtungen mit homogenen Transformationen manipuliert und die Transformationsmatrizen von unterschiedlichen

- **Solve Sync Time (CPU)** This is the time the CPU spends waiting for the GPU to complete the simulation work. If this is high then it means the GPU is spending a long time in the simulation kernels, use “stat Flexgpu” to get more details on the GPU bottlenecks.
- **Solver Tick Time (CPU)** This is the CPU time it takes to launch all the Flex GPU work from the CPU.
- **Update Actors Time (CPU)** The CPU time to update all the FlexActors. This includes time to update actor bounds and process any particle updates from the simulation.
- **Skin Mesh Time (CPU)** Time spent on the render thread updating vertex buffers for clothing actors.
- **Update Data Time (CPU)** The time it takes to push any parameter or particle changes from the CPU to Flex. The largest cost in this is usually memory transfer overhead related to the number of particles in the container.
- **Gather Collision Shapes Time (CPU)** The CPU time spent gathering the collision shapes overlapping each FlexActor bounds. If it is high then it may mean there are too many collision shapes overlapping FlexActors, or simply too many actors.
- **Update Collision Shapes Time (CPU)** The CPU time spent updating collisions shapes.

Abbildung 3.4: Alle gemessenen Arbeitsschritte und ihre Bedeutungen [UE4, 2017]

3D-Objekten miteinander verrechnet werden. Dies ist eine wichtige Grundlage um 3D-Objekte bewegen, rotieren und skalieren zu können.

Die Unreal Engine bietet hierfür den Datentyp *FTransform* und die Bibliothek *TransformCalculus.h* an. Jedes 3D-Objekt besitzt eine eigene Transformation, die zur Laufzeit manipuliert werden kann. Häufig werden die beiden Funktionen *SetWorldTransform(*T_{world}*)* und *SetRelativeTransform(*$T_{relative}$*)* genutzt um 3D-Objekte relativ zur Welt oder relativ zum derzeitigen Parent-Objekt zu bewegen.

Bei der Unreal Engine mit Flex Integration wurde das Transformations-Framework nur teilweise für Flex-Objekte angepasst. Obwohl alle FlexComponents eine eigene Transformation vom Typ FTransform besitzen, kann diese weder relativ noch absolut gesetzt werden. Die beiden Funktionen *SetWorldTransform()* und *SetRelativeTransform()* haben keine Auswirkung auf FlexComponents.

Diese wichtige Funktionalität muss also noch hinzugefügt werden. Dabei wollen wir das bereits bestehende Transformations-Framework der Unreal Engine nutzen.

SetWorldTransform()

Um die Welt-Transformation eines Flex-Objekts zu ändern, müssen die Positionen all seiner Partikel mithilfe einer homogenen Transformationsmatrix, T_{world}, transformiert werden. Über die NVIDIA FleX Bibliothek können die lokalen Positionen, $\vec{p}_{local}$, der Partikel eines Flex-Assets ermittelt werden. Das sind die lokalen Partikelpositionen des Flex-Assets, in seiner Ruheposition. Die neue Position, $\vec{p}_{new}$ eines Partikels ergibt sich dann mit Gleichung 3.1.

$$\vec{p}_{new} = T_{world} \cdot \vec{p}_{local} \tag{3.1}$$

Gleichung 3.1 ist im Grunde nichts anderes als die Transformation eines Punktes vom Objekt-Koordinatensystem ins Weltkoordinatensystem. Im Rahmen dieser Arbeit hat es sich als hilfreich erwiesen, die Geschwindigkeiten der transformierten Partikel auf null zu setzen, um das versetzte Flex-Objekt sofort in einen ruhigen Simulationszustand zu versetzten.

Soll ein Flex-Objekt über einen längeren Zeitraum entlang einer Trajektorie bewegt werden, ist es sinnvoll, die bewegten Partikel von der Flex-Simulation auszuschließen, indem ihre invertierten Massen auf null gesetzt werden. Dadurch sind die bewegten Partikel unendlich schwer und können nicht mehr von der Flex-Simulation bewegt werden [Müller et al., 2007] [Macklin et al., 2014].

Weil bewegte Partikel von der Simulation ausgeschlossen werden, erscheinen bewegte Weichkörper starr. Um bewegte Weichkörper weiterhin elastisch erscheinen zu lassen, wird nur eine Teilmenge der Partikel bewegt. Dann erscheint nur ein Teil des bewegten Körpers starr und der Rest bleibt elastisch und folgt den bewegten Partikeln.

SetRelativeTransform()

Um ein Partikel relativ zu einer Parent-Szene zu bewegen (*SetRelativeTransform()*), wird zunächst die allgemeine Situation aus Abbildung 3.5 betrachtet:

Gegeben sei die Welt-Transformation einer Parent-Szene, T_{parent} und die gewünschte Transformation, T_{rel}, die das Partikel relativ zur Parent-Szene einnehmen soll (siehe Abbildung 3.5). Dann kann mit Gleichung 3.2 die neue Weltposition, $\vec{p}_{new}$, des Partikels berechnet werden.

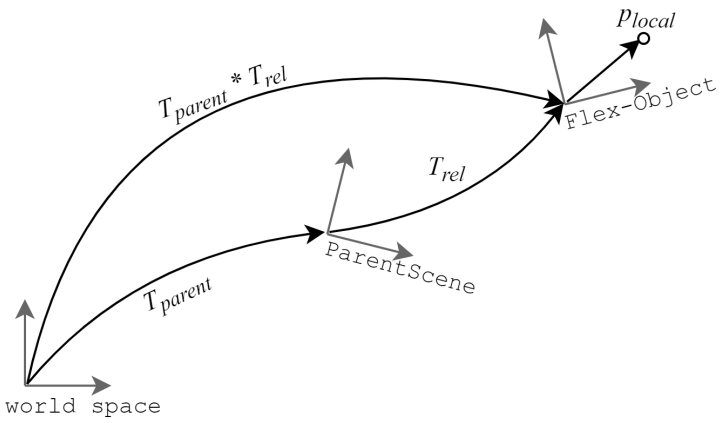

Abbildung 3.5: Um ein Partikel mit T_{rel} relativ zu *ParentScene* zu transformieren, muss das Matrixprodukt aus der Welt-Transformation von *ParentScene*, T_{parent}, und T_{rel} gebildet werden.

$$\vec{p}_{new} = T_{parent} \cdot T_{rel} \cdot \vec{p}_{local} \tag{3.2}$$

Gleichung 3.1 und 3.2 sind einfache homogene Matrix-Vektor-Multiplikationen und sollten daher effizient von heutiger Hardware gelöst werden können. Außerdem kann die Berechnung der Partikelpositionen parallelisiert werden.

3.2.3 Attachment

TODO!!!!!!

Für gewöhnlich arbeiten Game Engines mit sogenannten Scene Trees um 3D-Objekte hierarchisch zu organisieren (REFERENZ). Dabei repräsentiert jedes 3D-Objekt einen Knoten im Scene Tree. 3D-Objekte werden aneinander befestigt, so das sich eine Baumstruktur bildet, in der die Kinderobjekte an ihre Elternobjekte befestigt sind. Ändert sich die Transformation eines Elternknotens, bewegen sich all seine Kinderobjekte mit.

Die Struktur eines Scene Trees lässt sich zur Laufzeit eines Spiels ändern, indem 3D-Objekte per Funktionsaufruf von ihren Eltern gelöst (*Detach()*) oder an andere 3D-Objekte befestigt (*AttachTo(NewParent)*) werden. Diese beiden Funktionen werden in Computerspielen häufig benötigt, beispielsweise damit der Spieler 3D-Objekte aufheben (*AttachTo(Hand)*) und wieder loslassen (*Detach()*) kann.

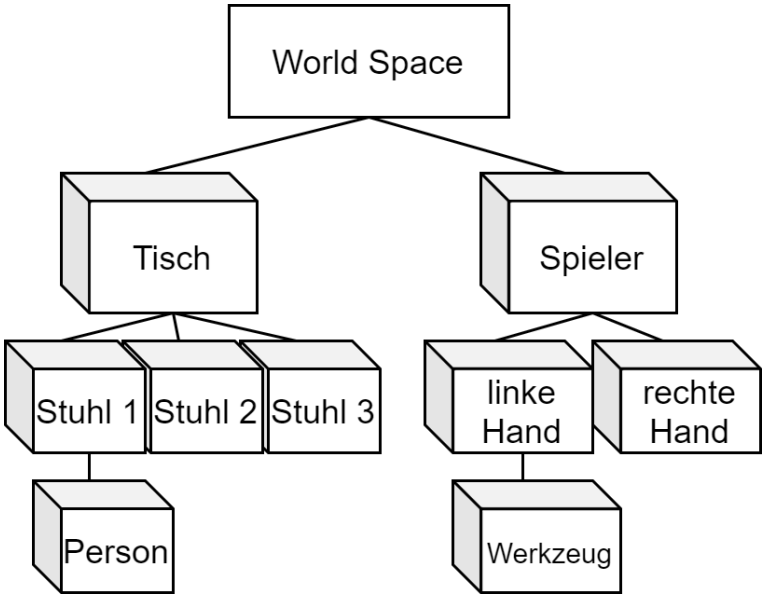

Abbildung 3.6: Alle Szenen und 3D-Objekte werden in einem Szene Tree organisiert

Befestigung von Flex-Partikeln an Szenen

In der Unreal Integration können einzelne Flex-Partikel an ein SceneComponent befestigt werden. Hierfür stehen allerdings nur Funktionen zur Verfügung, bei denen alle Partikel, die sich innerhalb einer Kugel befinden, befestigt werden. Auf Grundlage dieser Funktion wurden daher weitere Funktionen angelegt, um einzelne Partikel via Partikelindex an SceneComponents befestigen zu können.

Des weiteren musste eine Funktion zum Lösen von befestigten Partikel angelegt werden (*DetachAllFlexParticles()*), weil auch solch eine Funktionalität fehlte.

Befestigung von SceneComponents an Flex-Objekte

Funktioniert schon ganz gut, wobei 3D-modelle nicht mit soft bodies rotieren...

Fehlte: Klasse die sich an einzelne partikel anhängen kann...

Befestigung von Flex-Objekte an Flex-Objekte

siehe SoftJoint

...

3.2.4 Kollisionserkennung

Eine wichtige Funktionalität, die die meisten Game Engines bieten, ist ein Framework für die Erkennung von Kollisionen zwischen 3d-Modellen. Ein Kollisionserkennungs-Framework wird für Computerspiele häufig benötigt, beispielsweise um zu erkennen ob der Spieler in tödliche Lava fällt, ob der Spieler ein Zielgebiet erreicht hat oder ob eine geschwungene klinge ein Ziel getroffen hat. Die Programmierschnittstelle für solche Kollisionserkennungen existiert meist in der Form von *Event-Bindings* (hier evt irgenwas REFERENZIEREN).

So verhält es sich auch bei der Unreal Engine 4. Hier bieten 3D-Objekte unterschiedliche Kollisions-Events an. Zum Beispiel wird ein *BeginOverlap*-Event ausgelöst, wenn sich zwei 3D-Modelle berühren oder überlappen. Beliebige Klassen können auf das *BeginOverlap*-Event eines 3D-Modells hören (Event-Binding), dann werden Sie immer darüber informiert, wenn das 3D-Modell etwas berührt hat und können dann entsprechend reagieren (Spieler Leben abziehen, nächstes Level laden, oder ähnliches...).

Für chirurgische Simulatoren wird ein Kollisionserkennungs-Framework benötigt, beispielsweise um zu erkennen ob ein Skalpell ein empfindliches Organ berührt oder ob eine die Spitze einer Kanüle eine Blutader getroffen hat.

In der Unreal Integration können Flex-Objekte zwar grundsätzlich miteinander kollidieren, allerdings gibt es nicht die Möglichkeit, auf das Kollisions-Event eines Flex-Objekts zu hören (*Event-Binding*) oder eine Kollision zu erkennen.

Um diese fehlende Funktionalität zu ergänzen, wird das bereits bestehende Kollisionserkennungs-Framework der Unreal Engine 4 genutzt, welches auf NVIDIA PhysX aufbaut [ue4, 2021]. Dieses Kollisionserkennungs-Framework ist nämlich bereits sehr umfangreich und ist mittels Spacial-Data-Ansatz (REFERENZ!!!) optimiert und entsprechend effizient.

Hier kommt es uns zugute, dass bei der Unreal Integration 3D-Modelle an Flex-Objekte befestigt werden können (siehe kap **XXXSceneTree**). So können auch PhysX-Kollisionskörper für Flex-Objekte modelliert und an diese befestigt werden, so wie in Abbildung 3.7 zu sehen.

Die separaten Kollisionskörper werden dann so konfiguriert, dass sie keine Physik simulieren. Nur die Überlappungserkennung wird genutzt, um Überlappungen und Berührungen zu erkennen.

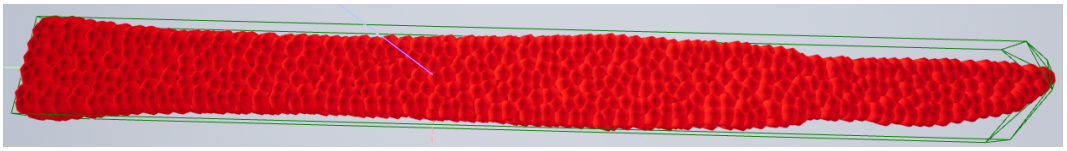

Abbildung 3.7: Ein passender PhysX-Kollisionskörper (grünes Polygonmodel) wird an ein Flex-Asset befestigt um so die umfangreiche Kollisionserkennung der Unreal Engine nutzen zu können. Hier reicht ein ungenauer Kollisionskörper aus, weil in diesem Fall nur erkannt werden soll, ob eine virtuelle Hand nach dem Skalpell greift.

An ein Flex-Asset können auch mehrere Kollisionskörper befestigt werden um zwischen unterschiedlichen Berührungen unterscheiden zu können. Beispielsweise um zu erkennen ob die Klinge oder der Griff eines Skalpells etwas berührt hat.

Hier evt nochmal ein Bild mit komplexeren Kollisionskörpern für Klinge, Griff, Kanülenspitze, Hauptader usw...

3.2.5 Interaktion zwischen Flex und PhysX

Die Unreal Engine 4 besitzt bereits eine traditionelle Physics Engine für Rigid Bodys, die auf NVIDIA PhysX basiert [ue4, 2021]. Mithilfe der Unreal Integration können also sowohl FleX-Simulationen, als auch PhysX-Simulationen ausgeführt werden. Nun stellt sich die Frage ob und wie FleX-Objekte mit PhysX-Objekte interagieren können.

TODO...

3.2.6 Kräfte und Momente

Eine Funktionalität, die einige Game Engines anbieten, ist die Möglichkeit per Funktionsaufruf Kräftevektoren auf einzelne Körper wirken zu lassen.

Diese Funktionalität wird meist genutzt um kräftebasierte Effekte oder Systeme zu programmieren. Beispielsweise können Kräfte auf Körper, die sich innerhalb einer Explosion befinden, angewandt werden, damit sie von der Explosion weg geschleudert werden. In REEEEEFFFF wird diese Funktionalität genutzt, um Auftriebskräfte auf schwimmende Objekte wirken zu lassen. Außerdem können mit dieser Funktionalität realistische Regelsysteme implementiert werden, die mit Kräften Körper balancieren oder zu einer Zielkonfiguration bewegen. In dieser Arbeit wird solch

ein kräftebasiertes Regelsystem implementiert und dient als wichtige Grundlage für die Interaktion mit Soft Bodies in virtueller Realität (siehe Kapitel XXX).

Bei der verwendeten Unreal Engine mit Flex Integration gibt es nicht die Möglichkeit, Kräfte auf einzelne Flex-Objekte oder Flex-Partikel wirken zu lassen.

Die NVIDIA FleX Bibliothek bietet zwar die Möglichkeit an, Kraftfelder zu erstellen (*NvFlexExtForceField*), allerdings gibt es nicht die Möglichkeit, Kräfte oder Momente gezielt auf einzelne Flex-Objekte oder Partikel wirken zu lassen.

Eine einfache Möglichkeit die gewünschte Funktionalität zu ergänzen, ist die Manipulation der Partikelgeschwindigkeiten. Um eine Kraft, die auf ein Partikel wirken soll, zu simulieren, wird eine Geschwindigkeitsänderung, $\Delta\vec{v}$, zur aktuellen Geschwindigkeit, $\vec{v}$, des Partikels hinzu addiert. So kann die Flex-Simulation ungestört weiterlaufen und alle Constraints, wie das Kollisions-Constraint, wirken sich wie zuvor auf die simulierten Partikel aus.

Kräfte

Gegeben sei eine Kraft, $\vec{F}$, die auf ein Flex-Objekt, das aus n Partikeln besteht, wirken soll. Dann wird $\vec{F}$ in n gleichgroße Partikelkräfte, $\vec{F_i} = \vec{F}/n$ unterteilt. Jede Partikelkraft, $\vec{F_i}$, wird dann auf ein Partikel, i, mit einer Punktmasse von m_i, angewandt.

Weil wir es mit Punktmassen zu tun haben, gilt für die Partikelkraft:

$$\vec{F_i} = m_i \cdot \vec{a} = m_i \frac{\Delta\vec{v_i}}{\Delta t} \tag{3.3}$$

Also ergibt sich die Geschwindigkeit, $\Delta\vec{v_i}$, die zur Geschwindigkeit des Partikels addiert werden muss durch:

$$\Delta\vec{v_i} = \frac{\vec{F_i}}{m_i}\Delta t \tag{3.4}$$

Wobei Δt der derzeitige Zeitschritt der Flex-Simulation ist.

Mit dieser einfachen Methode können Flex-Objekte überraschend realistisch umher geschoben werden. Wird eine Kraft von $(0, 0, 9.81)$ Newton auf ein Flex-Objekts

mit einer Masse von 1 kg angewandt, wirkt es schwerelos (bei $g = (0, 0, -9.81)$). Das zeigt, dass alle Einheiten korrekt berechnet wurden.

Momente

Wirkt ein Moment, M, auf ein Flex-Objekt, müssen auf all seine Partikel Winkelgeschwindigkeiten, $\Delta\omega_i$, hinzu addiert werden.

Die Berechnung von $\Delta\omega$ erfolgt wie zuvor, nur im rotatorischem Sinne (Gleichung 3.5):

$$\begin{aligned} M_i = m_i\dot{\omega} = m_i\frac{\Delta\omega_i}{\Delta t} \\ \Rightarrow \Delta\omega_i = \frac{M_i}{m_i}\Delta t \end{aligned} \tag{3.5}$$

Um die Rechenkomplexität zu senken, gehen wir davon aus, dass alle Partikel die selbe Masse haben, wodurch nun alle Partikel um die selbe Winkelgeschwindigkeit ($\Delta\omega_i$) rotiert werden müssen.

Neben der Größe des Moments, M, sei auch eine örtlich gebundene Achse definiert, die angibt in welche Richtung das Drehmoment wirkt. Diese Achse besteht aus einem normalisierten Richtungsvektor, $\vec{e}$, der sich an der Position, $\vec{c}$, befindet (siehe Abbildung 3.8). $\vec{e}$ und $\vec{c}$ definieren also die örtlich gebundene Achse, um die sich das Flex-Objekt drehen wird, aufgrund des wirkenden Drehmoments. Das bedeutet, alle n Partikel sollen sich nun zusätzlich mit einer Winkelgeschwindigkeit von $\Delta\omega_i$ um diese Drehmomentachse drehen.

Nun muss berechnet werden, in welche vektorielle Richtung, $\Delta\vec{v_i}$ sich ein Partikel i drehen muss, damit es sich mit der gewünschten Winkelgeschwindigkeit, $\Delta\omega_i$, um die Drehmomentachse dreht. Dabei habe das Partikel die Position $\vec{p_i}$. Abbildung 3.8 visualisiert diese Problemstellung.

$\Delta\vec{v_i}$ besteht aus einer translatorischen Geschwindigkeit, s und einer Bewegungsrichtung, $\vec{d}$, mit $|\vec{d}| = 1$. s ist die Umfangsgeschwindigkeit des Partikels, um die gegebene Achse und berechnet sich mit Gleichung 3.6.

$$s = \Delta\omega_i \cdot r \tag{3.6}$$

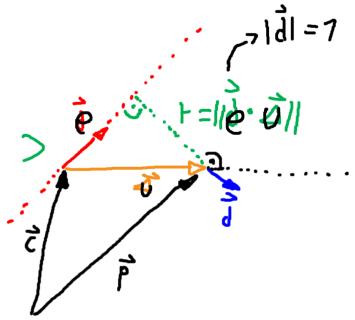

Abbildung 3.8: Das Partikel i muss sich zusätzlich in die Richtung $\vec{d}$ bewegen, damit es damit beginnt, sich um die Drehmomentachse zu drehen. Die Drehmomentachse habe die Position $\vec{c}$ und Ausrichtung $\vec{e}$

r ist dabei die kürzeste Entfernung, die das Partikel zur Drehmomentachse hat. Weil $|\vec{e}| = 1$, kann r aus dem folgenden Skalarprodukt berechnet werden (siehe Gleichung 3.7):

$$r = ||\vec{e} \cdot \vec{u}|| \tag{3.7}$$

Mit $\vec{u} = \vec{p_i} - \vec{c}$.

Der Richtungsvektor, $\vec{d}$, muss orthogonal zu $\vec{u}$ und $\vec{d}$ sein und eine Länge von 1 haben. Also kann $\vec{d}$ nach Gleich 3.8 berechnet werden:

$$\vec{d} = e(\vec{u} \times \vec{d}) \tag{3.8}$$

Wobei mit $e(...)$ der Einheitsvektor aus dem Kreuzprodukt $\vec{u} \times \vec{d}$ gemeint ist.

Schlussendlich ergibt sich die Geschwindigkeit, die zur Partikelgeschiwindigkeit hinzu addiert werden muss durch $\Delta\vec{v_i} = s \cdot \vec{d}$.

3.3 ALT

Kapitel 4

Implementation von Chirurgischen Szenarien

Um die Eignung der erweiterten Unreal Integration für chirurgische Simulatoren zu testen, werden in diesem Kapitel einige Szenarien implementiert, die für chirurgische Simulatoren interessant sein könnten.

4.1 Soft Body Interaktion mit Motion Controller

Idee und Umsetzung von “VR-Physics2.0”, wie ich es genannt habe

4.2 Nadel-Gewebe Interaktion

Wie umgesetzt, welche Tricks angewandt? Constraints? Geeignete Simulationsparameter?

4.3 Weitere Chirurgische Szenarien

Häute, schneiden und Reißen, Pinzette SPH und Körperflüssigkeiten?

4.4 ALT

Kapitel 5

Fazit und Ausblick

Literaturverzeichnis

[Fle, 2017] (2017). D3d async compute for physics: Bullets, bandages, and blood. gdcvault.com/play/1024344/. aufgerufen: 15:00, 5 März 2021.

[Sur, 2017] (2017). Surgical simulators. wiki.tum.de/display/btt/Group+4%3A+Surgical+Simulators. aufgerufen: 14:30, 4 März 2021.

[UE4, 2017] (2017). Unreal 4 flex integration - documentation. gameworksdocs.nvidia.com/FleX/1.2/ue4_docs/FLEXUe4_Intro.html. aufgerufen: 13:30, 5 März 2021.

[ue4, 2021] (2021). Unreal engine documentation - physics. docs.unrealengine.com/en-US/InteractiveExperiences/Physics/index.html. aufgerufen: 14:00, 9 März 2021.

[Macklin et al., 2014] Macklin, M., Müller, M., Chentanez, N., and Kim, T.-Y. (2014). Unified particle physics for real-time applications. *ACM Trans. Graph.*, 33(4).

[Müller et al., 2007] Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. (2007). Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118.