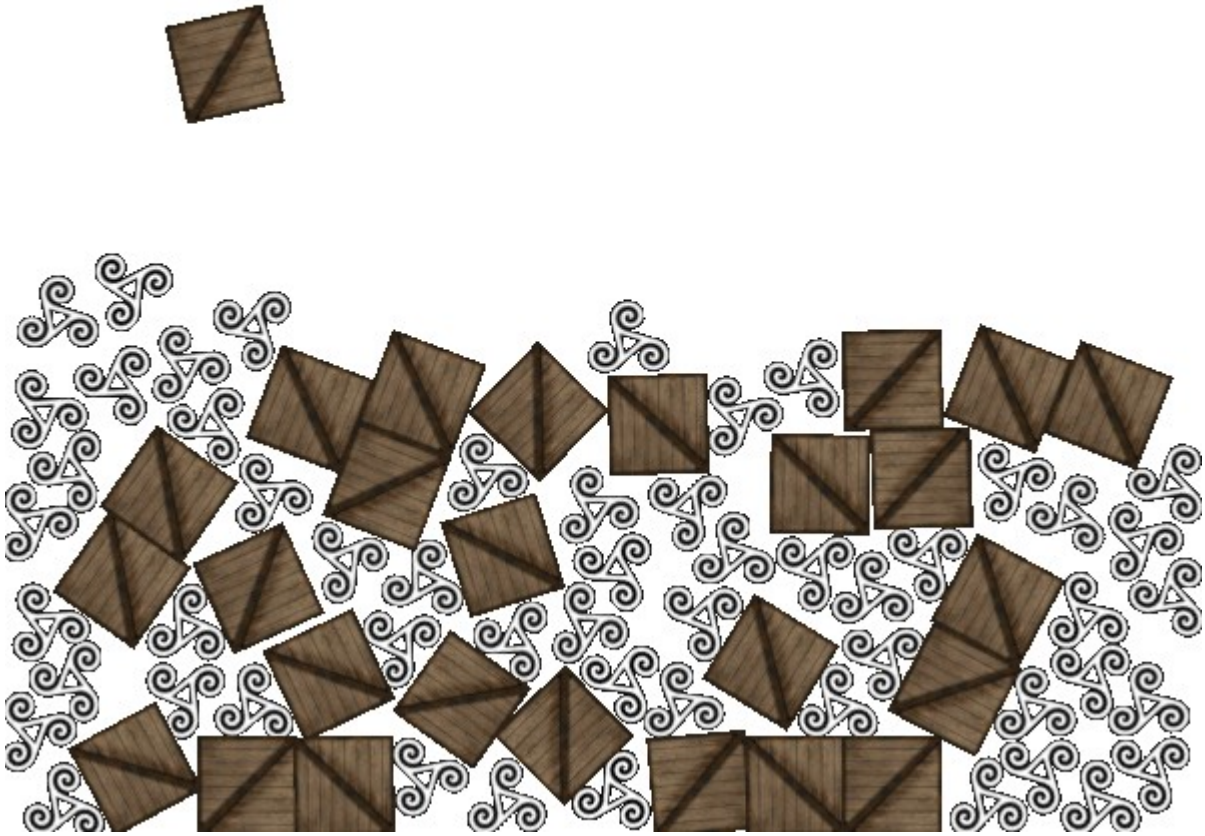
Pixel im freien Fall

Programmierung einer Physiksimulation mithilfe der Verletintegration

Benedikt Bitterli, 3BS, 2009

Betreuungsperson: Matthias Baumgartner



1. Inhaltsverzeichnis

2. Einführung / Ziel der Arbeit

Physik spielt in der heutigen Generation der digitalen Medien eine wichtige Rolle. Sei es nun der neuste Katastrophenfilm, in dem tausende virtuelle Häuser während eines Erdbebens möglichst realistisch zusammenfallen sollen oder ein Autorennspiel, in dem sich die Fahrzeuge auf realistische Art und Weise überschlagen sollen – überall sind im Hintergrund Physikprogramme tätig, die mit verschiedensten Ansätzen versuchen, ein möglichst gutaussehendes Resultat zu erzeugen.
Mich hat zum einen interessiert, wie genau ein solches Physikprogramm funktioniert, und zum anderen, ob es mir gelingen würde, ebenfalls ein solches Programm zu schreiben.
Als Endresultat soll ein Programm entstehen, welches dem Benutzer erlaubt, durch verschiedene Eingaben Objekte zu erstellen und sie zu beeinflussen.
Ausserdem soll das Programm zur Berechnung der Physik in Form einer *DLL*[1] anderen Programmierern zugänglich gemacht werden, um damit selber Spiele oder andere Anwendungen um eine Physikberechnung erweitern können.

In dieser Arbeit werde ich primär darauf eingehen, wie einzelne Teile meines Physikprogramms funktionieren. Die Texte sind dabei stets theoretisch gehalten, um dem Leser unverständliche Programmcodes zu ersparen. Die Theorie und die Formeln werden aber 1:1 wie im Text beschrieben im Programm verwendet, so dass jeder mit Hilfe dieser Arbeit selber ein Physikprogramm schreiben könnte (vorausgesetzt, er beherrscht eine Programmiersprache). Dabei konzentriere ich mich vorliegend auf den zweidimensionalen Bereich. Alle Berechnungen können selbstverständlich auch mit einigen Änderungen im dreidimensionalen Raum angewendet werden, jedoch erweist sich der verwendete Ansatz für 3D als eher ungeeignet. Die Herangehensweise, die meinem Programm zugrunde liegt, nennt sich Verletintegration. Ihre Funktionsweise werde ich ebenfalls in dieser Arbeit erklären.
Als Programmiersprache verwende ich Blitz Basic 3D sowie C, wobei in C vor allem die rechenaufwändigeren Teile des Codes geschrieben werden, da Blitz Basic dafür nicht schnell genug ist.
Alle Programme und die dazugehörigen Dateien sind sowohl auf der beigelegten CD als auch auf meiner Website zu finden, deren Adresse sich im Anhang befindet.

[1] DLL: *Dynamic Link Library,* eine Funktionsbibliothek, die anderen Programmen zugänglich gemacht werden kann.

An dieser Stelle möchte ich mich herzlich bei den Personen bedanken, ohne die diese Arbeit kaum möglich gewesen wäre. Dazu gehört selbstverständlich meine Betreuungsperson, Matthias Baumgartner, die mir durch viele Tipps und Ratschläge beim schreiben dieser Arbeit geholfen hat. Ein besonderer Dank geht ausserdem an akaz, der mir bei meinen ständigen Problemen mit C geholfen hat, ChristianK, der bei einem fatalen Problem mit dem Linker von VC++ Abhilfe geschafft hat, und hectic, der durch ständige Anregungen das Physikprogramm auf den Stand gebracht hat, auf dem es heute ist.

Ich wünsche dem Leser nun viel Spass beim Lesen meiner Arbeit.

3. Was ist die Verletintegration?

Die Verletintegration ist eine numerische Methode, um die Newtonschen Bewegungsgesetze zu integrieren. Sie wurde erstmals von Carl Störmer für die Berechnung von Partikeln im Magnetfeld verwendet und wenig später von Loup Verlet im Bereich der Molekulardynamik bekannt gemacht. Die Verletintegration wird heute sehr oft für Computersimulationen verwendet, speziell auch für Computerspiele.

Kurz gefasst lassen sich mithilfe der Verletintegration die Bewegungen einzelner Punkte zuverlässig berechnen.
Sie benutzt dafür einen speziellen Ansatz, der einige Vorteile mit sich bringt – um diese zu verdeutlichen, betrachten wir einmal die Berechnung der Position eines Punktes aus klassischer Sicht:

$$x(t) = x(t-\Delta t) + v(t-\Delta t) * \Delta t + a(t-\Delta t) * \Delta t^2$$

Hierbei steht x für die Position des Punktes, während v und a die Geschwindigkeit bzw. die Beschleunigung darstellen. Δt beschreibt dabei die vergangene Zeit seit der Berechnung der letzten Position. Zur Position des Teilchens zur Zeit t - Δt wird also seine damalige Geschwindigkeit und Beschleunigung dazugerechnet, um seine jetzige Position zu erhalten. Δt beschreibt dabei eine frei wählbare Zeitspanne - je kleiner sie ist, desto genauer wird aber die Simulation. Hat sich nämlich die Geschwindigkeit zwischen t - Δt und t geändert, so wird dies nicht berücksichtigt – es wird ja davon ausgegangen, dass der Punkt sich während der Zeitspanne Δt immer mit der gleichen Geschwindigkeit bewegt hat. Als Beispiel nehmen wir ein Auto, das sich vor 10 Sekunden mit 5 Metern pro Sekunde bewegt hat. Δt ist in diesem Fall also 10 Sekunden - würden wir also die momentane Position ausrechnen wollen, so multiplizieren wir seine Geschwindigkeit mit der verstrichenen Zeit und kommen zum Schluss, dass sich das Auto nun 50 Meter weiter vorne befinden muss. Hat es aber in der Zwischenzeit gebremst oder beschleunigt, so versagt die Methode. Die Abweichung der tatsächlichen Position des Punktes von seiner errechneten Position wird als *Fehlerwert* bezeichnet. Dieser Fehlerwert muss so klein wie möglich gehalten werden, um eine möglichst realistische Simulation zu erhalten – grössere Fehlerwerte werden nämlich sehr schnell als unrealistisch empfunden.

In der klassischen Betrachtungsweise bestehen nicht viele Möglichkeiten, den Fehlerwert klein zu halten. Im Normalfall würde man versuchen, die Zeitspanne Δt sehr klein zu halten, um immer mit einer möglichst aktuellen Geschwindigkeit weiterzurechnen. Dies ist jedoch im Anwendungsbereich der Computersimulationen nicht praktikabel, denn die Simulation muss in Echtzeit laufen - einen Anwender wird es nämlich kaum vor dem Bildschirm halten, wenn alles in Zeitlupe abläuft.
Was bedeutet das für Δt? Ein Computerprogramm läuft nämlich in so genannten Frames ab. Ein Frame ist ein kompletter Durchlauf des Codes – in einer Simulation also die Berechnung der Positionen aller Objekte für das aktuelle t. Da das t in der Simulation aber dem t ausserhalb entsprechen muss, entspricht Δt also der Zeit, die der Computer benötigt, um ein Frame zu berechnen.
Da ein Computer aber nur begrenzt viel Rechenkapazitäten zur Verfügung hat, führt das dazu, dass Δt bei vielen Objekten unweigerlich grösser wird, da auch der Computer mehr zu berechnen hat und so mehr Zeit benötigt.
Die Simulation wird also ungenauer, je komplexer die Berechnungen werden; dies ist aber ein grosses Hindernis für Programme wie Videospiele, die sowohl Realismus und Schnelligkeit erfordern als auch komplexe Szenerien zu berechnen haben.
Als Lösung für dieses Problem bietet sich die Verletintegration an. Sie kennt keine Geschwindigkeit an sich, sondern errechnet die aktuelle Geschwindigkeit des Punktes aus seiner momentanen Position und seiner Position zu der Zeit t – Δt.
Wenn man sich das kurz überlegt, erscheint es logisch: Befindet sich ein Auto vor 10 Sekunden noch 40 Meter weiter entfernt und steht es jetzt vor uns, so muss es sich in der Zwischenzeit mit 4 Metern pro Sekunde bewegt haben.
Wenn wir also berechnen wollen, wo sich der Punkt als nächstes befinden wird, so gehen wir nach der Verletintegration folgendermassen vor:

$$x(t+\Delta t) = x(t) + (x(t) - x(t-\Delta t)) + a(t-\Delta t) * \Delta t^2$$

Der Term x(t) - x(t - Δt) ersetzt also die Geschwindigkeit aus der klassischen Gleichung – er entspricht der Differenz zwischen jetziger Position und der Position zur Zeit t – Δt.
Diese Art der Berechnung bringt sehr viele Vorteile mit sich. Beispielsweise stelle man sich die Situation vor, dass ein Punkt sich in Richtung eines Würfels bewegt. Nun kann es sein, dass sich der Punkt zwischen der aktuellen Zeit und der Zeit t - Δt in den Würfel hineinbewegt hat, was ja nicht sein sollte. Um die Kollision zu beheben, genügt es in der Ver-

letintegration bereits, den Punkt wieder aus dem Würfel hinauszubewegen. Seine Geschwindigkeit passt sich automatisch an - er wird langsamer, da seine jetzige Position näher an der alten Position liegt. In der klassischen Herangehensweise müsste man nun neben der Position auch die Geschwindigkeit neu berechnen, was unter Umständen kompliziert werden kann, wenn es nicht mehr nur um Punkte, sondern um komplexe Objekte geht.
Die Verletintegration bringt also für den Bereich der Echtzeit - Computersimulationen einige Vorteile, sowohl in der Handhabung wie auch in der Geschwindigkeit. Leider hat sie auch Nachteile, auf die ich später noch eingehen werde.

4. Erweiterung der Integration

Wie bereits im vorherigen Kapitel erwähnt, ermöglicht die normale Verletintegration nur die Bewegung von einzelnen Punkten.
In gewissen Fällen ist dies bereits ausreichend, da die Integration oftmals im Zusammenhang mit Partikelsystemen verwendet wird, wo die Partikel der Einfachheit halber als Punktförmig angesehen werden.
Anders ist es jedoch in unserer Physiksimulation. Dort sind die zu berechnenden Objekte nur in seltenen Fällen punktförmig, sondern sollen eine beliebige Form annehmen können. Um die Bewegung eines solchen *Polygons* zu berechnen, muss die Verletintegration erweitert werden.
Und zwar geschieht das durch das Einführen von Federn. Diese Federn haben theoretisch eine unendliche Stärke, sie lassen sich also weder zusammenstossen noch auseinanderziehen. Es wäre also korrekter, sie vorerst als Eisenstangen anzusehen – warum dies aber doch nicht so stimmt, werde ich später noch erklären.

Diese Federn werden nun dazu verwendet, einzelne Punkte zu verbinden. Dadurch erhalten wir Objekte mit einer definierbaren Form – drei Punkte untereinander verbunden ergeben beispielsweise ein Dreieck.
Eine Feder hat neben dem visuellen Effekt noch eine andere Aufgabe, und zwar, die beiden Punkte, mit denen sie verbunden ist, in einem konstanten Abstand zu halten. Sie lässt sich ja in der Länge nicht verändern, weil sie unendlich stark ist; folglich können sich die beiden verbundenen Punkte weder annähern noch voneinander entfernen.

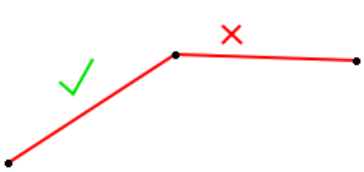
Um das Ganze zu verdeutlichen, stelle man sich ein fallendes Dreieck vor: Es besteht aus drei Eckpunkten, deren Bewegung nach wie vor durch die Verletintegration berechnet wird. Zusätzlich wird das Verhalten der Federn miteinbezogen.
Stösst nämlich einer der Punkte auf den Boden auf, so wird seine Bewegung gestoppt. Die anderen beiden Punkte bewegen sich jedoch weiter, da sie unabhängig vom anderen Punkt sind. Nun kommt aber die Federberechnung ins Spiel: Das Programm bemerkt, dass zwei Federn gestaucht sind und korrigiert die Position der verbundenen Punkte so, dass die Feder wieder ihre Ursprungslänge erreichen. Der untere Punkt bleibt weiterhin am Boden, während die anderen beiden Punkte nach oben gestossen werden.

Da aufgrund der Verletintegration aber die Geschwindigkeit der Punkte aus der Position im letzen Frame und der jetzigen Position berechnet wird, verändert sich die Geschwindigkeit der beiden Punkte automatisch. Je nachdem, wie sie zueinander standen, werden sie in unterschiedliche Richtungen beschleunigt. Das führt ohne jegliches Zutun von der Seite des Programms zu einer Drehung des Objekts.
Die Federn entfalten ihre Wirkung natürlich nicht nur bei Dreiecken, sondern auch bei beliebigen Polygonen. Die Berechnung der Federn ist dabei denkbar simpel, trotzdem erhalten wir sehr realistische Bewegungen, ohne dass wir komplizierte Formeln für den Drehimpuls etc. angewendet haben.

Soweit der Sinn der Federn - aber wie lassen sie sich berechnen?
Da wir wissen, dass die Federn unendlich stark sind, müssen sie folglich immer die gleiche Länge haben. Erstellen wir also eine Feder, so wird der Abstand der beiden verbundenen Punkte berechnet und gespeichert; dies geschieht über den Satz des Pythagoras.
Im Programm werden dann zunächst Punkte wie von der Verletintegration beschrieben bewegt. Je nach dem kann es nun sein, dass der Abstand der Punkte zueinander verändert wurde - dies muss überprüft werden.
Das geschieht wieder über den Satz des Pythagoras. Der aktuelle Abstand der Punkte wird errechnet und bei Abweichung von der Ursprungslänge muss die Feder korrigiert werden. Die Abweichung wird dabei wie folgt berechnet:

$$Abweichung = \frac{Momentane\ Länge - Ursprungslänge}{Momentane\ Länge}$$

Diese Formel errechnet die prozentuale Abweichung der momentanen Länge von der Ursprungslänge. Nun bewegen wir die Punkte entlang der Längsachse der Feder in entgegengesetzte Richtungen, und zwar jeweils um die Hälfte der Abweichung von der Ursprungslänge. Damit ist die Feder wieder korrekt ausgerichtet - jedoch ergeben sich nun andere Probleme.



Figur 4.1

Figur 4.2

Ein Punkt hängt nämlich eher selten an nur einer Feder, sondern meistens an mehreren. Korrigieren wir also eine Feder in der Länge, indem wir die beteiligten Punkte verschieben, verändern wir dadurch die Länge der anderen Federn, die auch mit diesen Punkten verbunden sind. Damit komme ich auf den bereits erwähnten Punkt zu sprechen - in der Praxis erreichen wir nie den Zustand, in dem alle Federn die korrekte Länge haben.
Je komplexer die Objekte werden, sprich, je mehr Federn ein Objekt besitzt, desto instabiler wird es – es bewegt sich, als ob es aus Gummi wäre.
Als Lösung für dieses Problem bietet es sich an, die Federn mehrmals pro Frame zu berechnen. Damit nähern sich die Federn immer mehr der Ursprungslänge an – sie erreichen sie zwar nie, aber nach ein paar Durchläufen werden die Unterschiede so gering, dass es keine sichtbaren Auswirkungen mehr auf das Verhalten des Objekts hat; es erscheint solide.
Leider hat eine höhere Genauigkeit (mehr Berechnungsdurchläufe) Auswirkungen auf die Geschwindigkeit. Eine höhere Genauigkeit bedeutet leider auch grösseren Rechenaufwand, was das Programm verlangsamt. Es muss also durch Ausprobieren ein Wert gefunden werden, der sowohl für relativ solide Objekte und auch für eine akzeptable Rechenzeit sorgt. In meinem Programm hat sich ein Wert von 10 bis 20 Durchläufen bewährt, jedoch ist das nicht immer ausreichend - je komplexer die Objekte werden, desto höher muss dieser Wert sein, da Objekte sonst 'explodieren' (siehe dazu im Anhang, Grafik 1).

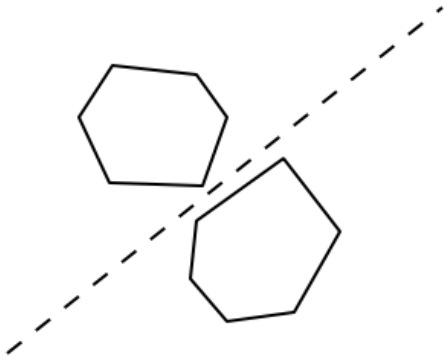
5. Implementierung von Kollision

Unsere Umsetzung der Verletintegration hält schon einiges her. Wir können die Bewegung beliebiger Objekte berechnen lassen, jedoch fehlt noch ein essentieller Teil - die Kollision. Prallen zwei Objekte aufeinander, so passiert im Moment überhaupt nichts. Sie gleiten einfach durcheinander durch, was ja nicht sein sollte. Aber wie geht man dieses Problem an?

Im Grunde genommen brauchen wir einen *Algorithmus*, der eine Kollision erkennt, wenn sie stattfindet, und die Objekte wieder auseinanderbewegt. Um das Verhalten der Objekte nach der Kollision brauchen wir uns nicht zu kümmern, da sich das dank der Verletintegration automatisch ergibt; die Geschwindigkeiten der Punkte verändern sich entsprechend, wenn wir die Objekte wieder auseinanderbewegen.

Bevor eine Kollisionsbearbeitung aber überhaupt möglich ist, benötigen wir eine zuverlässige Kollisionserkennung. Es genügt nicht, nur zu wissen, dass eine Kollision stattfindet; um sie anschliessend vernünftig bearbeiten zu können, werden zusätzliche Informationen benötigt: Zum einen die beiden Objekte, die kollidieren, wie auch die drei Punkte, die beteiligt sind. Im Grossteil der Fälle liegt nämlich die Situation vor, dass ein Punkt des einen Objekts eine Feder des anderen Objektes durchstösst. Die eher seltenen Fälle, dass sich zwei Ecken überschneiden oder sich zwei Federn berühren, betrachten wir nicht – sie werden teilweise durch die Behandlung des ersten Falls abgedeckt und es hat keine negativen Auswirkungen auf das Verhalten der Objekte, wenn wir diese Fälle vernachlässigen; eine explizite Fallunterscheidung würde zusätzliche Rechenzeit benötigen, weswegen wir sie weglassen.
Für mein Programm habe ich das sogenannte *Separating Axis Theorem*, kurz SAT, gewählt. Auf den folgenden Seiten beschreibe ich diesen Algorithmus - selbstverständlich könnte man auch einen anderen nehmen.

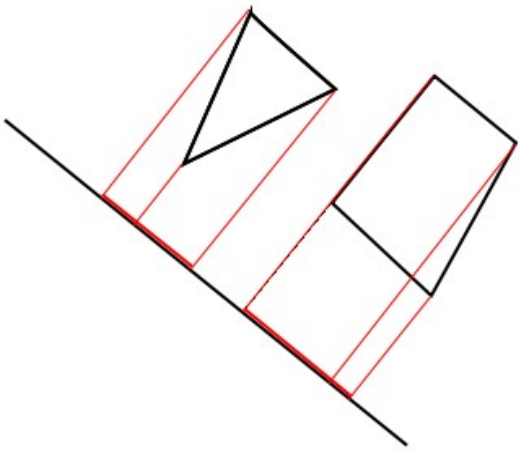
Das SAT geht davon aus, dass zwei Objekte nicht kollidieren, wenn man eine Gerade zwischen die beiden Objekte legen kann, ohne dass sie eines der beiden schneidet.



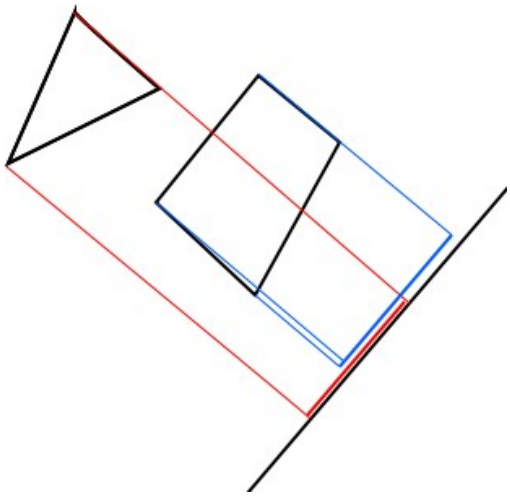
Figur 5.3

Figur 5.4

Figur 5.1 veranschaulicht das Prinzip der trennenden Achse. Leider funktioniert dieses Prinzip nur für *konvexe* Objekte; bei konkaven Objekten kann es auch keine trennende Gerade geben, auch wenn die beiden Objekte nicht kollidieren – Figur 5.2 zeigt dies.
Wie findet man aber heraus, ob man eine Gerade zwischen die beiden Objekte legen kann? Einfach alle möglichen Geraden durchzuprobieren, bis eine richtige gefunden wird, ist äusserst suboptimal und daher nicht in unserem Sinne.
Es zeigt sich aber, dass die trennende Gerade immer parallel zu einer Feder eines der beiden Objekte liegt. Es genügt also, wenn wir alle Federn beider Objekte durchgehen und sie als potenzielle Trenngeraden vermerken. Nun gilt es, herauszufinden, ob sich unter den vermerkten Geraden eine befindet, die sich zwischen die beiden Objekte legen lässt.
Dies geschieht über *Projektion*. Wenn wir eine Gerade auf Trenncharakter untersuchen wollen, errichten wir eine Senkrechte auf diese Gerade und projizieren beide Objekte darauf. Nach der Projektion erhalten wir zwei Streckenabschnitte auf der Senkrechten.



Figur 5.5



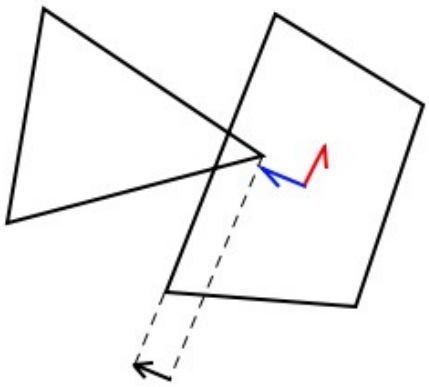
Figur 5.6

Wenn sich die beiden Streckenabschnitte nicht überlappen, wie in Figur 5.3, so lässt sich die Gerade zwischen die beiden Objekte legen; folglich kollidieren sie nicht. Überlappen sich die Projektionen aber, so wie in Figur 5.4, so lässt sich die Gerade nicht dazwischenlegen und wir müssen die nächste Gerade betrachten. Liegt am Schluss keine Gerade vor, die sich dazwischenlegen lässt, so kollidieren die beiden Objekte.

Wenn die beiden Objekte nicht kollidieren, ist alles in bester Ordnung und es gibt nichts zu tun. Kollidieren sie aber, so müssen wir die beiden Objekte geschickt auseinanderbewegen, so dass die Verletintegration greift.
Damit wir jedoch die Kollision korrigieren können, benötigen wir mehr Informationen über die Kollision. Diese holen wir zuerst einmal aus dem SAT, denn wir haben das volle Potenzial dieses Algorithmus' noch nicht ausgenutzt. Kollidieren die beiden Objekte nämlich, so muss sich das Programm die Gerade merken, auf der die geringste Überlappung der Intervalle stattgefunden hat.
Dies hat den Grund, dass wir die beiden Objekte nicht in eine beliebige Richtung auseinanderbewegen können. Damit wir keine Probleme mit der Physik bekommen, müssen wir sie nämlich um den kleinstmöglichen Vektor bewegen. Dieser Vektor verläuft parallel zu der Geraden mit der geringsten Überlappung, Seine Länge entspricht dem Betrag der Überlappung.
Mithilfe dieser Informationen können wir die beiden Objekte so auseinanderbewegen, dass sie nicht mehr kollidieren - allerdings sieht es noch nicht sehr realistisch aus. Da wir nämlich mit diesen Informationen die Objekte nur als Ganzes bewegen können, verändert sich die Geschwindigkeit und Position der Teilchen im Objekt relativ zueinander nicht,

weswegen sich die Objekte auch nicht drehen – sie bleiben starr in ihrer Orientierung, egal, wie sie aufeinander auftreffen.
Damit wir das beheben können, dürfen wir nicht das Objekt als ganzes, sondern nur die beteiligten Teilchen bewegen. Das sind exakt drei Stück; wir gehen ja nur von dem Fall aus, dass ein Punkt des einen Objekts eine Feder des andern Objekts durchstösst.
Der Punkt, der das andere Objekt durchstösst, wird *penetrierender* Punkt genannt; die durchstossene Feder *penetrierte* Feder. Die zugehörigen Objekte nennen wir entsprechend penetriertes und penetrierendes Objekt.

Damit wir diese Teilchen bestimmen können, benötigen wir ein eher komplizierteres Gebiet der Mathematik - die Matrizenrechnung. Wir müssen nämlich ein neues Koordinatensystem aufspannen, dessen Ursprung im Zentrum des penetrierten Objektes liegt und dessen Y - Achse in Richtung des Kollisionsvektors zeigt.



Figur 5.7

Figur 5.5 verdeutlicht den Zusammenhang zwischen Kollisionsvektor (schwarz) und Koordinatensystem (rot/blau), dessen Y – Achse in Richtung des Vektors zeigt.
Der Sinn hinter diesem neuen Koordinatensystem liegt darin, eine der Eigenschaften des SATs auszunutzen: Die zur trennenden Geraden gehörende Feder ist immer ein Teil jenes Objekts, welches penetriert wird. Da wir die Feder kennen, wissen wir automatisch, welches der Objekte penetriert wird und somit, von welchem Objekt wir eine Feder suchen und von welchem einen Punkt.
Die Feder, die durchstossen wird, steht dabei stets senkrecht zum Kollisionsvektor.
Dies können wir uns nun zunutze machen, indem wir alle Punkte des penetrierten Objektes in den neuen Koordinatenraum transformieren und uns die zwei Punkte heraussuchen, die den höchsten Y - Wert haben. Die penetrierte Feder beziehungsweise die mit ihr

verbunden Punkte liegen nämlich am nächsten beim anderen Objekt und haben dieselbe Y – Koordinate, da die neue Y – Achse ja senkrecht zu der Feder liegt, die zwischen ihnen aufgespannt ist.

Nun müssen wir nur noch alle Punkte des penetrierenden Objekts in das neue Koordinatensystem transformieren und den Punkt mit dem niedrigsten Y - Wert heraussuchen – das ist nämlich der Punkt, der im anderen Objekt steckt.

Nachdem wir die beteiligte Feder und den Punkt haben, müssen wir sie nur noch korrekt verschieben. Dies ist nun vergleichsweise einfach: Den penetrierenden Punkt verschieben wir einfach um die Hälfte des Kollisionsvektors zurück, während wir die penetrierte Feder um die Hälfte des Vektors nach vorne verschieben.
Die Feder erfordert jedoch noch ein wenig Zusatzberechnung – würden wir einfach die beiden verbundenen Punkte um den gleichen Vektor bewegen, ergibt sich keine Drehung. Wir müssen also feststellen, wo sich der penetrierende Punkt auf der Feder befände, wenn er um den ganzen Kollisionsvektor zurückverschoben wird. Dabei bestimmen wir die Werte t_1 und t_2, sodass gilt:

$$P_{penetrierend} = P_1 * t_1 + P_2 * t_2$$

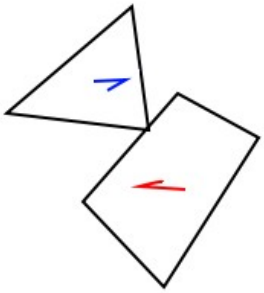
Die Punkte P_1 bzw. P_2 stellen dabei die durch die penetrierte Feder verbundenen Punkte dar. Nachdem t_1 und t_2 bestimmt sind (mit der Vektorrechnung kein Problem), müssen wir den Kollisionsvektor nur noch mit t_1 bzw. t_2 multiplizieren und zum entsprechenden Punkt aufaddieren und die Kollision ist behoben.

Damit ist die Kollisionsberechnung abgeschlossen und ein essentieller Bestandteil der Simulation geschrieben. Die Berechnung funktioniert mit beliebigen konvexen Objekten, deckt also einen sehr grossen Bereich ab. Für konkave Objekte bietet sich die Möglichkeit, sie in einzelne konvexe Objekte aufzuteilen, die miteinander verbunden sind.

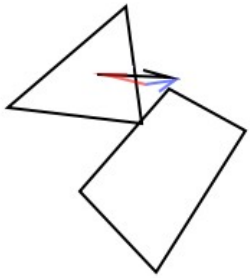
6. Implementierung von Reibung

Die Simulation auf dem jetzigen Stand ist schon relativ ausgereift und lässt sich schon sehen - das schwierigste Thema, die Kollision, ist abgeschlossen. Allerdings fehlt dem Ganzen noch etwas, um es ein grosses Stück realistischer aussehen zu lassen. Der Titel verrät es bereits - es geht um Reibung.
Reibung fehlt momentan noch komplett; die Objekte gleiten voneinander ab, als bestünden sie aus Eis. Das sieht natürlich nicht besonders gut aus, weswegen das behoben werden muss. Würde man beispielsweise ein Auto simulieren wollen, wäre ein herumfahren unmöglich, weil sich die Räder ohne Reibung auf der Stelle drehen.

Zuerst einmal müssen wir uns überlegen, wo die Reibung überall auftreten kann, damit ein Lösungsansatz herausgearbeitet werden kann. Reibung tritt an sich nur dann auf, wenn zwei Objekte kollidieren - sei es nun ein Ball, der auf dem Boden langsam ausrollt oder ein paar Würfel, die aufeinander gestapelt sind und aufgrund der Reibung nicht herunterfallen: Überall findet eine Kollision statt.
Um die Reibung errechnen zu können, bemühen wir ein wenig die klassische Physik und suchen nach einer Formel, die zu unserem Problem passen könnte.
Nun ergibt sich aber ein Problem – die Reibung ist nämlich nur für den Fall definiert, dass sich nur eines der beiden Objekte bewegt und das andere statisch ist. Das ist natürlich nicht in unserem Sinne, da sich in den meisten Fälle beide Objekte bewegen - trotzdem soll aber die Reibung berechnet werden.
Den momentan unlösbaren Fall müssen wir nun also auf den Fall zurückführen, den wir lösen können - die Situation, in der sich beide Objekte bewegen, gilt es also so umzuformulieren, dass sich nur eines der Objekte bewegt.
Dies stellt sich als relativ einfach heraus, wenn man das Problem überdenkt. Betrachtet man nämlich nur die beiden Objekte, so ist es dasselbe, wenn sich nur das eine Objekt bewegt und das andere still steht, sofern die Geschwindigkeit des Objekts der Differenz der Geschwindigkeiten der beiden Objekte in der ursprünglichen Situation entspricht.



Figur 6.8



Figur 6.9

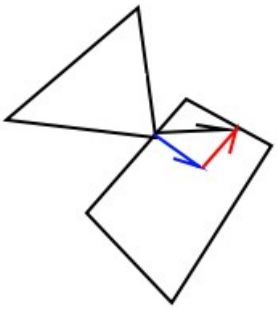
So errechnen wir zuerst die neue Geschwindigkeit vom ersten Objekt: Wir subtrahieren von seiner Geschwindigkeit die Geschwindigkeit des anderen Objekts und erhalten so die Geschwindigkeit, mit der sich das erste Objekt bewegen würde, wenn das andere stillstände. Figur 6.1 zeigt dabei die ursprüngliche Situation, Figur 6.2 den neuen Zustand. Diese Situation lässt sich nun ganz normal mithilfe der klassischen Physik lösen, welche die Reibungskraft folgendermassen formuliert:

$$Fr = N * f$$

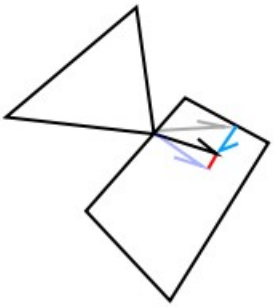
Die Reibkraft ist also ein Vielfaches der Normalkraft. Um die zugehörige Reibungskonstante zu berechnen, lassen wir der Einfachheit halber den Benutzer für jedes Objekt eine Reibzahl wählen; die Reibungskonstante errechnet sich dann aus dem Durchschnitt der Reibzahlen der beiden Objekte. In der Realität würde die Reibungskonstante ja vom Material der beiden Objekte und etwaigen Schmiermitteln etc. abhängen – es wäre viel zu kompliziert, das Programm all diese Faktoren miteinbeziehen zu lassen.

Die Normalkraft zu berechnen erweist sich jedoch zuerst schwieriger als gedacht. Eine Kraft errechnet sich ja normalerweise aus der Masse multipliziert mit der Beschleunigung des Objekts – unsere Simulation jedoch kennt weder Masse noch Beschleunigung. Für die Masse nehmen wir also vorerst 1 an, denn die Objekte werden grundsätzlich als gleich schwer betrachtet; die Beschleunigung setzen wir ausserdem der Geschwindigkeit des Objektes gleich. Damit können wir die Kraft berechnen, die in Bewegungsrichtung des Objektes wirkt.

Diese Kraft gilt es nun zu zerlegen. Der eine Anteil der erhaltenen Kraft ist nämlich die gesuchte Normalkraft, mit der wir die Reibung ausrechnen können. Diese zu bestimmen ist nicht schwer, denn die Normalkraft steht ja, wie ihr Name sagt, senkrecht zur Reibfläche, welche in unserem Fall die penetrierte Feder ist, die wir im Kollisionsteil bestimmt haben. Multipliziert mit der Reibungskonstante erhalten wir die Reibkraft – aber wie verfahren wir nun damit?
Die Reibkraft wirkt nämlich immer entgegengesetzt zur Bewegrichtung des Objekts. Damit ist aber nicht etwa die Kraft in Bewegungsrichtung gemeint, die wir vorher berechnet haben, sondern die Kraft *parallel* zur penetrierten Feder.



Figur 6.10



Figur 6.11

Figur 6.3 veranschaulicht die Zerlegung der Kraft in parallele Kraft und Normalkraft. Figur 6.4 zeigt ausserdem, wie die Reibung (in Form des türkisen Vektors) von der parallelen Kraft subtrahiert wird, um den neuen Bewegungsvektor zu erhalten.

Diese parallele Kraft ist zugleich der andere Anteil der Kraft in Bewegungsrichtung. Von ihr subtrahieren wir die Reibkraft und erhalten die neue parallele Kraft. Hier muss man aber aufpassen, wenn die Reibkraft grösser ist als die parallele Kraft – hier erhält man nach der Subtraktion nämlich einen negativen Wert und das Objekt bewegt sich plötzlich rückwärts. In diesem Fall wird für die parallele Kraft einfach 0 eingesetzt.
Die am Ende übrig bleibende parallele Kraft addieren wir zur Normalkraft und setzen sie der neuen Geschwindigkeit des Objektes gleich - man darf jedoch nicht vergessen, auf diese Geschwindigkeit wieder die Geschwindigkeit des anderen Objektes zu addieren, um die ursprüngliche Situation zu erhalten.

Nun müssen wir nur noch das gleiche Verfahren auf das zweite Objekt anwenden und wir sind fertig.

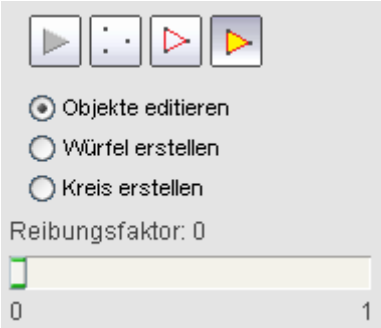
Es wird vielleicht dem einen oder anderen aufgefallen sein, dass keinerlei Unterscheidung zwischen Haft- und Gleitreibung gemacht wird. Hierbei unterschiedliche Reibungskonstanten zu verwenden ist relativ schwierig, da sich die Objekte immer ein kleines bisschen bewegen, selbst wenn es so aussieht, als stünden sie still – die Haftreibung würde also nie angewandt, da die Objekte nie stillstehen. Das Programm kommt aber relativ gut ohne Haftreibung aus, weswegen ich vorerst darauf verzichtet habe.

7. Das Physikprogramm

Während der Zeit, die mir für die Maturarbeit zur Verfügung stand, habe ich selbstverständlich nicht nur die obige Theorie studiert und hergeleitet, sondern sie auch umgesetzt in Form eines kleinen Programms, welches es dem Benutzer erlaubt, alle beschriebenen Funktionen auszutesten und damit herumzuspielen. Da die Benutzung anfangs nicht intuitiv ist, werden hier kurz die wichtigsten Tasten erklärt.
Das Programm selbst ist auf der beiliegenden CD zu finden.

Screenshot 7.1

Das Programm besteht aus 4 Modi. Sie lassen sich im Toolfenster oben rechts durch klicken auf den entsprechenden Reiter aktivieren. Von links nach rechts wären das der Simulationsmodus, der Vertexmodus, der Edgemodus und der Objektmodus.



Screenshot 7.2

Im Objektmodus werden alle Objekte erstellt und können auch zu einem gewissen Anteil bearbeitet werden. Sind entweder 'Würfel erstellen' oder 'Kreis erstellen' angewählt, so lässt sich durch das Klicken ins Fenster mit gedrückter linker Maustaste das Objekt erstellen und auf die gewünschte Grösse ziehen. Der Schieberegler bestimmt dabei, wie gross der Reibungsfaktor ist, mit dem das Objekt erstellt wird.
Ist 'Objekte editieren' aktiviert, so kann man ein Objekt durch Mausklick anwählen. Das angewählte Objekt lässt sich durch gedrückte linke Maustaste bewegen und mit gedrückter rechter Maustaste drehen. Ausserdem besteht die Möglichkeit, mit gedrückter linker Shifttaste und gedrückter linker Maustaste ein Objekt zu skalieren oder es mit gedrückter Ctrl – Taste und gedrückter linker Maustaste zu kopieren. Durch den Schieberegler kann man ausserdem den Reibungsfaktor des ausgewählten Objekts nachträglich ändern.
Im Objektmodus werden alle Objekte grau gezeichnet und das angewählte Objekt rot hervorgehoben.

Screenshot 7.3

Im Vertexmodus kann man mit Rechtsklick einen neuen Punkt zum aktuell ausgewählten Objekt hinzufügen. Mit Linksklick kann man ausserdem einzelne Punkte vom aktuell angewählten Objekt auswählen und mit gedrückter linker Maustaste verschieben. Mit der Delete – Taste lässt sich ausserdem der momentan angewählte Punkt löschen.

Screenshot 7.4

Im Edgemodus lassen sich durch Rechtsklick auf zwei verschiedene Punkte neue Federn erstellen. Mit dem Schieberegler kann man ausserdem die Federkraft der Feder einstellen, um gummige Objekte zu erhalten. Linksklick wählt die nächstliegende Feder aus und die Delete – Taste löscht die momentan angewählte Feder.

Screenshot 7.5

Der Simulationsmodus schlussendlich ist der vermutlich wichtigste Modus. Hier spielt sich die eigentliche Simulation ab; in den anderen Modi ist die Physik nämlich pausiert. Mit Linksklick in den schwarzen Kreis lässt sich die Gravitationsrichtung einstellen, ausserdem kann man mit dem Schieberegler bestimmen, wie exakt die Physik berechnet werden soll. Ein höherer Wert bedeutet natürlich eine langsamere Simulation, allerdings sollte man die Rechendurchläufe möglichst in den höheren Bereichen halten, um eine realistische Simulation zu erhalten.
Mit gedrückter linker Maustaste kann man noch den nächstgelegenen Punkt packen und das Objekt mit der Maus herumschleudern, um selber ein wenig eingreifen zu können.

8. Die Integration im Vergleich

In den meisten Spielen werden heutzutage PhysX, ODE oder ähnliche *Physikengines* verwendet. Diese betrachten im Gegensatz zu unserem Ansatz die Objekte als Ganzes; ein Objekt hat eine starre Aussenhülle, die sich nicht zusammendrücken lässt und auf welche die uns bekannten Physikgesetze angewendet werden.
Bei uns ist das anders: Wir sehen die einzelnen Punkte und Verbindungen, aus denen ein Objekt besteht. Dies bringt den Vorteil mit sich, dass alles sehr viel einfacher wird - wir müssen nicht alle physikalischen Gesetze kennen und anwenden können, die alle auf ein Objekt einwirken könnten, sondern können alles auf relativ einfache Berechnungen zurückführen.
Auch komplizierte Fälle wie bewegliche Scharniere sind kein Problem, da sich alles komplett von alleine ergibt. Gewisse Eigenschaften sind sogar nur in der Verletintegration verfügbar, zum Beispiel weiche Objekte wie etwa ein Gummiball, der sich beim Aufprall zusammendrücken lässt.
Die Integration benötigt also weit weniger Berechnungen als eine herkömmliche Physikengine und ist in den meisten Fällen darum schneller.
Andererseits ist die Verletintegration nicht wirklich für komplexere Objekte geeignet. Objekte mit vielen Verbindungen benötigen viel mehr Rechendurchläufe als ein simples Objekt, wie etwa ein Würfel, um stabil zu bleiben, und verlangsamen dadurch die Simulation als ganzes. Bei einer zu geringen Anzahl an Rechendurchläufen ist das Objekt relativ instabil und gummig, ausserdem kommt es vor, dass Objekte 'explodieren', wenn die Anzahl Durchläufe unter einen bestimmten Schwellenwert fällt.
Desweiteren ist es schwierig, die Form der Objekte zu erhalten. In einer normalen Physikengine ist das kein Problem, weil das Objekt als Ganzes als solid angesehen wird. In der Verletintegration ist uns das Objekt aber egal und die Punkte können sich bewegen, auch wenn dies die Form des Objekts verändert. Daher kommt es vor, dass Objekte einknicken oder zusammengefaltet werden; vor allem kleinere Objekte leiden unter diesem Problem.

Im 2D - Bereich halte ich die Verletintegration für die bessere Wahl, da sie relativ schnell und zuverlässig arbeitet. Im Normalfall hat man auch keine derart komplexen Objekte, als dass die Nachteile zum Tragen kommen.
In 3D jedoch zeigt die Verletintegration ihre Schwächen; es benötigt sehr viel mehr Verbindungen als in 2D, um ein einigermassen stabiles Objekt zu erhalten. Ausserdem steigt der Rechenaufwand für die Kollisionsberechnung immens, so dass der Vorteil der einfachen Berechnungen zerfällt und nur noch sehr wenig für den Einsatz der Integration spricht. [2]

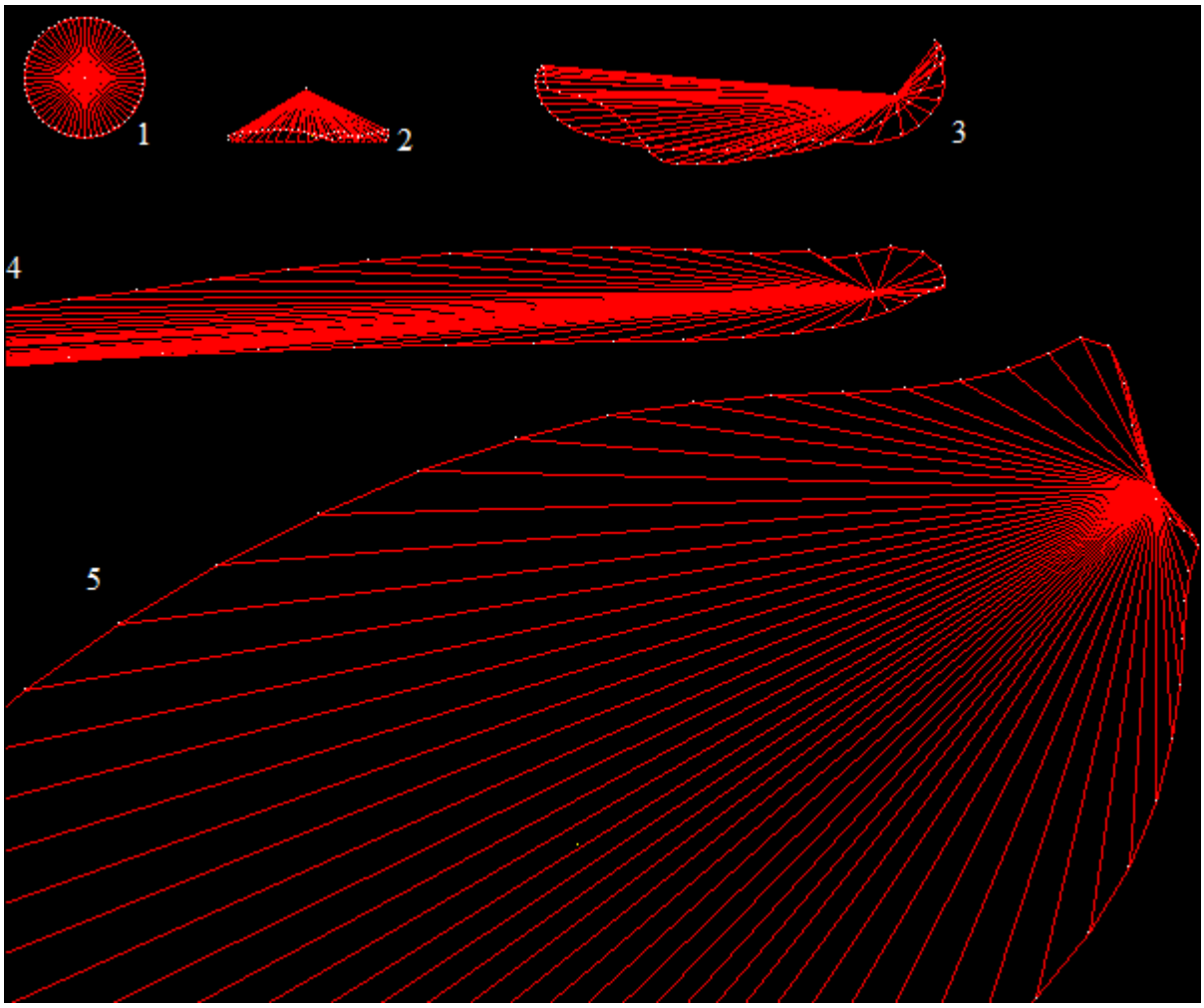
[2] Nichtsdestotrotz wurde die Verletintegration im 3D - Spiel 'Hitman: Codename 47' eingesetzt

9. Fazit

Das Schreiben dieser Arbeit hat mir sehr viel Spass bereitet. Ich konnte sehr viel daraus lernen, nicht zuletzt deswegen, weil ich auch viele Fehler gemacht habe.
Dadurch, dass vieles schief ging, war die Arbeit zum Teil auch sehr nervenaufreibend. Teilweise verbrachte ich bis zu 6 Stunden vor dem Bildschirm, nur um kleinste Tippfehler zu finden, die sich leider fatal auf das Programm auswirkten. Dabei war die Motivation entsprechend niedrig und zwischendurch liess ich die Arbeit einfach mehrere Wochen lang liegen, weil ich keine Lust mehr hatte.
Wenn ich nun auf die Zeit des Programmierens zurückblicke, hätte ich sicher einiges anders gemacht. Vor allem bei den beiden kompliziertesten Themen, Kollision und Reibung, habe ich viel Zeit damit vergeudet, einfach verschiedenste Dinge auszuprobieren, anstatt das Problem auf Papier zu formulieren und Lösungsansätze herauszuarbeiten, wie ich es schlussendlich dann getan habe. Diese Mentalität von ‚probieren geht über studieren' hat mich dann relativ viel Zeit gekostet und führte auch regelmässig zu Motivationstiefs. Es war jedoch eine gute Erfahrung, einmal etwas komplett ohne Denkanstösse aus dem Internet zu entwickeln und trotz Fehlschlägen weiterzumachen, um irgendwann auf das Ergebnis stolz sein zu können.
Ich bin relativ zufrieden mit dem Endergebnis. Das Physikprogramm und die zugehörige DLL funktionieren genau so, wie ich mir das vorgestellt hatte, als ich mit der Maturarbeit anfing. Natürlich sind sie noch nicht ganz fehlerfrei, aber sie bieten eine gute Basis für zukünftige Entwicklungen.
In der nächsten Zeit werde ich sicherlich noch an den Programmen weiterarbeiten, wobei ich aber das Hauptaugenmerk auf die DLL lege, um anderen Programmierern grössere Schnelligkeit, Robustheit und mehr Features bieten zu können. Geplant sind unter anderem Wasser und Gase, um neuartige Spielprinzipien zu ermöglichen. Mit der Veröffentlichung der DLL werden mit der Zeit sicher auch noch Fehler auftauchen, die mir noch unbekannt sind, die es dann zu entfernen gilt.
Ausserdem werde ich mich intensiver mit dem Problem befassen, unter dem meine Methode leidet; die Instabilität schmaler und komplexer Objekte muss daher behoben oder zumindest verringert werden.

Ich danke dem Leser herzlich für die Aufmerksamkeit und wünsche nun viel Spass mit dem beigelegten Programm.

10.Anhang



Grafik 1
Verhalten eines Kreises bei zu wenigen Berechnungsdurchläufen.

Links:

- Mein Worklog im Blitz Basic Portal:
  http://www.blitzforum.de/worklogs/165/

- Download des Physikprogramms mit Sourcecode:
  http://www.noobody.org/Maturarbeit/Verlet.rar

- Ein (eher älteres) Tutorial von mir zum Thema Verletintegration:
  http://www.blitzforum.de/forum/viewtopic.php?t=29286

11.Quellenverzeichnis

Bilderquellen:

Alle Grafiken und Illustrationen wurden von mir mit dem Freeware – Programm GIMP 2 erstellt.

Literatur:

- Thomas Jakobsen, *Advanced Character Physics*
  http://www.teknikus.dk/tj/gdc2001.htm (13. 9. 2008)

- Chris Hecker, *Rigid Body Dynamics*
  http://chrishecker.com/Rigid_Body_Dynamics (24. 1. 2009)

- Helmut Erlenkötter, *C – Programmieren von Anfang an*
  Rowohlt Taschenbuch Verlag, 1990

- Ulrich Breymann, *C++ - Einführung und professionelle Programmierung*
  Carl Hanser Verlag, 2007