

Physikalisch-basierte Simulation in der Computergraphik

Schriftliche Ausarbeitung

Echtzeit SPH Wasserkanal

Peter Wichert, Kirill Menke, Linda Stadter



1 Einleitung

Das Projekt soll einen realistischen Wasserfluss in einer begrenzten 3D-Umgebung in Echtzeit simulieren. Als Inspiration dient das Kinderspielzeug 'Aquaplay', bei dem Kinder über Schleusen, Kurbeln, Pumpen oder Schranken den Wasserfluss verändern können.

Zur Visualisierung des Wassers sollen zunächst kugelförmige Objekte dienen, welche durch Screen Space Rendering oder Marching Cubes zu realistisch aussehendem Wasser erweitert werden könnten.

Für ein ausreichend detailliertes Verhalten der Simulation werden etwa 30.000 Partikel simuliert. Als Kriterium für Echtzeitanforderungen sollen 30 Bilder pro Sekunde erreicht werden. Dies soll auf handelsüblicher Hardware wie der NVIDIA RTX 2070 möglich sein, ohne die Anzahl der Partikel zu verringern.

Als Entwicklungsumgebung dient die Spiel-Engine Unity, welche hauptsächlich für das Rendering eingesetzt werden soll. Echtzeit-Interaktionen lassen sich mit dieser Laufzeitumgebung einfach umsetzen, was einen weiteren Vorteil darstellt. Ausserdem bietet Unity eine übersichtliche Benutzeroberfläche.

2 Theorie

Unsere Echtzeit Wassersimulation basiert hauptsächlich auf der Methode der Geglätteten Teilchen-Hydrodynamik (SPH), welche durch eine verbesserte Suche der Nachbarpartikel beschleunigt werden soll. Zur numerischen Integration wird sowohl das Leapfrog-Verfahren als auch das explizite Euler-Verfahren eingesetzt.

2.1 Stand der Forschung

Die Methode von SPH wurde erstmalig von Gingold und Monaghan [Gingold and Monaghan, 1977] und Lucy [Lucy, 1977] vorgestellt. Ursprünglicherweise wurde sie dabei verwendet, um astrophysikalische Probleme wie die Dynamiken von Sternen darzustellen.

Müller et al. [Müller et al., 2003a] wandte zuerst SPH für Computergrafik Simulationen in Echtzeit an. Es wurde eine Methode gefunden, die schnell genug war, bis zu 5.000 Partikel in interaktiven Systemen abzubilden.

2.2 SPH mit effizienter Nachbarsuche

Bei SPH wird die zu simulierende Flüssigkeit durch eine endliche Anzahl an Partikeln diskretisiert. Die Partikel interagieren innerhalb eines bestimmten Radius h miteinander und besitzen eigene physikalische Eigenschaften wie Dichte, Masse oder Druck.

Die Nachbarpartikel eines Partikels sind die Menge der Partikel, welche sich innerhalb von h befinden und somit Einfluss auf den Partikel besitzen. Die Suche dieser Nachbarpartikel stellt einen sehr aufwendigen Schritt des SPH-Algorithmus dar. Um diese effizient zu finden, teilen wir den Raum in ein uniformes Gitter auf. Eine Zelle besitzt die Grösse des Radius h , sodass die Nachbarpartikel nur in den anliegenden 26 Zellen und in der Zelle des Partikels gesucht werden müssen. Durch ihre Positionen im Raum werden die Partikel je einer Zelle zugewiesen. Dies erfolgt zunächst durch eine Diskretisierung jeder Partikelposition (x, y, z) in Eq. (1).

$$(i, j, k) = \left(\left\lfloor \frac{x}{h} \right\rfloor, \left\lfloor \frac{y}{h} \right\rfloor, \left\lfloor \frac{z}{h} \right\rfloor \right) \quad (1)$$

Anschließend kann der diskretisierte Zellenwert in Kombination mit der Hash-Funktion aus Eq. (2) eingesetzt werden. Diese bildet eine 3D-Position eines Partikels auf einen flachen 1D-Wert ab.

$$\text{hash}(i, j, k) = (i \cdot p_1 \text{ xor } j \cdot p_2 \text{ xor } k \cdot p_3) \bmod n \quad (2)$$

Die Variable n stellt die Anzahl der Zellen beziehungsweise der Partikel dar und p_1 , p_2 und p_3 sind große Primzahlen. Dies wird Spatial Hashing genannt und wird eingesetzt, damit eine endliche Anzahl an Zellen für einen unendlich grossen Raum ausreicht. [Müller et al., 2003b]

Dann können die Partikel anhand ihrer zugewiesenen Zelle sortiert werden. Für jede Zelle wird der

Abstand zu dem ersten Partikel der Zelle gespeichert. Diese Idee wurde 2008 von Nvidia vorgestellt. [Nvi]

$$\vec{f} = \vec{f}^{pressure} + \vec{f}^{viscosity} + \vec{f}^{external} \quad (3)$$

$$\rho_i = \sum_j m_j W_{ij} \quad (4)$$

$$p_i = K(\rho - \rho_0) \quad (5)$$

$$p_i = k\left(\left(\frac{\rho}{\rho_0}\right)^\gamma - 1\right) \quad (6)$$

$$\vec{f}_i^{pressure} = - \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W(\vec{r}_i - \vec{r}_j, h) \quad (7)$$

$$\vec{f}_i^{external} = \vec{g} \quad (8)$$

$$\vec{f}_i^{viscosity} = \mu \sum_j m_j \frac{\vec{v}_i - \vec{v}_j}{\rho_j} \nabla^2 W(\vec{r}_i - \vec{r}_j, h) \quad (9)$$

Smoothing Kernels

$$W_{poly6}(\vec{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - |\vec{r}|^2)^3, & \text{falls } 0 \leq |\vec{r}| \leq h \\ 0, & \text{sonst} \end{cases} \quad (10)$$

$$\nabla W_{poly6}(\vec{r}, h) = \frac{945}{32\pi h^9} \begin{cases} \vec{r}(h^2 - |\vec{r}|^2)^2, & \text{falls } 0 \leq |\vec{r}| \leq h \\ 0, & \text{sonst} \end{cases} \quad (11)$$

$$\nabla W_{spiky}(\vec{r}, h) = -\frac{45}{\pi h^6} (h^2 - r^2)^3 \begin{cases} \frac{\vec{r}}{|\vec{r}|} (h - |\vec{r}|)^2, & \text{falls } 0 \leq |\vec{r}| \leq h \\ 0, & \text{sonst} \end{cases} \quad (12)$$

$$\nabla^2 W_{viscosity}(\vec{r}, h) = \frac{45}{\pi h^9} \begin{cases} (h - |\vec{r}|), & \text{falls } 0 \leq |\vec{r}| \leq h \\ 0, & \text{sonst} \end{cases} \quad (13)$$

Oberflächenspannung Um ein überzeugendes Verhalten bei Wasserspritzern und an der Wasseroberfläche zu erzeugen wird zusätzlich die molekulare Attraktion und Repulsion zwischen Wassermolekülen modelliert. Das gewählte Modell folgt der Arbeit von [Akinci et al., 2013].

Eine neue Kraft für Oberflächenspannung wird wie folgt in die bisherige Kraftberechnung eingefügt:

$$\vec{f} = \vec{f}^{pressure} + \vec{f}^{viscosity} + \vec{f}^{external} + \vec{f}^{surface} \quad (14)$$

Das molekulare Verhalten wird durch Eq. (16) ausgedrückt. γ sei hier ein von uns wählbarer Koeffizient.

Die Spline-Funktion aus Eq. (15) sorgt ab einem gewählten Radius h für die zunächst anziehende Wirkung, sowie den ansteigenden abstoßenden Effekt, sollten zwei Partikel zu nahe beieinander sein.

$$C(r) = \frac{32}{64\pi h^9} \begin{cases} (h-r)^3 r^3, & \text{falls } 2r > h \wedge r \leq h \\ 2(h-r)^3 r^3, & \text{falls } r > 0 \wedge 2r \leq h \\ 0, & \text{sonst} \end{cases} \quad (15)$$

$$\vec{f}_{i \leftarrow j}^{cohesion} = \gamma m_i m_j C(|\vec{x}_i - \vec{x}_j|) \frac{\vec{x}_i - \vec{x}_j}{|\vec{x}_i - \vec{x}_j|} \quad (16)$$

Um zu verhindern, dass beim Formen eines Tropfens durch Oberflächenspannung, die Ausgangsposition der betroffenen Partikel maßgeblich die finale Form beeinflusst, fügen wir eine weitere Kraft aus Eq. (18) hinzu. Diese sorgt dafür, dass Partikel-Mengen unabhängig von Ausgangsposition versuchen eine Kugelform einzunehmen und dadurch die Oberfläche ihres Tropfens minimieren.

Zunächst wird für jedes Partikel ein Normal-Vektor wie in Eq. (17) berechnet. Hier sollen sich Partikel an der Oberfläche von denen innerhalb eines Wasserkörpers dadurch unterscheiden, dass letztere einen Normal-Vektor von $\sim \vec{0}$ besitzen. Hier würde das Benutzen eines geglätteten Feldes optimale Ergebnisse erreichen. (?)

$$\vec{n}_i = h \sum_j \frac{m_j}{\rho_j} \nabla W(\vec{r}_i - \vec{r}_j, h) \quad (17)$$

$$\vec{f}_{i \leftarrow j}^{curvature} = -\gamma m_i (\vec{n}_i - \vec{n}_j) \quad (18)$$

Um den Zusammenhalt von wenigen Partikeln an den von grösseren Gruppen anzugleichen, wird in Eq. (19) ein Korrekturfaktor beim Zusammensetzen (?) der Oberflächenspannung eingeführt.

$$\vec{f}_{i \leftarrow j}^{surface} = \frac{2\rho_0}{\rho_i + \rho_j} (\vec{f}_{i \leftarrow j}^{cohesion} + \vec{f}_{i \leftarrow j}^{curvature}) \quad (19)$$

2.3 Integration

Für die Implementation von Leapfrog wir eine von [Cossins, 2010] vorgestellte Variation benutzt.

$$v_{i+1/2} = v_i + \frac{\Delta t}{2} a_i \quad (20)$$

$$r_{i+1/2} = r_i + \frac{\Delta t}{2} v_i \quad (21)$$

$$v_{i+1} = v_i + \Delta a_{i+1/2} \quad (22)$$

$$r_{i+1} = r_i + \frac{\Delta t}{2} (v_i + v_{i+1}) \quad (23)$$

Der Raum der Partikel wird durch eine quadratische Box begrenzt. Kollisionen mit den Seiten der Box werden durch einfache Positionsabfragen behandelt. Diese verhindern, dass die Partikel die Begrenzungen überschreiten. Ausserdem werden die Geschwindigkeiten der entsprechenden Richtungen umgekehrt, um die Partikel von der Box abprallen zu lassen. Eine Dämpfungsvariable wird eingeführt, die die Geschwindigkeit dabei zusätzlich abmildern kann.

3 Implementierung

Das Projekt wird in der Laufzeit und Entwicklungsumgebung Unity implementiert. Dazu wird ein sogenanntes GameObject erstellt, welches ein eigens geschriebenes C#-Skript zugewiesen wird. Dieses Skript wird verwendet, um die Simulation zu initialisieren und aufzurufen. Ein Zeitschritt wird mithilfe der Update-Funktion von Unity einmal pro Frame durchgeführt. Der eigentliche SPH-Algorithmus wird auf sieben Shader aufgeteilt, welche von der GPU ausgeführt werden. Es ist je ein Shader dafür zuständig, die Buffer zu initialisieren, die Partikel in Zellen einzuteilen, die Partikel anhand ihrer Zelle zu sortieren, die Dichte und im Anschluss die Force eines Partikels zu berechnen und zum Schluss den Integrationsschritt durchzuführen.

Die Positionen in x-,y- und z-Richtung eines Partikels werden diskretisiert und anschliessend gehasht, um daraus den flachen Zellindex zu bestimmen. Um die Anzahl der Hash-Kollisionen möglichst gering zu halten, werden die 8-stellige Primzahlen 73856093, 19349663 und 83492791 eingesetzt und eine grosse Anzahl an Partikeln - und damit auch Zellen - benutzt.

Für das Sortieren der Partikel anhand ihrer zugewiesenen Zelle wird eine fertige Implementierung des Algorithmus Bitonisches Sortieren verwendet. Dieser Sortieralgorithmus kann parallel auf der GPU ausgeführt werden.

Das Einbeziehen der Nachbarpartikel bei der Berechnung der Dichte und im Anschluss der Force wird im Dichte- und im Force-Shader auf ähnliche Weise implementiert. Hierbei muss über die 27 möglichen Nachbarzellen iteriert werden. Mithilfe

des zuvor gespeicherten Abstands kann dann effizient auf alle Partikel einer bestimmten Nachbarzelle zugegriffen werden. Zusätzlich wird überprüft, ob der Abstand zwischen den Partikeln kleiner als der Radius h ist.

Im Integrations-Shader wird neben der Berechnung der Position und der Geschwindigkeit auch die Kollision mit der Box behandelt.

Die Partikel werden in der Update-Funktion in jedem Frame gerendert. Dazu wird eine von Unity vorgegebene Zeichen-Funktion aufgerufen, welche das gleiche Kugel-Mesh parallel auf der GPU zeichnet. Dadurch verhindern wir den unnötigen Overhead, eigenständige GameObjects pro Partikel zu erstellen. In diesem Schritt werden die Partikel anhand ihrer Dichte in einen blassen bis kräftigen Blautönen gefärbt.

Neben dem GameObject, welches die Simulation behandelt, werden auch weitere Objekte benötigt. Der Raum, in welchem sich die Partikel bewegen können, wird durch eine Box begrenzt, welche zur Laufzeit erzeugt wird. Die Box wird dabei durch ein eigenes Skript kontrolliert, welches die Positionen der Seitenplatten anhand einer vorgegebenen Bodenplatte berechnet. Ausserdem werden zwei Nutzeroberflächen benötigt, um Parameter einerseits vor dem Beginn der Simulation wie in Bild 1 und andererseits während der laufenden Simulation wie in Bild 2 einstellen und anpassen zu können. Dazu werden fertige UI-Elemente von Unity verwendet. Beim Starten der Simulation findet ein Szenenwechsel statt. Dadurch werden nicht mehr benötigte Objekte entfernt und neue Objekte eingeblendet.

Algorithm 1 C# Skript

```

1: Berechne Parameter
2: Initialisiere Partikel
3: Initialisiere Buffer
4: Initialisiere Shader
5: while true do                                ▷ Unitys Update Schleife
6:   for Zeitschritt  $i + 1/2$  und  $i + 1$  do
7:     Initialization Shader
8:     ▷ Kurze Beschreibung zum Shader
9:     Partition Shader
10:    Sort Shader
11:    Offset Shader
12:    Density Shader
13:    Normals Shader
14:    Oder eine Auflistung der Formeln
15:    Force Shader
16:    7, 14
17:    Integration Shader                                ▷ 23, 22
18:  end for
19: end while

```

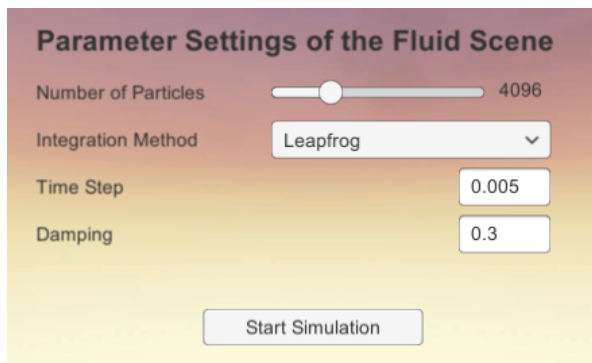


Fig. 1: Oberfläche vor dem Start, um die entsprechenden Parameter der Simulation einzustellen.

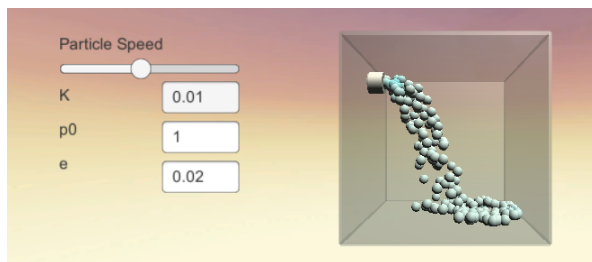


Fig. 2: Oberfläche während der Simulation, um die Parameter des SPH-Algorithmus anzupassen.

4 Ergebnisse und Evaluierung

Peter PC:

AMD Ryzen 7 3700X

32GB RAM

NVIDIA GeForce GTX 1070

Benötigte Daten:

Euler Stabilitätsgrenze

Leapfrog Stabilitätsgrenze

(vll noch je einmal mit/ohne Tait)

32768 Partikel, Leapfrog, Timestep X -j FPS
 4096 Partikel, Leapfrog, Timestep X -j FPS
 32768 Partikel, Euler, Timestep X -j FPS
 4096 Partikel, Euler, Timestep X -j FPS
 (vll noch mit/ohne Tait und Box/Rohr Test)

5 Beiträge

Kirill Menke hat über den SPH-Algorithmus recherchiert und die Methode gefunden, wie effizient auf die Nachbarpartikel zugegriffen werden kann. Er hat das Grundgerüst der Simulation in Unity aufgesetzt, sodass die einzelnen Schritte des SPH-Algorithmus auf der GPU ausgeführt werden können. Zusätzlich hat er das Rendering der Partikel implementiert, bei der Umsetzung des SPH-Algorithmus mitgewirkt

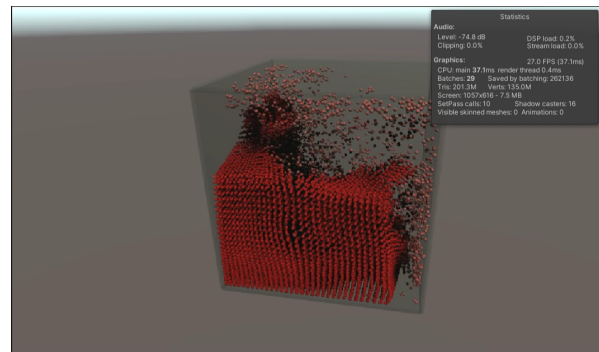


Fig. 3: Testsimulation wird instabil.

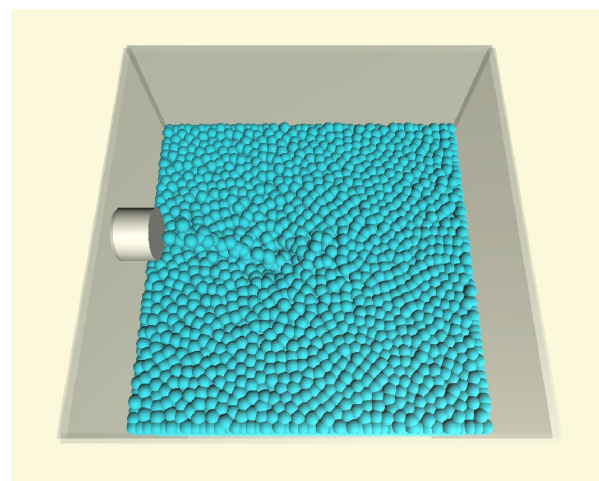


Fig. 4: Finale Simulation mit ~ 32.000 Partikeln.

und einen Modus zum vereinfachten Debuggen eingeführt. Außerdem hat er die Milestonepräsentation erstellt und vorgetragen.

Linda Stadter hat ebenfalls bei der Umsetzung des SPH-Algorithmus mitgewirkt. Zusätzlich hat sie sich um das Erzeugen der Box und die Kollisionen der Kugeln mit den Seiten gekümmert. Sie hat die beiden Nutzeroberflächen umgesetzt, die Farbcodierung der Partikel eingeführt und ein kontinuierliches Erzeugen der Partikel umgesetzt. Außerdem hat sie die Projektplanpräsentation erstellt und vorgetragen, bei der Milestonepräsentation mitgewirkt und den finalen Bericht geschrieben.

Peter Wichert hat über die Leapfrog Integration und Oberflächenspannung recherchiert und diese implementiert. Zusätzlich hat er eine automatische Berechnung der Parameter in Kombination mit dem Erzeugen der Partikel eingeführt, den Debug-Modus ergänzt und Stabilitäts-Tests durchgeführt. Außerdem hat er die Milestone Präsentation erstellt und vorgetragen und am finalen Bericht mitgewirkt.

6 Diskussion

Durch das Verwenden von Spatial Hashing und der Sortierung nach Zellen lässt sich die Laufzeit der

aufwändigen Suche der Nachbarpartikel von $O(n^2)$ auf $O(n)$ reduzieren. In Kombination mit der parallelen Ausführung des SPH-Algorithmus auf der GPU lassen sich grosse Anzahlen an Partikeln in Echtzeit simulieren. Hash-Collisions lassen sich leider nicht vermeiden und fallen bei einer sehr geringen Anzahl an Partikeln auf. Daher werden diese geringen Partikelanzahlen von der Simulation ausgeschlossen. Da in unserem Modell die Geschwindigkeit v abhängig ist von der Beschleunigung a müssen wir für die Leapfrog Integration zwei Berechnungen der Kräfte ausführen. Der dadurch entstehende zusätzliche Rechenaufwand wird jedoch durch die erhöhte Stabilität und dadurch größere Zeitschritte übertroffen. Ungewiss ist ob zukünftige Erweiterungen in Sachen Visualisierung des Wassers und Interaktion mit Festkörpern die aktuelle Echtzeit-Performance signifikant negativ beeinflussen würden. Allgemein zeigt die Simulation realistisches Wasserverhalten auf und kurzfristig könnte nur das Verhalten der Oberflächenspannung durch Glätten des Feldes für die Normalen Berechnung etwas verbessert werden.

7 Zusammenfassung

Das Projekt hat letztendlich die meisten gesetzten, sowie einige optionale, Ziele erreicht. Ein realistisches Wasserverhalten durch SPH mit zusätzlicher Oberflächenspannung wurde erfolgreich umgesetzt. Die zunächst optionale GPU Implementierung und Spatial Hashing sorgten dafür, dass die gewünschte Menge an Partikeln mit stabiler Frameanzahl simuliert werden konnten. Abstriche mussten allerdings in Sachen Visualisierung gemacht werden, da in dem Projekt der Fokus hauptsächlich auf die physikalische Simulation gelegt wurde. Weitere Zusätze wie die Interaktion mit Objekten und die Komplexität des Wasserstroms wurden aus zeitlichen Gründen nicht implementiert, eignen sich aber gut, um sie im Rahmen eines weiteren Projekts umzusetzen.

Literatur

- Particle-based fluid simulation.
http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_ParticleFluids.pdf.
 Accessed: 2021-02-24.
- Nadir Akinci, Gizem Akinci, and Matthias Teschner. Versatile surface tension and adhesion for sph fluids. *ACM Transactions on Graphics*, 10 2013. doi: 10.1145/2508363.2508395.
- Peter Cossins. Smoothed particle hydrodynamics. pages 38–40, 07 2010.
- Robert A Gingold and Joseph J Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181(3):375–389, 1977.
- Leon B Lucy. A numerical approach to the testing of the fission hypothesis. *The astronomical journal*, 82:1013–1024, 1977.
- Matthias Müller, David Charypar, and Markus H Gross. Particle-based fluid simulation for interactive applications. In *Symposium on Computer animation*, pages 154–159, 2003a.
- Matthias Teschner Bruno Heidelberger Matthias Müller, Danat Pomeranets, and Markus Gross. Optimized spatial hashing for collision detection of deformable objects. *Vision, Modeling, and Visualization 2003: Proceedings, November 19-21, 2003, München, Germany*, page 47, 2003b.