Simulationsverfahren von Wasseroberflächen in Echtzeit anhand des Beispiels von Height Field Fluids mit WebGL

Felix Riehm
Hochschule für Technik und Wirtschaft
Berlin, Deutschland
Felix.Riehm@student.htw-berlin.de

***Abstract*—Die Darstellung von Wasser ist in der Computergrafik eine herausfordernde Aufgabe. Optisch muss die Oberfläche glaubwürdig aussehen und besonders in Computerspielen muss die Berechnung in Echtzeit geschehen. In dieser Arbeit werden Herangehensweisen und Anforderungen von Simulationsverfahren für Wasseroberflächen gezeigt. Der Hauptfokus liegt auf dem Ansatz des Height Field Fluid Algorithmus von Matthias Müller und Nuttapong Chentanez. Der Height Field Fluid Algorithmus ist eine einfache und effiziente Methode, um Wasseroberflächen in Echtzeit zu simulieren. Es wird gezeigt, wie dieser funktioniert und in einer eindimensionalen Variante mit WebGL implementiert werden kann.**

***Index Terms*—computer graphic, water surfaces, simulation of water, height field fluid, webgl, realtime rendering**

I. Einleitung

Die Darstellung von Wasser in der Computergrafik kann in zwei Segmente unterteilt werden. Die Simulation von Wasser, also die Berechnung der Geometrie der Wasseroberfläche, und das Rendern der Oberfläche, was das visualisierte Bild der Oberfläche entspricht. Bei der Visualisierung spielt beispielsweise das gebrochene Licht mit der Wasseroberfläche eine Rolle [1], da man damit die typischen Brennlinien Effekte auf dem Wasserboden darstellen kann (Dieser Effekt wird oft mit einem Shader imitiert, anstelle einer genaueren Pathtracing Methode). Aber auch die Gischt des Wassers, das in der Regel mit Partikeleffekten realisiert wird, spielt eine wichtige Rolle für eine glaubwürdige Darstellung. Weitere Aspekte können in den Arbeiten [6], [7] und [1] gefunden werden. Diese Arbeit grenzt sich in diesem Hinblick auf die Simulation ein.

Nicht behandelt werden in dieser Arbeit zudem die Theorien der Ozeangraphie, z.B. die Strömungen oder Turbulenzen des Meers. Ebenso wenig wird genauer auf die verschiedenen Wellenarten eingegangen, die z.B. durch Wind oder Erdbeben entstehen. Für eine genaue und realistische Entwicklung einer Simulation kann dies jedoch relevant sein. Für Echtzeitverfahren eher weniger.

Um verschiedene Verfahrensarten kennenzulernen, werden im Abschnitt "Herangehensweisen" die wichtigsten diskutiert. Diese lassen sich in Simulation von Wasseroberflächen und Wasservolumen unterteilten. Diese Arbeit konzentriert sich jedoch auf Echtzeitsimulationen, weswegen im Abschnitt "Anforderungen", nur Anforderungen gelistet werden die für eine Echtzeitsimulation zutreffen. Eine Methode, für die Implementierung einer Echtzeitsimulation von Wasser, ist die Height Field Fluid Methode von Matthias Müller und Nuttapong Chentanez [2]. Diese wird als Beispiel genommen, da sie einfach umzusetzen ist und gute Performance bietet. Diese Methode wird im Detail erläutert und im Anschluss für diese Semesterarbeit im eindimensionalen Fall mit WebGL umgesetzt. Eine Implementation im zwei dimensionalen Fall kann in der Bachelorarbeit [4] und Projektarbeit [21] gefunden werden.

II. Herangehensweisen

Es gibt grundlegend zwei Arten, wie man Wasser simulieren kann:

- **Wasseroberlächen** (auch grid-based methods [3], liquid surfaces [4] oder procedural water [5] genannt)
- **Wasservolumen** (auch particle-based methods [3], fluid dynamics [4] oder particle systems [5] genannt)

Diese werden in den zwei nächsten Unterabschnitten kurz erläutert.

A. Wasseroberlächen

Bei der Simulation von Wasseroberflächen wird, im Gegensatz zu den Verfahren von Wasservolumen, nur die Oberfläche behandelt und geometrisch beschrieben. Man kann die Generierung in drei Varianten beschreiben:

Vordefiniert: In den meisten Spielen, wo das Wasser nicht zu den Hauptspielementen gehört, wird die Fläche einfachheitshalber als flache Ebene definiert und dient hauptsächlich dazu, die Szene auf simple Weise lebensechter wirken zu lassen [4]. Mit Shadern kann für eine ausreichende Illusion von bewegendem Wasser gesorgt werden, indem im simplesten Fall, beispielsweise mit einer Normal Map die Normalvektoren für die Beleuchtung auf der Oberfläche im Fragment Shader verändert werden, oder im komplexesten Fall, anhand einer Textur, nachträglich im Vertex Shader Vertices geändert werden. Interaktionen mit der Oberfläche, abseits von Wasserspritzer, die mit Partikeleffeken erzeugt werden, sind nicht möglich.

Height Field Fluids: Möchte man einen Schritt weiter gehen und eine begrenzte Interaktion mit dem Wasser zulassen, zum Beipsiel das Erzeugen von kleinen Wellen, wenn

ein Objekt in das Wasser fällt, dann eignen sich Height Field Fluid Methoden, wie diese, die in dieser Arbeit beschrieben wird. Die erzeugten Wellen sind mit dieser Methode jedoch nur temporär vorhanden, da das Wasser kurz nach einer Erzeugung wieder in den flachen Ursprungszustand zurück fällt. Permanente Wellen, angetrieben durch den Wind, wie man sie auf einem unruhigen Meer sieht, werden nicht unterstützt. Weswegen sich diese Methode hauptsächlich für Teiche, Seen oder Flüsse eignet [4] [5].

Prozedural: Möchte man Ozeane darstellen, sind Modelle, die auf Gerstner Wellen [9] [10] [11], Fast Fourier Transformation (FFT) [6] [13] [14] oder der einfachen Überlagerungen von Sinuswellen basieren, die gängigsten. Diese Methoden sind prozedural generiert und haben dadurch gegenüber der Height Field Fluid Methode den Vorteil, dass die Berechnung der Bewegung einer generierten Welle nicht über die komplette Flächendistanz fortgesetzt werden muss.

Bei dem einfachsten Modell, der Überlagerung von Sinus- bzw. Cosinusfunktionen, werden Kurven mit verschiedenen Frequenzen, Wellenlänge und Amplitude erzeugt (siehe Fig. 1 obere Hälfte). Dieser werden überlagert (siehe Fig. 1 untere Hälfte) und anschließend mit anderen solcher Erzeugnissen mit unterschiedlichen Richtungsvektoren aufsummiert (siehe Fig. 2 obere Hälfte). Schlussendlich ergibt sich ein relativ gutes Ergebnis (siehe Fig. 2 untere Hälfte). Bei dieser Methode werden, wie bei den Height Field Fluids, nur die Höhenwerte geändert. Die Werte der x und z Ebene bleiben bestehen. Das Polygonnetz bewegt sich also nicht mit der Bewegung des Wassers mit. Wenn man nahe an das Erzeugnis heranzoomt, ist ein sich wiederholendes Muster nicht erkennbar. Bei ferner Betrachtung kann dies jedoch auffallen. Ein anderes Problem ist, dass bei sehr hohen Wellen die Berge abgerundet sind, was nicht den stürmischen hohen Wellen entspricht, die man aus der Realität kennt [1]. Gerstner Wellen beschreiben Wellen

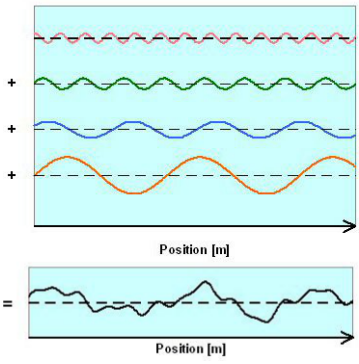

Fig. 1. Summe verschiedener Sinuswellen [15].

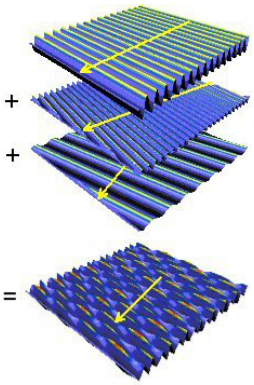

Fig. 2. Summe von Wellen mit unterschiedlichen Richtungen [15].

mit Trochoiden (siehe Fig. 3). Trochoiden bilden eine Linie um einen sich rotierenden und bewegenden Kreis, welche als Grundlage für die Wasseroberfläche genommen wird (siehe Fig. 4). Die Idee der Trochoiden führt von den Kreisbewegungen der Wasserteilchen im Tiefwasser her. Gerstner Wellen haben die Eigenschaft, dass die Wellen spitzförmiger zulaufen, was zu realistischeren Wellen bei stürmischen Wetter sorgt. Anders als bei der einfachen Überlagerung von Sinusfunktionen, ändern sich hier auch die Werte der x und z Ebene, und nicht nur die Höheninformation. Es kommt auch bei dieser Methode zur Musterbildung [1]. Modelle, die die FFT zur

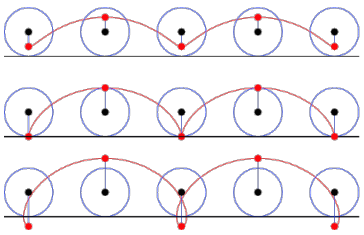

Fig. 3. Verschiedene Trochoiden und deren Lauflinien eines fixierten Punktes [16].

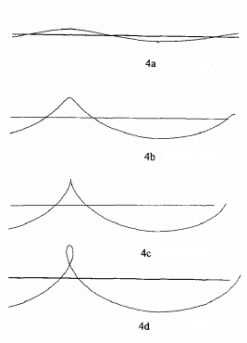

Fig. 4. Verschiedene Wellenformen, die sich aus Trochoiden ergeben. Ringbildungen (4d) müssen jedoch mit geeigneter Parameterwahl vermieden werden [9].

Generierung verwenden, sind komplexer zu implementieren, liefern aber bessere Ergebnisse. Wie in Fig. 5 zu sehen, bilden Gerstner Wellen ein erkennbares Muster, während Fourier Wellen (siehe Fig. 6) turbulenter sind und daher realistischer wirken. Außerdem ist ein sich wiederholendes Muster nicht erkennbar. Ein Vorteil an diesem Modell ist, dass

Fig. 5. Gerstner Wellen mit Phong Beleuchtung. Die Wellen erzeugen ein erkennbares Muster [1].

man viel Kontrolle über die Generierung der Wellen hat. Jedoch ist die Implementierung von Interaktionen mit dem Wasser schwierig [5]. Eine praktische Lösung liefert Nvidia mit ”WaveWorks”. Eine API für Filme und Spiele, die mit FFT Berechnungen, Wasseroberflächen anhand von Parameter, wie Windgeschwindigkeit und Windrichtung, generiert und unter anderem auch eine limitierte Interaktion zulässt [12].

B. Wasservolumen

Eine andere Herangehensweise ist die Betrachtung des Wassers als Volumen, welches aus vielen simulierten Par-

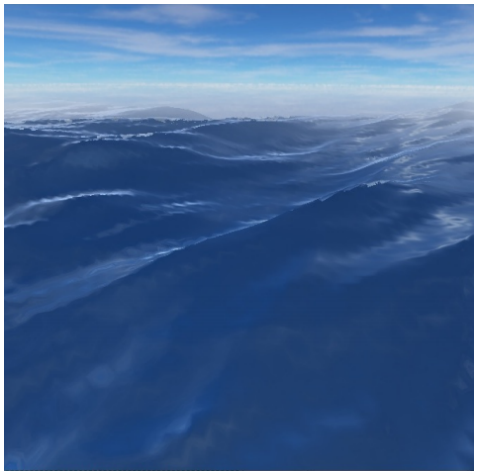

Fig. 6. Fourier Wellen mit Phong Beleuchtung. Die erzeugten Wellen sind turbulenter als Gerstner Wellen [1].

tikeln besteht. Solche Partikel basierende Methoden, wie zum Beispiel in [18] beschrieben, können nicht nur für Flüssigkeiten, sondern auch für Gase verwendet werden. Die bekannteste Methode hierfür nennt sich SPH (Smooth Particle Hydrodynamics), welche auf vereinfachte Navier-Stokes Gleichungen basiert, die in der Physik in der Strömungsmechanik Anwendung finden. Bei SPH kann sich jeder Partikel im Raum bewegen. Die Partikel stehen unter Einfluss von anderen Partikeln, Druck, Gravitation und anderen Faktoren. Besitzt eine Physik Engine bereits eine Partikelsystem-Komponente, ist SPH relativ einfach zu implementieren, jedoch ist die Performance nicht geeignet für eine Echtzeitsimulation. Des Weiteren muss für die Partikel zusätzlich eine Oberfläche berechnet werden, was sich, wie in [17] zu sehen, als schwierig erweist. Diese Herangehensweise eignet sich in Computerspielen daher hauptsächlich nur für geringe Mengen von Wasser [3].

III. Anforderungen

Aus den Arbeiten [3] und [4] ergeben sich fünf Hauptkriterien für eine Generierung einer Wasseroberfläche in Echtzeit: Performance, Glaubwürdigkeit, Grad der Benutzbarkeit, Robustheit und Speicherallokierung.

Performance ist das wichtigste Kriterium, wenn es sich um Echtzeitsimulationen handelt. In Offline-Simulationen, die für Voraussagungen beim Brückenbau oder in Animationfilmen verwendet werden, ist dies nicht wichtig. Ein realistischeres Ergebnis ist wichtiger, weswegen man bei diesen Anwendungen, Wasservolumen mit einem zeitintensiveren SPH Verfahren simuliert, wobei die Berechnung eines Frames mehrere Minuten gehen kann. Diese Rechenzeit ist, zum Beispiel für den Zuschauer eines Films, nicht sichtbar, da die Frames im Nachhinein, zu einer Videodatei mit 24 Bildern pro Sekunden zusammengebaut wird. Anders ist dies in Computerspielen. Um eine Bildwiederholungsrate von 60 bzw. 30 Bilder pro Sekunde zu erreichen, stehen 16 ms bzw. 33 ms zur Verfügung. Von diesem Renderbudget, steht nur ein Bruchteil, der Wassersimulation zur Verfügung. Der Rest wird für die Berechnung von Audio, Netzwerk, AI, Spieler Eingaben, Animationen, Kollisionsabfragen, Weltaktualiserung, etc. verwendet. Die Generierung muss also möglichst schnell sein und dabei wenigstens die Glaubwürdigkeit der Oberfläche maximieren. Eine simple vordefinierte Fläche, ist daher oft nicht wünschenswert. Bei komplexen Spieleentwicklungen ist natürlich auch die Robustheit und der Grad der Benutzbarkeit ein wichtiges Kriterium. Es dürfen keine Sprünge in der Simulation vorkommen oder unerwartetes, merkwürdiges Verhalten, welche das Spielerlebnis trübt. Das kann zum Beispiel passieren, wenn der Zeitschritt der Simulation, nicht mit der fixen Bildwiederholungsrate des Spiels nachkommt, weil es plötzlich zu unerwarteten hohen Werten in der Simulation kommt. Des Weiteren ist es für einen Entwickler wichtig, wenn das Verfahren, einfach in das Spiel eingebaut und konfiguriert werden kann, wie zum Beispiel die API "WaveWorks" von Nvidea. Und schlussendlich stehen Spielekonsolen und Heimrechner, weniger Arbeitsspeicher zur Verfügung, als groß angelegte Serverfarmen, die für das Rendern eines Animationfilms verwendet werden, weshalb die Speicherallokierung der Wassersimulation möglichst effizient sein muss.

IV. Height Field Fluid nach Matthias Müller

Ein Nachteil, bei den Verfahren für die Generierung einer Oberfläche für Ozeane, ist, dass nur der Effekt simuliert wird und nicht die Ursache. Das macht es schwieriger, eine Interaktion mit dem Wasser zu gewährleisten und Parameter für die Konfiguration zu finden, die auf physikalischen Gegebenheiten fundieren. Die Height Field Fluid Methode greift dazu das Konzept von Newtons zweiten Grundgesetz der Bewegung auf, was eine Interaktion mit der Oberfläche ermöglicht. Die Idee ist eine kontinuierliche Funktion $u(x, y, t)$ zu finden, die die Höhe der Wasseroberfläche beschreibt. Für die Implementierung, definiert man die diskrete Variante der Funktion als 2D-Array $u[i, j]$. Diese Idee hat außerdem den Vorteil, dass, das Problem eine 3D-Oberfläche zu beschreiben, auf ein 2D-Problem reduziert wird. Dies führt potentiell zu einer besseren Performance. Der Nachteil jedoch ist, dass keine brechenden Wellen dargestellt werden können, weil man für eine Position, mindestens drei Höhenwerte bräuchte.

Eine vollständige Beschreibung, einer vollwertigen Implementierung dieser Methode, mit tieferen Detailbeschreibungen und größerem Umfang, kann in der Arbeit [2] von Matthias Müller gefunden werden. Die Beschreibungen in diesem Abschnitt und dessen Unterabschnitten, lehnen sich stark an den SIGGRAPH Kurs über "Fluid Simulation for Computer Animation" aus dem Jahr 2007 [19] und der Präsentation über "Fast Water Simulation for Games Using Height Field" [5] von der Game Developer Conference 2008 an.

A. Einleitung

Für die Erklärung des Algorithmus ist es geeignet eine kurze Einleitung, am Beispiel einer eindimensionalen Welle, zu geben, die an den Hintergrund und die Beschreibung des Algorithmus heranführt.

Wie schon beschrieben, speichert der Algorithmus die Höhe der Wasseroberfläche für jede Position (x, y). Für die Bewegung einer Welle, wie sie im nächsten Abschnitt erklärt wird, ist jedoch noch die Information von Geschwindigkeit und einer Kraft von Nöten. Und weil die Implementierung des Algorithmus ein diskretes Problem ist, ist die Breite einer Welleneinheit notwendig. Visualisiert ergibt sich das Bild wie in Fig. 7.

Die Höhe einer Welleneinheit ist u, die Breite h und die Geschwindigkeit, wie sich eine Welle in der Höhe ändert ist v. Randbemerkung: Die Geschwindigkeit ist nur vertikal definiert. Das bedeutet, um Wellen erzeugen zu können, muss für eine Wasserzelle zum Start, eine, zu den anderen Zellen abweichende, Höhe oder Geschwindigkeit definiert werden. Die Wellen fallen also mit Zeit wieder in eine flache Lage zurück. Dies ist der Grund wieso keine Wellenerzeugung unterstützt wird, die zum Beispiel vom Wind herbeiführt.

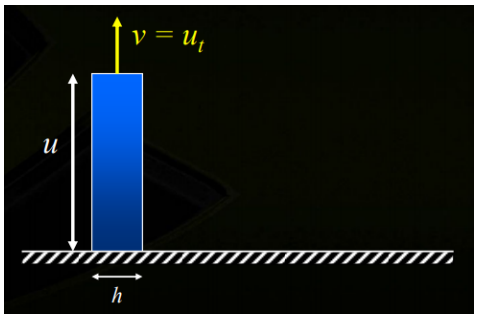

Fig. 7. Visualisierung der Begriffsdefinitionen. Abgebildet ist eine Welleneinheit mit der Höhe u, der Höhengeschwindigkeit v und der Zellenbreite h [5].

Die Kraft steht in direkten Zusammenhang mit der Geschwindigkeit und ist das, was das Wasser schlussendlich zum bewegen bringt. Bei der Methode von Müller spielt nur die vertikale Kraft eine Rolle, wie in [19] physikalisch weiter beschrieben ist. Sie ist, wie man im nächsten Abschnitt sehen wird, gegeben durch die Höhenänderung einer Welleneinheit in Bezug auf die x-Achse - im eindimensionalen Fall. Würde man diese Höhenänderungen visuell darstellen, ergäbe sich eine positive und eine negative gekrümmte Kurve (siehe Fig. 8, grün und rot markierter Bereich).

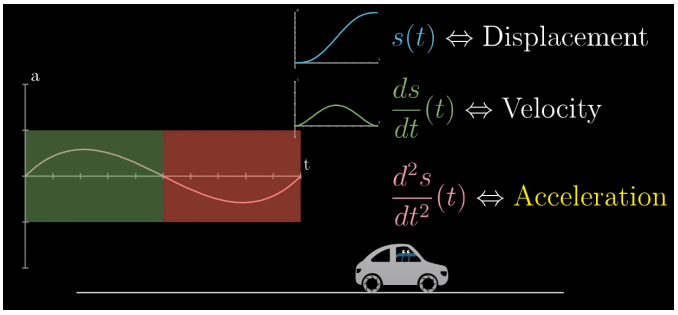

Fig. 8. Allgemeines Beispiel zum Zusammenhang der ersten und zweiten Ableitung. $s(t)$ beschreibt die Entfernung die ein Auto zurückgelegt hat. s_t beschreibt die Geschwindigkeit und s_{tt} die Beschleunigung in Bezug auf die Zeit [20].

Ist die Krümmung positiv, nimmt die Höhenänderung zu, ist sie negativ, nimmt sie ab. D.h. sie ist proportional zu der Krümmung der Wellenkurve. Im diskreten Fall sähe dies für eine negative und positive Beschleunigung, wie in Fig. 9 aus. Bleibt die Neigung konstant, existiert keine Geschwindigkeitsänderung und dementsprechend gibt es keine Kraft, wie in in Fig. 10.

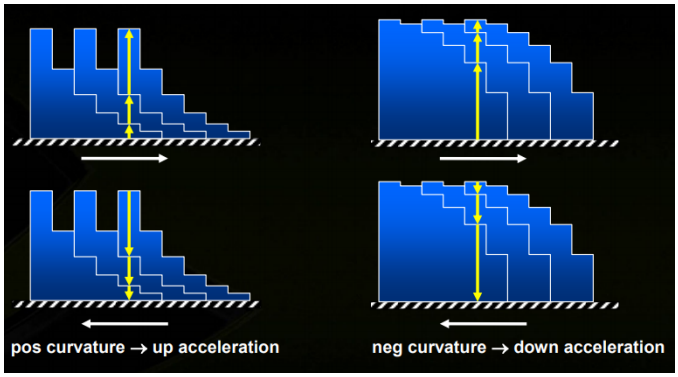

Fig. 9. Links im Bild: positve Kurvenkrümmungen, die eine zunehmende Beschleunigung der Höhengeschwindigkeit bedeuten. Die obere Welle verschiebt sich entlang der x-Achse nach rechts, die untere Welle nach links. Rechts im Bild: analog entgegensetzt wie links. Die gelben Pfeile deuten die Änderung der Höhe an einer fixierten Stelle an. Sie repräsentieren die positive oder negative gekrümmte Kurve, wie in Fig. 8 [5].

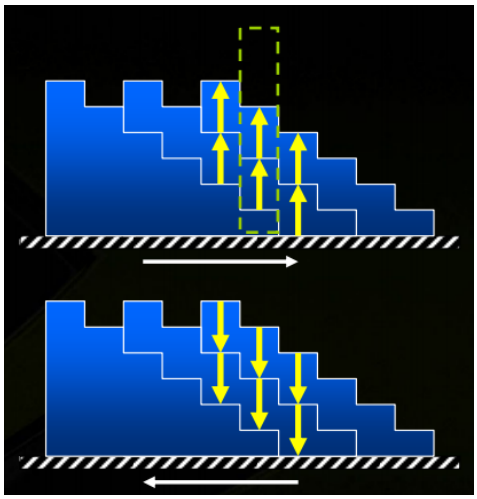

Fig. 10. Eine Kurvenkrümmung existiert nicht, so auch die vertikale Kraft. Dies repräsentiert den Wendepunkt der zweiten Ableitung bzw. den Sattelpunkt in der ersten Ableitung in Fig. 8 [5].

B. Hintergrund

Als Grundlage für die Height Field Fluid Methode liegt die zweidimensionale Wellenfunktion

$$u_{tt} = c^2 \cdot \nabla^2 u \tag{1}$$

in Form einer partiellen Differentialgleichung zweiter Ordnung vor, welche die Bewegung eines Membrans oder einer Wasserwelle beschreibt. u ist unserem Fall eine Funktion für die Höheninformation einer Welle an der Stelle (x, y). c ist die

Geschwindigkeit der Wellenausbreitung und $\nabla^2 u$ entspricht der Summe der zweiten Ableitungen in Bezug auf x und y (u_{xx} bzw. u_{yy}). Prinzipiell versteckt sich in dieser Gleichung Newtons Gesetz

$$f = m \cdot a \tag{2}$$

welches, umgeformt auf a, die Änderung der Geschwindigkeit per Zeiteinheit wiedergibt, was in unserem Fall u_{tt} entspricht - die Änderung der Höhe im Raum, abhängig von der Zeit t. Die Beschleunigung entspricht daraufhin f/m, was der rechten Seite aus (1) entspricht, wobei f dem Ausdruck $\nabla^2 u$ entspricht und die Form, der Änderung der Höhengeschwindigkeit in Bezug auf die x-Achse, hat. $1/m$ kann mit c^2 ersetzt werden (siehe hierzu [5]). Die Wellenfunktion ist deshalb wichtig, weil für die gesuchte Funktion $u(x, y, t)$ eine Funktion gebraucht wird, die beschreibt, wie die Höhenfelder sich über die Zeit entwickeln. Dies ist mit (1) gegeben.

Ab dieser Stelle wird der Einfachheit halber das Problem im eindimensionalen Fall weiter beschrieben. u_{xx} wird im diskreten eindimensionalen Fall über u_x der Nachbarzellen mit

$$u_{xx}[i] = \frac{u_x[i + \frac{1}{2}] - u_x[i - \frac{1}{2}]}{h} \tag{3}$$

berechnet. u_x der Nachbarzellen ergibt sich aus

$$u_x[i - \frac{1}{2}] = \frac{u[i] - u[i-1]}{h}$$
$$u_x[i + \frac{1}{2}] = \frac{u[i+1] - u[i]}{h}$$

Visuell ergibt sich das Bild in Fig. 11. u_{xx} ist also ein Indikatorwert dafür, ob die Krümmung der Beschleunigung der Höhengeschwindigkeit positiv oder negativ ist.

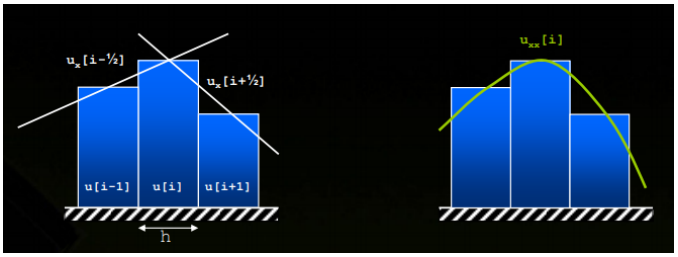

Fig. 11. Links im Bild: Die Ableitungen u_x der Nachbarzellen. Rechts im Bild: Die zweite Ableitung u_{xx} von u in Bezug auf x. Die grün markierte approximierte Kurve entspricht der Kurve der Geschwindigkeit, die zu den Tangenten auf der linken Seite passt [5].

Die Gleichung (3) lässt sich durch Einsetzen weiter vereinfachen zu

$$u_{xx}[i] = \frac{u[i+1] - 2u[i] + u[i-1]}{h^2} \tag{4}$$

Für den zweidimensionalen Fall ergibt sich

$$u_{xx}[i] = (u[i+1, j] + u[i-1, j] + u[i, j+1] + u[i, j-1] - 4u[i, j])/h^2$$

C. Der Algorithmus

Die Schritte des Algorithmus basieren auf dem Prinzip der Berechnung mit Partikel. Ein allgemeines Beispiel ist in Fig. 12 zu sehen und in Algorithmus 1 zu lesen.

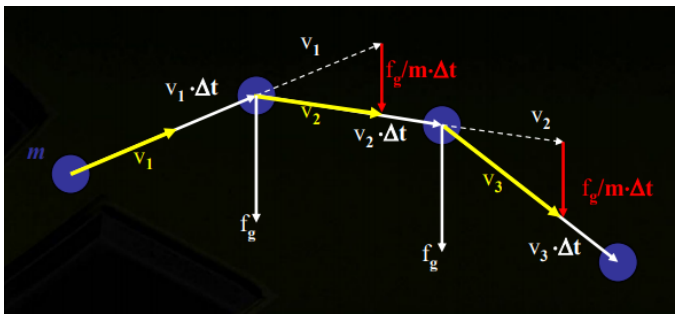

Fig. 12. Allgemeines Beispiel zum Prinzip der Berechnung für eine simple Bahnbewegung eines Paritkels [5].

Algorithm 1 Pseudocode für eine simple Bahnbewegung eines Paritkels [6]

```
loop
    compute force
    velocity += force/mass * time step
    position += velocity * time step
end loop
```

Übertragen auf den Anwendungsfall ergibt sich der Pseudocode in Algorithmus 2

Algorithm 2 Pseudocode für den Height Field Fluid Algorithmus für den eindimensionalen Fall [6]

```
for all i do
    f = c**2 * (u[i-1] + u[i+1] - 2u[i]) / h**2
    v[i] = v[i] + f * dt
    uNew[i] = u[i] + v[i] * dt
end for
for all i do
    u[i] = uNew[i]
end for
```

Zuerst berechnet wird die Kraft, in Form der Änderung der Höhengeschwindigkeit u_{tt}, dann die Geschwindigkeit einer Welleneinheit und letztendlich die neue Höhe einer Welleneinheit. Die Position ist in diesem Fall die Höhe der Welleneinheit. Da während der Schleife auf die Werte der Wellenhöhen im vorherigen Schritt zugegriffen werden muss, wird temporär ein neuer Array $uNew$ befüllt, mit dem man anschließend den Array u, mit den aktualisierten Werten überschreibt. Der Algorithmus hat die drei folgenden Variablen, die sich verändern lassen: c, dt und h. Der angegebene Algorithmus ist für den zweidimensionalen Fall geeignet, wenn man den Teil aus (4) mit dem dort darunter stehenden zweidimensionalen Fall ersetzt.

Eine Eigenschaft dieses Algorithmus ist es, dass nur eine Höhenänderung einer Zelle den anderen umliegenden Zellen

weitergegeben werden kann. D.h wenn man einen ganzen Bereich in einem Schritt platt drücken würde, so das kein Wasser an dieser Stelle ist, weil man es zur Seite drückt, dann hätte dies einen Wasserverlust zur Folge. Dies liegt an der Berechnung der neuen Höhe anhand der Nachbarzellen. So entstehen die Konditionen für die Wellengeschwindigkeit $c = h/dt$ bzw. für den Zeitschritt $dt = h/c$.

Bis zu diesem Punkt fehlen dem Algorithmus noch zwei kritische Aspekte. Der eine ist die Behandlung der Randbedingungen. Ist die *for* Schleife am Anfang oder am Ende des Arrays, kann nicht auf die Stelle $u[i-1]$ bzw. $[u+1]$ zugegriffen werden. Hier empfiehlt sich entweder den Wert zu limitieren, also den Wert $[u-1]$ auf $u[i]$ zu setzen, was zu einer Reflexion der Welle führt, oder den Anfang- bzw. Endwert des Arrays zu nehmen, aus $[u-1]$ wird somit $u[uLaenge-1]$, was die Welle am anderen Ende durch laufen lässt.

Der andere Aspekt ist, dass mit dem oben genannten Algorithmus, Wellen endlos bestehen bleiben. In der Realität flachen Wellen jedoch mit der Zeit ab. Dieses Problem wird in dem Modell, welches Matthias Müller benutzt hat, nicht berücksichtigt, weshalb er vorschlägt, die Wellen nachträglich zu dämpfen. Es gibt mehrere Möglichkeiten dazu. Die simpelste Methode, aber unphysikalisch, ist die Geschwindigkeit einer Welleneinheit mit einem Faktor s herunter zu skalieren, so das sich $v = s \cdot v$ ergibt. s muss deshalb kleiner als 1 und größer als 0 sein.

D. Erweiterung des Algorithmus

Dem Algorithmus, wie im letzten Abschnitt beschrieben, fehlen noch Komponenten, um in einem Spiel potentiell verwendet werden zu können. Diese werden folgend kurz beschrieben.

Objektinteraktion: Eines der wichtigsten Elemente einer Wasseroberfläche, um den Spielspaß zu erhöhen, ist die Interaktion von Objekten mit dem Wasser. Hier gibt es zwei verschiedene Szenarios: ein Objekt kann auf der Wasseroberfläche liegen, oder darin versinken. Das erstere betrachtet man wiederum aus zwei Blickwinkeln. Der erste: das Objekt beeinflusst das Wasser, indem es die Wassereinheiten nach unten drückt. Die Implementierung ist einfach. Man addiert die Wasserhöhe, die entfernt wurde, den Nachbarzellen hinzu. Der zweite: das Wasser beeinflusst das Objekt, indem das Wasser das Objekt wieder nach oben drückt. Hier muss für die darunterliegenden Wassereinheiten eine neue Kraft mit

$$f = -\Delta u \cdot h^2 \cdot \rho \cdot g \tag{5}$$

berechnet werden, wobei Δu die Wasserhöhe ist, die vom Objekt entfernt wurde, ρ die Wasserdichte und g die Gravitation.

Das zweite Szenario betrachtet Objekte die vollständig in dem Wasser versunken sind. Hier wird ein neuer Array $r[i,j]$ eingeführt. Dieser speichert die Höhe, die derzeit abgedeckt ist durch das Objekt. Bei jedem Zeitschritt wird nun die Differenzen der Stellen im Array $r[i,j]$, zu denen im letzten Zeitschritt $t-1$ berechnet mit

$$\Delta r[i,j] = r^t[i,j] - r^{t-1}[i,j] \tag{6}$$

Diese Werte werden anschließend auf die Nachbarzellen im Array u verteilt. Das bedeutet, dass falls sich das Objekt im Wasser bewegt, wird vor dem Objekt eine positive Welle erzeugt und hinter dem Objekt eine negative Welle. Der Effekt sollte anschließend noch anhand der Tiefe, in die das Objekt gesunken ist, skaliert werden. So dass man den Effekt nur bei Objekten hat, die sich nahe der Oberfläche befinden.

Offene Grenzen: In dem Fall, dass man die Wasseroberfläche nicht in einem kleinen abgeschlossenen Areal platzieren möchte, sondern in einem potentiell endlosen Areal, muss mit den Randbedingungen gesondert umgangen werden. Zum Beispiel kann man sich vorstellen, dass man einen großen Behälter mit Wasser und einem Tor hat. Öffnet man nun das Tor, möchte man, dass das Wasser abfließt und nicht reflektiert. Um das zu erreichen, fügt man bei der Grenze eine Geisterzelle g, welche nur die Höheninformation speichert, hinzu, und mit

$$g_{new} = \frac{c \cdot \Delta t \cdot u + g \cdot h}{h + c \cdot \Delta t} \tag{7}$$

aktualisiert wird. Die Height Field Fluid Oberfläche bleibt aber trotzdem stehen, weswegen man um den Behälter, zum Beispiel mit einer prozedural generierten Wasseroberfläche, umschließen muss.

Irreguläre Areale: Möchte man zum Beispiel eine Z-förmige Fläche mit einer Wasseroberfläche abdecken, so muss auch hier gesondert umgegangen werden. In diesem Fall teilt man die Fläche in Kacheln auf, die jeweils ein Height Field Fluid enthält. Die Grenzzellen, bei denen das Wasser auf ein Hindernis stößt, markiert man wie gewohnt als reflektierend, und beim Ende des Spiellevels, markiert man die Zellen wie im letzten Absatz beschrieben, als offen. Bei allen Situationen, wo das Terrain nicht vertikal die Grenze bildet, sondern andersartig geformt in das Wasser hinein läuft, wie bei einem Strand, lässt man die Wassersimulation naiv über den Strand laufen. Der Effekt, dass das Wasser auf den Strand hoch fließt, fehlt damit jedoch. Die Wasseroberfläche wird dabei einfach abgeschnitten.

Horizontale Bewegung: Objekte die sich auf oder unter dem Wasser befinden, bleiben, mit dem gezeigten Algorithmus, im Wasser horizontal stehen. Das liegt daran, dass es keine horizontale Kraft in dem Algorithmus gibt. Ein schneller Hack, wie ihn Matthias Müller beschreibt, ist die horizontale Kraft mit

$$v_h[i] = (v[i] - v[i-1])h + (v[i+1] - v[i]) \cdot h \tag{8}$$

zu berechnen. Er weist aber darauf hin, dass es eine bessere Alternative gibt, die momentan jedoch Nvidias Betriebsgeheimnis bleiben soll.

V. Implementation

Die Implementierung setzt den Algorithmus in der eindimensionalen Variante um, ohne die genannten Erweiterungen. Es wird für die Darstellung WebGL benutzt. Außerdem werden die externen Bibliotheken *m3* und *webgl-utils* von *webglfundamentals.org* verwendet. *m3.js* bietet Funktionen für

Matrixmultiplikationen an und *webgl-utils.js* bietet Funktionen für das Laden von Shadern und das Erstellen von einem Shaderprogramm Objekt, welches die Shader verlinkt. Der Fragment und Vertex Shader befinden sich in der *index.html*. Der eigentliche Algorithmus befindet sich in der Methode *stepWave()* und das Zeichnen der Wassereinheiten geschieht in der Methode *drawColumns()*. Für eine flüssige Darstellung wird *drawColumns()* mit *requstAnimationFrame* kontinuierlich aufgerufen. Die Vertex Daten für den ArrayBuffer werden in *setGeometry()* gesetzt und verwenden normalisierte Koordinaten, um eine einfache Größenveränderung der Wassereinheiten durch eine Skalierung gewährleisten zu können. Der Ursprungpunkt des Canvas ist oben links.

Es ist möglich sämtliche Parameter einzustellen, die der Algorithmus zu bieten hat (siehe Fig. 13). Ungültige Eingaben werden entsprechend behandelt. Parameter werden übernommen und gegebenenfalls korrigiert nachdem man ”Apply” drückt.

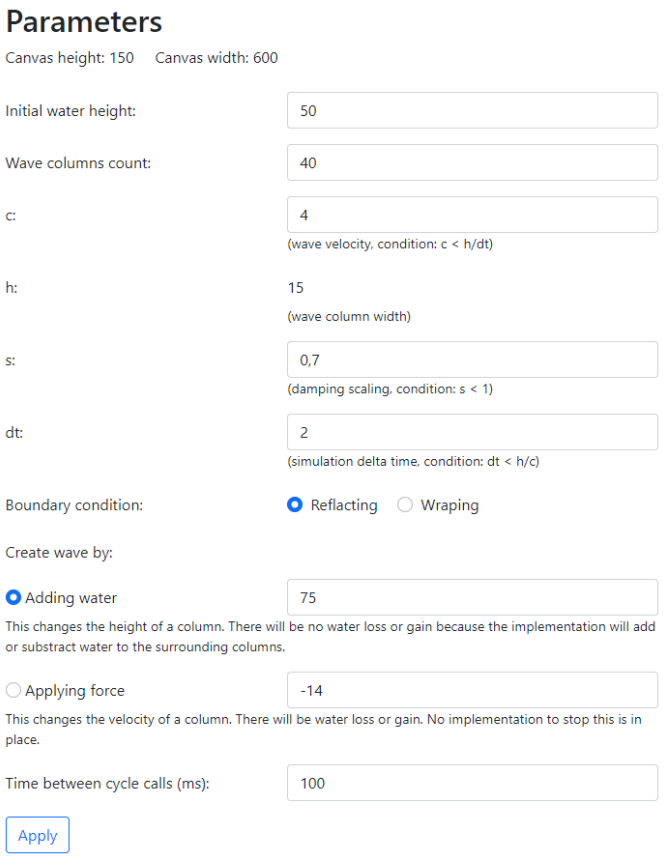

Fig. 13. Darstellung der einstellbaren Parameter der Implementierung.

Um mit der Simulation interagieren zu können, klickt man eine Wassereinheit an, wodurch eine Welle erzeugt wird. Die Wassereinheiten werden rot gefärbt, sobald man mit der Maus darüber fährt. Die Simulation wird bei einem Klick sofort in einer endlos Schleife gestartet. Stoppen kann man die Simulation anschließend mit ”Stop”. Mit ”Cycle” kann man bei Bedarf einzelne Schritte des Algorithmus durchführen. Die Debug Ansicht, die mit einem Switch eingeschaltet werden kann, zeigt den momentanen Zustand des u und v Arrays. Beim Erstellen einer Welle hat man zwei Möglichkeiten. Man kann die Welle erzeugen, indem man Wasser hinzufügt, wodurch der Höhenwert einer Wassereinheit verändert wird, oder, indem man die Geschwindigkeit einer Wassereinheit verändert. Beim ersteren wird gewährleistet, dass kein Wasserverlust oder -zunahme stattfindet, indem der hinzugefügte Wert bei den Nachbarzellen gleichverteilt hinzugefügt wird. Dies geschieht bei der zweiten Variante nicht. Aufgrund dessen das man den Ausgangszustand ändert, ergibt sich daraus folgend ein Wasserverlust bzw. -zunahme in Bezug auf die initiale Wasserhöhe. Die Darstellung der Simulation wird in Fig. 14 dargestellt.

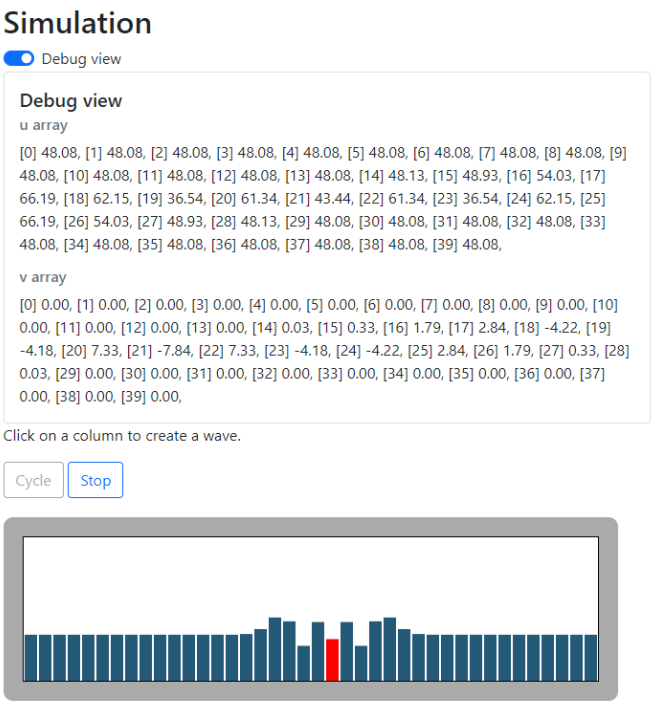

Fig. 14. Darstellung der eindimensionalen Wassersimulation.

VI. Fazit

Wie zu sehen ist, ist der Height Field Fluid Algorithmus von Matthias Müller sehr einfach zu Implementieren und erzeugt dafür gute Ergebnisse. Er bietet gute Performance, indem er nur die zweidimensionale Wasseroberfläche berechnet, im Gegensatz zu den Partikel basierenden Verfahren wie SPH, und bietet ein realistischeres Ergebnis durch die Interaktion mit dem Wasser, die bei den prozedural generierten Oberflächen nicht vorhanden ist. Er bildet somit das Mittelstück zwischen den prozedural generierten Oberflächen und den Oberflächen, die mit Partikelsystemen berechnet werden. Auf eine Betrachtung des Renderings der Simulation wurde, wie bereits in der Einleitung erwähnt, verzichtet. Entsprechende Shader, Texturen, Partikeleffekte, etc. sind jedoch für eine spätere potentielle Anwendung in einem Videospiel notwendig. Mögliche

Artefakte die dabei entstehen können, wie in [2] beschrieben, wurden deshalb nicht genannt.

Eine eindimensionale Implementierung ohne die genannten Erweiterungen, wurde mit WebGL demonstriert. Eine solche Implementierung eignet sich gut, um das Prinzip des Algorithmus zu verstehen, zu testen und ein "Proof of Concept", für eine mögliche zweidimensionale Implementierung in einer dreidimensionalen Welt, zu geben. Die eindimensionale Implementierung läuft stabil, jedoch fehlt noch eine Dämpfung für große spitze Wellen, wie sie Müller unter anderem vorgeschlagen hat. Erzeugt der Nutzer eine große Anzahl von Wasserzellen in einer fixen Umgebung, wie sie durch die Canvas breite gegeben ist, wird die Breite h der Wasserzellen sehr klein. Im Programm wird eine Welle dadurch erzeugt, dass die Höhe oder die Kraft, nur einer Zelle geändert wird. Ist diese Änderung groß genug, werden die Ableitungen u_x auch sehr groß, wodurch die Simulation instabil wird und es zu einer numerischen Explosion kommt.

REFERENCES

[1] Christ, Vera. "Simulation und Visualisierung von Wasseroberflächen.", 2014.

[2] Chentanez, Nuttapong, and Matthias Müller. "Real-time Simulation of Large Bodies of Water with Small Scale Details." Symposium on Computer Animation, 2010.

[3] Kellomäki, Timo. "Water simulation methods for games: a comparison." Proceeding of the 16th International Academic MindTrek Conference, 2012.

[4] Gritzner, Daniel. "Simulation of Liquid Surfaces for Games.", 2010.

[5] Matthias Müller-Fischer, "Fast Water Simulation for Games Using Height Fields", https://www.gdcvault.com/play/203/Fast-Water-Simulation-for-Games, GDC 2008, 2008, 11.12.2020

[6] Belyaev, Vladimir. "Real-time simulation of water surface." GraphiCon-2003, 2003.

[7] Truelsen, Rene. "Real-time shallow water simulation and environment mapping and clouds.", 2007.

[8] Fleck, Bernhard. "Real-time rendering of water in computer graphics." ,2007.

[9] Fournier, Alain, and William T. Reeves. "A simple model of ocean waves." Proceedings of the 13th annual conference on Computer graphics and interactive techniques, 1986.

[10] Hinsinger, Damien, Fabrice Neyret, and Marie-Paule Cani. "Interactive animation of ocean waves." Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation, 2002.

[11] Qiaoyu, Li, and Meng Xiangxu. "A simple method to simulate waved water." 2011 International Conference on Image Analysis and Signal Processing. IEEE, 2011.

[12] NVIDIA Corporation, "NVIDIA WaveWorks", https://developer.nvidia.com/waveworks, 14.12.2020

[13] Flügge, Fynn-J. Realtime GPGPU FFT ocean water simulation. Universitätsbibliothek der Technischen Universität Hamburg-Harburg, 2017.

[14] Tessendorf, Jerry. "Simulating ocean water." Simulating nature: realistic and interactive techniques. SIGGRAPH 1.2 (2001): 5.

[15] Habib, "Water Mathematics", https://habibs.wordpress.com/water-mathematics/, 14.12.2020

[16] Weisstein, Eric W., "Trochoid", https://mathworld.wolfram.com/Trochoid.html, 14.12.2020

[17] Müller, Matthias, Simon Schirm, and Stephan Duthaler. "Screen space meshes." Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation, 2007.

[18] Premžoe, Simon, et al. "Particle-based simulation of fluids." Computer Graphics Forum. Vol. 22. No. 3. Oxford, UK: Blackwell Publishing, Inc, 2003.

[19] Bridson, Robert, and Matthias Müller-Fischer. "Fluid simulation: SIGGRAPH 2007 course notes" ACM SIGGRAPH 2007 courses, 2007.

[20] Sanderson, Grant. "Ableitungen höherer Ordnung — Essenz der Analysis, Kapitel 10", https://www.youtube.com/watch?v=BLkz5LGWihw, 2017, 20.12.2020.

[21] Tiptur Ravindra, Thrilok. "Simulation of water using opengl/jogl." (2019).