Welleninterferenz

Das Handout zum Applet

Von Mitja Stachowiak
Für Physik
Klasse bg 13.4, Heinrich Emanuel Merck Schule, Darmstadt
April & Mai 2012

Die Übersicht:

1 Überblick

Dieses Handout beschreibt die Ziele und Probleme bei der Entwicklung des Applets zur Welleninterferenz. Das Applet soll in der Lage sein, elektromagnetische Wellen mit unterschiedlichen Frequenzen im 2D-Bereich darzustellen. Die Berechnung soll nicht das Ziel haben, besonders realitätsnahe zu sein, sondern lediglich ausreichen, um bestimmte Phänomene und Versuche zu veranschaulichen, von welchen einige sind:

- Wellenkrümmung hinter kleinen Objekten
- Überlagerung mehrerer Wellen
- Reflexion an dünnen Schichten
- Geschwindigkeitsreduktion und Strahlbeugung in dichteren Medien
- Absorption und Reflexion

Als Compiler verwende ich Lazarus. Dieser ist komplett Open Source, unkompliziert überall installierbar und das Programmieren mit Object Pascal ist spürbar übersichtlicher, als mit C++. Die meisten anderen Sprachen sind nicht ausreichend entwickelt, für andere Zwecke vorgesehen und es gibt keine Header für DirectX, und ähnliche Techniken, die ich unter Umständen brauchen werde. Aber auch für Pascal gibt es nicht alle Header.

2 Zeitplan und grobe Struktur des Programms

Als ich das Projekt angenommen habe, dachte ich, ich hätte mehr Zeit, aber dann kamen haufenweise andere Referate und Präsentationen dazwischen, was den Zeitplan stark unter Druck gesetzt hat. Fest steht: Berechnungen mit Wellen wollen gut durchdacht sein. Wenn ich nicht volle Konzentration habe, kann ich folglich kaum weiter arbeiten. Damit sind Schultage quasi aus dem Zeitplan zu streichen.
Dennoch bleiben derartige Berechnungen auch für den Computer eine schwierige Aufgabe. Kreisförmig ausbreitende Wellen sind nicht weiter schwierig. Jedoch kann man mit einem Modell, das Wellen als Kreise oder Streifen darstellt kaum Reflexionen und Krümmungen darstellen. Deswegen fiel meine Entscheidung auf ein Feld-Zu-Feld-Prinzip, ähnlich der Finiten Elemente Methode.
Wählte man als Grundlage hier ein Raster aus Sechsecken, hätte man eine deutlich bessere Qualität, als mit Vierecken, aber die notwendigen Formeln für Sechsecke wären einfach zu kompliziert für diesen Zeitplan. Deswegen verwende ich viereckige Felder. Die Wellen werden als Teilvektoren eingetragen, welche je die Informationen für Wellenlänge, Amplitudenhöhe, Ausbreitungsrichtung und Phasenwinkel tragen. Mit 8 und 16 Bit großen Werten komme ich so auf 8 Byte pro Vertex. Wollte man im Falle der Reflexion an dünnen Schichten auf einer 400 x 300 Felder großen Karte mit hochauflösenden Spektren arbeiten, könnte man den nötigen Speicher wie folgt abschätzen: 400 x 300 x 2 (wegen doublebuffered) x 6 (Durchschnittliche Anzahl wellenlängengleicher Vertices mit unterschiedlichen Richtungen pro Feld) x 100 (Anzahl unterschiedlicher Frequenzen) x 7 (Größe eines Vertex'). So käme man auf gut 1 GB Arbeitsspeicher, welcher mit jedem Frame komplett von der CPU bearbeitet werden müsste. Für die Erkennung von Wellenfronten ist es notwendig, für jedes Feld auch die Darumliegenden abzufragen – eine Wellenfront zu „verfolgen“ um diesen Aufwand zu minimieren erscheint eine kaum lösbare Aufgabe, besonders im Hinblick auf das Multithreading. Ohne Unterstützung der Grafikprozessoren ist es also nicht denkbar, bei dieser Art von Simulation auf Maps mit sehr vielen Quellen und unterschiedlichen Frequenzen in angemessener Zeit sinnvolle Ergebnisse zu erzielen. Bislang habe ich noch nicht intensiv mit Cuda oder OpenCL gearbeitet, weswegen die Zeit dafür nicht reichen wird. Das Applet wird wohl oder übel auf umfangreiche Darstellungen im Frequenzspektrum verzichten müssen.
Für die Darstellung werden die Wellen in einer Textur, welche der Auflösung nach der Mapgröße entspricht, als Schwarz-Weiß-Farbwerte dargestellt. Diese wird mit DirectX gerendert. Dadurch wird die Berechnung zumindest nicht von der Renderprozedur gestört. Der Nachteil ist, dass das Applet so von DirectX abhängig wird: Für die Ausführung muss DirectX 9.0c installiert sein. Die SDK-Version ist 33, es muss also die d3dx9_33.dll installiert sein und zusätzlich die entsprechende d3dx9.dll dem Programm beiliegen. Beides könnte als Resourcen enthalten sein, wodurch aber mindestens ein Speicher von 3MB verbraucht würde. Mit DirectX an Board gibt es aber die Möglichkeit, die Wellen später mal in einen Indexbuffer zu schreiben und als animierte Heightmap darzustellen. Außerdem geplant ist die Einbindung eines Video-Encoders, damit man rechenintensive Maps über Nacht als Video rendern lassen kann.

3 Aufbau des Sourcecodes

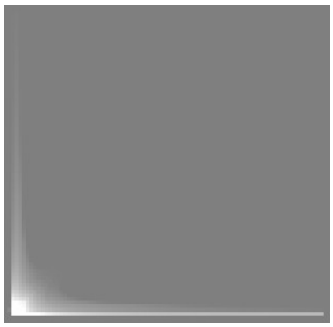
Das Applet wird, wenn es fertig ist, im wesentlichen aus zwei Komponenten bestehen: Einem Map-Editor und dem eigentlichen Rechenprogramm. Letzteres werde ich als erstes fertig stellen. Dieses Handout wird jedoch nur die Entwicklung bis zum Prototyp 0.6 erklären.
Die eigentliche Rechenkomponente des Simulators steht in der Datei WaveSimulation.pas. Diese enthält die Klasse TWaveSim, welche alle notwendigen Funktionen enthält:

CurrentTex	Pointer auf die aktuell renderbare Textur vom Typ IDirect3DTexture9
ShownWaveAmp	Darstellungsempfindlichkeit: Felder, deren Amplitude diesen Wert erreicht werden komplett weiß, bzw. schwarz dargestellt
ShowType	Dargestellter Inhalt. Neben 0 (= Wellen) kann auch die Speicherauslastung oder die Anzahl signifikanter Vertices eines Feldes dargestellt werden
onStop	Ein Methodenzeiger (procedure od object). Die hier eingestellt Methode wird immer dann aufgerufen, wenn die Simulation nach dem Aufruf von BreakCalc beendet ist.
onStepReady	Ein Methodenzeiger (procedure of object). Die hier eingestellte Methode wird nach jeder Berechnung eines Frames aufgerufen. In dieser kann dann z.B. der Aufruf zum Rendern erfolgen.
constructor create	Erwartet die zu berechnende Map, sowie das von der Anwendung erstellte DirectX-Device
procedure StartCalc	Startet die Berechnung
property IsWorking	Ist true, wenn die Berechnung gerade läuft
procedure BreakCalc	Bricht die Berechnung nach dem nächsten Frame ab
procedure Free	gibt die Klasse frei

4 Von der Idee zum Programm

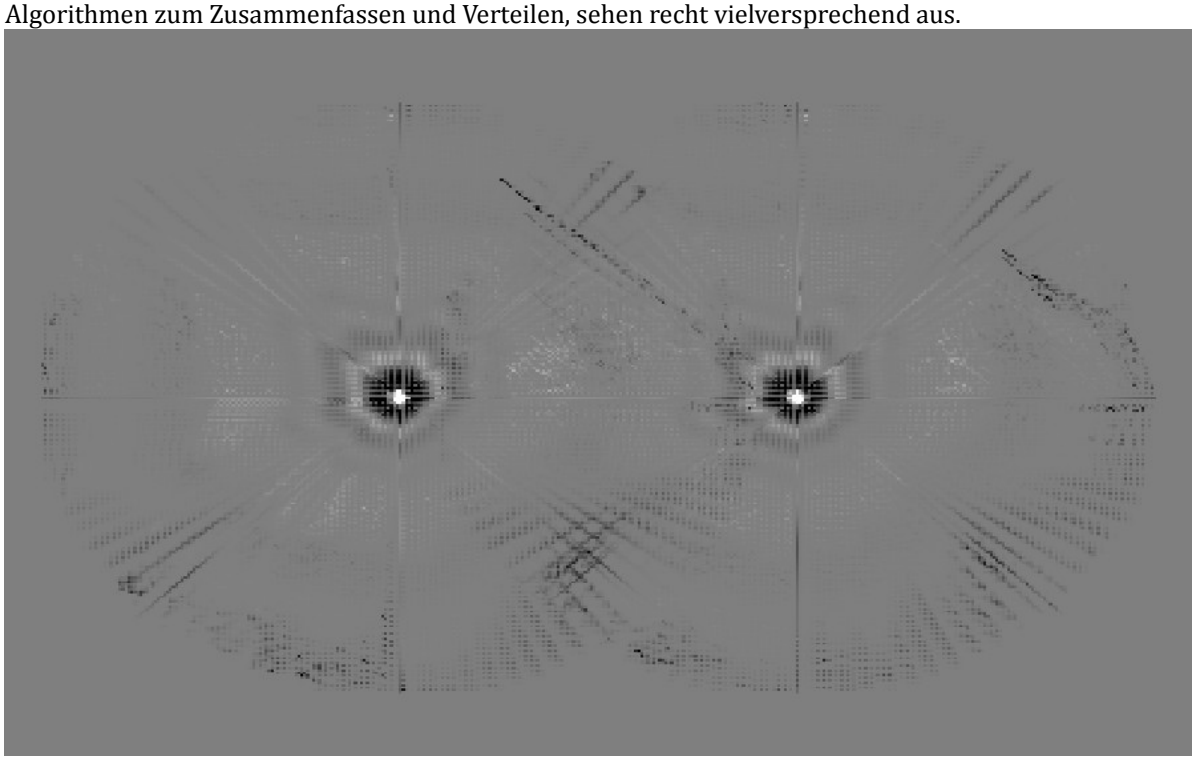
4.1 Version 0.1

Meine erste Idee war es, die Wellenvektoren von Pixel zu Pixel weiterzugeben. Wenn sich eine Welle mit der Geschwindigkeit „1“, was in dieser Simulation die Höchstgeschwindigkeit ist, von links nach rechts bewegt, müssen folglich mit jedem Rechendurchgang (Frame) alle Vertices dieser Welle ein Feld nach Rechts verschoben werden. Wenn sich aber eine Welle diagonal nach rechts oben bewegt, bleibt von jedem Vertex ein Teil im aktuellen Feld übrig, denn sonst wären die Wellen ja eckig und nicht kreisförmig. Dieser Teil sollte dann im nächsten Durchgang verschoben werden.
Im Bezug auf das Multithreading war diese Idee sehr einfach, weil ich so einfach nur jede Reihe der Map für einen Thread zu sperren brauchte, in welcher nun die neuen Vertices angelegt würden. Die umliegenden Reihen konnte dieser Thread abfragen, um zu prüfen, von wo aus neue Vectices in das aktuelle Feld „schwappen“ würden. Da so aber oftmals mehrere Threads den gleichen Speicher auslesen, brachte diese Methode den kleinen Nachteil mit sich, dass der Prozessor mit 10% Hardwareinterrupts behindert wurde.
Der Eigentliche Denkfehler an diesem Konzept ist aber, dass jedes Feld einzeln behandelt wurde: Wenn sich eine Welle diagonal ausbreitet, fällt im ersten Durchgang z.B. 60% in das jeweils neue Feld, während 40% im Aktuellen verbleiben. Diese werden aber nicht im nächsten Durchgang weiter verschoben, sondern es bleiben wieder 16% zurück, 24% (60% von den 40%) fallen in das nächste Feld plus 24%, die in diesem Feld verbleiben und nur 36% fallen in das übernächste Feld. Durch diese Hyperbelförmige Ausbreitung in diagonaler Richtung, kommt die Ausbreitung nach wenigen Feldern komplett zum Erliegen, wie in der aufsummierten Darstellung rechts zu sehen ist.



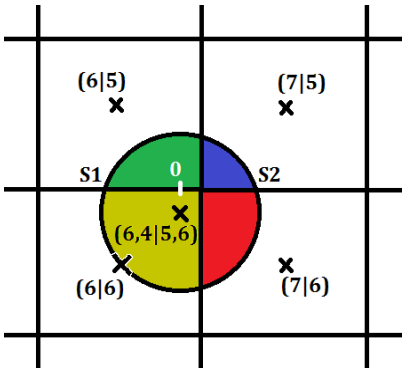
4.2 Version 0.2

Der nächste Ansatz war es, die Wellenvektoren aus je vier Feldern zusammenzufassen, zu verschieben und wieder auf die Felder zu verteilen. Die erste Schwierigkeit bei diesem Konzept war es, die Map für das Multithreading kompatibel zu machen. Denn nun musste ich umgekehrt arbeiten: Das Programm musste vier Felder auswählen, die Berechnung durchführen und dann die Vertices in die Felder schreiben, die eben bei dieser Berechnung heraus kommen. Diese können höchstens ein Feld entfernt sein, also lasse ich die Map wie bei einem Kamm abarbeiten: Es wird je ein Streifen mit vier Feldern für einen Thread gesperrt. Die darum liegenden Felder müssen nun ebenfalls gesperrt werden. Daher lasse ich die Map wie einen Kamm abarbeiten: Im ersten Durchgang werden die Streifen 0&1, 4&5, 8&9, usw. bearbeitet, im zweiten Durchgang dann die übersprungenen Streifen 2&3, 6&7, 10&11, usw.
Die Entwicklung dieses Konzeptes war durchaus knifflig, aber erste Ergebnisse, noch ohne vollständige Algorithmen zum Zusammenfassen und Verteilen, sehen recht vielversprechend aus.



4.3 Version 0.3

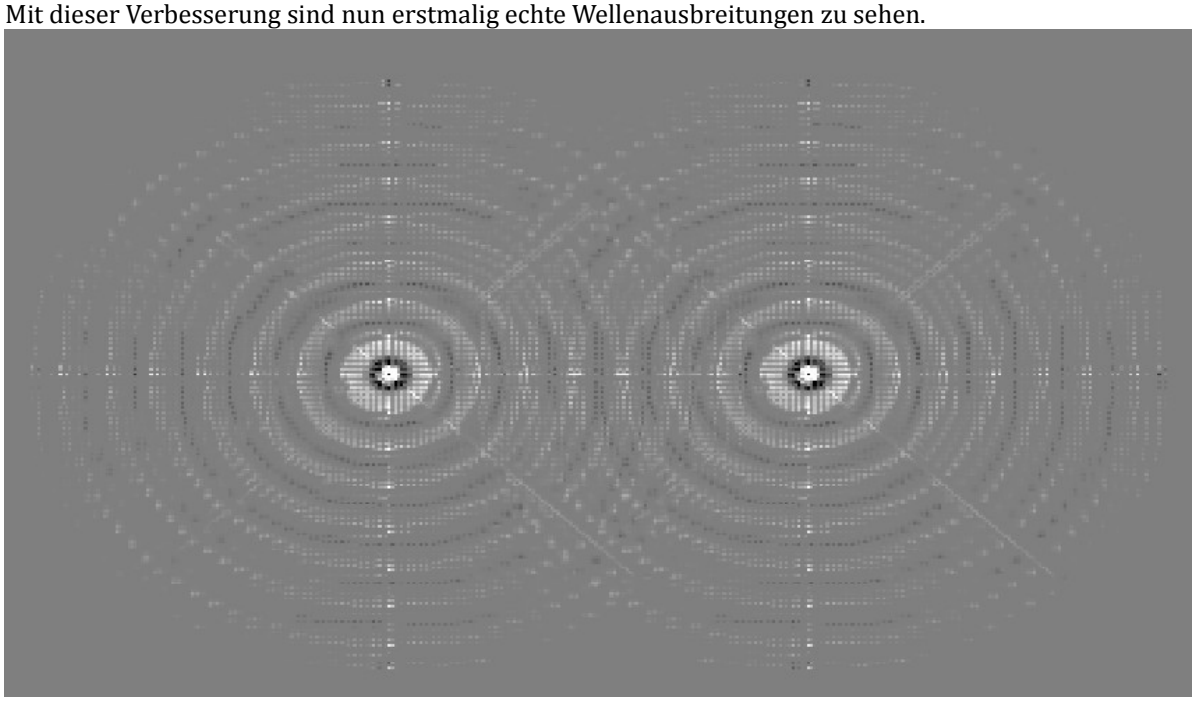
Als nächstes galt es, die Funktionen zum Verteilen einzuprogrammieren. Wird ein zusammengerechneter Wellenvektor auf die Position (6,4|5,6) geschoben, so fallen dessen Anteile in die Felder (6|5), (7|5), (6|6) und (7|6). Dies geschieht über das Integral der Kreisfunktion. Die Ausdehnung des Vertex' wird als Kreis angenommen. Nun ist die Frage, zu welchen Teilen ein Kreis mit dem Durchmesser „1 Feld“ sich bei nicht-natürlicher Position auf die umliegenden Felder verteilt.



Wie man erkennen kann, müssen die Funktionen des Integrals abschnittsweise definiert werden. Für die rote und die gelbe Fläche muss von der Stammfunktion der unteren Kreishälfte die der oberen Hälfte bis zum Schnittpunkt S1 bzw. S2 abgezogen werden. Die grüne und blaue Fläche bestehen in diesem Fall aus nur einer Stammfunktion, jedoch ist es möglich, dass z.B. die blaue Fläche für eine Menge von Positionen des Mittelpunktes Null wird, wenn der Kreis noch weiter unten links liegt. Neben den Stammfunktionen müssen auch die Nullstellen S1 und S2, sowie die Unterscheidungstabelle für die abschnittsweise Definition einprogrammiert werden.

$$\int \pm\sqrt{r^2-x^2}+d_y \; dx = \pm\frac{r^2\,asin\left(\frac{x}{r}\right)+x\sqrt{r^2-x^2}}{2}+d_y\cdot x \qquad S_{1,2}=\pm\sqrt{r^2-{d_y}^2}$$

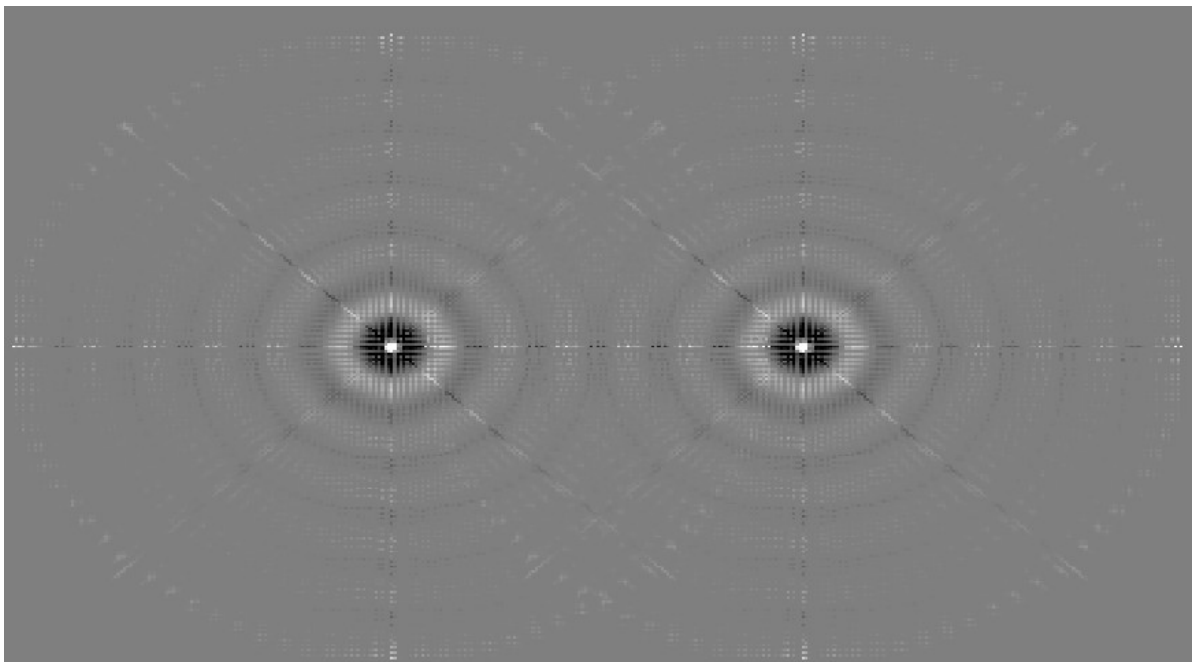
Da die oben dargestellten Funktionen sehr rechenintensiv sind, werden die Kreisanteile zum Start des Programms in einem zweidimensionalen Array gebuffert. Dies geschieht in der Funktion CreateSquare. Mit dieser Verbesserung sind nun erstmalig echte Wellenausbreitungen zu sehen.



4.4 Version 0.4

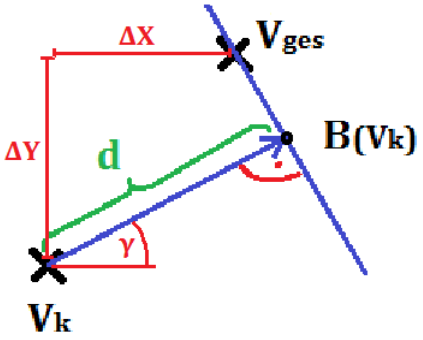
Nun mussten die Vertices noch nach den richtigen Formeln zusammengefasst werden. Die Position des zusammengefassten Vertex' ergibt sich aus dem Schwerpunkt der verteilten Vertices, wenn deren Amplitudenhöhe eine Masse wäre. Zusammengefasst werden immer alle wellenlängengleichen Vertices aus den vier zu bearbeitenden Feldern, deren Richtungsvektordifferenz eine gesetzte Grenze unterschreitet. Amplitudenhöhe und Phasenwinkel des zusammengefassten Vertex' ergeben sich aus der komplexen Addition der Teilvertices. Sei A die Amplitudenhöhe und P der Phasenwinkel:

$$A_{ges} = \sqrt{\left(\sum_{1}^{n} A_n \cdot \cos(P_n)\right)^2 + \left(\sum_{1}^{n} A_n \cdot \sin(P_n)\right)^2} \qquad P_{ges} = arg\left(\sum_{1}^{n} A_n \cdot \cos(P_n) + i \sum_{1}^{n} A_n \cdot \sin(P_n)\right)$$



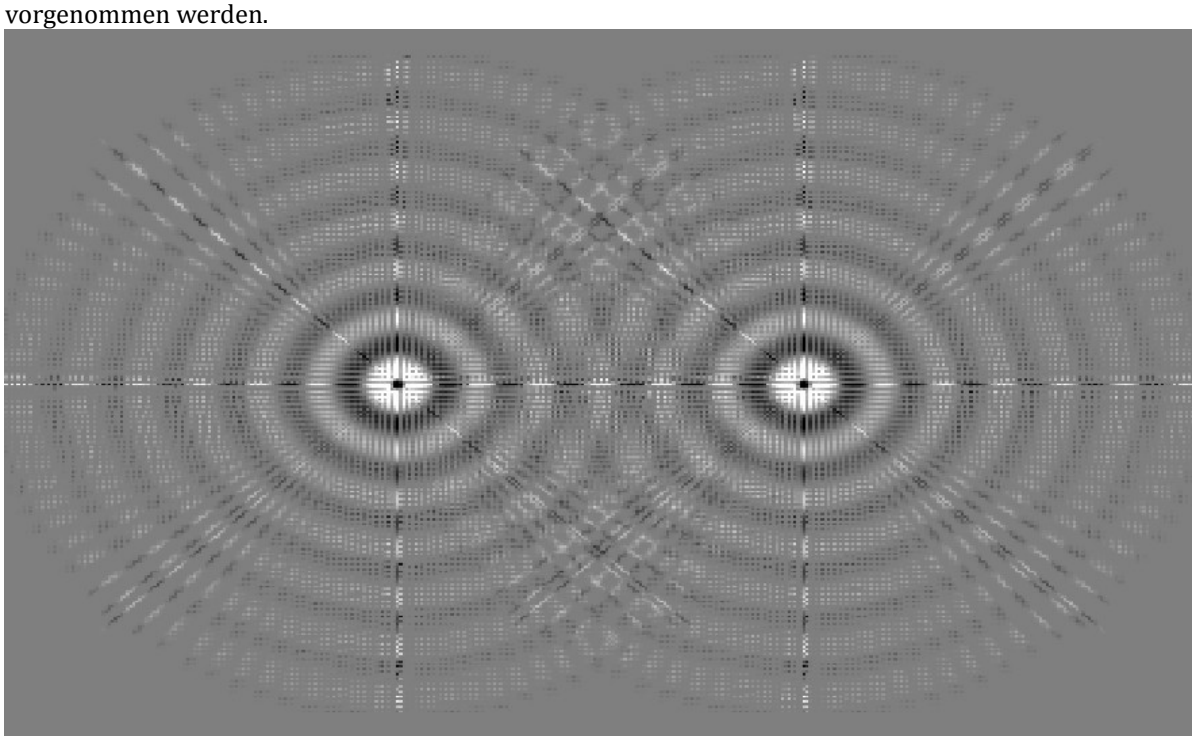
4.5 Version 0.5

In der letzten Version war eine erhebliche Abschwächung der Amplitude zu beobachten. Das liegt daran, dass die Teilvertices Räumlich ein kleines Stück auseinander liegen. Das führt dazu, dass auch die Vertices der gleichen Wellenfront leicht unterschiedliche Phasenwinkel haben. Bei der Addition entsteht so jedes mal eine Abschwächung. Die Lösung hierfür ist es, die Phasenwinkel vor der Addition zu justieren, womit ich meine, dass man, nachdem die Position des zusammengefassten Vertex' bekannt ist, den Phasenwinkel aller Teilvertices so anpasst, als befänden sich diese an der Position des neuen Vertex'. Um diese Verschiebung zu berechnen, müssen Wellenlänge und der Verschiebungsweg bekannt sein. Letzterer verläuft immer parallel zum Richtungsvektor. Daher könnte man alle Entfernungen berechnen, indem man die Vertices mit einer Matrix multipliziert, die alle Bildpunkte auf kürzestem Weg auf die Orthogonale zum Richtungsvektor verschiebt und die Entfernung von Ursprungsvektor und Bildpunkt berechnet. Da ich aber noch ohne GPU arbeite, habe ich die dafür notwendigen Rechnungen in ein Paar Formeln zusammen gefasst. Sei P_k der anzupassende Phasenwinkel, λ die Wellenlänge und γ die Ausbreitungsrichtung:



$$P_k := P_k + \frac{\sqrt{\Delta X^2 + \Delta Y^2} \cdot \cos(\gamma - arg(\Delta X + i\Delta Y))}{\lambda}$$

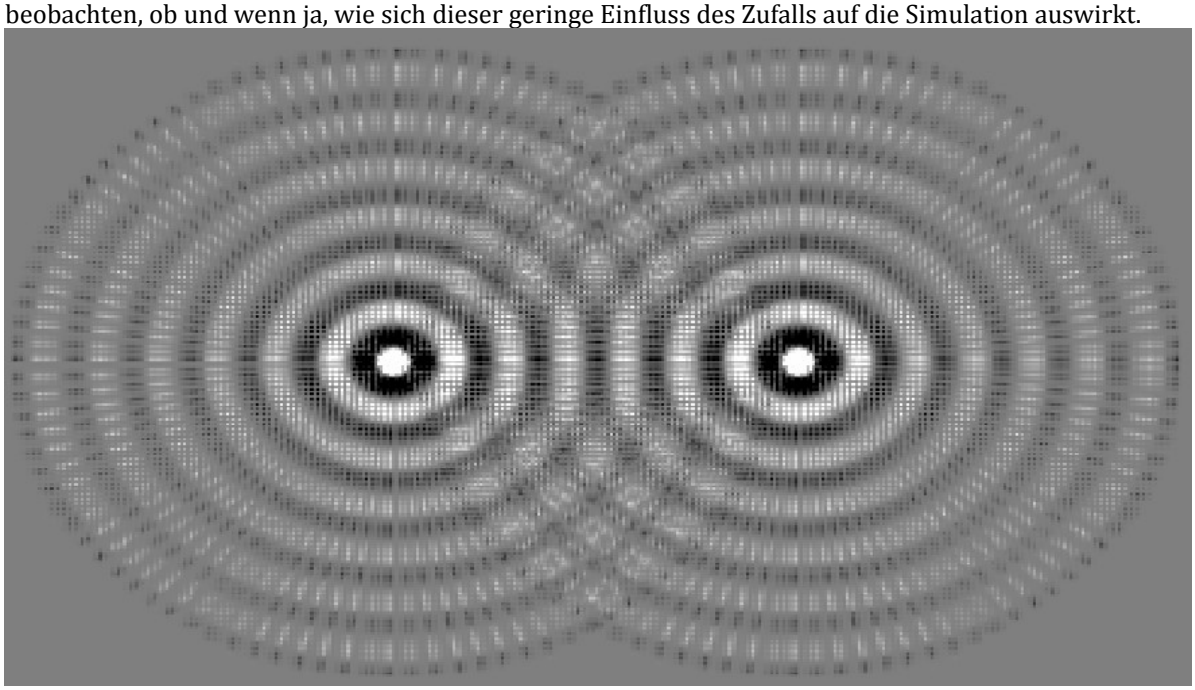
Diese Justierung muss beim Zusammenfassen und beim Verteilen in umgekehrter Richtung vorgenommen werden.



4.6 Version 0.6

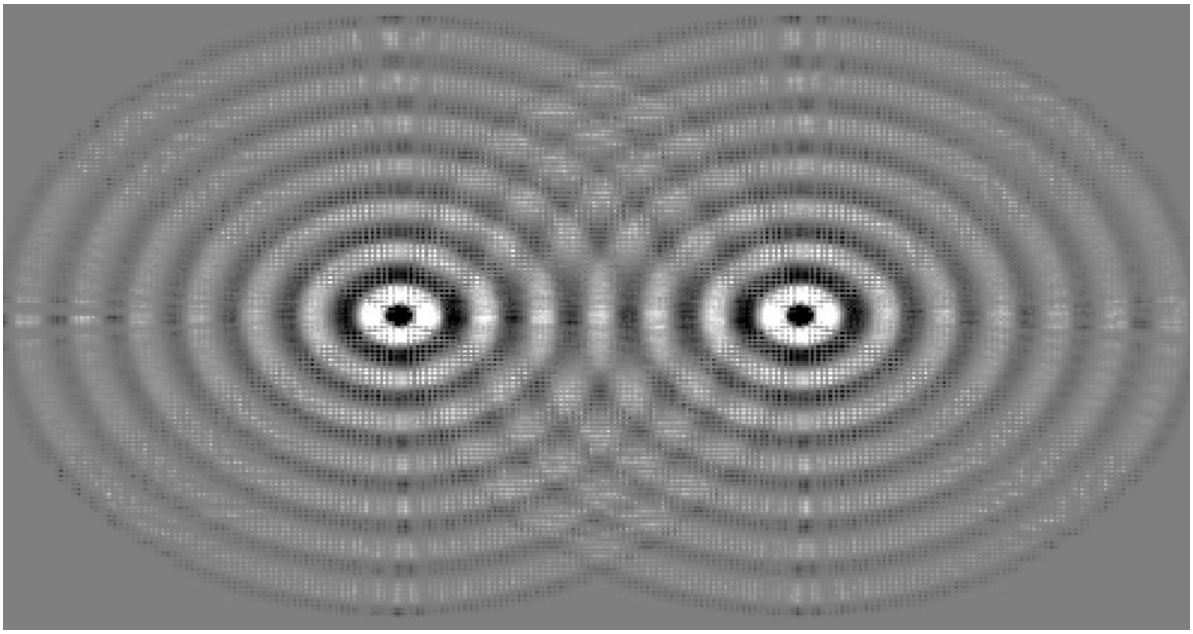
In der letzten Version war eine deutliche Streifenbildung und eine Körnung der Wellen zu beobachten. Die Streifen entstehen, weil die Wellen nur mit einer Auflösung von 100 Vektoren in unterschiedliche Richtungen ausgestrahlt werden. Die hierfür notwendigen Algorithmen müssen dann die Wellenfronten erkennen und die verteilten Vertices entsprechend ausrichten. In dieser Version widme ich mich aber zunächst dem Unterdrücken der Körnung. Warum diese entsteht, ist mir nicht ganz klar. Vielleicht sind die Algorithmen zum Verteilen noch fehlerhaft, vielleicht entsteht sie aber auch einfach durch die Rechenungenauigkeit. Eine erste Lösung hierfür habe ich mit zwei Erweiterungen vorgenommen:
Zunächst habe ich einen Glättungsalgorithmus eingebaut, der beim Aufteilen des zusammengesetzten Vertex' die Amplitudenhöhen etwas angleicht, besonders bei horizontaler und vertikaler Ausbreitungsrichtung.
Als nächstes habe ich den Vier-Felder-Kamm durch eine Art Rüttelsieb ersetzt. Das heißt, es werden

nicht immer wieder die gleichen vier Felder berechnet, sondern mit jedem Frame wird eine Bias geändert, die die komplette Berechnung um ein Feld verschiebt oder eben nicht. Die Mapgröße im Speicher muss so in Länge und Breite ein Feld größer werden. Ein kleiner Nachteil, den man zwar beheben könnte, der aber auch nicht wirklich stört, ist, dass die erste Spalte und die erste Zeile der Map nicht jedes mal berechnet wird. Wichtig bei diesem Verfahren ist, dass sich die Bias nach einem Zufallsalgorithmus ändert. Denn ein kontinuierliches Abwechseln verschiebt die Körnung von der Horizontalen bzw. Vertikalen in die Diagonale, wie ich feststellen musste. Nur mit einem Zufallsalgorithmus wird sie gleichmäßig verteilt. Man könnte hier zum Beispiel einen Mersenne-Twister verwenden, der in jeder Simulation die gleiche Sequenz erzeugt, aber ich finde es gerade spannend, zu beobachten, ob und wenn ja, wie sich dieser geringe Einfluss des Zufalls auf die Simulation auswirkt.



4.7 Version 0.7

Als nächstes habe ich mich mit der Problematik der Streifenbildung beschäftigt. Die hierfür vorgesehenen Glättungsalgorithmen müssen in einem kleinen Bereich die Wellenfront erkennen und prüfen, ob sich diese Ausbreitet, gerade verläuft oder zusammen läuft, etwa nach der Reflexion an einem Hohlspiegel. Diese Erkennung ist noch nicht fertig, aber in Planung und wird deswegen am Ende des Handouts näher beschrieben. In dieser Version habe ich zunächst einen einfachen Algorithmus eingebunden, der die aufgeteilten Vertices deren Richtung nach leicht auseinander laufen lässt.
Das Ergebnis sieht zumindest nach dem richtigen Ansatz aus.



5 Weitere Planung und Fazit

Als nächstes gilt es, die Aufsplittung der Richtungen der aufgeteilten Vertices speziell nach deren Umgebung anzupassen. Als nächstes werde ich daher Algorithmen schreiben, die in einem Kleinen Bereich die Wellenfront erkennen. Da dann für jedes Feld die Darumliegenden abgefragt werden müssen, kann die Performance dadurch stark in den Keller gehen. Außerdem ist es schwierig, in diesem Felder-System die Wellenfronten zu erkennen – exakte Algorithmen müssten wieder nach der Zusammenfassungsmethode arbeiten, was aber zu viel Rechenaufwand bedeuten würde. Daher bleibt zu hoffen, dass das Ergebnis auch so zufriedenstellend sein wird.

Was dieses Projekt mir vor Augen geführt hat, ist, dass bei der Entwicklung von derartigen Simulationen die eigentlichen Formeln, welche man in Übungsaufgaben verwendet, verhältnismäßig wenig hilfreich sind. So etwa taucht die klassische Formel für Welleninterferenz, mit Weg, Zeit, Winkel und Amplitudenhöhe in dieser Form gar nicht auf. Für diese Berechnung ist es notwendig, Wellen mit unterschiedlichen Amplitudenhöhen zu überlagern, was nur mit der komplexen Rechnung möglich ist. Die Interferrenz von Wellen mit unterschiedlichen Richtungen und Frequenzen reduziert sich auf eine simple Addition – dafür aber muss man es erst einmal schaffen, kreisförmig ausbreitende Wellen zu erzeugen. Der geplante Glättungsalgorithmus darf jedoch bei Lasern nicht dazu führen, dass sich diese ebenfalls am Rand kreisförmig ausbreiten. Eine schwierige Aufgabe... Auch die Heisenbergsche Unschärferelation taucht nicht als Formel auf. Zwar wird die Krümmung von Wellen hinter dünnen Spalten damit begründet: Wenn ein Photon einen dünnen Spalt passiert hat, wurde dessen Position relativ exakt gemessen – es musste durch den Spalt. Dadurch verschiebt sich dessen Impuls und die Welle, die hinter dem Spalt entsteht, breitet sich von dort aus kreisförmig aus. Wie genau deren Intensität und Richtung aber über die Fläche verteilt in Abhängigkeit der Spaltbreite verläuft, kann man mit den normalen Formeln nur schwer herleiten. Ich werde daher versuchen, einen Algorithmus zu finden, der ungefähr die gewünschten Ergebnisse liefert.

Auf jeden Fall braucht man viel Zeit, für solche Simulationen.