# GPU-beschleunigte Webtechnologien für Scientific Computing and Visualization

Nils Hoyler

Institute for Parallel and Distributed Systems – Scientific Computing

**Abstract** Mit neuen Technologien im Browser erhalten Websiten immer mehr Zugang zu Funktionen, welche früher zwingend die Installation eines eigenständiges Programms erforderten. Diese Arbeit schaut sich dabei einerseits die Möglichkeiten an Scientific Computing and Visualization einer Gravitationssimulation mit WebGPU und WebGl im Browser zu machen, anderseits auch neben dem klassischen 2D bild durch WebXR direkt auch in VR (Virtuelle Realität) zu visualisieren.
Dabei stellt sich raus, dass der Funktionsumfang dieser Technologien für die meisten Scientific Computing Anwendungen nicht ausreichend ist, aber Scientific Visualization im Browser seine Berechtigung hat.

# 1 Introduction

Im Rahmen dieses Fachpraktikums haben wir uns mit Gravitationssimulation mit MPI und deren Visualisierung mit ParaView beschäftigt. Darauf aufbauend soll nun eine ähnliche Funktionalität mit Webtechnolgien umgesetzt werden. Da werde ich aus Gründen der Performance die Grafikkarte für die Simulation und das Rendern der Visulasierung verwenden. Neben einer klassischen 2D-Ansicht bietet das System auch eine immersive Darstellung in Virtual Reality (VR) über WebXR, wodurch Nutzer das Geschehen dreidimensional erleben und interaktiv erkunden können. Ziel des Projekts war es, sowohl die technische Machbarkeit als auch das Potenzial von GPU-beschleunigte Webtechnologien zu beleuchten.

# 2 Technologien

## 2.1 WebGL und WebGPU

Für den Zugriff auf die Grafikkarte verwendet dieses Projekt entweder WebGL (Web Graphics Library) oder WebGPU. WebGL ist eine weit verbreitete JavaScript-API, die auf OpenGL ES basiert und hauptsächlich für die Darstellung von 2D- und 3D-Grafiken verwendet wird.[web]. Des weiteren können wir nur WebGl und nicht WebGPU verwenden, wenn wir mit WebXR die Simulation in VR visualsieren, da WebXR noch keine Bindings für WebGPU hat.
WebGPU ist der moderne Nachfolger von WebGL und basiert auf aktuellen Grafikstandards wie Vulkan, Metal und Direct3D 12 und wurde entwickelt, um die Leistung moderner Hardware besser auszuschöpfen. [BJ].
Um etwas allgemein etwas (im Browser) auf der Grafikkarte auszuführen, werden sogenannte Shader verwendet. Dies sind kleine Programme, welche hochspezialisiert auf der GPU ausgeführt werden können. Wir unterscheiden in dieser Arbeit zwischen vertex, fragment und compute shader. [AMHH19] Vertex und fragment shader sind dabei Teil


**Figure 1.** Einfache Darstellung einer Renderpipeline [AMHH19]

der Renderpipeline (siehe Figure 1). Diese beschreibt den Ablauf, wie Rohdaten aus 3D-Modellen – also Geometrie und Vertex-Daten – in ein fertiges Bild auf dem Bildschirm umgewandelt werden. Zunächst werden die 3D-Daten, bestehend aus Punkten (Vertices), Kanten und Flächen, an die GPU übergeben. In der ersten Stufe kommt der Vertex Shader zum Einsatz. Dieser berechnet für jeden einzelnen Punkt die genaue Position im Raum, indem er Transformationen wie Drehung, Skalierung oder Perspektivprojektionen durchführt. Das Ergebnis ist eine Position im sogenannten Clipping

Space – ein Koordinatensystem, das für die spätere Projektion auf den Bildschirm vorbereitet ist. Anschließend folgt die Rasterization. In diesem Schritt werden die geometrischen Formen – meist Dreiecke – in ein zweidimensionales Bild umgewandelt. Dabei entsteht für jede Fläche eine Reihe von Fragmenten, die später zu Bildpunkten (Pixeln) werden. Zu diesem Zeitpunkt existieren jedoch noch keine Farbwerte. Diese Farbwerte werden im nächsten Schritt durch den Fragment Shader berechnet. Er entscheidet für jedes Fragment, welche Farbe es haben soll, ob Texturen oder Lichtquellen berücksichtigt werden und ob Transparenz oder Schatten angewendet werden müssen. Das Ergebnis ist ein farbiges Pixel, das in das finale Bild übernommen wird.
Dahingegen ermöglichen Compute-Shader allgemeine Berechnungen auf der GPU abseits RenderPipeline. Deshalb verwenden wir diese für die Berechnung der einzelnen Zeitschritte unserer Simulation. Dort liest der Compute-Shader (siehe Listing 1) die aktuellen Position und Geschwindigkeiten, und berechnet mit der Masse daraus neue Position und Geschwindigkeiten, welche er dann ausgibt. Compute-Shader werden aber nur von WebGPU nicht aber von WebGL unterschtützt. [AMHH19] Deshalb bauen wir uns in WebGL eine getrennte RenderPipeline, welche jeweils den nächsten Zeitschritt der Simulation berechnet. In WebGL gibt es die Möglichkeit mit einer RenderPipeline direkt auf eine Textur zu rendern, um beispielsweise dynamische Texturen oder Spieglungen zu ermöglichen. Wir nutzten das und codieren unsere Simulationsdaten in einer Texture, welche die für die Berechnung verwendete RenderPipeline erhält. Diese berechnet daraus dann "Farbwerte" (vier Floats), in welche wir jeweils die Geschwindigkeit oder Position encoden, einer neuen Textur. Dieser Workaround ermöglicht aber kein Update von Position und Geschwindkeit in einem Durchlauf, da für eine Instanz des fragment Shaders immer nur vier Floats ausgegeben werden können.

### 2.2 WebXR

WebXR ist eine Technologie, welche für das Tracking eines VR-Headsets und Controller übernimmt. Bei einer aktiven WebXR session, übergibt diese mir dann die zwei Kameras (eins für jedes Auge). Mit dieser Information erzeugen wir dann mit der Renderpipeline jeweils ein Bild für jedes Auge.

## 3 Implementierte Funktionen

### 3.1 Simulation auf der GPU

Da die Implementierung der Simulation in WebGPU sehr ähnlich zu der in WebGL ist, aber übersichtlicher, schauen wir uns im folgenden den verwendeten Compute Shader in WebGL an:

**Listing 1.** Compute Shader with WebGL for Simulation

```
1 struct Object {
2     pos: vec3<f32>,
```

```
3       mass: f32,
4       vel: vec3<f32>,
5       classID: f32,
6   };
7
8   @group(0) @binding(1)
9   var<uniform> constant: Const;
10  @group(0) @binding(2)
11  var<storage> objectsIn: array<Object>;
12  @group(0) @binding(3)
13  var<storage, read_write> objectsOut: array<Object>;
14
15  fn computeAcc(i: u32) -> vec3<f32> {
16      var acc = vec3<f32>(0, 0, 0);
17      var compensation = vec3<f32>(0, 0, 0);
18      for (var j: u32 = 0u; j < constant.numberOfObjects; j = j + 1u) {
19          if (j == i) { continue; }
20          let dir = objectsIn[j].pos - objectsIn[i].pos;
21          let bracket = (dot(dir, dir) + constant.epsilonSq);
22          let value = objectsIn[j].mass * inverseSqrt(bracket * bracket * bracket) * dir;
23
24          //kahan summation
25          //let y = value - compensation;
26          //let t = acc + y;
27          //compensation = (t - acc) - y;
28          //acc = t;
29          acc += value;
30      }
31      return constant.G * acc;
32  }
33
34  @compute @workgroup_size(WORKGROUP_SIZE)
35  fn computeMain(@builtin(global_invocation_id) global_id: vec3<u32>) {
36      let i = global_id.x;
37      objectsOut[i].vel = objectsIn[i].vel + computeAcc(i) * constant.dt;
38      objectsOut[i].pos = objectsIn[i].pos + objectsOut[i].vel * constant.dt;
39  }
```

Zuerst schauen wir uns an, wie die Daten dem Shader übergeben werden. Dabei fällt auf, dass wir single precision float und nicht double precision floats verwenden. Dies liegt daran, dass WebGPU diese nicht direkt unterstützt [gpu]. Man könnte, um die Genauigkeit von 64bits zu erhalten, seine eigene Operatoren implementieren, welche dann immer zwei 32Bit Zahlen als eine betrachten. Dies würde aber die Performance erheblich beeinträchtigen. Um aber etwas die numerischen Fehler zu verringern, ist es möglich die aus NumGL bekannte Kahan Addition zu verwende. Dieser Algorithmus hilft, falls viele kleine Beschleunigung sich zu einem großen Wert addieren. Bei unserem Input, war dies nicht notwendig, da die Sonne und der Zentralkörper meist den Großteil der resultierenden Beschleunigung ausmachen. Deshalb ist dieser Teil des Codes auch auskommentiert. Bei WebGl ist der Support für double precision floats nicht besser, so unterstützten die meisten Desktop-GPUs 32bit Floats in Texturen, aber manche GPUs, wie die in der Meta Quest 3, für das Rendern in Texturen sogar nur 16 Bit Floats.
Die Daten der Himmelskörper werden in einem Array an Structs gespeichert. Dabei

werden die Skalare und die Vektoren in ihrer Anordnung durchgemischt, um Padding zu vermeiden. Denn *vec3<f32>* hat ein Alignment von 16 Bytes und *f32* von 4 Bytes. In dem Shader gibt es zwei Storage Buffer für diese Himmelskörper. Dabei wird immer von einem der beiden gelesen und in den anderen geschrieben. Nach einem Zeitschritt tauscht dann die Richtung. Dieses Verfahren nennt sich Ping Pong Buffer und ermöglicht es dem Shader die Position schon zu updaten ohne eventuell parallel laufenden Berechnung der Beschleunigung anderer Objekte zu beeinflussen. Denn die Mainfunktion wird einmal pro Objekt möglichst parallel aufgerufen. In dieser wird die Beschleunigung mit dem naiven Verfahren, welches ein Laufzeit von $O(N^2)$ für die Anzahl an $N$ an Objekten, hat. Im Gegensatz zu der Implementierung der Leapfrog Integration aus Phase 1, update wir hier die Geschwindigkeit in einem Schritt. Das können wir machen, da sowohl die Geschwindigkeit als auch die kinetisches Energie nicht ausgeben werden und es somit nur eine Neuordnung der einzelnen Schritte ist. Durch dieses Umordnen spart man sich eine Synchronisierung der GPU-Threads, da der Zeitpunkt für diese dann auf das Ende der Shaderberechnung fällt, wo wir sowieso synchronisieren. [BJ]

### 3.2 Visualisierung der Simulationsdaten

Da wir in diesem Projekt den Vertex und Fragment Shader komplett selbst schreiben, haben wir die vollständige Kontrolle wie wir die Ergebnisse der Simulation darstellen. Pro Objekt wird eine Kugel gerendert, welche sich in Position, Farbe und Größe unterscheiden kann. Um diese große Anzahl effizient zu rendern, verwenden wir die Möglichkeit, sowohl in WebGPU als auch WebGL, mehrere Versionen des gleichen 3D-Objekts zu zeichnen. Dabei wird dann anhand des des Instanzindexes im Shader die entsprechende Position für das Objekt aus dem Storagebuffer der Simulation gelesen und dort gezeichnet. Somit werd das zeichnen der einzelnen Kugeln extrem gut parallelisiert und man benötigt auch nur einmal Speicher für die Geometrie des Objekts.
Neben der Position der Kugel, wird die Farbe verwendet, um die Klasse nach dem Abbildung in Table 1anzuzeigen. Die Größe der Kugel ist in der Visualisierung proportional zum Logarithmus der Masse, sodass kleine Objekte noch erkennbar sind, aber die Sonne nicht die ganze Visualisierung dominiert.
Wenn man die Implementierung in WebGL verwendet, dann ist wie in Figure 2 oben links in der Ecke ein Button, um die Visualiserung in VR anzuschauen. Wenn man diesen drückt und das eigene System WebXR unterstützt, sieht man die aktuelle Simulation mit dem aktuellen Zeitschritt in VR. Man kann diese auch einfach wider verlassen mit der von der Platform abhängigen Art Wenn man zum Beispiel den *Immersive Web Emulator* existiert bei dessen Steuerung ein Knopf dafür oben rechts.

| **Klasse** | **Farbe** |
|---|---|
| Stern | Gelb |
| Planet | Green |
| Zwergplanet | Cyan |
| Rest | helles Grau |

**Table 1.** Bedeutung der Farben der Himmelskörper

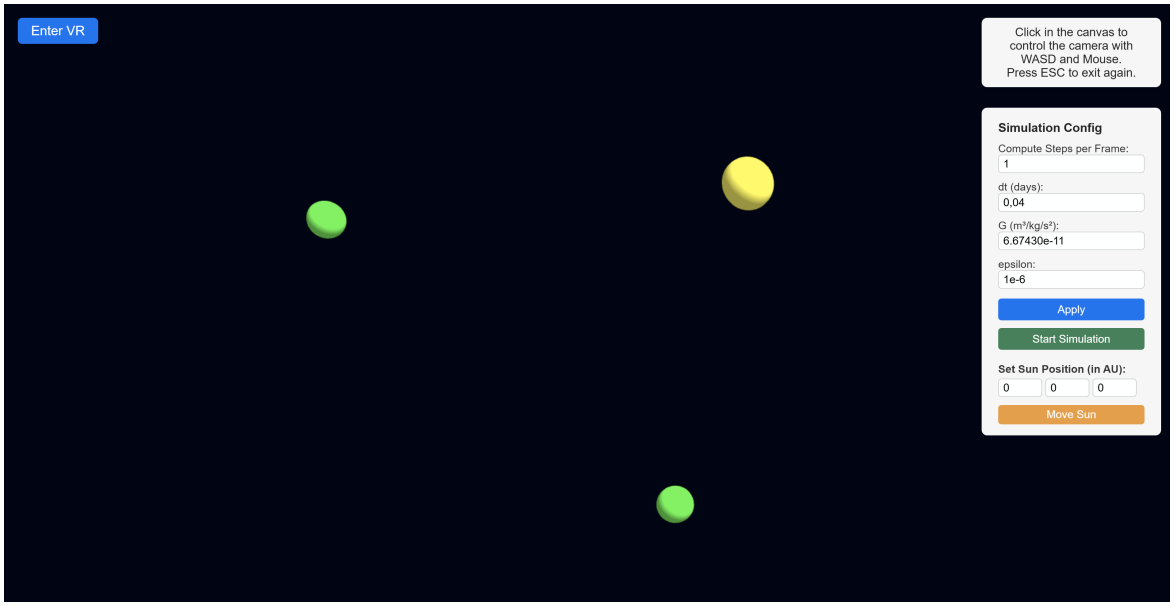


**Figure 2.** Visualiserung mit WebGL

### 3.3 Interaktivität der Simulation und Visualisierung

Da die Simulation meist in Echtzeit läuft, ist möglich diese Interaktiv zu gestalten. In diesem Projekt gibt es deshalb folgende Möglichkeiten zur Interaktion mit der Simulation:

- Navigation der Kamera mit WASD und Mouse oder in VR durch den Joystick auf dem linken Controller und die Blickrichtung
- das Starten und Stoppen der Simulation
- das Festlegen der Anzahl an Timesteps in der Simulation pro gerenderten Frame
- das Ändern der Simulationsparamter: G, dt, $\epsilon$
- das Bewegen der Sonne an eine andere Position

Diese können über das Menü auf der rechten Seite, welches man Figure 2 erkennen kann, geändert werden.

## 4 Analyse

### 4.1 Performance

Um die Qualität des Algorithmus zu beurteilen, sollte man die Performance der compute Steps genau messen könne. Dies ist insbesondere bei WebGPU nicht einfach genau zu bestimmen, da zum Beispiel timestamp queries nicht mehr unterstützt werden. Die Motivation dafür war es side channel attacks schwieriger zu machen und Funktionen

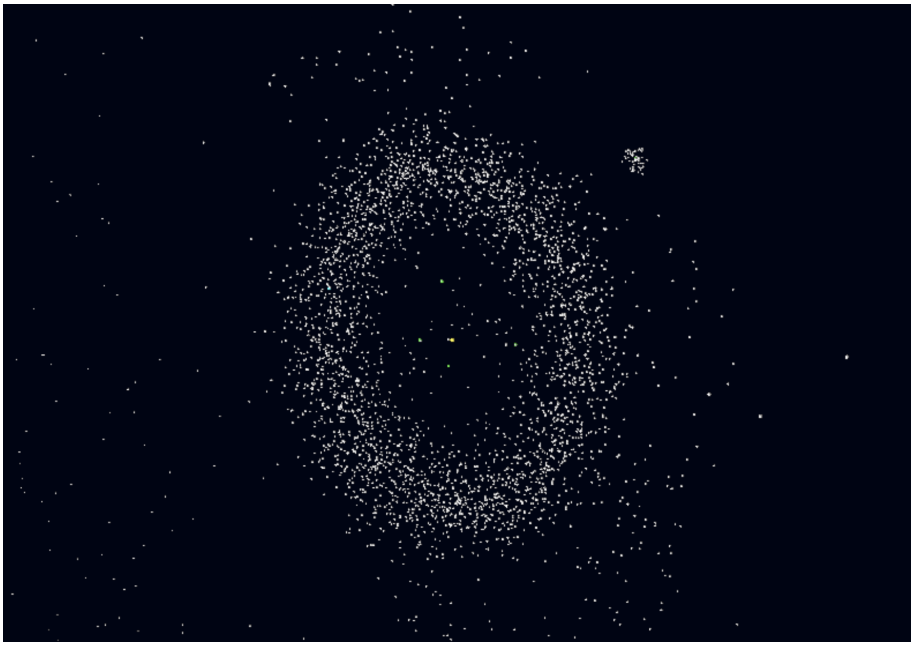

**Figure 3.** Visualiserung von 8000 Objekten mit WebGPU

zu entfernen, welche nicht alle GPUs unterstützen. Da der Algorithmus auch nicht der Fokus dieser Arbeit war, würde die Performance it dem sehr ungenaue Weg, welcher die visuell die Zeit zwischen den Frames mit einer großen Anzahl an Simulationschritten misst, bestimmt. Dieser kam zum erwarteten Ergebnis, dass die Algorithmus mit $O(N^2)$ skaliert. Dabei ist $N$ die Anzahl an Elementen. Somit ist die jetzige Implementierung bei größeren Inputs zu langsam für real-time und man sollte über bessere Algorithmen als den naiven nachdenken. Dies sollte mit WebGPU auch gut möglich sein, hingegen bei WebGL müsste man sich überlegen, wie man es schafft eine Octree parallelisiert mit einer Renderpipeline zu bauen.

### 4.2 Fazit

In dieser Arbeit zeigten Scientific Computing and Visualization im Browser diverse Nachteil gegenüber traditionellen Ansätzen, wie zum Beispiel CUDA, aber auch ein paar Vorteile.
So ermöglichen diese Browser Technologien, dass die Simulation und Visualisierung sehr portable sind. Denn man muss, wenn man einen modernen Browser verwendet, nichts installieren und meist auch keine spezielle Hardware haben. Es reicht aus einfach auf die entsprechende Website zu gehen. Zusätzlich lassen sich im Browser auch Scientific Computing und Visualization gut verbinden, was interaktive Simulationen ermöglicht.
Dafür gibt es aus Gründen der Kompatibilität nur eine eingeschränktere Menge an unterstützten Funktionen. So unterstützt WebGPU und WebGL, wie oben erwähnt, nicht direkt das Rechnen mit Double Precision Floats, was für Scientific Computing

meist aber notwendig ist.
Somit komme ich zum Schluss, dass diese Technologien sich primär für die Berechnung kleinerer Simulation eignet, aber für die meiste Forschung im Bereich Scientific Computing ungeeignet sind. Aber die Visualisierung auch größere Projekte ist gut möglich, somit kann die Technologie genutzt werden um die Visualisierungen von Forschung im Bereich Scientific Computing besser zu publizieren.

# References


AMHH19. Akenine-Moller, Tomas ; Haines, Eric ; Hoffman, Naty: *Real-time rendering*. AK Peters/crc Press, 2019

BJ. Brandon Jones, François B.: *Your first WebGPU app | Google Codelabs — codelabs.developers.google.com*. `https://codelabs.developers.google.com/your-first-webgpu-app`, . – [Accessed 21-07-2025]

gpu. *F64Value | @webgpu/cts — gpuweb.github.io*. `https://gpuweb.github.io/cts/docs/tsdoc/classes/webgpu_util_conversion.F64Value.html`, . – [Accessed 21-07-2025]

web. *WebGL Fundamentals — webglfundamentals.org*. `https://webglfundamentals.org/`, . – [Accessed 21-07-2025]