Optimierung von Voxel Engines mittels Culling und Greedy Meshing

MINT Hausarbeit an der Martin Luther Schule

Arne Daude

4. Januar 2025

Inhaltsverzeichnis

1 Einleitung

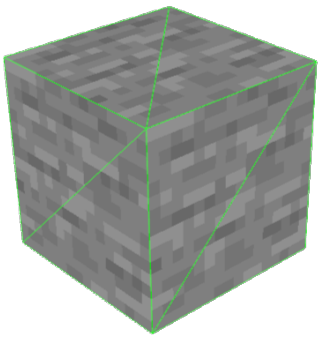
Eine Voxel Engine ist zuständig Voxels auf den Bildschirm zu projizieren. Voxels sind dabei wie Pixel, nur 3D (der Name „Voxel“ kommt von „Volumen“ kombiniert mit „Pixel“). Also werden viele kleine Würfel zusammengesetzt, um ein drei dimensionale Objekt darzustellen.

Dies wird in manchen Videospielen benutzt, wie Minecraft. Als Spieleentwickler erstelle ich auch ein Voxel Spiel und bin auf das Problem getreten, wie man diese implementiert. Es gibt hauptsächlich zwei Varianten, wie Voxel Engines implementiert werden:

- Volumengrafik: Es wird ein komplett neues Verfahren entwickelt, um die Voxels zu rendern, welches direkt mit den Voxels arbeitet. Dies hat zwar gute Performance, hat aber auch die Limitation, dass alle Voxel nur einfache Würfel seien können.
- Polygonnetz: Die Voxels werden zuerst in viele Dreiecke (oder allgemein Polygone) umgewandelt und dann von einem typischen 3D Renderer angezeigt. Somit muss man keinen neuen Renderer erstellen und kann auch andere Modelle als nur Würfel anzeigen, aber dieses Verfahren hat dafür schlechtere Performance.

Wegen der einfacheren Implementation, und dass man komplexere Modelle als nur Würfel erstellen kann, werden in den meisten Spielen die Polygonnetz Variante bevorzugt.

Mit der Polygonnetz Implementation sieht dann ein Voxel wie rechts gezeigt aus. Die grünen Linien zeigen dabei wie es in Dreiecke eingeteilt ist.



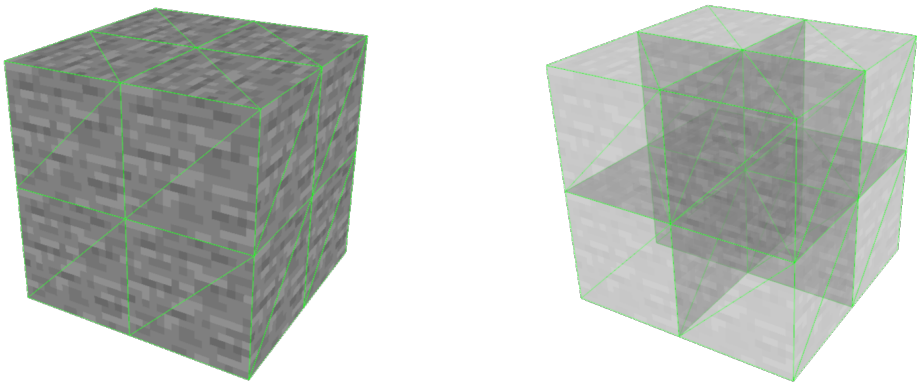
Ein Quadrat wird aus 2 Dreiecken zusammengesetzt und ein Würfel hat 6 Seiten. Somit besteht ein Würfel aus 12 Dreiecken. Wenn der Spieler nur 100 Voxels weit sehen könnte, wäre der Durchmesser 200 Voxels und somit das gesamte Volumen das angezeigt werden muss $200^3 = 8.000.000$ Voxels groß, also $12 \cdot 200^3 = 96.000.000$ Dreiecke.

Wir sehen also, dass schon bei einer sehr kleinen Sichtweite sehr viele Dreiecke erstellt werden. Diese Hausarbeit beschäftigt sich deswegen mit der Optimierung, die Anzahl der Dreiecke so weit wie möglich zu reduzieren.

Die meisten Optimierungen in dieser Hausarbeit kommen von https://youtu.be/qnGoGq7DWMc?si=Ke8vgcHWcgCdMGka und https://github.com/TanTanDev/binary_greedy_mesher_demo, und wurden dann noch ausgeweitet, damit sie auch für Voxels mit Texturen funktionieren.

2 Culling

Der erste Weg eine Voxel Engine zu optimieren wird offensichtlich, wenn man sich einen Würfel aus 8 Voxels betrachtet. Dabei beobachten wir, dass die Hälfte der Seiten der Voxels nach innen schauen und somit nicht sichtbar sind. Wenn man die Seitenlänge dieses Würfels verdoppelt, multipliziert man die Oberfläche mit 4 und das Volumen mit 8. Somit entstehen immer mehr Seiten, die nach innen schauen, umso größer das Objekt ist. Also wird dies bei großen Objekten dazu führen, dass die meisten Seiten nicht sichtbar sind.



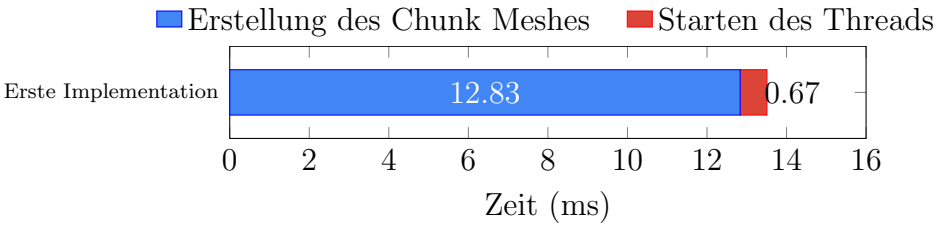
Mit „Culling“ beschreibt man die Optimierung, diese Seiten zu entfernen.

2.1 Erste Implementation

Die jetzige Methode ein Polygonnetz der Voxels zu erstellen besteht darin, für jeden Voxel ein Würfelnetz zu erstellen. Um dies also zu cullen, überprüft man noch die Nachbarvoxels, um zu entscheiden, welche Seiten sichtbar sind, und erstellt nur die sichtbaren Seiten.

Da die Spielwelt unendlich groß ist, muss sie in viele einzelne Polygonnetze aufgeteilt werden. Somit ist die Welt in sogenannte „Chunks“ eingeteilt, die $32 \times 32 \times 32$ Voxels beinhalten. Das Polygonnetz, das die Voxels in einem Chunk anzeigt, werde ich als „Chunknetz“ bezeichnen.

Da das Erstellen eines Chunknetzes lang dauern kann, soll das Spiel nicht darauf warten, da dies sonst sichtbar beim Spielen wäre. Deswegen werden die Chunknetze in separaten Threads erstellt. Zudem können somit mehrere Chunknetze gleichzeitig erstellt werden.



Dadurch entsteht aber ein neues Problem:
Wenn mehrere Threads zugriff auf die gleichen Daten haben, muss dieser Zugriff synchronisiert werden, da sonst eine Wettlaufsituation (genauer gesagt ein Data Race) entstehen kann. [1] Durch diese Synchronisierung müsste aber jeder Zugriff zu Chunks auf andere Threads warten, was das gesamte Spiel langsamer machen würde. Deswegen werden die Daten der nötigen Chunks zu dem Thread rüberkopiert.

Culling braucht Information aus den benachbarten Chunks, um zu entscheiden, ob die Ränder des Chunks sichtbar sind. Somit müssen die 6 Nachbarchunks auch zu dem Thread rüberkopiert werden, was sehr viele Daten sind. Man könnte zwar nur die Voxels rüberkopieren, die am Rand des Chunks sind, aber wir werden gleich eine Methode sehen, die dieses Problem und andere auf einmal löst.

[1] https://doc.rust-lang.org/nomicon/races.html

Ein weiteres Problem besteht darin, dass dieser Algorithmus für jeden Voxel noch die 6 Nachbarn betrachten muss. Somit wird jeder Voxel 7-mal betrachtet.

2.2 Binäres Culling

Wir können beide Probleme der ersten Implementation mit einer neuen Methode lösen. Wir betrachten dabei eine Reihe von Voxels als einen 64-Bit Integer, wobei die einzelnen Bits darstellen, ob sich dort ein Voxel befindet. Es werden dabei 64 Bits gebraucht und nicht nur 32, da man für das Culling auch wissen muss, welche Voxels sich am Rand eines Chunks befinden, und ein Chunk 32 Voxels lang ist. Somit könnten die Voxels in einem Chunk (und dem Rand) als ein 2 dimensionales Array von 64-Bit Integers dargestellt werden, wobei die 2 Dimensionen des Arrays die y- und z-Achse darstellen und der Integer eine Reihe von Voxels in der x-Achse darstellt. Der Vorteil davon ist, dass wir somit in der x-Achse binäre Arithmetik anwenden können für das Culling, was sehr schnell ist.

Beim Culling brauchen müssen wir die Seiten von Voxels finden, die nicht von einem anderen Voxel verdeckt werden. Mit binärer Arithmetik geht dies jetzt sehr einfach:

```
fn find_faces(bits: u32) -> u32 {
    bits & !(bits >> 1)
}
```

Diese Funktion gibt eine Bitmaske für alle Voxels, die von links sichtbar sind. Definiert man also eine Funktion mit `bits << 1`, dann bekommt man alle, die von rechts sichtbar sind.

Um dies für jede Achse zu wiederholen, erstellt man so ein Array für jede Achse. Diese kann man mit nur einem Zugriff pro Voxel wie folgt erstellen:

3 Greedy Meshing

3.1 Erste Implementation

3.2 Binäres Greedy Meshing

3.3 Korrektur der Texturen

4 Fazit

5 Quellen / Literaturverzeichnis

- https://youtu.be/qnGoGq7DWMc?si=Ke8vgcHWcgCdMGka
- https://github.com/TanTanDev/binary_greedy_mesher_demo
- https://de.wikipedia.org/wiki/Volumengrafik
- https://de.wikipedia.org/wiki/Polygonnetz
- https://de.wikipedia.org/wiki/Thread_(Informatik)
- https://doc.rust-lang.org/nomicon/races.html

Der Quellcode für dieses LaTeX Dokument:
https://github.com/BlueSheep3/MINT-Hausarbeit.

Meine Implementation dieser Optimierungen:
https://github.com/BlueSheep3/voxel_game.