Optimierung von Voxel Engines mittels Culling und Greedy Meshing

MINT Hausarbeit an der Martin Luther Schule

Arne Daude

2. April 2025

Inhaltsverzeichnis

1 Einleitung

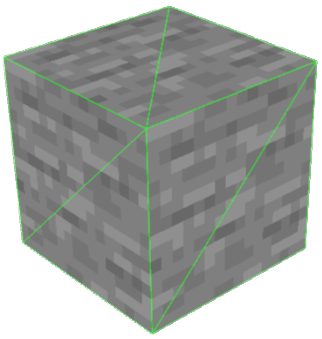
Eine Voxel Engine ist zuständig Voxels auf den Bildschirm zu projizieren. Voxels sind dabei wie Pixel, nur 3D (der Name „Voxel“ kommt von „Volumen“ kombiniert mit „Pixel“). Also werden viele kleine Würfel zusammengesetzt, um ein drei dimensionale Objekt darzustellen.

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 [3]: 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 [4]: 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.



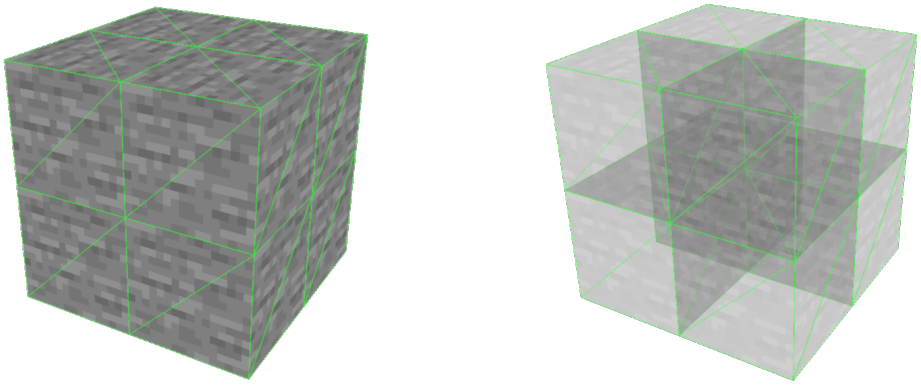
Ein Quadrat wird aus 2 Dreiecken zusammengesetzt und ein Würfel hat 6 Seiten. Somit besteht ein Würfel aus 12 Dreiecken. Wenn der Spieler nur 100 Voxels weit sehen könnte, wäre der Durchmesser 200 Voxels und somit das gesamte Volumen das angezeigt werden muss $200^3 = 8.000.000$ Voxels groß, also $12 \cdot 200^3 = 96.000.000$ 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 den Quellen [1] und [2], und wurden dann noch ausgeweitet, damit sie auch für Voxels mit Texturen funktionieren.

2 Culling

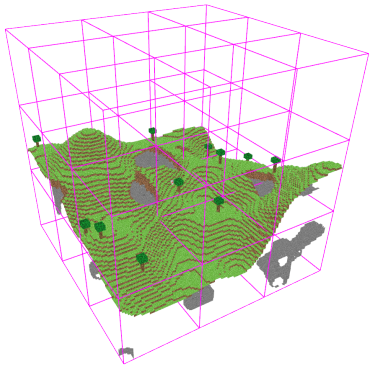
Der erste Weg eine Voxel Engine zu optimieren wird offensichtlich, wenn man sich einen Würfel aus 8 Voxels betrachtet. Dabei beobachten wir, dass die Hälfte der Seiten der Voxels nach innen schauen und somit nicht sichtbar sind. Wenn man die Seitenlänge dieses Würfels verdoppelt, multipliziert man die 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.

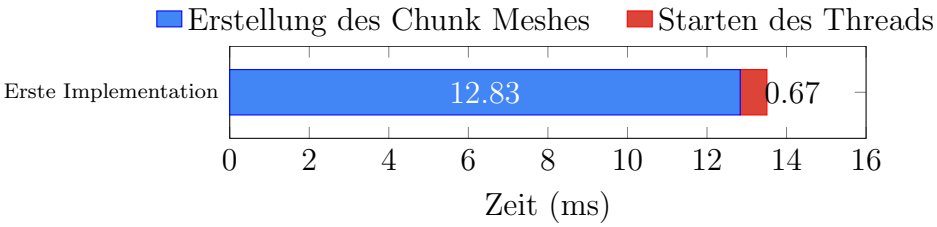
2.1 Erste Implementation

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 [5] 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 [6] (genauer gesagt ein Data Race) entstehen kann [7]. 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, was langsam ist, da es sich hier über sehr große Daten handelt.

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.

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

```
1 fn find_faces(mask: u64) -> u32 {
2   ((mask & !(mask >> 1)) >> 1) as u32
3 }
```

Diese Funktion gibt eine Bitmaske für alle Voxels, die von links sichtbar sind. Dabei ist das `mask & !(mask >> 1)` zuständig die sichtbaren Seiten zu erkennen. Definiert man also eine Funktion mit `mask & !(mask << 1)`, dann bekommt man alle, die von rechts sichtbar sind. Das `((..) >> 1) as u32` ist dann zuständig die einzelnen Bits, die den Rand des Chunks beschreiben, zu entfernen, da diese nicht mehr gebraucht sind. Diese neue Bitmaske kann dann wie folgt verwendet werden, um die Seiten der Voxels zu kreieren:

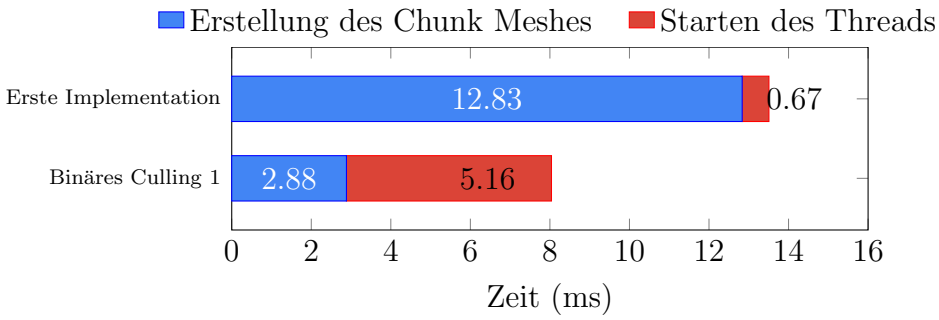
```rust
let mut mask = /* Bitmaske von dieser Reihe von Voxels */;
let mut k = 0;
while mask != 0 {
  let zeros = mask.trailing_zeros();
  mask = (mask >> zeros) & !1;
  k += zeros;
  // berechne die position dieser Seite basierend auf k,
  // und erstelle dann ein Mesh dafuer...
}
```

Um diese Funktion anwenden zu können, müssen wir die Bitmaske aber erst konstruieren. Dies werden wir für jede Achse wiederholen, da jede Bitmaske nur für eine Achse anwendbar ist:

```rust
for (pos, block) in chunk.blocks.iter_xyz() {
  let BlockInChunkPos { x, y, z } = pos;
  let [x, y, z] = [x as usize, y as usize, z as usize];
  if block_models[&block.id].should_cull {
    blocks_mask[Axis::X][y][z] |= 1 << (x + 1);
    blocks_mask[Axis::Y][x][z] |= 1 << (y + 1);
    blocks_mask[Axis::Z][x][y] |= 1 << (z + 1);
  }
}
```

Dabei brauchen wir auch nur einen Zugriff pro Voxel, während wir in der ersten Implementation 7 Zugriffe pro Voxel brauchten. Diese Bitmasken können wir erst erstellen und dann dem neuen Thread rübersenden.

Insgesamt erhalten wir folgende Performance:



Von 13,51 ms auf 8,04 ms ist eine Verbesserung von 40,5 %. Das sieht erstmal gut aus, jedoch hat sich die Zeit einen neuen Thread zu erstellen stark

erhöht. Wir haben nämlich die Bitmaske erst auf dem Main Thread [1] erstellt, damit nicht die gesamten Daten des Chunks rübergesendet werden müssen. Die Bitmaske zu erstellen braucht aber länger, als den Chunk rüberzusenden.

Wir wollen, dass den Thread zu erstellen möglichst schnell geht, da der Rest des Spiels warten muss, bis dies fertig ist. Somit könnte die Performance vom Spiel beeinträchtigt werden. Dieses Problem gibt es nicht beim Erstellen des Chunk Meshes, da dies auf einem separaten Thread passiert, was heißt, dass der Chunk Mesh nicht in einem Frame generiert werden muss, und somit die Performance nicht beeinflusst.

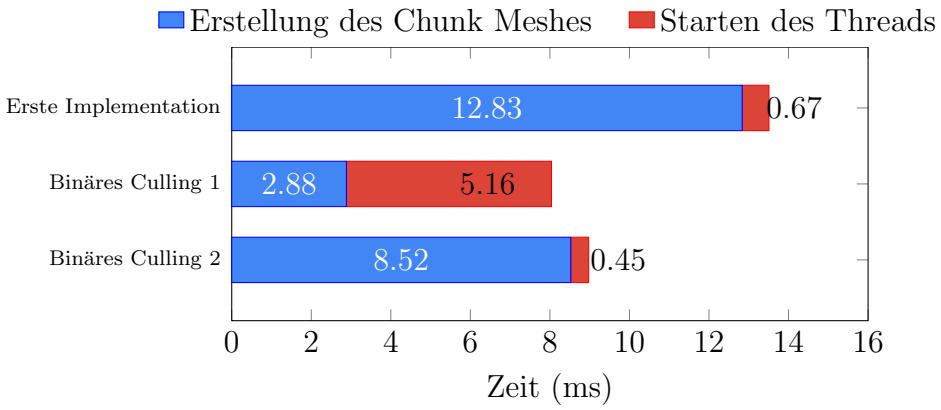
Obwohl es die gesamte Zeit den Chunk zu generieren langsamer machen wird, werden wir deswegen die gesamten Daten des Chunks und dessen Nachbarn an den neuen Thread senden, der dann selbst eine Bitmaske erstellt. Diesmal habe ich nicht die einfachere Implementation benutzt, und habe wirklich nur den Rand rüberkopiert:

```
pub fn chunk_padding_from_neighbour_chunks(
  neighbours: FaceMap<&Chunk>
) -> ChunkPadding {
  let mut chunk_padding = FaceMap::from_map(|_|
    [[Air::BLOCK; CHUNK_LENGTH]; CHUNK_LENGTH]
  );

  macro_rules! axes {
    ($(($a:ident, $b:ident) in ($axis:expr, $axis_name:ident)
    => [$x:expr, $y:expr, $z:expr]);* $(;)?) => {
      $(
      for $a in 0..CHUNK_LENGTH {
        for $b in 0..CHUNK_LENGTH {
          let mut pos = BlockInChunkPos::new($x, $y, $z);

          pos.$axis_name = CHUNK_LENGTH as u8 - 1;
          let block = neighbours[$axis.face_neg()].blocks[pos];
          chunk_padding[$axis.face_neg()][$a][$b] = block;

          pos.$axis_name = 0;
```

[1] Um genau zu sein, gibt es in Bevy nicht einen „Main“ Thread, wo die meisten Sachen gemacht werden, sondern alles ist auf mehrere Threads verteilt. Jedoch müssen alle Systeme [8] fertig sein, damit Bevy den nächsten Frame anzeigt, und das wird hier blockiert.

```
21          let block = neighbours[$axis.face_pos()].blocks[pos];
22          chunk_padding[$axis.face_pos()][$a][$b] = block;
23        }
24      }
25      )*
26    };
27  }
28  axes! {
29    (y, z) in (Axis::X, x) => [0, y as u8, z as u8];
30    (x, z) in (Axis::Y, y) => [x as u8, 0, z as u8];
31    (x, y) in (Axis::Z, z) => [x as u8, y as u8, 0];
32  }
33  chunk_padding
34 }
```

Mit dieser Methode wird die Zeit viel mehr auf den separaten Thread verschoben, während die gesamte Zeit nur gering erhöht wurde:



3 Greedy Meshing

3.1 Erste Implementation

3.2 Binäres Greedy Meshing

3.3 Korrektur der Texturen

4 Fazit

5 Quellen / Literaturverzeichnis

[1] https://youtu.be/qnGoGq7DWMc?si=Ke8vgcHWcgCdMGka

[2] https://github.com/TanTanDev/binary_greedy_mesher_demo

[3] https://de.wikipedia.org/wiki/Volumengrafik

[4] https://de.wikipedia.org/wiki/Polygonnetz

[5] https://de.wikipedia.org/wiki/Thread_(Informatik)

[6] https://de.wikipedia.org/wiki/Wettlaufsituation

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

[8] https://bevy-cheatbook.github.io/programming/systems.html

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

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