Optimierung von Voxel Engines mittels Culling und Greedy Meshing

MINT Hausarbeit an der Martin Luther Schule

Arne Daude

13. Mai 2025

Inhaltsverzeichnis

1 Einleitung **2**

2 Culling **3**
2.1 Erste Implementation 4
2.2 Binäres Culling 5

3 Greedy Meshing **9**
3.1 Binäres Greedy Meshing 10
3.2 Korrektur der Texturen 14

4 Fazit **18**

5 Quellen / Literaturverzeichnis **18**

1 Einleitung

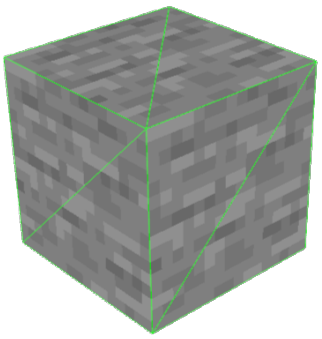
Eine Voxel Engine ist zuständig Voxels auf den Bildschirm zu projizieren. Voxels sind dabei wie Pixel, nur 3D (der Name „Voxel“ kommt von „Volumen“ kombiniert mit „Pixel“). Also werden viele kleine Würfel zusammengesetzt, um ein drei dimensionale Objekt darzustellen.

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.



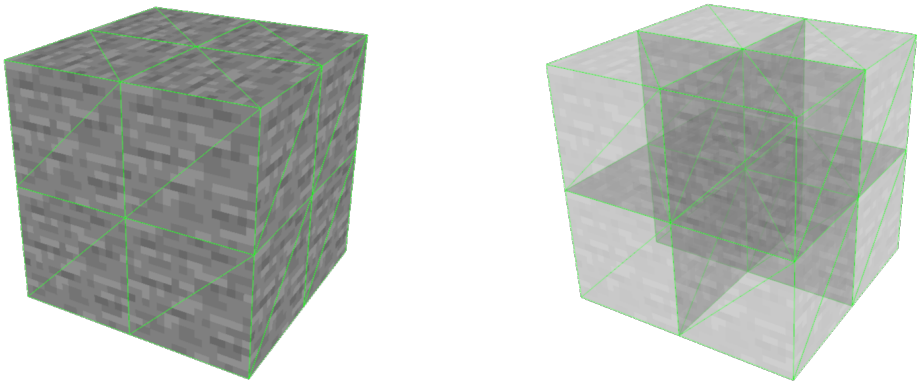
Ein Quadrat wird aus 2 Dreiecken zusammengesetzt und ein Würfel hat 6 Seiten. Somit besteht ein Würfel aus 12 Dreiecken. Wenn der Spieler nur 100 Voxels weit sehen könnte, wäre der Durchmesser 200 Voxels und somit das gesamte Volumen das angezeigt werden muss $200^3 = 8.000.000$ Voxels groß, also $12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

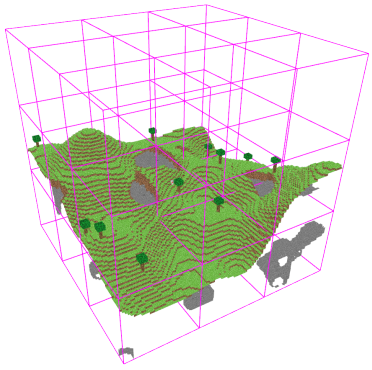
Der erste Weg eine Voxel Engine zu optimieren wird offensichtlich, wenn man sich einen Würfel aus 8 Voxels betrachtet. Dabei beobachten wir, dass die Hälfte der Seiten der Voxels nach innen schauen und somit nicht sichtbar sind. Wenn man die Seitenlänge dieses Würfels verdoppelt, multipliziert man die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

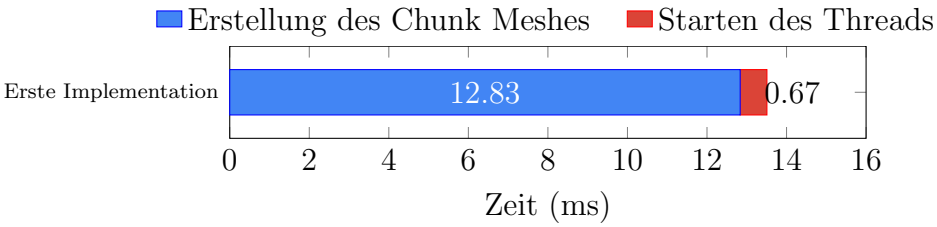
2.1 Erste Implementation

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:

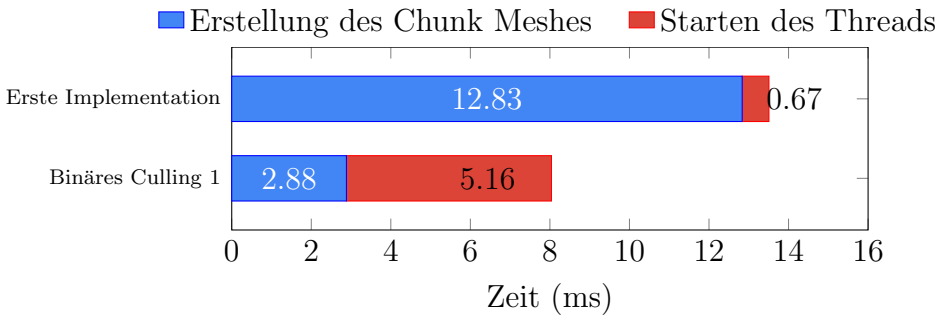
```rust
let mut mask = /* Bitmaske von dieser Reihe von Voxels */;
let mut k = 0;
while mask != 0 {
  let zeros = mask.trailing_zeros();
  mask = (mask >> zeros) & !1;
  k += zeros;
  // berechne die position dieser Seite basierend auf k,
  // und erstelle dann ein Mesh dafuer...
}
```

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.

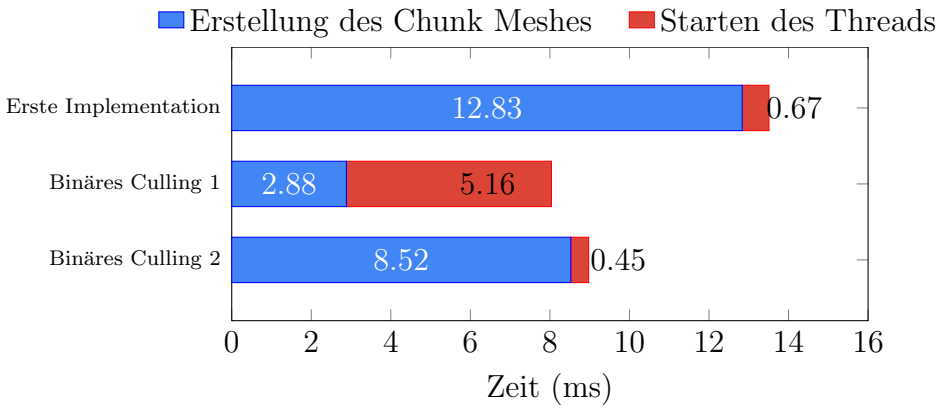
Obwohl es die gesamte Zeit den Chunk zu generieren langsamer machen wird, werden wir deswegen die gesamten Daten des Chunks und dessen Nachbarn an den neuen Thread senden, der dann selbst eine Bitmaske erstellt. Diesmal habe ich nicht die einfachere Implementation benutzt, und habe wirklich nur den Rand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

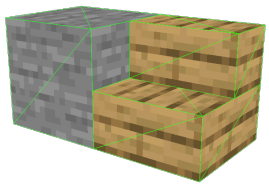
[1]Um genau zu sein, gibt es in Bevy nicht einen „Main“ Thread, wo die meisten Sachen gemacht werden, sondern alles ist auf mehrere Threads verteilt. Jedoch müssen alle Systeme [8] fertig sein, damit Bevy den nächsten Frame anzeigt, und das wird hier blockiert.

```
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:

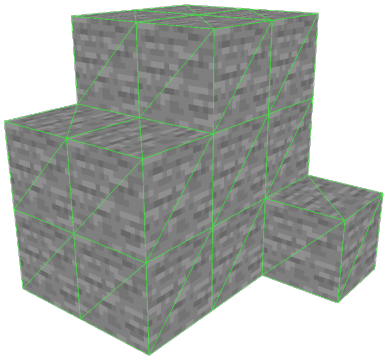


Anders als in [1] und [2] gibt es bei meinem Spiel auch Voxels, die nicht ein ganzer Würfel sind. Dafür funktioniert dieser Algorithmus nicht, da ein Voxel direkt neben diesem trotzdem sichtbar sein kann.
Da in den meisten Fällen aber nur wenige von diesen „nicht-vollen“ Voxels existieren, habe ich mich entschieden diese einfach in eine separate Bitmaske zu tun und auf diese dann kein Culling angewandt.

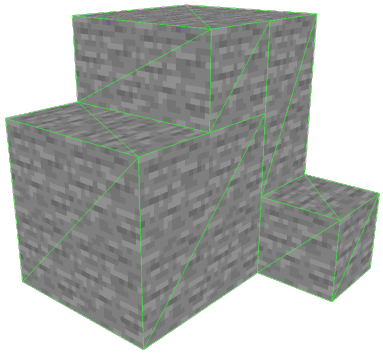


3 Greedy Meshing

Nun kommen wir zum zweiten Weg eine Voxel Engine zu optimieren:



In der Grafik oben gibt es viele flache Flächen, die aus unnötig vielen Dreiecken aufgebaut sind. Wir könnten es wie folgt in Dreiecke einteilen, um die Anzahl zu reduzieren:



Somit werden zum Beispiel nur 2 Dreiecke verwendet, um eine 2x2 Fläche von Voxels zu bilden, anstatt 8 Dreiecke.

Diese Optimierung wird „Greedy Meshing“ genannt.

3.1 Binäres Greedy Meshing

Ich habe erst geplant eine einfachere Implementation von Greedy Meshing zu machen, bevor ich es mit einem binären Algorithmus mache, aber dadurch, dass das Culling schon binär ist, ist die binäre Greedy Meshing Implementation sogar einfacher als eine andere.

Der Grundgedanke in dieser Implementation ist, dass jedes Mal, wenn wir eine Seite betrachten, wir versuchen diese erst in eine Richtung so weit zu erweitern, bis keine Seite mehr da ist und dann erweitern wir diesen ganzen Streifen in die andere Richtung. Während man das macht, entfernt man immer die Bits in der Bitmaske, die gerade für diese Dreiecke verwendet werden, damit sie nicht später für andere Dreiecke wieder verwendet werden.

Wenn wir eine Bitmaske haben, in der ein 32-Bit Integer eine Reihe von Seiten darstellt, dann geht es sehr schnell die Streifen zu erkennen, da man mit den x86-Befehlen BSF [9] und BSR [10] die Anzahl von Nullen oder Einsen am Anfang oder Ende eines 32-Bit Integers mit einem einzigen Befehl berechnen kann. Zudem kann man mit einem einfachen bitweisen ODER überprüfen, ob der Streifen in die andere Richtung erweitert werden kann, und man kann die Einträge in der Bitmaske mit einem bitweisen UND entfernen.

```
let mut mask_copy = array[j];
let mut k = 0;
while mask_copy != 0 {
```

```
  let zeros = mask_copy.trailing_zeros();
  mask_copy >>= zeros;
  k += zeros;

  let ones = mask_copy.trailing_ones();
  mask_copy = mask_copy.checked_shr(ones).unwrap_or(0);
  let from = k;
  k += ones;

  // this entire strip of blocks, as a bitmask.
  // `<<` doesnt overflow in the way you would expect,
  // so we use `checked_shl` here instead.
  // `from != 32`, because otherwise `mask_copy == 0`, so the
  // left shift there will never overflow (in any problematic way).
  let ones_exp2 = 1_u32.checked_shl(ones).unwrap_or(0);
  let strip_mask = (ones_exp2.wrapping_sub(1)) << from;

  // expand the strip into a rectangle,
  // and unset any bits along the way
  array[j] &= !strip_mask;
  let mut strip_expand = 0;
  for next_mask in &mut array[(j + 1)..] {
    if *next_mask & strip_mask == strip_mask {
      strip_expand += 1;
      *next_mask &= !strip_mask;
    } else {
      break;
    }
  }

  // ...create a face
}
```

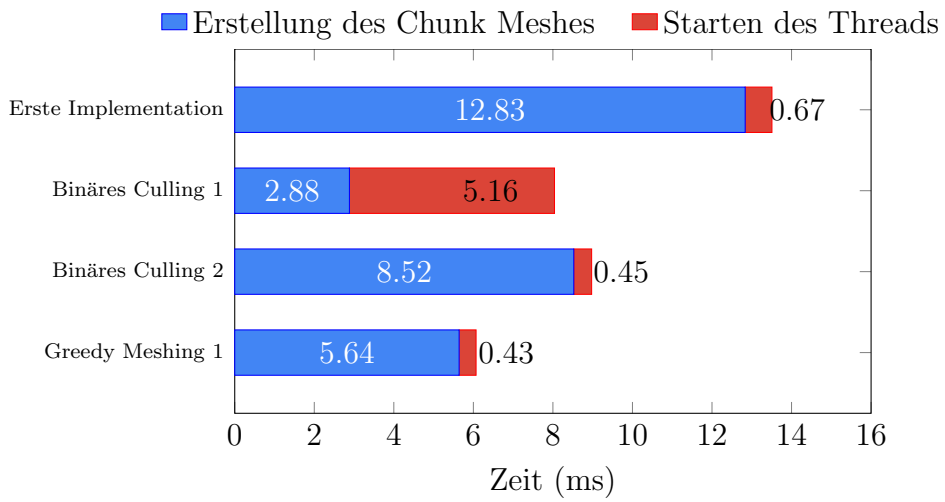
Vielleicht ist ihnen aber schon aufgefallen, dass die Bitmaske, die wir für das Binäre Culling verwendet haben nicht in die richtige Richtung ausgerichtet ist. Dabei ist nämlich ein Integer eine Reihe von Voxels, die sich gegenseitig verdecken können, während wir hier eine Reihe von Voxels (oder genauer: sichtbare Seiten) brauchen, die nebeneinander sind. Deswegen müssen wir aus der Culling Bitmaske eine Bitmaske konstruieren, die getauschte Koordinaten hat.

```rust
let culled_blocks_mask = // maske von Binaeres Culling
let mut greedy_mask = Box::<FaceMap<ChunkArray2D<u32>>>::default();
for (face, array2d) in culled_blocks_mask.iter_face() {
  for (i, array) in array2d.iter().enumerate() {
    for (j, mask) in array.iter().enumerate() {
      let mut mask = *mask;
      let mut k = 0;
      while mask != 0 {
        let zeros = mask.trailing_zeros();
        mask = (mask >> zeros) & !1;
        k += zeros;
        greedy_mask[face][k as usize][i] |= 1 << j;
      }
    }
  }
}
greedy_mask
```

2

Wenn wir nun diese Bitmaske haben, können wir mit dem oben genannten Algorithmus die Seiten zu größeren Seiten kombinieren. Wenn wir also diese Information benutzen, um die Dreiecke größer zu machen sind wir schon mit Greedy Meshing fertig!

[2]Da horizontale Streifen von Voxels häufiger vorkommen als vertikale Streifen, mache ich, dass die Bitmasken in den x- und z-Achsen beide horizontal ausgerichtet sind. Deswegen wird in der wirklichen Implementation für die z-Achse `i` und `j` getauscht.

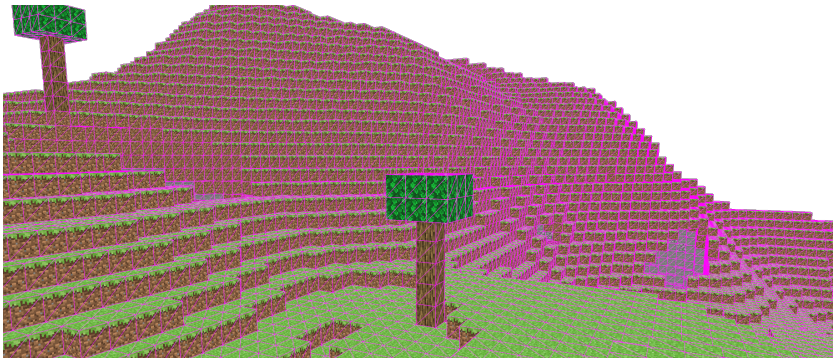


Mit diesem Algorithmus erhalten wir sogar bessere Performance, da wir weniger Dreiecke konstruieren müssen. Zudem ist die Anzahl von Dreiecken in einer typischen Spielwelt jetzt von 1.857.984 Ecken und 928.992 Dreiecken runter auf 809.484 Ecken und 404.742 Dreiecken gesunken. Somit haben wir etwa 2,3-mal weniger Dreiecke.

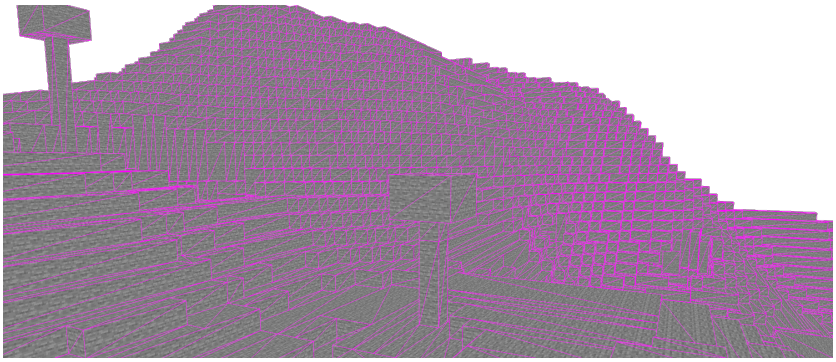
... eigentlich sind wir aber noch nicht ganz fertig!
Wir haben noch ein Problem übersehen.

3.2 Korrektur der Texturen

Vor der Greedy Meshing Implementation sah eine typische Spielwelt so aus:



Aber jetzt mit der neuen Implementation sieht die gleiche Spielwelt wie folgt aus:



Das Problem ist, dass wir noch nicht überlegt haben, wie die Texturen der Voxels in das Polygonnetz des Chunks eingebaut werden. Bevor wir Greedy Meshing benutzt haben, konnte man einfach jedem Dreieck eine Textur geben, basierend darauf zu welchem Voxel es gehört. Jedoch kann jetzt ein Dreieck für mehrere Voxels zuständig sein.

In [1] und [2] wurden hierfür separate Polygonnetze und Bitmasken für jede Art von Voxel verwendet (und es gab auch keine Texturen, sondern nur Farben). In meinem Spiel will ich jedoch viele unterschiedliche Arten von Voxels haben. Deswegen wäre es unrealistisch ein separates Polygonnetz für jede Art zu haben. Meine Strategie um dieses Problem zu lösen ist es, mehrere Texturen auf einem Dreieck anzuzeigen.

Um dies zu verstehen, müssen wir erst einmal angucken, wie die Texturen in einem Polygonnetz ohne Greedy Meshing angewandt werden. Es wird erst ein Array von Texturen erstellt, das alle möglichen Texturen von Voxels beinhaltet. Beim Erstellen des Polygonnetzes bekommt dann jedes Dreieck[3] in dem Polygonnetz einen Index, den es benutzen kann, um die richtige Textur für dieses Voxel in dem Array zu finden.

Die einfachste Lösung wäre also, anstatt nur einen Index zu speichern, ein 2 dimensionales Array von Indexen zu speichern für jeden Voxel, der von diesem Dreieck dargestellt wird. Leider geht dies nicht, da alle Informationen, die man für einzelne Dreiecke (also nicht das gesamte Polygonnetz) speichern will, müssen in dem `GPUVertexFormat` [11] sein. Dabei gibt es nur Datentypen mit konstanter Größe, wir brauchen aber ein Array mit dynamischer Länge. Zudem wäre es unrealistisch immer das größtmögliche Array zu speichern und unbenutzte Werte zu haben, wenn wir ein kleineres Dreieck haben, da das größte Dreieck $32 \cdot 32 = 1024$ Voxels groß sein kann, aber die meisten Dreiecke weniger als 4 Voxels groß sind. Somit würden wir sehr viele Daten verschwenden.

Wir müssen also ein Array für das gesamte Polygonnetz verwenden. Mein erster Gedanke dabei war es ein `Vec` rüber zu senden, um so ein Array mit dynamischer Länge zu erhalten, jedoch meint Bevy:
`"runtime-sized array can't be used in uniform buffers"` [12]
... was komisch ist, da WebGPU eigentlich dynamische Arrays unterstützen sollte.

Zum Glück gibt es aber einen anderen Weg Daten mit einer dynamischen Länge zu haben: Wir können eine Textur verwenden. Um damit ein Array von Indexen (also `u32`) darzustellen, verwenden wir das Format `TextureFormat::R32Uint` [13]. Dieses Format hat nur einen roten Kanal, der als `u32` gespeichert wird.

Wenn die Textur 3 dimensional wäre könnte somit jedes Dreieck einen z-Index speichern und die x-y-Ebene von der Textur wäre die Textur des Voxels. Jedoch muss bei einer 3 dimensionalen Textur jede solche Ebene die gleiche Größe haben. Deswegen müssen wir eine Dimension tiefer gehen. Jedoch müssten wir bei einer 2 dimensionalen Textur die genaue Anordnung

[3] In Wirklichkeit kann man mit WebGPU nicht Daten mit jedem Dreieck assoziieren, sondern nur mit jedem Vertex. Somit speichern wir diesen Index doppelt so oft, wie wir es brauchen. Aber es geht hier um nur sehr wenig Daten, also ist das nicht relevant.

der einzelnen Voxel Texturen überlegen. Also habe ich mich entschieden eine 1 dimensionale Textur zu verwenden. Dabei merkt sich jedes Dreieck bei welchem Index in diesem Array es startet, und die Breite dieser Seite, damit es weiß, wie weit es in dem Array nach vorne springen muss, wenn es eine Textur, die weiter oben ist, benutzen will.

Es gibt also erstens die 1 dimensionale Textur, für diesen gesamten Chunk:

```
@group(2) @binding(102) var face_texture_indices: texture_1d<u32>;
```

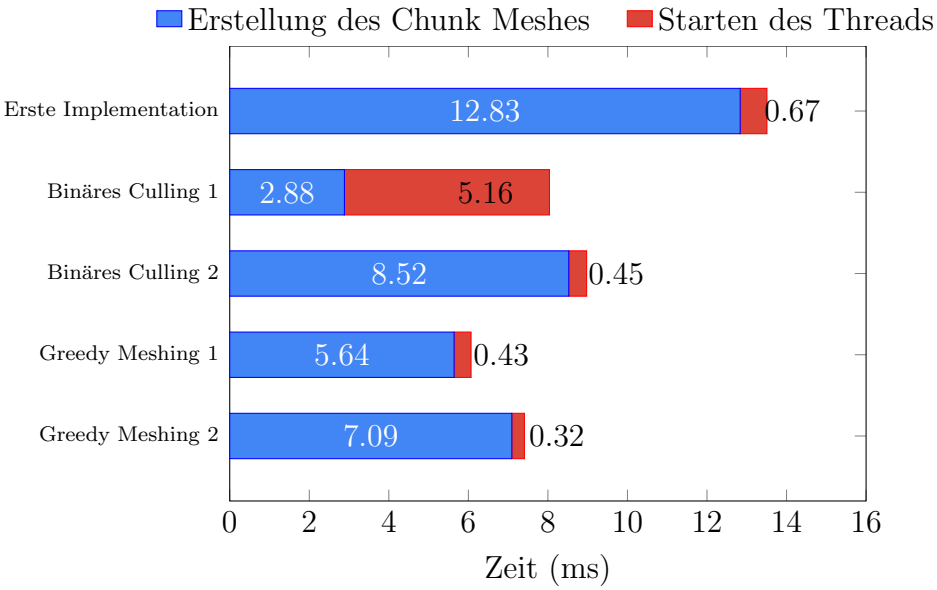
Und dann noch die zwei Werte, die jedes einzelne Dreieck wissen muss:

```
@location(7) start_texture_index: u32,
@location(8) rect_width: u32,
```

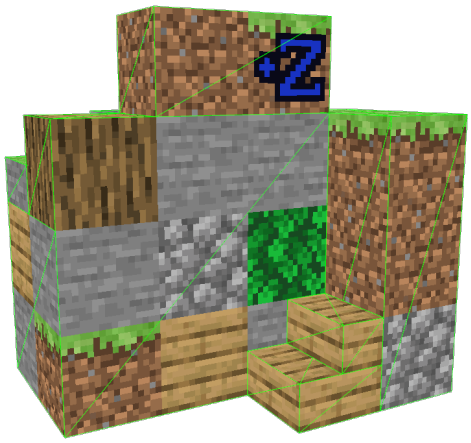
Diese werden dann im `fragment` Shader wie folgt verwendet, um die wirkliche Textur an dieser Position zu bekommen:

```
// Berechne den Index, an der die wirkliche Textur
// dieses Voxels ist, basiert auf der Position (uv).
let uv_offset = u32(in.uv.x) + u32(in.uv.y) * rect_width;
let face_index = start_texture_index + uv_offset;
let block_index = textureLoad(face_texture_indices, face_index, 0);

// Bekomme die Farbe an dieser Position in
// der wirklichen Textur dieses Voxels.
pbr_input.material.base_color = textureSample(
  block_textures, block_texture_sampler, in.uv, u32(block_index.r)
);
```

Um `face_texture_indices` zu berechnen, müssen wir beim Erstellen eines Polygonnetzes alle Voxels in einer kombinierten Seite zu einem Array hinzufügen. Das verbraucht etwa 1,45 ms, also erhalten wir folgende Performance für diese Implementation:



Mit dieser Implementation können jetzt viele unterschiedliche Texturen auf nur einem Dreieck angezeigt werden:



4 Fazit

Die Optimierungen von Culling und Greedy Meshing kombiniert ergeben etwa eine Verdopplung der Performance und einer Halbierung der Anzahl von Dreiecke in jedem Polygonnetz. Wir können mit diesen Optimierungen immer noch nicht-volle Voxels anzeigen, und haben wir gesehen, wie mehrere Texturen auf einem einzigen Dreieck angezeigt werden können.

Mögliche Optimierungen, die man noch implementieren könnte, wären:

- Level Of Detail [14]: Chunks, die weiter weg sind, könnten mit geringerer Genauigkeit erstellt werden, da man die Ungenauigkeiten von der Distanz nicht bemerken würde.
- Occlusion Culling [15]: Gesamte Chunks sind oft nicht sichtbar. Zum Beispiel Höhlen sind oft nicht sichtbar, da sie unter der Erde sind. Diese Chunks könnten übersprungen werden.
- Culling und Greedy Meshing bei nicht-vollen Voxels: Meine Implementation wendet Culling und Greedy Meshing bei nicht-vollen Voxels nicht an, da diese komplizierter zu implementieren sind, weil sie zum Beispiel nur an bestimmten Seiten geculled werden können. In Situationen, wo es sehr viele nicht-volle Voxels gibt, könnte das aber ein großes Problem für Performance werden.

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

[9] https://www.felixcloutier.com/x86/bsf

[10] https://www.felixcloutier.com/x86/bsr

[11] https://gpuweb.github.io/gpuweb/#enumdef-gpuvertexformat

[12] https://github.com/teoxoy/encase/blob/v0.10.0/src/types/runtime_sized_array.rs#L146

[13] https://docs.rs/bevy/0.15.0/bevy/render/render_resource/enum.TextureFormat.html#variant.R32Uint

[14] https://de.wikipedia.org/wiki/Level_of_Detail

[15] https://en.wikipedia.org/wiki/Hidden-surface_determination#Occlusion_culling

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

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