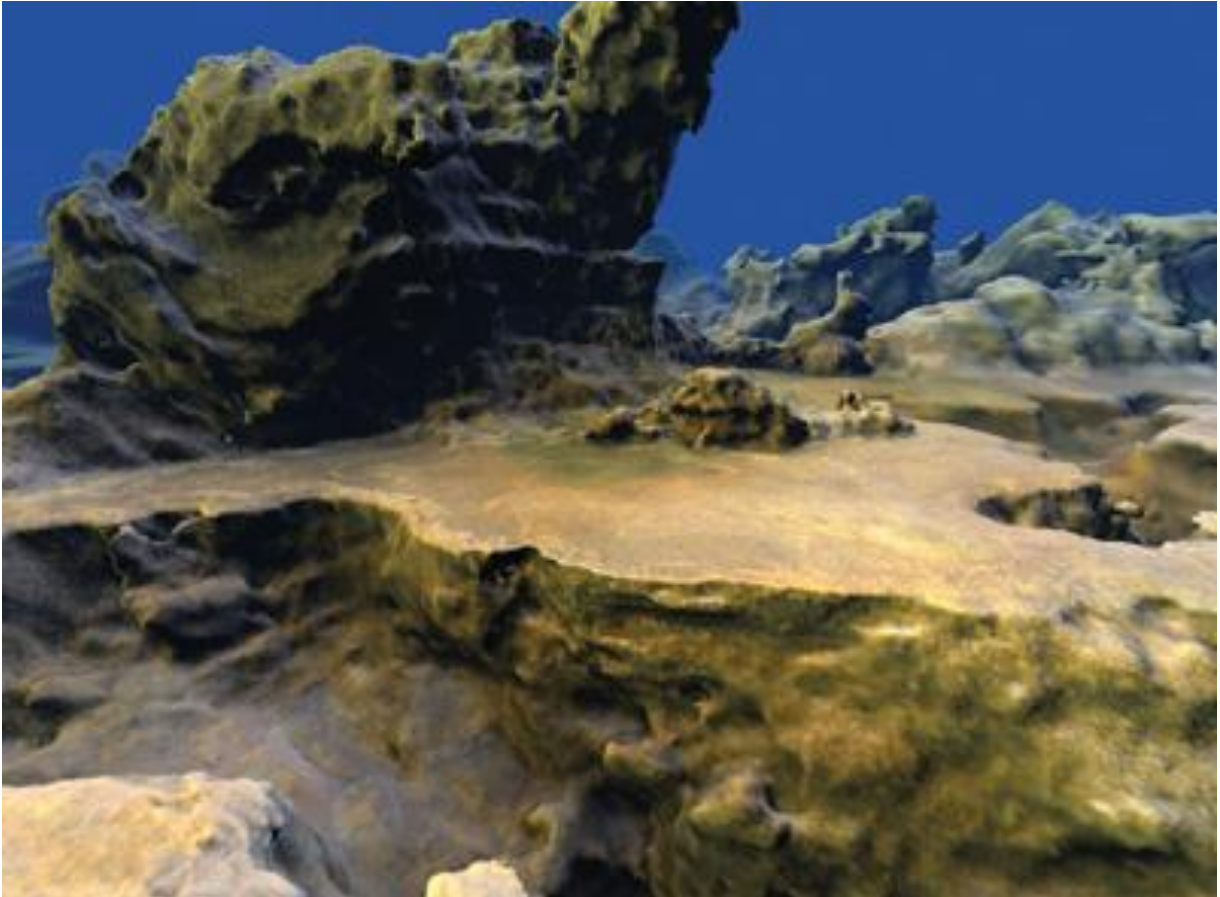


# Marching Cubes



Quelle: <https://developer.nvidia.com/gpugems/gpugems3/>

Teilnehmer:

Fabian Berkowitz und Christoph Kaiser

# 1. Einleitung

Es gibt in der Computergrafik einige Methoden, um dreidimensionale Objekte zu modellieren. Die Ansätze variieren hierbei vom Drahtgittermodell (Eckige/polygonale Flächen ergeben Objektoberflächen) bis hin zur Voxel (Welt wird in ein würfelförmiges Gitter aus Cubes/Voxeln unterteilt) Darstellung. Als eine Art Hybrid ist hieraus der Marching Cubes Algorithmus entstanden.

Die Idee hinter Marching Cubes ist recht simpel. Die Umgebung wird in eine beliebige Anzahl Voxel unterteilt. Im Anschluss wird auf jeden Punkt eines Voxels eine Dichtefunktion angewendet (bspw.  $d = \sin(x) - \sin(y) + \sin(z)$ ) und der entsprechende Wert gespeichert. Dieser gibt nun an ob sich ein Punkt inner- oder außerhalb des zu erstellenden Objektes befindet. Die einfachste Unterscheidung wäre hierbei (wird in diesem Projekt nicht angewendet, sondern nur  $d \leq \text{Schwellwert}$  und  $d > \text{Schwellwert}$  also inner- bzw. außerhalb):

- $d < 0$ : Punkt befindet sich innerhalb des Objekts
- $d = 0$ : Punkt befindet sich genau auf der Oberfläche des Objekts
- $d > 0$ : Punkt befindet sich außerhalb des Objekts

Nun erfolgt für jeden Voxel dieselbe Auswertung zur Bestimmung von zu zeichnenden Polygonen (Dreiecke). Ist der Wert  $d$  bei allen Punkten eines Voxels positiv oder negativ, so werden keine Polygone gezeichnet. Für alle anderen Fälle wird mindestens ein Polygon gezeichnet. Um diese Auswertung durchzuführen gibt es allerdings eine einfache Art und Weise. Es existieren Lookup Tabellen, welche anhand des Cubeindex (auch Case genannt) angeben wo und wie viele Polygone gezeichnet werden müssen. Der Cubeindex wird hierbei wie folgt berechnet:

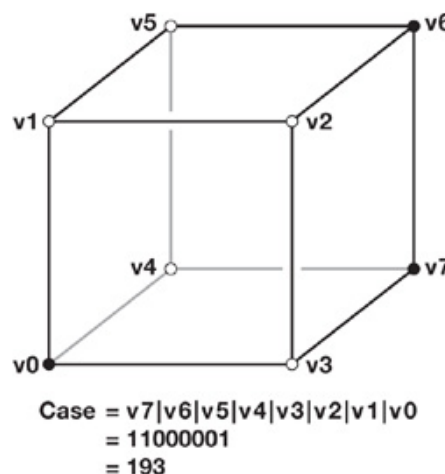


Abbildung 1: Voxel mit Indizierung

In Abbildung 1 ist ein Voxel mit seiner Vertexnummerierung zu sehen. Diese Nummerierung ist hierbei für jeden Voxel gleich und wichtig zur Berechnung des Cubeindex. Dafür wird der Wert  $d$  jedes Punktes genutzt um, wie oben zu sehen, eine Bitmaske zu erstellen. Ist der Wert  $d$  positiv so wird für ihn eine 1, andernfalls eine 0, in die Bitmaske geschrieben.

Diese Bitmaske kann nun genutzt werden, um die richtigen Einträge aus den Tabellen auszulesen. Eine Tabelle gibt an auf welchen Kanten (Abbildung 2) die Polygone einen Vertex haben und eine weitere gibt an wie viele Vertices genutzt werden ( $\#Vertices \% 3 == 0$ ).

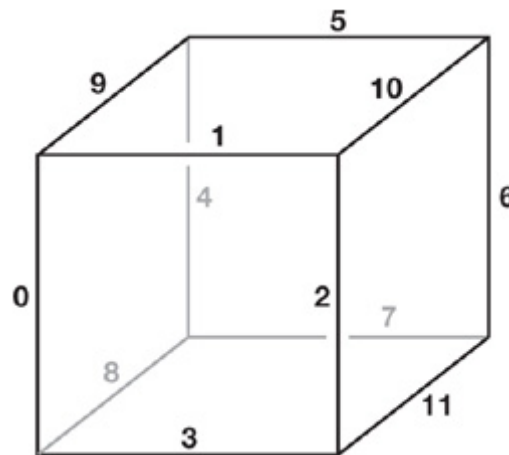


Abbildung 2: Kantenbeschriftung

Über eine Interpolation der beiden Punkte, welche an dieser Kante anliegen, können die Koordinaten eines Vertex berechnet werden. Wurden nun alle Vertices berechnet können die Polygone gezeichnet werden, um das Objekt bzw. die Landschaft darzustellen.

## 2. Projektdefinition

Das Ziel dieses Projekts ist die Implementierung des Marching Cubes Algorithmus. Hierbei soll die Implementierung auf der GPU stattfinden. D.h. die Dreiecke werden in der Grafikpipeline gerendert und geeignete Berechnungen auf der Grafikkarte ausgeführt. Die Dichtefunktion kann hierbei frei gewählt werden, wobei es gilt verschiedene Dichtefunktionen zu implementieren. Die Lookup-Tabelle muss nicht selbst erstellt werden, sondern kann aus dem Internet entnommen werden.

Als Technologien steht es frei CUDA oder OpenCL zu nehmen. Die Wahl fällt hierbei auf die Nutzung von CUDA zur Berechnung auf der GPU. Für das Rendering soll die Bibliothek OpenGL genutzt werden.

Fokus dieses Projekts liegt auf folgenden Schwerpunkten:

- Auswahl der Dichtefunktion
- Auswahl der Szeneneigenschaften
- Berechnung des Algorithmus unter Nutzung der GPU
- Nutzung der Grafikpipeline (inklusive der Färbung des Meshes)

Zudem gilt es folgende Aspekte zu evaluieren:

- Einfluss der Anzahl Cubes
- Einfluss der Dichtefunktion
- Granularitäten
- Renderformen verändern

### 3. Allgemeine Implementierung

Die prinzipielle Implementierung des Algorithmus wurde bereits in der Einleitung vorgestellt. Diese wird nun noch präzisiert und die Gedanken dahinter erläutert. Auf die Erstellung des Fensters und der damit verbundenen Kontexterzeugung (OpenGL kann über den Kontext auf das Fenster zugreifen) wird nicht weiter eingegangen.

Zu Beginn des Programms wird der OpenGL Part initialisiert. Hierfür ist es nötig ein sogenanntes Vertex Array Objekt (VAO) zu erstellen und zu binden. Dieses liefert eine Umgebung, in der Vertex Buffer Objekte (VBO) erstellt und zugewiesen werden können. Dadurch sind diese mit dem VAO, in dessen Kontext sie erzeugt wurden, verbunden. Die VBOs werden genutzt, um Daten (Eigenschaften) zu speichern und sie auf die Grafikkarte zu laden. Somit steht jedes VBO für eine Eigenschaft, welche gezeichnet werden soll (bspw. die Vertices, Farbe). Für unser Projekt werden drei Arrays angelegt:

- vao[2]
- vbo1[4]
- vbo2[4]

Somit haben wir die Möglichkeit zwei VAOs und je VAO vier VBOs zu erzeugen. Das erste VAO ist gedacht um die Polygone mit weißer Füllung zu zeichnen und das zweite soll eine schwarze Umrandung für die Polygone zeichnen (Abbildung 3).

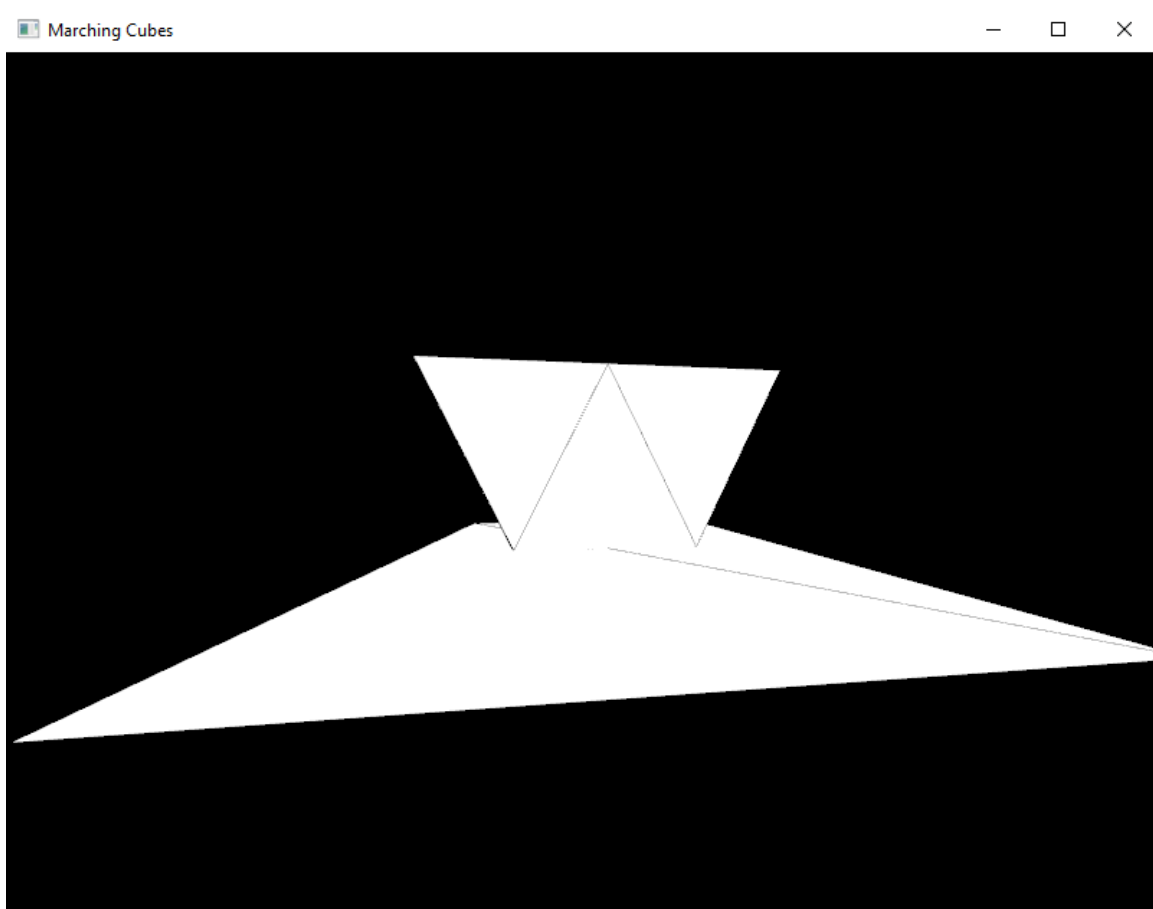


Abbildung 3: Polygone mit zwei VAOs (Füllung und Rand)

Die beiden VBOs (vbo1 und vbo2) sind fast identisch aufgebaut. Das erste Element enthält die Punkte, die zur Unterteilung des Raums in Cubes, benötigt werden (zeichnet ein Grid). Das zweite Element beinhaltet alle Punkte, die gezeichnet werden können. Im dritten Element sind die für uns wichtigsten Daten gespeichert – die Vertices, welche durch den Marching Cubes Algorithmus berechnet werden und im vierten Element ist die Farbe, in der gezeichnet werden soll abgespeichert. Das wäre für vbo1 die Farbe Weiß und für vbo2 Schwarz.

Wichtig für unsere Implementierung ist es, dass Pointer zu diesen drei Arrays an die *marching\_cubes\_kernel.cu* übergeben wird. In dieser findet die Generierung der VAOs und VBOs statt. Nachdem die Buffer generiert wurden, beginnt das Erzeugen der Daten. Hierfür werden für die jeweiligen Buffer Speicherbereiche alloziert und gefüllt.

```
// Initialize points data.
float delta = (xmax - xmin) / (numPoints - 1);
for (int i = 0; i < numPoints; i++) {
    for (int j = 0; j < numPoints; j++) {
        for (int k = 0; k < numPoints; k++) {

            int idx = i + j * numPoints + k * numPoints * numPoints;

            // Set initial position data
            points[idx].x = xmin + delta * i;
            points[idx].y = xmax - delta * j;
            points[idx].z = xmin + delta * k;
            points[idx].w = 1.0f;

            color_white[idx].x = 1.0f;
            color_white[idx].y = 1.0f;
            color_white[idx].z = 1.0f;
            color_white[idx].w = 1.0f;

            color_black[idx].x = 0.0f;
            color_black[idx].y = 0.0f;
            color_black[idx].z = 0.0f;
            color_black[idx].w = 1.0f;

        }
    }
}
```

Abbildung 4: Erstellung der Punkt und Farbdaten

In Abbildung 4 ist die Füllung des Punkt- und der beiden Farbbuffer zu sehen. Über die Variable *delta* wird die Skalierung vorgenommen und durch den Index *idx* werden die Punkte an die passende Stelle gesetzt, um nachher das Grid zu befüllen. Der *w*-Wert der Punkte wird mit 1.0f initialisiert. Diese Variable wird später für die Dichtefunktion relevant. Die Farbdaten werden entsprechend mit 1.0f für Weiß und 0.0f für Schwarz gefüllt.

Wie eben schon erwähnt werden die nun erstellten Punktdaten verwendet, um das Grid mit Daten zu versorgen. Hierbei ist es wichtig zu beachten das es mehr Griddaten geben wird als Punktdaten. Das liegt daran, dass sich die Cubes aus dem Grid an ihren Eckpunkten berühren. D.h. es werden pro Cube zwar nur 8 Punkte benötigt, jedoch muss der selbe Punkt auch in anderen Cubes abgespeichert werden, um später getrennt voneinander bewertet zu werden.

```
// Initialize grid data.
for (int i = 0; i < (numPoints - 1); i++) {
    for (int j = 0; j < (numPoints - 1); j++) {
        for (int k = 0; k < (numPoints - 1); k++) {

            int idx_pt = i + j * numPoints + k * numPoints * numPoints;
            int idx_sq = idx_pt - j + k - 2 * k * numPoints;

            // Set initial position data
            grid[16 * idx_sq + 0] = points[idx_pt];
            grid[16 * idx_sq + 1] = points[idx_pt + 1];
            grid[16 * idx_sq + 2] = points[idx_pt + numPoints + 1];
            grid[16 * idx_sq + 3] = points[idx_pt + numPoints];

            grid[16 * idx_sq + 4] = points[idx_pt + numPoints * numPoints];
            grid[16 * idx_sq + 5] = points[idx_pt + numPoints * numPoints + 1];
            grid[16 * idx_sq + 6] = points[idx_pt + numPoints * numPoints + numPoints + 1];
            grid[16 * idx_sq + 7] = points[idx_pt + numPoints * numPoints + numPoints];

            grid[16 * idx_sq + 8] = points[idx_pt];
            grid[16 * idx_sq + 9] = points[idx_pt + 1];
            grid[16 * idx_sq + 10] = points[idx_pt + numPoints * numPoints + 1];
            grid[16 * idx_sq + 11] = points[idx_pt + numPoints * numPoints];

            grid[16 * idx_sq + 12] = points[idx_pt + numPoints];
            grid[16 * idx_sq + 13] = points[idx_pt + numPoints + 1];
            grid[16 * idx_sq + 14] = points[idx_pt + numPoints * numPoints + numPoints + 1];
            grid[16 * idx_sq + 15] = points[idx_pt + numPoints * numPoints + numPoints];

        }
    }
}
```

Abbildung 5: Griderzeugung

Zum Schluss wird der letzte Buffer für die Geometrie mit 0en befüllt. Damit ist die Erzeugung der von uns, zur Implementierung des Marching Cubes Algorithmus, benötigten Punktdaten beendet.

Die auf dem Heap allozierten Buffer werden anschließend mit ihren entsprechenden VBOs verbunden und die Daten auf die Grafikkarte geladen. Bei den beiden VBOs für Punkte und Geometrie gibt es noch jeweils einen zusätzlichen Schritt. Über den Aufruf `cudaGLRegisterBufferObject()` werden die VBOs bei der Grafikkarte registriert. Diese Registrierung wird genutzt, um im Anschluss ein Mapping von der Grafikkarte zu den Bufferdaten herzustellen. Hierfür wird der Aufruf wie in Abbildung 6 zu sehen genutzt. Zuvor werden allerdings noch die allozierten Speicherbereiche freigegeben.

```

glBindVertexArray(vao[0]);
// Map OpenGL buffer object for writing from CUDA
float4* dev_points;
float4* dev_geometry;

// Map OpenGL buffers to CUDA
if (cudaGLMapBufferObject((void**)&dev_points, vbo1[1]) != cudaSuccess)
{
    printf("Points buffer could not be mapped to CUDA!\n\n");
}

if (cudaGLMapBufferObject((void**)&dev_geometry, vbo1[2]) != cudaSuccess)
{
    printf("Geometry buffer could not be mapped to CUDA!\n\n");
}

```

Abbildung 6: Mapping von VBOs zur GPU

Durch dieses Mapping ist es möglich die Pointer an die Kernelfunktionen zu übergeben, sodass die Daten direkt geändert werden können.

Nachdem nun alle Vorbereitungen für den Marching Cubes Algorithmus abgeschlossen sind widmen wir uns der Implementierung auf der Grafikkarte mittels CUDA. Zuerst werden die Blockgröße und die Threadanzahl festgelegt. Die Blockgröße ist entweder die von uns festgelegten Variable *maxBlocks = 50* oder die Anzahl Punkte geteilt durch die Anzahl Threads pro Block. Welche der beiden Varianten genutzt wird entscheidet eine min-Funktion. Im Anschluss wird unser erstes Kernel zur Berechnung und Auswertung der Dichtefunktion gestartet.

```

// This kernel checks whether each point lies within the desired surface.
__global__
void points_kernel(float4* points, int size, int func)
{
    unsigned int globalID = blockIdx.x * blockDim.x + threadIdx.x;

    for (int k = globalID; k < size * size * size; k += gridDim.x * blockDim.x) {
        float4 pt = points[k];
        points[k].w = density_func(pt, func);
    }
}

```

Abbildung 7: Ausführung der Dichtefunktion

In Abbildung 7 ist unser erstes Kernel zu sehen. Wir übergeben dem Kernel den Pointer mit dem Mapping von GPU zu VBO, die Anzahl Punkte und welche Dichtefunktion ausgeführt werden soll. Wichtig ist das wir nun eine *globalID* berechnen. Diese gibt an, welchen globalen Index der aktuelle Thread hat. Berechnet wird diese ID anhand des aktuellen Blockindizes multipliziert mit der Anzahl Threads pro Block und das ganze addiert um den Index des aktuellen Threads. D.h. der Vorteil dieser Umsetzung ist es, dass wenn es einmal mehr Punkte als Threads gibt diese trotzdem bearbeitet werden können, da der Sprung von  $k += \text{gridDim.x} * \text{blockDim.x}$ , also der Sprung um die Größe aller Blocks multipliziert mit der Anzahl Threads stattfindet. Dadurch springt man um die maximale Anzahl Threads (Anzahl Blöcke \* Threads pro Block). Es ist wichtig, dass die Implementierung so stattfindet, da for-



Schleifen allgemein eine ungünstige Umsetzung auf Grafikkarten sind. In dieser Umsetzung, aber nötig sind, um eine größere Anzahl Punkte zu bearbeiten. Ist die Anzahl Punkte geringer ist es nicht wie eine Schleife anzusehen, da sie nur einmal (wie sequenzieller Code) ausgeführt wird.

Innerhalb der Schleife wird die Dichtefunktion für einen spezifischen Punkt aufgerufen. Das Ergebnis (in unserem Fall 0 oder 1) wird in die Variable `w` des `float4` gespeichert. Die Implementierung der Dichtefunktion (Abbildung 8) ist übersichtlich und selbsterklärend gestaltet. Je nachdem welche Funktion gewählt wird findet stets eine etwas abweichende Dichtefunktion statt. Dabei basieren alle Funktionen prinzipiell auf der Formel  $x^2 + y^2 + z^2$ . Dadurch ist die geforderte Implementierung einer Dichtefunktion erfolgreich dargestellt. Diese können nun durch beliebiges Abändern und Hinzufügen erweitert werden.

```
__device__ __host__
int density_func(float4& point, int func)
{
    float fun; int flag;
    switch (func) {

        case 0:
            fun = point.x * point.x + point.y * point.y + point.z * point.z;
            flag = (fun < 9);
            break;

        case 1:
            fun = point.x * point.x / 5.0 + point.y * point.y / 3.0
                - point.z * point.z / 7.0;
            flag = (fun < 5);
            break;

        case 2:
            fun = point.x * point.x / 10.0 - point.y * point.y / 3.0
                - point.z / 2.0;
            flag = (fun < 0);
            break;

    }

    return flag;
}
```

Abbildung 8: Dichtefunktion

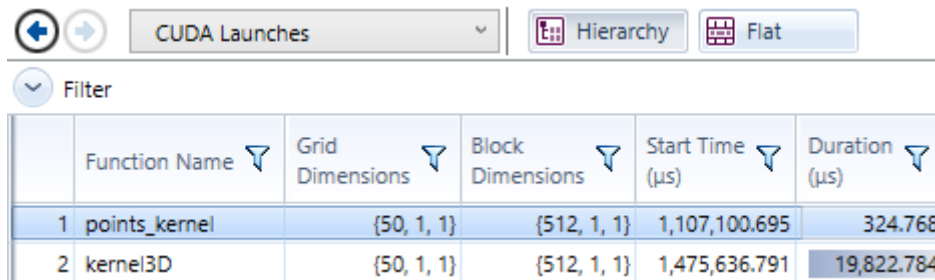
Damit wäre unser erstes Kernel fertig und das zweite Kernel wird gestartet. In diesem werden die Vertices eines Cubes in ein `verts[]` Array geladen. Im Anschluss daran wird wie in der Einleitung schon geschildert der Cubeindex berechnet. Hierfür wird der Dichtewert aus der Variable `w` genutzt. Nun kann durch den Cubeindex der richtige Eintrag aus der Lookup Tabelle geholt werden. Solch ein Eintrag besteht aus einem Array, dass von vorne mit den benötigten Kantenindizes und von hinten mit dem Wert -1 gefüllt ist. -1 bedeutet hierbei einfach das, wenn man über das Array iteriert und der Wert -1 kommt, es keine weiteren Kantenindizes mehr gibt. Wir iterieren dementsprechend über den Eintrag der Lookup Tabelle und holen uns, entsprechend dem Kantenindex, aus einem weiteren Array die zur Kante gehörenden Punkte. Diese Punkte werden interpoliert und das entsprechende Ergebnis (interpolierter Punkt) in die Geometrie gespeichert. Im Anschluss wird die Variable `w` wieder auf 1.0f gesetzt (homogene Koordinate) und das Kernel beendet. Nun kann das



Mapping von GPU zu VBO wieder aufgehoben werden und das Programm kehrt in die main() zurück. Nachdem die Shader initialisiert wurden und das erste Mal gezeichnet wird muss zuerst dann entsprechende VAO gebunden werden. Danach wird der Polygonmodus gesetzt, dieser wäre für vao[0] GL\_FILL und für vao[1] GL\_LINE. Zum Abschluss werden nun die Daten gezeichnet und die Bindung an das VAO aufgehoben (selber Ablauf für beide VAOs).

## 4. Ergebnis

Zuallererst wollen wir einen Vergleich zwischen CPU und GPU Laufzeit erstellen.



	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)
1	points_kernel	{50, 1, 1}	{512, 1, 1}	1,107,100.695	324.768
2	kernel3D	{50, 1, 1}	{512, 1, 1}	1,475,636.791	19,822.784

Abbildung 9: GPU-Laufzeit

In Abbildung 9 ist zu sehen, dass der point\_kernel auf der GPU 325 µs und kernel3D 19.800 µs benötigen. Also insgesamt 20.125 µs = 20 ms = 0,02 sec. Dem gegenübergestellt dauert der Ablauf auf der CPU 2.050.000 µs = 2.050 ms = 2,05 sec (siehe Abbildung 10).

```
Kernel call initialized!
  Number of CUDA blocks: 50
  Number of threads per block: 512
  Time for kernel in microseconds: 2050000.000000
Kernel call ended successfully!
```

Abbildung 10: CPU-Laufzeit

Dies stellt sehr anschaulich den Geschwindigkeitsgewinn um den Faktor 102,5 durch die GPU dar. Dieser Geschwindigkeitsvorteil kommt durch die Ausnutzung der großen Daten-Parallelisierung. D.h. die Punkte können unabhängig voneinander in bspw. der Dichtefunktion genutzt werden. Zudem wurde der Kernelcode eher fein granular erstellt, somit wird pro Thread kein Sammelsurium an Instruktionen ausgeführt, sondern eher kleine Teilaufgaben. Diese Art von Code eignet sich sehr gut für die Ausführung auf der GPU, da dort eher leistungsschwächere Prozessoren als in der CPU vorkommen. Allerdings sind auf der GPU eine Vielzahl solcher Prozessoren vorhanden, was die Geschwindigkeit für fein granulare Aufgaben, im Gegensatz zur CPU, deutlich erhöht. Somit ist gezeigt das der Algorithmus effektiv auf der GPU implementiert werden konnte.

Der Einfluss der Dichtefunktion auf das Ergebnis konnte nicht direkt abgebildet werden, da es Probleme beim Zeichnen der Geometrie gab. Der Fehler konnte leider nicht genau nachvollzogen und verfolgt werden, da es einen OpenGL Fehler ist, welcher auftritt bevor OpenGL überhaupt genutzt wird. Jedoch sollte sich der Einfluss der Dichtefunktion nur dahingehend bemerkbar machen, dass eine andere Umgebung bzw. ein anderes Objekt generiert wird, wenn sich die Dichtefunktion ändert.

Wie in Abbildung 3 zu sehen konnten wir mit den Rendereigenschaften des Zeichnens den gewünschten Effekt der weißen Füllung, gepaart mit den schwarzen Rändern, erfolgreich abbilden. Wie jedoch zuvor schon erwähnt konnten wir aufgrund des OpenGL Fehlers keine abschließende Landschaft generieren. In Abbildung 11 können wir allerdings sehen, dass es uns gelungen einen Cube in mehrere Voxel zu unterteilen.

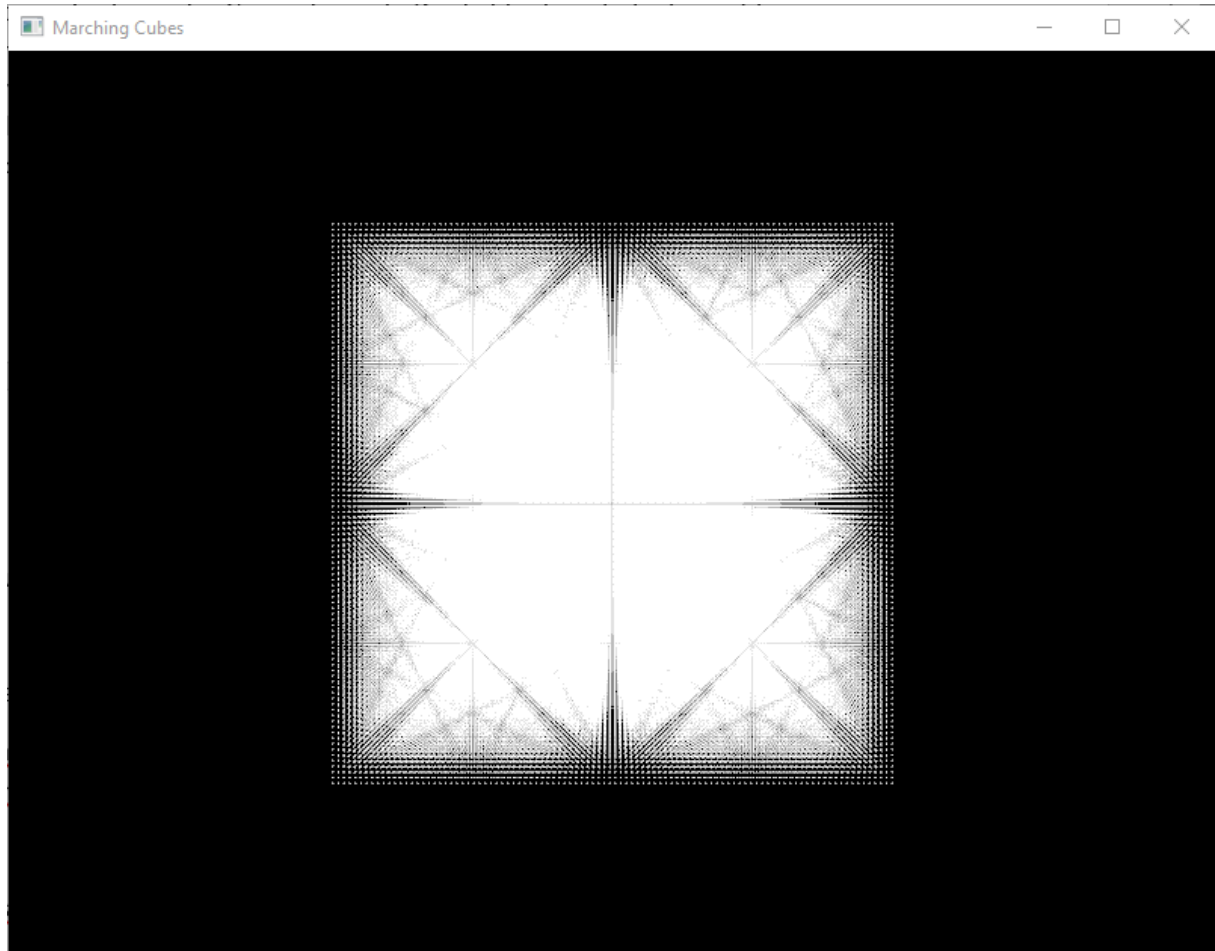


Abbildung 11: Grid

Die Anzahl der zu generierenden Cubes steigert offensichtlich die Last, wenn es auf der CPU ausgeführt wird. Auf der GPU ist eine Erhöhung der Last erst zu merken, sobald die Anzahl der zu berechnenden Punkte die der Anzahl Threads (Hardwarekerne) übersteigt.

## 5. Schlussfolgerung

Zu guter Letzt unsere Schlussfolgerung. Wir sind zu dem Schluss gekommen, dass es nicht schwer ist Berechnungen von der CPU auf die GPU auszulagern. Zumal die Effizienzsteigerung in keinem Verhältnis zu dem Aufwand der CUDA Implementierung steht. Für geeignete Datenstrukturen würden wir somit jederzeit auf die GPU Berechnung setzen. Das Grundprinzip des Programmierens bleibt gleich, jedoch muss man sich mehr Gedanken um die Indizierung machen.

Schwerer hat sich für uns prinzipiell der Umgang mit OpenGL gestaltet. Prinzipiell deshalb, da es uns soweit nicht schwer gefallen ist das Grid bzw. Polygone mit weißer Füllung und

schwarzen Rändern zu generieren. Allerdings hat der nicht nachvollziehbare OpenGL Fehler zu keinem zufriedenstellenden Ergebnis geführt. In Abbildung 12 ist eine durch Marching Cubes generierte Landschaft zu sehen, wie sie von uns hätte implementiert werden können.

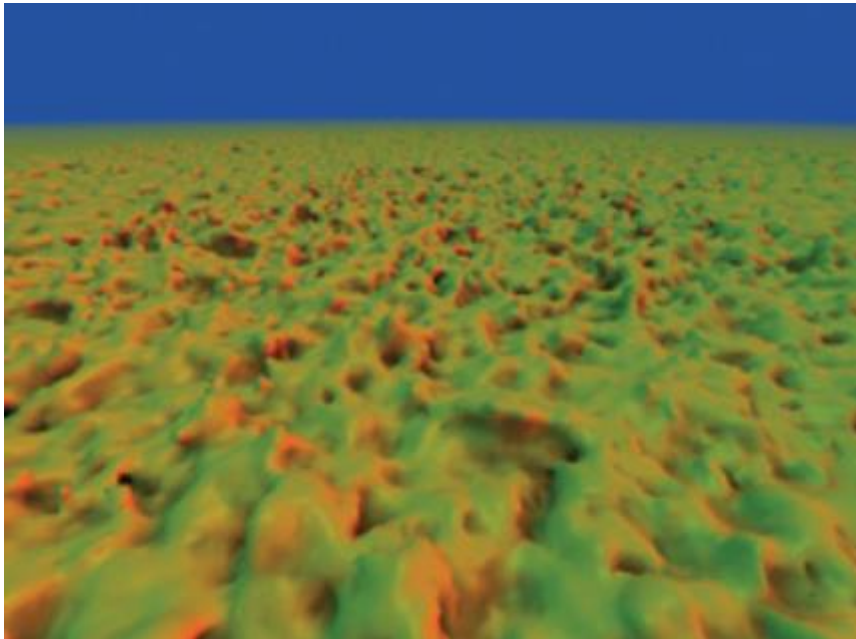


Abbildung 12: Generierte Landschaft

- Quelle für die Abbildungen 1, 2 und 12:  
<https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu>