# Studienarbeit - Mathematik II

Martin Kohnle

Sommersemester 2021

## Marching Algorithmen

## Squares & Cubes

**Anwendung**

**Voxelmodelle**
Daten
Dichte
Datenstruktur

**Polygon Mesh**

**Marching Squares**
Funktionsweise
Verarbeitung
Projekt

**Kombinationen**

**Marching Cubes**
Funktionsweise
Verarbeitung
Projekt

**Fazit**
**Quellen**

# Anwendung

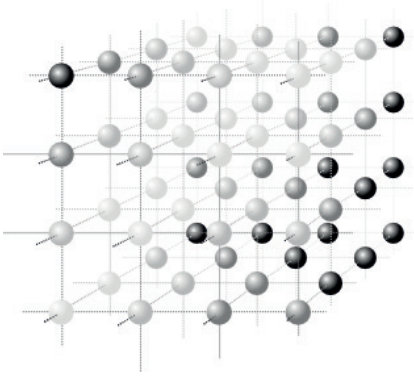
In der heutigen Computergrafik gibt es verschiedene Ansätze, dreidimensionale Objekte zu modellieren. Einer der Populärsten ist die Erstellung eines Polygon Meshs (Drahtgittermodell). In diesem Verfahren werden Dreiecke an ihren Eckpunkten verbunden und zu Oberflächen kombiniert. Ausgehend von trivialen Polygonen lassen sich so komplexe Objekte in Form von Meshs darstellen. Je dichter das Netz wird, desto höher wird der Detailgrad. Bei der Modellauflösung unterscheidet man zusätzlich zwischen den High und Low Poly Modellen. Bisher war das Verfahren für die Problemstellungen in der 3-D Modellierung ausreichend. In diesem Bereich steht die Spielebranche aber vor völlig neuen Herausforderungen. Die Spielwelten sollen größer, dynamischer und interaktiver werden. Als Vorreiter dieser Entwicklung könnte man den Microsoft Flight Simulator 2020 nennen. Mit der manuellen Modellierung ist eine Erstellung von enorm großen und realistischen Objekten schlicht unmöglich. Es benötigt einen neuen Ansatz für diese Anforderungen. In der Wissenschaft wird schon länger ein Verfahren eingesetzt, dass volumetrische Voxel Datenmengen visualisieren kann. Das System iteriert durch ein 3-D Gitter und entnimmt in bestimmten Abständen einen Dichte- oder Farbwert. Disziplinen wie Geologie können so Höhendaten in ein Terrain oder Höhenprofil umwandeln. In der Medizin werden damit MRT Scans dargestellt. Die Voxel Darstellung hat sehr interessante Aspekte, allerdings auch viele Nachteile. So ist der Speicher- und Rechenbedarf deutlich höher und die nachträgliche Bearbeitung eines Modells sehr komplex. Der Grund dafür liegt hauptsächlich daran, dass heutige Grafikkarten nur für Polygonberechnung optimiert sind. Der Marching Cubes Algorithmus verfolgt den Ansatz, die Vorteile aus beiden Systemen zu kombinieren. Die unhandlichen Voxelmodelle sollen durch die einfacheren Polygon Meshs ersetzt und zusätzlich die prozedurale Generierung ermöglicht werden.

# Voxelmodelle

Die Daten müssen entsprechend aufbereitet und in der richtigen Form vorliegen, bevor der Algorithmus angewendet werden kann.
Das wird im Folgenden beschrieben.

### Daten im Voxelgitter:

In der Computergrafik wird der Voxel als Gitterpunkt mit einem Farb- oder Datenwert beschrieben. Räumlich betrachtet ist der Voxel ein dreidimensionaler Pixel. Diese Datenelemente haben für den Algorithmus zwei essenzielle Eigenschaften. Per Iteration in einem 2D oder 3D Gitter werden diese binären Werte als $\rho(x,y,z) = [0, 1]$ ausgelesen und in Polygone umgewandelt. Zusätzlich bilden die Voxelpositionen neue Eckpunkte der Polygone.



[1] Voxelgitter

### Dichtewert (Isowert):

Die in [1] weißen und schwarzen Kugel kennzeichnen die binären Werte des Modells. In einem System mit zwei Zuständen kann man so zwischen solide (weiß) und transparent (schwarz) unterscheiden. Für dieses Projekt liegen diese Werte in einem 2-D bzw. 3-D Array vor und werden dann entsprechend iteriert.
Dazu später mehr.

### Triangle Lookup Table - Datenstruktur

Die Unterschiede zwischen Marching Squares und Marching Cubes liegen hauptsächlich in der Menge der Darstellungsmöglichkeiten (siehe Seite 7).
Marching Squares (2D): 4 Eckpunkte mit  2**4  = 16 Möglichkeiten
Marching Cubes (3D):  8 Eckpunkte mit 2**8 = 256 Möglichkeiten

Bei Squares kann man sich theoretisch die Arbeit machen und diese 16 Möglichkeiten (Symmetriefälle eingeschlossen) selbst austüfteln. Marching Cubes dagegen besteht eigentlich auch nur aus 16 Möglichkeiten, jedoch summiert sich die Anzahl durch Symmetriefälle deutlich auf. Der Triangle Table in Form eines Arrays kommt hier zum Einsatz. Dieser enthält bereits alle möglichen Dreieckkombinationen. Durch die Iteration werden die Zellen durchlaufen, der jeweilige Index ermittelt und eine Oberfläche gerendert. Das Dreieck wird hier als Grundform verwendet, da sich dadurch auch Rechtecke und komplexere Polygone abbilden lassen.

### Beispiel aus dem Datensatz:

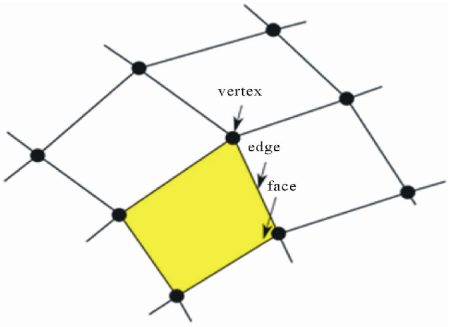
[4] = { 1, 2, 10,  -1, -1, -1,  -1, -1, -1,  -1, -1, -1,  -1, -1, -1,  -1 },

Pro Dreiecksnetz werden 3 Punkte benötigt. Eine Kombination kann aus bis zu 5 Dreiecken bestehen. Die -1 Werte bedeuten, dass hier nichts passiert. Es werden nur {1, 2, 10} verarbeitet.

Musterbeispiel für die Dreiecksnetze:
[x] = { a1, b1, c1,  a2, b2, c2,  a3, b3, c3,  a4, b4, c4,  a5, b5, c5,  -1 }

## Polygon Mesh

Das Polygon ist eine 2-D Form mit geraden Linien und geschlossenen Seiten. Der Begriff Poly bedeutet in diesem Sinne, dass die Form beliebig viele Seiten haben kann. Dreieck, Hexa- und Oktagon gehören somit zur Familie der Polygone. Sind mehrere dieser Formen über Punkte (Vertices) verbunden, spricht man von einem Polygonnetz oder auch Poly Mesh genannt. Die Vertices enthalten die 3-D Koordinaten und verbinden zwei oder mehrere Edges miteinander. Die Edges (Linien) grenzen das Oberflächensegment (Face) ein.

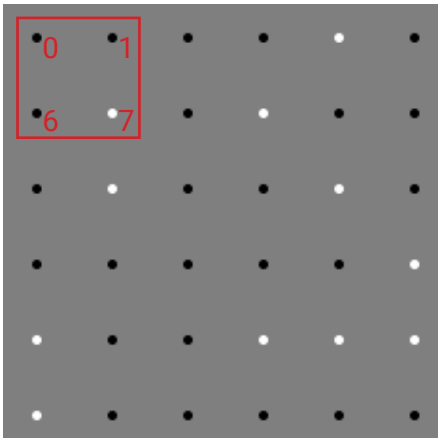


# Marching Squares

Bevor man sich dem Marching Cubes zuwendet, ist es hilfreich, sich nochmals den Marching Squares Algorithmus zu vergegenwärtigen. Das Verständnis wird hier durch die 2-D Darstellung mit nur 16 möglichen Kombinationen erleichtert. Dieser Teil des Projekts wurde mit Processing realisiert. In diesem Beispiel wird ein statisches Grid verwendet. Den Begriff Voxel verwende ich zur Vereinfachung auch im 2-D, obwohl man eigentlich nur von Pixel bzw. Punkten spricht.

**Grid erzeugen:**
Wie im vorherigen Kapitel beschrieben, benötigt man zunächst ein 2-D Grid in dem Daten ausgelesen werden können. Bestehend aus den Arrays col [ ] (column) und row [ ].

**Voxel generieren:**
Für die Erstellung der Voxel wird ein zusätzliches Array bin_color[ ] verwendet. Die Werte von bin_color [ ] werden durch Iteration von col[ ] und row[ ] generiert. Bei jedem Durchlauf wird zufällig für jede Position entschieden, ob der Wert 0 oder 1 gewählt wird. Mit anderen Worten, ob dieser Voxel ein Polygon rendern wird oder nicht. Für die Demonstrierung werden die Voxelwerte einfach per Zufall erzeugt. In der realen Werte, würde man an dieser Stelle bereits erhobene Daten auslesen. In der Abbildung sieht man die Selektierung des ersten „Squares“ mit vier Voxel (0, 1, 6, 7).




**Marschieren:**
Der Algorithmus sieht vor, dass jeweils vier benachbarte Voxel ausgelesen werden. Iteriert man zeilen (k)- und dann spaltenweise (i), würde es folgendermaßen aussehen:

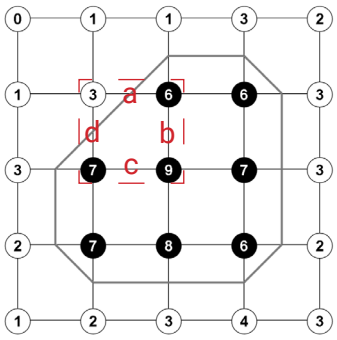
Voxel 1 = [ k ] [ i ], Voxel 2 = [ k+1 ] [ i ],
Voxel 3 = [ k+1 [ i+1 ] und Voxel 4 = [ k ][ i+1 ]

Durch dieses Verfahren ist hier die Rede von dem Marching Square, das sozusagen als Sliding Window über die Zellen marschiert und den Index ermittelt. Damit zusammenhängende Formen entstehen können, erfolgt das Marschieren pro einen Schritt in Zeilen und Spalten. Somit ist die 2. Spalte im aktuellen Square, die 1. Spalte im nächsten usw.

**Index berechnen:**
Aus dem bin_color [ ], welches oben zufällig mit 0 und 1 befüllt wurde, kann jetzt der Index berechnet werden. Betrachtet man das in der Iteration, besteht bin_color [ ] aus den x und y Positionen von col[ ] und row[ ]. Der abgebildete Term wurde so gewählt, dass der minimale Wert 0 und der maximale 15 darstellt. Auf diese Weise lässt sich der Wert einfach auf den Index abbilden. Anhand der obigen Grafik:

Index[ 1 ] = return (0 * 8 + 0 * 4 + 0 * 2 + **1 * 1**)  ---> Kombination [ 1 ] -->

**Intersections und Vektoren:**
Damit die Edges der Polygone korrekt miteinander verbunden werden können, benötigt es noch einen weiteren Parameter - die Intersections. Wie in der Grafik zu sehen ist, werden diese genau zwischen den Voxelpunkten platziert. In der Funktion werden die Intersections als Vektoren berechnet, um diese in einem Schritt als x und y Positionen auslesen zu können. In einem Beispielschritt sollen die Intersections für das Square (3, 6, 7, 9) ermittelt werden. Vektor a liegt beispielsweise auf der Linie zwischen Voxel 3 und 6. Vektoren b und c befindet sich innerhalb des Objektes. Vektor d liegt auf der Linie zwischen Voxel 3 und 7. Im Uhrzeigersinn werden so vier 2-D Vektoren berechnet.




**Funktion im Überblick:**

```
void point_status() {
  for (int i = 0; i < row.length; i++) {
    for (int k = 0; k < col.length; k++) {
      float x = col[ k ];
      float y = col[ i ];
      PVector a = new PVector(x + scale / 2, y);
      PVector b = new PVector(x + scale, y + scale / 2);
      PVector c = new PVector(x + scale / 2, y + scale);
      PVector d = new PVector(x, y + scale / 2);
      int status = calc_status(bin_color[ k ][ i ], bin_color[ k + 1 ][ i ],
        bin_color[ k + 1 ][ i + 1 ], bin_color[ k ][ i + 1 ]);

      switch (status) {
        case 0:
          break;
        case 1:
          line(c.x, c.y, d.x, d.y);
          break;
        case 2:
          line(b.x, b.y, c.x, c.y);
          break;
            ...
```

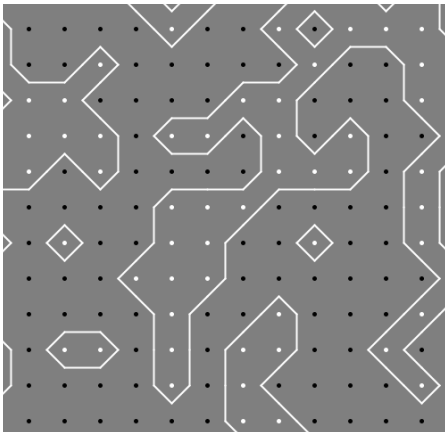
Iteriere durch Zeilen und Spalten

Übergebe Werte (x,y)

2D Vektoren für dieses Voxel „Square“ berechnen.

Index [0, 15] berechnen.

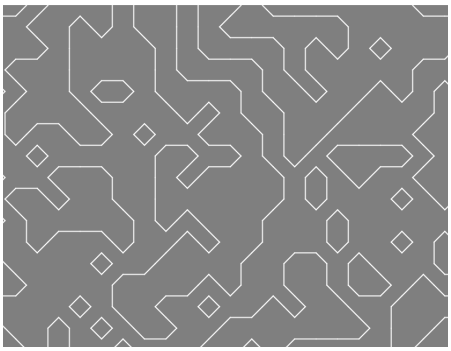
Case entspricht dem Index und rendert ein Liniensegment.



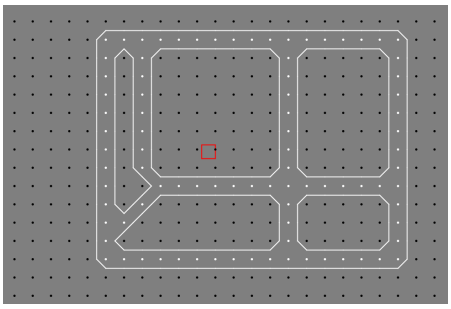
Möglicher Output

**Anwendung:**
In Processing habe ich eine Klasse marchingSquares erstellt, die drei verschiedene Funktionen ausführen kann. Das staticGrid war die Erste davon und wurde in diesem Kapitel ausführlich erklärt. Zusätzlich gibt es noch die Varianten animation() und den brush() für Interaktion.

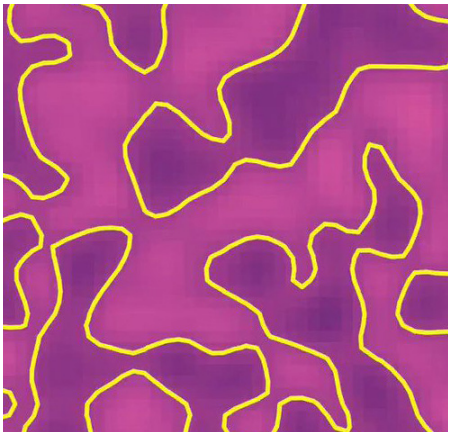
**Animation:**
In dieser Funktion ist das Voxeldatenset zu anfangs nicht vorhanden. Durch Iteration wird für jedes „Square“ in Echtzeit ein zufälliger binärer Wert gewählt und das entsprechende Case gezeichnet. So entsteht der Effekt, dass eine Art Drucker das Gebilde prozedural aufbaut. Nach jedem Durchlauf beginnt die Iteration von neu und nimmt zufällige Veränderungen vor. Obwohl die Visualisierung etwas trivial scheint, lässt sich auf diese Weise eine Spielwelt erzeugen, die sich über die Zeit dynamisch verändern kann.



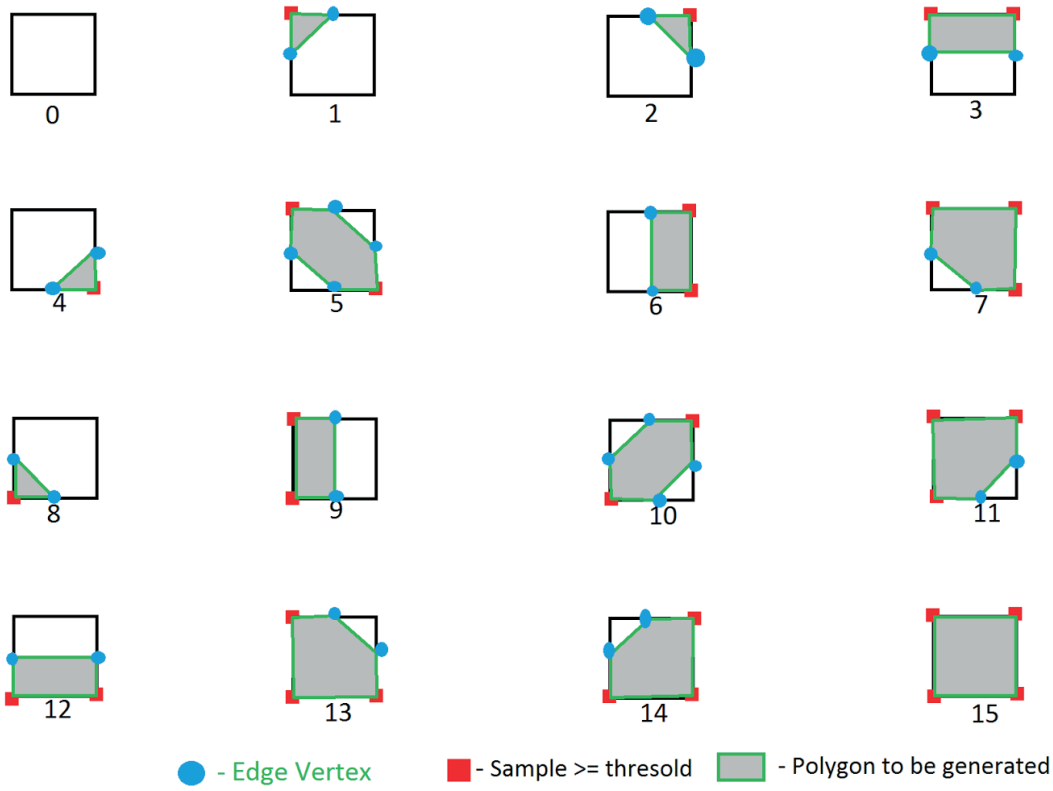
**Interaktion**:
Auch in dieser Funktion ist das Datenset zunächst leer.
Die Idee ist, dass der User per Interaktion bzw. Mousehover über die Voxel fährt und dadurch die Objekte entstehen. In dieser Spielwelt kann der User nun selbst Einfluss nehmen. Ein Konzept dieser Art könnte man sich zur Generierung von Straßen in einem Städtebau Simulator vorstellen.



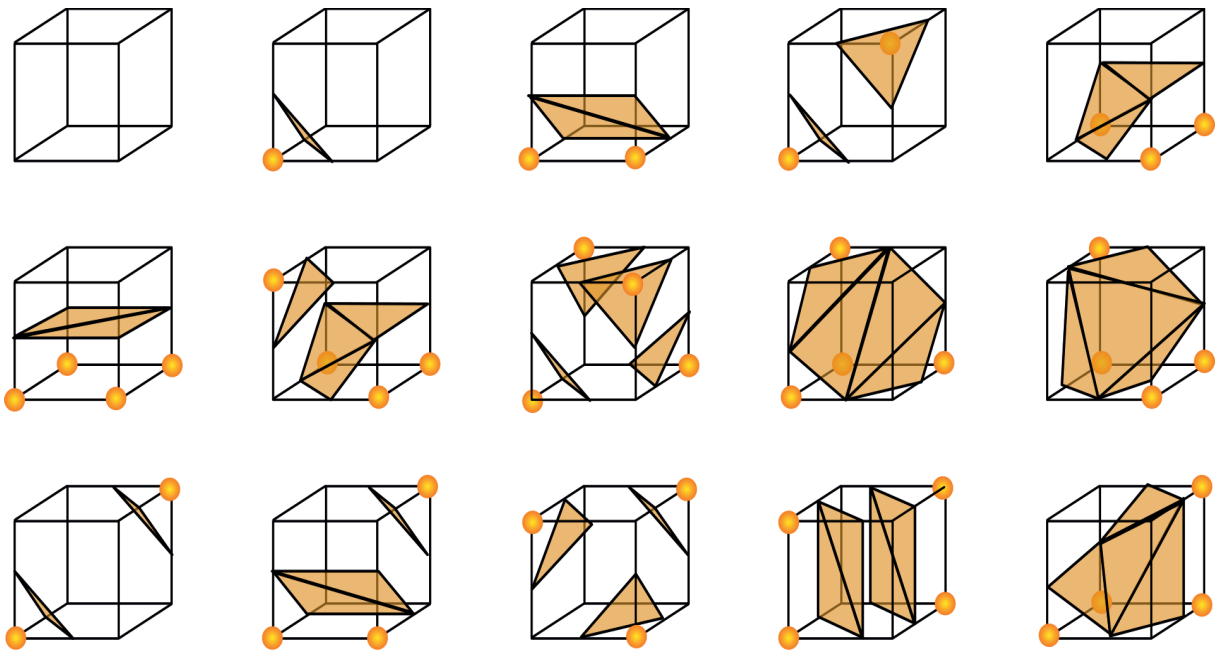
**Ausblick:**
Bei meinem kurzen Ausflug in die Marching Squares habe ich nur an der Oberfläche der Möglichkeiten gekratzt. Mit Animation und Interaktionen lassen sich dynamische, endlose und abwechslungsreiche 2-D Welten prozedural generieren. Der große Vorteil dabei ist, dass keine manuelle Modellierung notwendig ist. Spannend wäre es auch, den Algorithmus mit Noise Generierung bspw. Open Simplex Noise zu erweitern.



# Marching Squares - Kombinationen




# Marching Cubes - Kombinationen



# Marching Cubes

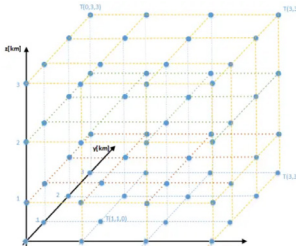
Grundsätzlich ist der Marching Cubes Algorithmus identisch zum Marching Squares. Dennoch gibt es natürlich eine Reihe an Unterschieden und Erweiterungen, die im Folgenden erklärt werden. Für die 3-D Visualisierung habe ich mich für die Unity Engine und C# entschieden.

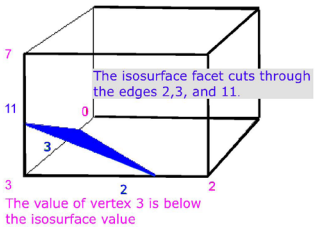
### Iso Surface

Der Algorithmus wurde entwickelt, um Iso Surfaces als Triangle Mesh möglichst genau nachzubilden. Diese Isoflächen verbinden im dreidimensionalen Punkte gleicher Merkmale miteinander. Iso Werte können so in einem Skalarfeld verschiedene Materialien oder Typen unterscheiden. In der Medizin werden mit Isoflächen aus Datensätzen mit Dichtewerten Organoberflächen und Knochenstrukturen nachgebildet. Für diese Anwendung liegen die Datensätze mit binären Werten vor.

### Iso Level

In jeder Volumenzellen soll die Isofläche nachkonstruiert werden. Die Isofläche kann dabei die Zelle schneiden, darüber oder darunter verlaufen. Liegt beispielsweise ein Vertex über dem Iso Level und eine anderer darunter, lässt sich daraus ableiten, dass die Fläche durch die Intersection zwischen beiden Vertices verläuft. Wie bei den Marching Squares bereits erklärt, liegt die Intersection immer genau zwischen zwei Vertices. Es lässt sich allerdings noch präziser formulieren. Die Position zwischen den beiden Vertices wird linear interpoliert. Damit entspricht das Verhältnis der Länge von zwei Vertices, dem Verhältnis des Isoflächenwertes zu den Werten an den Intersections. Auf der nachfolgenden Seite wird dies an einer Beispielgenerierung motiviert.



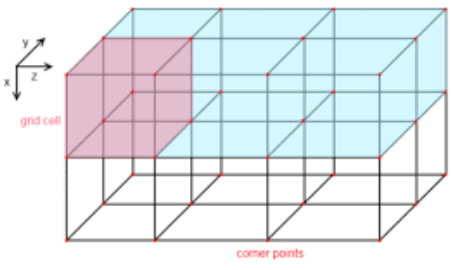



### Datenstruktur

In einem vorherigen Kapitel wurde bereits beschrieben, dass die Kombinationen nun nicht mehr selbst festgelegt werden, sondern in einem vorgefertigten Triangulation Table [0, 255] vorliegen. Das hat hauptsächlich den Hintergrund, Zeit und Aufwand zu sparen, die es benötigen würde, diese eigenständig zu ermitteln.

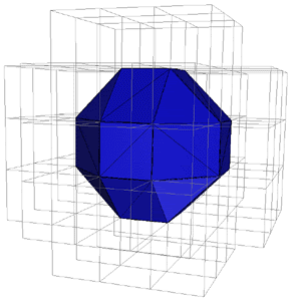
### Volumen erzeugen

Der Ablauf folgt dem gleichen Prinzip wie bei Marching Squares. Zunächst wird ein Voxelvolumen angelegt. Bestehend aus den Arrays grid_x[ ], grid_y[ ], grid_z[ ]. Dabei wird zunächst in die Tiefe und dann zeilen- und spaltenweise iteriert. Bei der Generierung wird wieder jeder Voxel zufällig binär belegt. Diese Werte sollen wieder in einem zusätzlichen Array color[x, y, z] gespeichert werden. Später habe ich festgestellt, dass durch die absolut zufällige Generierung der Voxel im 3-D zu viele einzelne Polygone entstehen, die sehr nach Fragmenten aussehen (obwohl sie technisch keine sind). Die Generierung habe ich dann anschließend durch bestimmte Parameter eingegrenzt, um ein zusammenhängendes Mesh zu erzeugen.




Iteration durch das Voxelvolumen

**Konstruieren:**
Während der Iteration „marschiert“ der Cube durch die Volumenzellen und berechnet den jeweiligen Index. Dieser wird aus acht benachbarten Voxel - die ein Volumen einschließen - berechnet.

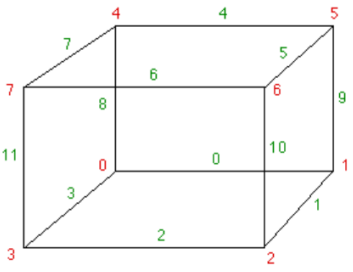


```
int CalcStatus(int a, int b, int c, int d, int e, int f, int g, int h)
        return(128 * h + 64 * g + 32 * f + 16 * e + 8 * d + 4 * c + 2 * b + 1 * a)
```

**Intersections:**
Der Index wird dann in eine Funktion übergeben, welche das entsprechende Dreiecksnetz aus dem Triangulation Table zurückgibt. Damit das Netz nun korrekt gezeichnet werden kann, müssen wieder die Intersections zwischen den Vertices festgelegt werden. In diesem Fall müssen 12 Intersections in Form von 3-D Vektoren angelegt werden. Dies kann entweder auf eigene Weise oder mit Formel berechnet werden. Formel für Intersection P:

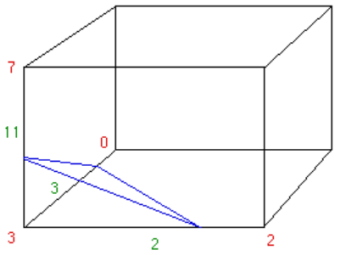
P1, P2 = Edges
V1, V2 = Skalarwerte der Vertices
P = P1 + (Isowert - V1) * (P2 - P1) / (V2 - V1)

Ein Beispiel für die Intersection[0] zwischen Vertex 0 und 1, wenn der Ursprung in (0, 0, 0) liegt:
0.5 Einheiten in positive X - Richtung
Y - Achse bleibt unverändert
1 Einheiten in positive Z - Richtung




Edges
Vertices

**Beispielgenerierung mit einfachem Polygon:**
In diesem Fall ist nur ein Voxel (3) aktiv und somit unterhalb des Iso Levels. Der Schwellwert ist in diesem Fall [4].
Die Berechnung des Index würde wie folgt aussehen:




return(128 * **0** + 64 * **0** + 32 * **0** + 16 * **0** + 8 * **0** + **4 * 1** + 2 * **0** + 1 * **0**)

Die Funktion gibt den Wert 4 zurück und wird dem Triangulation Table als Index [4] übergeben.

{3, 2, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

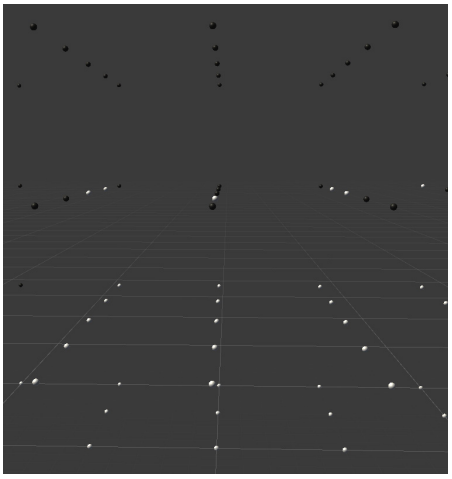
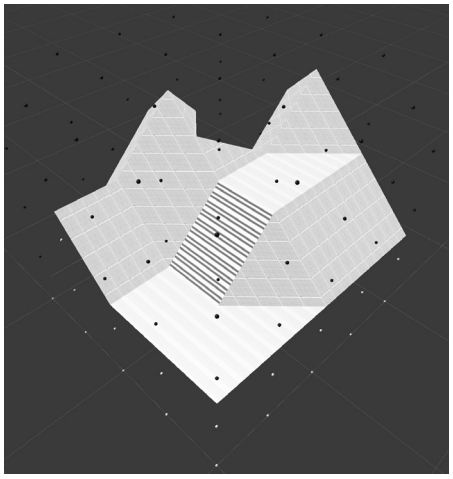
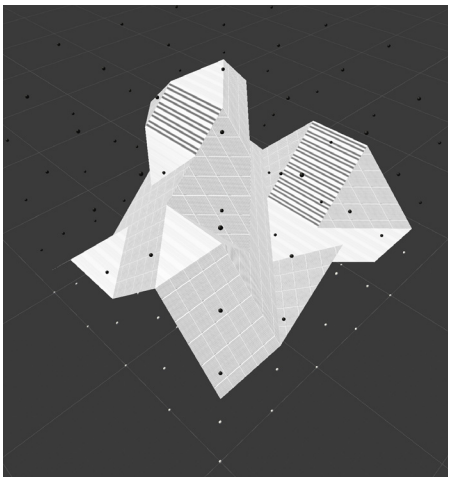
Die Eckpunkte des Dreiecks werden durch die Edges auf jeder Achse festgelegt. Somit wäre für diese Volumenzelle ein Surface erzeugt.

**Algorithmus im Überblick:**

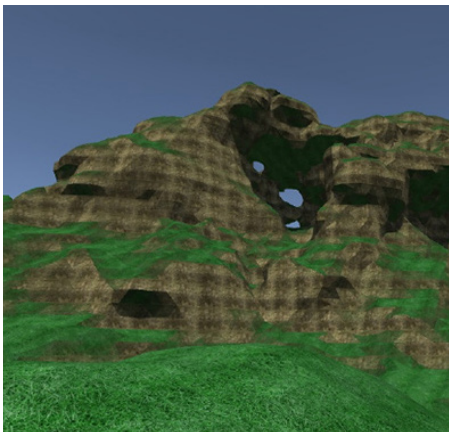
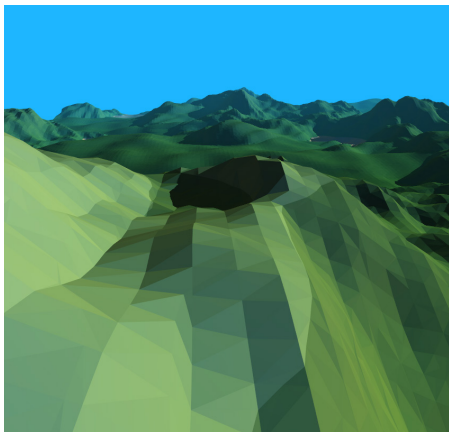
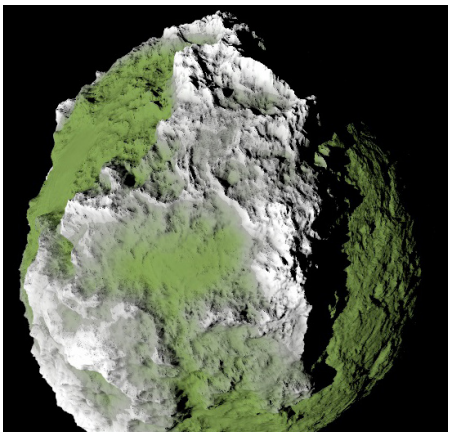
1. **Generierung eines Voxelgitter (Skalarfeld)**
    a. **Iteriere durch das Volumen (x, y, z)**
        **Zufällige Generierung der Voxel**

2. **Marching Algorithmus**
    a. **Iteriere durch das Volumen (x, y, z)**
        **Berechne Intersections**
        **Berechne Index**
        **Render die Kombination[Index]**

**Terraingenerierung**:
Parallel zur Map Generierung mit Marching Squares wollte ich jetzt ein Low Poly Terrain erstellen. Dazu werden auf mehreren Ebenen die Voxel zufällig belegt. Per Mausklick wird eine neue Generierung erzeugt. Mit diesem Konzept wäre eine begehbare Welt möglich, die sich dynamisch verändert.

**Ausblick:**
Bei den Marching Cubes habe ich mich sehr auf die Terraingenerierung fokussiert, da ich hier sehr viel Potenzial für große und dynamische Spielewelten sehe. Interessant wären Spiele, die ähnlich wie Minecraft (Voxelterrain) sehr komplex aufgelöste Welten erzeugen könnten.

## Fazit

In Zukunft wird es sicherlich einen erhöhten Bedarf an komplexen und großen Objekten geben. Sowohl bei Spielen als auch bei Animation wird dies zu einem erheblich größeren Arbeitsaufwand führen, welcher durch diese prozeduralen Technologien beherrschbar bleibt. Nach der Programmierung des Marching Algorithmus, hat sich für mich herausgestellt, dass die eigentliche Arbeit im sinnvollen Generieren der Voxel liegt. Die zufällige Erstellung sollte deshalb in einem Rahmen realer Werte stattfinden.

# Quellen

https://www.sciencedirect.com/topics/computer-science/marching-cube-algorithm
https://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html
http://paulbourke.net/geometry/polygonise/
https://de.wikipedia.org/wiki/Voxel
https://de.wikipedia.org/wiki/Isofl%C3%A4che
https://en.wikipedia.org/wiki/Marching_cubes
https://en.wikipedia.org/wiki/Marching_squaresmar
https://journal-bcs.springeropen.com/articles/10.1186/s13173-019-0086-6
https://www.youtube.com/watch?v=M3iI2l0ltbE