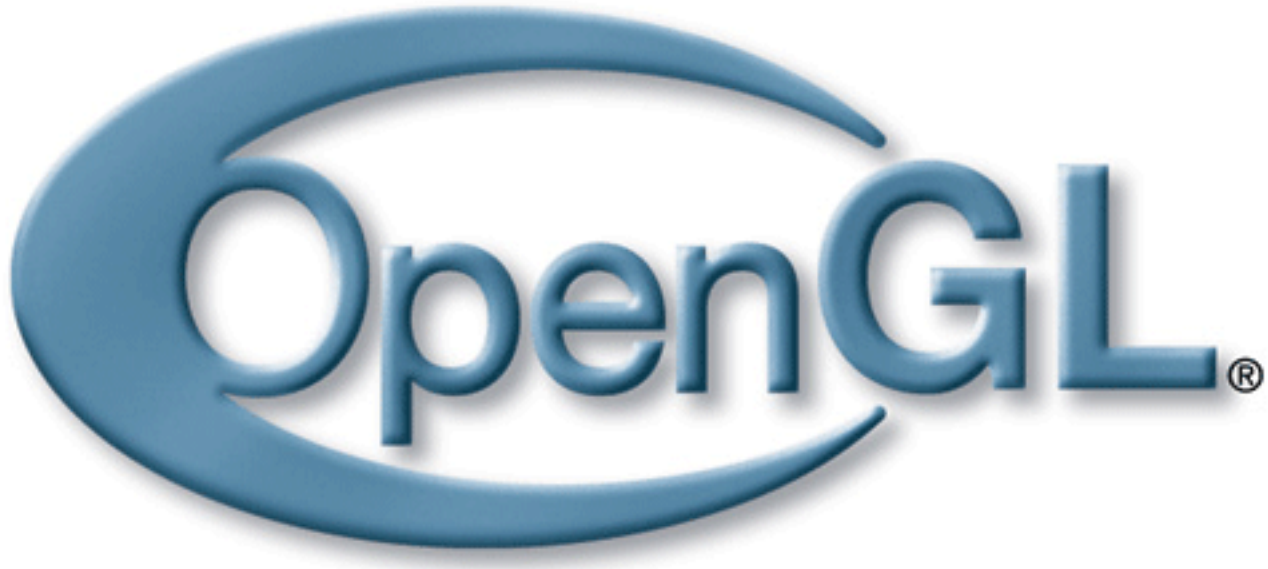
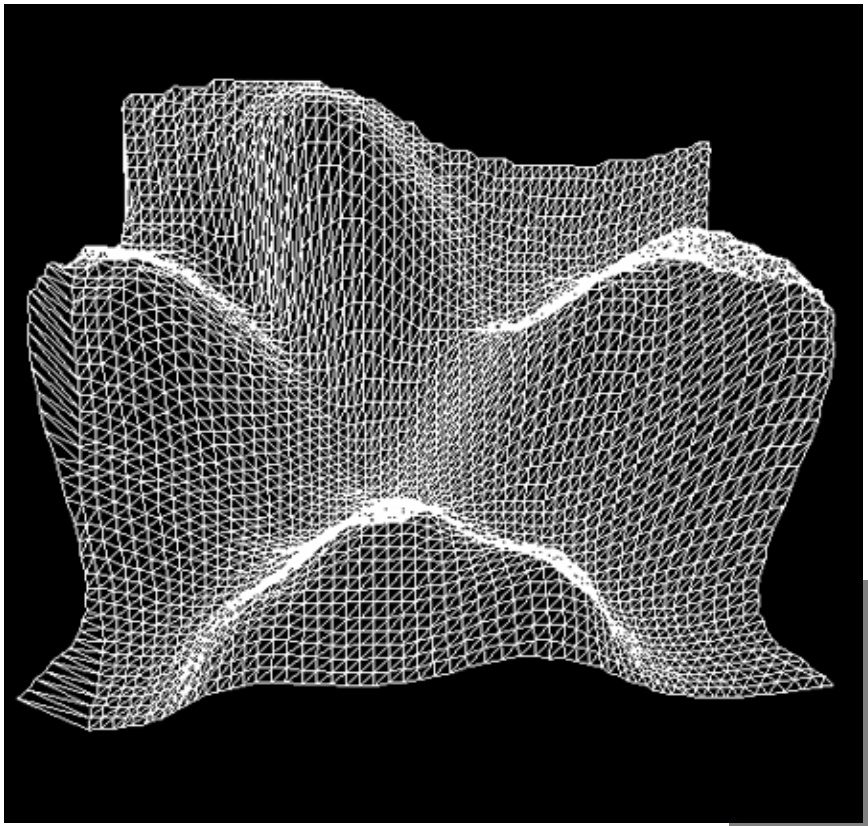
OpenGL - Teil 5

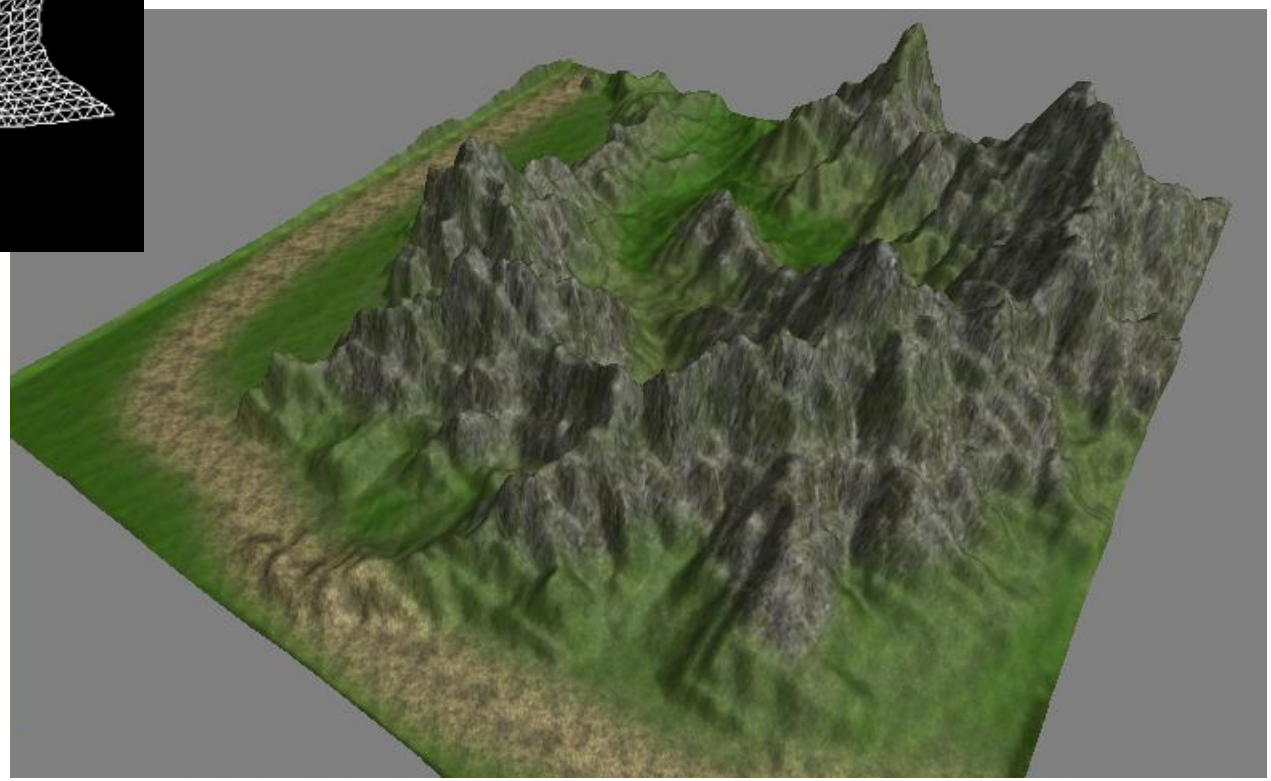
- Terrains mit OpenGL -



Bernhard Bittorf und Andy Reimann

Das finale Produkt

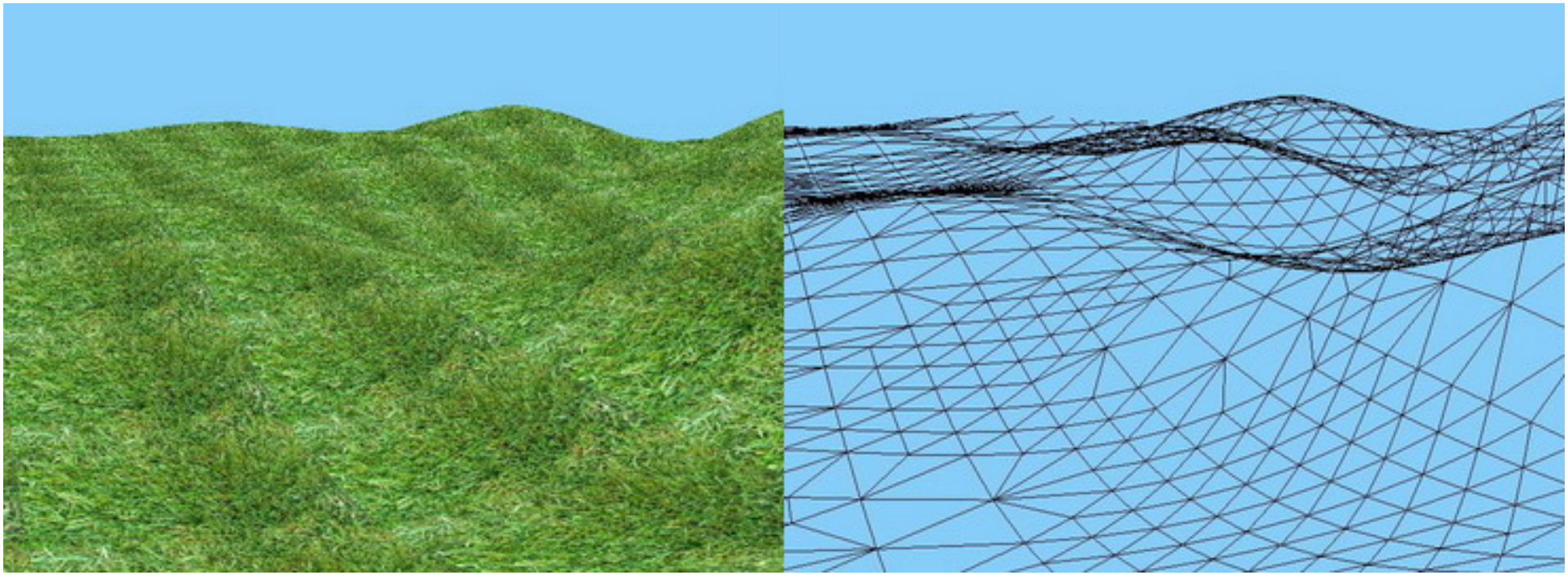




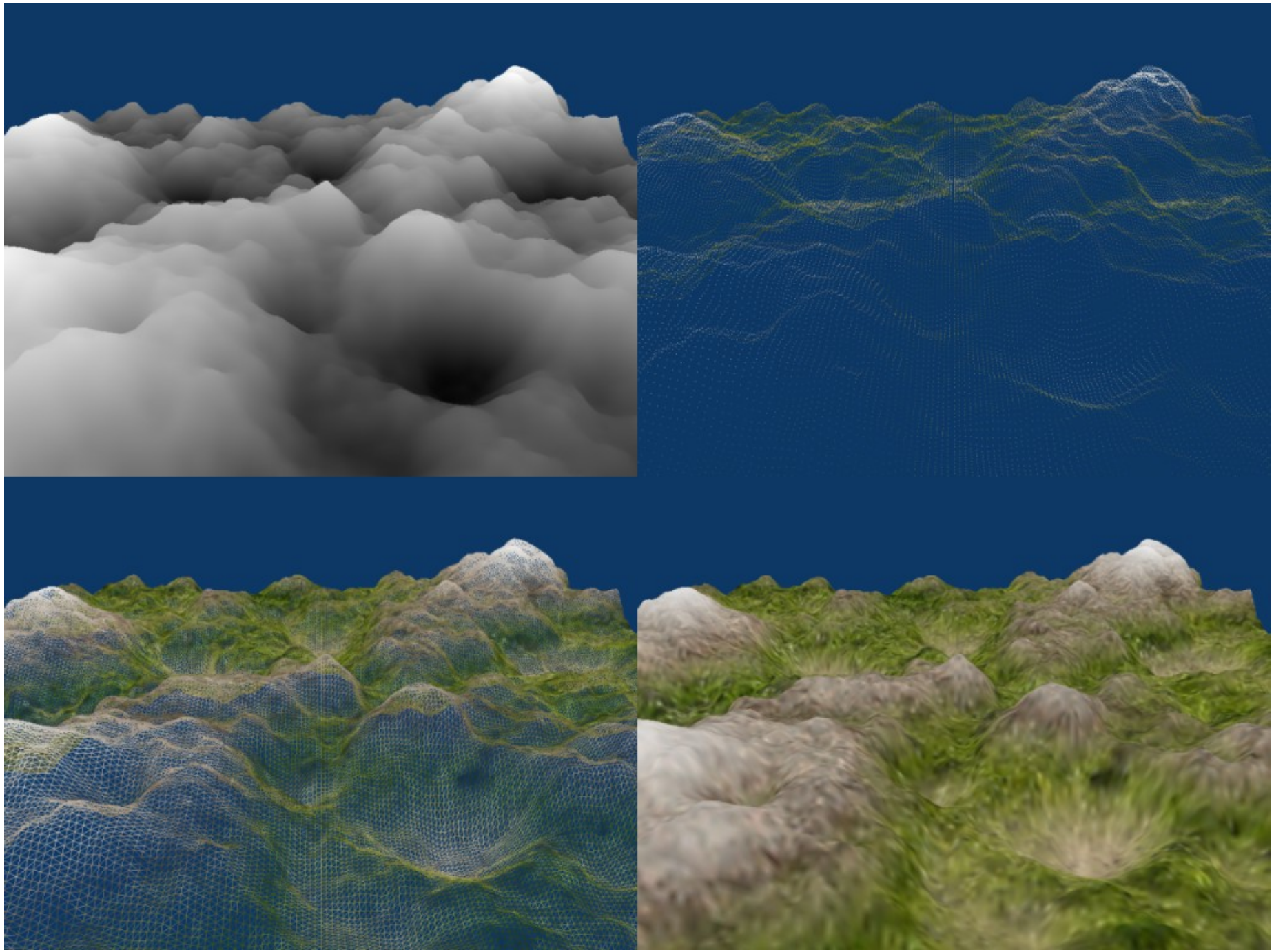
Verschiedene Algorithmen

- Bruteforce
- Terrain Patches mit LOD
- ROAMing Terrain - Real-time Optimally Adapting Meshes
- Geometry Clipmaps
- ...

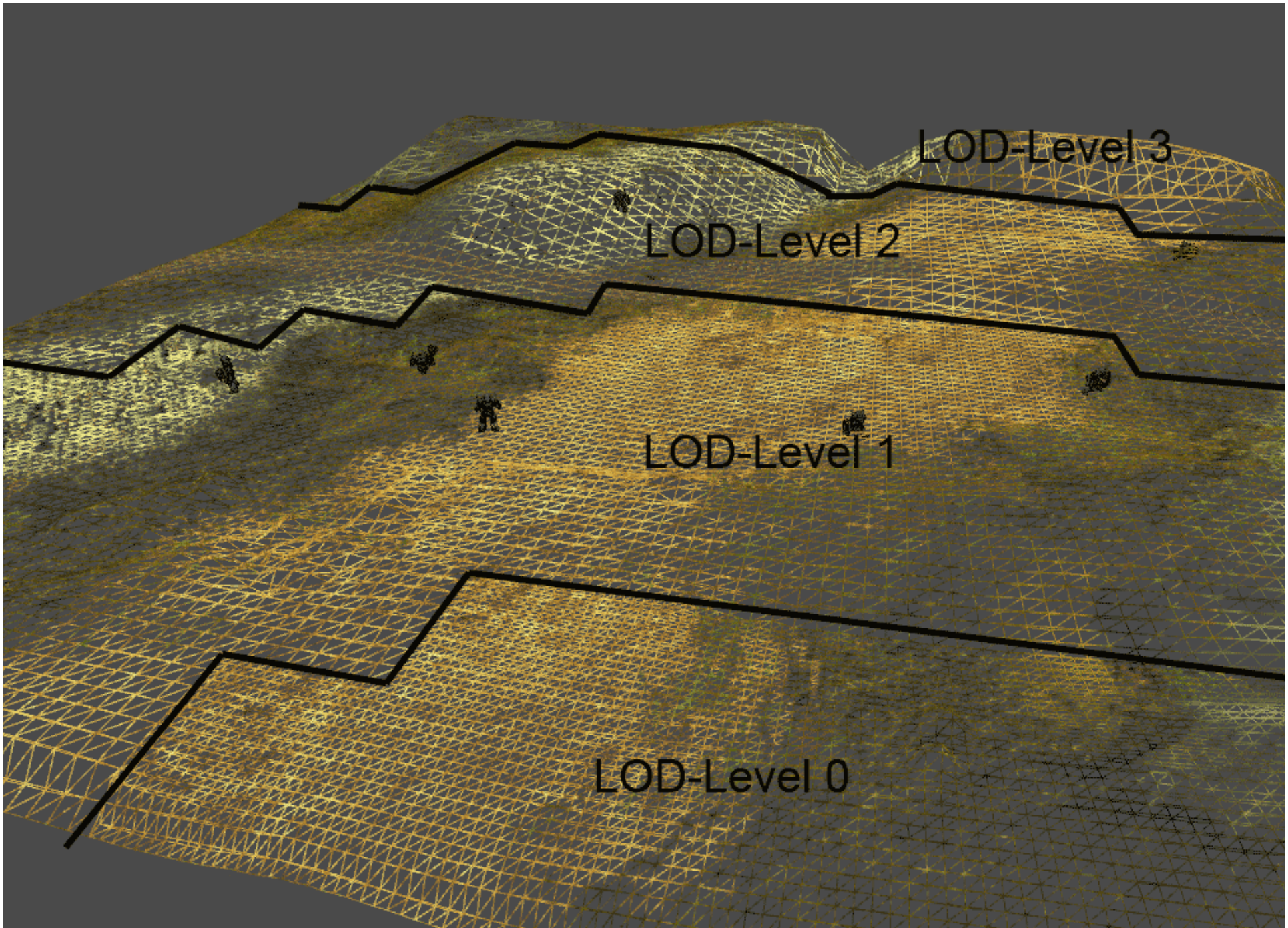
Real-time Optimally Adapting Mesh



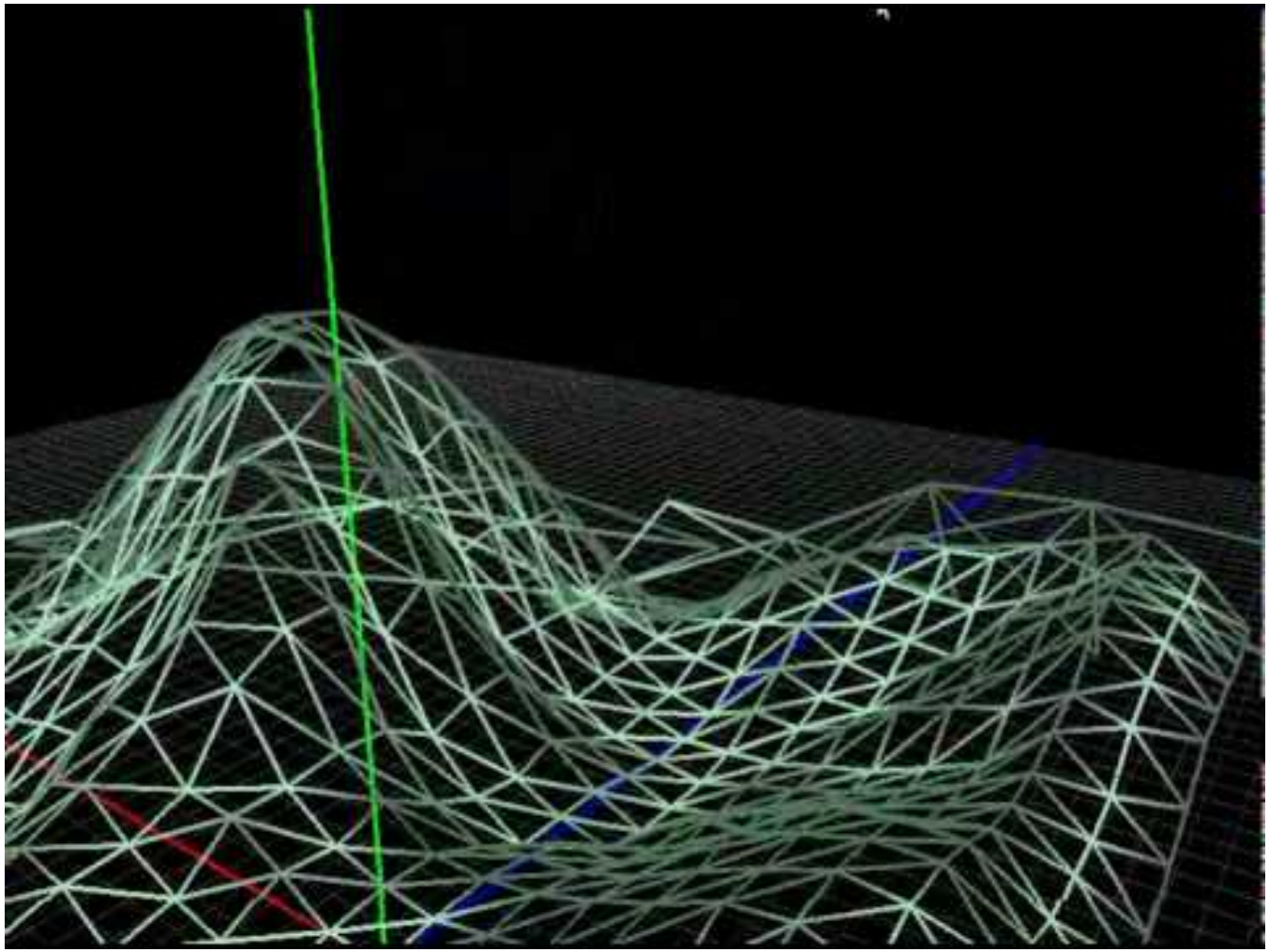
Bruteforce



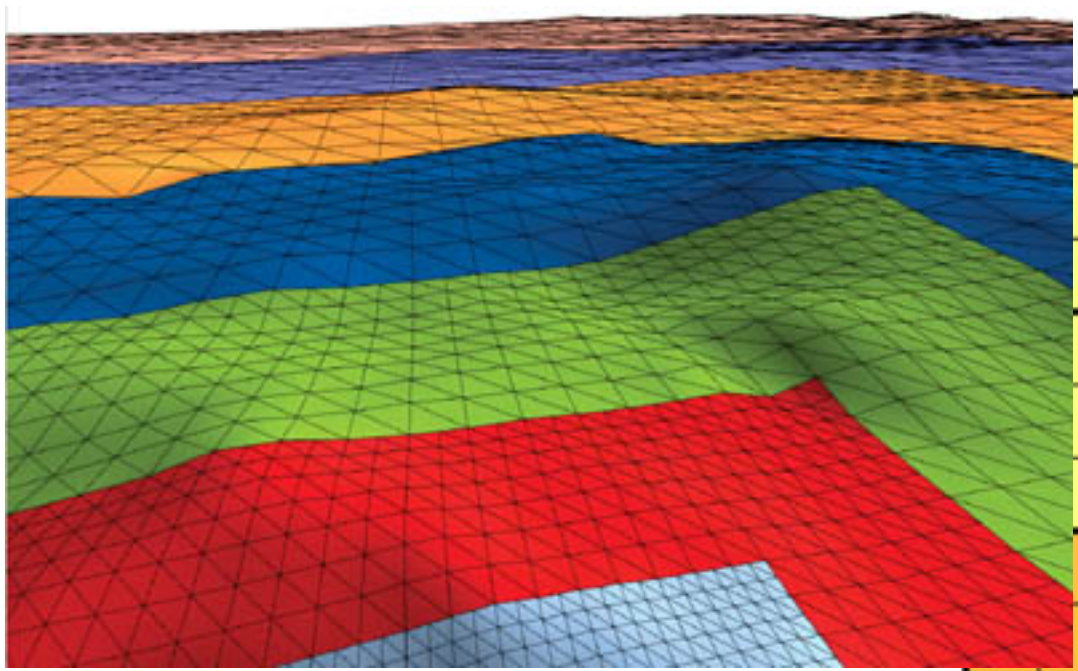
Terrain Patches mit LOD

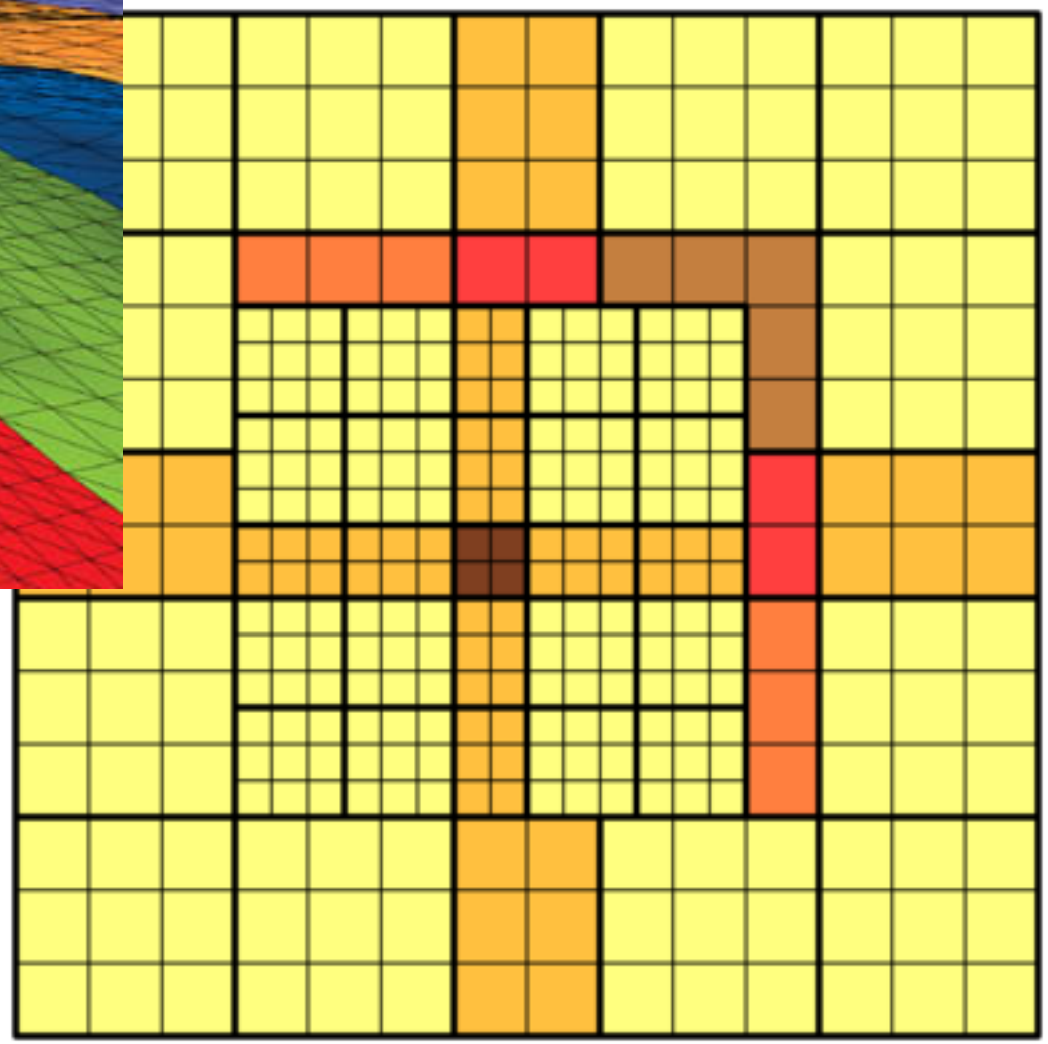


Quadtree Chunked LOD

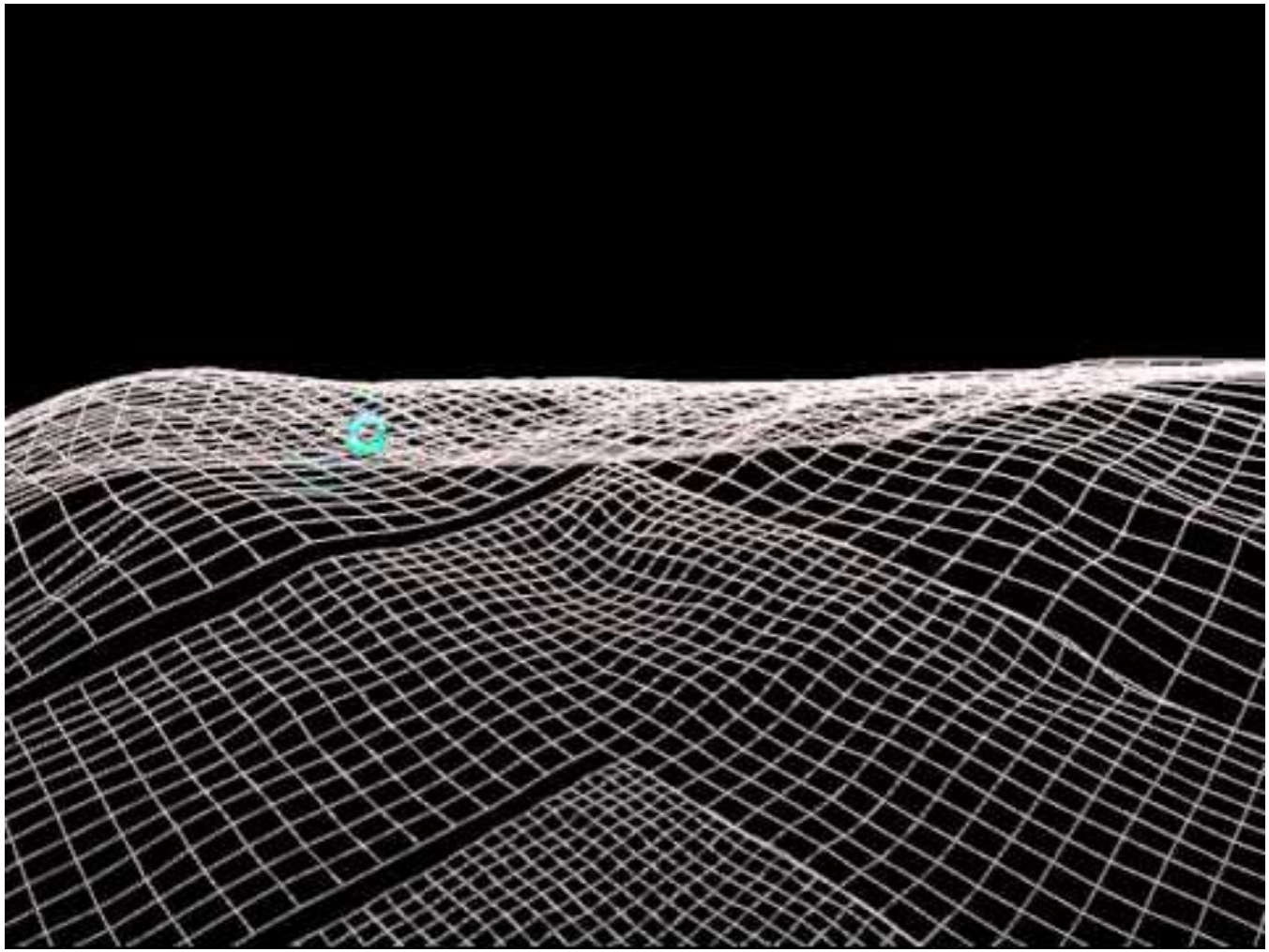


Geometry Clipmaps





Geometry Clipmaps

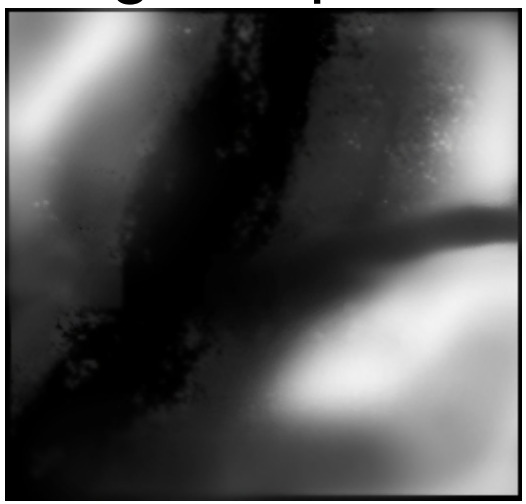


Verschiedene Algorithmen

- Bruteforce - Die nächste Aufgabe
- Terrain Patches mit LOD
- ROAMing Terrain - Real-time Optimally Adapting Meshes
- Geometry Clipmaps
- ...

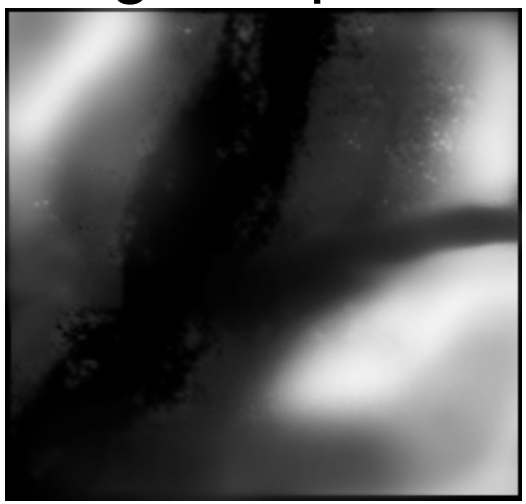
Die Terrain-Renderpipeline - Komplex

Heightmap

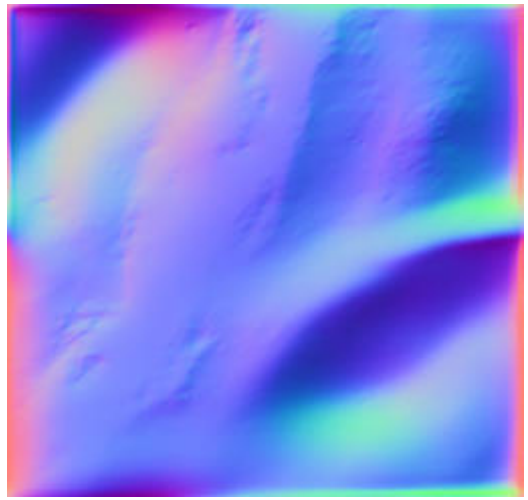


Die Terrain-Renderpipeline - Komplex

Heightmap

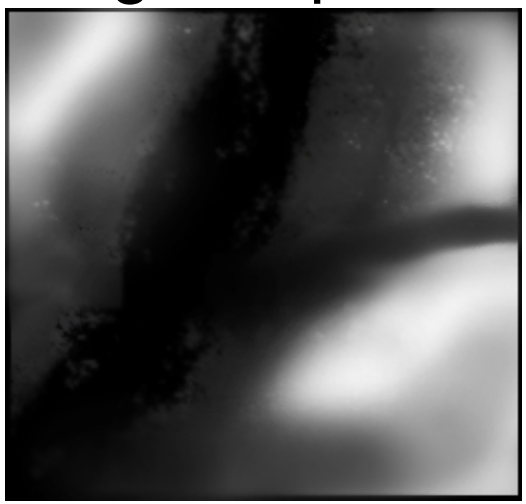


Normalmap

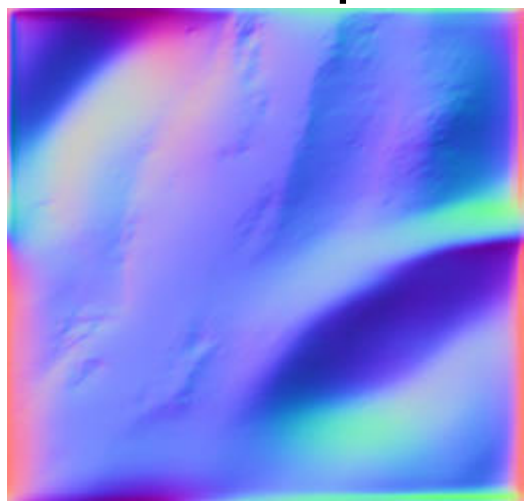


Die Terrain-Renderpipeline - Komplex

Heightmap



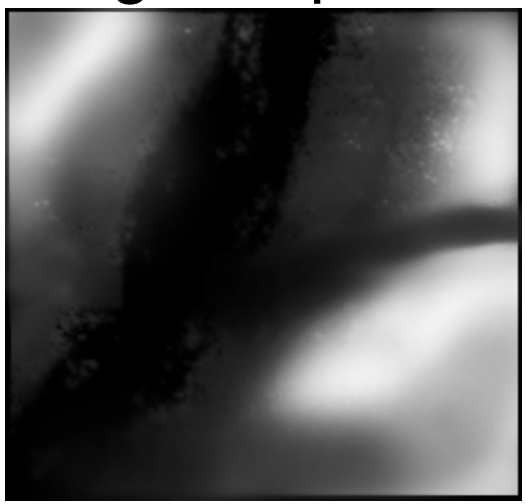
Normalmap



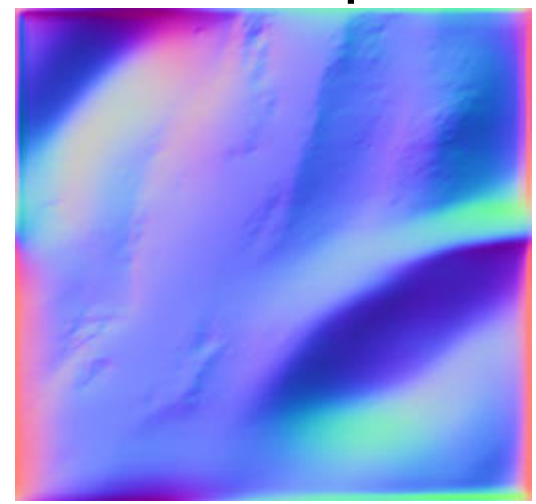
Detailtexture

Die Terrain-Renderpipeline - Komplex

Heightmap

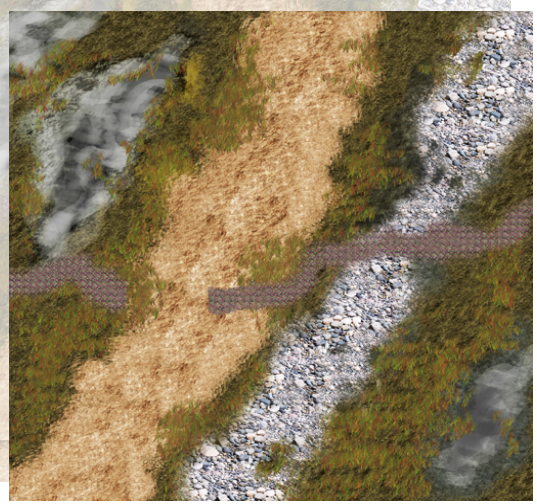


Normalmap



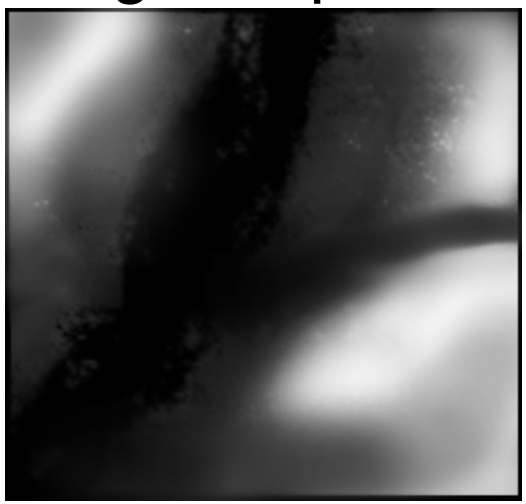
Detailtexture

Colortextures

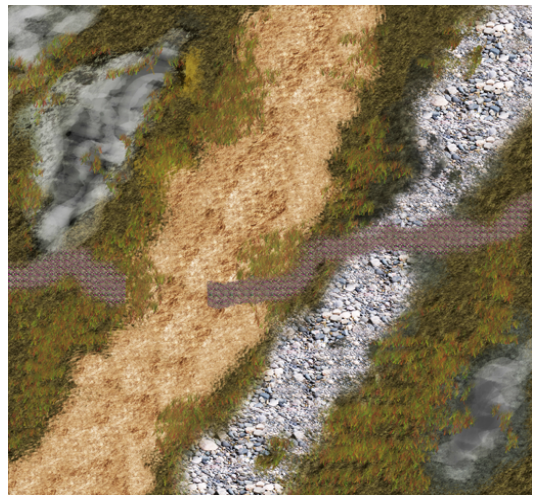


Die Terrain-Renderpipeline - Simple

Heightmap

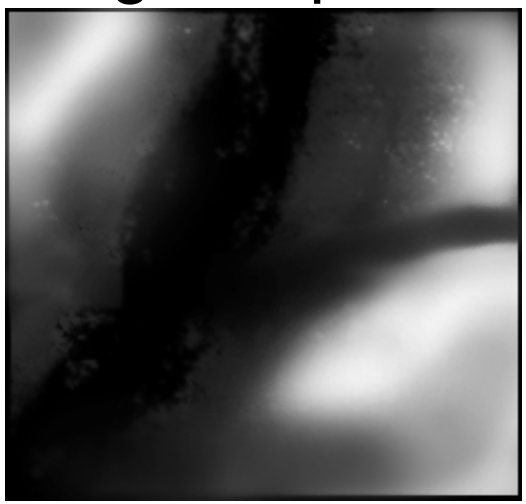


Colortexture

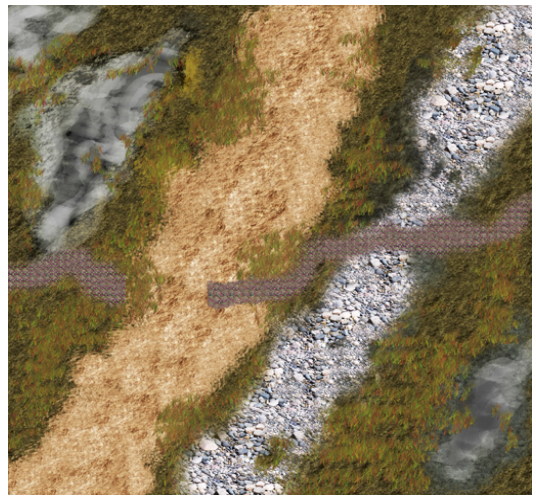


Die Terrain-Renderpipeline - Simple

Heightmap

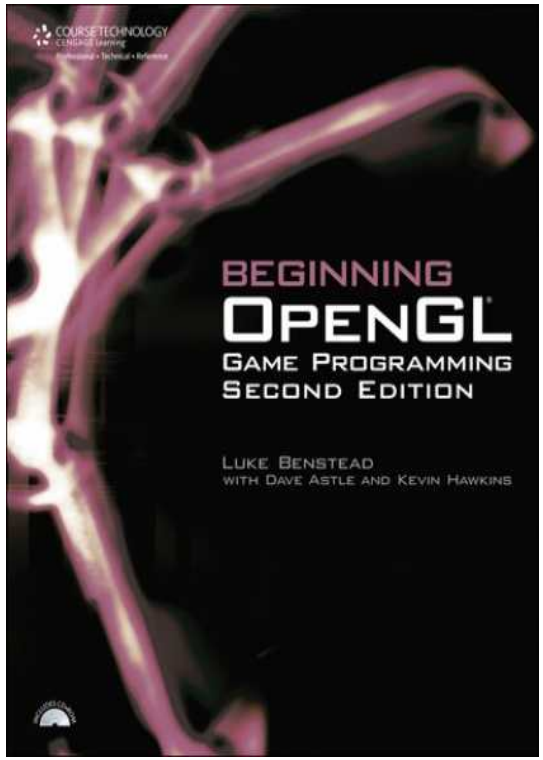


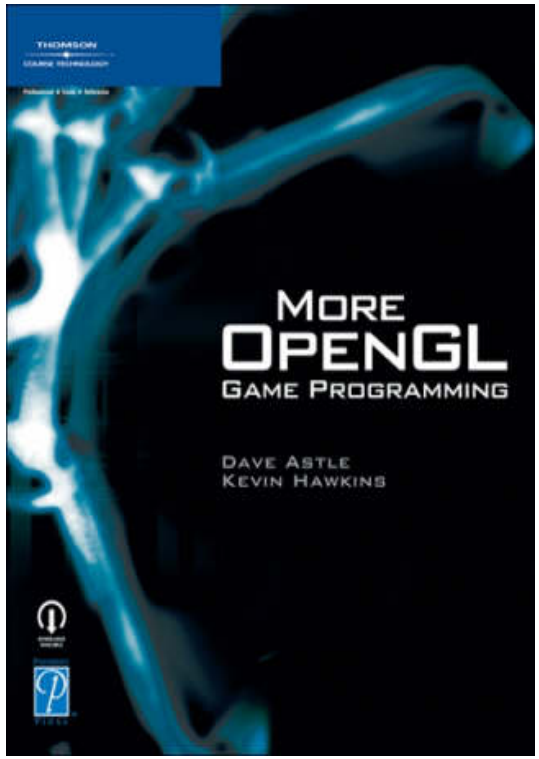
Colortexture

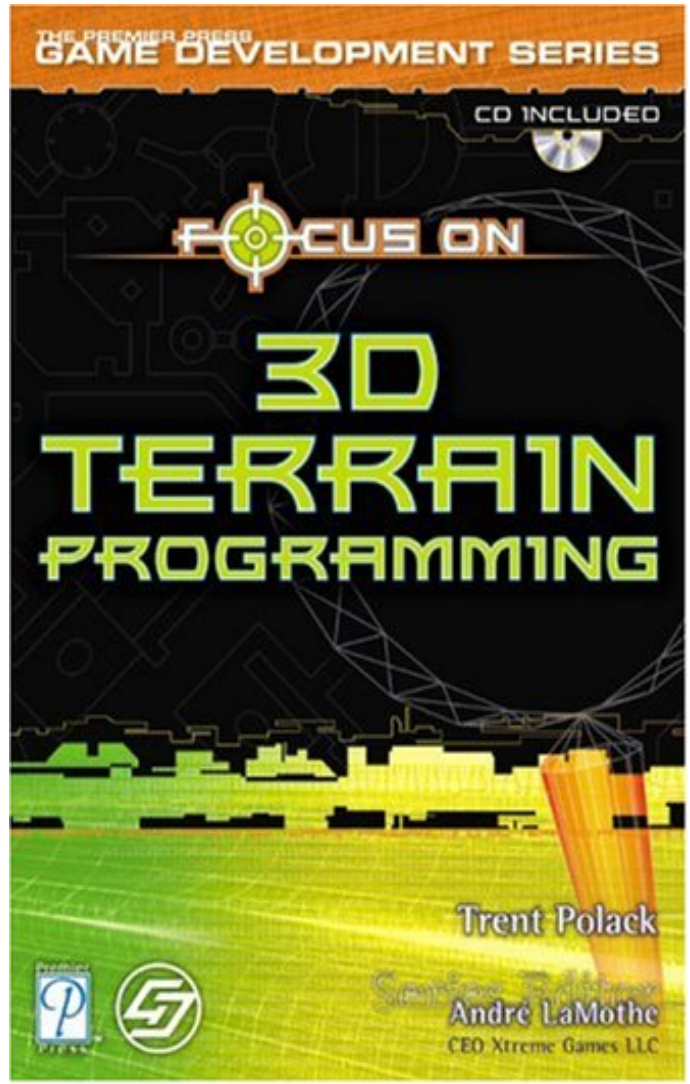


- Normalen werden on-the-fly aus der *Heightmap* berechnet

Buchempfehlung







Terrain mit Bruteforce

Rechenbeispiel:

- Heightmap: 512*512 Px

Terrain mit Bruteforce

Rechenbeispiel:

- Heightmap: 512*512 Px
- Triangles: (h-1)*w*2 = 523264

Terrain mit Bruteforce

Rechenbeispiel:

- Heightmap: 512*512 Px
- Triangles: (h-1)*w*2 = 523264
- Daten: 523264*3*sizeof(float)= 6279168 Byte = 6123 KByte

Terrain mit Bruteforce

Rechenbeispiel:

- Heightmap: 512*512 Px
- Triangles: (h-1)*w*2 = 523264
- Daten: 523264*3*sizeof(float)= 6279168 Byte = 6123 KByte
- Rohdatenübertragung bei 50FPS: ~ 300 MB/Sec
- Immenser Overhead durch 523264 Drawcalls/Frame

Terrain mit Bruteforce - Simple Lösung

- Verwendung von VertexBufferObjects (VBO's)

Terrain mit Bruteforce - Simple Lösung

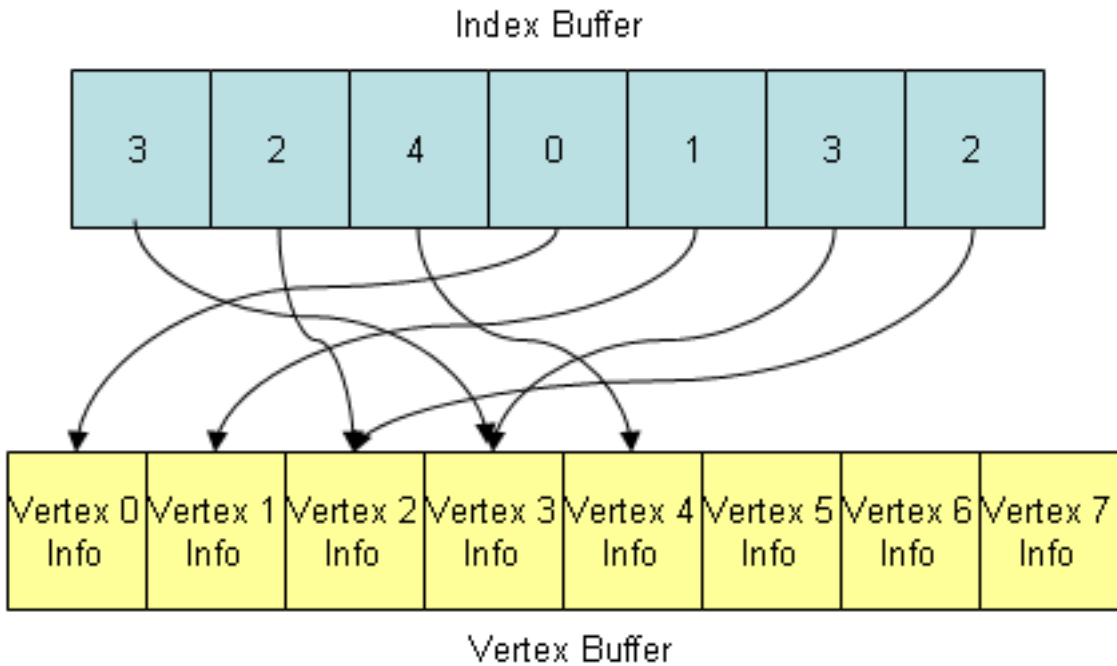
- Verwendung von VertexBufferObjects (VBO's)
- **Vorteil:** Nur noch ein Draw-Call mit `glDrawArrays` anstatt tausende mit `glVertex3f` um das gesamte Terrain zu Zeichnen.
- **Nachteil:** Generierung des VBO's mit den richtigen Daten ist selbst zu finden.

Terrain mit Bruteforce - Komplexer

- Verwendung von VertexBufferObjects (VBO's) und IndexBufferObjects (IBO's)
- **Vorteil:** Nur noch ein Draw-Call mit `glDrawElements` anstatt tausende mit `glVertex3f` um das gesamte Terrain zu Zeichnen.
- **Vorteil:** Einsparung von ca. 70% Vertexdaten und deutliche Performancesteigerung
- **Vorteil:** Algorithmus zur Erstellung wird hier vorgestellt.

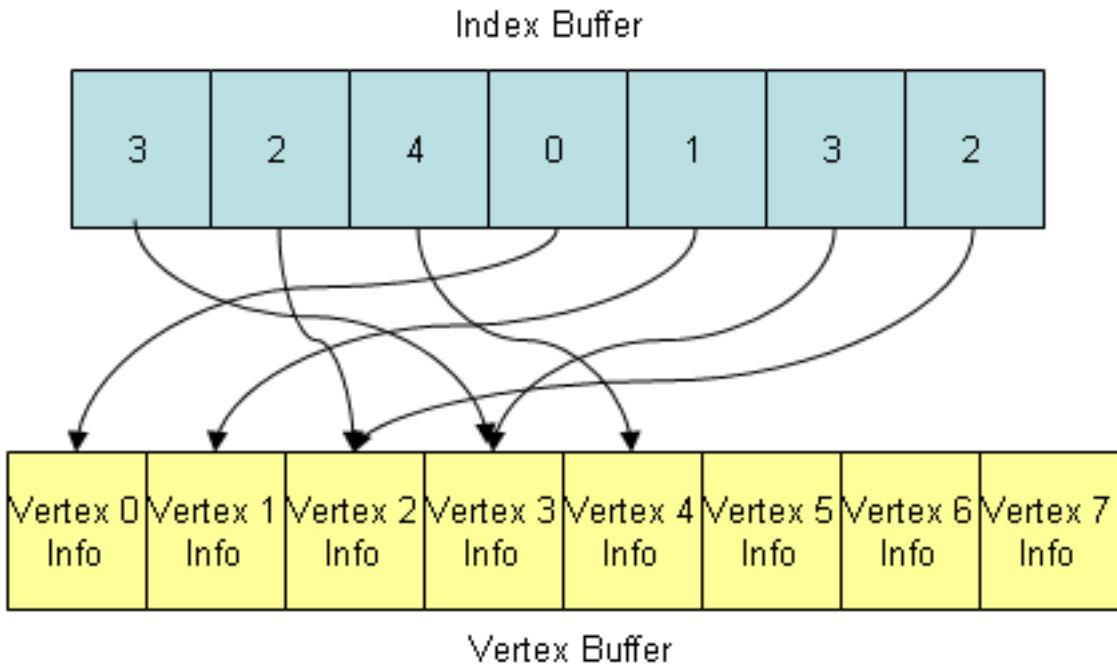
Terrain mit Bruteforce

- Vertexbuffer hält die Vertexdaten in einem kontinuierlichen Array auf der GPU
- Indexbuffer hält die Indizes in der Reihenfolge, in der sie gezeichnet werden sollen als Integerwerte



Terrain mit Bruteforce

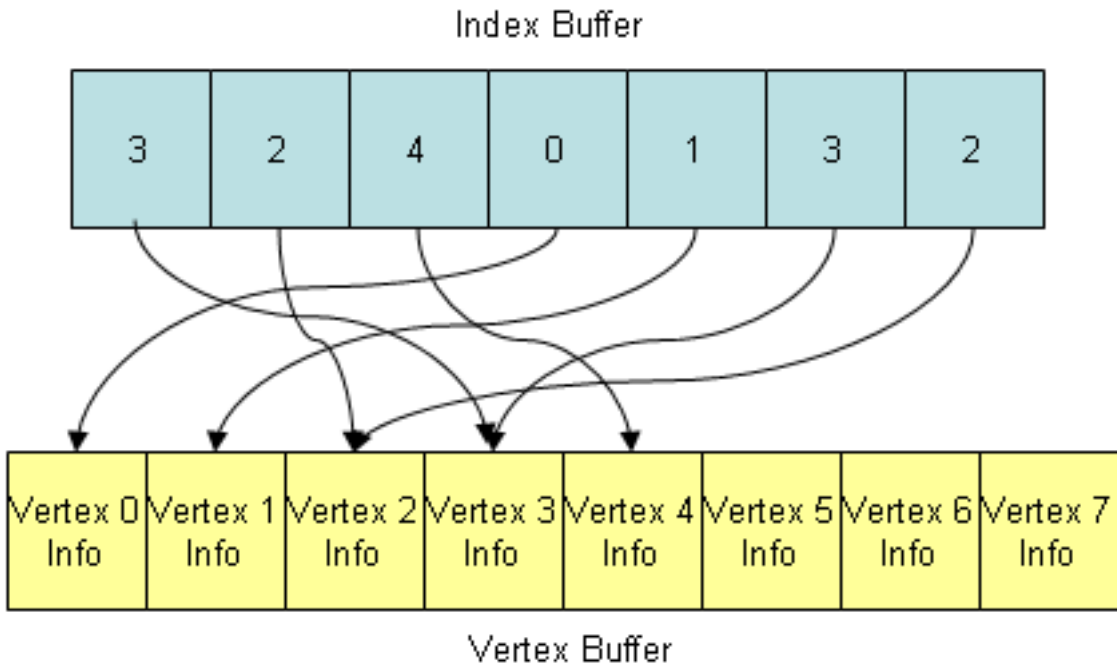
- Vertexbuffer hält die Vertexdaten in einem kontinuierlichen Array auf der GPU
- Indexbuffer hält die Indizes in der Reihenfolge, in der sie gezeichnet werden sollen als Integerwerte



Warum Indexbuffer?

Terrain mit Bruteforce

- Vertexbuffer hält die Vertexdaten in einem kontinuierlichen Array auf der GPU
- Indexbuffer hält die Indizes in der Reihenfolge, in der sie gezeichnet werden sollen als Integerwerte



Warum Indexbuffer?
Indexbuffer-Draw-Calls sind weniger aufwendig für die GPU als Vertexbuffer-Draw-Calls.
Wenn man Vertexdaten direkt zeichnet, müssen doppelte Vertices auch doppelt im Speicher liegen.

Terrain mit Bruteforce - VBO

- Generiere Zeilenweise ein Vertex für jedes Pixel der Heightmap (Pseudocode):

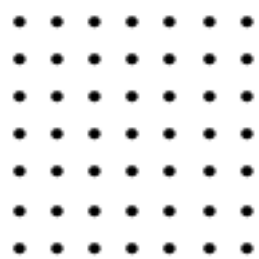
```
// heightmapdaten in dynamisches 2D-Float-Array von Datei laden
float** heightdata = loadFromHeightmap("heights.jpg");
Vertexbuffer vbo(w*h); // ... VBO mit ausreichend Vertices allokieren
for( int y = 0; y < h; ++y ) {
  for( int x = 0; x < w; ++x ) {
      // setze das Vertex in dem VBO
      vbo.setVertex(y*w+x, Vec3(x, heightdata[y*w+x], y}
  }
}
```

Terrain mit Bruteforce - VBO

- Generiere Zeilenweise ein Vertex für jedes Pixel der Heightmap (Pseudocode):

```
// heightmapdaten in dynamisches 2D-Float-Array von Datei laden
float** heightdata = loadFromHeightmap("heights.jpg");
Vertexbuffer vbo(w*h); // ... VBO mit ausreichend Vertices allokieren
for( int y = 0; y < h; ++y ) {
  for( int x = 0; x < w; ++x ) {
      // setze das Vertex in dem VBO
      vbo.setVertex(y*w+x, Vec3(x, heightdata[y*w+x], y}
  }
}
```

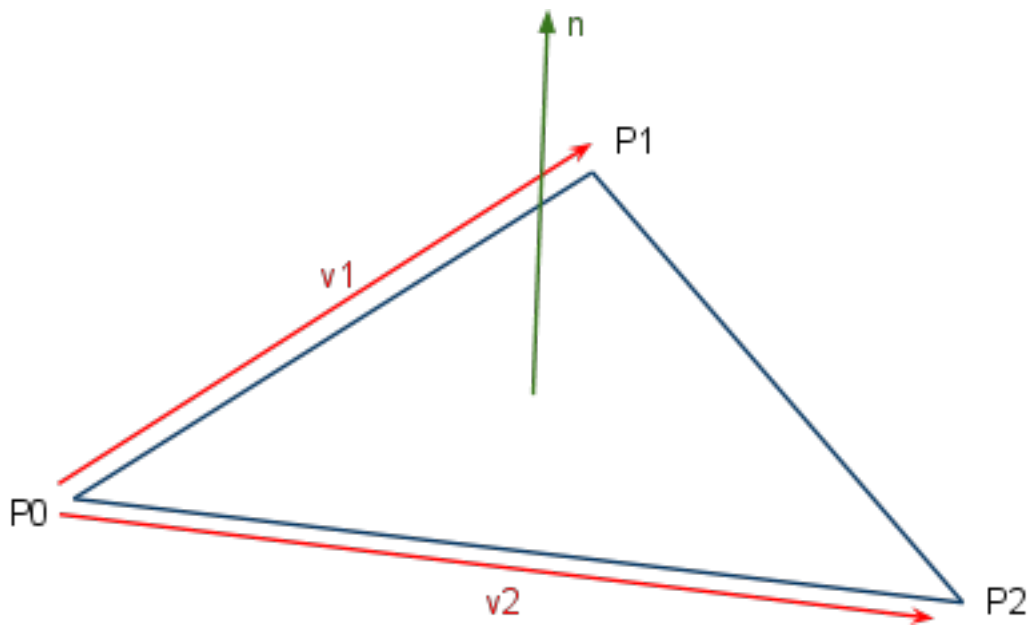
- Debugging: VBO Als GL_POINTS gerendert (Draufsicht):



Terrain mit Bruteforce - VBO

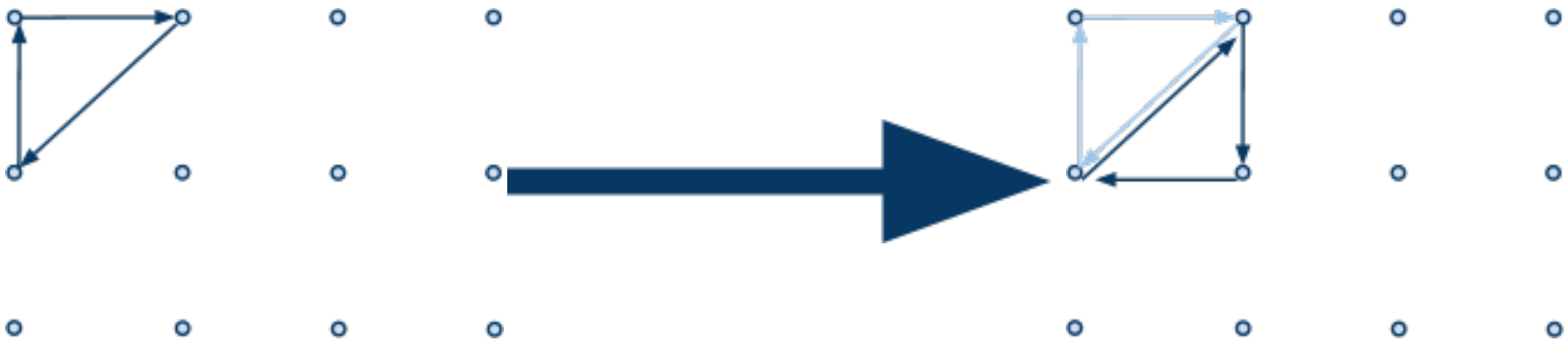
Normale eines Dreiecks:

1. Einen Eckpunkt als Bezugspunkt (P0) wählen.
2. Die beiden Vektoren v1 = P1-P0 und v2 = P2-P0 bestimmen.
3. Diese beiden Kreuzmultiplizieren v1 X v2.



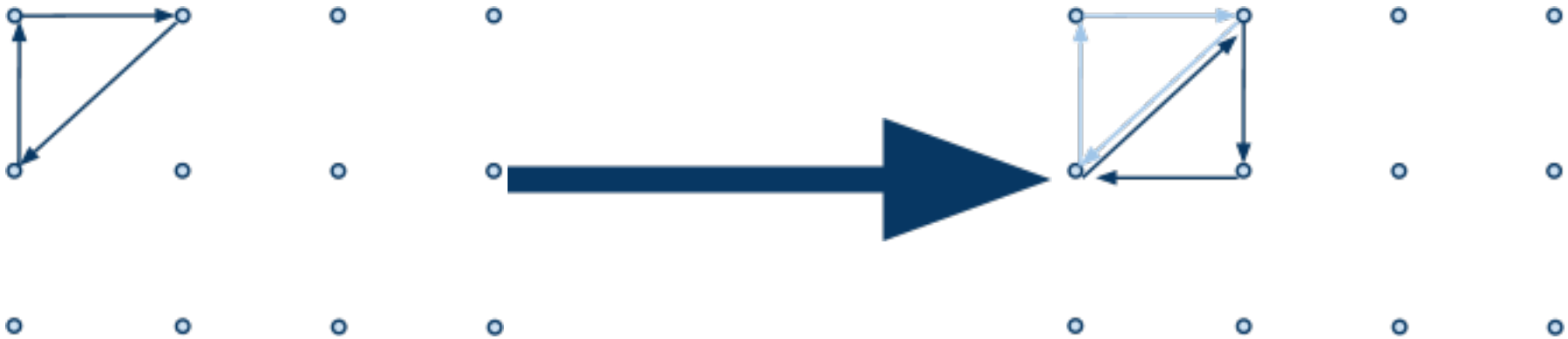
Terrain mit Bruteforce - IBO

Vertexdaten werden mehrfach verwendet:

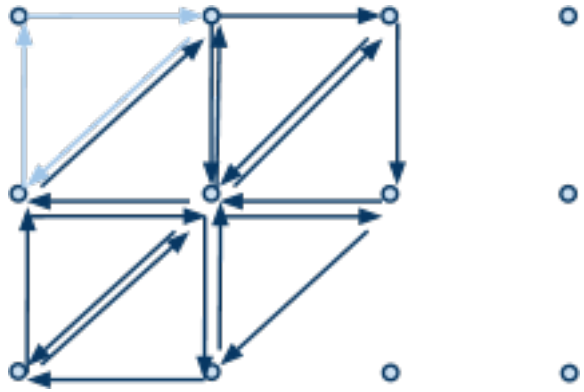


Terrain mit Bruteforce - IBO

Vertexdaten werden mehrfach verwendet:



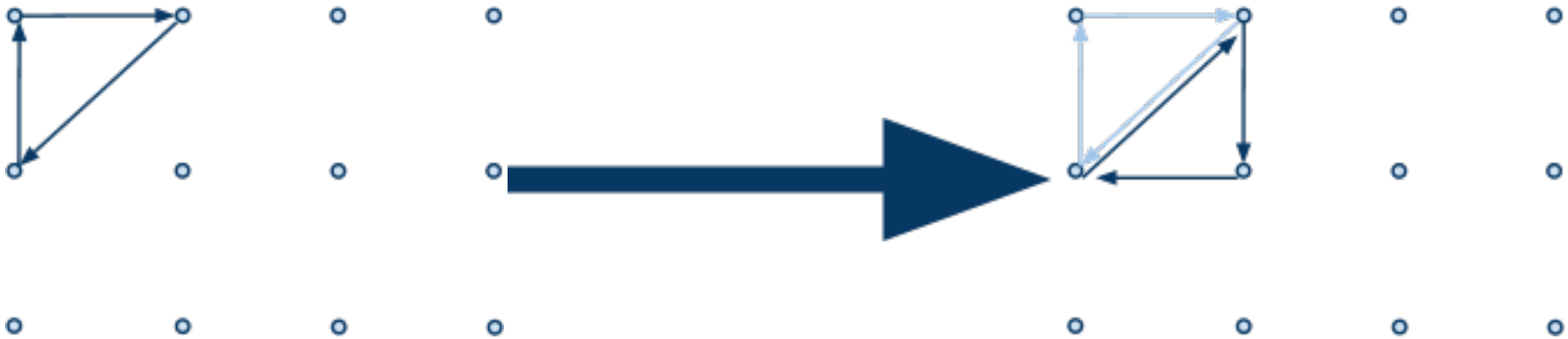
Innere Vertices werden bis zu 6 Mal verwendet:



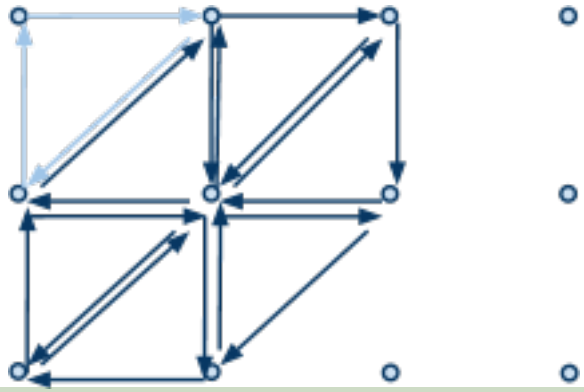
Einsparung von ca. 70% Vertexdaten!

Terrain mit Bruteforce - IBO

Vertexdaten werden mehrfach verwendet:



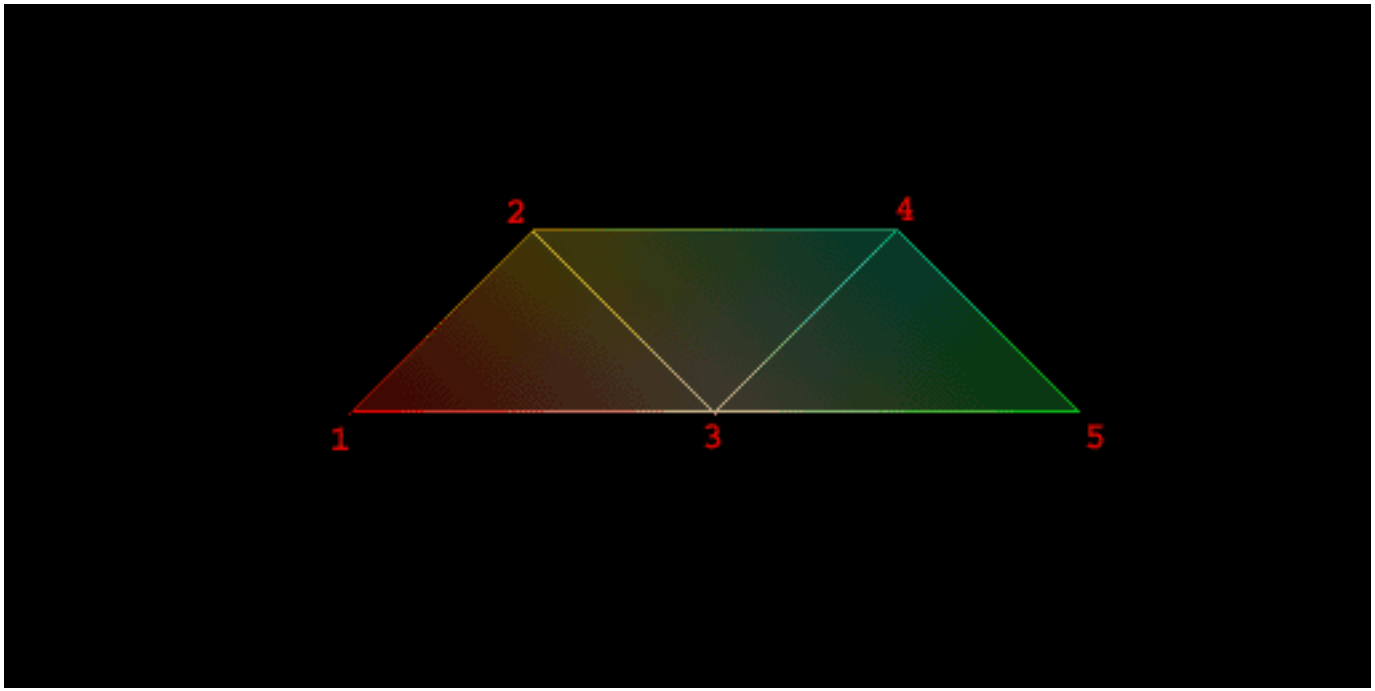
Innere Vertices werden bis zu 6 Mal verwendet:



Einsparung von ca. 70% Vertexdaten!

Terrain mit Bruteforce - IBO

Gezeichnet wird am Ende ein GL_TRIANGLE_STRIP:



Indexbufferinhalt: 1 2 3 4 5
Drawcall: GL_TRIANGLE_STRIP
Dreieck 1: 1 2 3
Dreieck 2: 2 3 4
Dreieck 3: 3 4 5

Vorteil:
Neben Vertices werden auch noch ca. 60% Indices gespart!

Terrain mit Bruteforce - IBO

Was wir brauchen:

- Algorithmus der die Indices erzeugt

Terrain mit Bruteforce - IBO

Was wir brauchen:

- Algorithmus der die Indices erzeugt

Probleme:

- Welche Indices werden am Ende einer Zeile gewählt, um den Übergang zu der nächsten Nahtlos zu bekommen?

Terrain mit Bruteforce - IBO

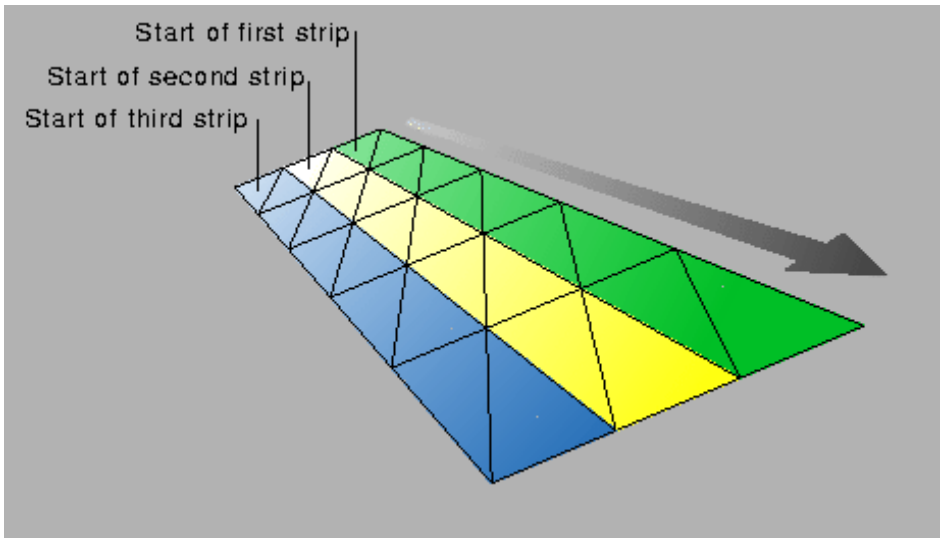
Was wir brauchen:

- Algorithmus der die Indices erzeugt

Probleme:

- Welche Indices werden am Ende einer Zeile gewählt, um den Übergang zu der nächsten Nahtlos zu bekommen?

Lösung 1:



Nachteil:
Der Vorteil des einzelnen Draw-Calls fällt weg!

Terrain mit Bruteforce - IBO

Was wir brauchen:

- Algorithmus der die Indices erzeugt

Probleme:

- Welche Indices werden am Ende einer Zeile gewählt, um den Übergang zu der nächsten Nahtlos zu bekommen?

Lösung 2:

- Stichwort *Degenerated Triangle*

Terrain mit Bruteforce - IBO

Was wir brauchen:

- Algorithmus der die Indices erzeugt

Probleme:

- Welche Indices werden am Ende einer Zeile gewählt, um den Übergang zu der nächsten Nahtlos zu bekommen?

Lösung 2:

- Stichwort *Degenerated Triangle*

Wenn ein Dreieck 2 gleiche Vertices besitzt wird es im GL_FILL Modus vom Treiber ignoriert!

Terrain mit Bruteforce - IBO

Was wir brauchen:

- Algorithmus der die Indices erzeugt

Probleme:

- Welche Indices werden am Ende einer Zeile gewählt, um den Übergang zu der nächsten Nahtlos zu bekommen?

Lösung 2:

- Stichwort *Degenerated Triangle*

Wenn ein Dreieck 2 gleiche Vertices besitzt wird es im GL_FILL Modus vom Treiber ignoriert!

- Ein Zeilensprung benötigt 2 weitere Degenerated Triangles, um den Zusammenhang unsichtbar zu machen.

Terrain mit Bruteforce - IBO

Was wir brauchen:

- Algorithmus der die Indices erzeugt

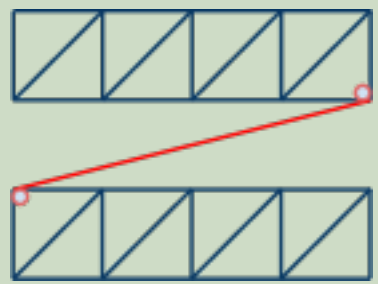
Probleme:

- Welche Indices werden am Ende einer Zeile gewählt, um den Übergang zu der nächsten Nahtlos zu bekommen?

Lösung 2:

- Stichwort *Degenerated Triangle*

Wenn ein Dreieck 2 gleiche Vertices besitzt wird es im GL_FILL Modus vom Treiber ignoriert!



Terrain mit Bruteforce - IBO

Der Algorithmus:

```
// Berechnungsvorschrift für die Anzahl der ben. Indices
int length = ((h-1) * (w*2+2))-2;
Indexbuffer ibo( length ); // ... IBO mit ausreichend Indices allokieren
```

Wenn ein Dreieck 2 gleiche Vertices besitzt wird es im GL_FILL Modus vom Treiber ignoriert!

Terrain mit Bruteforce - IBO

Der Algorithmus:

```
// Berechnungsvorschrift für die Anzahl der ben. Indices
int length = ((h-1) * (w*2+2))-2;
Indexbuffer ibo( length ); // ... IBO mit ausreichend Indices allokieren
// Für jede Zeile ausser der letzten (diese braucht kein degenerated)!
int index = 0;
for( int y = 0; y < h-1; ++y ) {
  for( int x = 0; x < w; ++x ) {
```

```
  }
}
```

Terrain mit Bruteforce - IBO

Der Algorithmus:

```
// Berechnungsvorschrift für die Anzahl der ben. Indices
int length = ((h-1) * (w*2+2))-2;
Indexbuffer ibo( length ); // ... IBO mit ausreichend Indices allokieren
// Für jede Zeile ausser der letzten (diese braucht kein degenerated)!
int index = 0;
for( int y = 0; y < h-1; ++y ) {
  for( int x = 0; x < w; ++x ) {

      // Bildung des normalen Triangle-Strips innerhalb der Zeile
      ibo.setIndex( index++, x + (y * w));
      ibo.setIndex( index++, x + ((y+1) * w));

  }
}
```

Terrain mit Bruteforce - IBO

Der Algorithmus:

```
// Berechnungsvorschrift für die Anzahl der ben. Indices
int length = ((h-1) * (w*2+2))-2;
Indexbuffer ibo( length ); // ... IBO mit ausreichend Indices allokieren
// Für jede Zeile ausser der letzten (diese braucht kein degenerated)!
int index = 0;
for( int y = 0; y < h-1; ++y ) {
  for( int x = 0; x < w; ++x ) {
      // bildung des Degenerated Triangles - Teil 1
      if( x == 0 && y != 0 ) { // erster Index einer Zeile ungleich der ersten
          ibo.setIndex( index++, x + (y * w));
      }
      // Bildung des normalen Triangle-Strips innerhalb der Zeile
      ibo.setIndex( index++, x + (y * w));
      ibo.setIndex( index++, x + ((y+1) * w));

      // bildung des Degenerated Triangles - Teil 2
      if( x == w-1 && y != h-2 ) { // letzter Index einer Zeile ungleich der letzten
          ibo.setIndex( index++, x + ((y+1) * w));
      }
  }
}
```

IBO and VBO

- Bufferobjekte sollten von Anfang an von euch in einer Klasse abstrahiert werden
- Klasse und Interface können hier selbst gewählt werden
- Wichtig sind vereinfachte Funktionen, wie `setVertexPosition` oder `setVertexTexCoord`, bzw. `setVertexNormal`

Hinweis:
Wenn man einen Vertexbuffer mit z.B. 100 Vertices erstellt, muss man `100*3*sizeof(float)` Byte allokieren. Wenn man z.B. noch Normalen mit speichern will, so sind es `2*100*3*sizeof(float)` Byte. Selbiges System gilt z.B. auch für Texturkoordinaten.

IBO and VBO

- Erstellung eines VBO's (Auszug einer bestehenden C++ Klasse):

```
// Ein neues VBO erstellen
glGenBuffers( 1, &mBuffer_ );
// Hier wird das neue VBO gebunden
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, mBuffer_ );
// Allokierung des Speichers
glBufferData( GL_ARRAY_BUFFER, mNumFloats_ * sizeof(GLfloat), NULL, GL_STATIC_DRAW );
// Reallokatoin von neuem Speicherund Löschung aller bisherigen Speicher.
// Vorteil: Man muss nicht auf die GPU warten, wenn diese gerade mit den Daten arbeitet.
// Nachteil: Kann zu Fehlern führen, wenn VBO zu groß ist.
glBufferData( GL_ARRAY_BUFFER, mNumFloats_ * sizeof(GLfloat), NULL, GL_STATIC_DRAW );
// hier holen wir uns einen Pointer auf den Speicher (readable und writeable 1D-Floatarray)
mBufferData_ = (float*)glMapBuffer( GL_ARRAY_BUFFER, GL_READ_WRITE );

// ... schreiben von Daten in das Floatarray

// Wir verlassen die Bearbeitung des VBO's und geben es dem Treiber wieder frei
glUnmapBuffer( GL_ARRAY_BUFFER );
// Hier wird das gebundene VBO auf 0 gesetzt
glBindBuffer( GL_ARRAY_BUFFER, NULL );
```

IBO and VBO

- Erstellung eines VBO's (Auszug einer bestehenden C++ Klasse):

Reallokierung ist nur notwending, wenn man die Daten später verändern möchte.
Sie sollte daher nicht im Konstruktor gemacht werden.

```
// Ein neues VBO erstellen
glGenBuffers( 1, &mBuffer_ );
// Hier wird das neue VBO gebunden
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, mBuffer_ );
// Allokierung des Speichers
glBufferData( GL_ARRAY_BUFFER, mNumFloats_ * sizeof(GLfloat), NULL, GL_STATIC_DRAW );
// Reallokatoin von neuem Speicherund Löschung aller bisherigen Speicher.
// Vorteil: Man muss nicht auf die GPU warten, wenn diese gerade mit den Daten arbeitet.
// Nachteil: Kann zu Fehlern führen, wenn VBO zu groß ist.
glBufferData( GL_ARRAY_BUFFER, mNumFloats_ * sizeof(GLfloat), NULL, GL_STATIC_DRAW );
// hier holen wir uns einen Pointer auf den Speicher (readable und writeable 1D-Floatarray)
mBufferData_ = (float*)glMapBuffer( GL_ARRAY_BUFFER, GL_READ_WRITE );

// ... schreiben von Daten in das Floatarray

// Wir verlassen die Bearbeitung des VBO's und geben es dem Treiber wieder frei
glUnmapBuffer( GL_ARRAY_BUFFER );
// Hier wird das gebundene VBO auf 0 gesetzt
glBindBuffer( GL_ARRAY_BUFFER, NULL );
```

IBO and VBO

- Erstellung eines IBO's (Auszug einer bestehenden C++ Klasse):

```
// Ein neues IBO erstellen
glGenBuffers( 1, &mBuffer_ );
// Hier wird das neue IBO gebunden
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, mBuffer_ );
// Allokierung des Speichers
glBufferData( GL_ELEMENT_ARRAY_BUFFER, mNumIndices_ * sizeof(GLfloat), NULL, GL_STATIC_DRAW );
// Reallokatoin von neuem Speicherund Löschung aller bisherigen Speicher.
// Vorteil: Man muss nicht auf die GPU warten, wenn diese gerade mit den Daten arbeitet.
// Nachteil: Kann zu Fehlern führen, wenn IBO zu groß ist.
glBufferData( GL_ELEMENT_ARRAY_BUFFER, mNumIndices_ * sizeof(GLfloat), NULL, GL_STATIC_DRAW );
// hier holen wir uns einen Pointer auf den Speicher (readable und writeable 1D-Integerarray)
mBufferData_ = (unsigned*)glMapBuffer( GL_ELEMENT_ARRAY_BUFFER, GL_READ_WRITE );

// ... schreiben von Daten in das Integerarray

// Wir verlassen die Bearbeitung des IBO's und geben es dem Treiber wieder frei
glUnmapBuffer( GL_ELEMENT_ARRAY_BUFFER );
// Hier wird das gebundene IBO auf 0 gesetzt
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, NULL );
```

IBO and VBO

- Erstellung eines IBO's (Auszug einer bestehenden C++ Klasse):

Reallokierung ist nur notwending, wenn man die Daten später verändern möchte.
Sie sollte daher nicht im Konstruktor gemacht werden.

```
// Ein neues IBO erstellen
glGenBuffers( 1, &mBuffer_ );
// Hier wird das neue IBO gebunden
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, mBuffer_ );
// Allokierung des Speichers
glBufferData( GL_ELEMENT_ARRAY_BUFFER, mNumIndices_ * sizeof(GLfloat), NULL, GL_STATIC_DRAW );
// Reallokatoin von neuem Speicherund Löschung aller bisherigen Speicher.
// Vorteil: Man muss nicht auf die GPU warten, wenn diese gerade mit den Daten arbeitet.
// Nachteil: Kann zu Fehlern führen, wenn IBO zu groß ist.
glBufferData( GL_ELEMENT_ARRAY_BUFFER, mNumIndices_ * sizeof(GLfloat), NULL, GL_STATIC_DRAW );
// hier holen wir uns einen Pointer auf den Speicher (readable und writeable 1D-Integerarray)
mBufferData_ = (unsigned*)glMapBuffer( GL_ELEMENT_ARRAY_BUFFER, GL_READ_WRITE );

// ... schreiben von Daten in das Integerarray

// Wir verlassen die Bearbeitung des IBO's und geben es dem Treiber wieder frei
glUnmapBuffer( GL_ELEMENT_ARRAY_BUFFER );
// Hier wird das gebundene IBO auf 0 gesetzt
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, NULL );
```

Ressources - Linkliste

Terrain Algorithmen

The Virtual Terrain Project
10 Parts Terrain Tutorial
Lighthouse3D Terrain Tutorial
Advanced Terrain Tesselation
Evaluation of LOD-Techniques for GPU-based heightfield rendering
Hardware Based Terrain Clipmapping
Geometry Clipmaps: [1], [2], [3], [4], [5]
Terrains using OpenGL's 4.x Features
Real-Time Dynamic Level of Detail Terrain Rendering with ROAM
Continuous LOD Terrain Meshing Using Adaptive Quadtrees
Terrain Tutorial Teil [1], [2], [3]
Weiteres Tutorial

Triangle Strips

A Triangle Strip Manual

VertexBuffer und IndexBuffer

GEAR Interface von VertexBuffer
GEAR Interface von IndexBuffer
VBO creation, drawing and updating
Official VBO Spec
Umfassender VBO Wiki Eintrag