Genesis-Projekt: Eigene Landschaftsdaten generieren

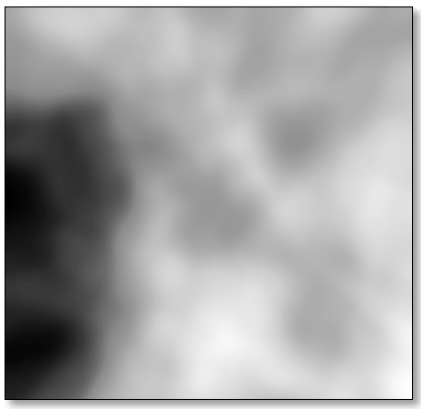
# Blühende Landschaften

Generieren Sie eigene **realistische Landschaftsdaten** mit dafür passenden Algorithmen. Die aufbereiteten 3D-Daten geben Sie über OpenGL aus.

CARSTEN DACHSBACHER

Die einfachste Methode, Landschaftsdaten zu generieren, geht von einer Art Landschaftskarte aus, in der die Höheninformation gespeichert ist (*Heightmap* oder einem *Heightfield*). Dahinter verbirgt sich eine Bitmap mit Graustufen.

Das Bild zeigt eine Heightmap. Hellere Pixel bedeuten, dass die Landschaft an dieser Stelle höher ist, dunklere Pixel stehen für tiefer gelegene Landschaftsteile.



**EINE GENERIERTE HEIGHTMAP** legt die Landschaft fest.

## DAS GENESIS-PROJEKT

Unser Genesis-Projekt gliedert sich in folgende drei Teile, die Sie von den OpenGL-Grundlagen bis zum Einsatz praxistauglicher Algorithmen führen.

**Teil 1:** Landschaften rendern mit OpenGL
**Teil 2:** Eigene Landschaftsdaten generieren
**Teil 3:** Methoden des Landschafts-Texturierens und Spezialeffekte

Diese Bitmap enthält die später notwendigen Informationen, um die 3D-Daten zu generieren. Zunächst erzeugen Sie die Heightmaps. Dabei helfen Ihnen Algorithmen, mit denen Sie charakteristische Heightmaps für hügelige, flache oder vulkanische Landschaften erzeugen. Alle Sourcecodes, die hier teilweise in Ausschnitten gezeigt werden, finden Sie komplett auf der Heft-CD.

## Fault Formation Algorithmus

Der Fault Formation Algorithmus eignet sich, um Landschaftsdaten für Küstenregionen, steile Hänge oder Felsplateaus anzulegen. In der Natur entstehen solche Landstriche unter anderem dadurch, dass sich Platten tektonisch verschieben, Küsten erodieren und Landschaften sich durch Wasser- oder Wind-Phänomene verändern.

Der Algorithmus ist einfach: Sie starten mit einer leeren Heightmap der Größe *SIZE x SIZE*. Jeder Punkt erhält die Höhe Null. Nun wählen Sie eine zufällige Linie in der Heightmap:

```
x1 = rand()%SIZE;
  y1 = rand()%SIZE;
  do
  {
    x2 = rand()%SIZE;
    y2 = rand()%SIZE;
  } while ( x2==x1 && y2==y1 );

  float dx = (float)(x2 - x1);
  float dy = (float)(y2 - y1);
```

Diese Linie teilt die Map in zwei Teile. Entscheiden Sie sich für eine Seite, und erhöhen Sie alle Punkte um eine bestimmte Höhendifferenz.

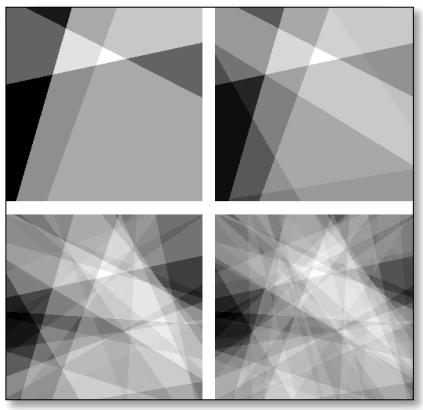
```
// Entscheidung, welche Seite
int upDown =
```

**AUF CD 1**

Die Quelltexte sowie die fertig übersetzten Routinen finden Sie im Verzeichnis *Praxis/Programmierung/PC Underground*.

```
  (dx>0&&dy<0)||(dx>0&&dy>0);

// die Steigung der Linie
if ( dx )
  dy /= dx; else dy = 0.0f;

x = 0;
y = y1 - x1 * dy;

// Punkte gegen die Linie testen
for ( x2 = 0; x2 < SIZE;
 x2 ++, y += dy )
{
  for ( y2 = 0;
 y2 < SIZE; y2 ++ )
if ( (  upDown && y2 < y )
/*up*/ || ( !upDown && y2 > y )
/*down*/ )
terrain[ x2 + y2 * SIZE ] +=
 heightDifference;
}
```

Wiederholen (iterieren) Sie diesen Vorgang mehrmals, wobei Sie jedes Mal die Höhendifferenz *heightDifference* verkleinern, bis Sie genügend Details auf der Heightmap haben.



**DIESE MIT FAULT-FORMATION** angelegten Heightmaps weisen 4, 8, 32 und 64 Iterationen auf.

Im Bild sehen Sie die Heightmap nach vier, acht, 32 und 64 Iterationen. Es ergeben sich unnatürliche Höhenunterschiede benachbarter Bereiche in der Bitmap. In den hohen Frequenzen sind Helligkeitssprünge bei einer Frequenzanalyse der Bitmaps zu finden. Wenden Sie den *FIR*-Filter (Finite Impulse Response) an, um den natürlichen Erosionseffekt auf der Landschaft nachzubilden. Er wandelt eine Folge von Eingabewerten *x[1..n]* in eine Sequenz *y[1..n]* um. Verwenden Sie folgende Formel:

```
y[ i ] =
k*y[ i-1 ] + (1-k)*x[ i ]
```

Die Konstante *k* bestimmt die Stärke des Filters. Der Wert von *k* befindet sich im Bereich von *0* bis *1*. Kleine Werte stehen für eine schwache Erosion, größere Werte für stärkere. Ein Wert um *0.5* ist

gut geeignet für die Landschaftsgenerierung. Den *FIR*-Filter wenden Sie auf jede Zeile und auf jede Spalte in beiden Richtungen an.

Wenn Sie den Filter auf die Heightmap anwenden, lassen sich schöne Landschaftsdaten erzeugen.

**1 FIRFilter**

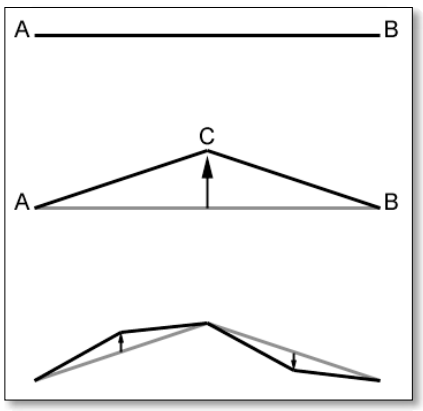
```
// "ofs" legt die Richtung in
// der Heightmap fest
void filterFIR ( float *value,
int ofs, float filter )
{
  float v  =  value[ 0 ];
  float *b = &value[ ofs ];
  float ifilter = 1.0f - filter;

  for ( i = 0; i <\<> SIZE-1;
        i++,b+=ofs)
  {
  *b =filter * v + ifilter * *b;
    v = *b;
  }
}
// Landschaftfilter in 4 Richt.
void filterTerrain
 ( float *t, float f )
{
// Zeilen von lks nach rechts
  // und umgekehrt filtern
  for ( int i = 0;
      i <\<> SIZE; i++ )
  {
   filterFIR( &t[SIZE*i], 1, f);
   filterFIR
   ( &t[SIZE*i+SIZE-1],-1,f);
  }

  // Spalten von oben nach unten
  // und umgekehrt filtern
  for ( i = 0; i <\<> SIZE; i++ )
  {
    filterFIR( &t[i], SIZE, f);
    filterFIR
( &t[SIZE*(SIZE-1)+i],-SIZE, f);
  }
}
```

**Diesen *FIRFilter* wenden Sie auf jede Zeile und jede Spalte in beiden Richtungen an.**

## Midpoint-Displacement-Algorithmus

Mit dem Midpoint-Displacement-Algorithmus (Fraktal-Plasma oder Diamond-Square-Algorithmus) können Sie bergige und hügelige Landschaften generieren. Sie erzeugen die Heightmap wieder iterativ, wobei Sie die Start-Höhen-Differenz *dh* wählen. Veranschaulichen Sie sich den Algorithmus zuerst im Eindimensionalen.

Sie beginnen im ersten Schritt mit einer Linie AB im oberen Teil des Bildes. Unterteilen Sie die Linie in der Mitte am Punkt *C*, und addieren Sie zur Höhe des neuen Punkts einen Zufallswert zwischen *-dh* und *+dh*, wie dies der mittlere Teil des Bildes zeigt. In den weiteren Schritten verringern Sie *dh*, indem Sie *dh* mit *pow( 2.0, -roughness )* multiplizieren und die neuen Liniensegmente unterteilen.

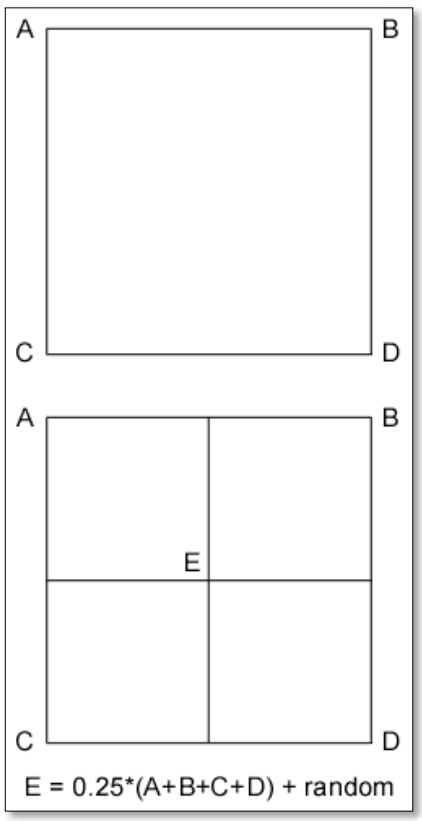



**DIE GRAFIK** verdeutlicht den Midpoint-Displacement-Algorithmus im Eindimensionalen.

Mit der *roughness*-Konstante steuern Sie, wie rauh und detailreich die Landschaft wird. Beim Wert *1* wird der Spielraum für die zufällige Höhenänderung in jedem Iterationsschritt halbiert. Für Werte über 1 haben die ersten Iterationen größeren Einfluss auf die Form der Landschaft, und Sie generieren große Hügel und Berge. Bei Werten unter 1 haben die späteren Iterationsschritte größeres Gewicht, und Sie erhalten eher chaotische Daten. Wie die Werte das Ergebnis beeinflussen, sehen Sie im folgenden Bild für die Werte *4*, *1* und *0.25*.

Um solche Bilder zu berechnen, müssen Sie den Algorithmus ins Zweidimensionale übertragen. Hier beginnen Sie nicht mit einer Linie, sondern mit einem Quadrat. Bei einem Quadrat müssen Sie fünf Mittelpunkte berechnen: vier an den Kanten zwischen den Eckpunkten und einen in der Mitte.

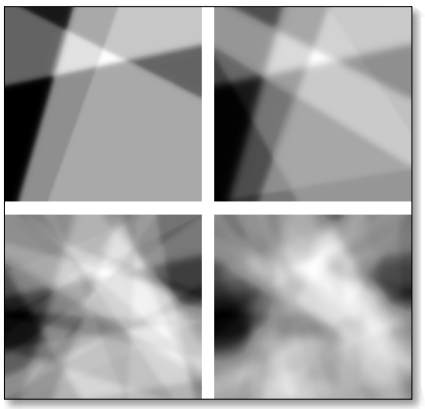
Mit dem *Diamond Step* berechnen Sie den Punkt in der Mitte des Quadrats. Sie beginnen mit dem Quadrat *ABCD*. Die Höhe des Punkts *E* in der Mitte erhalten Sie durch Mittelung der Höhen bei *A*, *B*, *C*, *D* und dem Addieren eines Zufallswerts. Die Mittelpunkte auf den Kanten berechnen Sie durch die Eckpunkte des Quadrats und der Quadratmittelpunkte der beiden anliegenden Quadrate. Diesen *Square Step* zeigt das folgende Bild.




**MIT DEM DIAMOND STEP** berechnen Sie Midpoint Displacement in 2D.

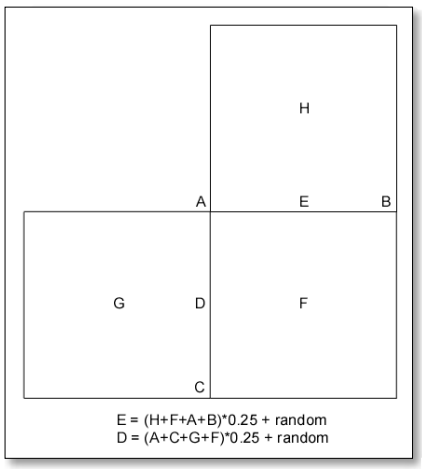
## Particle-Deposition-Algorithmus

Der Particle-Deposition-Algorithmus eignet sich dazu, vulkanische Landschaften nachzubilden. Sie programmieren eine Art Partikelfallsystem. Sie beginnen mit einem leeren Heightfield. An einer Stelle lassen Sie eine Reihe von Partikeln fal- ▶



**DIE HEIGHTMAPS** aus dem vorigen Bild hat ein *FIR*-Filter modifiziert.

**VERSCHIEDENE ROUGHNESS-WERTE** (*4, 1, 0.25*) beeinflussen Form und Detail der Landschaft.

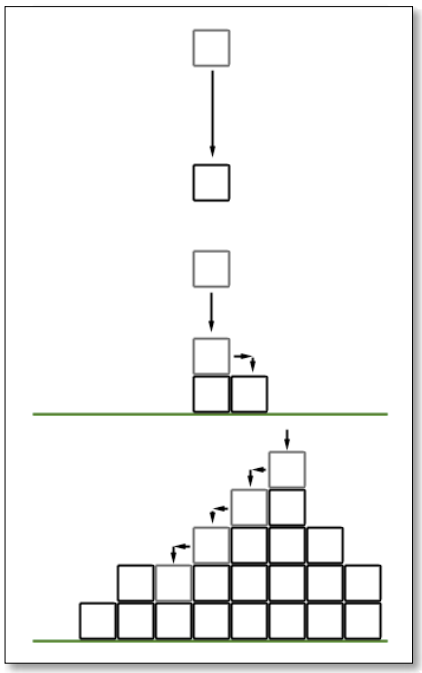



**DER SQUARE STEP** ist die zweite Operation des 2D-Midpoint-Displacement.

len. Der erste bleibt am Boden liegen, wie der obere Teil des unteren Bildes zeigt. Der zweite trifft auf den ersten auf und fließt weiter, bis er zur Ruhe kommt, wenn es keinen Nachbarpunkt im Heightfield gibt, der eine geringere Höhe aufweist (Bild Mitte).

Lassen Sie einige Partikel fallen, wobei Sie von Zeit zu Zeit den Ort variieren, bis Sie einen Vulkankegel erhalten. Die Form der Landschaft beeinflussen Sie durch die Zahl der Partikel und der Änderung des Ursprungsort der Partikel.

Wenn die Lava abgekühlt ist, bricht die Spitze des Kegels ein. Diesen Effekt können Sie nachbilden. Dazu suchen Sie den höchsten Punkt eines Kegels. Alle

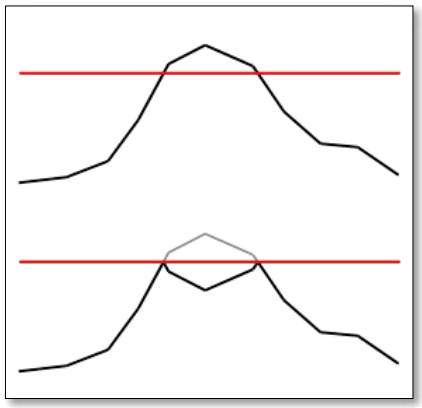


**DER PARTICLE-DEPOSITION-ALGORITHMUS** lässt Lava fließen.

Punkte in der Umgebung, deren Höhe nicht zu stark von der des Gipfels abweicht (dazu definieren Sie eine Konstante), spiegeln Sie an der Kraterlinie.

## Polygondaten generieren

Wenn Sie an dieser Stelle angelangt sind, liegt eine Heightmap vor, aus der Sie Polygondaten erzeugen müssen, um die Landschaft mit OpenGL darzustellen. Zum Rendern der Landschaft eignen sich von den OpenGL-Render-Primitiven besonders die Triangle Strips, die Sie



**DIE GRAFIK** symbolisiert, wie Sie Vulkankrater anlegen.

in der letzten Ausgabe kennengelernt haben. 3D-Beschleuniger können diese besonders schnell zeichnen. Zuerst definieren Sie eine Struktur, in der Sie die Daten für jeden Vertex speichern:

```
typedef struct
{
  // Texturkoordinaten
  float texCoord[ 2 ];
  // Farbwerte
  float color[ 3 ];
  // Vertexposition
  float vertex[ 3 ];
}INTERLEAVEDVERTEX;
INTERLEAVEDVERTEX *pVertex;
```

Nun generieren Sie die Vertizes für die Triangle Strips aus der Heightmap der Größe *SIZE x SIZE*, die im Array *terrain* gespeichert ist:

```
// SIZE: Größe der Heightmap
unsigned char terrain
   [ SIZE * SIZE ];
pVertex = new INTERLEAVEDVERTEX
   [ SIZE * SIZE * 2 * 3 ];
INTERLEAVEDVERTEX *p = pVertex;
// Macro
#define addVertex( x, y ) \
{ \
height=
 terrain[(x)+(y)*SIZE]; \
p->vertex[ 0 ] = x - SIZE/2;\
p->vertex[ 1 ] = y - SIZE/2;\
p->vertex[ 2 ] = height;\
p->color[..] = ...;\
p->texCoord[ 0 ] = (x)/SIZE;\
p->texCoord[ 1 ] = (y)/SIZE;\
p ++;          \
nVertices ++;        \
}
// Vertexdaten
for ( int j = 0;
   j < SIZE-1; j+=1 )
  for ( int i = 0;
   i < SIZE-1; i+=1 )
  {
   addVertex( i, j+RESOLUTION );
   addVertex( i, j );
  }
```

Zeichnen Sie die Daten mit den bekannten OpenGL-Befehlen:

```
glBegin( GL_TRIANGLE_STRIP );
for ( i = 0; i < SIZE; i++ )
{
  INTERLEAVEDVERTEX *p =
    &pVertex[ i * ( SIZE*2-2 ) ];
  for ( j = 0;
  j < SIZE*2; j++ )
  {
glTexCoord2fv(&p->texCoord[0] );
glColor3fv( &p->color[0] );
glVertex3fv( &p->vertex[0] );
p ++;
  }
}
glEnd();
```

Die Befehle verbrauchen jedoch Performance. Bei einer Größe der Heightmap von *256 x 256* Pixeln haben Sie es mit einer Anzahl von *131 072* Dreiecke zu tun. Allein der Overhead, der durch das Aufrufen der Funktionen (mal 3 = *393 216* Aufrufe) entsteht, ist beachtlich. Optimieren Sie deshalb die Übertragung der Vertexdaten zu OpenGL.

## Effizientes Rendering

Verwenden Sie Interleaved Arrays. Diese enthalten Vertex-, Farb- und Texturdaten des 3D-Objekts oder der 3D-Landschaft. In diesem Array können Sie auch andere Daten speichern, die mit dem Rendering nichts zu tun haben. Dazu teilen Sie OpenGL mit, wie viele Bytes diese in Anspruch nehmen. Mit folgenden Zeilen zeichnen Sie die Landschaft:

```
glInterleavedArrays
( GL_T2F_C3F_V3F, 0, pVertex );
  for ( int i = 0;
  i < SIZE-1; i++ )
glDrawArrays( GL_TRIANGLE_STRIP,
     i * (SIZE*2-2), SIZE*2-2 );
```

Der Parameter *GL_T2F_C3F_V3F* ist eine OpenGL-Konstante, die den Aufbau der Struktur im Array beschreibt. Dieser Aufbau entspricht unserem *INTERLEAVEDVERTEX*. Der zweite Parameter *glInterleavedArrays* ist der Stride-Wert (die Größe der zusätzlichen gespeicherten Daten in Bytes). Wenn Sie Sie die Vertex-, Textur- und Farbdaten jeweils in einem Array ohne weitere Daten speichern, spricht man von Streams. Diese können noch schneller von OpenGL bearbeitet werden. Bereiten Sie die Streaming-Daten für die Landschaft während der Initialisierung vor:

```
float *pTexCoordStream = new...;
float *pColorStream = new...;
float *pVertexStream = new...;
 p = pVertex;
for ( i = 0;
  i < nVertices; i++, p++ )
{
memcpy(&pTexCoordStream[i * 2],
   &p->texCoord[ 0 ],
   2 * sizeof( float ) );
 memcpy( &pColorStream[ i * 3 ],
    &p->color[ 0 ],
   3 * sizeof( float ) );
  memcpy( &pVertexStream[i * 3],
    &p->
vertex[ 0 ], 3 * sizeof(float));
}
```

Teilen Sie OpenGL mit, dass Sie Streaming-Daten verwenden wollen:

```
glEnableClientState
 ( GL_COLOR_ARRAY );
glEnableClientState
 ( GL_VERTEX_ARRAY );
glEnableClientState
 ( GL_TEXTURE_COORD_ARRAY );
//Pointer auf die Arrays setzen
glVertexPointer
(3, GL_FLOAT, 0, pVertexStream);
glTexCoordPointer
(2, GL_FLOAT,0,pTexCoordStream);
glColorPointer
( 3, GL_FLOAT, 0, pColorStream);
...
// und Zeichnen
for ( int i = 0;
  i < SIZE-1; i++ )
glDrawArrays(GL_TRIANGLE_STRIP,
     i * (SIZE*2-2), SIZE*2-2 );
glFlush();
// Streaming wieder abschalten
glDisableClientState
 ( GL_COLOR_ARRAY );
glDisableClientState
 ( GL_VERTEX_ARRAY );
glDisableClientState
 ( GL_TEXTURE_COORD_ARRAY );
```

Auch dass die Daten der Landschaft statisch sind, können Sie OpenGL mitteilen. Damit optimieren Sie die Daten für den Treiber oder kopieren die Daten in den eigenen Speicher der Grafikkarte, falls diese Optionen unterstützt werden. Bei den dazu benötigten Funktionen handelt es sich um OpenGL-Extensions (Erweiterungen), die nicht immer in OpenGL vorhanden waren. Deren Existenz in einem OpenGL-Treiber müssen Sie abfragen. Dazu versuchen Sie, die Adresse der benötigten Funktionen im Speicher zu bekommen:

```
typedef void
(APIENTRY *LOCKARRAYS_PROC)
    (int first, int count);
typedef void
(APIENTRY *UNLOCKARRAYS_PROC)
 (void);
LOCKARRAYS_PROC   pfLockArrays;
UNLOCKARRAYS_PROC
 pfUnlockArrays;
pfLockArrays = (LOCKARRAYS_PROC)
  wglGetProcAddress
 ( „glLockArraysEXT“ );
pfUnlockArrays =
 (UNLOCKARRAYS_PROC)
  wglGetProcAddress
 ( „glUnlockArraysEXT“ );
 if ( !pfLockArrays ||
   !pfUnlockArrays )
 { /* nicht unterstützt! */ }
```

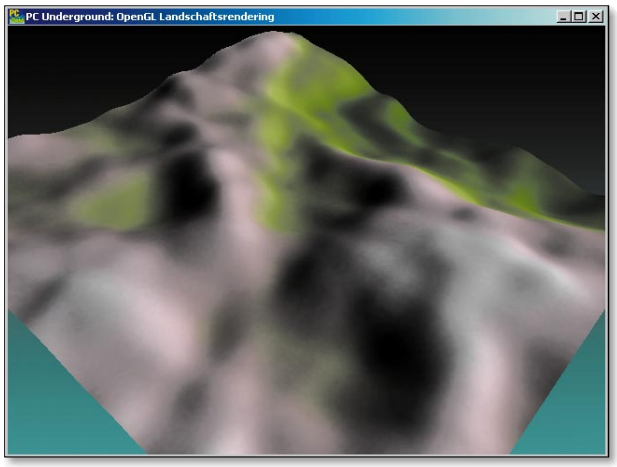
Werden die beiden Funktionen unterstützt, teilen Sie OpenGL mit, dass Sie Streaming-Daten verwenden wollen, und sperren („locken“) die Streams mit:

```
(*pfLockArrays)
 ( 0, nVertices
);
```

Zeichnen Sie mit *glDrawArrays*. Am Programmende heben Sie die Sperre auf und schalten das Streaming ab:

```
(*pfUnlockAr-
rays)();
```

Die Farbwerte bestimmen Sie durch die Höhe der Landschaft. Die Beleuchtungseffekte auf der Landschaft entstehen durch eine Textur, die Sie entweder mit Paintshop Pro oder Photoshop anlegen. Wenden Sie einen Emboss-Filter auf die Heightmap (Bild rechts) an.




**EINE GENERIERTE LANDSCHAFT** in 3D-Darstellung wartet auf Besucher.

Übergeben Sie die Texturkoordinaten der Landschaft mit den anderen Daten an OpenGL. Lassen Sie sich zunächst eine ID geben, die Ihre Textur in Zukunft eindeutig identifiziert:

```
int ID;
glGenTextures(1, &ID);
```

Nun wählen Sie die Textur aus:

```
glBindTexture
( GL_TEXTURE_2D, ID );
```

Bei OpenGL gibt es eine Reihe von Parametern, die das Texture Mapping beeinflussen – etwa ob sich die Textur wiederholen soll, wie sie vergrößert oder verkleinert dargestellt werden soll.Arbeiten Sie mit folgenden Einstellungen:

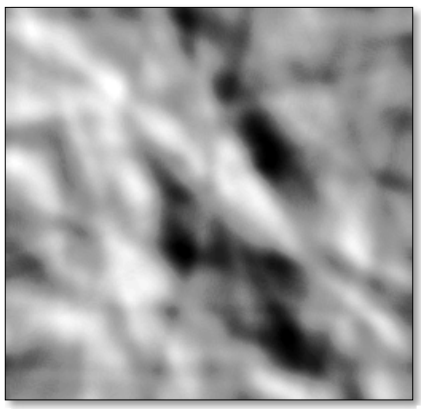
```
glTexParameterf( GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER,
   GL_LINEAR );
glTexParameterf( GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST );
```

Welche Tricks Ihnen die Funktion *glTexParameterf* eröffnet, entnehmen Sie der Online- oder jeder OpenGL-Dokumentation. Im nächsten Schritt laden Sie eine Bitmap im *bmp*-Format und übergeben sie an OpenGL:

```
AUX_RGBImageRec *texture;
  texture = auxDIBImageLoad
 ( „texture.bmp“ );
  gluBuild2DMipmaps
( GL_TEXTURE_2D, 3,
   texture->sizeX,
   texture->sizeY,
   GL_RGB, GL_UNSIGNED_BYTE,
   texture->data );
```

Der letzte Befehl übergibt die Daten und verkleinert die Textur-Versionen. Schalten Sie das Texturieren an- und aus:

```
glEnable( GL_TEXTURE_2D );
  glDisable( GL_TEXTURE_2D );
```



**DIE TEXTUR** zeigt Licht- und Schatteneffekte zur Landschaft aus dem ersten Bild.

Alle Texturfunktionen, die Sie für unser Beispielprogramm benötigen, finden Sie in der Wrapper-Klasse *PCUTexture*. Dort verwenden Sie das Texture Mapping, um die Lichtverhältnisse auf der Landschaft darzustellen. ET/TR

**Literatur**

Jackie Neider u.a.: OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1

A. L. Barabási und H. E. Stanley: Fractal Concepts in Surface Growth, Cambridge University Press, 1995

Robert Krten: Generating Realistic Terrain, Dr. Dobbs Journal (July 1994)