Dreidimensionale Landschaftserstellung mit Delphi und DirectX9

1. Einleitung

2.1 Perlin Noise

2.1.1. Der Algorithmus

2.1.2. Anwendung auf Heightmaps

2.1.2.1. Die Heightmap

2.1.2.2. Perlin Noise und Heightmaps

2.2. Die Lightmap

2.3. Die Texturemap

2.4. Die Datamap

2.5. Blur - Weichzeichner

2.6. Umwandlung der Pixelfarbe und –position in Vertexdaten

2.6.1 ohne einen QuadTree

2.6.2 Das Laden der Datamap unter Verwendung des QuadTrees

2.7. Multitexturig – Verwendung von Gras, Sand und Fels

2.8. Das Gewässer

2.8.1 Änderung an der Heightmap

2.8.2.1. Die Reflectionmaps

2.8.2.2 Das Rendern der Reflectionmap

2.9. gesamter Quellcode „cTerrain“

2.10. Resultate

2.11 Abschließende Informationen

2.12. Quellen

3. Nachwort

1. Einleitung

Ich habe ein ganzes Jahr damit verbracht, ein eigenes Terrain zu erstellen. Dies war eine
Menge Aufwand und kostete sehr viele Nerven.
Da ich sehr gerne programmiere und jetzt den ganzen Vorgang einer Landschaftserstellung verinnerlicht habe, habe ich mir die Landschaftserstellung zum Thema gemacht.
Ich habe mich auf die grundlegendsten Elemente bezogen, die man benötigt, um eine Landschaft mit DirectX zu rendern.
Basis für das Rendering ist zudem noch meine Sammlung von Units mit nützlichen Funktionen und Prozeduren, die ich im Laufe der Zeit programmiert und gesammelt habe.

Es ist teils sehr schwer, Prozesse zu beschreiben, die man frei Laune programmiert. Deshalb bitte ich um Entschuldigung, wenn man manch einen Satz nicht auf Anhieb versteht. Auch der Quelltext ist nicht im perfektesten Stil programmiert, auch wenn ich mich sehr bemüht habe, aber unter Zeitproblemen schreibt man ab und zu ein paar Zeilen eher auf Funktionalität, als auf guten Programmierstil.

2.1 Perlin Noise

2.1.1. Der Algorithmus

Mit „Perlin Noise“ bezeichnet man eigentlich eine Funktion, welche jedem X-Wert einen zufälligen Y-Wert zuordnet, also eine Zufallsfunktion. Zwischen den Punkten der Funktion wird interpoliert, was wir hier als Hauptaufgabe von Perlin Noise sehen.
Es gibt mehrere Arten der Berechnung.
Vorerst braucht man die Punkte, zwischen denen dann interpoliert werden soll. Dafür erstellt man sich nur einen array von TPoint, setzt durch eine for-Schleife die X-Werte, abhängig von der Zählvariable der Schleife. Dazu die dazugehörigen Y-Werte, welche lediglich Zufallszahlen mit einen gewählten Definitionsbereich sind.
Mit diesen array hat man dann alle Funktionswerte hintereinander gespeichert und kann nun problemlos auf die erforderlichen Werte zugreifen, die man benötigt, um die Interpolationsberechnung durchzuführen.
Die drei am meist Verbreitesten Methoden sind die lineare, cubische und Kosinus-Interpolation

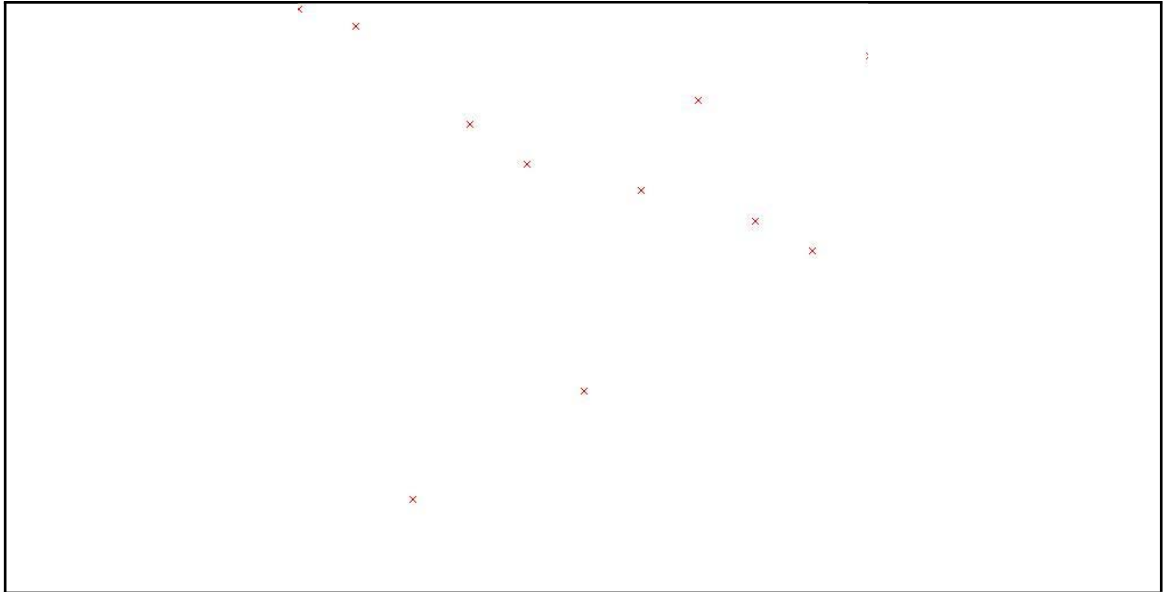
Hierfür legt man sich drei Funktionen an:

```
function LinearInterpolation( y1, y2, x: Single ): Single;
begin
 Result := ( y1 * (1-x) ) + ( y2 * x );
end;

function CosinusInterpolation( y1, y2, x: Single ): Single;
var
  s: Single;
begin
 s := ( 1 - cos(x*Pi) ) * 0.5;
 Result := ( y1*(1-s) ) + ( y2*s );
end;

function CubicInterpolation( y0, y1, y2, y3, x: Single ): Single;
var
  P, Q, R: Single;
begin
 P := (y3 - y2) - (y0 - y1);
 Q := (y0 - y1) - P;
 R := y2 - y0;
 Result := P*(x*x*x) + Q*(x*x) + R*x + y1;
end;
```

Diese Funktionen berechnen den interpolierten Y-Wert zu einen beliebigen X-Wert zwischen zwei Punkten.

Die Ergebnisse sind sehr unterschiedlich. Während die lineare Interpolation sehr geradlinig aussieht, erscheint die cubische Interpolation sehr abgerundet.

Wenn man die Liste optisch darstellt, sieht diese beispielsweise so aus:



Zwischen diesen Punkten wird dann interpoliert.

Für diese Liste würden so die interpolierten Punkte aussehen.

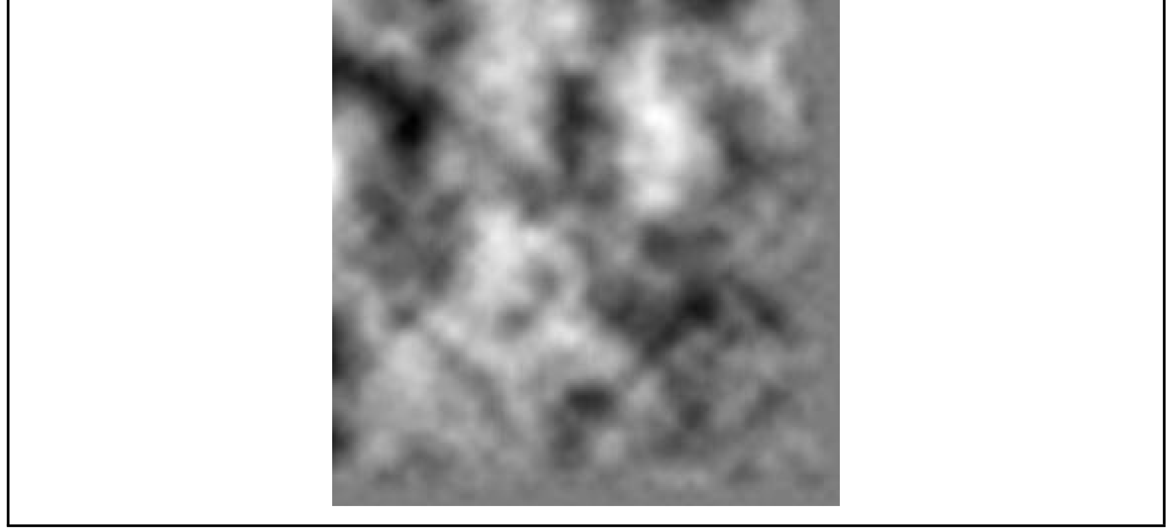
Lineare Interpolation	Cosinus-Interpolation	Cubische Interpolation

2.1.2. Anwendung auf Heightmaps

2.1.2.1. Die Heightmap

Eine Heightmap ist eine Graustufen-Bitmap, in der Höhendaten für ein Terrain gespeichert werden.

Die meisten, simplen Terrains werden aus einer Heightmap geladen, weil es die einfachste Methode ist, um die Höhendaten für die einzelnen Vertices der Landschaft zu speichern und zu laden.
Das Prinzip ist ganz einfach. Dunkle Stellen der Bitmap entsprechen Täler, somit wird Schwarz die tiefste Stelle kennzeichnen, und hellen Stellen sind Berge / Hügel, woraus man schlussfolgern kann, dass ein weißer Punkt auf der Heightmap die höchste Stelle des Terrains kennzeichnen wird.
Um es sich noch einfacher vorstellen zu können denkt man sich, dass ein schwarzer Pixel der Zahl 0 entspricht, und dass ein weißer Pixel für die Zahl 1 steht. Alle Farbwerte dazwischen stehen für eine entsprechende Zahl zwischen 0 und 1. Nun multipliziert man diese Zahl mit einer gewünschten Höhe und erhält somit die gespeicherte bzw. geladene Höhe.
So könnte eine Heightmap aussehen.



Helle Stellen entsprechen Berge und dunkle Stellen Täler.

2.1.2.2. Perlin Noise und Heightmaps

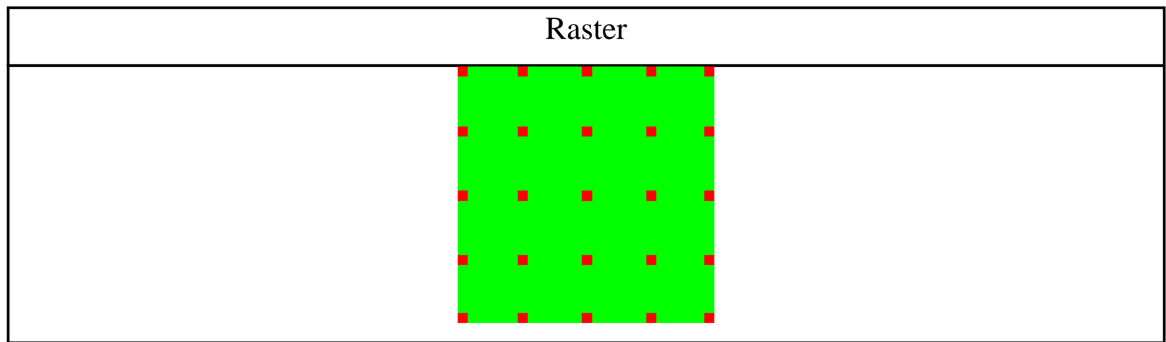
Wenn man eine zufällige Anordnung von unterschiedlichen hohen bzw. tiefen Bergen und Tälern haben möchte, greift man meist auf Perlin Noise zurück, um die Landschaftsdaten zu erstellen. Um eine diese Daten zu speichern und zu laden, nimmt man oft, wie schon gesagt, Heightmaps.
Dies beides kann man nun in ein paar wenigen Schritten kombinieren um eine Zufallslandschaft zu erstellen.

Zuerst braucht man eine Liste mit den Zufallspunkten. Da eine Heightmap aber zweidimensional ist, brauchen wir auch einen zweidimensionalen und quadratischen array. Quadratisch, weil unsere Heightmap immer quadratisch sein wird, weil man so besser damit rechnen kann.

Dafür erstellt man sich ganz einfach einen neuen Record.

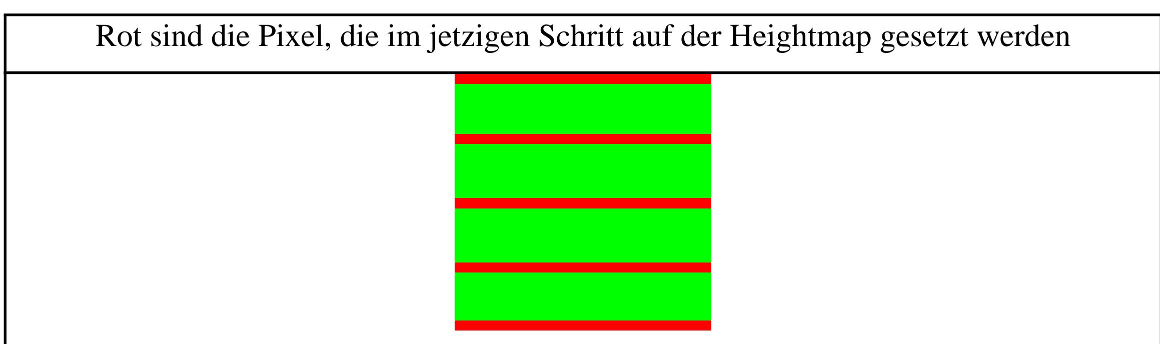
```
type
  TMapSI = class
  public
   data: array of array of SmallInt;
   constructor Create( size: WORD );
  end;

constructor TMapSI.Create;
begin
 inherited Create;
 setlength( data, size, size );
end;
```

Man errechnet die „Breite“ des arrays, indem man die Breite der zukünftigen Heightmap
durch das Intervall teilt. Das Intervall gibt an, nach wie viel Pixel ein Zufallspixel gesetzt werden soll. Diesen array füllt man mit Zufallswerten. Somit erstellt man ein Raster mit Zufallswerten. Je kleiner das Intervall, desto enger sind die Punkte im Raster angeordnet und umso strukturierter wird die Landschaft.
So kann man sich das Raster vorstellen. An den roten Stellen setzen wir die Zufallspixel und an den grünen Stellen wird in späteren Schritten an Hand der Zufallspixel interpoliert.


Raster

Nachdem man den array mit den Zufallswerten erstellt haben, erzeugt man eine quadratische Bitmap, mit der gewünschten Breite, und einen zweidimensionalen, quadratischen array, der als Zwischenspeicher dient, mit der selben Größe. Die Höhe und Breite sollte aus einer 2er Potenz entstehen, damit lässt sich später auch besser arbeiten. Der Inhalt der Bitmap ist egal, da er überschrieben wird. Diese Bitmap ist die, noch ungefüllte, Heightmap.

Die Zufallswerte und die leere Heightmap sind erstellt, also wird diese jetzt gefüllt. Vorerst werden aber alle in den nächsten Schritten ausgerechnete Werte in den Zwischenspeicher (den gleichgroßen array wie die Heightmap) geschrieben und erst später in die Heightmap geschrieben. Im Abschnitt wird trotzdem von “Pixel schreiben” gesprochen, weil der Zwischenspeicher fast genauso wie die Heightmap ist, nur einen größeren Definitionsbereich hat. Diese Formulierung klingt schöner.
Zum Füllen der Daten geht mal erst die Zeilen im Raster durch, und danach erst alle Spalten.

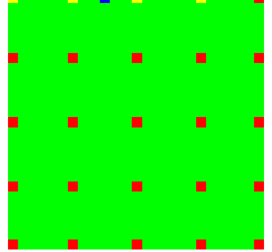

Rot sind die Pixel, die im jetzigen Schritt auf der Heightmap gesetzt werden

Dabei füllt man jede Zeile wiederum schrittweise. Man fängt mit den ersten

Rasterpunkt in der Zeile an, nimmt den rechts anliegenden (also den zweiten) Rasterpunkt dazu und interpoliert zwischen den beiden. Danach interpoliert man zwischen den Rasterpunkten zwei und drei, bis die Zeile fertig ist und man in der nächsten Zeile fortfährt.

Wenn man die cubische Interpolation nimmt, braucht man zu den 2 genannten Rasterpunkten die noch den vorherigen und nachfolgenden Rasterpunkt. Zu diesen 4 Punkten, braucht man noch eine Angabe, wo sich der aktuelle Pixel befindet, den man schreiben möchte. Diese Angabe ist ein Wert zwischen 0 und 1. Die Ziffer 0 sagt aus, dass sich der zu schreibende Pixel genau auf den ersten unseren 4 benötigten Rasterpunkten befindet. Und die Ziffer 1, wenn der zu schreibende Pixel genau auf dem vierten unserer 4 Rasterpunkte liegt.

Darstellung der erklärten Vorgehensweise



gelbe Pixel: die 4 benötigten Rasterpunkte (davon werden nicht die Positionen benötigt,
sondern die Werte, die diesen zugeordnet sind. Das waren die Zufallsziffern,
die am Anfang erstellt wurden)
blauer Pixel: der zu schreibende Pixel, der auf die Heightmap geschrieben werden soll.
Die Positionsangabe entspräche hier ‘0.5’ - genau der mittlere Pixel
zwischen den ersten und vierten Rasterpunkt

Der Funktion *CubicInterpolation* übergibt man nun die Werte unserer 4 Rasterpunkte und diese Positionsangabe.

Danach muss eine weitere ineinander verschachtelte Schleife alle Werte von 0 bis

Bitmapbreite-1 und alle dazugehörigen Y-Werte durchlaufen. Jetzt werden wieder alle nötigen Parameter für die Funktion CubicInterpolation gesucht um diese Funktion aufrufen zu können. Den erhaltenen Wert speichert man wieder in den quadratischen array.

Wenn man alle Zeilen des Rasters fertig hat, muss man die Spalten durchgehen. Hier aber ohne Intervall, sondern jeder Pixel wird durchlaufen. Für diese Berechnung brauchen wir wieder die Positionsangabe und die vier Rasterpunkte, die aber dieses mal auf der Vertikalen aneinander liegen, denn es sollen ja alle Spalten gefüllt werden.

Wenn man dies für jede Spalte der Heightmap getan hat, sind nun alle Höhendaten zusammen.

Wir werden diese Daten später in einen einzigen Farbkanal einer Bitmap speichern. Da ein Farbkanal einer Bitmap nur 1 Byte groß ist, was den Wert von 0 bis 255 entspricht, müssen wir die Werte in den array (Definitionsbereich von 0 < Wert < 256) anpassen. Dazu sucht man den kleinsten und den größten Wert im array und berechnet jeden Wert mit der folgenden Formel neu:

```
Wert = round( (Wert - Minimum) * (255/(Maximum-Minimum)) );
```

Die Funktion dazu würde so aussehen:

```
procedure SetMinMaxValues;
var
  ix,iy: WORD;
  map : TMapSI;
  min : Byte;
  max : Byte;
  col : Byte;
begin

 map := TMapSI.Create( heightmap.Width );
 min := 255;
 max := 0;

 for iy := 0 to heightmap.Height-1 do
 begin

  for ix := 0 to heightmap.Width-1 do
  begin

   map.data[ix,iy] := GetRValue( heightmap.Canvas.Pixels[ix,iy] );
   if map.data[ix,iy] < min then min := map.data[ix,iy];
   if map.data[ix,iy] > max then max := map.data[ix,iy];
```

```
  end;

 end;

 for iy := 0 to heightmap.Height-1 do
 begin

  for ix := 0 to heightmap.Width-1 do
  begin

   col := round( (map.data[ix,iy] - min) * (255/(max-min)) );
   heightmap.Canvas.Pixels[ix,iy] := RGB( col, col, col );
  end;

 end;

end;
```

Jetzt muss man nur noch alle Werte des quadratischen Arrays durchgehen und in die Bitmap schreiben. Zur optischen Darstellung beschreibt man den roten, grünen und blauen Kanal mit dem Wert, damit man die Graustufenbitmap erhält.
Später, wenn dann noch die Lightmap und Texturemap hinzukommen, schreibt man den Wert nur noch in den roten Kanal.

Alles zusammengefasst sieht dann so aus:

```
procedure GenerateHeightmap( const Intervall: WORD; const Amplitude: WORD;
const Resolution: WORD; var Result: TMapSI );
var
  ix,iy : WORD; // SchleifenVariablen
  i, col: SmallInt; // SpeicherVariablen
  values: array of array of SmallInt; // Zufallswerte
  valmap: array of array of SmallInt; // Zufallswerte mit X-Richtung interpoliert
begin

 // Bitmap erstellen
 Result := TMapSI.Create( Resolution );

 // Arraylängen setzen
 setlength( values, (Resolution div Intervall)+1, (Resolution div Intervall)+1 );
 setlength( valmap, Resolution, (Resolution div Intervall)+1 );
 col := 0; // unsaved code verhindern!
 Randomize;

 // Zusallswerte setzen

 for iy := 0 to (Resolution div Intervall)-1 do
 begin

  for ix := 0 to (Resolution div Intervall)-1 do
  begin

   values[ix][iy] := random(2*Amplitude)-Amplitude;
  end;

 end;

 // in X-Richtung interpolieren

 for iy := 0 to (Resolution div Intervall)-1 do
 begin

  for ix := 0 to (Resolution div Intervall)-1 do
  begin

   for i := 0 to Intervall-1 do
```

```
   begin

    // auf Randposition prüfen, da es dort keine Nachbarpunkte gibt
    // wenn X am linken Rand ...
    if ix = 0 then
    begin

     col := trunc(CubicInterpolation( values[ix][iy],
                                      values[ix][iy],
                                      values[ix+1][iy],
                                      values[ix+2][iy],
                                      i/Intervall ) );
    end;

    // wenn X zwischen linken und rechten Rand ...
    if ix in [1..(Resolution div Intervall)-2] then
    begin

     col := trunc(CubicInterpolation( values[ix-1][iy],
                                      values[ix][iy],
                                      values[ix+1][iy],
                                      values[ix+2][iy],
                                      i/Intervall ) );
    end;

    // wenn X am rechten Rand ...
    if ix in [(Resolution div Intervall)-2..(Resolution div Intervall)] then
    begin

     col := trunc(CubicInterpolation( values[ix-1][iy],
                                      values[ix][iy],
                                      values[ix+1][iy],
                                      values[ix+1][iy],
                                      i/Intervall ) );
    end;

    // interpolierten Punkt speichern
    valmap[ix*Intervall+i][iy] := col;
    Result.data[ix*Intervall+i][iy*Intervall] := col;
   end;

  end;

 end;

 // in Y-Richtung interpolieren

 for iy := 0 to (Resolution div Intervall)-1 do
 begin

  for ix := 0 to Resolution-1 do
  begin

   for i := 0 to Intervall-1 do
   begin

    // auf Randposition prüfen, da es dort keine Nachbarpunkte gibt
    // wenn Y am oberen Rand ...
    if iy = 0 then
    begin

     col := trunc(CubicInterpolation( valmap[ix][iy],
                                      valmap[ix][iy],
                                      valmap[ix][iy+1],
                                      valmap[ix][iy+2],
                                      i/Intervall ) );
    end;

    // wenn Y zwischen oberen und unteren Rand ...
    if iy in [1..(Resolution div Intervall)-2] then
    begin

     col := trunc(CubicInterpolation( valmap[ix][iy-1],
                                      valmap[ix][iy],
                                      valmap[ix][iy+1],
                                      valmap[ix][iy+2],
```

```
                                   i/Intervall ) );
    end;

    // wenn Y am unteren Rand ...
    if iy in [(Resolution div Intervall)-2..(Resolution div Intervall)] then
    begin

     col := trunc(CubicInterpolation( valmap[ix][iy-1],
                                   valmap[ix][iy],
                                   valmap[ix][iy+1],
                                   valmap[ix][iy+1],
                                   i/Intervall ) );
    end;

    // Ergebnis speichern
    Result.data[ix][iy*Intervall+i] := col;
   end;

  end;

 end;

end;
```

Wenn man sich jedoch jetzt diese Heightmap anschaut, denn sieht die nicht sehr schön aus.

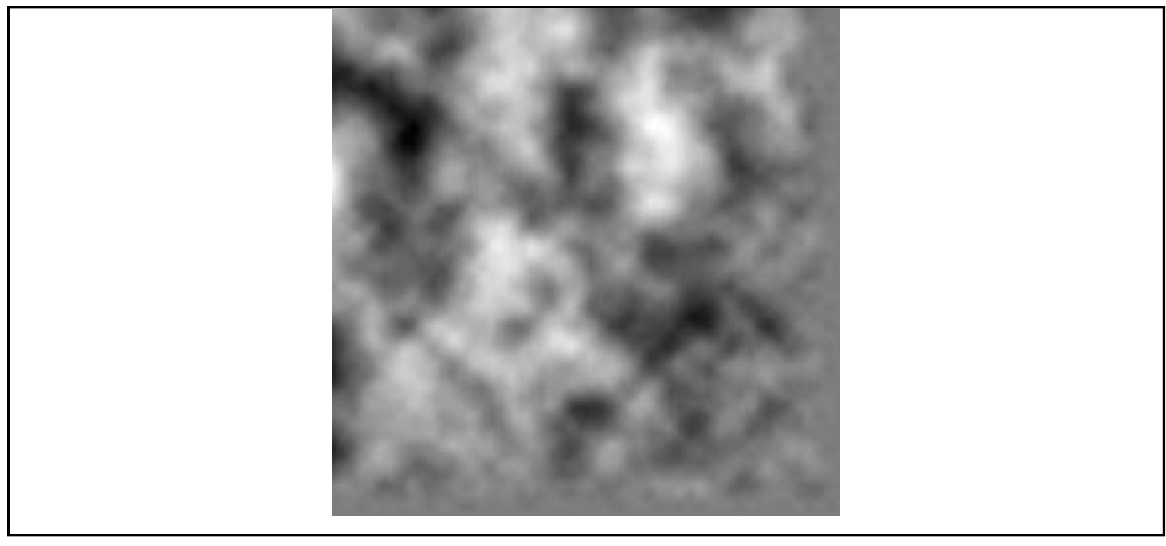
Hier ein Beispiel:

Der Trick ist, einfach mehrere solcher Heightmaps, mit unterschiedlichen Intervall, also unterschiedlich großen Rastergröße, zu erstellen und diese dann mit einen bestimmten Prozentsatz kombinieren.

```
procedure CreateHeightmap( const size : WORD;
const Intervall: WORD );
var
  ix,iy : WORD; // Schleifenvariablen
  col : SmallInt; // Speichervariable
```

```
  val1,
  val2,
  val3,
  val4 : TmapSI; // die vier Speicherarrays für die Heightmaps
begin

 // Btmap erstellen
 heightmap := TBitmap.Create;
 heightmap.Width := size;
 heightmap.Height := size;

 // 4 Heightmaps erstellen
 GenerateHeightmap( Intervall, trunc( 5/100 * 128), size, val1 ); // 8
 GenerateHeightmap( Intervall*2, trunc(15/100 * 128), size, val2 ); // 16
 GenerateHeightmap( Intervall*4, trunc(35/100 * 128), size, val3 ); // 32
 GenerateHeightmap( Intervall*8, trunc(45/100 * 128), size, val4 ); // 64

 for iy := 0 to size-1 do
 begin

  for ix := 0 to size-1 do
  begin

   col := val1.data[ix,iy] +
          val2.data[ix,iy] +
          val3.data[ix,iy] +
          val4.data[ix,iy] + 128;

   if col >= 255 then col := 255;
   if col <= 0 then col := 0;
   heightmap.Canvas.Pixels[ix,iy] := RGB( col, col, col );
  end;

 end;

 SetMinMaxValues;

 // SpeicherFreigabe
 FreeAndNil( val1 );
 FreeAndNil( val2 );
 FreeAndNil( val3 );
 FreeAndNil( val4 );
end;
```

Und damit erhält man dann eine schöne Heightmap, wie diese hier:



2.2. Die Lightmap

Die Lightmap ist, genauso wie die Heightmap, eine Bitmap, in der Informationen in Form einer “Farbe” enthalten sind. In diesem Fall sind es Informationen über die Beleuchtung der Landschaft. Jeder Pixel der Lightmap definiert die Beleuchtungsintensität der dazugehörigen Position in der Heightmap. Da sowohl Heightmap als auch Lightmap die gleiche Größe haben, braucht man bei der Terrainbeleuchtung nur die Höheninformationen aus der Heightmap auslesen und an den gleichen Stellen die Beleuchtungsinformationen aus der Lightmap holen und miteinander kombinieren. Wie man alles kombiniert, wird in späteren Kapiteln behandelt.

Da auch später die Lightmap zusammen mit anderen Daten in eine Datamap gespeichert werden soll, darf sie nur Graustufenfarbinformationen enthalten, damit wir dann nur noch einen Farbkanal dieser Bitmap brauchen, um die Beleuchtungsinformationen zu lesen. Eine Schattenfarbe müsste man dann über andere Wege realisieren.

Die hier beschriebene Methode ist eine von vielen, da die Mathematik viele Möglichkeiten zur Berechnung liefert.
Um die Lightmap berechnen zu können benötigt man die schon erstellte Heightmap und eine weitere gleichgroße Bitmap, deren Inhalt egal ist, da er überschrieben wird. Zudem braucht man einen Richtungsvektor vom Koordinatenursprung zur Lichtquelle, dessen Komponenten nicht größer als 1 sein dürfen, da sonst Pixel bei der Berechnung übersprungen werden, denn der Abstand von 2 Pixel beträgt immer 1. Zur Berechnung kann man sich der Geradengleichung bedienen.
Man geht jeden Pixel der Heightmap durch, berechnet daraus die dreidimensionale Koordinate, indem man X und Z, die der Pixelposition (X|Y) entspricht, beibehält und für Y den roten Farbwert des Pixels übernimmt, welcher einen Wert zwischen 0 und 255 ist.
Für diese Position stellt man die Geradengleichung wie folgt auf:

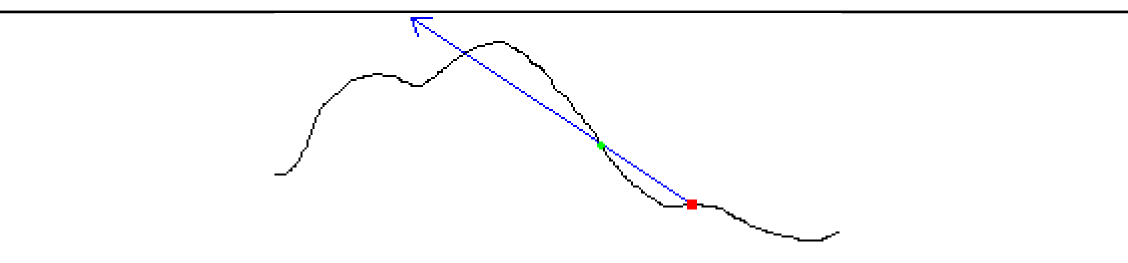
(X|Y|Z) = P(X|Y|Z) + t * L(X|Y|Z)

P ist die Position, die sich aus der Pixelposition ergibt;

L ist der Richtungsvektor zur Lichtquelle;

t ist für diese Zwecke eine positive, ungebrochene Zahl von 1 bis n;

Diese Gerade beschreibt den Lichtstrahl. Man verfolgt diesen vom Punkt auf der Heightmap bis hin zur Lichtquelle, die im Unendlichen liegt. Von der Seite könnte man sich das so vorstellen



schwarz: Heightmap von der Seite

rot: Pixel, zu dem die Beleuchtungsinformation berechnet werden soll

blau: Licht-Gerade

grün: Schnittpunkt mit Heightmap (auf roten Pixel fällt also Schatten)

Die Berechnung geht so.

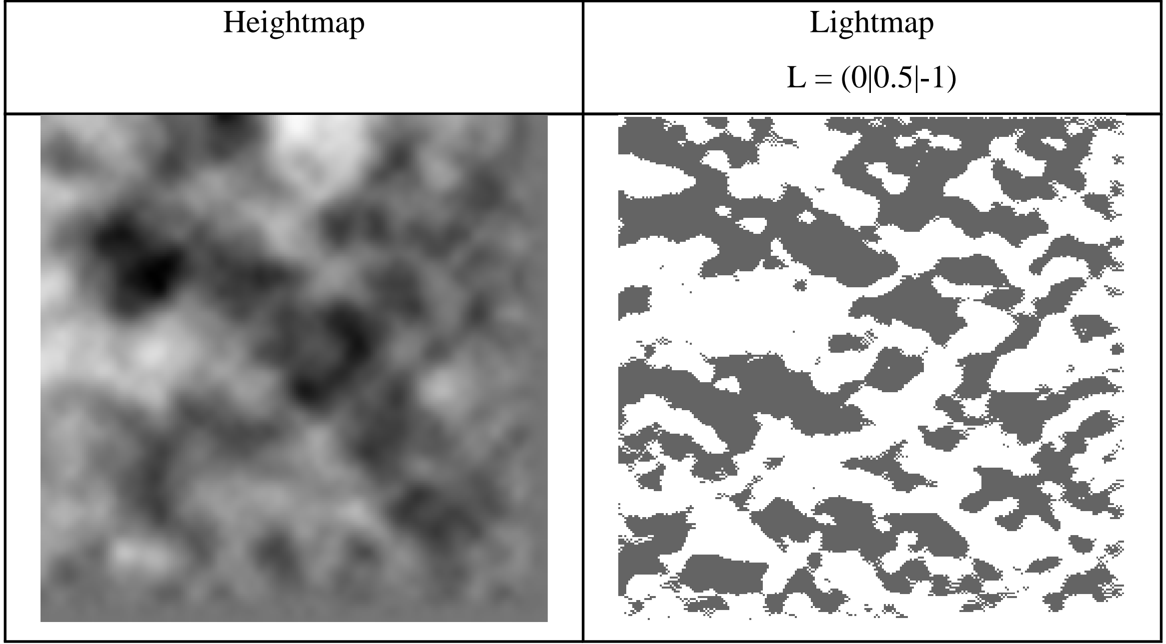
Eine weitere Schleife, die sich innerhalb der beiden Schleifen für den Durchlauf aller Pixel befindet, geht für t alle im Wertebereich liegenden Werte durch, bis entweder der Punkt auf der Gerade die Grenzen der Bitmap überschritten hat, oder der Pixel aus der Heightmap an der berechneten Stelle der Gerade einen größeren Y-Wert (Höhenwert) hat, als der Punkt der Gerade.

Weiterhin bricht die Schleife ab, wenn Y am berechneten Punkt höher als 255 ist, denn kein Pixel kann einen größeren Wert besitzen.

Nach dieser Schleife wird abgefragt, warum sie abgebrochen wurde. Wenn sie wegen des

Erreichens der Bitmapgrenzen, d.h. Das Erreichen der Kanten der Bitmap oder eines Wertes über 255 für Y, abgebrochen wurde, wird der Pixel der Lightmap, aus dessen

Koordinate man den Punkt P ausgerechnet hatte, voll beleuchtet und einen hellen Graustufenwert dem Lightmappixel zugewiesen.
Wenn aber die Schleife abgebrochen wurde, weil der Y-Wert der Position (von der Gerade) einen kleineren Wert als den Farbton des Heightmappixel hat, wird der Punkt der Lightmap mit einen dunklen Graustufenwert gefüllt, dort ist also Schatten.



Heightmap	Lightmap L = (0|0.5|-1)

Quelltext (Perlin Noise/cLightmap.pas):

```
function Calc3DPosition( PosX, PosY: WORD; siteIndex: WORD ): TD3DXVector3;

procedure CreateLightmap( LightDir: TD3DXVector3 );
var
  lightmap: TBitmap = nil;
  SunX : WORD = 60;
  SunY : WORD = 60;

  implementation

  // ---------------------------------------------------------------------------
  // +++ Calc3DPosition
  // ---------------------------------------------------------------------------

function Calc3DPosition( PosX, PosY: WORD; siteIndex: WORD ): TD3DXVector3;
var
  X,Y,Z: Single;
  sx,sy: Single;
begin

 sx := ((PosX / 117) - 0.5) * 2;
 sy := -((PosY / 117) - 0.5) * 2 * 8;
 X := 0;
 Y := 0;
 Z := 0;
 // links
 if siteIndex = 0 then
 begin
```

```
  X := -1;
  Y := sy;
  Z := sx;
 end;

 // vorne
 if siteIndex = 1 then
 begin

  X := sx;
  Y := sy;
  Z := 1;
 end;

 // rechts
 if siteIndex = 2 then
 begin

  X := 1;
  Y := sy;
  Z := sx;
 end;

 // hinten
 if siteIndex = 3 then
 begin

  X := sx;
  Y := sy;
  Z := -1;
 end;

 Result := D3DXVector3( X, Y, Z );
end;

// ---------------------------------------------------------------------------
// +++ CreateLightmap
// ---------------------------------------------------------------------------

procedure CreateLightmap( LightDir: TD3DXVector3 );
var
  hmap : TMapB;
  ix,iy: WORD;
  i : WORD;
  point: TD3DXVector3;
  p2 : TD3DXVector3;
  light: Boolean;
  done : Boolean;
begin

 if heightmap = nil then
 begin

  ShowMessage( 'Erst eine heightmap generieren!' );
  exit;
 end;

 //D3DXVec3Normalize( lightDir, lightDir ); // evtl.
 hmap := TMapB.Create( heightmap.Width );
 lightmap := TBitmap.Create;
 lightmap.Width := heightmap.Width;
 lightmap.Height := heightmap.Height;

 for iy := 0 to lightmap.Height-1 do
 begin

  for ix := 0 to lightmap.Width-1 do
  begin

   hmap.data[ix,iy] := GetRValue(heightmap.Canvas.Pixels[ix,iy]);
  end;

 end;

 for iy := 0 to lightmap.Height-1 do
```

```
 begin

  for ix := 0 to lightmap.Width-1 do
  begin

   point := D3DXVector3( ix, hmap.data[ix,iy], iy );
   light := TRUE;
   done := FALSE;
   i := 0;

   repeat

    inc(i);
    p2.x := point.x + i*(lightDir.x);
    p2.y := point.y + i*(lightDir.y);
    p2.z := point.z + i*(lightDir.z);

    if (p2.x > lightmap.Width-1) or (p2.x < 0) or
       (p2.z > lightmap.Height-1) or (p2.z < 0) then
    begin
     done := TRUE

    end else
    begin

     if hmap.data[trunc(p2.x),trunc(p2.z)] >= p2.y then
     begin
      light := FALSE;
     end;

    end;

   until (light = FALSE) or (done = TRUE);

   if (light = FALSE) and (done = FALSE) then
   begin
    lightmap.Canvas.Pixels[ix,iy] := RGB(100,100,100);
   end else
   begin
    lightmap.Canvas.Pixels[ix,iy] := RGB(255,255,255);
   end;

  end;

 end;

end;

end.
```

2.3. Die Texturemap

Jedes Terrain hat verschiedene Texturen. So ist am Wasser Sand, an Stellen ohne Neigung Wiese, an Neigungen Fels und in hohen Höhen Schnee oder mehr Fels. Für diese Selektion kann man Texturemaps nehmen.
Es gibt viele verschiedene Arten von Texturemaps, aber letztendlich sagen sie alle, an welcher Stelle des Terrains welche Textur gesetzt werden muss.
Ich habe mich an eine spezielle Methode gesetzt. Bei meiner Methode, bei der ich nicht weiß, in wie weit sie verbreitet ist, kann man 4 verschiedene Texturen verarbeiten.
Es handelt sich um eine Bitmap, die 4 Kanäle mit je 8 Bit hat, also jeder Pixel 32 Bit groß ist. Jeder Kanal steht für die Intensität einer Textur. Ist z.B. der rote Kanal auf 255 gesetzt, so wird die erste Textur an der Stelle zu 100% genommen. Bei grün die zweite Textur. Wenn mehrere Kanäle beschrieben sind, dann wird zwischen mehreren Texturen interpoliert.
Die Berechnung ist einfach.
Auch hier braucht man wieder zuerst die Heightmap und eine gleich große Bitmap als Texturemap, die 32 Bit groß sein muss, damit man den Alphakanal mit beschreiben kann.
Man geht jeden Pixel der Heightmap durch und berechnet die dazugehörige Koordinate P1(x,y,z). Für diesen Punkt berechnet man die Normale. Dazu nimmt man diesen Punkt und rechnet aus den links und unter dem Pixel liegenden Bildpunkte der Heightmap die Koordinaten für P2(x,y,z) und P3(x,y,z) aus. Nun zieht man jeweils von P2 und P3 den Vektor P1 ab und erhält daraus V1 und V2. Danach berechnet man das Vektorprodukt aus V1 und V2 und normalisiert das Ergebnis, um die Normale zu erhalten. Da aber die Normale nicht nur von den Pixel links und unter dem aktuellen abhängig ist, muss man auch noch die gleiche Reihenfolge für den unteren und rechten, rechten und oberen, oberen und linken Pixel durchführen. Die Komponenten der 4 Normalen, die man erhalten hat, addiert man miteinander und teilt sie durch 4. Jetzt hat man die richtige Normale.
Diese Normale nimmt man jetzt für das Skalarprodukt mit den Vektor(0,1,0), welcher der Normalenvektor der X-Z-Ebene ist. Das Ergebnis, was ein reeller Wert zwischen 0 und 1 sein wird, multipliziert man mit 255 und schreibt den erhaltenen Wert in den

roten Kanal der Texturemap. Das steht für Gras an ebenen Flächen. Das Ergebnis des Skalarprodukts rechnet man danach minus 1 und multipliziert diesen Wert dann wieder mit 255 und schreibt das Ergebnis in den grünen Kanal. Das steht dann für Fels an kleinen Hängen. Jetzt nimmt man den roten Farbwert des Pixels der Heightmap und testet, ob er sich wischen 0 und einer beliebigen Höhe bis 255 befindet, denn in diesen Bereich soll die dritte Textur (Sand) genommen werden. Zu empfehlen wäre ein Wert um 35. Jetzt interpoliert man den blauen Farbwert so, dass wenn die rote Farbe des Heightmappixels die gewünschte Höhe beträgt der blaue Kanal gleich 0 ist und bei einen schwaren Heightmappixel der blaue Kanal mit 255 beschrieben wird. Vom erhaltenen blauen Wert errechnet man den Prozentsatz, wobei 255 gleich 100% entsprechen. Danach zieht man soviel Prozent von den roten und den grünen Kanal ab. Den gleichen Vorgang wiederholt man für eine obere Grenze, für die der Wert 200 sehr gut ist.
Man beschreibt den Alphakanal mit 255, wenn der Farbwert des Heightmappixels gleich 255 entspricht und mit 0, wenn der Pixel der Obergrenze entspricht.

Das Ergebnis ist eine Bitmap, deren 4 Kanäle Informationen über die Intensität der dazugehörigen Textur enthalten.

Quellcode (...Perlin Noise/cTexturemap.pas):

```
var
  texturemap: TBitmap = nil;
  implementation
  // ---------------------------------------------------------------------------
  // +++ GetNormal
  // ---------------------------------------------------------------------------

  function GetNormal(point1, point2, point3: TD3DXVector3): TD3DXVector3;
var
  v1, v2: TD3DXVector3;
begin

 D3DXVec3Subtract(v1, point2, point1);
 D3DXVec3Subtract(v2, point3, point1);
 D3DXVec3Cross(Result, v1, v2);
end;

// ---------------------------------------------------------------------------
// +++ CreateTexturemap
// ---------------------------------------------------------------------------

procedure CreateTexturemap( DeepHeight, HighHeight: Byte );
type
Color = record
b,g,r,a: Byte;
end;

var
  ix,iy : WORD;
```

```
  map : TMapSI;
  n1,n2,n3,n4: TD3DXVector3;
  normal : TD3DXVector3;
  dot : Single;
  col : Color;
  pxl : PCardinal;
begin

if heightmap = nil then
begin

 ShowMessage( 'Zuerst die Heightmap generieren!' );
 exit;
end;

map := TMapSI.Create( heightmap.Width );

for iy := 1 to heightmap.Height-2 do
begin

 for ix := 1 to heightmap.Width-2 do
 begin

  map.data[ix,iy] := GetRValue(heightmap.Canvas.Pixels[ix,iy]);
 end;

end;

texturemap := TBitmap.Create;
texturemap.Width := heightmap.Width;
texturemap.Height := heightmap.Height;
texturemap.PixelFormat := pf32Bit;

for iy := 1 to texturemap.Height-2 do
begin

 pxl := texturemap.ScanLine[iy];
 inc(pxl);

 for ix := 1 to texturemap.Width-2 do
 begin

  n1 := GetNormal( D3DXVector3(ix+1, map.data[ix+1,iy-1], iy-1),
                   D3DXVector3(ix-1, map.data[ix-1,iy-1], iy-1),
                   D3DXVector3(ix, map.data[ix,iy], iy) );
  n2 := GetNormal( D3DXVector3(ix+1, map.data[ix+1,iy+1], iy+1 ),
                   D3DXVector3(ix+1, map.data[ix+1,iy-1], iy-1 ),
                   D3DXVector3(ix, map.data[ix,iy], iy) );
  n3 := GetNormal( D3DXVector3(ix-1, map.data[ix-1,iy+1], iy+1),
                   D3DXVector3(ix+1, map.data[ix+1,iy+1], iy+1 ),
                   D3DXVector3(ix, map.data[ix,iy], iy) );
  n4 := GetNormal( D3DXVector3(ix-1, map.data[ix-1,iy-1], iy-1 ),
                   D3DXVector3(ix-1, map.data[ix-1,iy+1], iy+1 ),
                   D3DXVector3(ix, map.data[ix,iy], iy) );

  normal.x := (n1.x + n2.x + n3.x + n4.x) / 4;
  normal.y := (n1.y + n2.y + n3.y + n4.y) / 4;
  normal.z := (n1.z + n2.z + n3.z + n4.z) / 4;
  D3DXVec3Normalize( normal, normal );
  dot := D3DXVec3Dot( normal, D3DXVector3(0,1,0) );
  dot := sqr(1-dot);

  // - Rot = Flach -
  col.r := trunc((1-dot)*255);
  // - Grün = Hang -
  col.g := trunc(dot*255);
  // - Blau = GewässerTal -
  col.b := 0;
  if map.data[ix,iy] <= DeepHeight then
  begin

   col.b := 255 - trunc( map.data[ix,iy]*(255/DeepHeight) );
   col.b := trunc( sqrt(col.b/255)*255 );
   col.r := trunc( col.r * (1-(col.b/255)) );
   col.g := trunc( col.g * (1-(col.b/255)) );
  end;
```

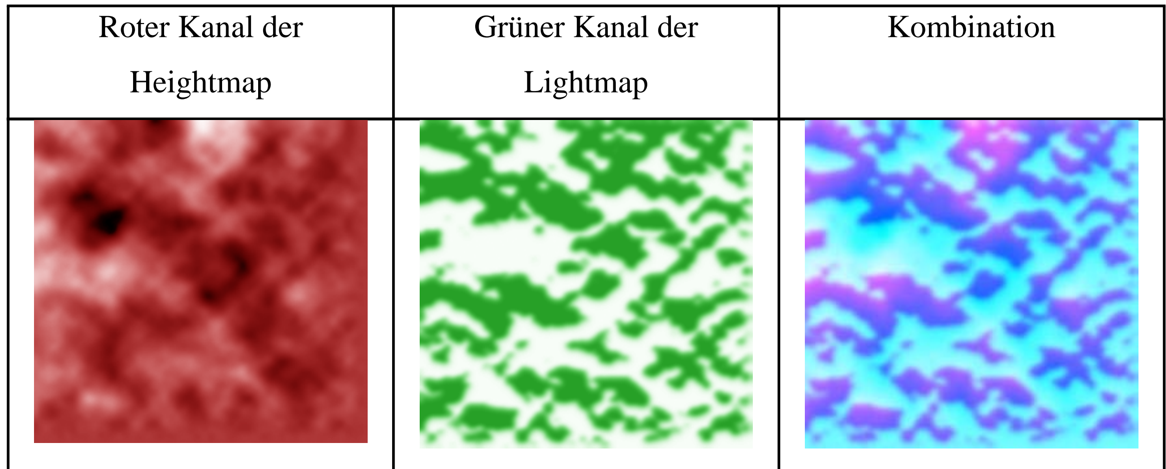
```
  // - Alpha = BergSpitze -
  col.a := 0;
  if map.data[ix,iy] >= HighHeight then
  begin

   col.a := trunc( (map.data[ix,iy]-HighHeight)*(255/(255-HighHeight)) );
   col.a := trunc( sqrt(col.a/255)*255 );
   col.r := trunc( col.r * (1-(col.a/255)) );
   col.g := trunc( col.g * (1-(col.a/255)) );
  end;

  // Pixel setzen
  pxl^ := Cardinal(col);
  inc(pxl);
 end;

end;

end;
```

2.4. Die Datamap

Die bis jetzt behandelte Methode, um ein Terrain zu speichern, ist ineffektiv. Man hat die

Heightmap, Lightmap und Texturemap jeweils in eine separate Bitmap gespeichert. Diese 3 Maps kann man zu nur einer einzigen 32-Bit Bitmap zusammenfügen, welche man als „Datamap“ bezeichnen könnte. Die Bezeichnung ist frei gewählt und könnte ebenso „Terrainmap“ lauten.

Da die Height- und Lightmap nur Graustufenbitmaps sind, benötigen diese nur einen Farbkanal. Somit kann man diese beiden Maps ohne weitere Probleme kombinieren, indem man bei der Datamap den roten Farbwert der Heightmap und den Grünen der Lightmap übernimmt.

Roter Kanal der Heightmap	Grüner Kanal der Lightmap	Kombination
		

Um die Datamap zu vervollständigen muss man auch die Texturemap hinzufügen. Jedoch geht dies nicht so einfach, da diese selbst alle 4 Kanäle der Bitmap benötigt und die Datamap nur noch 2 freie hat.

Der Trick ist, innerhalb eines Farbkanals zwischen 2 Farben zu interpolieren. Zuerst kombiniert man die rote und grüne Farbe der Texturemap zu einer Variablen C, die einen Wert zwischen 0 und 255 annehmen kann. C ist gleich 0, wenn die rote Farbe gleicht 255 ist und die grüne Farbe 0. Wenn die Rot gleich 0 und Grün gleich 255 ist, setzt man C auf 255. Dazwischen wird interpoliert, d.h. je grüner die Farbe ist, desto mehr konvergiert C an 255 und umso röter die Farbe wird, desto mehr nimmt C den

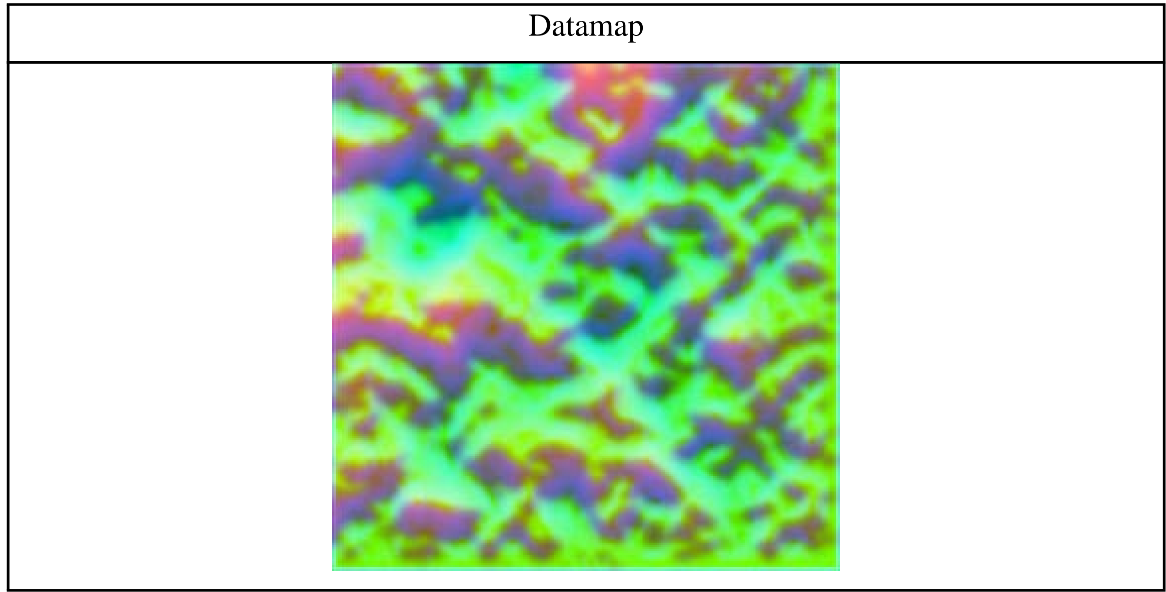
Wert 0 an.

Den interpolierten Wert C schreibt man anschließend in den blauen Farbkanal der Datamap.

Daraufhin führt man selbige Berechnung für den Alpha-Kanal der Datamap durch, um den blauen und transparenten (Alpha) Kanal der Texturemap zu kombinieren.

Texturemap	Rot und Grün der Texturemap zum blauen Kanal der Datamap kombiniert	Blau und Alpha der Texturemap zum Alpha-Kanal der Datamap kombiniert

Nun erhält man die fertige Datamap, welche die Heightmap, Lightmap und Texturemap beinhaltet:

Datamap


Quelltext (...Perlin Noise/cSave.pas):

```
procedure SaveToTerrainmap( fileName: TFileName );
type
TCol = record
b,g,r,a: Byte;
end;

var
  ix,iy: WORD;
  col : TCol;
  coltx: TCol;
  c1,c2: Byte;
  pixel: PCardinal;
  pxltx: PCardinal;
  map : TBitmap;
begin

map := TBitmap.Create;
map.PixelFormat := pf32Bit;
map.Width := heightmap.Width;
map.Height := heightmap.Height;

for iy := 0 to lightmap.Height-1 do
begin

 pixel := map.ScanLine[iy];
 pxltx := texturemap.ScanLine[iy];

 for ix := 0 to lightmap.Width-1 do
 begin

  coltx := TCol(pxltx^);
  c1 := ((255 div 2)-(coltx.r div 2)) + (coltx.g div 2);
  c2 := ((255 div 2)-(coltx.b div 2)) + (coltx.a div 2);

  col.r := GetRValue( heightmap.Canvas.Pixels[ix,iy] );
  col.g := GetRValue( lightmap.Canvas.Pixels[ix,iy] );
  col.b := c1;
  col.a := c2;

  pixel^ := Cardinal(col);

  inc( pxltx );
  inc( pixel );
 end;

 dec( pxltx );
 dec( pixel );
end;

map.SaveToFile( fileName );
FreeAndNil( map );
end;

end.
```

2.5. Blur - Weichzeichner

Besonders an der Lightmap fällt ganz stark auf, dass es sich um eine Pixelgrafik handelt, die das typische „Treppenmuster“ aufweist, was bedeutet, dass die Kanten nicht glatt und gerade, sondern in horizontalen und vertikalen Abschnitten verlaufen. So würde der Schatten auf der Landschaft nicht schön aussehen, deshalb besteht hier noch ein Überarbeitungsbedarf.

Um die Kanten geradlinig aussehen zu lassen verwendet man „Blur“, welcher ein Weichzeichner ist und eigentlich nur alles verwischt.

Da es hier nur darum geht, die Graustufenbitmaps, also die Lightmap und evtl. auch die Heightmap, etwas zu verwischen, kann man den Algorithmus vereinfacht aufbauen.

Man geht alle Pixel der Graustufenbitmap durch und holt sich jeweils den roten Wert des

Pixels und dessen Position P (X|Y). Selbige Daten benötigt man von allen anliegenden Pixel.

Die 9 erhaltenen Farbwerte werden miteinander addiert und das Ergebnis wird durch die Anzahl 9 geteilt. Der erhaltene Wert ist der neue Farbwert, den man an P in den entsprechenden Farbkanal schreibt.

Nachdem man den Vorgang bei allen Pixel durchgeführt hat, erhält man die neue weichgezeichnete Bitmap.

Farbige Bitmaps bedingen lediglich, dass der oben genannte Algorithmus für jeden Farbkanal einzeln angewandt werden muss.

Den Blur kann man für jede Bitmap beliebig oft wiederholen, wobei die Effektivität mit jeder Wiederholung nachlässt, da die Farbdifferenzen zwischen den Pixel immer geringer werden.

Den Blur sollte man im Endeffekt einmal, nach Belieben auch ein weiteres mal anwenden, um ein schöneres Ergebnis zu erzielen.

Lightmap	Einmal weichgezeichnet	Fünfmal weichgezeichnet

2.6. Umwandlung der Pixelfarbe und –position in Vertexdaten

2.6.1 ohne einen QuadTree

Eine kleine Heightmap lässt sich relativ einfach in Vertexdaten umrechnen, die dann letztendlich auf den Bildschirm als ein zusammengefügtes, dreidimensionales Objekt gezeichnet werden.
Wenn man nur eine kleine Heightmap benutzt, kann man das ganze Terrain mit nur einen DrawPrimitive-Aufruf rendern und muss es somit nicht in einzelne Stücke unterteilen. Im Allgemeinen lädt man eine Heightmap, indem man jede Pixelposition in einen Positionsvektor, der sich auf der X-Z-Ebene befindet, umrechnet und die Farbe jedes Pixels zu den Y-Wert umwandelt.

Zuerst legt man sich eine Variable vom Typ TBitmap an, die man z.B. „map" nennen kann, und lädt in diese die Heightmap. Zudem braucht man einen Vertex- und Indexbuffer und jeweils einen Array für die Vertices vom Typ TPosDifVertex (in meinen Units so definiert), der Position und Farbwert speichert, und für die Indices vom Typ Word, da in dieser Liste nur ganzzahlige, positive Zahlenwerte gespeichert werden. Abschließend benötigt man eine Zählvariable, z.B. iVertices, deren Startwert 0 beträgt, für die Anzahl der Vertices. Es werden natürlich noch weiter Variablen benötigt, die aber hier nicht genannt werden müssen.
Zwei ineinander verschachtelte Schleifen gehen alle Pixel der Heightmap durch.
An der Stelle iVertices, von der Vertexliste, speichert man die Position des Vertex, welche sich aus der Pixelposition ergibt, die man durch die Hälfte der Heightmapbreite teilt, damit das Terrain im Punkt (0|Y|0) zentriert wird. Die dazugehörige Y-Position errechnet man aus den roten Farbwert des jeweiligen Pixel, dividiert durch eine beliebige Zahl, um die Höhe der einzelnen „Berge" zu skalieren. Um die Landschaft etwas zu dehnen, kann man die Koordinaten der Vertexposition mit einen beliebigen Wert multiplizieren. Als Farbwert für den Vertex kann man die Pixelfarbe übernehmen.
Damit im nächsten Durchlauf der Schleifen auch der nächste Vertex beschrieben wird, muss man die Variable iVertices um 1 erhöhen.
Am Ende der beiden Schleifen hat man eine Liste, in der alle Pixel der Heightmap,

umgewandelt in Vertexdaten, vorliegen.

Danach muss man den Indexbuffer erstellen, der beim Rendern entscheiden wird, in welcher Reihenfolge die Vertices gerendert werden sollen, damit ein lückenloses Objekt aus Dreiecken entsteht.

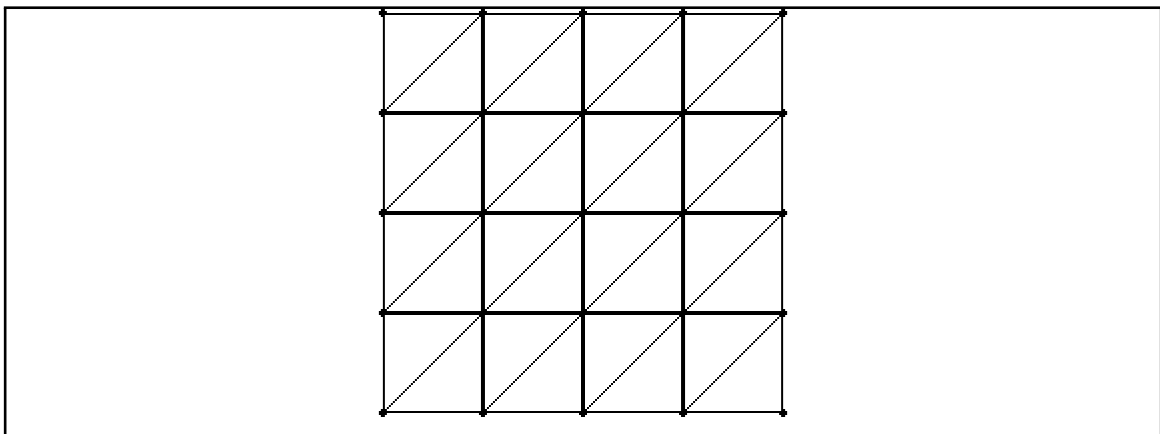
Eine Schleife erstellt die Liste der Indices so, dass immer 3 Punkte zu einen Dreieck verbunden werden, indem sie hintereinander die Positionen von 3 Vertices speichert, die als Eckpunkte des Dreiecks dienen sollen. Dieser Vorgang wird solange wiederholt, bis alle Vertices so zu Dreiecken verbunden wurden sind, dass die Oberfläche keine Lücken mehr aufweist.

Visuell kann man sich diesen Vorgang so vorstellen:

Anordnung der Vertices (von oben betrachtet)	Verbindung der Vertices zum 1. Dreieck	Verbindung zum 2. Dreieck	Verbindung zum 3. Dreieck

Die Schleife endet mit dem letzen Dreieck, mit dem das Gebilde so aussieht:

Alle Vertices zu Dreiecken verbunden



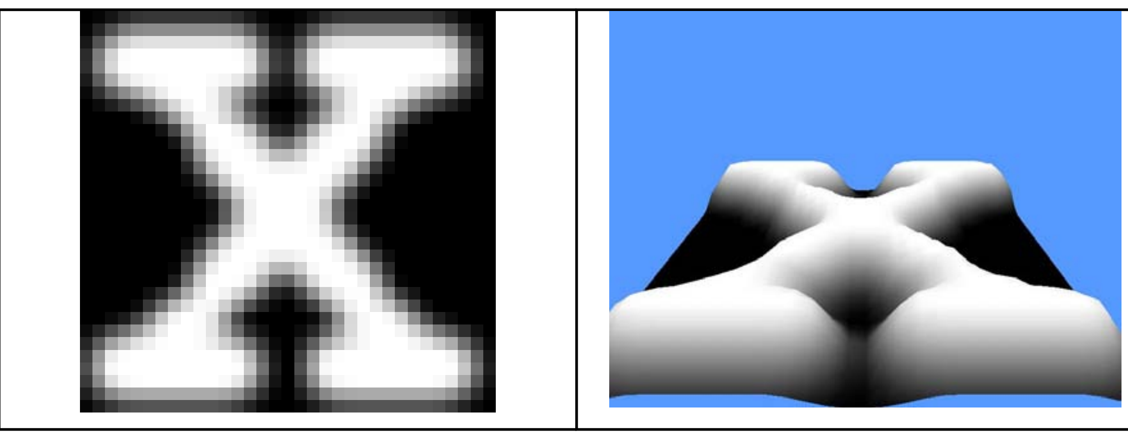
Um diese Dreiecke effektiv rendern zu können, muss man alle Daten in einen Vertex- und Indexbuffer schreiben. Diese Buffer erstellt man mit den Befehlen CreateVertexBuffer und CreateIndexBuffer vom IDirect3DDevice9. Indem man anschließend die Buffer „lockt", holt man sich einen Pointer auf den reservierten Speicher und schreibt dort jeweils die Daten vom Array der Vertices und der Indices hinein und „unlockt" die sie wieder.

Man könnte alles auch mit den Befehl DrawPrimitiveUp rendern, der nur den Array mit den Vertices als Parameter verlangt. Jedoch ist diese Methode langsam, weil sie jedes mal intern die genannten Buffer erstellt.

Jetzt, wo man die renderfähigen Daten alle in den richtigen Variablen hat, muss man sie rendern. Dafür setzt man den Vertex- und Indexbuffer, das „D3DFVF", was angibt, welche Informationen jeder Vertex beinhaltet, und ruft den Befehl DrawIndexedPrimitvie, vom Device, auf.

Wenn alles korrekt gelaufen ist und auch die Daten richtig berechnet und gespeichert wurden, kann man eine kleine, untexturierte Landschaft auf dem Bildschirm erkennen.

Die verwendete Heightmap (vergrößert, da das Original nur 32*32 Pixel groß ist)	Das dazu gerenderte Bild aus einer beliebigen Kameraposition



Quellcode (...einfache Heightmap/Unit1.pas):

```
procedure CreateSimpleTerrain;
var
  heightmap: TBitmap;
  Vertices : array of TPosDifVertex;
  Indices : array of WORD;
  ix,iy : WORD;
  height : Byte;
  iVertices: WORD;
  iIndices : WORD; // Indices-Zähler
  P : Pointer;
begin

 // 32x32 große heightmap laden
 heightmap := TBitmap.Create;
 heightmap.LoadFromFile( 'heightmap2.bmp' );
 heightmap.Width := 32;
 heightmap.Height := 32;
 setlength( Vertices, 1024 );
 setlength( Indices, 31*31*6 );
 iVertices := 0;

 for iy := 0 to 31 do
 begin

  for ix := 0 to 31 do
  begin

   height := heightmap.Canvas.Pixels[ix,iy] mod 256;
   Vertices[iVertices] := GetVertex( ix-32, height/64, iy-32,
                                     RGB(height,height,height));
   inc( iVertices );
  end;

 end;

 Device.CreateVertexBuffer( iVertices*sizeof(TPosDifVertex), D3DUSAGE_WRITEONLY,
                            D3DFVF_POSDIF, D3DPOOL_MANAGED, VB, nil );

 VB.Lock( 0, iVertices*sizeof(TPosDifVertex), P, 0 );
  move( Vertices[0], P^, iVertices*sizeof(TPosDifVertex) );
 VB.Unlock;

 iIndices := 0;

 for iy := 0 to 30 do
 begin

  for ix := 0 to 30 do
  begin

   Indices[iIndices+0] := (iy*32) + ix;
```

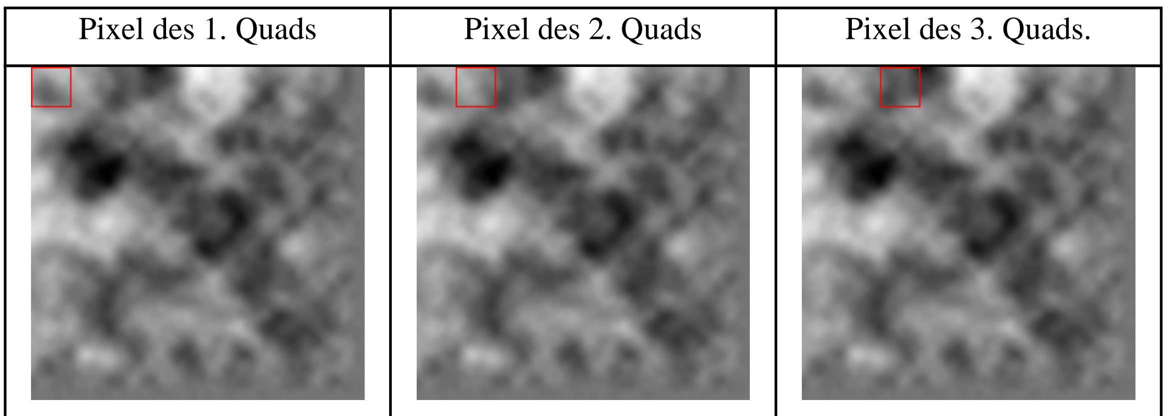
```
   Indices[iIndices+1] := (iy*32) + ix + 1;
   Indices[iIndices+2] := ((iy+1)*32) + ix + 1;
   Indices[iIndices+3] := (iy*32) + ix;
   Indices[iIndices+4] := ((iy+1)*32) + ix + 1;
   Indices[iIndices+5] := ((iy+1)*32) + ix;
   inc( iIndices, 6 );
  end;

 end;

 Device.CreateIndexBuffer( iIndices*sizeof(Indices[0]), D3DUSAGE_WRITEONLY,
                           D3DFMT_INDEX16, D3DPOOL_MANAGED, IB, nil );
 IB.Lock( 0, iIndices*sizeof(Indices[0]), P, 0 );
  move( Indices[0], P^, iIndices*sizeof(Indices[0]) );
 IB.Unlock;
end;

procedure TForm1.OnIdle(Sender: TObject; var done: Boolean);
begin

 done := FALSE;
 Engine.BeginScene( TRUE );
  Device.SetTransform( D3DTS_WORLD, D3DXMatrixIdentity );
  Device.SetTexture( 0, nil );

  Device.SetRenderState( D3DRS_LIGHTING, 0 );
  Device.SetRenderState( D3DRS_CULLMODE, D3DCULL_CW );

  Device.SetFVF( D3DFVF_POSDIF );

  Device.SetStreamSource( 0, VB, 0, sizeof(TPosDifVertex) );
  Device.SetIndices( IB );
  Device.DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0, 1024, 0, 31*31*2 );
 Engine.EndScece;
end;
```

2.6.2 Das Laden der Datamap unter Verwendung des QuadTrees

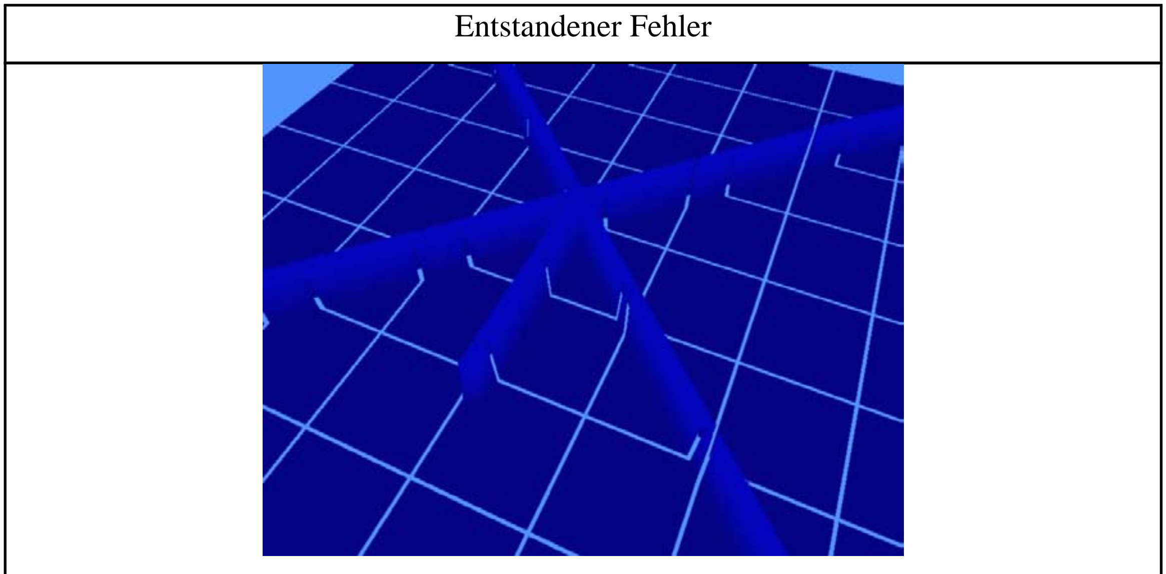
Das Laden der Datamap ist ähnlich dem einer Heightmap, nur dass zusätzlich Beleuchtungs- und Texturinformationen verarbeitet werden müssen. Die Einbindung des QuadTrees fordert zudem eine kleine Umgestaltung des Algorithmus.

Die Datamap wird in eine Variable vom Typ TBitmap geladen, die man z.B. „Map“ nennen kann. Man braucht wieder einen Array für die Vertices, aber, anders als beim Laden einer normalen Heightmap, vom Typ TPosDif2Tex1Vertex (in meinen Units so definiert), da jeder Vertex Informationen zur Position, Beleuchtungsintensität und Textur (Texturselektion und Texturposition) erhält. Weitere Variablen, wie die Buffer und die Liste der Indices müssen nicht noch einmal näher erläutert werden.

Anders als beim Laden einer kleinen Heightmap ohne QuadTree, darf man nicht mehr alle Pixel der Datamap periodisch von links nach rechts durchgehen, sondern muss ein bestimmtes Muster einhalten, in welcher Reihenfolge die Pixel verarbeitet werden. Die kleinsten Quads des hier verwendeten QuadTrees, die am Ende jeden Frames gerendert werden, haben eine Größe von 32*32 Vertices, welche 32*32 Pixel der Height- bzw. Datamap entsprechen. Alle Vertices eines kleinsten Quads müssen im Vertexbuffer direkt hintereinander liegen, um ein schnelles und problemloses Rendern der einzelnen Stellen zu garantieren. Deshalb müssen auch immer die Pixel hintereinander ausgelesen werden, die zusammen einen kleinsten Quad bilden. Das ist der einzigste Unterschied, den man beim Laden unter Verwendung eines QuadTrees beachten muss.
Beim Beginn eines jeden Quads, muss man zu diesen den Startvertex sichern, der beim Rendern von Bedeutung ist und der der aktuellen Vertexanzahl entspricht.

Pixel des 1. Quads	Pixel des 2. Quads	Pixel des 3. Quads.
		

Diese Lösung scheint zwar logisch und richtig zu sein, hat aber einen kleinen Fehler, der oft passiert. Wenn man diese Quads rendert werden die Punkte gezeichnet, aber die Verbindung der Quads untereinander fehlt.

Entstandener Fehler


Angenommen, diese vereinfachte Darstellung wären die Punkte für 4 aneinanderliegende renderbare Quads:

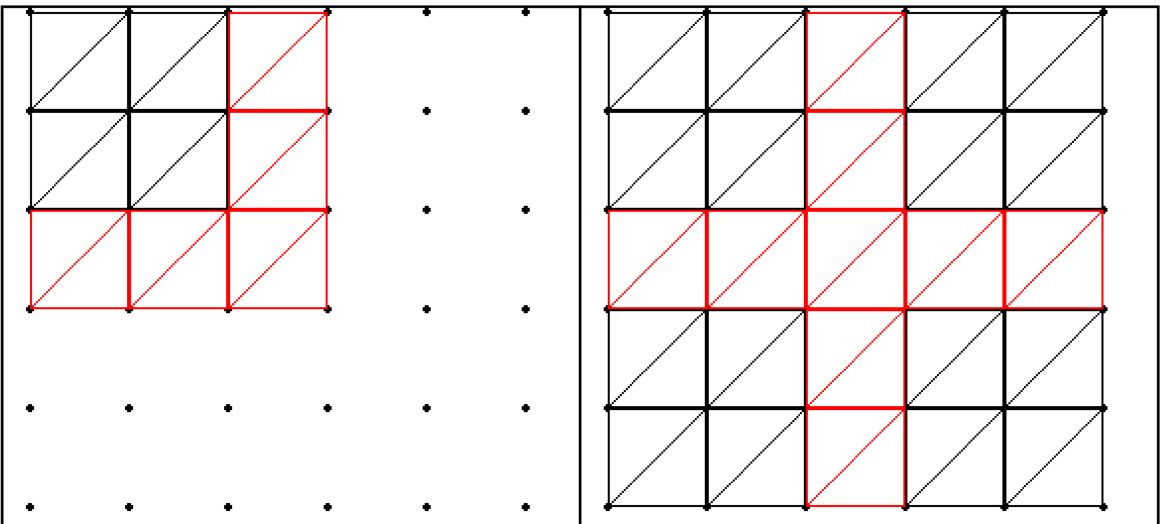
Vereinfachte Darstellung

Würde man nur die Punkte miteinander verbinden, die auch wirklich zu den jeweiligen Quad gehören, würde sich Folgendes ergeben:

Erster Quad	Zweiter Quad	4 Quads

Um diese Lücken zu verhindern, muss man eine Pixelreihe und –zeile, falls vorhanden, mehr auslesen und zu den Quad hinzufügen. Damit hat jeder Quad eine unmittelbare Verbindung zum Nächsten.

Beim 1. Quad die benachbarten Punkte mit hinzufügen	Endbild



Das Verarbeiten der Pixeldaten in Positionsdaten erfolgt wie beim Laden einer kleinen Heightmap.

Hinzukommt, dass man den grünen Kanal der Datamap in die erste Farbvariable vom Vertex schreibt.

Den blauen und transparenten Kanal der Map muss man wieder in die 4 Farben der Texturemap aufspalten und in die zweite Farbvariable des Vertex speichern. Wenn der blaue Wert der Map 0 beträgt, dann ist Rot der Texturemap 255 und Grün gleich 0. Bei den Wert 255 (für den blauen Kanal der Map), wird die entstehende Farbe einen roten Wert von 0 und einen grünen Wert von 255 annehmen. Dazwischen wird interpoliert. Den gleichen Vorgang wiederholt man, um den transparenten Kanal der Datamap in den Blau- und Alpha-Kanal der Texturemap aufzuteilen.

Die Texturkoordinaten werden errechnet, indem man die Position des Pixels der Map durch einen beliebigen Zahlenwert teil. Umso kleiner dieser Wert ist, umso mehr werden die Texturen gekachelt, d.h. sie wiederholen sich auf dem Terrain nach einen kleineren Distanz.

Die Erstellung der Indices erfolgt genauso, wie beim Laden einer kleinen Heightmap, die

33*33 Pixel groß ist. Man braucht die Indices nur einmal für einen Quad erstellen, da die

Punkte jedes Quads immer in der gleichen Reihenfolge vorliegen. Das Rendern fängt bei der Stelle an, die man als Startvertex angegeben hat, und zeichnet in der

Reihenfolge, die im Indexbuffer steht, die folgenden Punkte und verbindet sie automatisch zu Dreiecken.

Der Rendervorgang verläuft ähnlich wie beim Terrain einer kleiner Heightmap. Der Unterschied liegt in der Einbindung des QuadTrees, welcher die Quads rekursiv nach Sichtbarkeit testet. Sichtbare bzw. halbsichtbare, kleinste Quads werden zum Rendern an die Grafikkarte geschickt.

2.7. Multitexturig – Verwendung von Gras, Sand und Fels

Zu jeder Landschaft gehört eine ordentliche Texturierung, denn das Terrain kann noch so detailliert aussehen. Wenn die Texturen schlecht sind, ist auch der Gesamteindruck kein guter.

Zur Einbindung der verschiedenen Texturen gibt es viele Varianten, wobei sich die effektivste mit einen Pixelshader erstellen lässt, da man so auf die mordernste Technik der 3D-Grafik zurück greift.

Die meiner Texturierung wird ein Pixelshader 1.1. benutzt, da dieser von sehr vielen Grafikkarten unterstützt wird und ein emulierter Shader extrem langsam laufen würde. Jedoch ist man dadurch auf maximal vier verschiedene Texturen beschränkt.

Der Shader multipliziert lediglich die vier Texturen mit den einzelnen Farbkanälen der Texturemap und addiert die einzelnen Werte zusammen.

Da man mit dieser Shaderversion stark begrenzt ist und man nicht die einzelnen Werte eines Vektors ansprechen kann, muss man erst die einzelnen Farbwerte der Texturemap
herausfiltern, in alle Komponenten eines vierdimensionalen Vektors speichern und mit der entsprechenden Textur multiplizieren. Diesen Vorgang wiederholt man für alle Farbwerte der Texturemap und addiert die Ergebnisse.

Zum Endergebnis muss man die Lightmap hinzumultiplizieren, damit das Terrain beleuchtet wird.

Da die Kamera nur eine bestimmte Reichweite hat und alle Objekte außerhalb dieser nicht mehr gerendert werden, fügt man Nebel in der Ferne ein, damit das „Verschwinden“ der fernen Teile nicht so stark auffällt.

Textur 1	Textur 2	Textur 3	Textur 4

(Die Texturen stammen zu Testzwecken aus UT2003 - sie werden später durch freie Texturen ersetzt!)

Der Shader wurde teilweise in der High-Level-Shader-Language (HLSL) und in der Assembler Shader-Sprache geschrieben.

Shader für die Texturierung (.../Terrain/Shader/Multitexturing.fx):

```
// --- global var ---
float4x4 matWorldViewProj: WORLDVIEWPROJECTION;

texture Tex1; // < string Name = "gras1.jpg"; >;
texture Tex2; // < string Name = "gras2.jpg"; >;
texture Tex3; // < string Name = "gras3.jpg"; >;
texture Tex4; // < string Name = "gras4.jpg"; >;

float FNear = 500.0f;
float FFar = 750.0f;
float3 FColor = {0.5f, 0.3f, 1.0f};

// --- samplers ---
sampler SamplerRed = sampler_state
{
        Texture = <Tex1>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
};

sampler SamplerGreen = sampler_state
{
        Texture = <Tex2>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
};

sampler SamplerBlue = sampler_state
{
        Texture = <Tex3>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
};

// --- VS_OUTPUT structure ---
struct VS_OUTPUT
{
        float4 Pos : POSITION;
        float2 TexCoord0: TEXCOORD0;
        float2 TexCoord1: TEXCOORD1;
        float2 TexCoord2: TEXCOORD2;
        float2 TexCoord3: TEXCOORD3;
        float4 Dif0: COLOR0;
        float4 Dif1: COLOR1;
};

// --- RenderVS ---
VS_OUTPUT RenderVS(     float4 vPos : POSITION,
                        float4 vDif0: COLOR0,
                        float4 vDif1: COLOR1,
                        float4 vTexCoord0: TEXCOORD0 )
{
        VS_OUTPUT Out;

        Out.Pos = mul( vPos, matWorldViewProj );
        Out.TexCoord0 = vTexCoord0;
```

```
	Out.TexCoord1 = vTexCoord0;
	Out.TexCoord2 = vTexCoord0;
	Out.TexCoord3 = vTexCoord0;
	Out.Dif0 = vDif0;
	Out.Dif1 = vDif1;

	float F = (1/(FFar-FNear)) * (Out.Pos.w-FNear);
	Out.Dif0.a = clamp( F, 0.0f, 1.0f );

	return Out;
}

// --- technique Render ---
technique Render
{
	pass P0
	{
		Lighting = FALSE;
		CullMode = CW;
		ZEnable = TRUE;
		ZWriteEnable = TRUE;

		Texture[0] = <Tex1>;
		Texture[1] = <Tex2>;
		Texture[2] = <Tex3>;
		Texture[3] = <Tex4>;

		MinFilter[0] = Anisotropic;
		MagFilter[0] = Anisotropic;
		MipFilter[0] = Linear;

		MinFilter[1] = Anisotropic;
		MagFilter[1] = Anisotropic;
		MipFilter[1] = Linear;

		MinFilter[2] = Anisotropic;
		MagFilter[2] = Anisotropic;
		MipFilter[2] = Linear;

		MinFilter[3] = Anisotropic;
		MagFilter[3] = Anisotropic;
		MipFilter[3] = Linear;

		VertexShader = compile vs_1_1 RenderVS();
		PixelShader =
		asm
		{
			ps.1.1

			def c0, 1, 0, 0, 0
			def c1, 0, 1, 0, 0
			def c2, 0, 0, 1, 0

			tex t0
			tex t1
			tex t2
			tex t3

			dp3 r1, c0, v1
			mul r0, r1, t0
			dp3 r1, c1, v1
			mad r0, r1, t1, r0
			mov r1.a, v1.b
			mad r0, r1.a, t2, r0
			mad r0, v1.a, t3, r0
			mad r0, r0, v0, v0.a
		};
	}
}
```

Das Terrain sieht jetzt schon so aus:

2.8. Das Gewässer

Die Landschaft sieht bis jetzt etwas eintönig aus, da sie nur aus einer Aneinanderreihung von Bergen und Tälern besteht. Um etwas Leben in das triste Bild zu bringen, fügt man ein kleines Gewässer ein, welches die Monotonie des Terrains unterbricht.

2.8.1 Änderung an der Heightmap

Das Gewässer kann man nicht sofort in die Landschaft einfügen, denn es bedarf noch einer kleinen Änderung in der Heightmap, die sich in der Datamap befindet. Man muss eine geeignete Stelle, z.B. ein Loch, für das Wasser in die Heightmap hineinzeichnen.

Vorerst erstellt man eine mit Weiß gefüllte Bitmap mit der gleichen Größe der Heightmap. Im Zentrum dieses Bildes fügt man ein beliebig großen, schwarzen Kreis ein, dessen Kanten man stark weich zeichnet, damit eine sanfter Übergang von der schwarzen zur weißen Farbe vorliegt.

Alle Pixel der Heightmap, welche einen Wert unter der gewünschten Wasserhöhe haben, muss man auf diesen Wert setzen, damit später die Wasserfläche an allen Seiten am Terrain abschließt.

Nun geht man alle Pixel dieser Bitmap und der Heightmap parallel durch und dividiert die Farbwerte an den identischen Positionen durch 255, um jeweils einen Wert zwischen 0 und 1 zu erhalten. Diese beiden Werte multipliziert man miteinander und noch einmal mit 255, damit man wieder auf einen Wert zwischen 0 und 255 kommt. Dieses Ergebnis speichert man in die Heightmap.

Jetzt sind die Höheninformationen so geändert, dass sich in der Mitte des Terrains ein geeignetes Loch für das Wasser befindet:

Heightmap	Wasser-Bitmap	Resultat

2.8.2.1. Die Reflectionmaps

Reflectionmaps sind Texturen, die eine gespiegeltes Bild beinhalten. Diese Texturen rendert man separat für jede spiegelnde Fläche.

Da die Wasserfläche nur auf der X-Z-Ebene liegt, kann man den Algorithmus vereinfachen.

Zuerst erstellt man eine virtuelle Kamera, die sozusagen das Auge der spiegelnden Fläche ist, weil sie das erfasst, was gespiegelt wird.

In unseren Fall muss man lediglich die Y-Position der echten Kamera an der Fläche spiegeln, indem man zu den negativen Y-Wert die Höhenposition der Fläche addiert.

Den Richtungsvektor der Kamera muss man ebenfalls an der Fläche spiegeln, indem man den Y-Wert mit –1 multipliziert.

Danach muss man nur noch den Up-Vektor der Kamera umkehren und kann mit den Vektoren die virtuelle Kamera erstellen.

Wenn die Kamera gesetzt wurde, muss man das RenderTarget auf die Textur setzen, welche die Spiegelinformationen enthalten soll.

Jetzt rendert man alle Objekte, die sich gespiegelt werden sollen, was in unserem Fall das

Terrain und die SkyBox wären.

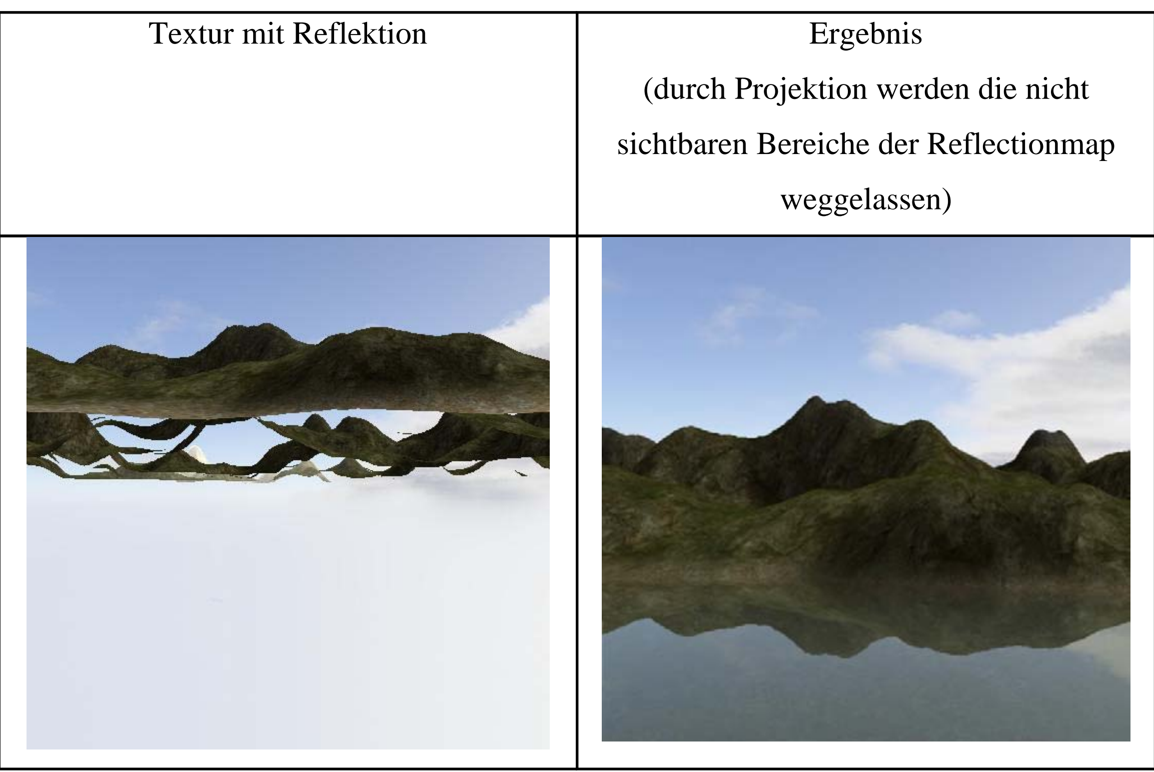
2.8.2.2. Das Rendern der Reflectionmap

Zuerst lässt man das Terrain mit der echten Kamera rendern, danach zeichnet man die Reflectionmap.

Zuletzt kommt die Wasserfläche. Ihr gibt man die Textur der Reflectionmap und errechnet mit einen Vertexshader die Texturkoordinaten. Diese müssen nach der Kameraposition gedreht werden. Weiterhin muss man die Textur auf die Fläche projizieren, damit nur die Stellen zu sehen sind, die auch tatsächlich dem Spiegelbild entsprechen. Würde man die Projektion nicht hinzufügen, wäre die Textur mehrfach auf der Fläche gekachelt und würde das Bild verfälschen.

Um das Wasser natürlich aussehen zu lassen, muss man Transparenz hinzufügen, denn diese ist bei jeden echten Wasser vorhanden. Stellen, die sich nahe der Kamera befinden, bekommen einen hohen Transparenzwert, wobei Stellen, welche ferner von der Kamera sind, eine niedrigere Transparenz erhalten.

Außerdem ist ein Wasser oft trübe, deshalb fügt man ein paar farbige, transparente Flächen unter der Wasseroberfläche ein, um eine gewisse Trübe zu erreichen.



Quellcode zu den Reflectionmaps (...Terrain/cReflectionMap.pas):

```
type
  TReflectionMap = class
  private

   ReflectSur: IDirect3DSurface9;
   ReflectEff: ID3DXEffect;
   planeMesh : ID3DXMesh;
   savedSur  : IDirect3DSurface9;
   matsaveV  : TD3DXMatrix;
   HasSky    : Boolean;
   height    : Single;

  public

   ReflectTex: IDirect3DTexture9;
   matView : TD3DXMatrix;
   matWorld : TD3DXMatrix;

   constructor Create( size : WORD; FileOfMesh: PChar; meshHeight: Single );

   procedure EnableSkyBox( Enable: Boolean );
   procedure BeginScene;
   procedure EndScene;
   procedure RenderPlane;

  end;

implementation

constructor TReflectionMap.Create( size : WORD;
                                   FileOfMesh: PChar;
                                   meshHeight: Single );
begin

 inherited Create;

 // leere Textur
 D3DXCreateTexture( Device, size, size, 0, D3DUSAGE_RENDERTARGET, D3DFMT_R8G8B8,
                    D3DPOOL_DEFAULT, ReflectTex );

 // Surface für Textur
 ReflectTex.GetSurfaceLevel( 0, ReflectSur );

 // shader laden
 D3DXCreateEffectFromFile( Device, 'Shader/Reflection.fx', nil, nil, 0, nil,
                           ReflectEff, nil );
 ReflectEff.SetTechnique( 'Render' );

 // Mesh laden
 D3DXLoadMeshFromX( FileOfMesh, 0, Device, nil, nil, nil, nil, planeMesh );

 // Vars
 HasSky := FALSE;
 Height := meshHeight;
 D3DXMatrixTranslation( matWorld, 0, Height, 0 );
 ReflectEff.SetMatrix( 'mWorld', matWorld );

end;

procedure TReflectionMap.EnableSkyBox( Enable: Boolean );
begin
 HasSky := Enable;
end;

procedure TReflectionMap.BeginScene;
var
  plane : TD3DXPlane;
  vEye : TD3DXVector3;
  vUp : TD3DXVector3;
  vRight : TD3DXVector3;
  vLook : TD3DXVector3;
  mWVP : TD3DXMatrix;
```

```
begin

 Device.GetRenderTarget( 0, savedSur );
 Device.SetRenderTarget( 0, ReflectSur );
 Device.Clear( 0, nil, D3DCLEAR_TARGET or D3DCLEAR_ZBUFFER, $ff550000, 1, 0 );

 // ---CAMERA------------------------------------------------------------------
 // matView sichern
 matsaveV := Engine.Camera.matView;

 // Daten
 vEye := Engine.Camera.Eye;
 vLook := Engine.Camera.Look;

 // Reflect-View berechnen
 vEye.y := -vEye.y + 2.01*Height; // vEye reflect
 vLook.y := -vLook.y; // vLook reflect
 D3DXVec3Cross( vRight, vLook, D3DXVector3(0,-1,0) ); // new vUp
 D3DXVec3Cross( vUp, vLook, vRight );
 D3DXVec3Add( vLook, vLook, vEye ); // vLookAt

 // SetView
 D3DXMatrixLookAtLH( matView, vEye, vLook, vUp );
 Device.SetTransform( D3DTS_VIEW, matView );

 // ---RENDER------------------------------------------------------------------
 // Render SkyBox
 if HasSky = TRUE then Engine.SkyBox.Render( vEye );

 // ClipPlane - bei SkyBox sollte sie ausgeschalten bleiben
 D3DXPlaneFromPointNormal( plane,
 D3DXVector3( 0, Height-0.001, 0),
 D3DXVector3( 0, -1, 0) );

 D3DXMatrixMultiply( mWVP, matView, Engine.Camera.matProj );
 D3DXMatrixTranspose( mWVP, mWVP );
 D3DXMatrixInverse( mWVP, nil, mWVP );
 D3DXPlaneTransform( plane, plane, mWVP );

 Device.SetClipPlane( 0, P4SingleArray(@plane) );
 Device.SetRenderState( D3DRS_CLIPPLANEENABLE, D3DCLIPPLANE0 );
 Device.SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );

end;

procedure TReflectionMap.EndScene;
begin

 Device.SetRenderState( D3DRS_CLIPPLANEENABLE, 0 );
 Device.SetRenderTarget( 0, savedSur );
 Device.SetTransform( D3DTS_VIEW, matsaveV );
end;

procedure TReflectionMap.RenderPlane;
var
  matTrans: TD3DXMatrix;
  matScale: TD3DXMatrix;
  matView : TD3DXMatrix;
  matProj : TD3DXMatrix;
  matTex : TD3DXMatrix;
  passes : Cardinal;

begin

 // ---EINSTELLUNGEN-----------------------------------------------------------
 Device.SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT4 or
                              D3DTTFF_PROJECTED );
 Device.SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_CAMERASPACEPOSITION
);

 D3DXMatrixTranslation( matTrans, 0.5, 0.5, 0.0 );
 D3DXMatrixScaling( matScale, 0.5, 0.5, -1.0 );

 matView := Engine.Camera.matView;
 matProj := Engine.Camera.matProj;
```

```
 D3DXMatrixMultiply( matProj, matProj, matScale );
 D3DXMatrixMultiply( matTex , matProj, matTrans );

 //Device.SetTransform( D3DTS_TEXTURE0, matTex );
 ReflectEff.SetMatrix( 'mTex', matTex );
 ReflectEff.SetMatrix( 'mView', Engine.Camera.matView );
 ReflectEff.SetMatrix( 'mProj', Engine.Camera.matProj );
 ReflectEff.SetTexture( 'ReflectTex', ReflectTex );
 ReflectEff.SetFloat( 'fTime', GetTickCount/1000 );

 // ---RENDERN-----------------------------------------------------------------
 ReflectEff._Begin( passes, 0 );
 ReflectEff.Pass(0);
 planeMesh.DrawSubset(0);
 ReflectEff._End;

 // ---ENDE--------------------------------------------------------------------
 Device.SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_DISABLE );
 Device.SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_PASSTHRU );

end;

end.
```

Quellcode zu der Trübe (...Terrain/cGloomy.pas):

```
type
  TGloomy = class
  private
   VB : IDirect3DVertexBuffer9;
   iFa: Cardinal;

  public
   eff: ID3DXEffect;

   constructor Create( minHeight, maxHeight: Single; iPlanes: WORD );

   procedure Render;
  end;

implementation

constructor TGloomy.Create( minHeight, maxHeight: Single; iPlanes: WORD );
var
  i : WORD;
  iVe : Cardinal;
  py : Single;
  Vert: array of TPosVertex;
  P : Pointer;
begin

 inherited Create;
 setlength( Vert, iPlanes*6 );
 iVe := 0;
 iFa := 0;

 for i := 1 to iPlanes do
 begin

  py := minHeight + ( ((maxHeight-minHeight)/iPlanes) * i );
  Vert[iVe+0].pos := D3DXVector3( -1, py, 1 );
  Vert[iVe+1].pos := D3DXVector3( 1, py, -1 );
  Vert[iVe+2].pos := D3DXVector3( -1, py, -1 );
  Vert[iVe+3].pos := D3DXVector3( -1, py, 1 );
  Vert[iVe+4].pos := D3DXVector3( 1, py, 1 );
  Vert[iVe+5].pos := D3DXVector3( 1, py, -1 );
  inc( iVe, 6 );
  inc( iFa, 2 );
 end;

 Device.CreateVertexBuffer( iVe*sizeof(Vert[0]), D3DUSAGE_WRITEONLY, D3DFVF_XYZ,
                            D3DPOOL_DEFAULT, VB, nil );
```

```
 VB.Lock( 0, iVe*sizeof(Vert[0]), P, 0 );
  move( Vert[0], P^, iVe*sizeof(Vert[0]) );
 VB.Unlock;

D3DXCreateEffectFromFile( Device, 'Shader/gloomy.fx',
                           nil, nil, 0, nil, eff, nil );
 eff.SetTechnique( 'Render' );
 eff.SetVector( 'Color', D3DXVector4( 0.25, 0.3, 0.25, 0.1 ) );
 eff.SetFloat( 'HalfWidth', 1.0 );

end;

procedure TGloomy.Render;
var
  passes: Cardinal;
begin

 Device.SetStreamSource( 0, VB, 0, sizeof(TPosVertex) );
 Device.SetFVF( D3DFVF_XYZ );
 Device.SetTexture( 0, nil );

eff.SetMatrix( 'mView', Engine.Camera.matView );
 eff.SetMatrix( 'mProj', Engine.Camera.matProj );
 eff.SetMatrix( 'mWorld', D3DXMatrixIdentity );

 eff._Begin( passes, 0 );
 eff.Pass( 0 );
  Device.DrawPrimitive( D3DPT_TRIANGLELIST, 0, iFa );
 eff._End;

end;

end.
```

Shader für die Trübe (...Terrain/Shader/Gloomy.fx):

```
// --- global var ---
float4x4 mWorld: WORLD;
float4x4 mView : VIEW;
float4x4 mProj : PROJECTION;
float4x4 mWorldView : WORLDVIEW;
float4x4 mWorldViewProj: WORLDVIEWPROJECTION;
float4 Color = { 0.1f, 0.5f, 0.2f, 0.1f };
float HalfWidth = 1.0f;
float FNear = 500.0f;
float FFar = 750.0f;

void Matrix()
{
        mWorldView = mul( mWorld, mView );
        mWorldViewProj = mul( mWorldView, mProj );
};

// --- VS_OUTPUT structure ---
struct VS_OUTPUT
{
        float4 Position: POSITION;
        float4 Diffuse : COLOR0;
};

// --- VS_OUTPUT RenderVS ---
VS_OUTPUT RenderVS( float3 vPos: POSITION )
{
        VS_OUTPUT Out;
        Matrix();

        // -------------------------------------------------------------
        // --- Position ------------------------------------------------
        vPos.xz *= HalfWidth;
        Out.Position = mul( float4(vPos, 1), mWorldViewProj );

        // -------------------------------------------------------------
```

```
        // --- Diffuse + Fog-------------------------------------------
        float F = (1/(FFar-FNear)) * (Out.Position.w-FNear);
        F = clamp( F, 0.0f, 1.0f );
        Out.Diffuse.rgb = Color.rgb + F;
        Out.Diffuse.a = Color.a;

        // ---------------------------------------------------------------
        return Out;
}

struct PS_OUTPUT
{
        float4 RGB: COLOR0;
};

PS_OUTPUT RenderPS( float4 Dif: COLOR0 )
{
        PS_OUTPUT Out;
        // ---------------------------------------------------------------
        Out.RGB = Dif;
        // ---------------------------------------------------------------

        return Out;
};

// --- technique Render ---
technique Render
{
        pass P0
        {
                AlphaBlendEnable = TRUE;
                SrcBlend = SrcAlpha;
                DestBlend = InvSrcAlpha;
                CullMode = CCW;

                VertexShader = compile vs_1_1 RenderVS();
                PixelShader = compile ps_1_1 RenderPS();
        }
}
```

2.9. gesamter Quellcode „cTerrain“

Diese Unit beinhaltet die ganze Initalisierung des Terrains

```
unit cTerrain;

interface

uses
  Windows, Graphics, d3d9, d3dx9, DXEngine, DXEngineVars, DXEngineDInput,
  cQuadTree, cVFD, cReflectionMap, cGloomy;

type
  // Terrain-Klasse
  TTerrain = class
  private

   VertexBuffer: IDirect3DVertexBuffer9; // VertexBuffer
   IndexBuffer : IDirect3DIndexBuffer9; // IndexBuffer
   iVertices : Cardinal; // Anzahl Vertices
   iIndices : Cardinal; // Anzahl Indices

   iWait : WORD; // nur für TastenAbfrage <> speed

   effect : ID3DXEffect;
   watermesh : ID3DXMesh;
   gloomy : TGloomy;

   Tex1 : IDirect3DTexture9; // Gras
   Tex2 : IDirect3DTexture9; // Hang - Dreck bis Fels
   Tex3 : IDirect3DTexture9; // Sand
   Tex4 : IDirect3DTexture9; // Fels

   map : TBitmap; // Datamap

   QuadTree : TQuadTree; // QuadTree
   iQuads : Cardinal;

   procedure CreateVertexBuffer;

   procedure CreateIndexBuffer;

   procedure LoadShaderAndTexes;

  public

   Reflect : TReflectionMap;
   constructor Create( fileName: PChar ); // constructor der Klasse

   procedure Render( RenderReflection: Boolean ); // Render-Prozedur

   procedure RenderEnvMap;

  end;

implementation

constructor TTerrain.Create( fileName: PChar );
begin

 inherited Create;

 // Laden der Heightmap
 map := TBitmap.Create;
 map.PixelFormat := pf32Bit;
 map.LoadFromFile( fileName );

 // QuadTree erstellen - noch nicht füllen
 QuadTree := TQuadTree.Create;

 // VertexBuffer
```

```
 CreateVertexBuffer;

 // IndexBuffer
 CreateIndexBuffer;

 // Shader und Texturen
 LoadShaderAndTexes;

 // RenderQuadCubes
 QuadTree.GenerateQuadTree;

 // WasserFläche
 D3DXLoadMeshFromX( 'half.x', 0, Device, nil, nil, nil, nil, watermesh );

 // Gloomy
 gloomy := TGloomy.Create( 0.0, 4.5, 20 );
 gloomy.eff.SetFloat( 'HalfWidth', 128.0 );

end;

procedure TTerrain.RenderEnvMap;
begin

 // ReflektionWasser
 Reflect.BeginScene;
 self.Render( TRUE );
 Reflect.EndScene;

end;

procedure TTerrain.Render( RenderReflection: Boolean );
var
  i : WORD;
  passes: Cardinal;
  matWVP: TD3DXMatrix;
begin

 // -- Terrain rendern
 Device.SetTransform( D3DTS_WORLD, D3DXMatrixIdentity );
 Device.SetStreamSource( 0, VertexBuffer, 0, sizeof(TPosDif2Tex1Vertex) );
 Device.SetIndices( IndexBuffer );
 Device.SetFVF( D3DFVF_POSDIF2TEX1 );
 Device.SetTexture( 0, nil );

 // Viewing Frustum Detection
 UpdateVFDPlanes;
 QuadTree.Quads.VFD( @QuadTree, 1,
 TestVFD( @QuadTree.Quads.min, @QuadTree.Quads.max ) );

 effect._Begin( passes, 0 );
 effect.Pass( 0 );

 // Shader - Daten senden
 if RenderReflection = FALSE then
 begin

  D3DXMatrixMultiply( matWVP, Engine.Camera.matView, Engine.Camera.matProj );
  effect.SetMatrix( 'matWorldViewProj', matWVP );

 end else
 begin

  D3DXMatrixMultiply( matWVP, Reflect.matView, Engine.Camera.matProj );
  effect.SetMatrix( 'matWorldViewProj', matWVP );

 end;

 // Rendern der sichtbaren Quads
 for i := 0 to iQuads-1 do
 begin

  if QuadTree.RenderQuads[i].visible = TRUE then
  begin

   Device.DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                QuadTree.RenderQuads[i].StartVertex,
```

```
                                0, 33*33, 0, 32*32*2 );

  end;

 end;

 effect._End;

 // -- WasserFläche rendern
 if RenderReflection = FALSE then
 begin

  gloomy.Render;
  Reflect.RenderPlane;

 end;

 // -- zusätzliche Tastenabfragen
 if iWait > 0 then inc( iWait );
 if iWait >= 100 then iWait := 0;
 if iWait = 0 then
 begin

  if Engine.DInput.Key[DIK_F2] then // RenderQuadTrees zeigen
  begin

   iWait := 1;
   //if QuadCubes.DoRender = FALSE
   //then QuadCubes.DoRender := TRUE
   //else QuadCubes.DoRender := FALSE;

  end;

  if Engine.DInput.Key[DIK_R] then // ReflectionMap speichern
  begin

   iWait := 1;
   D3DXSaveTextureToFile( 'Reflect.jpg', D3DXIFF_JPG, Reflect.ReflectTex, nil);

  end;

 end;

end;

procedure TTerrain.CreateVertexBuffer;
type
  Color = record // Typ für Farbverwaltung der Heightmap
  blue : Byte;
  green: Byte;
  red : Byte;
  alpha: Byte;
end;

const
  factor: WORD = 2; // StretchFactor des Terrains
  height: WORD = 64; // maximale Höhe des Terrains

var
  Vert : array of TPosDif2Tex1Vertex; // array für alle Vertices
  MoveX: WORD; // Verschiebung auf X-Achse
  MoveZ: WORD; // Verschiebung auf Y-Achse
  ix,iy: WORD; // SchleifenVariable
  px,py: WORD; // SchleifenVariable
  i : WORD; // SchleifenVariable
  pxl : PCardinal; // Pointer auf HeightmapPixel
  col : Color; // Farbe vom HeightmapPixel
  dif1 : Color; // Farbe vom Vertex
  newQuad: Boolean; // Beginn eines neuen Quads?
  mal : Single; // Multiplikator
  P : Pointer; // Pointer für VertexBuffer
begin

 // VarInit
 mal := 0;
 pxl := nil;
```

```
// ArrayLängen festlegen
setlength( Vert, (map.Width div 32)*(map.Height div 32) * (33*33) + 6 );

// Verschiebung jedes Vertices auf X|Z Achse
MoveX := (map.Width div 2) * factor;
MoveZ := (map.Height div 2) * factor;

// Anzahl Vertices auf 0 zurück setzen
iVertices := 0;

// Schleife durch alle Quads
for iy := 0 to (map.Height div 32)-1 do
begin

 for ix := 0 to (map.Width div 32)-1 do
 begin

  newQuad := TRUE; // neuer Quad beginng

   // Schleife innerhalb eines Quads durch alle Vertices
   for py := 0 to 32 do
   begin

    // pointer auf HeightmapPixels holen
    if (iy <> (map.Height div 32)-1) or (py <> 32) then
    begin

     pxl := map.ScanLine[(iy*32)+py];

     for i := 1 to ix*32 do inc( pxl );
    end;

    for px := 0 to 32 do
    begin

     // HeightmapPixel - Farbe ermitteln
     col := Color(pxl^);
     inc( pxl );

     // Randfehler beheben
     if ((ix = (map.Width div 32)-1) and (px = 32)) or
        ((iy = (map.Height div 32)-1) and (py = 32)) then col.red := 0;

     // Position des Vertex
     Vert[iVertices].pos := D3DXVector3( ix*32*factor + px*factor - MoveX,
                                         (col.red/255) * height,
                                         iy*32*factor + py*factor - MoveZ );

     // Farbe Vertex = BeleuchtungsFarbe
     Vert[iVertices].dif0 := D3DColor_RGBA(col.green, col.green, col.green, 0);

     // 2.Farbe = TexturSelektion
     dif1.red := 0;
     dif1.green := 0;
     dif1.blue := 0;
     dif1.blue := 0;

     if col.alpha < 127.5 then mal := col.alpha / (255/2);
     if col.alpha > 127.5 then mal := 1 - (col.alpha-(255/2)) / (255/2);

     dif1.red := trunc( (255 - col.blue) * mal );
     dif1.green := trunc( (col.blue) * mal );
     dif1.blue := trunc((255 - col.alpha) * (1-mal) );
     dif1.alpha := trunc( col.alpha * (1-mal) );

     if col.alpha < 127.5 then dif1.alpha := 0;
     Vert[iVertices].dif1 := D3DCOLOR_RGBA( dif1.red,
                                            dif1.green,
                                            dif1.blue,
                                            dif1.alpha );

     // TexturKoordinaten
     Vert[iVertices].u := px / 16;
     Vert[iVertices].v := py / 16;
```

```
     // --- QuadTree
     QuadTree.AddToQuad( newQuad, iVertices, Vert[iVertices].pos );
     if newQuad then newQuad := FALSE; ;

     // Anzahl Vertices + 1
     inc( iVertices );

    end; // for px
   end; // for py

   inc( iQuads );

  end; // for ix
 end; // for iy

 // WasserVertices hinzufügen
 Vert[iVertices+0].pos := D3DXVector3( (map.Width/2)*factor - 64*factor - MoveX,
                                       (25/255)*height,
                                       (map.Height/2)*factor - 64*factor - MoveZ );
 Vert[iVertices+1].pos := D3DXVector3( (map.Width/2)*factor - 64*factor - MoveX,
                                       (25/255)*height,
                                       (map.Height/2)*factor + 64*factor - MoveZ );
 Vert[iVertices+2].pos := D3DXVector3( (map.Width/2)*factor + 64*factor - MoveX,
                                       (25/255)*height,
                                       (map.Height/2)*factor + 64*factor - MoveZ );
 Vert[iVertices+3].pos := D3DXVector3( (map.Width/2)*factor - 64*factor - MoveX,
                                       (25/255)*height,
                                       (map.Height/2)*factor - 64*factor - MoveZ );
 Vert[iVertices+4].pos := D3DXVector3( (map.Width/2)*factor + 64*factor - MoveX,
                                       (25/255)*height,
                                       (map.Height/2)*factor + 64*factor - MoveZ );
 Vert[iVertices+5].pos := D3DXVector3( (map.Width/2)*factor + 64*factor - MoveX,
                                       (25/255)*height,
                                       (map.Height/2)*factor - 64*factor - MoveZ );

 inc( iVertices, 6 );

 // VertexBuffer erstellen + schreiben
 Device.CreateVertexBuffer( iVertices*sizeof(TPosDif2Tex1Vertex), D3DUSAGE_WRITEONLY,
 D3DFVF_POSDIF2TEX1, D3DPOOL_MANAGED, VertexBuffer, nil );

 VertexBuffer.Lock( 0, iVertices*sizeof(TPosDif2Tex1Vertex), P, 0 );
  move( Vert[0], P^, iVertices*sizeof(TPosDif2Tex1Vertex) );
 VertexBuffer.Unlock;

end;

procedure TTerrain.CreateIndexBuffer;
var
 Indi : array of WORD; // array für alle Indices
 ix,iy: WORD; // SchleifenVariable
 P : Pointer; // Pointer für IndexBuffer
begin

 // ArrayLängen festlegen
 setlength( Indi, 32*32*6 );
 // Anzahl Indices auf 0 zurück setzen
 iIndices := 0;
 // Indices errechnen

 for iy := 0 to 31 do
 begin

  for ix := 0 to 31 do
  begin

   Indi[iIndices+0] := (iy*33) + ix;
   Indi[iIndices+1] := (iy*33) + ix + 1;
   Indi[iIndices+2] := ((iy+1)*33) + ix + 1;
   Indi[iIndices+3] := (iy*33) + ix;
   Indi[iIndices+4] := ((iy+1)*33) + ix + 1;
   Indi[iIndices+5] := ((iy+1)*33) + ix;
   inc( iIndices, 6 );
  end;

 end;
```

```
 // Indexbuffer erstellen und schreiben
 Device.CreateIndexBuffer( iIndices*sizeof(Indi[0]), D3DUSAGE_WRITEONLY,
 D3DFMT_INDEX16, D3DPOOL_MANAGED, IndexBuffer, nil );

 IndexBuffer.Lock( 0, iIndices*sizeof(Indi[0]), P, 0 );
  move( Indi[0], P^, iIndices*sizeof(Indi[0]) );
 IndexBuffer.Unlock;
end;

procedure TTerrain.LoadShaderAndTexes;
begin

 // --- Shader laden
 if FAILED( D3DXCreateEffectFromFile( Device, 'Shader\MultiTexturing.fx',
 nil, nil, 0, nil, effect, nil ) ) then
 begin
  ErrorMessage( 'can''t load MultiTexturing-shader', TRUE );
 end;

 if FAILED(effect.ValidateTechnique( 'Render' )) then
 begin
  ErrorMessage( 'can''t load technique', TRUE );
 end;

 effect.SetTechnique( 'Render' );

 // --- wasser ---
 Reflect := TReflectionMap.Create( 512, 'half.x', 5 );

 // --- Texturen laden
 D3DXCreateTextureFromFileEx( Device, 'Textures\tex1.jpg', 512, 512, 0,
                              D3DUSAGE_DYNAMIC, D3DFMT_R5G6B5, D3DPOOL_DEFAULT,
                              D3DTEXF_ANISOTROPIC, D3DTEXF_LINEAR, $ff000000,
                              nil, nil, Tex1 );
 D3DXCreateTextureFromFileEx( Device, 'Textures\tex2.jpg', 512, 512, 0,
                              D3DUSAGE_DYNAMIC, D3DFMT_R5G6B5, D3DPOOL_DEFAULT,
                              D3DTEXF_ANISOTROPIC, D3DTEXF_LINEAR, $ff000000,
                              nil, nil, Tex2 );
 D3DXCreateTextureFromFileEx( Device, 'Textures\tex3.jpg', 512, 512, 0,
                              D3DUSAGE_DYNAMIC, D3DFMT_R5G6B5, D3DPOOL_DEFAULT,
                              D3DTEXF_ANISOTROPIC, D3DTEXF_LINEAR, $ff000000,
                              nil, nil, Tex3 );
 D3DXCreateTextureFromFileEx( Device, 'Textures\tex4.jpg', 512, 512, 0,
                              D3DUSAGE_DYNAMIC, D3DFMT_R5G6B5, D3DPOOL_DEFAULT,
                              D3DTEXF_ANISOTROPIC, D3DTEXF_LINEAR, $ff000000,
                              nil, nil, Tex4 );

 // Texturen Shader übergeben
 effect.SetTexture( 'Tex1', Tex1 );
 effect.SetTexture( 'Tex2', Tex2 );
 effect.SetTexture( 'Tex3', Tex3 );
 effect.SetTexture( 'Tex4', Tex4 );
 effect.SetFloat( 'HalfWidth', 128.0 )
end;

end.
```

2.10. Resultate

Das Ergebnis ist eine „schöne“ Landschaft:

2.11 Abschließende Informationen

Auf CD-ROM befindet sich mein Framework, was jediglich das Device erstellt und ein paar nützliche Funktionen und Typen enthält.

Die Kamerasteuerung funktioniert mit W-A-S-D.

In „Perlin Noise“ befindet sich das komplette Programm zur Landschaftserstellung.

In „DXEngine/Anwendungen/Terrain“ befindet sich die komplette Anwendung zum Laden und Darstellen des Terrains.

Um ein fehlerfreies Rendern zu gewährleisten, braucht man mindestens Shaderversion 1.1.

Die Programme liefen auf einen AthlonXP 2400+ mit 2GHz und einer GeForce Ti4600 ruckel- und fehlerfrei.

2.12. Quellen

Das Microsoft Software-Development-Kit (SDK) von
www.microsoft.com/DirecX
Die Foren www.DelphiDX.de und www.ZFX.info
Beratungen von Personen aus ICQ (Coolcat, Holly, Portstorm und weitere)
Die Texturen stammen zu Testzwecken aus UT2003 und werden später durch freie Texturen ersetzt.
Den Rest habe ich selbst entwickelt, da ich Landschaftserstellung innerhalb eines halben Jahres verinnerlicht habe

3. Nachwort

Ich hoffe ich konnte eine gute und ausführliche Hilfestellung für Neulinge schreiben, die ein Terrain erstellen möchten. Des weiteren, dass keine inhaltlichen Fehler aufgetreten sind, die ich einmal falsch gelernt habe.
Der Abschnitt zu QuadTrees ist etwas kurz ausgefallen. Hier werde ich später noch einmal ein Kapitel einfügen.

Vielen Dank,
artzuk