Computergrafik Praktikum 5

Allgemeines zum Praktikum

Im Rahmen des Computergrafik-Praktikums lernen Sie, die aus der Vorlesung gewonnenen Erkenntnisse praktisch mit der Programmiersprache C/C++ umzusetzen.

Bei der Konzipierung der Praktika wurde darauf geachtet, dass die verwendeten Bibliotheken sowohl für Windows als auch für OS X erhältlich sind. Sofern Sie die Aufgaben im Medienlabor bearbeiten, steht Ihnen die Entwicklungsumgebung XCode zur Verfügung, mit der Sie Ihre C++-Programme entwickeln können. Wenn Sie die Aufgaben an Ihrem privaten Rechner unter Windows bearbeiten möchten, empfehle ich Ihnen die Entwicklungsumgebung Visual Studio 2013, die Sie als Student/-in über das Microsoft ELMS-Programm kostenlos herunterladen können.

Bitte beachten Sie bei der Bearbeitung der Praktikumsaufgaben, dass Ihre Lösungen für folgende Praktikumsaufgaben weiter genutzt werden müssen. Die Praktikumsarbeiten bauen also ineinander auf. Bearbeiten Sie deshalb die Lösungen sorgfältig und löschen Sie diese nicht, wenn die Aufgabe abgenommen wurde. Achten Sie bei der Entwicklung darauf, dass nur die Schnittstellen in den Header-Dateien stehen, während die Implementierung ausschließlich in den cpp-Dateien steht.

Zum Bestehen des kompletten Praktikums müssen Sie jede Praktikumsaufgabe erfolgreich abschließen und mindestens 80% der Gesamtpunkte für das komplette Praktikum erhalten. Wie viele Punkte Sie für eine einzelne Aufgabe erhalten, hängt von Ihrer Implementierung und Ihren Erläuterungen ab.

Um dieses Praktikum zu bestehen, müssen Ihre entwickelten Teillösungen lauffähig sein (5 Punkte) und Sie müssen die einzelnen Lösungen erläutern können (weitere 5 Punkte). Insgesamt können Sie bei diesem Aufgabenblatt 10 Punkte + 7 Zusatzpunkte erhalten.

Abnahmetermin: 07.06. (Di) & 10.06 (Fr).

Thema Praktikum 5

Bei diesen Praktikumsaufgaben geht es darum, dass Sie die folgenden Konzepte praktisch kennenlernen bzw. benutzen:

- Das Erzeugen und Darstellen von prozeduralen Terrain-Modellen mit Hilfe von Height-Maps.
- Die Mittelung von Normalen zur Glättung von Schattierungsverläufen der Objektoberflächen.
- Die CPU-seitige Implementierung einer einfachen linearen Filtermethode zur Kantenhervorhebung.
- Den Umgang mit mehreren UV-Koordinaten und Texturen innerhalb eines Modells sowie das Mischen mehrerer Texturebenen.

- Das Erweitern und Verändern von Shader-Programmen, um spezifische Effekte zu erzielen.

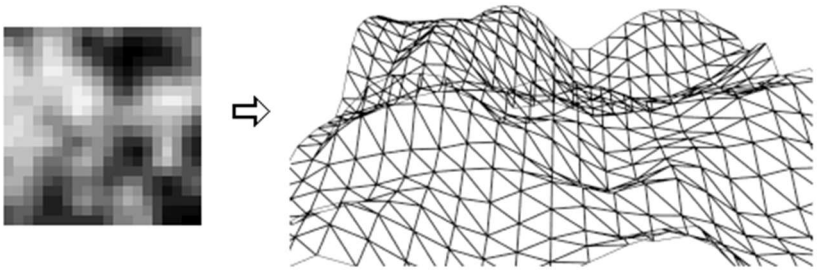
Für die Bearbeitung des Praktikums steht Ihnen wieder ein Visual-Studio- & XCode-Projekt im OSCA-Lernraum zur Verfügung (cgprakt5.zip). Sie müssen in dem jeweiligen Projekt die folgenden Dateien durch Ihre eigenen Implementierungen ersetzen:

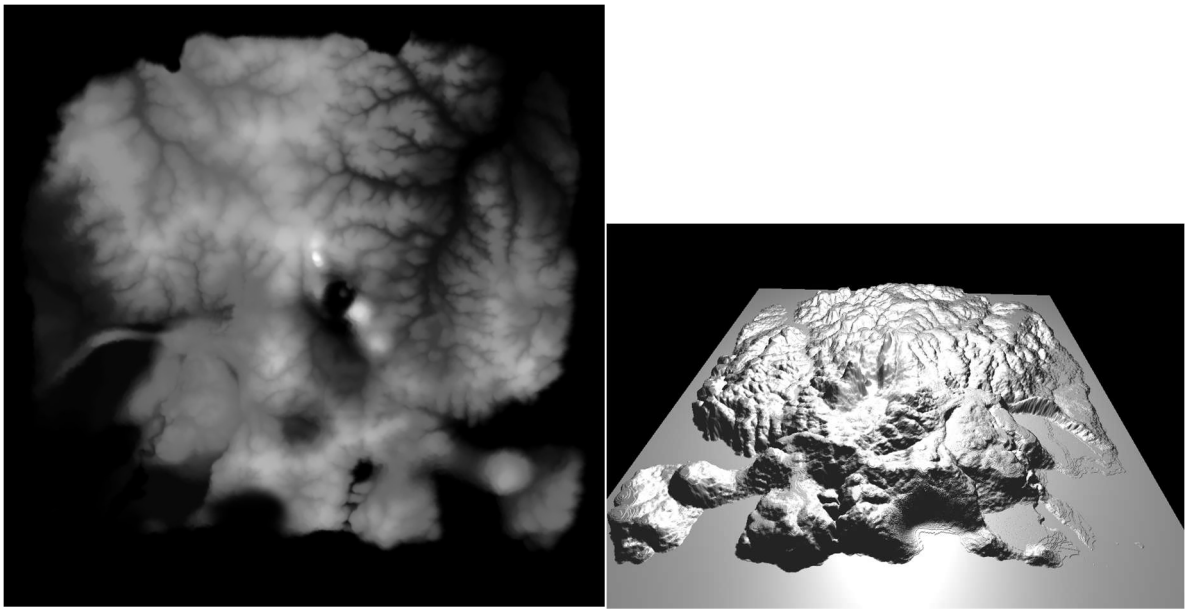
- vector.h & .cpp
- color.h & .cpp
- rgbimage.h & .cpp
- model.h & .cpp
- triangleboxmodel.h & .cpp
- lineboxmodel.h & .cpp

Für das Testat wird von Ihnen erwartet, dass Sie alle Konzepte des Praktikums verstanden haben und dass Sie diese erklären können.

Aufgabe 1 (4 Punkte)

In dieser Aufgabe geht es darum, dass Sie ein Terrain-Modell auf Basis einer Height-Map erzeugen. Eine Height-Map ist eine Bilddatei, die ein Höhenprofil für ein Modell enthält. Ein Vertex des Modells entspricht hier einem Pixel der Height-Map. Die Y-Koordinate für einen Vertex berechnet man, indem der Grauwert des korrespondierenden Pixels betrachtet wird (weiß=maximale Höhe, schwarz=minimale Höhe). Die XZ-Position eines Vertizes ergibt sich aus der XY-Koordinate des Pixels im Bild. Das folgende Bild soll das Prinzip veranschaulichen (links Height-Map, rechts korrespondierendes Modell):



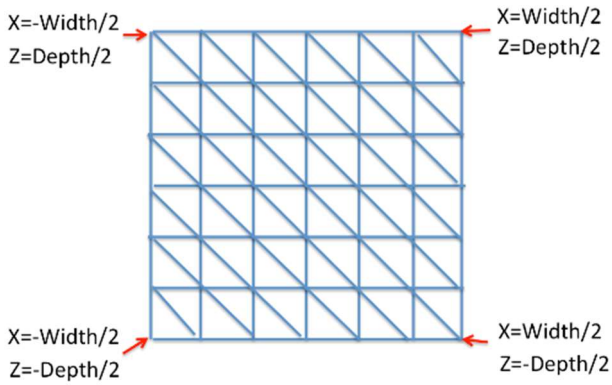


Ihre Aufgabe besteht darin, das Modell zur gegebenen Height-Map „assets/heightmap.bmp“ (liegt im OSCA-Lernraum) zu erzeugen. Dabei muss das Modell folgende Anforderungen erfüllen:

- Benutzen Sie Vertex- und Index-Buffer, um das Modell zu definieren.
- Das Modell hat nur so viele Vertizes wie die Height-Map Pixel hat.
- Der Index-Buffer soll Indizes für Dreiecke enthalten.
- Das Modell ist in XZ-Richtung zentriert zum Ursprung.
- Ein Vertex des Modells besitzt eine Position, eine Normale & Texturkoordinaten.
- Die Normalen der Vertizes werden gemittelt (nach dem Prinzip, welches in der Vorlesung vorgestellt wurde).
- Die Texturkoordinaten (u, v) können für diese Aufgabe alle auf 0 gesetzt werden.
- Die Klasse, die das Modell repräsentiert, soll die folgende Schnittstelle haben:

```
class Terrain : public BaseModel
{
public:
    Terrain(const char* HeightMap=NULL, const char* DetailMap1=NULL, const char* DetailMap2=NULL);
    virtual ~Terrain();
    bool load( const char* HeightMap, const char* DetailMap1, const char* DetailMap2);

    virtual void shader( BaseShader* shader, bool deleteOnDestruction=false );
    virtual void draw(const BaseCamera& Cam);

    float width() const { return Size.X; }
    float height() const { return Size.Y; }
    float depth() const { return Size.Z; }

    void width(float v) { Size.X = v; }
    void height(float v) { Size.Y = v; }
    void depth(float v) { Size.Z = v; }

    const Vector& size() const { return Size; }
    void size(const Vector& s) { Size = s; }
```

- Sie dürfen die Klasse beliebig erweitern.
- Der Parameter *HeightMap* soll den Dateinamen der Height-Map enthalten, die Parameter *DetailMap1* und *DetailMap2* können für diese Aufgabe ignoriert werden (diese werden für die nächsten Aufgaben benötigt).
- Die Methoden *width()* und *depth()* geben die Dimensionen des Modells entlang der X- und Z-Koordinate an:



- Die Methode *height()* gibt die maximale Höhe des Terrains an. Enthält zum Beispiel ein Pixel in der Height-Map den Wert 0.8f und ist *height*=5, so ergibt sich eine Höhe von y=4 für den korrespondierenden Vertex.
- Zur Darstellung des Modells benutzen Sie bitte die TerrainShader-Klasse (Erweiterung der PhongShader-Klasse). Diese lädt den Vertex-Shader „assets/vsterrain.glsl“ und Fragment-Shader „assets/fsterrain.glsl“.

- Die Größe des Terrains (width, height & depth) soll in Echtzeit anpassbar sein. Dies bedeutet, dass Sie die Skalierung, die in *Size* gespeichert wurde, im Vertex-Shader auf die Vertizes anwenden müssen (assets/vsterrain.glsl). Durch das Drücken der ‚s'-Taste soll man das Terrain skalieren können. Bewegen Sie die Maus bei gedrückter ‚s'-Taste nach oben bzw. unten, sollen die y-Koordinaten des Terrains vergrößert bzw. verkleinert werden (die Berge sollen wachsen oder flacher werden). Bewegen Sie die Maus nach rechts bzw. links soll das Terrain in der XZ-Koordinate skaliert (breiter oder schmaler) werden.

TIPP: Sie können die Height-Map mit Hilfe der Texture-Klasse laden und auf die Pixel zugreifen, indem Sie sich vom *Texture*-Objekt das *RGBImage*-Objekt geben lassen (*getRGBImage*()).

Aufgabe 2 (2 Punkte)

Auf Basis der Height-Map soll eine so genannte Mix-Map berechnet werden (auch Blend-Map genannt). Die Mix-Map kennzeichnet besonders steile Bereiche im Terrain, indem diese Bereiche durch helle Pixel hervorgehoben werden. In der nächsten Aufgabe wird dann diese Mix-Map genutzt, um das Terrain zu texturieren.
Zur Erzeugung der Mix-Map implementieren Sie den linearen Bildfilter zur Kantenhervorhebung nach Sobel. Der Filter kann wie folgt implementiert werden:

$$\mathbf{K} = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix},$$

$$\mathbf{U}_{x,y} = \sum_{j=0}^{2}\sum_{i=0}^{2}(\mathbf{I}_{x-i-1,y-j-1}\mathbf{K}_{i,j}), \qquad \mathbf{V}_{x,y} = \sum_{j=0}^{2}\sum_{i=0}^{2}(\mathbf{I}_{x-i-1,y-j-1}\mathbf{K}^{T}_{i,j}),$$

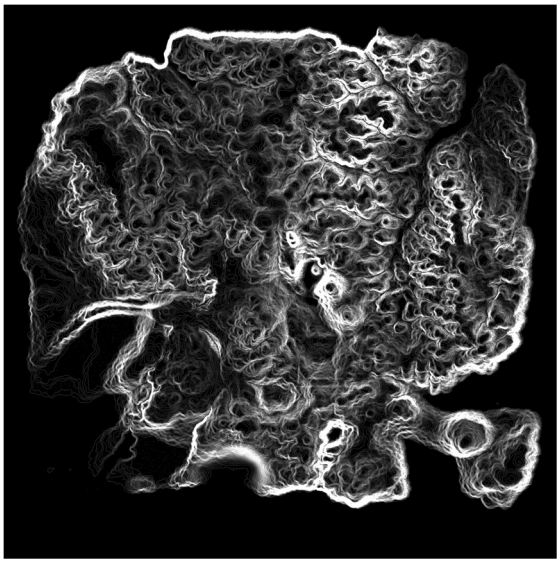
$$\mathbf{S}_{x,y} = \sqrt{\mathbf{U}^2_{x,y} + \mathbf{V}^2_{x,y}}$$

S entspricht dem Ergebnisbild, also dem Bild mit den hervorgehobenen Kanten. **I** bezeichnet das Eingangsbild (die Height-Map) und **K** ist die Filter-Maske für den Sobel-Operator (bitte beachten Sie, dass diese für die Berechnung von **V** transponiert wird).

Implementieren Sie den Sobel-Filter als statische Methode in Ihrer RGBImage-Klasse:

```
static RGBImage& SobelFilter( RGBImage& dst, const RGBImage& src, float factor=1.0f);
```

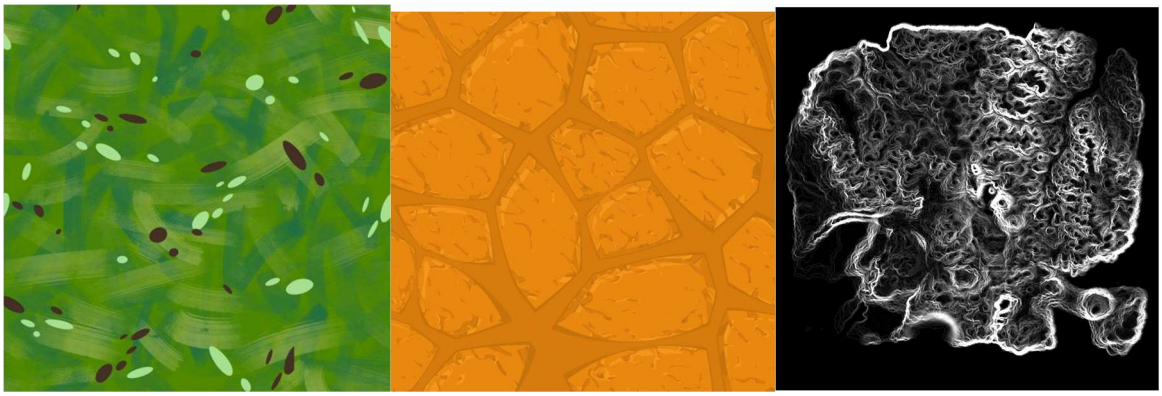
src bezeichnet das Eingangsbild (Height-Map) und *dst* das Ausgangsbild (Mix-Map). Zusätzlich kann ein Faktor angegeben werden, der mit **S** multipliziert wird, um die Kantenhervorhebung zu verstärken. *dst* und *src* müssen die gleiche Höhe und Breite besitzen (mit assert prüfen).
Wenn Ihre Implementierung korrekt ist, sollten Sie bei einem Faktor von 10 das folgende Bild zur Height-Map „assets/heightmap.bmp" erhalten:



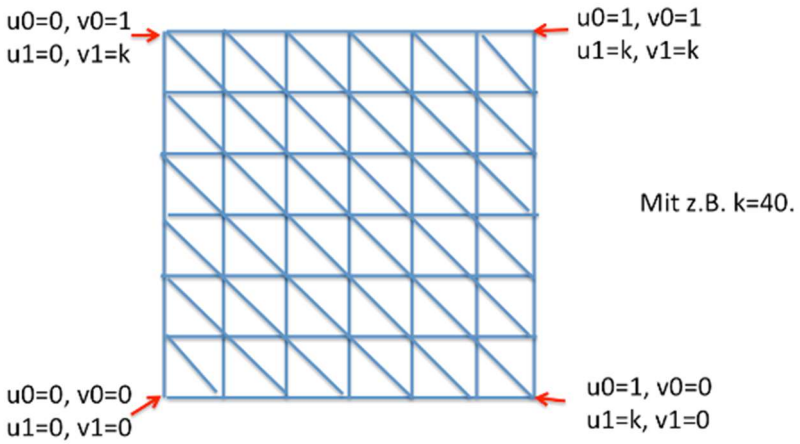
(Das Bild finden Sie auch im Assets-Ordner unter mixmap.bmp)

Aufgabe 3 (4 Punkte)

Als nächstes soll das Terrain-Modell aus Aufgabe 1 texturiert werden. Hierfür stehen Ihnen die folgenden Texturen zur Verfügung (grass.bmp, rock.bmp und Ihre generierte Mix-Map):



Die Gras- und die Stein-Textur bezeichnet man als Detailtexturen, diese werden über das Terrain „gekachelt“ (sie wiederholen sich mehrmals in X- und Z-Richtung). Die Mix-Map gibt ein Mischverhältnis an. In der Mix-Map bedeutet weiß, dass die Stein-Textur an dieser Stelle erscheinen soll, während für schwarze Pixel die Rasen-Textur verwendet werden soll. Für Grauwerte zwischen schwarz und weiß wird linear interpoliert.
Wichtig ist, dass die Mix-Map andere Texturkoordinaten verwendet als die Detail-Maps. Die Mix-Map benötigt UV-Koordinaten zwischen 0 und 1 und die Detail-Maps UV-Koordinaten zwischen 0 und 1*k (k bezeichnet die Anzahl der Wiederholungen):



Die schlussendliche Farbe für einen Texel auf der Terrain-Oberfläche ergibt sich aus:

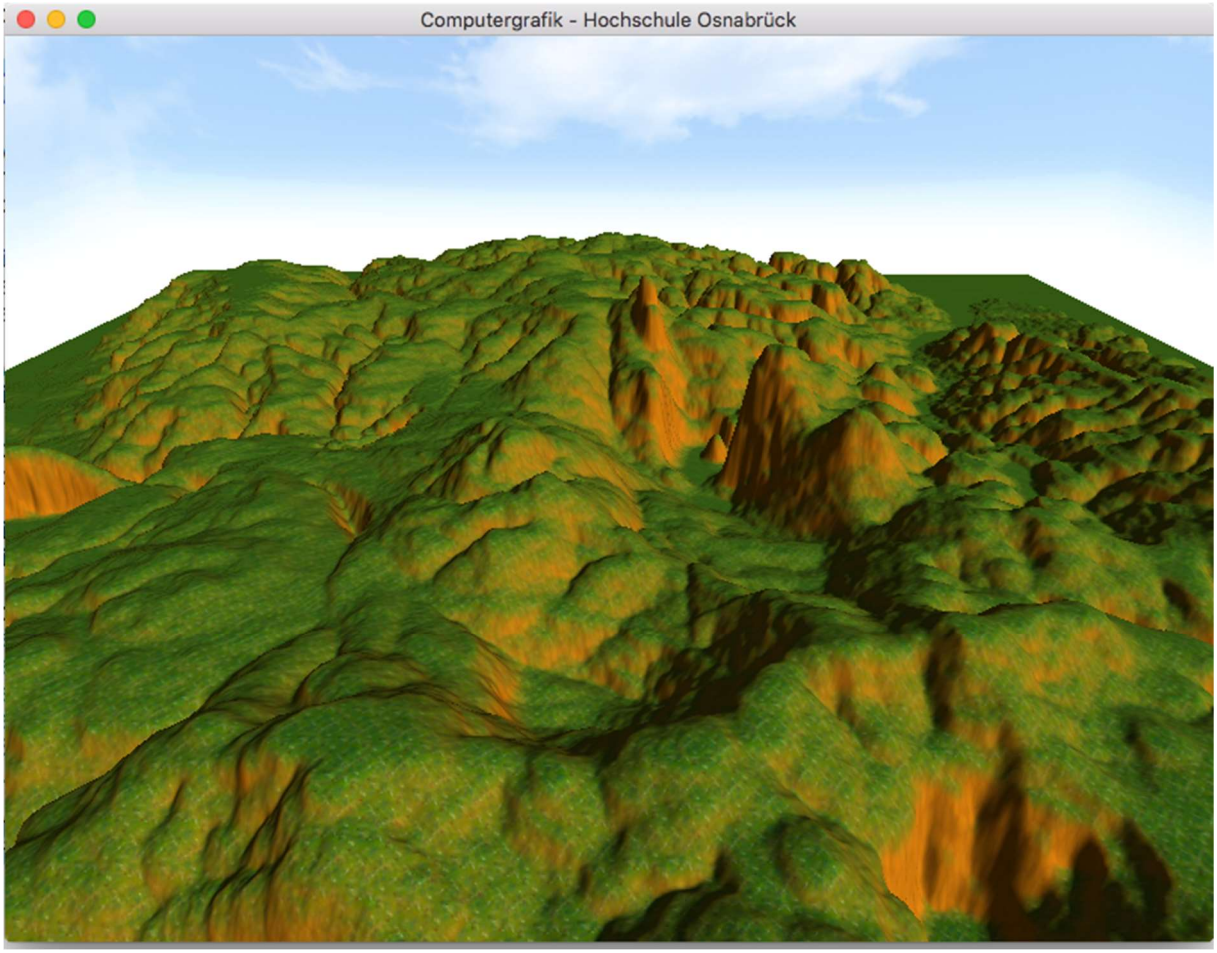
Texel_final = (Texel_Mixtextur * Texel_Steintextur
+ (1-Texel_Mixtextur)*Texel_Rasentextur)*Reflexionsergebnisfarbe.

Implementieren Sie diese Mischfunktion im Fragment-Shader (assets/fsterrain.glsl) und passen Sie die Terrain- sowie die TerrainShader-Klasse an.

Tipps:

- Für die lineare Interpolation können Sie die GLSL-Funktion *mix(...)* verwenden.
- Für das Texture-Mapping im Shader können Sie die GLSL-Funktion *texture(...)* nutzen.
- Die zwei unterschiedlichen Texturkoordinaten für die Mix- und Detail-Maps können auf zwei Arten erzeugt werden:
    - Methode 1: Der Faktor *k* wird dem Fragment-Shader übergeben (Uniform Parameter) und dort mit den normierten Texturkoordinaten (zwischen 0 und 1) multipliziert. In diesem Fall benötigt die Vertex-Struktur nur ein paar Texturkoordinaten (normiert zwischen 0 und 1).
    - Methode 2: Die Vertex-Struktur wird um ein zweites Texturkoordinaten-Paar erweitert, bei dem der Faktor *k* bereits eingerechnet wurde. Die zweiten Texturkoordinaten müssen dann von dem Vertex-Shader zum Fragment-Shader durchgereicht werden.

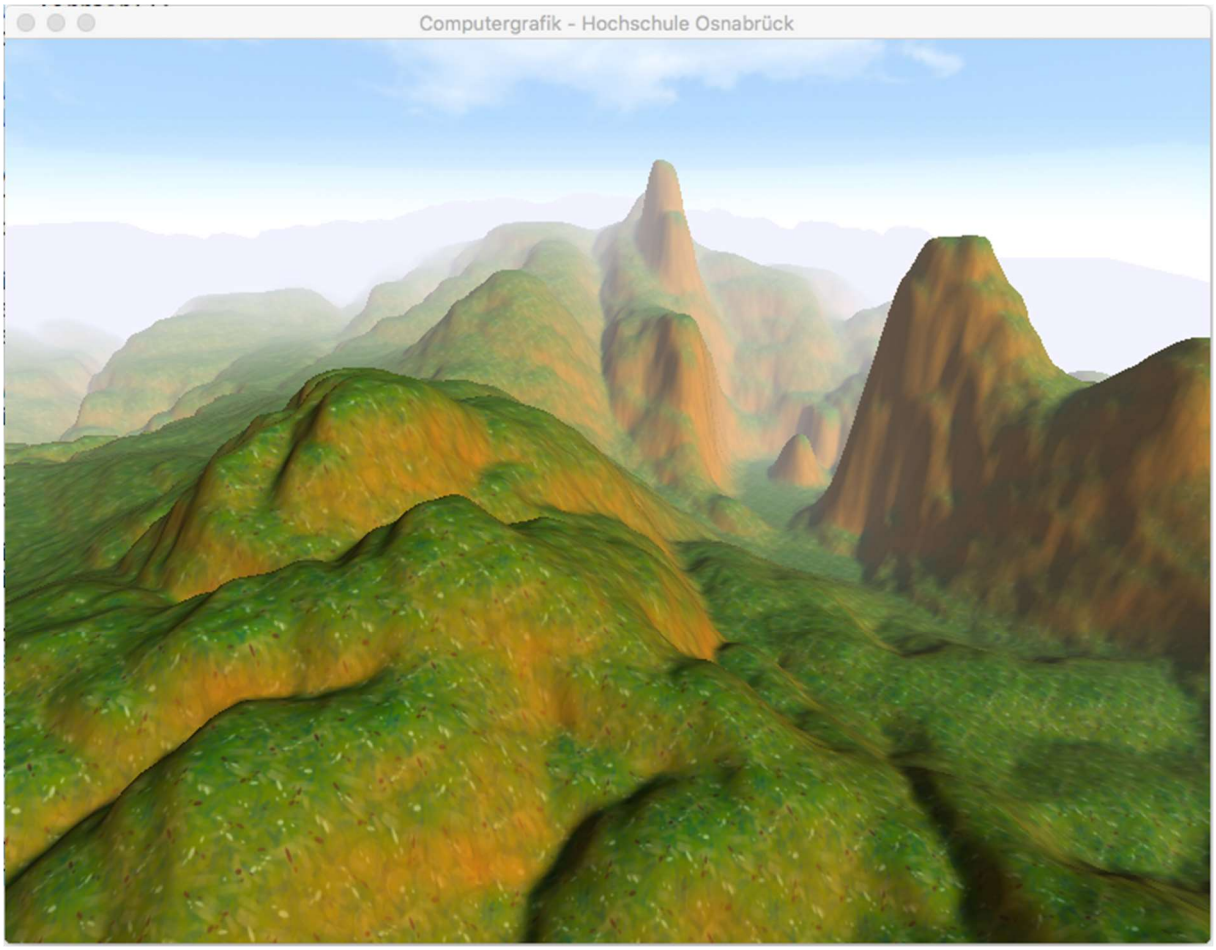
Wenn Ihre Implementierung korrekt ist, sollten Sie bei k=100, width=150, depth=150 und height=15 das folgende Bild erhalten:



(Das Skybox-Modell finden Sie unter assets/skybox.obj.)

Aufgabe 4 (Zusatzaufgabe, 2 Punkte)

Bei der gegenwärtigen Darstellung des Terrains setzt sich die Skybox noch relativ deutlich von dem Gebirge ab. Um den Kontrast zum Horizont etwas abzuschwächen, kann man im Fragment-Shader eine Funktion implementieren, die das Phänomen der so genannten atmosphärischen Streuung (Atmospheric Scattering) simuliert. Durch Streuungseffekte mit Partikeln in der Atmosphäre (In- & Out-Scattering) erscheinen Objekte in der Ferne weniger kontrastreich als nahe Objekte. Hierdurch wirken ferne Objekte „nebelig“. Das folgende Bild zeigt den Effekt:



Zur Berechnung der atmosphärischen Streuung kann die folgende Gleichung genutzt werden:

$$s = clamp\left(\left(\frac{d - d_{\min}}{d_{\max} - d_{\min}}\right)^{a}, 0, 1\right)$$

s bezeichnet den prozentualen Anteil des Nebels zwischen [0,1]. *s*=1 bedeutet, dass das Fragment komplett im Nebel liegt und *s*=0 bedeutet, dass kein Nebel vorhanden ist. *d* kennzeichnet den Abstand des Fragments (Pixels) zur Kamera. $d_{\min}$ ist die Distanz, ab der der Nebel einsetzen soll und $d_{\max}$ bezeichnet die Distanz, ab der die Fragmente komplett im Nebel liegen sollen. Der Exponent *a* steuert, ob der Nebel linear (*a*=1) oder exponentiell (*a*>=2) ansteigen soll.

s wird nachfolgend genutzt, um die Farbe des Fragments zu berechnen:

$$C' = (1 - s)C_{org} + sC_{fog}$$

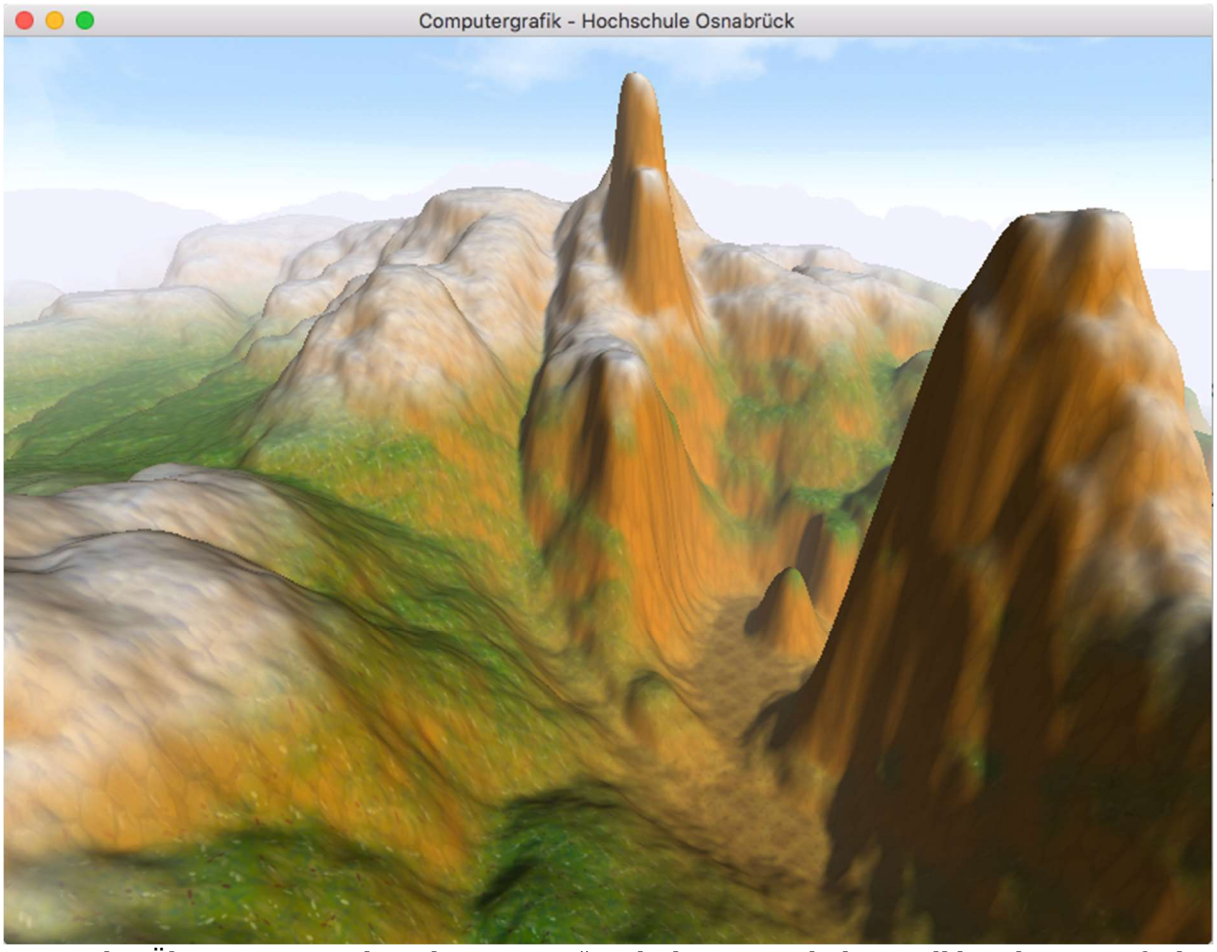
C_{org} entspricht der ursprünglichen Farbe des Fragments und C_{fog} entspricht der Farbe des Nebels.

Das obige Bild sollten Sie erhalten, wenn Sie die folgenden Parameter nutzen:

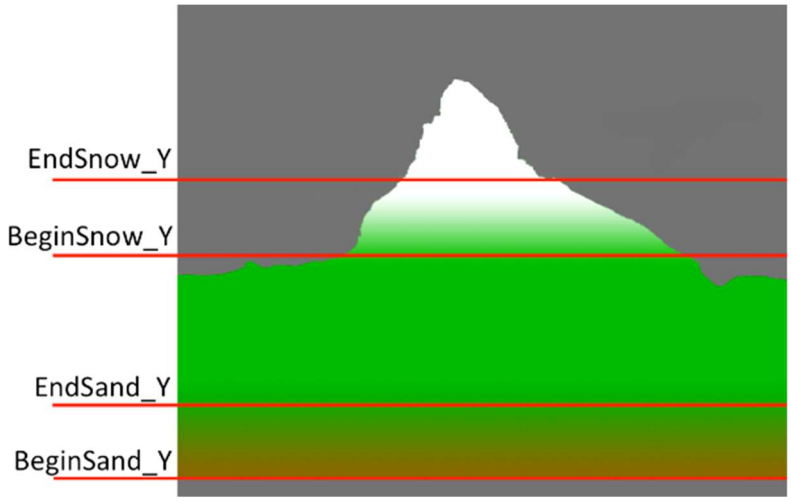
$$d_{\min} = 0, \quad d_{\max} = 50, \quad a = 3, \quad c_{fog} = (0.95, 0.95, 1).$$

Aufgabe 5 (Zusatzaufgabe, 3 Punkte)

Erweitern Sie das Terrain-System dahingehend, dass zusätzlich eine Schnee- und eine Sand-Textur als Detail-Textur genutzt werden können (snow.png & sand.png). Die Schnee-Textur soll für hohe Bereiche des Terrrains anstatt der Grass-Textur verwendet werden. Die Sand-Textur soll die Grass-Textur für tiefe Bereiche ersetzen. Betrachten Sie das folgende Bild, um das Prinzip zu verstehen:



Damit der Übergang zwischen den „Zonen“ nicht hart umschaltet, soll bei dieser Aufgabe ebenfalls der Übergang linear interpoliert werden. Hierfür sollten die folgenden Y-Grenzwerte genutzt werden:

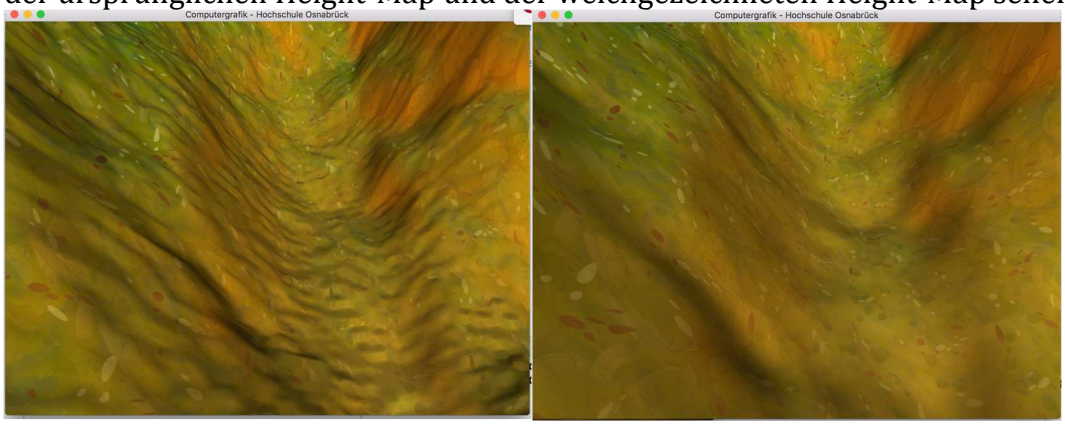


Zwischen Begin-und EndSand sollte linear zwischen der Gras- und der Sand-Textur interpoliert werden und zwischen Begin- und EndSnow zwischen der Gras- und Schnee-Textur.

Erweitern Sie die entsprechenden Klassen und Shader.

Aufgabe 6 (Zusatzaufgabe, 2 Punkte)

In der Height-Map sind die Höhenwerte leider nur in 256 unterschiedlichen Stufen kodiert (Pro Farbkanal nur 8-Bit). Dies führt zu relativ harten Abstufungen im Terrain. Um diese Abstufungen abzuschwächen, kann ein linearer Weichzeichnungsbildfilter nach Gauß auf die Height-Map angewendet werden. Einen Vergleich des Terrains mit der ursprünglichen Height-Map und der weichgezeichneten Height-Map sehen Sie hier:



(links ungefiltert, rechts gefiltert)

Für die Filterung wurde eine 7x7-Pixel breite diskrete Gaußfilter-Maske genutzt. Die Implementierung erfolgte CPU-seitig als statische Methode in der RGBImage-Klasse:

```
static RGBImage& GaussFilter( RGBImage& dst, const RGBImage& src, float factor=1.0f);
```

Um das Laufzeitverhalten des Filters zu verbessern, wurde die Filtermaske separiert und in einen vertikalen und horizontalen Pass unterteilt:

$$\mathbf{K} = \left(0.006,\ 0.061,\ 0.242,\ 0.383,\ 0.242,\ 0.061,\ 0.006\right),$$

$$\mathbf{G'}_{x,y} = \sum_{i=0}^{6} (\mathbf{I}_{x-i-3,y} \mathbf{K}_i),$$

$$\mathbf{G}_{x,y} = \sum_{i=0}^{6} (\mathbf{G'}_{x,y-i-3} \mathbf{K}_i)$$

I ist das Eingangsbild (Height-Map) und **G** entspricht dem gefilterten Ergebnis.

Viel Erfolg!