Computergrafikpraktikum 30.07. - 17.08.2012

Nico Marniok, Sascha Kolodzey, Oliver Vornberger
University of Osnabrück

In diesem Kurs sollte ein 3D-Terrain mit Hilfe von Java und OpenGL erstellt werden. Genutzt wurden dazu Java[1], OpenGL[2] und die LWJGL[3]. In verschiedenen Gruppen wurden einzelne Arbeitsschritte, die für die Erstellung eines großen Terrains benötigt werden, durchgeführt. Dazu gehörten die Generierung des Terrains, die Datenverwaltung, ClipMaps und Modellierung von Pflanzen sowie der Import der Models. Außerdem musste das Terrain beleuchtet und nachbearbeitet werden. Weiter sollte in das Terrain eine Wassersimulation eingearbeitet werden, die einen Fluss simuliert und darstellt.

Inhaltsverzeichnis

1 http://www.java.com
2 http://www.opengl.org
3 http://lwjgl.org/

1. Datenverarbeitung

1.1 Terraingenerierung

Unterthemen

- Einleitung
- Lineare Interpolation
- Terraingenerierung mit Noisemaps
- Terraingenerierung mit dem Diamond Square Algorithmus
- Smoothing
- Biome
  - Berg
  - Diamond Square Berg
  - Wüste
  - See
  - Flusslauf

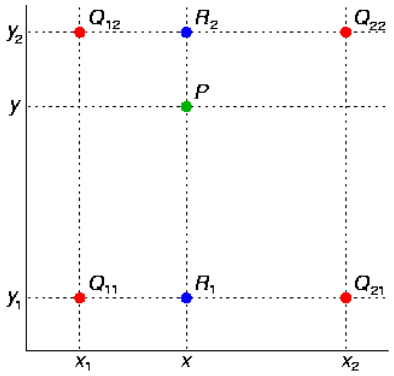
1.1.1 Einleitung

Ziel unserer Gruppe war es, eine Klasse zu schreiben, welche in sich eine Reihe von Algorithmen vereinigt, mit denen man sich ein ansprechendes Terrain ausgeben lassen kann. Das Format des Terrains war Ursprünglich ein 2 Dimensionales Array, welches an jedem Punkt eine Höhe speichert. Beim Auslesen würde sich demnach aus der Position des Wertes x- und z-Koordinate des darzustellenden Punktes berechnen lassen, und der Wert selber entspräche der y Koordinate. Dieses Konzept wurde allerdings erweitert, sodass sich nicht nur die y-Koordinate des Punktes ablesen lässt, sondern auch die Normale und das Material des Punktes. So kamen wir also zu einem Array der Form float[n][n][5], wobei n der Größe des gewünschten Terrains entspricht, da dessen Quadrat die Menge der Vertices ist. An Position 0 der Werte befindet sich die y-Koordinate, an Position 1, 2 und 3 die Koordinaten der Normale, und an Position 4 das Material, welches sich nur innerhalb unserer Arbeitsschritte sinnvoll ermitteln ließ und deswegen an dieser Stelle bereits übergeben werden musste.

Zurück

1.1.2 Lineare Interpolation

Die bei weitem am Häufigsten aufgerufene Methode während der Terraingenerierung ist die lineare Interpolation, weswegen sie an dieser Stelle einmal kurz umrissen wird, um spätere Ungenauigkeiten zu vermeiden. Viele Methoden benutzen Modellkarten von vorberechneten Strukturen mit Höhenwerten, wie Dünen, Seen, Bergen oder einfach Rauschen (-> Noise), und projizieren diese auf unsere Weltkarte. Die Modellkarten sind typischerweise verhältnismäßig klein (wir benutzen 32x32 Arrays). Wenn wir diese kleinen Karten nun auf einen deutlich größeren Bereich projizieren wollen, müssen wir eine mechanik bereitstellen, die Zwischenwerte aus Arrays richtig interpretiert. Der naive Ansatz dafür ist lineare Interpolation. Eine Anfrage des Wertes an der Stelle [17.4][9.9] würde erst die Werte bei [17][9] und [17][10] einlesen, und einen Mittelwert berechnen, welcher zu 10% von [17][9] und zu 90% von [17][10] gewichtet wäre, da die "tatsächliche" Position des Wertes bei [17][9.9] liegt. Exakt dasselbe würde für die Werte [18][9] und [18][10] passieren, wobei wieder [17][9] zu 10% und [17][10] zu 90% gewichtet wäre. Die beiden Mittelwerte würden nun erneut interpoliert werden, dieses Mal zu 60% aus dem zuerst berechneten und zu 40% aus dem danach berechneten, da [17.4][9.9] wieder der "tatsächlichen" Position des Wertes entspräche.



Zurück

1.1.3 Terraingenerierung mit Noisemaps

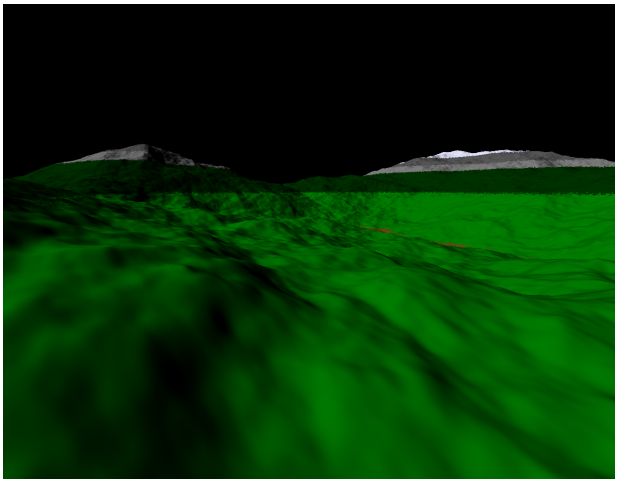
Das Grundprinzip von Terraingenerierung mit Noisemaps ist, dass man zufällige Werte mehrmals mit steigender Frequenz und sinkender Amplitude über ein Array laufen lässt, und so (für den Aufwand) relativ gute und realitätsnahe Höhenkarten bekommt. Wenn man zum Beispiel mit einer Frequenz von 0.3 beginnt, so erhält man bei einer Noisemapgröße von 32 * 32 Werten und einer Terraingröße von 1024 * 1024 auf jede jede Achse vom Terrain etwa 10 Noisewerte. Noisewerte werden folgenderweise generiert:

```
// Gen Noisemap
for(int x=0; x < noiseMap.length; ++x) {
    for(int z=0; z < noiseMap[0].length; ++z) {
        this.noiseMap[x][z] = this.random.nextFloat()*2-1;
    }
}
```

Also liegen alle Werte zwischen -1 und +1, was bedeutet, dass bei 10 zufälligen Werten und einer Anfangsamplitude von 1 die Wahrscheinlichkeit auf eine hohe Diskrepanz zwischen zwei oder mehr aufeinanderfolgenden Werten sehr groß ist, und sich durch diesen ersten Durchlauf grobe Berg- und Talstrukturen bilden. Für alle folgenden Durchläufe wird dann die Frequenz verdoppelt und die Amplitude halbiert. Wir mussten allerdings, da wir immer die selbe Noisemap benutzten, in die Halbierung der Frequenz einen geringen Zufallsfaktor einbauen, da es sonst zu wiederkehrenden Mustern kommt.

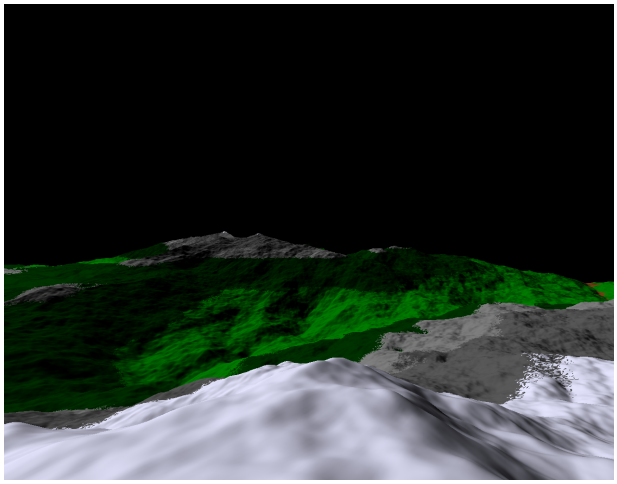
```
private void terraform(int surfaceWrink, int macroStructure, float scale){

    this.SCALE = scale;
    System.out.println("Terraforming");
    boolean needsRoughing = true;

    switch(macroStructure){
    case 1: Util.biLinIpol(this.terra.getBlock(), this.noiseMap,
                        0.5f*SCALE, 10f*SCALE);
    Util.biLinIpol(this.terra.getBlock(), this.noiseMap,
            1f*SCALE, 5.05f*SCALE);break;
    case 2: Util.biLinIpol(this.terra.getBlock(), this.desertMap1,
                        10f*SCALE, 4f*SCALE);break;
    case 3: Util.biLinIpol(this.terra.getBlock(), this.mountainMap1,
                        8f*SCALE, 20f*SCALE);break;
    case 4: Util.biLinIpol(this.terra.getBlock(), this.mountainMap2,
                        10f*SCALE, 4f*SCALE);break;
    case 5: Util.biLinIpol(this.terra.getBlock(), this.mountainMap3,
                        10f*SCALE, 10f*SCALE);break;
    case 6:
        int pow = 0;
        while(Math.pow(2, pow)< terra.getBlock().length){
            pow++;
        }
        float[][] dsMap = Util.diamondSquare(pow, surfaceWrink
                            * (float) terra.getBlock().length / 4096f, 0);

        for(int i = 0; i<dsMap.length-1; i++){
            for(int j = 0; j<dsMap.length-1; j++){
                terra.add(i, j, 0, dsMap[i][j]);
            }
        }
        needsRoughing = false;
    }
    if (needsRoughing) {
        float freq = 1f, amp = 0.9f * SCALE;
        for (int i = 0; i < surfaceWrink; i++) {

            if (i > 30)
                freq = 27f + (random.nextFloat() / 2f);

            Util.biLinIpol(this.terra.getBlock(), this.noiseMap, freq, amp);
            freq *= (2 + (random.nextFloat() / 5f - 0.2f));
            amp /= (2 + (random.nextFloat() / 5f - 0.2f));
        }
    }
    this.setMaterialsFromHeight(0,this.maxX,0,this.maxZ);

    System.out.println("Done");

}
```

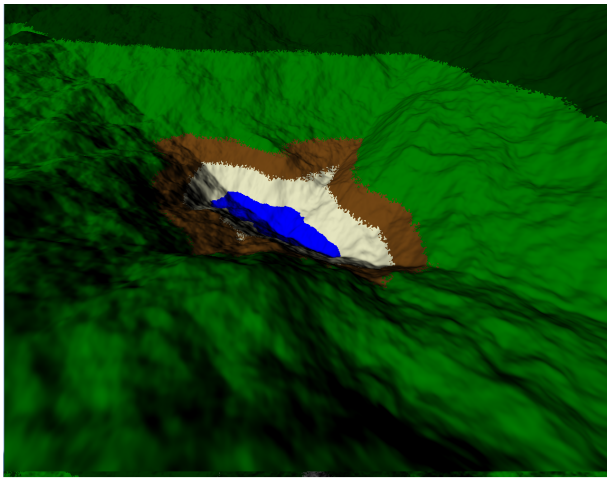
Dieser recht simple Code erzeugt, nach 8 Durchläufen und mit nach Höhe eingefärbten Vertices, ein Terrain wie zum Beispiel dieses hier:

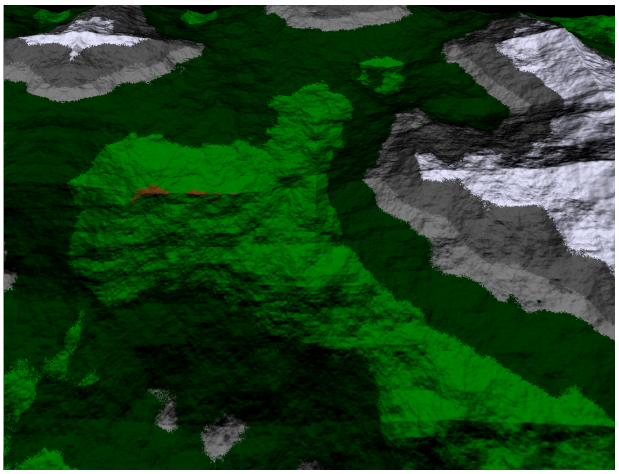


Es erfordert am Ende jedoch immer noch realtiv viel testen und Parameter-tuning, bis man ein mehr oder wenig Ansprechendes Terrain generieren kann. So ergab sich auch, dass mehr als acht Durchläuufe zu keinem merklichen Unterschied mehr führen.

Hier sind noch einige andere Beispiele von zufällig entstandenen interessanten Strukturen von Noisemaps, sowie eine Vogelperspektive:







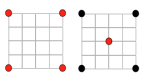
Zurück

1.1.4 Terraingenerierung mit dem Diamond Square Algorithmus

Der Diamond Square Algorithmus erzeugt zunächst ein zweidimensionales Array entsprechend der Größe der zu generierenden Welt. Durch diese zwei Dimensionen wird eine Fläche mit X- und Z-Werten aufgespannt, auf denen die Höheninformationen (Y-Werte) gespeichert oder berechnet werden. Dieser Algorithmus verdankt seinen Namen der Vorgehensweise, die verwendet wird, um das Array entsprechend zu füllen:

Square-Step

Zuerst werden die vier Eckpunkte der Karte mit zufälligen Werten gefüllt. Der Mittelpunkt wird berechnet, indem man den Mittelwert der vier Eckpunkte bildet.
Auf diesen Mittelwert wird nun noch eine zufällige Verschiebung addiert, um Struktur in das Terrain zu bringen. Damit ist der erste Teil des Algorithmus abgeschlossen. Diesen nennt man Square-Step, da er das erste Viereck generiert.

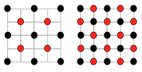


Diamond-Step

Der Diamond-Step funktioniert wie der Square-Step, jedoch wird das Viereck um 45 Grad gedreht und die Eckpunkte der entstehenden Raute werden aus den Eckpunkten des Vierecks gemittelt (aus dem Square-Step). Auch hier wird wieder eine zufällige Verschiebung miteingerechnet, jedoch wird diese durch einen konstanten Faktor verkleinert. So entstehen am Anfang grobe, bergige Strukturen; diese werden durch denselben Faktor verkleinert um den Berg letztendlich durch weitere kleine Buckel oder Hügel plastischer erscheinen zu lassen.

Rekursion

Die beiden beschriebenen Schritte werden nun rekursiv so lange durchgeführt, bis man das komplette Array gefüllt hat. Jedes Viereck wird in vier Quader aufgeteilt und wieder mithilfe des Square- und Diamond-Steps mit Höhenwerten gefüllt. So kann ein Array beliebiger Größe mithilfe des Diamond Square Algorithmus mit Höheninformationen für eine realistische, bergige Landschaft erstellt werden.

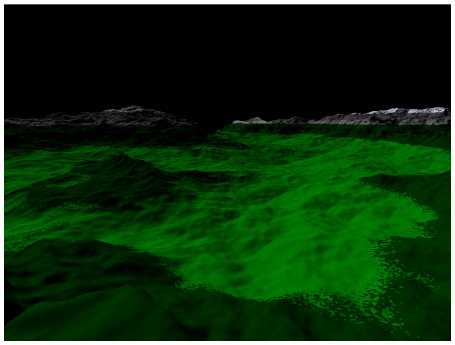


Durch den bereits erwähnten Faktor und die zufällige Verschiebung kann man nun auch einstellen, wie grob oder fein das enstehende Terrain werden soll. Die Grenzen für die zufällige Verschiebung gilt als Limit für das Verhältnis zwischen Berg und Tal; der Faktor bestimmt die Mikrostruktur und kann das Terrain zackiger erscheinen lassen oder auch abrunden.

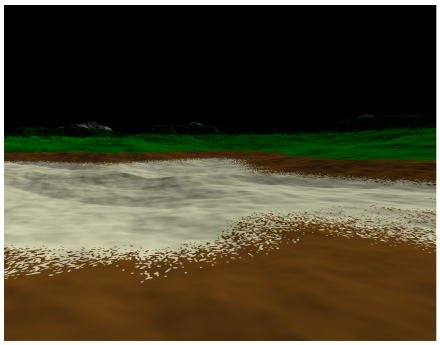
So entsteht eine Höhenkarte die ein realistisches Gebirge darstellt.

Umsetzung

Der Diamond Square Algorithmus ist durch seine Funktion zur Gebirgsgenerierung für uns leider nur begrenzt von Nutzen, da er eine Landschaft der Extremen erzeugt. Er generiert uns zwar ein abwechslungsreiches Gebirge, dennoch wollten wir die Möglichkeit einer Abflachung des Terrains, an den Gebieten in denen Gras, Erde oder Strand vorwiegend vorkommen, integrieren. Dafür haben wir eine Grenze anhand der Höheninformationen gesetzt, ab der die Höhenwerte, die durch den Algorithmus berechnet werden, verringert werden, indem wir die Wurzel aus ihnen ziehen. Auf diese Weise haben wir ab der Grenze kein flaches Gelände, sondern ein realistisches, unebenes Flachland.



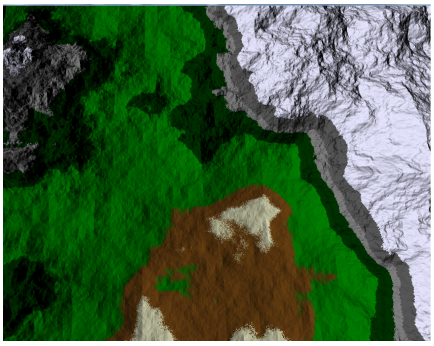
In diesem Bild kann man klar die Trennung von Gebirge und Flachland sehen, die durch das Wurzelziehen zustande kommt. Ohne dies würde die jetzt grüne Fläche stark in ein Meer übergehen und sozusagen einen negativen Berg bilden.



Hier ist eine zufällig entstandene Wüste zu sehen, die durch die integrierte Abflachung entstanden ist.

Abgewandelter Diamond Square Algorithmus

```
public static float[][] diamondSquare(int size, float rough, int seed){

    Random random = new Random(seed);
    int depth = size-1;
    float [][] dSMap = new float[(int) Math.round(Math.pow(2, size)+1)]
                                  [(int) Math.round(Math.pow(2, size)+1)];
    dSMap[0][0] = rough * (2*random.nextFloat()-1);
    dSMap[0][size-1] = rough * (2*random.nextFloat()-1);
    dSMap[size-1][0] = rough * (2*random.nextFloat()-1);
    dSMap[size-1][size-1] = rough * (2*random.nextFloat()-1);

    int iteration;
    boolean putX, putZ;
    while(depth > -1){
        iteration = (int) Math.round(Math.pow(2, depth));
        putX = false;
        for(int i=0; i<dSMap.length; i+=iteration){
            putZ = false;
            for(int j=0; j<dSMap[0].length; j+=iteration){
                if(putX == true && putZ == true){
                    // put diamond
                    dSMap[i][j] = (dSMap[i-(iteration)][j-(iteration)] +
                                   dSMap[i+(iteration)][j-(iteration)] +
                                   dSMap[i-(iteration)][j+(iteration)] +
                                   dSMap[i+(iteration)][j+(iteration)])/4
                                   + rough * (2f*random.nextFloat()-1);
                }
                if(putX != putZ){
                    // put squares
                    if(putX == true){
                        dSMap[i][j] = (dSMap[i-(iteration)][j] +
                                       dSMap[i+(iteration)][j]) /2
                                       + rough * (2f*random.nextFloat()-1);
                    }
                    else{
                        dSMap[i][j] = (dSMap[i][j-(iteration)] +
                                       dSMap[i][j+(iteration)]) /2
                                       + rough * (2f*random.nextFloat()-1);
                    }
                }
                putZ = !putZ;
            }
            putX = !putX;
        }
        rough/=2;
        depth--;
    }
    for(int i=0; i<dSMap.length; i++){
        for(int j=0; j<dSMap.length; j++){
            dSMap[i][j] = (float) (dSMap[i][j] <= 0 ?
                          -Math.sqrt(Math.abs(dSMap[i][j])) : dSMap[i][j]);
        }

    }
    return dSMap;
}
```



Hier wird eine "dSMap" nach dem Diamond Square Algorithmus mit der Begrenzung fürs Flachland erstellt. "Size" gibt hierbei die Größe des Terrains an, "rough" ist der Faktor welcher bestimmt, wie aufgeraut das Terrains sein soll, und mit "seed" kann man bestimmen, welches zufällige Terrain erstellt wird. So ergibt sich die Möglichkeit, die Parameter zu speichern, um das Terrain zu einem anderen Zeitpunkt wiederherzustellen.

Dieses Bild zeigt ein Terrainstück aus der Vogelperspektive. Die Vielfalt an unterschiedlichem Terrain, das durch den abgewandelten Algorithmus entsteht, ist dabei gut zu erkennen. Durch das Abändern einiger Parameter, wie beispielsweise der Initialhöhe, lässt sich auf einfache Art und Weise eine Küstenregion oder auch ein größeres Gebirge generieren. Leider ist der Diamond Square Algorithmus nicht in der Lage, alle "Biome", wie beispielsweise einen Flusslauf, zufällig zu generieren, . Aus diesem Grund haben wir die Möglichkeit integriert "Biome" wie einen Flusslauf im Nachhinein in das Terrain zu setzen (siehe Flusslauf).

Vor- und Nachteile

Vorteile

Durch den abgewandelten Diamond Square Algorithmus ist es möglich, eine plastische und realistische Landschaft zufällig zu erzeugen. Dabei ist nicht nur ein Landschaftstyp wie zum Beispiel Gebirge vertreten, sondern es ist möglich, eine ganze Palette von "Biomen", von Meer bis hin zu schneebesetzten Bergspitzen zu generieren. Durch die verschiedenen Parameter ist es auch möglich, das Terrain im Vorhinein an bestimmte Anforderungen anzupassen und so eine gewünschte Landschaft für jede Situation zu erstellen.

Wenn man den Diamond Square Algorithmus der "Noisemap Methode" gegenüberstellt, bemerkt man auch einen Leistungsschub, da beim Diamond Square Algorithmus die Höhenwerte nach der Berechnung nicht noch einmal verändert werden müssen. So ist eine schnellere Generierung möglich, da weniger Lese- und Schreibzugriffe nötig sind.

Nachteile

Da die Diamond Square Map - im Gegensatz zur "Noisemap Methode", bei der Werte nach und nach aufaddiert werden - separat erstellt werden muss, wird ein weiteres Array der Größe des zu generierenden Terrains benötigt.

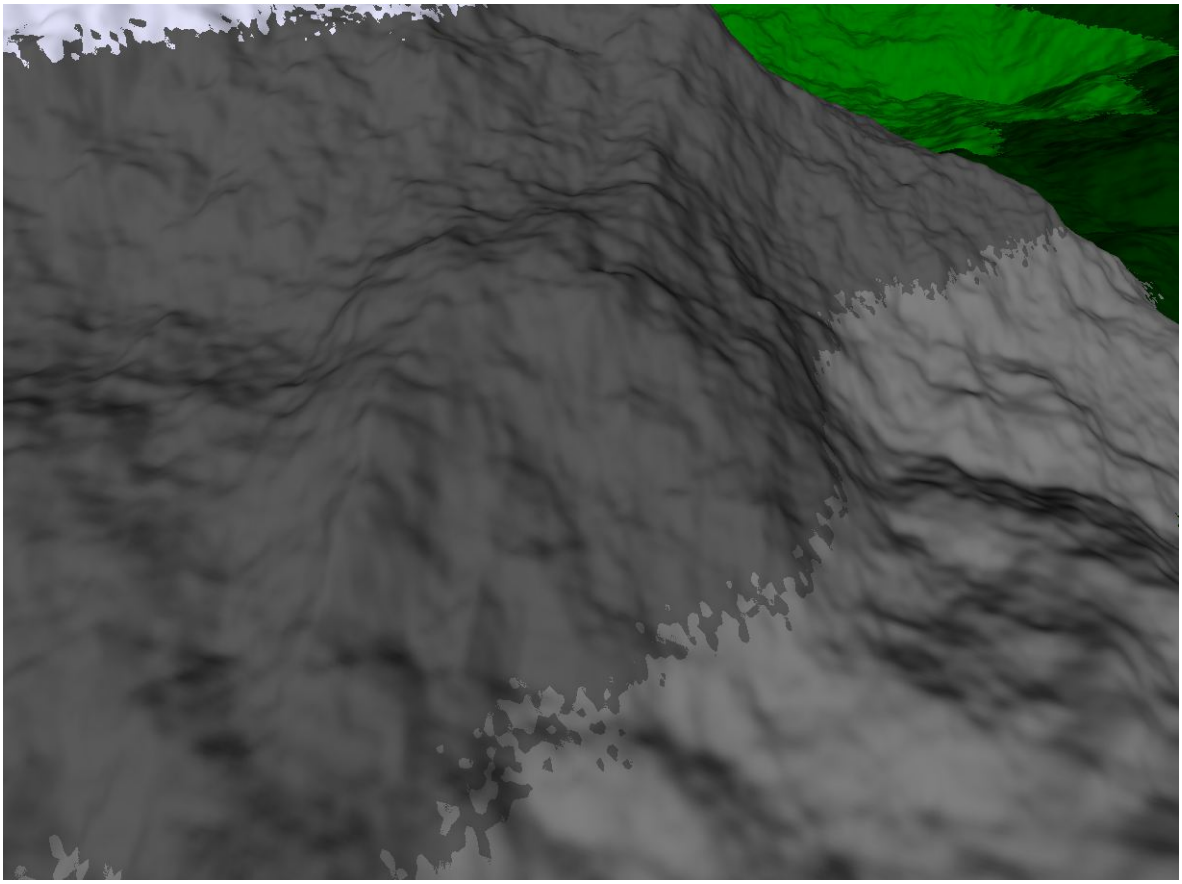
Da es lediglich die Aufgabe unserer Gruppe war, eine Klasse zur Terraingenerierung zu erstellen, besitzen wir keine fertige Datenstruktur die unser Terrain auf der Festplatte speichert. Daher verwenden wir den Arbeitsspeicher, der leider nur stark begrenzte Kapazitäten besitzt, und können daher nur eine stark begrenzte Terraingröße darstellen.
Dieser Nachteil kann durch eine ausgearbeitete Datenstruktur ausgeglichen werden.

Zurück

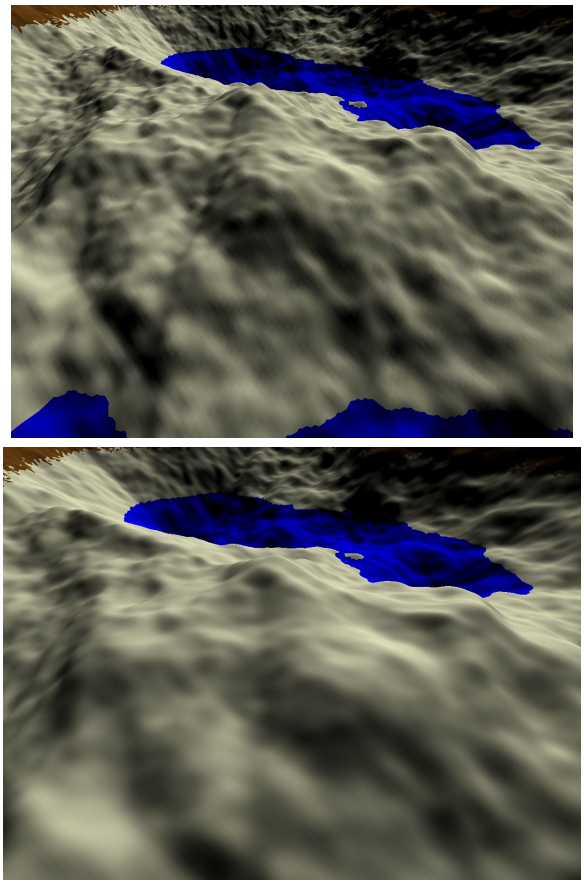
1.1.5 Smoothing

Smoothing war eines der ersten großen Probleme, welches wir ohne vorhandensein irgendwelcher Anleitungen zu lösen hatten. Ziel war es, anhand von Materialien des Terrains unterschiedlich stark Höhenwerte zu glätten, um zum Beispiel Sand oder Schnee nicht nur farblich von der Umgebung abgrenzen zu können. Nach anfänglichen Sachgassen wurden wir dann von den Tutoren in richtung Gaussfilter gestoßen, an denen wir auch bis zum Ende unseres Projekts zum Glätten festhielten. Diese wurden zuerst von uns in Handarbeit als Matrizen umgesetzt, einmal als einen schwach glättenden Filter der Größe 3x3 Vertices, und einen der Größe 7x7, der sehr zuverlässig auch starke Unebenheiten einebnet. Diese Filter lesen zunächst für jeden Punkt aus seiner unmittelbaren Umgebung (die konkrete Größe ist vom Filter abhängig) Höhenwerte benachbarter Vertices ein. Dann werden diese unterschiedlich stark abhängig vom Abstand des zu verändernden Vertex gewichtet, und das Mittel dieser Werte wird dann der neue Wert des Mittlersten.

An dieser Stelle sieht man den Unterschied zwischen 3x3-fach smoothing von hellem Gestein im unterschied zu komplett unverändertem dunklen Gestein:



Und hier ein vorher - nachher Bild von Sand.



Zurück

1.1.6 Biome

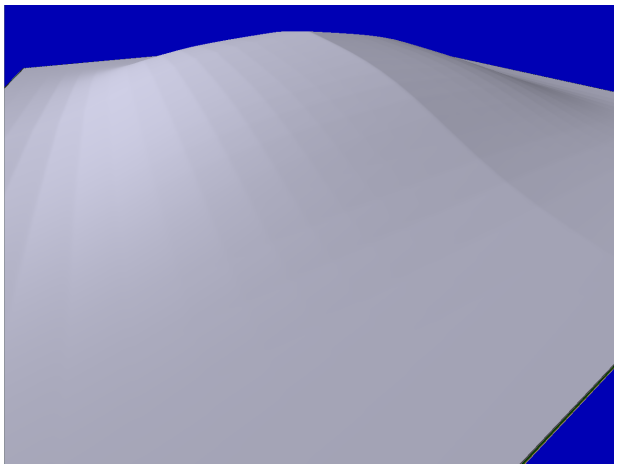
Zurück

1.1.6.1 Berg

Der Berg war das erste Biom, das wir setzen wollten, und war auch am einfachsten umzusetzen. In einigem hin und herprobieren erstellten wir insgesamt drei Bergstrukturen, die als einigendes Merkmal besitzen, dass sie am Rand alle die Höhe "0" (oder einen verschwindend geringen float-Wert), und in der Mitte die Höhe "1" haben.

```
//Gen MountainMap
for(int x=0; x < 32; ++x) {
    for(int z=0; z < 32; ++z) {

        this.mountainMap1[x][z] = ((16f-(Math.abs(-(x-16f))))/16f*(16f-(Math.abs(-(z-16f))))/16f);

        this.mountainMap2[x][z] = (float) (((Math.cos(((x-16)/12.9f)*((x-16)/12.9f)) *
                                            Math.cos(((z-16)/12.9f)*((z-16)/12.9f)))));

        this.mountainMap3[x][z] =  (mountainMap1[x][z] +(mountainMap2[x][z] * mountainMap2[x][z] * mountainMap2[x][z]
                                    * mountainMap2[x][z] * mountainMap2[x][z]))/2f;


    }
}
```

Der erste Berg stellt einen linearen Anstieg dar. Dies bedeutet, dass es von allen Seiten einfach gerade nach oben geht. Der zweite Berg multipliziert die cosinus-Werte der normierten Indices, was zu einem kurvenartigem Anstieg führt. Der dritte Berg schließlich ist eine Mischung der ersten beiden Varienten, und hat einen stetigen aber sanften Anstieg mit einem Plateau-artigem Wipfel, und ist unserer Meinung nach die beste Variante:



Überraschenderweise für uns ergab es sich fast gar nicht, dass wir diese Berge tatsächlich in unsere Landschaft setzten, da wir nach und nach deutlich bessere Methoden zum generieren realistischer Berge erstellten. Die "MountainMaps" behielten jedoch ihren Nutzen, da sie an vielen Stellen zur Gewichtung von Biomen eingesetzt wurden. Aufgrund ihrer Eigenschaft, mehr oder weniger stark zum Rand abzuflachen und ganz am Rand nur 0-Werte zu haben, waren sie sehr praktisch um gute Übergänge für andere Karten zu erstellen, oder generell nahtlose Übergänge sicherzustellen.

Zurück

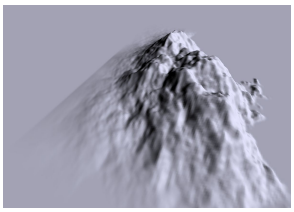
1.1.6.2 Diamond Square Berg

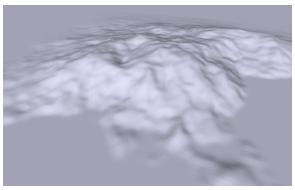
Unsere Methode "putDSMountain" ist in der Lage in ein bereits vorhandenes Terrain einen Berg von der Größe "range" an eine bestimmte Koordinate zu setzen. In dieser Methode verwenden wir wieder den **Diamond Square Algorithmus**, jedoch anders als in der grundlegenden Terraingenerierung.

Zuerst erstellen wir wie gewohnt ein Array, welches ein mit dem **Diamond Square Algorithmus** erzeugtes Terrain in der gewünschten Größe beinhaltet. Zudem nehmen wir außerdem die in der Initialisierung unserer Klasse erzeugten "MountainMap3" zur Hilfe. "MountainMap3" beinhaltet 32² Punkte in einem zweidimensionalen Array welches am "Rand" Nullen beinhaltet und zur Mitte hin steigt. Dieses Array hilft uns, den Berg nahtlos ins Terrain einzufügen, indem sie als Gewichtung für unseren Berg genutzt wird. So flacht der Berg zum Rand hin ab und lässt das vorherige Terrain unangetastet. Um "MountainMap3" unabhängig von der Größe des Berges zu erstellen, **interpolieren** wir dessen Werte prozedural.

```
public void putDSMountain(int x, int z, int range, float noise){
    int size = 0;
    while(Math.pow(2, size)<range){
        size++;
    }
    noise *= ((float)size * 0.1f);
    float[][] dSMountain = Util.diamondSquare(size-1, noise, 0);
    range = dSMountain.length/2;

    float[][] smoothStruct = mountainMap3;
    int mountainX = smoothStruct.length;
    int mountainZ = smoothStruct[0].length;
    int pX, pZ;
    float a, b;
    float dX, dZ;
    for(int i=0; i<2*range-1; i++){

        a = (float) mountainX / (float) (2f*range) * (float) i;
        pX = (int) a;
        dX = a - pX;

        for(int j=0; j<2*range-1; j++){

            b = (float) mountainZ / (float) (2f*range) * (float) j;
            pZ = (int) b;
            dZ = b - pZ;

            this.terra.add(x-range+i, z-range+j, 0, (dSMountain[i][j] > 0 ?
                                                     dSMountain[i][j] : 0)
                    * (Util.iPol
                            (Util.iPol(
                                    smoothStruct[pX % mountainX][pZ % mountainZ],
                                    smoothStruct[pX % mountainX][(pZ+1)%mountainZ],
                                    dZ),
                            Util.iPol(
                                    smoothStruct[(pX+1)%mountainX][pZ % mountainZ],
                                    smoothStruct[(pX+1)%mountainX][(pZ+1)%mountainZ],
                                    dZ),
                            dX)));

        }

    }
    setMaterialsFromHeight(x-range, x+range, z-range, z+range);
}
```

Hier sieht man einen "DSMountain" in seiner Rohform. Links wurde ein "DSMountain" mit der Methode "putDSMountain" in ein flaches Terrain eingefügt und rechts wurde ein anderer "DSMountain" mit einem geringeren Wert für den Parameter "rough" erzeugt .





Zurück

1.1.6.3 Wüste

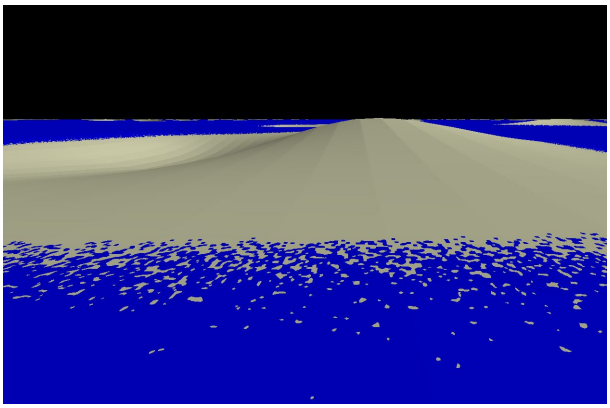
Die Wüste war eines der schwieriger zu setzenden Biome, da an ihrer Stelle nicht Werte auf vorherige aufaddiert wurden, sondern vorher alles eingeebnet werden musste, damit überhaupt eine Dünenartige Struktur sichtbar werden kann, und außerdem noch in einem gewissen Radius Materialswerte der Landschaft überschrieben werden mussten. Das Erstellen der Struktur einer Wüste haben wir foglendermaßen umgesetzt:

```
//Gen DesertMap1
for(int x=0; x < 32; ++x) {
    for(int z=0; z < 32; ++z) {
        this.desertMap1[x][z] = (float) (((16-(Math.abs(-(x-16))))/16f*
                                         (16-(Math.abs(-(z-16))))/16f)*
                                         (((0.05f*(((this.random.nextFloat()*2-1)/200)+
                                         (((Math.cos((x/6f)*Math.PI)))+(((Math.cos((z/12f)*Math.PI)))))))));
    }
}

//Gen DesertMap2
for(int x=0; x < 32; ++x) {
    for(int z=0; z < 32; ++z) {
        this.desertMap2[x][z] = (float) (((Math.sin(((x/16f)-1)*Math.PI )
                * Math.cos(((z*z/480.5f)-1) * Math.sqrt(Math.PI/2d))+2f)/2f* mountainMap3[x][z])/1.1180068f);
    }
}
```

Anschließend muss diese Düne in den wie oben beschreibenen präparierten Bereich gesetzt werden:

```
for(int i=0; i < 2*range-1; i++){

    a = (float) desertX / (float) (2f*range) * (float) i;
    pX = (int) a;
    dX = a - pX;

    for(int j=0; j < 2*range-1; j++){

        b = (float) desertZ / (float) (2f*range) * (float) j;
        pZ = (int) b;
        dZ = b - pZ;

        //interpolate height values

        this.terra.add(x-range+i, z-range+j, 0, amp *
                (Util.iPol
                        (Util.iPol(
                                desertMap2[pX % desertX][pZ % desertZ],
                                desertMap2[pX % desertX][(pZ+1)%desertZ],
                                dZ),
                                Util.iPol(
                                        desertMap2[(pX+1)%desertX][pZ % desertZ],
                                        desertMap2[(pX+1)%desertX][(pZ+1)%desertZ],
                                        dZ),
                                        dX)));
    }

}
```

In einer neutralen Umgebung sieht das so aus:



Diese Methode wurde gegen Ende unserer Kartenerstellung relativ redundant, da unsere vorherigen Terrainerstellungsmethoden bereits zufriedenstellende Wüsten zufällig erzeugten. Wenn einem diese allerdings nicht gefielen, hat man so immernoch die Möglichkeit gehabt eine an einen beliebigen Punkt in einem frei wählbaren Bereich zu setzen.

Zurück

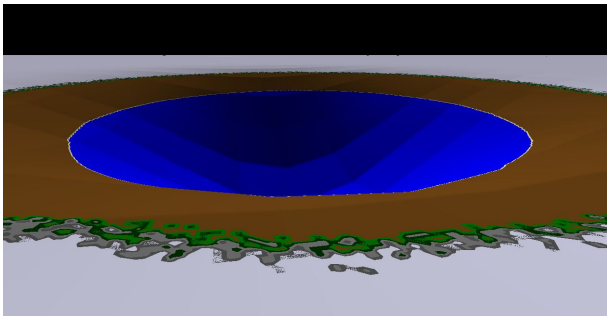
1.1.6.4 See

Der See verhielt sich in seinem Handling vergleichbar mit der Wüste, allerdings war bei ihm das setzen der Ufermaterilialien ein weiteres Problem. Als Höhenkarte benutzten wir an dieser Stelle die negativen Werte eines Gaussfilters, den wir vorher irgendwann einmal erstellt hatten, und der in Form eines float[17][17] Arrays vorlag, und auch die Kriterien "0 am Rand und 1 in der Mitte" erfüllte. Da diese Methode auch einen beliebig großen See setzen können sollte, werden in ihr auch alle zu lesenden Werte interpoliert. Updating der Höhendaten sowie anpassen der Materialien passiert alles im selben Schritt, was den vorteilt hat, dass sie sehr schnell läuft, und somit für die "setRiver" Methode problemlos recycled werden kann.

```
private void putLake(float scale, float depth, int x, int z, boolean putShore){

    System.out.println("Putting lake at "+x+" / "+z);
    // relict
    float rotation = 0;

    int riverX = gauss17.length-1;
    int riverZ = gauss17[0].length-1;
    int pX, pZ;
    float dX, dZ;
    float cosa = (float)Math.cos(rotation);
    float sina = (float)Math.sin(rotation);
    int range = (int)(scale * Math.ceil(Math.sqrt(riverX * riverX + riverZ * riverZ)));

    Matrix3f transformation = new Matrix3f();
    transformation.m20 = x;
    transformation.m21 = z;
    transformation.m00 = 0.5f * scale * riverX * cosa;
    transformation.m01 = 0.5f * scale * riverX * sina;
    transformation.m10 = 0.5f * scale * riverZ * -sina;
    transformation.m11 = 0.5f * scale * riverZ * cosa;
    transformation.invert();
    Vector3f vec = new Vector3f();
    vec.z = 1.0f;
    float riverVal;

    for(int i = x - range/2; i < x + range/2; i++){

        for(int j = z - range/2; j < z + range/2; j++){

            // set material to "river" if depth at that point is close to lvl



   if (i>0 && i<maxX && j>0 && j<maxZ) {
       vec.x = i;
       vec.y = j;
       Matrix3f.transform(transformation, vec, vec);
       if (vec.x < -1 || vec.x >= 1 || vec.y < -1
               || vec.y >= 1)
           continue;
       vec.x = 0.5f + 0.5f * vec.x;
       vec.y = 0.5f + 0.5f * vec.y;
       dX = (float) riverX * vec.x;
       pX = (int) Math.floor(dX);
       dX -= pX;
       dZ = (float) riverZ * vec.y;
       pZ = (int) Math.floor(dZ);
       dZ -= pZ;
       // interpolate height values
       riverVal = (Util.iPol(
               Util.iPol(this.gauss17[pX][pZ],
                       this.gauss17[pX][(pZ + 1) % riverZ],
                       dZ),
                       Util.iPol(this.gauss17[(pX + 1) % riverX][pZ],
                               this.gauss17[(pX + 1) % riverX][(pZ + 1) % riverZ],
                               dZ),
                               dX));

       this.terra.set(i, j, 0, Util.iPol(this.terra.get(i, j, 0), depth, riverVal));
```

```
                // set materials for lake
                if (riverVal > 0.125f * (float) scale / 20f) {
                    if (this.terra.get(i, j, 0) < (depth + 0.8f)) {
                        this.terra.set(i, j, 4, 1);
                    }
                }
                if (putShore) {
                    // set materials for shore around lake
                    if (((riverVal + ((random.nextFloat())) / scale) > 0.1f) && this.terra.get(i, j, 4) != 1) {
                        if((this.terra.get(i, j, 0) < (depth + 1.2f))){
                            this.terra.set(i, j, 4, 4);
                        }
                    }
                }
            }
        }
    }
}
System.out.println("Done");
```

In Anwendung sieht das Ganze dann so aus:



Zurück

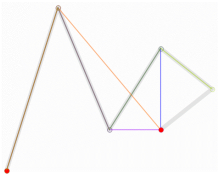
1.1.6.5 Flusslauf

Anforderung

Da wir in beiden Terraingenerierungsalgorithmen keine Möglichkeit haben, einen realistischen Flusslauf zu erzeugen, haben wir eine Methode implementiert, die einen Flusslauf nach der Terraingenerierung einfügen kann. Das Problem hierbei ergibt sich aus den Anforderungen, die wir an diese Methode stellen. Wir stellen uns unter einem realistischen Flusslauf vor, dass der Fluss natürliche Biegungen hat und sich abhängig von dem Material, durch das er fließen soll, verändert.

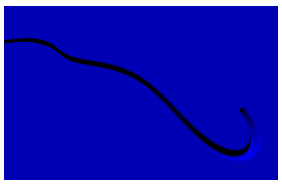
Ideen

Wir hatten viele Ideen einen Flusslauf mit einem Algorithmus zu erzeugen, kamen aber schließlich auf die Idee, Bezièrkurven zu verwenden da wir schon viel mit Interpolation gearbeitet haben und unsere bereits vorhandenen Methoden hier miteinbinden konnten.



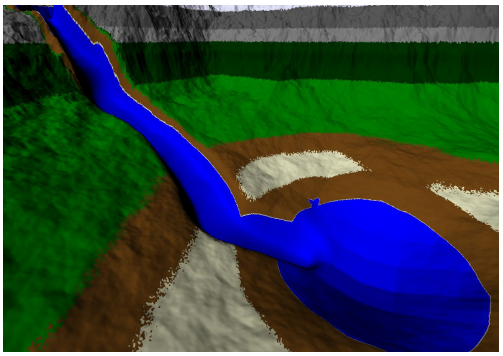
Bezièrkurven verwenden mehrere Punkte und interpolieren zwischen diesen um einen abgerundeten Verlauf zwischen dem Anfangspunkt und dem Endpunkt zu erzeugen. Jedoch wird nicht nur eine einfache Interpolation zwischen den Punkten vorgenommen, sondern es wird in mehreren Ebenen interpoliert. Erst werden die "Zwischenpunkte" interpoliert, die daraus folgenden Punkte werden auch wieder interpoliert und dies geschieht bis man nur noch einen Punkt hat. Dies geschieht nun für die komplette Linie indem man den jeweils zweiten Punkt der Interpolation stärker gewichtet; nun entsteht eine komplette Bezièrkurve. Bezièrkurven werden in Abhängigkeit der Anzahl der "Zwischenpunkte" aufwendiger; es muss pro "Zwischenpunkt" eine Interpolationsebene mehr durchlaufen werden.
Um den Flusslauf abhängig vom Material durch das er fließt breiter oder dünner werden zu lassen, überprüfen wir, durch welches Terrain unsere Bezièrkurve fließt. So vergrößern oder verringern wir die Breite des Flusses abhängig davon ob es in die Nähe von, beispielsweise Wasser oder Stein, gelangt.

Umsetzung



Der komplette Code würde hier zuviel Platz einnehmen, deswegen werden ich ihn kurz und simpel zusammenfassen.
Wir haben zwei Möglichkeiten implementiert Flüsse zu setzen, einmal indem man nur den Start- und Endpunkt festlegt und indem man zusätzlich ein Array mit den schon gesetzten "Zwischenpunkten" angibt.
Die Variante in der nur Start- und Endpunkt angegeben wird, berechnet anhand der Länge des zu generierenden Flusses unterschiedlich viele "Zwischenpunkte" indem entlang der Luftlienie zwischen Start- und Endpunkt mit zufälliger Verschiebung (Radius wird anhand der Länge festgelegt) neue Punkte gesetzt werden.
Mithilfe dieser Punkte kann nun in Etappen interpoliert werden und man erhält ein Array, welches die Punkte der Bezièrkurve enthält. So kann man an diesen Punkten im Terrain viele kleine **Seen** setzen und so einen realistischen Flusslauf generieren.
Außerdem wird am Rand des Flusses Erde als Material gesetzt um einen natürlichen Übergang ins Terrain zu erzeugen, wobei wir darauf achten, dass beim Übergang in einen See oder das Meer das vorhandene Flussbett nicht mit Erde überschrieben wird.

Zuletzt ein Beispiel, wie ein Fluss aus einem Gebirge in einen See mündet:



Zurück

1.2 Datenverwaltung

Inhaltsverzeichnis

- Datenstruktur
- I/O
- Probleme

- TerrainView
  - Datenmenge
  - Initialisierung der Datenmenge
  - Aktualisierung der Datenmenge
  - Initialisierung und Aktualisierung außerhalb des Terrains
  - Weitergabe von Daten
  - Laufzeit

Einleitung

Die erste Herausforderung unserer Gruppenarbeit lag darin, einen feineren Umriss der Aufgabenstellung zu formulieren. Zusammen mit der vorherigen und der nachfolgenden Gruppe sollte der Output eine Datei sein, die die Grafikkarte möglichst detailliert und trotzdem ressourcenschonend als Terrain darstellen kann. Wir einigten uns auf einen Datentyp, dem ein zweidimensionales Vertex-Array zugrunde liegt, wobei alle Vertexeigenschaften als float abgelegt werden. Weiterhin sollte das Terrain quadratisch sein, wobei die Größe eine Potenz von 2 ist. Erstrebenswert schien uns am Ende ein Terrain der Größe 65.536 (2^{16}) darstellen zu können, also fingen wir klein an und stießen bei einer Größe von 4.096 (2^{12}) an die Grenzen des Java-Heaps.

Um mehr Informationen verarbeiten zu können brauchte es jetzt einen Workaround, um den sich unsere Gruppe Gedanken gemacht hat. Als erstes wurde uns klar, dass es unmöglich wäre auf einer einzigen großen Datei zu operieren - stattdessen teilten wir unser Terrain in kleinere Blöcke auf, die jeweils 256x256 Vertices beinhalten. Das Terrain-Objekt ändert sich also dahingehend, dass nun ein zweidimensionales Block-Array zugrunde liegt. Nach Absprache mit den anderen Gruppen entschieden wir uns für folgende Arbeitsaufteilung:

1. **`Terrain.java`**
   Diese Klasse stellt die Schnittstelle zur vorherigen Gruppe dar und liefert Methoden, um Vertices in einem beliebig großen Terrain mit Werten zu versehen.
2. **`Block.java`**
   Diese Klasse speichert 256x256 Vertices und die Position im Terrain.
3. **`BlockUtil.java`**
   Diese Klasse liefert eine Reihe von statischen Methoden, um Blöcke vom Hauptspeicher auf die Festplatte zu schreiben und umgekehrt Blöcke von der Festplatte in den Hauptspeicher zu laden.
4. **`TerrainView.java`**
   Diese Klasse stellt die Schnittstelle zur nachfolgenden Gruppe dar und liefert insbesondere eine HeightMap der aktuell anzuzeigenden Vertices.

Auf die genaue Funktionsweise der einzelnen Klassen wird im Folgenden näher eingegangen.

Terrain

Als Schnittstelle zur vorherigen Gruppe bietet die Klasse Terrain getter- und setter-Methoden, um die Werte einzelner Vertex-Eigenschaften im Terrain abzufragen oder zu setzen. Jeder Vertex hat sieben Eigenschaften: x- und z-Koordinate, Höhe (y-Koordinate), Material und x-, y- und z-Koordinate der zugehörigen Normalen. Der naive Ansatz, ein Terrain-Object durch ein zweidimensionales Array von Vertices zu repräsentieren, stößt wie beschrieben schnell an die Grenzen des Java-Heaps:

Terraingröße	Speicherbedarf	Ladezeit*
1.024 x 1.024	20MB	4sec (30sec)

2.048 x 2.048	80MB	16sec (2min)
4.096 x 4.096	320MB	1min (8min)
8.192 x 8.192	1.3GB	4min (32min)
2^{16} x 2^{16}	82GB	4h (1.5d)

* Initialisierung (Alle Berechnungen fertig [Untergrenze])

Offensichtlich lässt sich die Terraingröße nicht ohne Weiteres beliebig groß wählen, weil der Speicherbedarf zuerst den Java-Heap (Standardgröße 128MB, sinnvoll erweiterbar bis etwa 2GB) sprengt und schließlich den insgesamt vorhandenen Arbeitsspeicher (Standardgröße etwa 4GB, bis zu 16GB). Für eine schonendere Nutzung wird das gesamte Terrain in einem weiteren Ansatz umgesetzt als Array von Blöcken der Größe 256x256 (1.28MB), welche je nach Bedarf im Arbeitsspeicher gehalten oder auf die Festplatte geschrieben werden können. Dieser Ansatz funktioniert problemlos, weil ein Terrain immer quadratisch ist und die Dimensionsgröße eine Potenz von 2 ist, und bietet den Vorteil des Konzepts "Divide and Conquer". Ein weiterer Vorteil dieses Ansatzes besteht darin, dass ein Terrain nur einmal erstellt werden muss und bei Neustart des Programms von der Festplatte ausgelesen werden kann. Die Ladezeit erhöht sich bei reinem Lesen von bzw. Schreiben auf der Festplatte allerdings erheblich, hier besteht Optimierungspotenzial.

Initialisierung

Wenn ein neues Terrain erstellt wird, werden der angegebenen Terraingröße entsprechend viele Blöcke auf die Festplatte geschrieben. Initial sind alle Vertex-Eigenschaften auf 0.0f gesetzt, bis auf eine einheitliche Grundhöhe, welche direkt im Kontruktor angegeben werden kann. Später werden von der vorherigen Gruppe alle Werte nach und nach sinnvoll belegt.

Aufruf:

```
Terrain terra = new Terrain(1024, 1.0f);
```

Umsetzung:

```
for(int i = 0; i < 1024 / 256; i++)
{
    for(int j = 0; j < 1024 / 256; j++)
    {
        // Position im Terrain
        Block block = new Block(i, j);

        // Initialhoehe setzen
        for(int k = 0; k < 256; k++)
            for(int l = 0; l < 256; l++)
                block.setInfo(k, l, 0, 1.0f);

        // Auf Festplatte schreiben
        BlockUtil.writeBlock(block);
    }
}
```

Kapselung

Ein Terrain-Objekt kapselt den Array-Zugriff auf einen Vertex mit einer get- und einer set-Methode. Die Paramter `x` und `z` sind dabei jeweils die x- bzw. z-Koordinate im Terrain, `pos` bestimmt das Vertexattribut.

`Position`	0	1	2	3	4
`Attribut`	Höhe	x-Richtung Normale	y-Richtung Normale	z-Richtung Normale	Material

```
public boolean set(int x, int z, int pos, float value);
public float get(int x, int z, int pos);
```

Standardmäßig liegen nach der Initialisierung alle Blöcke auf der Festplatte vor. Sollen Werte im Terrain geändert werden, wird der entsprechende Block in den Hauptspeicher geladen und nach dem Schreibvorgang wieder auf die Festplatte geschrieben. Analoges gilt für den Lesevorgang.

Aufruf:

```
Terrain terra = new Terrain(1024);
terra.set(300, 550, 0, 5.0f);
```

Umsetzung:

```
// Block-Position im Terrain
int idX    = 300 / 256;
int idZ    = 550 / 256;

// Vertex-Position im Block
int blockX = 300 % 256;
int blockZ = 550 % 256;

Block block = BlockUtil.readBlock(idX, idZ);
block.set(blockX, blockZ, 0, 5.0f);
BlockUtil.writeBlock(block);
```

Fehlerbehandlung

Es gibt mehrere Fehlerquellen, wenn auf einem Terrain-Objekt Operationen durchgeführt werden können:

- **Falsche Terraingröße**
  Um zu gewährleisten, dass ein Terrain stets quadratisch ist und dass die Länge eine Potenz von 2 ist, erhält der Konstruktor nur eine Dimensionsgröße, die auf die nächst kleinere Potenz von 2 gesetzt wird (mindestens 1.024).

  ```
  Terrain(int size) {
      int i = 1024;
      while(i < size) i *= 2;
      size = i;
      // usw.
  }
  ```

- **Falscher Zugriff**
  Um eine `ArrayOutOfBounds Exception` (und damit Programmabsturz) zu verhindern und trotzdem Rückmeldung zu erhalten, liefert die setter-Methode einen `boolean`: `true` bei gültigem Zugriff, `false` sonst. Die getter-Methode liefert bei ungültigem Zugriff einen Standardwert (`0.0f`) und druckt eine Fehlermeldung (`System.err.println("return is 0");`);

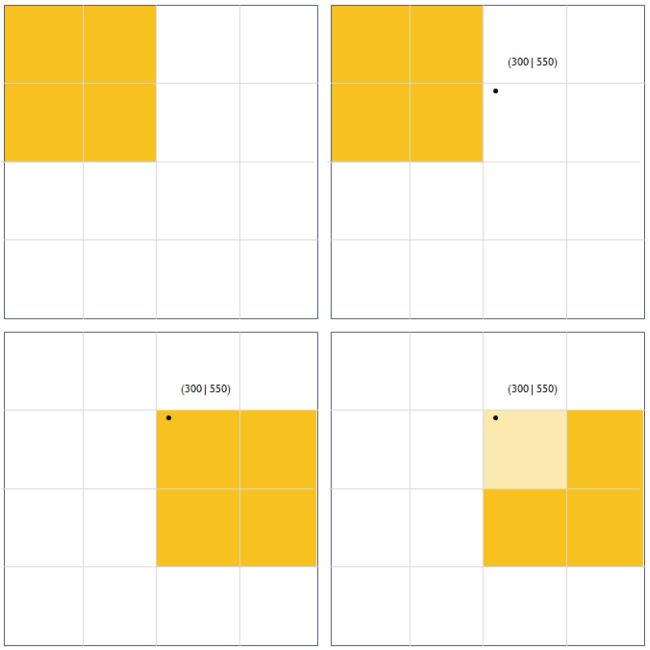
Speicherbelegung und Optimierung

Da Lese- und Schreibvorgänge von der vorherigen Gruppe besonders intensiv genutzt werden, sollte man sich überlegen, wie schnell diese ausgeführt werden. Die Geschwindigkeit eines einzelnen Vorgangs hängt von der Performanz der Klasse BlockUtil ab, aber tatsächlich wird auf alle Vertices zugegriffen. Bei einem einzelnen Vorgang wird ein Block von der Festplatte in den Arbeitsspeicher geladen und wieder zurückgeschrieben, wenn ein benachbarter Vertex bearbeitet werden soll muss der Block also erneut geladen werden. Es bietet sich an bestimmte Blöcke im Arbeitsspeicher zu behalten, weil die vorherige Gruppe zeilenweise alle Vertices im Terrain bearbeitet und die Zugriffszeiten auf Daten im Arbeitsspeicher wesentlich kürzer sind. Es können ohne weiteres bis zu 64 Blöcke (64 * 1.28MB = 82MB) im Speicher gehalten werden, damit lassen sich Operationen auf Terrains bis zu einer Größe von 2.048x2.048 komplett im Arbeitsspeicher durchführen.

Die Anzahl der Blöcke kann direkt in der Klasse angegeben werden und ist eine Quadratzahl, vorzugsweise eine Potenz von 2, und sollte die maximale Anzahl an möglichen Blöcken nicht überschreiten. Dieser Wert `MEM_BLOCKS_SIZE` ist `private` und per default auf 16 gesetzt. Welche Blöcke aktuell im Arbeitsspeicher liegen, hängt vom letzten get- bzw. set-Aufruf ab: Wird ein Vertex abgefragt, der nicht in einem Block im

Arbeitsspeicher enthalten ist, werden diese Blöcke auf die Festplatte geschrieben und dann aktualisiert - der erste neue Block enthält dann den abgefragten Vertex.

Die ersten Blöcke werden bei der Initialisierung des Terrains in den Hauptspeicher geladen, ausgehend vom obersten linken Block. Dadurch erhöht sich zwar die Initialisierung des Terrains etwas, aber die folgenden Lese- und Schreibe-Zugriffe erfolgen um ein Vielfaches schneller. Hier werden die Blöcke in quadratischer Form abgespeichert, was sich insgesamt als effizienter herausgestellt hat als eine zeilenweise Abspeicherung.




```
private static int MEM_BLOCKS_SIZE = 4;
Block[] currentBlocks = new Block[MEM_BLOCKS_SIZE];

// Block-Position im Terrain
int idX    = 300 / 256;
int idZ    = 550 / 256;

// Vertex-Position im Block
int blockX = 300 % 256;
int blockZ = 550 % 256;

// Test, ob Vertex im Speicher liegt
boolean test = false;
for(int i = 0; i < MEM_BLOCKS_SIZE; i++)
{
    Block b = currentBlocks[i];
    if(b.getID()[0] == idX && b.getID()[1] == idZ)
        test = true;
}

if(!test) // Aktualisieren
{
    // Hauptspeicher auf Festplatte schreiben
    for(int i = 0; i < MEM_BLOCKS_SIZE; i++)
        BlockUtil.writeBlock(currentBlocks[i]);

    // Neues Quadrat von Bloecken laden
    int count = 0;
    int dim = 1024 / 256;
    for(int i = 0; i * i < MEM_BLOCKS_SIZE; i++)
        for(int j = 0; j * j < MEM_BLOCKS_SIZE; j++)
            currentBlocks[count++] = BlockUtil.readBlock((idX + i) % dim, (idZ + j) % dim);
}

// ausgehend davon, dass hier ein Update vorgenommen wurde
// den gewuenschten Wert an entsprechender Stelle setzen
currentBlocks[0].set(blockX, blockZ, 5.0f);
```

Block und BlockUtil

In diesem Kapitel setzen wir uns mit der Strukturierung und dem Transfer der generierten Daten auseinander, u.a dem Lese- und Schreibprozess. Hier gehen wir der Frage nach, wieso und weshalb man die Daten so strukturieren sollte und wie wir diese möglichst effizient zwischen Arbeitsspeicher und Festplatte transferieren wollen.

Ansatz

Wir haben nun ein Terrain generiert und wollen dessen Daten nun verarbeiten und zur Darstellung bereitstellen. Wenn man die gegebenen Maße (Anzahl der `floats`) hochrechnet, wird man bei bestimmten Terrain Größen schnell eine Datenmenge erzeugen, die mehrere Gigabyte umfasst. Auch wenn wir im Zeitalter der 64-Bit Systeme angekommen sind, lässt sich der naive Ansatz, einfach alles im Arbeitsspeicher zu belassen, getrost als Utopie abschieben. Denn selbst heutzutage ist ein Computer mit 12 Gigabyte Arbeitsspeicher oder mehr nicht als Selbstverständlichkeit zu verstehen. Aber selbst wenn alles im Arbeitsspeicher stehen könnte, wollen wir bestimmt nicht bei jedem Start das Terrain neu generieren, es sollte am Ende der Generierung komplett auf der Festplatte gespeichert werden. Man muss also diese Datenmenge in kleine Stücke unterteilen, die man bei Bedarf von der Festplatte lesen und im Arbeitsspeicher halten kann.

Datenstruktur

Diese kleinen Stücke bezeichnen wir künftig als Blöcke. Ein Block-Objekt ist im Grunde nichts weiteres als ein `float[][][]` Array. Zuzüglich dessen besitzt jeder Block eine ID, bestehend aus einer X- und einer Z-Koordinate, welche die Position des Blocks im Terrain angeben. Ein Block umfasst standardmäßig ein Array bestehend aus 256 * 256 * 5 `float` Werten, wobei die 5 für Länge des Vertex Layouts steht.

Umsetzung:

```
    public Block(int posX, int posZ, int size, int vertexLayout)
    {
        this.posX = posX;
        this.posZ = posZ;

        if(size % 256 == 0)
{
     vertexInfo = new float[size][size][vertexLayout];
       }
       else
       {
            System.err.println("Error: Block Size ist kein Vielfaches von 256");
       }
    }
```

Die Seitenmaße (default 256) müssen je nach Terrain-Größe angepasst werden, da es andernfalls zu Problemen kommt, auf die später noch eingegangen wird. Auf der Festplatte werden die Blöcke dann als `blockfiles` (.bf) abgespeichert. Jedes dieser `blockfiles` wird dabei wie folgt bezeichnet:

```
    X-Koordinate_Z-Koordinate_.bf
```

Diese für manchen etwas seltsame Form der Bezeichnung liegt in der Funktionsweise des Lese-Prozesses begründet, welcher die `split()`-Methode zum Aufteilen von Strings nutzt.

Split String:

```
    String fileName = blockFile.getName();      // fileName = "1_4_.bf"
    String[] temp;
    temp = fileName.split("_");                 // temp = {"1", "4", ".bf"}
    int x = new Integer(temp[0]);
    int z = new Integer(temp[1]);
```

I/O

Nun haben wir unser Terrain in Blöcke unterteilt und wollen diese auf die Festplatte schreiben und später von dort in den Arbeitsspeicher laden. In Java nutzen wir dafür die Java I/O Streams um unser Blöcke zu transferieren. Bei den zu transferierenden Daten handelt es sich um reine float Werte, hierfür empfiehlt sich die Benutzung des `DataInput-` bzw. `DataOutputStreams`, da diese für das Ein- / Auslesen von einfachen Datentypen konzipiert wurden.

Umsetzung:

```
DataOutputStream output = new DataOutputStream(FileOutputStream out);
DataInputStream input = new DataInputStream(FileInputStream in);
```

Nach dieser Initialisierung wird ein Block Objekt wie ein gewöhnliches Array mittels `for`-Schleifen durchlaufen und die `float` Werte ein- bzw. ausgelesen. Auf die gezeigte Weise funktioniert der Datentransfer, nur dass er nicht sonderlich effizient ist. `DataInput-` bzw. `DataOutputStreams` sind sehr langsam, um diese Eigenschaft zu kompensieren, müssen die Streams gebuffert werden. Hierzu nutzt man `BufferedInput-` / `BufferedOutputStreams`, welche mittels stream chaining wie folgt eingebunden werden:

Umsetzung:

```
DataOutputStream output = new DataOutputStream(new BufferedOutputstream(FileOutputStream out));
DataInputStream input = new DataInputStream(new BufferedInputstream(FileInputStream in));
```

Schon durch diese kleine Abänderung lässt sich die I/O Perfomance immens steigern!

Probleme

Unsere I/O-Methoden laufen fehlerfrei und das ziemlich performant, dennoch konnte man hier da noch optimieren, speziell den Schreibprozess. Hierzu wurde der `ByteArrayOutputStream` in die `writeBlockData()`-Methode eingebunden:

Umsetzung:

```
try(FileOutputStream fos = new FileOutputStream(blockFile);
     ByteArrayOutputStream bos = new ByteArrayOutputStream();
     DataOutputStream output = new DataOutputStream(new BufferedOutputStream(bos)))
{
    /* write floats into ByteArrayOutputStream */

   fos.write(bos.toByteArray());    //write floats from ByteArrayOutputStream into FileOutputStream


    return blockFile;
}
```

Nach einigen darauffolgenden Testläufen zeigte sich jedoch, dass offenbar beim Schreibprozess einige Daten, ca. 1/10, verloren gingen. Der genaue Grund dafür ist uns unklar geblieben. Es ist uns nur bekannt, dass es mit dem `ByteArrayOutputStream` in Zusammenhang steht, weshalb dieser wieder entfernt wurde, da eine fehlerfreie Funktionalität der Performanz vorgezogen werden musste.

Ein weiterer Problemfall trat durch die bereits erwähnte Größe der Blöcke in Bezug auf die Gesamtgröße des Terrains auf.

Terrain Größe	Block Größe	Anzahl blockfiles
79.872 x 79.872	256 x 256	97.344
79.872 x 79.872	512 x 512	24.336
79.872 x 79.872	1.024 x 1.024	6.084
79.872 x 79.872	2.048 x 2.048	1.521

Hat man z.B. ein Terrain der Größe 79.872 * 79.872 Vertices, entspräche dies, im Falle der Standard Blockgröße, 97.344 `blockfiles` auf der Festplatte. Eine solche Menge von zu öffnenden Dateien führte zur Laufzeit zu einem `fatal error` seitens OpenGL. Um die Anzahl an Dateien zu verringern mussten diese dementsprechend vergrößert werden. Setzt man Block-Größe auf 2.048 * 2.048 Vertices, hätte man nur noch mit 1.521 `blockfiles` zu arbeiten. Durch diesen Aspekt wurde die Anwendung noch performanter, da das Ein- und Auslesen vieler kleiner Blöcke / Arrays mit häufigerem Wechseln der `file`-Objekte wohl mehr Rechenleistung zu benötigen scheint, als das weniger großer Arrays.

TerrainView

Dieser Teilabschnitt setzt sich mit der Datenweitergabe auseinander. Im Fokus stehen hier zum Einen die Größe der Datenmenge, die sich im lokalen Speicher befinden soll, zum Anderen die schnelle Beschaffung von Daten, die nicht im lokalen Speicher sind. Dabei können wir, durch den in Java vorhandenen Garbage Collector, die Freigabe von lokalem Speicherplatz außer acht lassen.

Datenmenge

Die Anzahl der Daten, die von Belang sind, sind von der Speicherung des Terrains in gleich große Blöcke abhängig. Die Datenmenge wird also von der Anzahl der Blöcke im lokalen Speicher definiert. Dabei wird darauf geachtet, dass nicht zu viele Daten weitergegeben wird, da der Arbeitsspeicher nur eine gewisse Menge verarbeiten kann, aber auch darauf, dass nicht zu wenige Daten vorhanden sind, da das Nachladen von Daten von der Festplatte die Performance beeinträchtigt.

Initialisierung der Datenmenge

In der TerrainView wird die Relevanz eines Blockes im Terrain durch die Kameraposition festgelegt. Das heißt, dass der Block, in dem die Kamera steht, definitiv angezeigt und somit im Speicher sein muss. Weiterhin ist es sinnvoll die Blöcke um den 'Kamerablock' herum im Speicher zu halten, da die Daten dieser Blöcke, je nach Größe der Blöcke (wieviel Vertices in einem Block) und wechseln der Kameraposition, von Interesse sind oder sein könnten.

Die TerrainView selber hat, bei einer Blockgröße von 65.536 Vertices (gespeichert in einem float[256][256][5]), 81 Blöcke im Arbeitsspeicher. Dabei stellt der mittlere Block den Kamerablock dar. Die ganzen Blöcke sind in einem zwei dimensionalen Block-Array gepeichert. (Hierbei ist zu beachten: Das Block-Array ist quadratisch und die Anzahl der Dimension ist ungerade, damit der Kamerablock sich weiterhin in der Mitte des Arrays befindet.) Der benötigte Speicherbedarf lässt sich leicht berrechnen:

81(Blöcke) * 256 * 256 (Anzahl der Vertices pro Block) * 5 (Werte pro Vertice) * 4 (Byte pro Float) = 106.168.320 (bytes) ~ 810 MB

In den folgenden Beispielen wird der Einfachheit halber von 25 Blöcken ausgegangen.

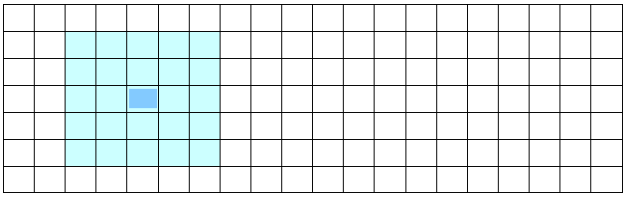


Bild 1: Ein Block-Array innerhalb eines Terrains

Im Bild 1 wird das Block-Array durch die hellblaue Farbe dargestellt und der Kamerablock mit der dunkelblauen Farbe.

Aktualisierung der Datenmenge

Nach der Initialisierung des Block-Arrays ist nun die Aktualisierung der Daten ein wichtiges Thema, da hier die Performance zur Laufzeit besonders von der Häufigkeit und Menge der neu zu holenden Daten abhängt. Dabei hängt das Auslesen der Daten von der Geschwindigkeit vom Austausch zwischen der Festplatte und dem Speicher ab. Die Ursache der Beschaffung der Daten ist hierbei der Wechsel der Kameraposition von einem Block in einen anderen Block.

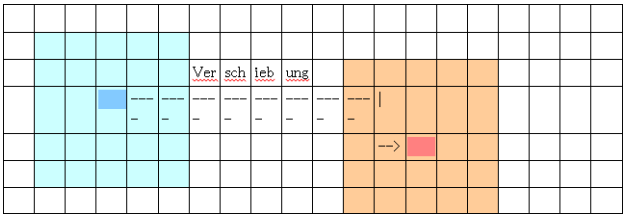


Bild 2: Verschiebung eines Blockes innerhalb des Terrains

Wie im Bild 2 zu sehen müsste das Block-Array aktualisiert werden, damit die Position der Kamera mit den wiedergegebenen Daten wieder übereinstimmt und der Kamerablock sich wieder in der Mitte des Block-Arrays befindet. Jeder Block im Block-Array muss also wieder initialisiert werden, was zur Laufzeit ein großer Zeitaufwand ist. Diesen Aufwand kann man an zum Teil einschränken. Hierzu gibt es in der TerrainView zwei von anderen Klassen und Implementierung unabhängige Möglichkeiten.

1. **Häufigkeit des Neuladen**
   Die Häufigkeit der Umsetzung hängt von der Anzahl der im Speicher stehenden Daten ab. Bei dem Block[9][9] in der TerrainView sind so viele Daten vorhanden, dass die Vertices am Rande des 9x9 Block-Arrays bei der Visualisierung durch das clippen nicht sichtbar sind. Dadurch wird das Neuladen von neuen Blöcken erst nötig, wenn die 'Sichtbarkeit' der Vertices über den Rand des Block-Arrays zu tragen kommt, genauer: Das Laden von neuen Blöcken muss erst ab einer gewissen Differenz von dem aktuellen Kamerablock zum vorherigen Kamerablock stattfinden.
2. **Schon im Speicher vorhandene Daten umsetzen anstatt neu auslesen**
   Der Abstand und die Richtung der Verschiebung in horizontaler (x) und vertikaler (z) Richtung zwischen dem alten Kamerablock und dem neuen Kamerablock ist hierbei vom großen Belang, denn je nach Größe des Block-Arrays kann es eine Überschneidung vom 'alten' Block-Array mit dem aktuellen Block-Array. Der Durchschnitt der Block-Arrays gibt an welche Blöcke nur eine neue Position innerhalb des Block-Arrays bekommen. Nur die Blöcke, die nur in dem einen, aber nicht in dem anderen Block-Array vorhanden sind werden neu geladen.

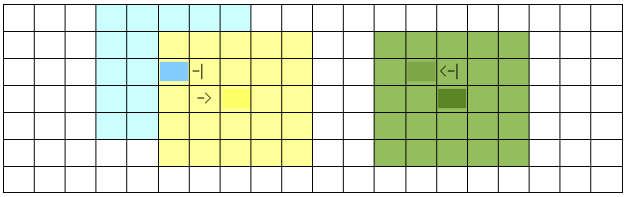


Bild 3: Ein Beispiel bei der Aktualisierung von Blau nach Gelb jedoch nicht bei Grün

Die Umsetzung von den Daten werden im Pseudocode(mit der in 1. beschriebenen Möglichkeit der Steigerung der Performance) und im Bild 4 dargestellt.

Pseudocode:

```
WENN aktueller Kamerablock vom vorherigen Kamerablock mehr als t Blöcke entfernt ist
  DANN
        SOLANGE Block im aktuelle Block-Array nicht gesetzt
                WENN Block im vorherigen Block existiert
                        DANN setze Block ins aktuelle Block-Array
                        SONST lade Block von Festplatte
```

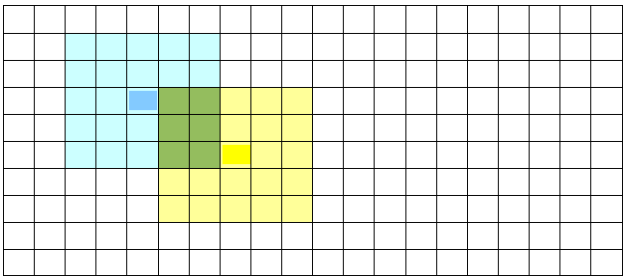


Bild 4: Grüne Blöcke werden einfach in den neuen Block umgesetzt, während die gelben neu geladen werden

Initialisierung und Aktualisierung außerhalb des Terrains

Ein noch zu beachtendes Problem ist die Initialisierung des Block-Arrays mit Blöcken außerhalb des Terrains. Bei der Terrainerzeugung werden nur die Blöcke auf die Festplatte geschrieben, die auch innerhalb des Terrains sind. Das bedeutet bei der Initialisierung der Blöcke kann kein Block außerhalb des Terrains eingelesen werden, da für diesen Block keine bestehende Datei auf der Festplatte existiert. Abhilfe des Porblems erfolgt hierbei durch ein sogenannten Dummy-Block mit 0.0f-Werten. Dieser bekommt die ID (-1,-1) im Terrain und liegt somit definitiv außerhalb und wird immer dann geliefert, sobald sich ein Block des Block-Arrays außerhalb des Terrains befindet, oder ein Block nicht von der Festplatte gelesen werden konnte. (Hierbei ist zu beachten: Wenn sich der Kamerablock außerhalb befindet, dann ist seine ID auch (-1,-1). Deshalb ist es nötig die richtige ID des Kamerablocks durch Variablen zu speichern.)

Weitergabe von Daten

Die Weitergabe von Daten ist durch das einfache Durchlaufen des Block-Arrays nur an die benötigten Daten für die nächsten Schritte (clippen,rendern und visualisieren) gebunden. Die Weitergabe kann durch ein float-Array, wie auch durch das Block-Array oder einen FloatBuffer erfolgen. Jedoch stellt die Aktualisierung der Daten ein großes Performance-Problem dar. Bei der Umsetzung des Block-Arrays müssten die Daten neu geholt und verarbeitet werden. Das bedeutet, das die aktualisierten Daten neu gerendert und angezeigt werden müssen, was den Schritt 1 und der dadurch resultierenden größeren Datenmenge bei der Aktualisierung um so wichtiger macht.

Laufzeit

Beim Testen der TerrainView ist die Aktualisierung beim Wechsel der Kamerablöcke zur Laufzeit anhand von kurzem Stocken (Millisekunden) zu beobachten. Dabei ist das erneute Rendern der Daten nicht mit einbezogen. Hierbei stellt sich die Frage, wie andere Java-Spielentwickler ihre Daten verwalten und effektiver mit diesen Daten umgehen (z.B.: Minecraft). Eine Möglichkeit wäre das Arbeiten mit Threads um mit der 'Nebenläufigkeit' größeres Stocken durch das Neuladen von Daten und das anschließende Rendern zu minimieren.

1.3 Clipmaps

- Einleitung
- Geometrie
- Bewegung und Aktualisierung
- Höheninformation und Vertexshader
- Artefakte und Probleme
- Fazit

Video unserer Implementation[4]

1.3.1 Einleitung

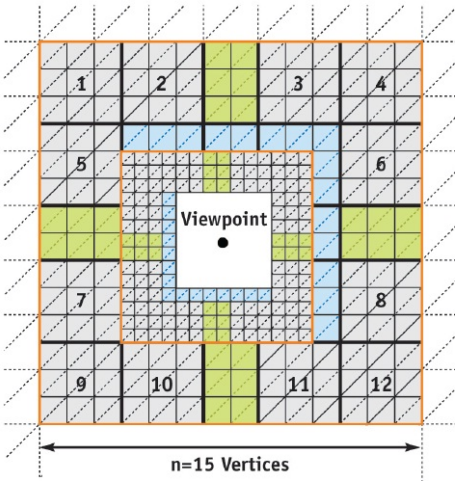
Sobald in der Computergrafik große weitläufige Landschaften dargestellt werden, ist es von Nöten sich Techniken zur Minimierung der Last der Grafikkarte sowie des Arbeitsspeichers zu überlegen. Eine Möglichkeit ist das implementieren einer Clipmap. Clipmaps verhalten sich ähnlich wie ein traditionelles Grid in der Landschaftsdarstellung, jedoch mit dem Unterschied dass die Vertex Anzahl vom Betrachter Standpunkt zu den Rändern hin abnimmt. D.h. im Zentrum dieser Map, um die Kameraposition herrscht die größtmögliche Auflösung. Danach folgt ein Ring in einer Auflösung die nur noch halb so groß ist wie im innersten Grid. Der nächste Ring hat dann nur noch ein Viertel der Initialen Auflösung. Bei unserer Implementierung ist es möglich beliebig viele Level of Detail anzulegen. Mit jedem Clipmapring (Level of Detail) verdoppelt vervierfacht sich dabei die Größe des gesamten Terrains, die Vertex Anzahl bleibt allerdings für jeden Ring konstant. Dadurch kann man bei gleicher Vertex Anzahl wesentlich größere Terrains darstellen als mit herkömmlichen Grids. Natürlich hört sich eine kontinuierliche Halbierung der Auflösung pro Level of Detail nach einem drastischen Verlust der Bild- und Terrainqualität an; man darf jedoch nicht vergessen, dass die Kameraprojektion die Vertices mit steigendem abstand sowieso staucht, der sich verringernden Auflösung also entgegen wirkt.

Die Implementation dieser Technik war unsere Aufgabe während dieses Praktikums. Wir haben größtenteils nach dem Paper von Arul Asirvatham und Hugues Hoppe gearbeitet:
http://research.microsoft.com/en-us/um/people/hoppe/gpugcm.pdf

1.3.2 Geometrie

Bei einer Clipmap handelt es sich grundlegend um eine quadratische Ringgeometrie, die ein Level-of-Detail(LoD) darstellt. Dieser Ring wird einmal aus 4 verschiedenen Geometrien zusammengesetzt:

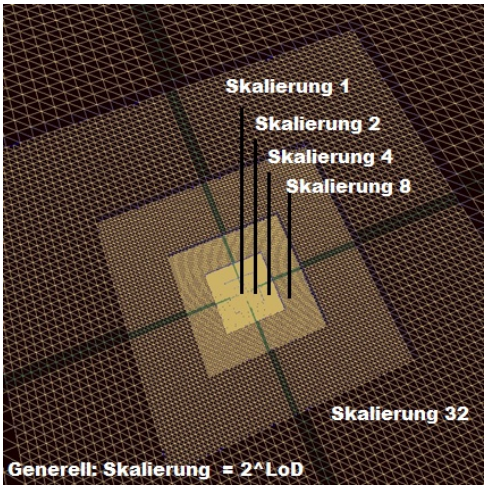
- Quadrat
  - Im Bild rechts die Felder 1 bis 12, grau eingefärbt. Das Quadrat stellt die größte Fläche der Clipmap.
- Rechteck
  - Im Bild rechts grün eingefärbt. Notwendig um Lücken zu schließen.
- L-Geometrie
  - Im Bild rechts blau eingefärbt. Dient zur Umsetzung der Clipmapbewegung.
- Outer Degenerated Triangles
  - Orange. Dient zur Vermeidung von Lücken zwischen zwei LoDs.



4 http://www.youtube.com/watch?v=49ozyJuC08w&feature=youtu.be

Hier sieht man gut, dass sowohl die grüne als auch die blaue Geometrie zum Füllen von Lücken dient. Die blaue, L-förmige Geometrie hat dabei insbesondere die Aufgabe, sich abhängig von der Bewegung des Beobachters neu zu positionieren.

Um den Betrachterstandpunkt herum befindet sich ein initiales Grid. Dieses Grid stellt die höchstmögliche Auflösung des Terrains dar: Detaillevel 0. Um dieses Nulllevel werden dann die Clipmapringe gezeichnet. Wir benutzen dafür immer die gleiche Ringgeometrie, die in unserer Anwendung mit der Methode createClip() erzeugt wird. Einzig der Skalierungsfaktor wird für jedes LoD verdoppelt. Dadurch erreicht man bei gleichbleibender Vertexzahl mit jedem weiteren Ring eine vervierfachung der Fläche und die für Clipmaps charakteristische Abnahme des Details durch sich verdoppelnde Vertexabstände entsteht.
Der Skalierungsfaktor wird hierbei als uniform Variable an den Vertexshader übergeben.



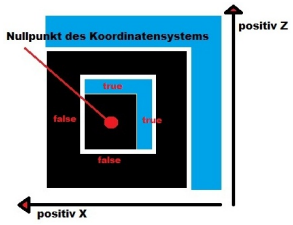

```
for (int i = 0; i < stage; i++)
{
    if (i == 0) // Wenn Stage == 0, schreibe innerstes Grid
    {
        Util.mul(translation,
                Util.translationX(2 * (-gridsize - middlesize) + middlesize + movement[0][0], null),
                Util.translationZ(-2 * gridsize + movement[0][1], null));
        setScale(scaleFaktor);
        setProgram();
        center.draw();
        outer.draw();
    } else
    // Zeichne ClipMap Ring
    {
        setScale(scaleFaktor * (float) pow(i));
        createClip(i);
        setLGrid(i);
    }
}
```

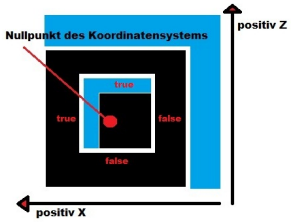
1.3.3 Bewegung und Aktualisierung

Da die Kamera sich mit konstanter Geschwindikeit im Raum bewegt, muss dafür gesorgt werden, dass die ClipMap sich passend aktualisiert, das höchste Detaillevel also unter dem Betrachter positioniert bleibt. Weiterhin muss jedoch der Aufbau der Clipmap aus mehreren, sich nicht überscheidenden Ringen, gewahrt bleiben. Im Folgenden erläutern wir unsere Implementation des Bewegungsablaufes und die zu Grunde liegende Logik.

Initial wird die Clipmap um den Nullpunkt des Koordinatensystems gezeichnet, sodass die L-Förmige Füllgeometrie in der rechten oderen Kante liegt. Das machen wir, um einen eindeutigen Wert für die Lage jedes Rings in der Clipmap zu bekommen. Dabei speichern wir jedoch nicht die Koordinaten ab, an denen der Ring liegt, sondern für jede Ebene, wo die, später durch die L-Geometrie gefüllte Lücke liegt. Wir verwenden also ein 2D-Array, das an erster Stelle die betreffende Ebene speichert und an zweiter für alle 4 Bewegungsrichtungen ein true(Platz vorhanden) bzw. false(kein Platz vorhanden). Das hat den Vorteil, dass wir dieses Array sowohl für die Platzierung der Geometrien verwenden können, als auch für die lageabhängige Verschiebung der Clipmap.

In den beiden Bildern wird die Speicherbelegung eines Bewegungsvorganges nach Rechts, also negativ X, dargestellt. Zu Beginn steht für die Rechtsbewegung für das innerste Grid im Array ein true, d.h. man kann das Grid verschieben, ohne das die äußeren Ringe der Clipmap beeinflusst werden. Nachdem dies geschehen ist, wird der Zustand sowohl für die positive, als auch für die negative X-Richtung gewechselt. Eine weitere Bewegung nach rechts wird also nur möglich wenn man den nächsten Ring ebenfalls verschiebt.





Erläuterung:
Die boolschen Werte stehen für die aktuelle Belegung des erwähnten Arrays.

Die Verschiebung der äußeren Ringe geschieht bei uns durch Rekursion. Wir haben eine Methode erstellt, bei der die Bewegung immer ausgehend von der nullten Ebene startet. Wenn Platz vorhanden ist, wird das innerste Grid verschoben. Ist kein Platz vorhanden, wird das innerste Grid verschoben, danach ruft sich die Methode selber mit der nächsten Ebene auf únd Prüft so alle Ebenen durch. Ist in der nächsten Ebene nun also Platz, wird diese ebenfalls verschoben und die Rekursion terminiert. Ist dort ebenfalls kein Platz wiederholt sich das Prozedere, bis ein Level mit Platz in die Richtung erreicht ist, oder der letzte und äußerste Ring erreicht ist.

```
if (i == stage - 1) // Abbruchbedingung wenn der äußerste
                    // ClipMapRing erreicht ist
{
    movement[i][dir] += 2;
} else
{
    if (alignment[i][dir]) // Wenn noch "Platz" zum bewegen ist
    {
        movement[i][dir] += 2;
        alignment[i][dir] ^= true;
        alignment[i][dir + 2] ^= true;
    } else
    // Ansonsten schiebe rekursiv alle ClipMapRinge in die
    // Bewegungsrichtung bis Platz ist oder äußerster Ring erreicht
    {
        alignment[i][dir] ^= true;
        alignment[i][dir + 2] ^= true;
        movement[i][dir] += 2;
        moveClip(i + 1, dir);
    }
}
```

Ein weiterer Vorteil unserer Art der Speicherung ist der, dass wir anhand der true-false-Belegung eines jeden Level die Füllgeometrien setzen können.

```
/** Erzeugt die für eine ClipMap benötigten L-Geometrien und zeichnet diese.
private void setLGrid(int i)
{

    int side = 0; // Lage der L Geometrien, abhängig von der Lage bestimmen
    if (alignment[i - 1][0] && alignment[i - 1][1])
    {
        side = 2;
    } else if (alignment[i - 1][1] && alignment[i - 1][2])
    {
        side = 1;
    } else if (alignment[i - 1][2] && alignment[i - 1][3])
    {
        side = 4;
    } else if (alignment[i - 1][3] && alignment[i - 1][0])
    {
        side = 3;
    } else
        throw new IllegalStateException("L Grid kann nicht gesetzt werden");
    // 1 = TopRight
    // 2 = TopLeft
    // 3 = BottomLeft
    // 4 = BottomRight
    switch (side)
    {
```

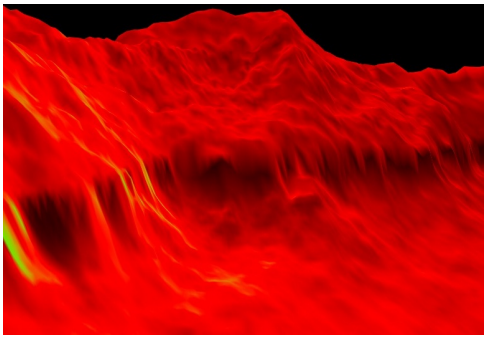
Um die berechnete Bewegung auf die Clipmap zu übertragen ist ein weiteres Array von Nöten, dass die Translation in X- und Z-Richtung für jeden Ring speichert. Dieses Array ist direkt in die Translationsmatrizen der Geometrien eingebunden und bei jedem Drawaufruf wird entsprechend der angegebenen Werte translatiert.\\\

1.3.4 Höheninformation und Vertexshader

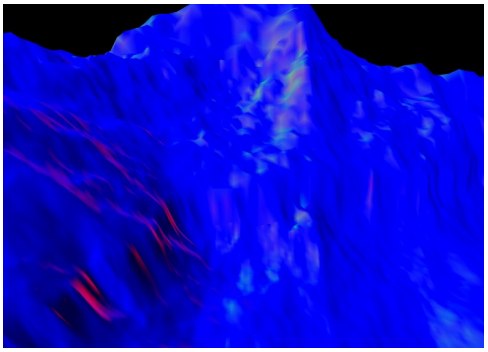
Im vorherigen Kapitel haben wir erläutert, wie wir die Geometrie der Clipmap angelegt, sowie deren Bewegung implementiert haben.

Im Folgenden werden wir nun darauf eingehen, wie wir die Clipmap mit Höheninformationen versehen, um ein plastisches Terrain entstehen zu lassen.
Anders als in der Vorlesung gelernt, werden die einzelnen Vertices nun nicht mehr, schon bei Erstellung in der Applikation mit einer Höheninformation versehen, sondern bekommen ihre Höheninformationen dynamisch im Vertexshader aus einer Höhentextur zugewiesen.
Die Höhentextur wird Initial beim Starten der Applikation aus den Daten der Terraingenerierung erzeugt. Diese übergeben uns über eine Schnittstelle ein zweidimensionales Array, das jeder Stelle [X][Z] die spezifische Höhe zuweist. Das Array wird danach zeilenweise in einen Floatbuffer gepackt, den man wiederrum vom Vertexshader als Textur interpretieren lassen kann.
Man kann auf diese Textur genauso wie im Fragmentshader mit texture(hightTexture, texCoords) zugreifen.
Das Festlegen der Texturkoordinaten stellte sich jedoch als schwierig heraus. Wir haben drei verschiedene Ansätze verfolgt.\\ Zum einen den, die Texturkoordinaten schon in der Applikation zu bestimmen und so jedem Vertex zusätzlich eine Texturkoordinate in X -Richtung und eine Texturkoordinate in Z-Richtung zuzuweisen. Diese Vorgehensweise hätte jedoch einen der Vorteile der Clipmap, das relativ speichergünstige Übertragen von Daten, zunichte gemacht. Außerdem ändern sich die Texturkoordinaten ständig mit den Weltkoordinaten des Vertices, was ebenfalls für eine Berechnung der Texturkoordinaten im Vertexshader spricht.
Unser zweiter Einfall war der, die Texturkoordinaten auf die Größe der Clipmap anzupassen, sodass, egal wie groß die Höhentextur aufgelöst ist, die Clipmap jedes Mal genau komplette Höhentextur darstellt. Das hat natürlich zur Folge, dass entweder Informationen verloren gehen (Texturauflösung > Clipmapauflösung) oder mehrere Vertices die gleichen Informationen haben (Clipmapgröße > Texturauflösung).
Letzten Endes entschieden wir uns dafür, die Texturkoordinaten an die Textur anzupassen, sodass gewährleistet ist, dass ein Vertex immer genau einen Punkt in der Textur wiederspiegelt. Dies erreichen wir indem wir die posWC.x und posWC.z eines jeden Punktes einfach durch die Größe der Höhentextur teilen. Die Größe wiederrum kann man sehr einfach im Vertexshader durch den Befehl textureSize(TexturXY) herausfinden.
Natürlich hat dieser dritte Ansatz, der auch den Einzug in unsere Implementation geschafft hat, den Nachteil, dass die dargestellte Welt, abhängig von der Texturgröße sehr klein ist. Da wir es nicht mehr geschafft haben, ein vernünftiges Datenmanagement einzubinden, sind unter Einsatz von 12GB Arbeitsspeicher Texturgrößen von 8192x8192 möglich. Dadurch, dass die Texturkoordinaten jedoch auf die Texturgröße angepasst werden, sieht das dargestellte Terrain mitunter trotzdem relativ klein aus. Dem kann man entgegenwirken, indem man in der Applikation mit dem Befehlen *"GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_WRAP_S, GL14.GL_MIRRORED_REPEAT)"* und *"GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_WRAP_T, GL14.GL_MIRRORED_REPEAT)"* arbeitet, die dafür sorgen, dass die Textur an den Rändern gespiegelt wiederholt wird.

Eine weitere Aufgabe unseres Vertexshaders bestand darin, für die Nachfolgenden Darstellungsgruppen Tangenten und Bitangenten eines jeden Punktes auszurechnen. Die Tangente gibt im Prinzip nichts anderes als die Steigung eines jeden Punktes in X-Richtung an. Die Bitangente wiederrum die Steigung in Z-Richtung. Tangenten werden also einfach durch das Kreuzprodukt eines Einheitsvektors in X-Richtung und der Normale des Jeweiligen Punktes errechnet. Die Bitangenten dann aus dem Kreuzprodukt von Tangente und Normale.



Visualisierung der Tangenten. Rotverfärbung charakteristisch, da die Steigung in X-Richtung angegeben wird und im Rotkanal immer eine 1 steht.



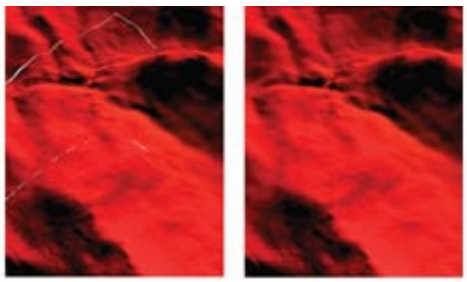
Visualisierung der Bitangenten. Blauverfärbung charakteristisch, da die Steigung in Z-Richtung angegeben wird und im Blaukanal immer eine 1 steht.

1.3.5 Artefakte und Probleme

1. Höheninformationsverlust zwischen 2 LoDs
2. Artefakte

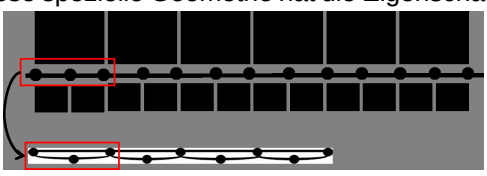
Höheninformationsverlust zwischen 2 LoDs

Sobald man eine Höheninformation in seiner Clipmap eingebaut hat kann es zwischen den LoDs zu Artefakten kommen. So teilt sich z.B ein LoD nur jeden zweiten Vertex mit dem nächst detailierteren LoD. Es können demnach Lücken in der Geometrie entstehen wenn diese Vertices unterschiedliche Höhen haben.



Lösung

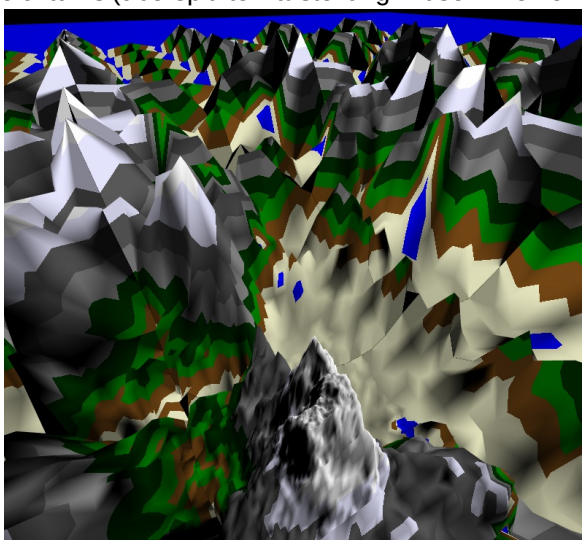
Mit dem einbinden einer weiteren Geometrie, den "degenerate outer triangles", die genau zwischen 2 LoDs platziert wird, werden diese Lücken trianguliert und geschlossen. Diese spezielle Geometrie hat die Eigenschaft, dass sie genau zwei innere Vertices mit einem äußeren verbindet.



Rauhe Übergänge

Rauhe Übergänge, bzw springende Artefakte treten erst bei der Bewegung im Raum auf. Man kann sich vorstellen, dass durch die Abnahme der Auflösung einige Höheninformationen verloren gehen. Hat man z.B. drei Vertices von denen die äußeren beiden auf einer Höhe liegen, der innere jedoch deutlich höher, so wird im groben LoD das Ganze als ebene Fläche dargestellt. Wird das feinaufgelöste Grid der Clipmap jedoch über diese stelle geschoben, so *"poppt"* der innere Vertex auf und es gibt einen unschönen Übergang.

Beispiel: Abnahme des Detailreichtums (überspitzte Darstellung mit sehr kleinem nullten Grid und vielen LoDs):



Lösung

Die Artefakte kann man nicht verhindern, man kann lediglich einen flüssigen, nicht auffälligen, Übergang schaffen. Für diesen Übergang wird im Vertex Shader ein alpha blending Wert berechnet und mit diesem die Höhe der aktualisierten Vertices interpoliert.

1.3.6 Fazit

Die drei Wochen haben uns einen guten Einblick in die Welt der Computergrafik, insbesondere der dahinter stehenden Datenverwaltung gewährt. Für uns war es das erste Mal, dass wir in unserer Art des Programmierens merklich auf die Performanceauswirkungen achten mussten. Von anfänglichen Speicherüberläufen durch ständiges Neuzeichnen und Neuanlegen von Geometrien bis hin zu Heapspace Erhöhungen auf bis zu 12GB um 8192x8192 Terrain zu ermöglichen.
Unsere Arbeit wurde dadurch entlohnt, dass es der Terrainerstellungsgruppe nach Einbindung unserer Map möglich war 4096x4096 Terrains bei flüssigen Frameraten darzustellen, wo vorher nur 2048x2048 darstellbar war. Weiterhin sind die Gewinne im Arbeitsspeicher deutlich festzustellen. Asirvatham und Hoppe sprechen von einer Verringerung um Faktor 100. Leider war es uns letzten Endes nicht möglich unsere Implementation in diesem Umfang zu testen, da es keine Funktionierende Datenhaltung gab.

Video unserer Clipmap: http://www.youtube.com/watch?v=49ozyJuC08w&feature=youtu.be

1.4 Modellierung

- Einleitung
- Voraussetzungen
- Schwierigkeiten
- Sackgassen
- Einzelne Modelle
  - Palme
  - Bäume und Arbeitsvorgang
  - Kaktus
  - Tanne
  - Pilze, Steine und Blumen
- Gallerie

1.4.1 Einleitung

Die leere Fläche unseres Terrains sollte mit 3D-Modellen von verschiedenen Pflanzen gefüllt werden. Die Modellierung dieser Pflanzen habe ich in Blender 2.6 vorgenommen, das Erstellen und Bearbeiten von Texturen in Adobe Photoshop CS2.

1.4.2 Voraussetzungen

Ein Ziel war es, für jedes Biom des Terrains mindestens ein Modell zu erschaffen. Eines der Biome jedoch war Tiefschnee, und statt einer Pflanze erstellte ich hierfür verschiedene Steine.

Da es aus Zeitgründen wahrscheinlich nicht möglich werden würde, viel mehr als ein Modell für ein Biom zu erstellen, bemühte ich mich bei allen Modellen darum, ihnen keine allzu herausstechenden Merkmale zu geben. Dies sollte bewirken, dass selbst wenn derselbe Baum viele Male gesetzt wird, man ihn nach entsprechender Skalierung und Rotierung doch nicht sofort als denselben erkennt.

Nach dem Fertigstellen der Modelle wurden diese ins OBJ-Format exportiert.

1.4.3 Schwierigkeiten

Ich hatte keinerlei vorherige Erfahrung mit Blender, oder 3D-Modellierung überhaupt, und verbrachte einen Teil der ersten Woche damit, mir die Steuerung und Grundlagen anzueignen. Die Blender-Dokumentation war jedoch noch nicht vollständig auf 2.6 umgestellt sondern verwies an manchen Stellen noch auf Version 2.4, und während ich dort alle wichtigen Dinge finden konnte, ließen sich einige speziellere Aspekte des Programms dort nicht nachlesen.

Auch verbrachte ich viel Zeit damit, frei nutzbare Texturen im Internet zu finden, die sich für meine Zwecke eigneten. Waren Baumrinden noch leicht zu finden, so ließ sich dasselbe nicht über Blüten, Äste und Blätter sagen. Oft waren die gefundenen Texturen im JPEG-Format, sodass ich manuell in Photoshop die entsprechenden Stellen des Bildes transparent machen musste, um es dann wieder als PNG zu speichern.

Manche Texturen waren komplett unauffindbar in einer für mich nutzbaren Form, sodass ich sie selber erstellte. Manchmal nahm ich auch eine gefundene Textur, die einer Rinde zum Beispiel, und machte sie randlos, sodass sie über einen ganzen Baumstamm verteilt werden konnte ohne Ränder zu bilden, da, wo die einzelnen Wiederholungen der Textur aufeinanderstießen.

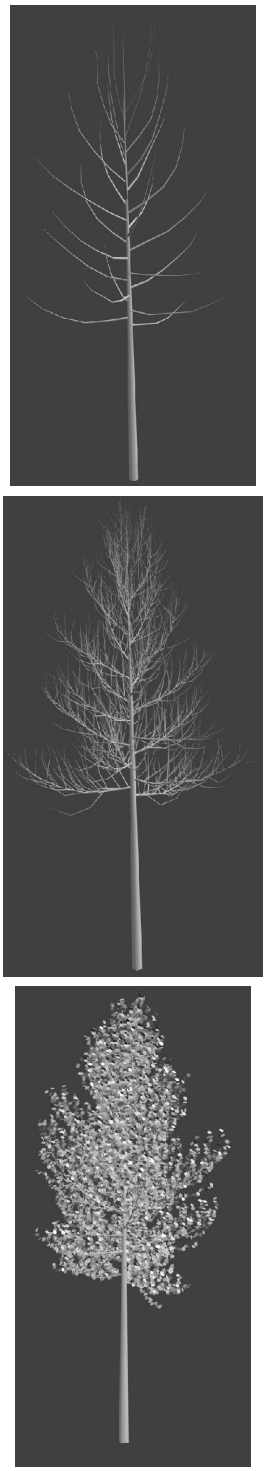
Beispiel

Original-Textur:

Randlose Version:

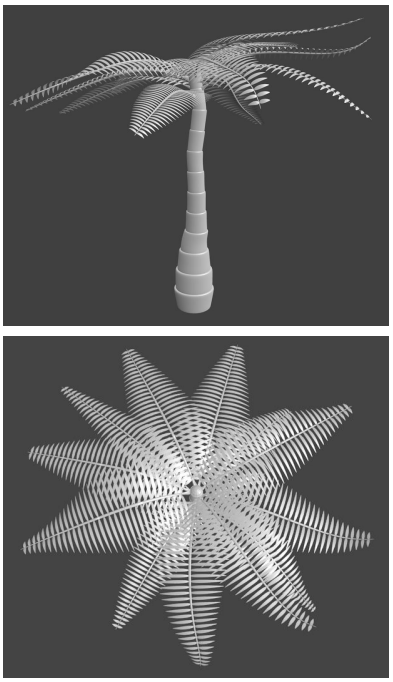
1.4.4 Sackgassen

Erst dachte ich, meine Arbeit würde sich als sehr kurz erweisen, denn Blender besitzt ein Add-In namens Add Sapling. Dieses nutzt Curves um sehr lebensechte Bäume zu modellieren. Der Nutzer muss nur die Parameter verändern und erhält so sehr schnell einen fertigen, wenn auch noch untexturierten, Baum. Während dieses Tool auf den ersten Blick sehr nützlich erschien, merkte ich jedoch schnell, dass die damit erstellten Bäume viel zu polygon-intensiv waren (>25000 Polygone). Ein einziger Baum verlangsamte Blender stark, die Vorstellung viele davon in unserem Terrain zu platzieren erschien mir unrealistisch.



Jeder Ast wird mehrfach geknickt, jedes einzelne Blatt hat 4-6 Vertices.

Ich orientierte mich also nun sehr stark an Tutorials, die ich im Internet fand. So erstellte ich eine Palme, die zwar hübsch anzuschauen war, mit jedem Blatt einzeln modelliert, jedoch tausende von Polygonen besaß.



Jedes Blatt hat mehr als 300 Polygone.

Nach einer Besprechung mit den Tutoren, bei der wir feststellten, dass dies eindeutig zu aufwendig sein würde, suchte ich nun nach Wegen, die Pflanzen so sparsam wie möglich zu halten. Da uns nicht genau klar war, wie viele Polygone ein einzelner Baum nun haben konnte, erstellte ich Pflanzen mit unterschiedlichen Mengen.

1.4.5 Einzelne Modelle

In dieser Kategorie folgt nun eine Auflistung der Modellarten, die ich erstellt habe, und eine kleine Erklärung der Arbeitsprozesse. Auch auf besondere Schwierigkeiten bei einzelnen Modellen wird eingegangen.

1.4.5.1 Palme

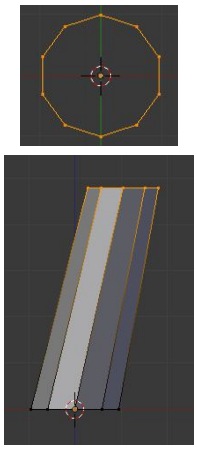
Mein erstes wirkliches Modell, dass auch später platziert werden würde, war eine Palme.

Hier hielt ich mich sehr stark an ein Youtube-Video, in dem jemand im Zeitraffer eine ähnliche Palme erstellte, leider ohne Anleitung. Die Texturen sind dem Internet entnommen.

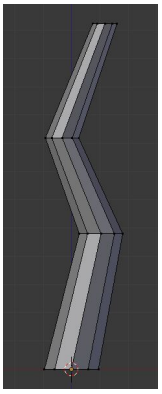
1.4.5.2 Bäume und Arbeitsvorgang

Als nächstes folgten einige Bäume, erstellt nach diversen Tipps und Tricks die ich Online gefunden hatte. Der generelle Arbeitsvorgang ist bei allen ähnlich.

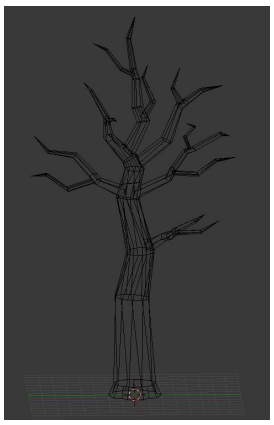
Nach einer ausgiebigen Recherche darüber, wie der geplante Baum in der Natur aussieht, beginnt die Modellierung. Erst wird ein Kreis aus 10 Vertices erstellt, dieser wird dann langgezogen, sodass ein Zylinder entsteht.



An einigen Stellen wird er aufs neue extended (langgezogen), was Knicke entstehen lässt.

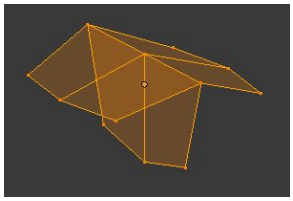


Dieser Zylinder bildet den Baumstamm, und die Breite wird durch den Baumtyp bestimmt. Mit derselben Methode werden Äste erstellt, diese haben jedoch häufig nur 5 Vertices als Grundkreis, aus Gründen der Sparsamkeit.

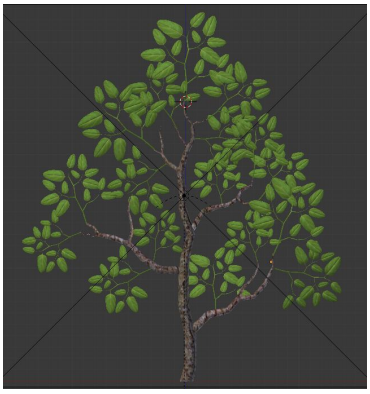


Habe ich das Gefühl, dass es genug Äste sind, beginne ich mit der Texturierung des Stammes.

Danach setze ich eine einfache Plane (Fläche) in den Raum, knicke diese in der Mitte und ziehe die drei mittleren Vertices nach unten.



Dies erweckt den Eindruck von Volumen im Blattwerk. Die Texturierung dieser Fläche erfolgt auf zwei Arten. Entweder ich nehme einen fertigen kleinen Ast mit Blättern, den ich im Internet gefunden habe, oder ich modelliere einen solchen Ast in Blender.



Nach fertiger Modellierung rendere ich diesen und speichere das Ergebnis im PNG-Format.

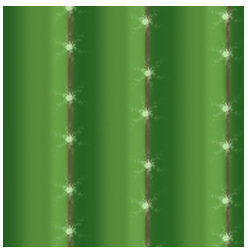
Habe ich nun eine fertige Textur durch eine der beiden Methoden, wird diese mithilfe von UV-Mapping auf die Fläche gelegt.

Danach wird sie dupliziert und an den Baum gelegt. Hierbei skaliere und rotiere ich die beiden Duplikate, um für Varietät zu sorgen. Die Menge der Duplikate entscheidet sich durch das Aussehen des Baumes und die Menge seiner Polygone.

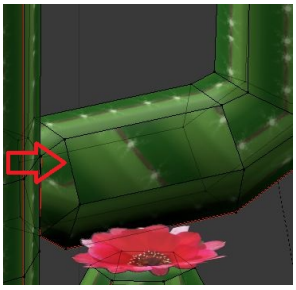
Diese Methode habe ich sowohl für die Palme und Bäume als auch für die Büsche verwendet.

1.4.5.3 Kaktus

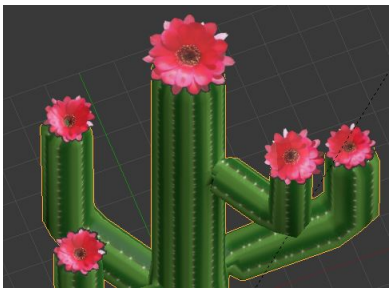
Nachdem ich nun genug Wissen in Blender gesammelt hatte, konnte ich anfangen, Modelle frei von jeder Anleitung zu erstellen. Das erste dieser Art war ein Kaktus. Während das Modell schnell erstellt war, und den Mustern der vorherigen folgte, erwies sich das Texturieren als sehr aufwendig aufgrund der Natur der Textur.



Diese hatte ich selber in Photoshop erstellt, und die geraden Linien erforderten ein exakteres Anbringen als bei den vorherigen Modellen, sodass ich bei vielen Polygonen die Textur manuell verschieben, rotieren und skalieren musste.



Die Spitzen wurden mit einer Blütentextur aus dem Internet dekoriert.

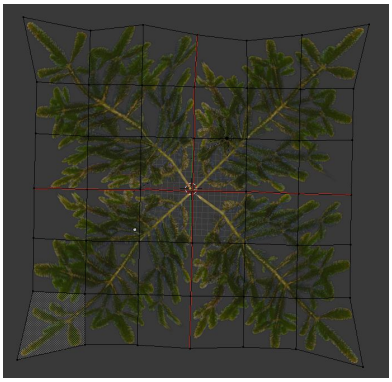


1.4.5.4 Tanne

Die Tanne zu erstellen war etwas trickreich, und ich musste ein bisschen rumprobieren, bis ich mit meiner Methode zufrieden war.

Die Basis stellte wieder ein Zylinder mit Knicken dar.

Im Internet fand ich eine schöne Textur für einen Tannenast. Ich mappte diese viermal auf eine Art Plane mit einem Loch in der Mitte.



Diese Plane wurde mehrmals geknickt, und dann verteilte ich einige Duplikate davon überall auf dem Stamm. Dann nutzte ich die Knicke, um diese Planes zu verformen, sodass nicht jede gleich aussah.

So entstand eine Tanne.

1.4.5.5 Pilze, Steine und Blumen

Die Pilze sind entstanden, weil sie relativ sparsam sind und sich sehr gut viele von ihnen auf einer Fläche verteilen lassen. Auch sie haben selbst erstellte Texturen.

Die Blumen und Steine sind ebenfalls simple Modelle. Die Blumen besitzen dieselben Zylinder wie die Baumstämme, auf der Spitze eine Blüte. Die Blätter sind wieder eine einfache Plane.

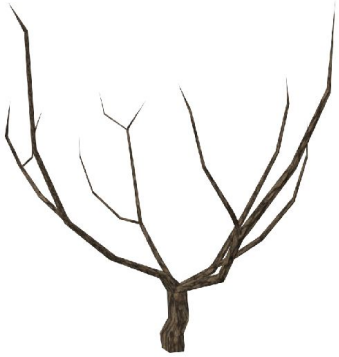
1.4.6 Gallerie

Hier noch eine Gallerie, die alle meine Modelle umfasst.

Steine

Blumen

Büsche



Pilze

Bäume

Anmerkung: Die Birke hat einen blauen Hintergrund, da sie mit weißem sehr schwer erkennbar wurde.

1.5 OBJ Importer

Einleitung

Das Programm zur Darstellung des Terrains verwendet zur Darstellung der Vegetation in Blender erstellte Modelle. Dies werden von Blender im OBJ-Format bereit gestellt und müssen mit Hilfe eines Importers in das Programm zur Darstellung des Terrains importiert werden. Im Folgenden soll die Funktionsweise eines solchen Importers beschrieben werden.

Aufbau und Inhalt einer OBJ- und MTL-Datei

Ein vollständiges Model besteht in der Regel aus mehreren klar unterscheidbaren Teilen. So hat eine Palme einen Stamm und mehrere Blätter. Stamm und Blätter haben sowohl verschiedene Form, als auch verschiedene Materialeigenschaften, was von Blender in zwei verschiedenen Formaten abgebildet wird, die miteinander verknüpftf werden. Das OBJ-Format stellt hierbei alle Daten zur Darstellung der Form bereit. Dazu gehören in der Regel Vertexkoordinaten, Texturkoordinaten und Normalen sowie eine weitere Information, sogenannte Faces, die Informationen darüber enthalten wie die einzelnen Vertices, Texturkoordinaten und Normalen miteinander zu Dreiecken zu verknüpfen sind.

Da in einer OBJ-Datei die Daten für jedes Teilmodel des Gesamtmodels enthalten sind, werden alle Daten, die zu einem bestimmten Teil des Models gehören, in einem Block zusammengefasst der Zeilenweise wie folgt aufgebaut ist (Pseudocode):

```
v 0.123 0.234 0.345 1.0 (Vertexkoordinaten)
vt 0.500 -1.352 0.234 (Texturkoordinate)
vn 0.807 0.001 0.707 (Vertexnormale)
f 1 2 3 (Face, enthält nur Indizies für Vertexkoordinaten)
f 3/1 4/2 5/3 (Face, enthält nur Indizies für Vertexkoordinaten und Texturkoordinaten)
f 6/5/1 3/5/3 7/6/4 (Face, enthält Indizies für Vertexkoordinaten, Texturkoordinaten und Normalen)
usemtl myMaterialName (Materialname, verweis auf MTL Datei)
```

Wärend in der OBJ-Datei die Form des Models gespeichert wird, werden in der zugehörigen MTL-Datei die Materialeigenschaften des Models gespeichert. Ähnlich wie bei einer OBJ-Datei werden dazu alle Eigenschaften eines Teilmodels in einem Block zusammengefasst, der die verschiedenen Informationen zeilenweise enthält. Auf diesen Block wird aus dem passenden Abschnitt der OBJ-Datei verwiesen. Er speichert Informationen, wie das anzuwendende Beleuchtungsmodell, die diffusen und spekularen Farbanteile, einen Dissolve-Faktor sowie verschiedene Texturvarianten. Beispielhaft sieht der Inhalt einer MTL-Datei folgendermaßen aus (Pseudocode):

```
newmtl myMaterialName
Ka 1.000 0.800 1.000
Kd 1.000 0.100 1.000
Ks 0.000 0.050 0.000
d 1.0
illum 3
map_Ka ka_map.jpg
map_Kd kd_map.jpg

map_Ks ks_map.jpg
map_Ns ns_map.jpg
map_d d_map.jpg
```

Auslesen der Dateien

Die Klasse GeometryFactory stellt eine Methode namens importFromBlender zur Verfügung. Sie stößt den Import Prozess eines Models an und bekommt den relativen Speicherort der OBJ-Datei, den relativen Speicherort der zugehörigen MTL-Datei, sowie den relativen Ordnerpfad der Texturdateien übergeben. Zurück liefert die Methode eine Liste, die Objekte vom Typ ModelPart enthält. Jedes ModelPart-Objekt enthält alle benötigten

Daten um ein vollständiges Teilmodel zu zeichnen, sodass es möglich ist über eine Iteration der von der Methode importFromBlender zurückgelieferten Liste ein vollständiges Model beliebig oft zu zeichnen.

Die Methode importFromBlender stößt dann die Methoden getMaterialListFromMTL und getModelPartListFromOBJ an.

Die Methode getMaterialListFromMTL bekommt den Pfad zur MTL-Datei des Models sowie den Ordnerpfad der Texturen des Models übergeben und liefert eine Liste, die Objekte vom Typ "Material" enthält, zurück. Objekte vom Typ Material fassen die aus der MTL-Datei ausgelesenen Daten zusammen. Dabei werden auch die verschiedenen Texturen geladen und im Material Objekt als Textur Objekt referenziert. Insbesondere zu bachten ist dabei der Materialname, über den später die Zuordnung zum Teilmodel aus der OBJ-Datei erfolgt. Das Auslesen der Daten erfolgt dabei pro Material zeilenweise. Begrenzer für die einzelnen Materialien ist der Materialname.

Die Methode getMaterialListFromMTL bekommt den Pfad zur OBJ-Datei, sowie die fertige Liste der Materialobjekte übergeben und liefert die Liste der ModelPart Objekte zurück, die letztendlich von der Methode importFromBlender gegenüber dem Anwender zurückliefert wird. Sie liest die OBJ-Datei ein und verfährt dabei ähnlich der Import Methode für die MTL-Datei zeilen- und blockweise für jedes Teilmodel des Gesamtmodels. Dabei werden zuerst alle Vertices, Texturkoordinaten und Normalen geparst und in einer Liste, die Objekte vom Typ Vector3f enthält, abgelegt. Anschließend werden alle Faces eingelesen und in einer Liste, die Objekte vom Typ Face enthält, abgelegt.

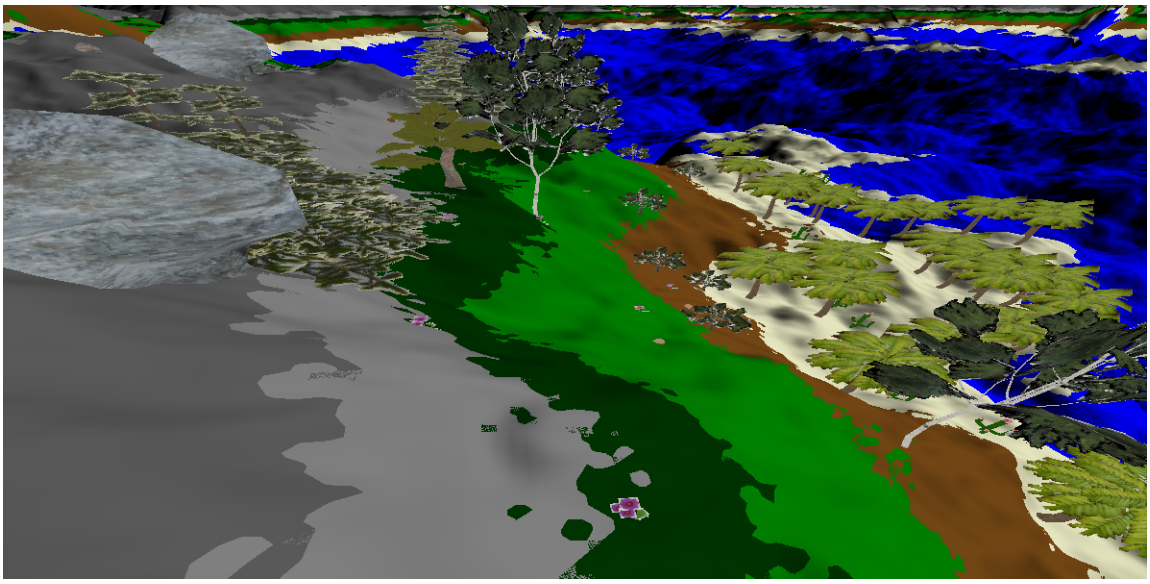
Ist ein Teilmodel fertig ausgelesen sodass alle Date in Listen vorliegen, wird die Methode createBuffers der Klasse PreStageModelPart aufgerufen. Sie bekommt die Liste der Faces, die Liste der Vertices, die liste der Texturkoordinaten sowie die Liste der Normalen übergeben und erstellt daraus in einem neuen Objekt vom Typ PreStageModelPart die zur Anbindung der Daten an die an die Grafikkarte benötigt Buffer. Dazu wird die Liste der Faces durchlaufen und Anhand der im Face gespeicherten Indizies die in den übrigen Listen gespeicherten Daten von der entsprechenden Stelle der Listen geholt und im den Buffer abgelegt.

Ist das PreStageModelPart Objekt fertig aufgebaut, das heißt alle Buffer erstellt, wird mit Hilfe der Methode createModelPart ein neues Objekt vom Typ ModelPart erstellt, das aus den Daten des PreStageModelPart Objekts ein neues Objekt vom Typ ModelPart erstellt. Diesem werden in dieser Methode noch Materialinformationen hinzugefügt. In dieser Methode wird zudem auch eine neues Objekt vom Typ Geometry angelegt. Die Klasse Geometry war bereits seitens der Tutoren vorgegeben und wird genutzt, um in dieser Methode die Vertex- und Indexdaten anzubinden sowie den Aufbau des Buffers bekanntzugeben.

Alle fertigen ModelPart-Objekte stellen nun die einzelnen Teile des Models dar und werden der Liste hinzugefügt, die letztendlich von der Ausgangsmethode importFromBlender gegenüber dem Anwender zurückgeliefert wird.

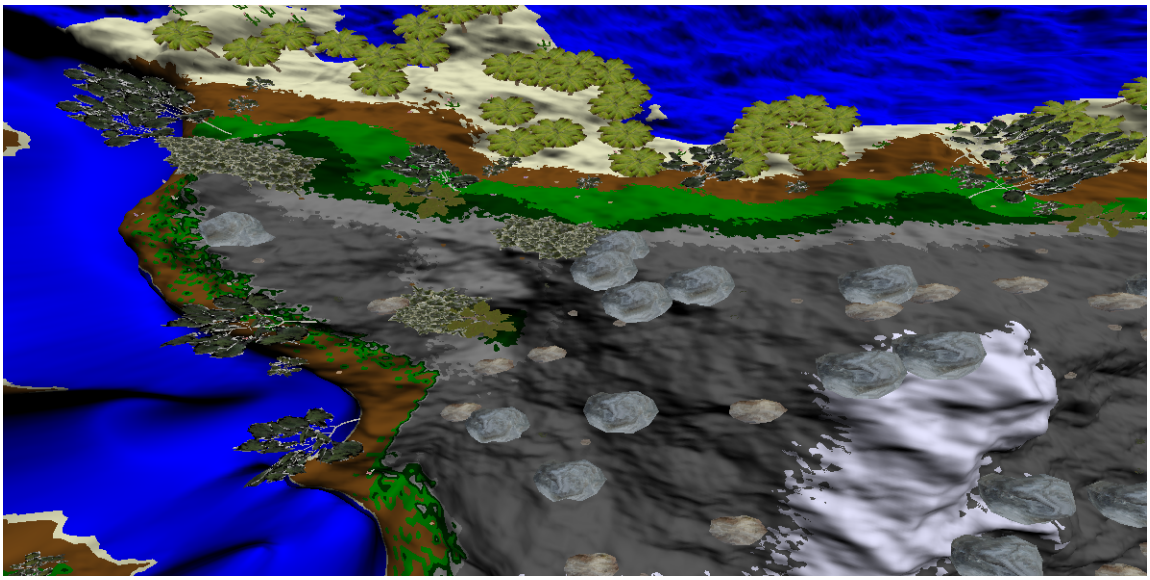
Anzeigen der Modelle

Neben dem Importieren der Daten war ebenfalls Aufgabe, die importierten Modelle geeignet auf dem fertigen Terrain zu verteilen. Um ein Model zu zeichnen, wird über die Liste, die die Teile des Models enthält, iteriert und die im ModelPart Objekt enthaltenen Daten jeweils an die entsprechenden Uniforms des Shaders angebunden.



Fertig gezeichnete Karte mit Modellen*(eigener Screenshot)*

Um eine Verteilung entsprechend des Untergrunds des Terrains zu ereichen, die eine zufällige Verteilung von verschiedenen Modellen auf dem jeweiligen Untergrund berücksichtigt, wird vor dem eigentlichen Zeichnen der Modelle eine Modell Karte mit Hilfe eines Arrays angelegt. In der Karte werden beim Erstellen mit Hilfe einer Zufallsfunktion die eine Gewichtung verschiedener Werte erlaubt, Nummern für die auf dem jeweiligen Untergrund zur Verfügung stehenden Modelle hinterlegt. Hier ist es auch möglich eine 0 zu vergeben, wenn kein Modell gezeichnet werden soll. Durch diese Möglichkeit sowie durch die Gewichtung entsteht ein einiger Maßen realistischer Eindruck.



Ansicht der Karte von oben. Zu sehen ist die verteilung nach Material und Zufall*(eigener Screenshot)*

Der Shader

Zur Anzeige der Modelle wurde ein einfacher Shader angelegt, der das Zeichnen der Modelle sowie eine rudimentäre Farbgebung übernimmt. Zu beachten ist hier insbesondere der Fragmentshader, der Transparenz von Texturen regelt. Ist der Dissolve Faktor des Materials ungleich 1 und der Alphakanal der Koordinate kleiner oder gleich dem Dissolve Faktor des Materials, so wird das entsprechende Fragment nicht gezeichnet. Das ermöglicht es mit hilfe von Texturtransparenz bspw. Palmblätter nur mit Hilfe eines Quadrats zu zeichnen, was extrem viele Triangles einspart.

Palme direkt nach Import. Gut zu sehen ist die Transparenz der Palmblätter*(eigener Screenshot)*

Sourcecode und Spezifikationen

- Der Sourcecode kann auf Github eingesehen und heruntergeladen werden: https://github.com/cgPrak12/Praktikum/tree/BlenderImportMerge2
- Die OBJ-Spezifikation kann hier eingesehen werden: http://www.martinreddy.net/gfx/3d/OBJ.spec
- Die MTL-Spezifikation kann hier eingesehen werden: http://paulbourke.net/dataformats/mtl/

2. Darstellung: Normal Mapping, Ambient Occlusion, Volumetric Light Scattering

2.1 Einleitung

Um effektiv mit Mapping-Methoden bzw. Post-Effects arbeiten zu können, ist eine Szene unumgänglich. Das lateinische „post“ weißt bereits auf den Zeitpunkt der Integration dieser Effekte in der Pipeline hin. Sie geschieht im nach hinein, wenn die Szene bereits gerendert als Rastergrafik vorliegt und sorgt bei performance-orientierter Programmierung, für ein deutlich Realismus-verstärkendes, optisches Fein-Tuning. Dabei reicht die Palette an Effekten von sogenannten God-Rays (Volumetric Light Scattering) über schlichte Blur-Effekte bis hin zu hochanspruchsvollen Volumetric Fog Artifacts, wie es die aktuelle Cry Engine 3 vorweisen kann. Im Folgenden gehen wir, im Rahmen unseres Projekts auf:

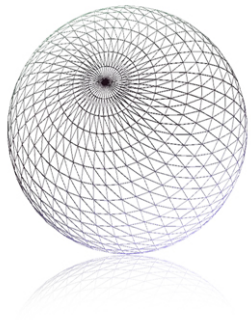
- Volumetric Light Scattering (God Rays)
- Screen Space Ambient Occlusion

ein und beschreiben deren Funktionsweise in Hinblick auf die realen Vorkommensweisen, wie auch die programmiertechnische Sicht in Codeform. Des Weiteren legten wir unsere Priorität auch auf eine für unsere Zwecke erfüllende Konstruktion einer Umgebungs-Atmosphäre. Um dies zu realisieren wählten wir das Konzept der SkyDome und der Wolkendarstellung via Wolken-Texturen mit Hilfe des „Blending“, dass uns im Rahmen der Computergrafik-Vorlesung näher gebracht wurde.

Präsentation

Unsere Präsentation haben wir bei Youtube als Video hochgeladen. Wir hoffen das Tempo ist nicht zu schnell.

2.2 Geometrie

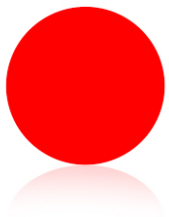


Da auch die Endversion unseres Projekts grundsätzlich aus einem 2-dimensionalen Grid besteht, findet auch bei unseren ersten Entwürfen und Versuchen ein einfaches Quad mit Textur Verwendung für eine konzeptionelle Szene. Es folgen Kugeln sowie Quader zur Repräsentation von Elementen in der Szene, um so, die Wechselwirkungen zwischen Gegenstand-Gegenstand und Gegenstand-Umgebung zu verstehen und zu implementieren.

2.3 Beleuchtung

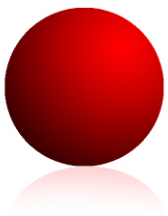
Zur Beleuchtung der Szene wurde eine Lichtquelle implementiert. Diese erhält eine initiale Position und wird dann via „Modelling Transformations“ ähnlich der Sonnenbewegung auf ihren Weg durch die SkyDome geschickt. Das implementierte Beleuchtungsmodell ist das Phong-Beleuchtungsmodell auf das im Folgenden kurz eingegangen wird um die verschiedenen Lichtarten näher zu erläutern.

2.3.1 Ambiente Beleuchtung



Das ambiente Licht ist mit einer Grundhelligkeit zu vergleichen, bei der nicht speziell von einer Lichtquelle ausgegangen wird, sondern die komplette Szene in eine gleichmäßige Grundhelligkeit getaucht wird. Dabei spielt die Position der Vertices keine Rolle, weshalb ambient beleuchtete Gegenstände auch während der Durchführung von „Modelling Transformations“ immer sichtbar sind (keine Schatten).

2.3.2 Diffuse Beleuchtung



Das diffuse Licht ist ähnlich der ambienten Beleuchtung. Auch die Berechnung läuft fast ähnlich ab, lediglich die Hinzunahme der Position des zu beleuchtenden Vertices und die Tatsache, dass die einfallenden Lichtstrahlen gestreut auf der Oberfläche reflektiert werden, bilden den entscheidenden Unterschied, der auch für den räumlichen Effekt verantwortlich ist.

2.3.3 Spekulare Beleuchtung

Das spekulare Licht ist ähnlich dem diffusen Licht, wobei die reflektierten Strahlen scharf in eine bestimmte Richtung fallen. Die spekularen Lichtanteile äußern sich vor allem im Glanzpunkt auf dem beleuchteten Gegenstand. Dieser resultiert aus dem Anteil des Winkels zwischen Oberflächen-Normale und dem ausfallenden Lichtstrahl. Ist Einfallswinkel gleich Ausfallswinkel ist der spekulare Glanzpunkt am stärksten sichtbar.

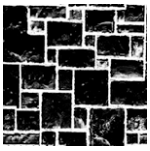
2.4 Texturierung

Ein erster Schritt für eine realistische Szene ist die Auseinandersetzung mit Texturen. Diese sind Bilder, die im Speicher der Grafikkarte abgelegt werden und dann während des Rendervorgangs auf einzelne Polygone oder ganze Objekte gemapped werden. Für unsere Zwecke wurde mit 2D Texturen gearbeitet, die praktisch auf jedes Objekt „aufgeklebt“ wurden. Dabei gilt es, gerade in Hinsicht auf Beleuchtungsmodelle, verschiedene Texturen zu unterscheiden, wobei auf die Normalen- und Bump-Texturen erst im nächsten Abschnitt eingegangen wird.

2.4.1 Diffuse Texturen

Diese Textur ist die initiale Optik eines Gegenstandes ohne Einwirkung von Post-Effects oder anderen Eingriffen wie Beleuchtung. und repräsentiert lediglich die Materialfärbung.

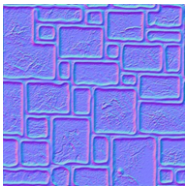
2.4.2 Spekulare Texturen



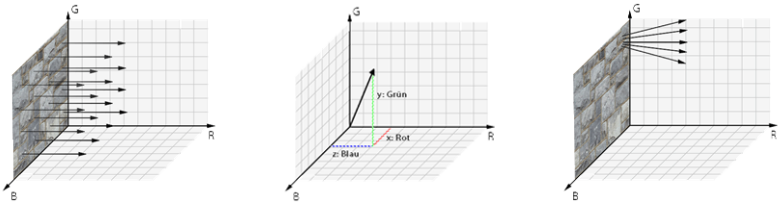
Bei der spekularen Textur handelt es sich um eine schwarz-weiß Textur, die die spekulare Lichtfarbe und Intensität beschreibt. Weiße stellen sorgen dabei für starke spekulare Reflektionen, wohingegen schwarze Stellen, die spekularen Anteile gänzlich absorbieren. So lassen sich recht einfach Oberflächen bzw. Materialien mit Glanzeffekten versehen (Feuchtigkeit, Plastik, Metall).

2.5 Mapping Methoden

2.5.1 Normal Mapping

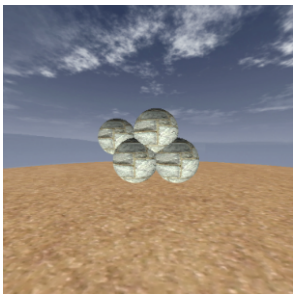


Grundsätzlich besitzt jeder Vertex einer Geometrie eine Normale, wobei diese über die Dreiecke hinweg interpoliert werden und zur Beleuchtung (durch Phong-Modell) herangezogen werden können. Beim Normal Mapping jedoch liegt eine weitere Textur vor, die nicht mehr für jeden Vertex eine Normale enthält, sondern pro Pixel eine Normale liefert. Dadurch wird bei der Beleuchtung der Eindruck einer detailreicheren Struktur suggeriert ohne, dass dabei die Anzahl der Vertices erhöht wird. Diese Mapping Methode läuft vollständig im Fragment Shader ab.

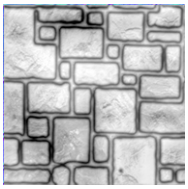


Die obigen drei Darstellung erklären den Prozess wie die Normalen der Normal-Map ihren Weg auf die diffuse Textur finden. Bekanntermaßen sind alle Normalen einer diffusen Textur in Richtung und Länge gleich. Für jedes Fragment wird nun der Farbwert der Normal-Map berrechnet und als RGB-Vektor abgespeichert.

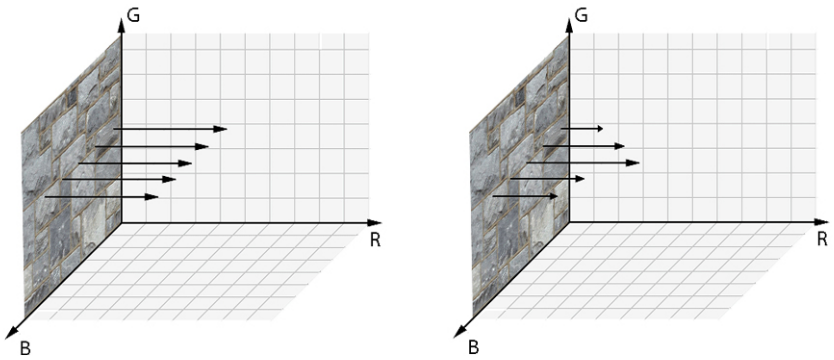
Vorführung



2.5.2 Bump Mapping



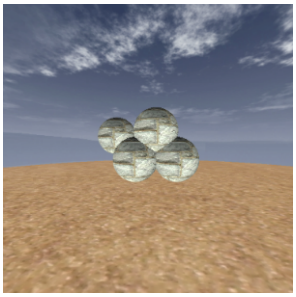
Das Bump Mapping Verfahren ist ähnlich dem Displacement Mapping. Wohingegen das Displacement Mapping direkt auf die Geometrie der Vertices eingreift und die Vertices durch eine Heightmap anhebt (diese beinhaltet als Rotwert die Höheninformation), verändert das Bump Mapping lediglich den Schatteneffekt, um einen höheren Detailgrad zu erreichen.



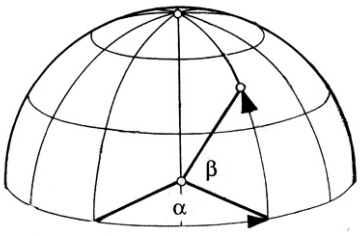
(links) Normalen vor Bump-Mapping (rechts) Normalen nach Bump-Mapping

Um beispielsweise die Beschaffenheit rauer Oberflächen zu simulieren, entnimmt man aus einer Bumpmap die Tiefenwerte und addiert sie auf die Oberflächen-Normalen auf. Da beim Phong-Beleuchtungsmodell der Lichteindruck lediglich von den Normalen der Oberflächen abhängt, wird so der gewünschte Eindruck gewährleistet. Die nebenstehenden Grafiken zeigen erst die identischen Normalen der diffusen Textur und unten die aufaddierten durch die Bumpmap. Allerdings ist der Effekt nur dann von Vorteil, wenn die Winkel der Normalen nicht allzu flach sind, da sonst der getäuschte Tiefeneindruck auffällt. Im Gegensatz zu Displacement-Mapping bietet Bump-Mapping zudem Performance-Vorteile, da keine neuen Geometrien erzeugt bzw. verändert werden, was die Vertices-Anzahl beeinflusst.

Vorführung

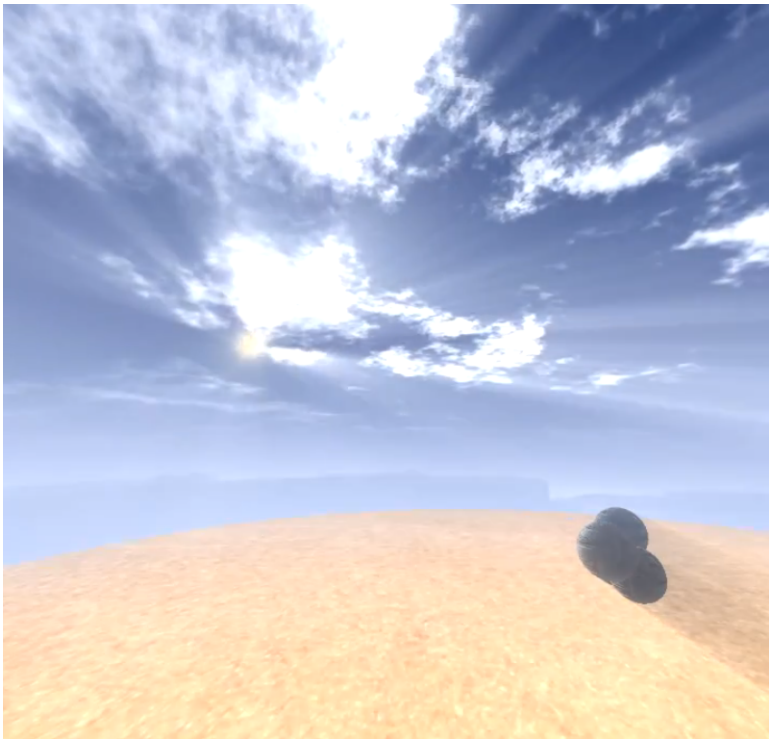


2.6 Skydome

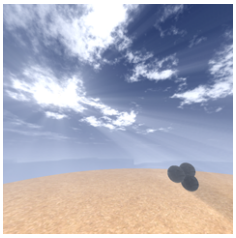


Die SkyDome, auch als sogenannte Skybox via CubeMaps implementierbar, bildet je nach Gebrauch, in unserem Fall eine Himmels-Hemisphäre, in der eine Szene eingebaut ist - später das Terrain. Die Komplexität die man möglicherweise hinter dieser Darstellung vermuten mag, ist jedoch sehr einfach implementierbar und im Folgenden werden die einzelnen Komponenten vorgestellt, die in zusammengeführter Ausführung das nebenstehende Ergebnis darstellen.

Vorführung



2.6.1 Himmel



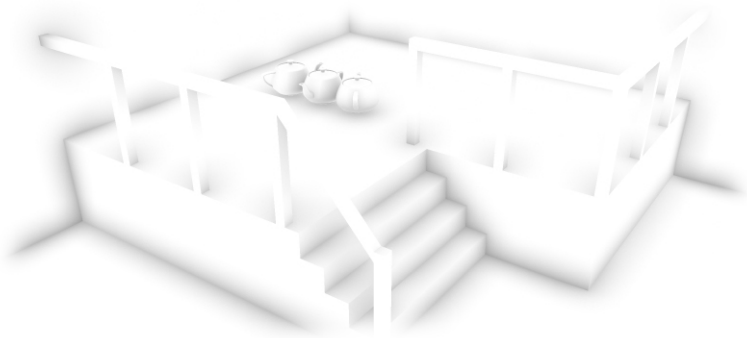
Der Himmel der SkyDome liegt im JPG-Format vor und wird über Texture-Befehle in OpenGL an eine Sphere gebunden, die durch ihren hohen Radius die gesamte Umgebung umschließt. Des Weiteren müssen die korrekten Culling-Parameter gesetzt werden. Da man sich im inneren der Kugel aufhält und Texturen sehen möchte, muss das Backface-Culling ausgeschaltet werden, sodass auch die vom Betrachter abgewandten Seiten gezeichnet werden. Im Gegenzug muss da Frontface-Culling ausgeschaltet werden, da man auch von außerhalb in die Kugel sehen soll, womit alle Polygone die im Uhrzeigesinn übergeben worden sind, nicht gerendert werden.

2.6.2 Sonne und Wolken

Die Wolken werden über eine spezielle Blend-Funktion (GL_ONE, GL_onE_MINUS_SRC_COLOR) vor den Himmel gesetzt. Dabei wird auch hier die Textur der Wolken an eine etwas kleinere Sphere gebunden, die im Rahmen des Programms transparent vor den Himmel geblendet wird. Äquivalent zu den Wolken wird auch die Sonne zwischen die Wolken und Himmel geblendet, wobei hier aber keine Sphere zu Grunde liegt, sondern die Sonnentextur auf ein „Quad“ gemapped wird. Über „Modelling Transformations“ werden Wolken und Sonne auf den richtigen Weg geschickt.

2.7 Post-Effects

2.7.1 Ambient Occlusion



2.7.1.1 Vorgehen

Unter Ambient Occlusion versteht man die Implementierung einer “Umgebungsverdeckung”, wie sie beispielsweise auf Objekten während eines bewölkten Himmels zu sehen ist. So lassen sich Gegenstände, ohne zusätzliche Beleuchtungsmodelle wie Phong, in einer Szene verbessert darstellen, da sich die Bildinhalte (Geometrien) optisch besser aufeinander abstimmen. Das Modell basiert auf realen Gegebenheiten und der Beobachtung das ein Punkt der von vielen Gegenständen umgeben ist weniger stark beleuchtet wird als solche, die direkt ohne Verdeckung von Licht beeinflusst werden. Ein zentrales Konzept der Ambient Occlusion ist der Verdeckungsfaktor, der beschreibt wie stark ein Punkt von Umgebungs-Geometrien verdeckt wird. Dies geschieht nach folgender Formel:

$$h_p = ambient \cdot (1 - v_p)$$
$$ambient = ambienterBeleuchtungsfaktor$$
$$v = Verdeckungsgrad$$
$$p = Punkt$$
$$v_p = VerdeckungsfaktorimPunktp$$

Um den Verdeckungsfaktor in einem Punkt zu berechnen, wird nun eine Hemisphäre um die Normal n der Oberfläche erzeugt. In ihr werden Zufallsvektoren gebildet, die die Umgebung nach verdeckenden Geometrien durchsuchen. Durch Integration über die Oberfläche der Halbkugel lässt sich der Verdeckungsfaktor berechnen:

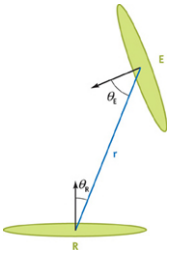
$$v_p = 1/\pi \cdot \int V(p,\omega) \cdot (m,\omega) d\omega$$

V ist hier eine Funktion, die die Verdeckung entlang eines Strahls vom fixierten Punkt in Richtung der Hemisphären Oberfläche validiert.

- Strahl wird behindert 1, sonst 0

Es wird klar, dass die wichtigen Strahlen eine annähernde Ausrichtung wie die Normale haben.

2.7.1.2 Vorgehensweise in Bezug auf die Implementation

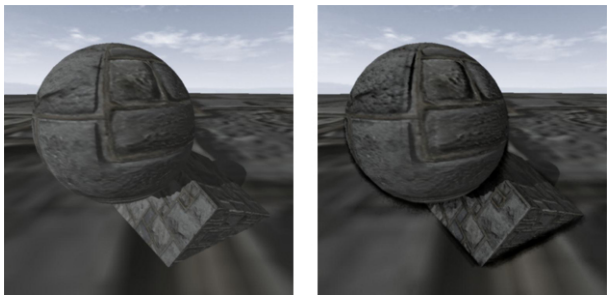


Wie die nebenstehende Abbildung zeigt, liegt dabei das Hauptaugenmerk auf den Normalen der Gegenstände (hier R, E). Man bildet nun zufällige Vektoren innerhalb einer Hemisphäre um jede Normale eines Pixels und schießt diese in den Raum. Treffen diese Zufallsvektoren auf Gegenstände so bilden die Normalen wie auch die entstehende Tiefe „r“ eine Gewichtung für die Färbung des Fragments in diesem Bereich. Es wird gezählt, wie viele von ihnen frei bleiben, also nicht mit einem anderen Objekt kollidieren, und wie vielen der Weg von einem anderen Objekt versperrt wird. Die Lichtmenge an einem Bildpunkt ist dann proportional zur Anzahl der freien Lichtstrahlen. Lichtstrahlen bestimmt, wie hell ein Bildpunkt wird. Das Ergebnis in den unteren Darstellungen, verdeutlicht diesen Prozess besonders stark an den Rändern, an denen die Wechselwirkung der Geometrien entsprechend hoch ist.

Vorteile gegenüber Ambient Occlusion

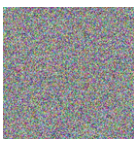
- unabhängig von der Szenen-Komplexität
- Arbeitet mit dynamischen Szenen
- Keine Arbeit auf der CPU
- Kein Daten-Preprocessing, keine Ladezeiten, keine Speicherplatz-Reservierung im Systemspeicher

Im Rahmen unserer Implementationen und Ausführungen wählten wir die Variante der „Screen Space Ambient Occlusion“, um so den Rendering-Ablauf in Echtzeit zu gewährleisten.



(links) ohne Ambient Occlusion (rechts) mit Ambient Occlusion

2.7.1.3 Algorithmus



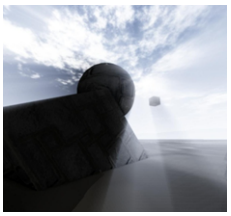
- Erzeuge Zufallsvektoren innerhalb einer Halbkugel mit Radius 1
- Auswahl einer Normalen aus der Noise-Textur (für mehr Zufall) an der die Sample-Rays später reflektiert werden
- Speicherung des aktuellen Pixel-Samples aus der Normalen-Textur
  - Auslagerung der Tiefeninformation des aktuellen Pixel-Samples
- Speicherung der aktuellen Fragment-Koordinaten im Screen-Space
- Speicherung der Normale des aktuellen Fragments

- Radius in Abhängigkeit der Tiefeninformation
- Für jedes Sample:
  - Hole einen Vektor (zufällig aus Kugel mit Radius 1.0) aus einer Textur und reflektiere ihn
  - Hole Tiefeninformation des Occluder-Fragments aus der Normalen-Textur
    - Wenn die Tiefen-Differenz negativ ist
    - Occluder ist hinter dem aktuellen Fragment
  - Berechnung der Differenz der Normalen (von aktuellen und Occluder-Fragment) als Gewichtung der Schattierung. Die „falloff“ Funktion beginnt bei einem „falloff“-Wert und fällt dann mit $1/x^2$ ab.
- Ausgabe der Farbe in Abhängigkeit der Samples

Fertiger Fragmentshader

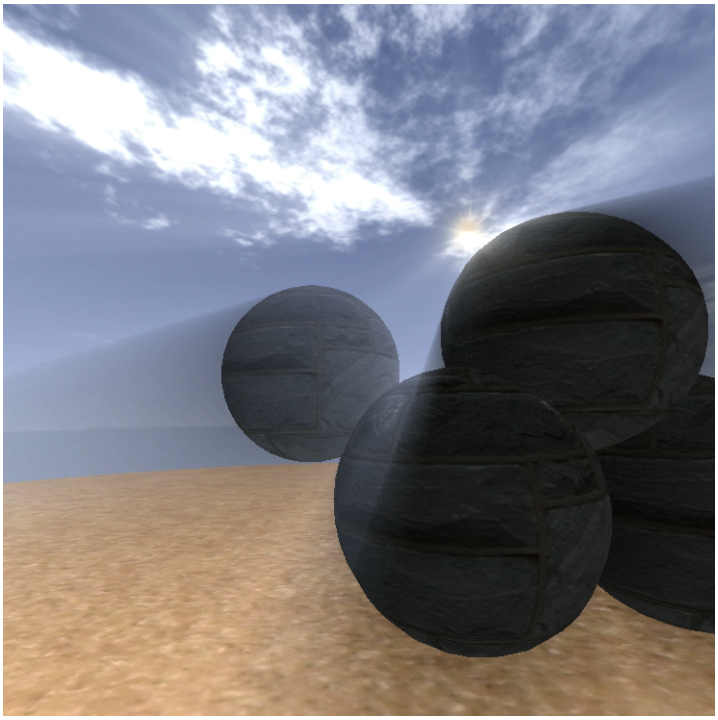
Der fertige Fragmentshader kann hier begutachtet werden:
https://github.com/cgPrak12/Praktikum/blob/darstellung_merge/shader/AmbientOcclusion_FS.glsl#L1

2.7.2 Volumetric Light Scattering



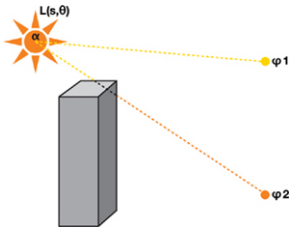
Beim Volumetric Light Scattering handelt es sich um einen Posteffekt der die Erscheinung von atmosphärischen Schatten realisiert. Jene entstehen durch kleine Teilchen, an der das Licht reflektiert, gebrochen oder abgelenkt wird. Je stärker die gegenwärtige Lichteinstrahlung ist umso stärker wird diese Interaktion und Erscheinung sichtbar. Diese Strahlen die dabei entstehen werden auch Crepuscular Rays genannt (Sun Rays, God Rays). Nun steht folgender Gedanke im Raum: Das einfallende Sonnenlicht auf der Erde ist fast parallel, weshalb eigentlich auch die Crepuscular Rays entsprechend parallel einfallen sollten. Bei der Beobachtung der Realität fällt hingegen auf, dass sie sich von der Lichtquelle in alle Richtungen ausbreiten. Dies ist ein Resultat der perspektiven Verzerrung die entsteht wenn eine 3D-Szene auf eine 2D-Fläche projiziert wird.

Vorführung



2.7.2.1 Vorgehen

- Man ermittelt für jedes Szenenpixel die Richtung zur Lichtquelle im Bildraum
- Unterteilt den Abstand des Szenenpixels zur Lichtquelle in gleichmäßige Bereiche (Samples)
- Man überprüft, ob der Lichtweg in den einzelnen Bereichen durch irgendwelche Hindernisse blockiert wird
- Sollte der Lichtweg nicht blockiert sein, dann addiert man die Helligkeit im betreffenden Bereich zur Gesamthelligkeit des Szenenpixels, andernfalls wirken die entsprechenden Faktoren für eine Abdunkelung



2.7.2.2 Algorithmus

Da es sich um einen Post-Effekt handelt, kann man von einem fertig gerenderten Bild ausgehen d.h. es existieren bereits Texturen, Beleuchtung und möglicherweise auch Schatten. Der Grundgedanke ist nun jene Pixel zu verdunkeln die nicht von Strahlen getroffen werden und umgekehrt jene die im Bereich der Strahlen liegen zu erhellen.

- Für jeden Pixel berechne einen Vektor von ihm zur Lichtquelle (in Bildkoordinaten mit z-Werten)
- Berechnung des Vektors von Lichtquelle im Screen-Space zum Pixel
- Teilen durch die Anzahl der Samples und Skalierung durch den Kontrollfaktor density (Dichte)
- Speichern des initialen Sample-Werts
- Setzen des Faktors illumination decay (Beleuchtungszerfall)
- Auswertung der Aufsummierung der folgenden Gleichung in Bezug auf die Sample-Anzahl

$$L(s, \theta, \phi) = \sum_{i=0}^{n} decay^i \cdot weight \cdot \frac{L(s_i, \theta_i)}{n}$$

$$s = Distanz\ des\ Lichtes\ durch\ das\ Medium$$

$$\theta = Winkel\ von\ Strahl\ zu\ Sonne$$

$$\phi = Position\ des\ Betrachters$$

$$n = Sample - Anzahl$$

$$exposure = Intensity\ des\ gesamten\ Effekts$$

$$weight = Intensity\ der\ einzelnen\ Samples$$

$$decay^i = Zerstreuungsfaktor\ der\ Intensity\ vom\ Licht\ weg$$

$$L(s_i, \theta_i) = Sample$$

- Für jedes Sample
  - Wander den Strahl weiter ab
  - Aufruf des Samples an der neuen Stelle
  - Wende illumination decay und weight auf Sample-Wert an
  - Aufaddierung der Farbe von sample und ursprünglicher Sample color
- Aktualisierung des illumination decay Faktors für den exponentiellen Abfall

- Ausgabe der Farbe
  - Skalierung über den Faktor exposure (Belichtung)

Fertiger Fragmentshader

Der fertige Fragmentshader kann hier begutachtet werden:
https://github.com/cgPrak12/Praktikum/blob/darstellung_merge/shader/GodRayFS.glsl

3. Darstellung: Beleuchtung,Tone Mapping, Bloom-Effekt und Schatten

3.1 Einleitung

Während des Computergrafikpraktikums m August 2012 sollten die Teilnehmenden Studenten gemeinsam mit Hilfe von Java, OpenGL mit Hilfe der LWJGL-Bibliothek und Shaderprogrammen ein Programm zur Erstellung und Echtzeitdarsellung eines dreidimensionalen Terrains entwickeln. Die Darstellung sollte dabei mit Techniken der Rastergrafik umgesetzt werden. Wir waren dabei mit anderen für die Darstellung dieses Terrains zuständig. Insbesondere sollten wir uns mit Post-Processing-Effekten beschäftigen und einige dieser, von modernen Echtzeit-Computergrafikanwendungen bekannten, Effekte umsetzen. Die Terraindarstellung sollte mit Hilfe dieser Techniken für den Betrachter realistischer wirken.

Wir entschlossen uns dazu, Shaderprogramme für Tone Mapping, einen Bloomeffekt und Schattendarstellung mit Shadow Maps zu programmieren und diese in das Programm zur Terraindarstellung sinnvoll zu integrieren. Natürlich durfte auch die Beleuchtung nicht fehlen. Hierbei entscheiden wir uns für die Umsetzung der Beleuchtung mit Blinn-Phong. Im Folgenden werden wir näher auf die genannten Techniken eingehen, sowie unsere Vorgehensweise, die Techniken und den Programmcode näher betrachten. Auch Die Anbindung im Programm werden wir kurz erläutern.

Zurück

3.2 Die Anbindung im Programm

Um die Szene sinnvoll und effizient mit unseren Posteffekten bearbeiten zu können, verwendeten wir das Deferred Shading:

Deferred Shading mit Frame Buffern

Zum einfachen Anpassen und Debuggen haben wir uns ein Grafik- und Splitscreen-Menü programmiert:

Grafikmenü und Splitscreen

Zurück

3.2.1 Deferred Shading mit Frame Buffern

Um die Lichtberechnung und Posteffekte effizient und einfach nutzen zu können, verwendeten wir das Konzept des Deferred Shadings. Bei dieser Technik werden Normalen, Farbe und Tiefenwert der Vertices in bildschirmgroßen Texturen gespeichert. Bei klassischen Rendermethoden hingegen werden diese Eigenschaften meist direkt für die Beleuchtung verwendet. Technisch verwendeten wir hierzu Framebufferobjekte, in welche durch Multiple Render Targets gleichzeitig geschrieben werden kann.

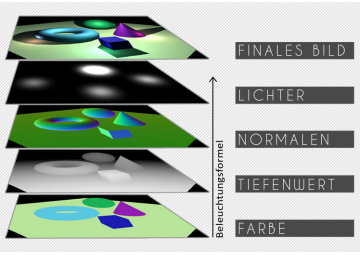


Abbildung 1: Framebufferobjekte eines Deferred Shaders

Deferred Shading hat somit den Vorteil, dass nur ein geometrieverarbeitender Schritt gemacht werden muss und dann die Beleuchtung und weitere Posteffekte nur noch auf die

einzelnen Pixel angewendet werden müssen. Also unabhängig von der Geometrie berechnet werden. Dies ist insbesondere bei mehreren Lichtquellen deutlich günstiger. Insbesondere war es uns durch diese Rendermethode möglich, für unsere Posteffekte zwischen Farben verschiedener Texturen die durch Beleuchtung oder einen Unschärfefilter beispielsweise entstehen, zu interpolieren. In Abbildung 1 sind die Inhalte der Framebuferobjekte zu sehen: Farbinformation (diffus), Tiefenwert, Normalen, Lichtinformtionen und das finale Bild.

Zurück

3.2.2 Grafikmenü und Splitscreen

Um uns das Debugging zu vereinfachen und die Postprocessing-Werte komfortabel anpassen zu können und damit die Szene nach unseren Ansprüchen zu gestalten, entschlossen wir uns dazu ein kleines Grafikmenü mit grafischer Benutzeroberfläche zu programmieren. In diesem ist es möglich die Faktoren für die Posteffekte Tone Mapping und Bloom direkt zu verändern und die Effekte somit optimal auf die Szene Anzupassen. Ausserdme lassen sich die verschiedenen Posteffekte einfach zu beziehungsweise abschalten, was die Fehlersuche komfortabler gestaltet und ebenfalls die Anpassung der Szene vereinfacht.

Abbildung 2: GUI des Grafikmenüs

In einem zweiten Reiter ist es außerdem möglich, Inhalte der verschiedenen Framebufferobjeke der selben Szene in einer Splitscreendarstellung anzeigen zu lassen. In Abbildung 3 sieht man in dieser Darstellung die beleuchtete Szene mit und ohne Tone Mapping, die Brightmap sowie die Shadowmap der Szene. Wertvoll sind hier ebenfalls die Debuggingmöglichkeiten.

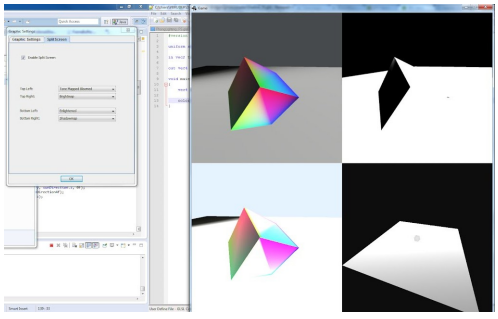


Abbildung 3: Split Screen

Zurück

3.3 Posteffekte

In diesen Unterkapiteln stehen die verschiedenen, von uns umgesetzten Posteffekte im Vordergrund. Hierbei nehmen insbesondere die Fragment-Shader eine wichtige Rolle ein.

- Beleuchtung mit Blinn-Phong
- Tone Mapping

- Bloomeffekt
- Schatten mit Shadow Maps

Zurück

3.3.1 Beleuchtung mit Blinn-Phong

Vorteile gegenüber Phong

Abbildung 4: Vektoren in Blinn und Phong

Das Beleuchtungsmodell nach Blinn und Phong orientiert sich stark an dem bereits aus der Computergrafikvorlesung bekannten Phong-Modell. Jedoch soll es durch Modifikationen die auf Jim Blinn zurückzuführen sind, schneller arbeiten. Dies wird insbesondere durch die Verwendung einer Winkelhalbierenden erreicht. In Abbildung 4 sieht man die zur Beleuchtung benötigten Vektoren sowohl für das klassische Phong-Modell als auch die von uns verwendete Variante. Dabei steht N für die Normale des betrachteten Fragments, L für den Vektor vom Fragment zur Lichtquelle, V für den vom Fragment zur Kameraposition und R für den Reflexionsvektor. Im Phong-Modell muss nun kontinuierlich das Skalarprodukt zwischen den Vektoren R und V berechnet werden. Nun kann man aber auch die Wikelhalbierende H zwischen L und V berechnen und damit R*V duch N*H ersetzen. Der somit berechnete Kosinus, welcher für die Berechnung des spekularen Anteils benötigt wird, ist ungefähr halb so groß wie der nach dem klassischen Phong-Modells. Den Faktor 2 kann man durch geeignete Wahl eines höheren Exponenten leicht wieder ausgleichten. Somit kann ohne großen Qualitätsverlust die Beleuchtung günstiger berechnet werden.

Sonnenauf- und Untergang

Zusammen mit Nils Bussman und Timo Bourdon versuchten wir in unserem Beleuchtungsshader eine Sonnenauf- beziehungsweise Untergangssimulaion zu integrieren. Dies gelang uns, indem wir einen leichten Rotton abhängig vom Stand der Sonne, also ihrem y-Wert, in die Beleuchtungsfarbe mischten. Schließlich überprüften wir noch anhand des y-Wertes der Sonnenposition, ob gerade Tag oder Nacht ist und mischten dann bei Nacht einen hohen ambienten Farbanteil bei, abhängig wiederrum von der Sonnenposition. Um eine hohe Effizienz zu gewährleisten, verwendeten wir die step-Funktion statt if-Konstrukten im Shader.

Somit entstand folgender Code, mit dem sich in Sonnen auf bzw. Untergang erstaunlich gut simulieren lässt:

```
 enlightenedColor += sunBurn*(1-cosa)*sunSetColor;
 enlightenedColor  =
            enlightenedColor*step(0, sunDir.y) + (1-step(0, sunDir.y))*mix(vec4(ambi*k_a, 1),
enlightenedColor, (1-cosa));
```

sunSetColor und sunBurn sind Konstanten, ambi der ambiene Farbanteil und cosa die normalisierte Länge des y-Wertes der Sonnenposition.

Zurück

3.3.2 Tone Mapping

High Dynamic Range Rendering

Abbildung 6: ohne Tone Mapping

High Dynamic Range (HDR) Bilder (zu deutsch Hochkontrastbild) haben im Gegensatz zu Low Dynamic Range (LDR) Bildern einen Dynamikumfang, welcher eher dem der wirklichen Welt entspricht.

Ein normales, vom Monitor dargestelltes Bild, ist ein LDR-Bild, da Bildschirme nur einen eingeschränkten statischen Kontrast haben: etwa 500 - 1000 : 1. In einer reellen Szene mit direktem Sonnenlicht kann in etwa ein Kontrastverhältnis von 50000 : 1 existiert. In älteren OpenGL Versionen waren nur bis zu 8 Bit pro Farbkanal möglich, was einen Kontrast von gerade einmal 256:1 ermöglichte. Mit einer aktuellen Version ist man heute glücklicherweise nicht mehr so eingeschränkt. 32 Bit pro Farbe sind durchaus möglich, was theoretisch Farbwerte von 1.18*10^-38 bis 1.18*10^38 ermöglicht. Jedoch kann ein durchschnittlicher Monitor ein solch großes Spektrum nicht abdecken. Um nun den erweiterten Farbraum und damit höheren Kontrast auf dem Montor anzeigen zu können, muss das Bild mittels Tone Mapping auf den anzeigbaren Farbraum abgebildet werden. Dabei sollte aber beachtet werden, dass der hohe Kontrast für den Anwender nicht verloren geht, es muss also der Eindruck eines höheren Konrastes geschaffen werden.

Tone Mapping

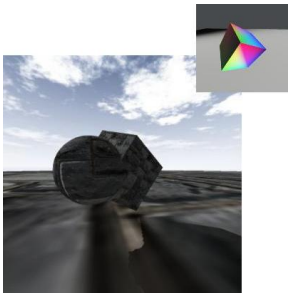


Abbildung 7: mit Tone Mapping

Tone Mapping (zu Deutsch auch Dynamikkompression) wird also dazu benutzt, High-Dynamic-Range-Bilder auf den Farbraum des Monitors abzubilden. Dies geschieht bei uns als letzter Posteffekt, damit alle anderen Posteffekte von dem erweiterten Farbraum profitieren können.

Um nun dieses Bild für die Ausgabe an einem Monitor vorzubereiten, müssen die Farben und Helligkeiten des Bilds angepasst werden. Hierfür gibt es verschiedene Ansätze. Der einfachste ist sicherlich, die Farben einfach auf den entsprechenden Bereich zu "stauchen". Ein erweiterter Ansatz ermöglicht durch einen variablen Blendfaktor (Exposure) zusätzlichen Spielraum für den Gestalter oder Programmierer. Wir entschieden uns für eine Variante mit variablem Blendfaktor, die aber diesen zusätzlich dynamisch mit lokaler Abhängigkeit anpasst. Dieses Verfahren hat den großen Vorteil, dass Details auch bei sehr großem Kontrastunteschied im Bild nicht so schnell verloren gehen, da der Blendfaktor für jeden Pixel in Abhängigkeit der 24 sie umgebenden Pixel angepasst wird.

3.3.3 Bloomeffekt

Was ist Bloom?

Abbildung 8: Bloom-Effekt

Bloom bezeichnet einen Effekt, dass wenn zwei benachbarte Punkte einen sehr großen Helligkeitunterschied haben diese leicht verschwimmen, das helle Licht also über die Kante strahlt. Dieser Effekt tritt zum Beispiel bei dunklen Räumen mit hellen Fenstern auf, durch die helles Licht, wie etwa Sonnenschein, fällt. Die Fensterkanten verschwimmen dann scheinbar, da das Licht im Gegensatz zur Helligkeit im Raum sehr hell ist und das Auge diesen Helligkeitsunterschied schwer verarbeiten kann. Noch deutlicher tritt dieser Effekt in Filmen oder Fotos auf, da er durch die Kameralinse verstärkt wird (diese können nicht perfekt fokussieren). In der Rastergrafik wird der Bloom als Posteffekt eingesetzt, um dieses Verhalten zu simulieren und außerdem harten Helligkeitsübergängen entgegenzuwirken.

Bestandteile des Bloomeffektes

Der Bloom wird erzeugt indem man mehrer Bilder mixt. Zuerst muss vom vorhanden Bild eine Lightmap erstellt werden, in welche die Helligkeitswerte des Bildes gespeichert werden. Insbesondere enthält diese Brihtmap Informationen darüber, ob ein Pixel über einem bestimmten Helligkeitswert liegt. Dieser Grenzwert kann auch angepasst werden.

Blureffekt

Um ein gut aussehendes “Ausbluten” des Blooms zu erreichen wird auch ein unscharfes Bild ("Blur") der Szene benötigt. In unserem Fall benutzen wir vier verschiedene Unschärfestufen desselben Bildes. Dabei ist das erste Bild das geblurte Orignalbild, das zweite Bild das nochmals geblurte erste Bild und so weiter. Im Folgenden sieht man unseren Code des Blur-Shaders. Dabei wird pro Texel die Farben der 25 umliegenden nach der "Gaußschen" Normalverteilung gewichtet, gemischt. Es entsteht ein weichgezeichetes Bild.

Blur:

```
vec4 sample[25];
for(int i = 0; i < 25; ++i) {
sample[i] = textureOffset(colorTex, texCoord.st, ivec2(tc_offset[i]));
}
bluredColor = (
( 1.0 * ( sample[0] +  sample[4] + sample[20] + sample[24])) +
( 4.0 * ( sample[1] +  sample[3] +  sample[5] +  sample[9] + sample[15] + sample[19] + sample[21]
+ sample[23])) +
( 7.0 * ( sample[2] + sample[10] + sample[14] + sample[22])) +
(16.0 * ( sample[6] +  sample[8] + sample[16] + sample[18])) +
(26.0 * ( sample[7] + sample[11] + sample[13] + sample[17])) +
(41.0 *  sample[12])
) / 273.0;
```

Finales Bloombild

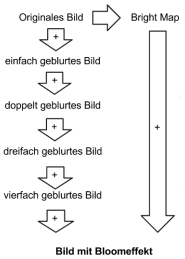


Abbildung 9: Zusammensetzung Bloom

Diese Bilder vier Unschärfebilder werden nun im Fragment-Shader addiert und dann zu bestimmten Anteilen mit dem originalen Bild und der Brightmap gemischt. Die Intensität des Blooms hängt hier bei besonders von der Lightmap und der “Exposure”-Variable (Stärke des Blooms) ab. Die Berechnung im Fragment-Shader ist unten dargestellt:

Bloom:

```
 vec4 baseImage  = texture(origImage, texCoord);
 vec4 brightPass = texture(brightImage, texCoord);
 vec4 blurColor1 = texture(blur1, texCoord);
 vec4 blurColor2 = texture(blur2, texCoord);
 vec4 blurColor3 = texture(blur3, texCoord);
 vec4 blurColor4 = texture(blur4, texCoord);

 vec4 bloom = brightPass + blurColor1 + blurColor2 +
blurColor3 + blurColor4;
 bloomColor = baseImage + (bloomLevel * bloom);

 bloomColor.a = 1.0;
```

Zurück

3.3.4 Schatten mit Shadow Maps

Schatten sind in der realen Welt überall zu beobachten wo es Licht gibt: Befindet sich ein Gegenstand zwischen dem zu betrachtenden Punkt und der Lichtquelle, so ist der Punkt dunkler als würde er direkt vom Licht angestrahlt. Will man eine möglichst authentische Szene virtuell erstellen sollten Schatten also wenn möglich nicht fehlen.

In der Rastergrafik ist dies leider nicht ganz so einfach umzusetzen, da alle Objekte einer Szene nur in Abhängigkeit ihrer Oberflächenbeschaffenheit und ihrer Position zur Sonne beleuchtet werden. Schatten werden hier nicht, wie in der Realität oder beim Raytracing "automatisch generiert", sondern müssen explizit berechnet werden.

Schattenberechnung mit Shadow Maps

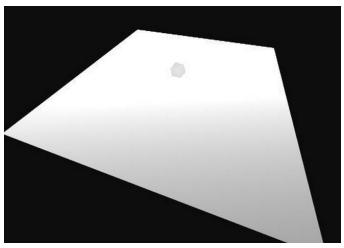


Abbildung 10: Shadow Map

Wir entschieden uns dazu, die Schatten mit Hilfe einer Shadow Map zu berechnen. In dieser, welche eigentlich nichts anderes als eine Textur ist, stehen die Tiefeninformationen der zu schattierenden Szene aus Sicht der Lichtquelle. Es wird also eine zweite Kamera an den Koordinaten der Lichtquelle erzeugt, welche die Szene mit Hilfe dieser neuen Kamerakoordinaten rendert und die Tiefeninformationen in einer Textur speichert. Dies ergibt die Shadow Map. In unserer aktuellen Szene können wir nun die Entfernung vom aktuell betrachteten Punkt bis zu Sonne betrachten. Ist dieser Wert größer als die Entfernung zu diesem Punkt in der Shadow Map, muss der Punkt aus Sicht der Lichtquelle von einem Objekt verdeckt sein und liegt somit im Schatten. Ergo muss dieser

dunkler, bei uns nur vom ambienten Farbteil des verwendeten Beleuchtungsmodells gefärbt, dargestellt werden.

Projektion der Schatten

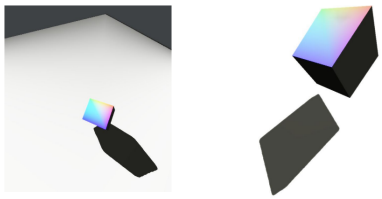


Abbildung 11: Perspektivische- (links) und parallele Projektion (rechts)

Wir entschlossen uns dafür, die Schatten parallel zu projizieren und nicht wie unsere Szene in perspektivischer Form. Hierfür verwendeten wir die orthogonale Projektion der Kamera die ja bereits in der Kameraklasse integriert war. wir mussten lediglich noch einen variablen Entfernungfaktor einbauen. Die parallele Projektion ergibt deshalb Sinn, weil wir in unserem Terrain nur eine Lichtquelle, nämlich die Sonne, haben und diese als beinahe unendlich weit entfernt angenommen werden kann. Bei perspektivischer Projektion würde nämlich ein extrem langer Schatten bei entsprechendem Lichteinfallswinkel geworfen werden, was nicht der in der Realität zu beobachtenden Schattenbildung entspricht. Der Unterschied ist auch in nebenstehender Abbildung 11 zu sehen.

Probleme bei der Darstellung

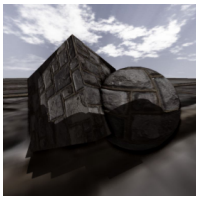


Abbildung 11: Self-Shadowing-Artefakte

Die Schattendarstellung mittels Shadow Maps ist zwar leichter umzusetzen als andere Techniken, bieten aber leider auch einige Nachteile. Zum Beispiel treten Self-Shadowing-Artefakte auf, das heißt die Geometrien werfen Schatten auf sich selbst. Ungenaue Tiefenberechnung bzw. Rundungsfehler können hierbei eine Rolle Spielen. Wir haben deshalb für die Schadow Map Berechnung Front-Face-Culling aktiviert, also nur die Rückseiten der Geometrien gerendert. Mit diesem "Trick" konnten wir den Artefakten in gewissem Rahmen entgegenwirken.

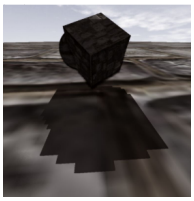


Abbildung 12: Aliasing bei Schattenkanten

Als zweites Problem bei der Darstellung hatten wir zu Beginn die harte Ecken an den Kanten der Schatten. Diese waren nicht mehr ganz so einfach zu eliminieren. Zu erst erhöhten wir die Texturgröße der Schadow Map auf 32 Bit pro Kanal, was schon eine leichte Verbesserung mit sich brachte. Nachdem wir die Kamera für die Schattenberechnung näher an der Szene Platzierten als unsere Lichtquelle, war das Ergebnis schon fast zufriedenstellend. Da wir aber noch nicht ganz zufrieden waren, änderten wir unseren Beleuchtungsshader dahingehend, dass wir einen einfachen Anti-Aliasing-Algorithmus ("Kantenglättung") für die Schattendarstellung integrierten. Bei diesem werden die Farben der umliegenden Texel gemittelt. Danach konnten wir recht ansehnliche Schatten generieren.

Schattenberechnung mit Anti Aliasing:

```
  vec4 shadowCoord = lProj * lView * vec4(positionWC.xyz,
1);
  shadowCoord /= shadowCoord.w;

  float sum = 0;
  int count = 0;

  for (int i = -2; i < 3; i++)
  {
    for (int j = -2; j < 3; j++)
    {
      sum += textureOffset(shadowTex, 0.5 +
0.5*shadowCoord.xy, ivec2(i,j)).w;
      count++;
    }
  }
  float shadow = sum / count;
```

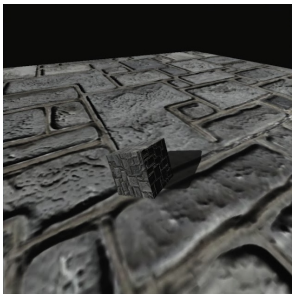


Abbildung 13: fertige Schatten

Zurück

3.4 Probleme und Fazit

Probleme

Texture-Units

Eines der Probleme zu Beginn waren die Texturen in OpenGL. Da wir für jeden Post-Effekt eigene FrameBuffer mit Texture erstellten, welche die gleiche Texture-Unit verwendeten kam es zu Problemen. Diese Texturen wurden zum Teil gleichzeitig benutzt, wodurch es zu unvorhersehbaren Ausgaben kam. Um das Problem zu beheben mussten wir jeder Texture eine eigene Texture-Unit zuweisen. Dadurch benötigten wir jedoch insgesamt über 60 Texture-Units, auch weil der Deferred Shader zusätzlich mehrere Texturen und damit auch Texture-Units brauchte. Dies funktionierte auf den Nvidia Grafikkarten (GTS 450) im CIP-Pool ohne Probleme, jedoch auf älteren AMD Karten sowie Notebookgrafiklösungen nicht immer fehlerfrei.

Merge

Das größte Problem war der abschließende Merge. Dadurch, dass es unter den einzelnen Gruppen kaum Absprachen gab, gestaltete es sich nicht einfach die Programme zusammenzuführen. Es hätte am Anfang des Projektes ein fester Plan für die Interfaces und Schnittstellen erstellt werden müssen und auch die Größe der einzelnen Bestandteile hätte im Vorfeld festgelegt werden müssen. Hilfreich wäre auch eine eindeutig Deadline ein paar Tage vor Ende des Projekts gewesen, dann hätten alle beteiligten noch ausreichend Zeit für den Merge und die nötigen Anpassungen gehabt. So funktionierte der Merge mit der Gruppe Normal Mapping, Ambient Occlusion und Volumetric Light Scattering noch ohne größer Schwierigkeiten innerhalb eines Tages. Mit den anderen Gruppen gestaltete es sich jedoch nicht so einfach. Die Wasseranimation war zum Beispiel im Vergleich zu unserer Testszene mit Skydome extrem klein. Das komplette Wasserbecken hatte in etwa ein Viertel der Größe unseres Testwürfels. Im Gegensatz dazu war das Terrain sehr groß. So ragten die Berge weit aus dem Skydome hinaus und man sah kaum etwas vom Terrain, da innerhalb des Skydomes nur ein Bruchteil des Terrains zu sehen war. Ein einfaches Vergrößern des Skydomes war leider nicht möglich, denn dies brachte das Problem von auftretenden Float-Ungenauigkeiten mit sich.

Ein weiteres Problem beim Merge waren die Shader. Dadurch das jede Gruppe eigene Fragement- und Vertex-Shader hatte mussten diese bei jedem Merge neu angepasst werden, was abhängig von der Gruppe mal relativ einfach ging, bei anderen jedoch in endlicher Zeit kaum möglich war.

Das Hauptproblem beim Merge war jedoch die Zeit. Den ersten Merge mit der Normal Mapping, Ambient Occlusion und Volumetric Light Scattering Gruppe machten wir zu Ende der zweiten Woche und brauchten dafür in etwa einen Tag. Der zweite Merge mit der Wasserdarstellung Gruppe starten wir zu Beginn der dritten Woche. Er gestaltete sich etwas schwieriger, brauchte aber auch nur etwa eineinhalb Tage. Nun war das Problem, dass viele der anderen Gruppen noch an ihren Umsetzungen arbeiteten und die Versionen noch nicht bereit waren für einen Merge. Dadurch mussten wir bis einen Tag vor der Präsentation warten bis wir eine mergebereite Version des Terrains bekamen. Ein Tag war jedoch, selbst mit der Unterstützung der Tutoren, leider nicht genug, um das Programm mit all unseren Effekten zum Laufen zu bringen.

Fazit

Da unsere Aufgabe die Umsetzung von Posteffekten waren, welche hauptsächlich in Fragement- und Vertex-Shadern erzeugt werden, konnten wir im Computergrafik-Praktikum viele praktische Erfahrungen im Umgang mit den Shadern und der Shading-Sprache GLSL sammeln. Wir sind im Verlauf des Praktikums auf viele Probleme mit OpenGL und den Shadern gestoßen, welche wir aber fast alle lösen konnten. Einige alleine, andere mit Hilfe des Internets oder unserer hilfsbereiten Tutoren.

Durch diese Schwierigkeiten und unsere Ergebnisse bekamen wir einen ganz anderen Blick auf die Arbeit der Entwickler aktueller Rastergrafik-Engines, welche beeindruckende Effekte erzeugen und trotzdem performant laufen. Im Gegensatz dazu lief unsere Beispielszene nach dem zweiten Merge mit durchschnittlich nur 30fps (frames per second).

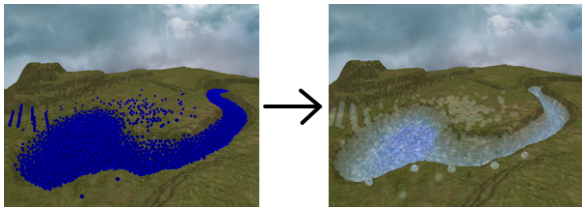
Des Weiteren konnte man einen guten Einblick in den Verlauf und die Schwierigkeiten von größeren Projekten bekommen. Am Ende ist uns klar geworden, dass die Absprache unter den einzelnen Gruppen sehr wichtig ist, damit sich die einzelnen Teile am Ende zu einem großen und funktionierenden Programm zusammensetzen lassen. Diese kam in diesem Praktikum leider etwas zu kurz (daran sind wir auch teilweise selbt schuld), sodass dies am Ende mit ohne Weiteres möglich war.

Letztendlich konnten wir viel aus dem Praktikum mitnehmen. Sowohl die Dinge die wir erfolgreich gemeistert haben, als auch unsere Fehler die wir im Verlauf des Praktikums gemacht haben bleiben uns für die Zukunft in Erinnerung. So werden wir unsere Fehler hoffentlich nicht wiederholen.

Nach diesem Praktikum steht unserer Zukunft als angehende Spieleprogrammierer nicht mehr im Wege!

4. Darstellung: Wasser

4.1 Einleitung



Die Simulationsgruppe befasste sich mit der realistischen Simulation von Wasser. Hierbei wurden einzelne Vertices miteinander verrechnet, sie kollidierten und flossen so entlang eines generierten Flussbetts.

Die Aufgabe der Wasserdarstellungsgruppe war es nun, die Partikel wie Wasser aussehen zu lassen. Mit Hilfe eines eigenen Deferred Shaders wurde dies durch das so genannte Surface Rendering realisiert. Die Methode des Surface Renderings wird im Folgenden mit Zielen und Problemen erläutert.

Deferred Shading

Für die Realisierung mit einem Deferred Shader werden mehrere Framebuffer Objects mit verschiedenen Textureattachments erzeugt. Der Vorteil ist, dass die Szene so nur sehr selten in ihrer Gesamtheit gerendert werden muss. Sie wird in einem ersten Schritt in eine Textur geschrieben, die dann weiter verarbeitet wird. Die Textur hat so nur noch die letztendlich sichtbaren Elemente, was in späteren Schritten Rechenaufwand spart. Ein weiterer Vorteil ist, dass viele Texturen eines Framebuffer Objects gleichzeitig geschrieben werden können. Durch dieses Vorgehen können Texturen von einem Path zum nächsten weitergereicht und effizient weiterverarbeitet werden.

Wichtig für die Wasserdarstellung ist, dass zuerst die Szene ohne Wasser gerendert wird. Erst wenn diese fertig beleuchtet und nachbearbeitet ist, wird das Wasser simuliert und gerendert. Der letzte Schritt im Gesamtprojekt ist ebenfalls ein Teil der Wasserdarstellung: Das Zusammensetzen des finalen Bildes, das auf den Bildschirm gebracht wird.

Dabei entstehen einige Probleme, unter anderem muss z.B. dafür gesorgt werden, dass Wasser, das vom Gelände verdeckt wird, nicht angezeigt wird. Ein anderes Problem ist, dass einige Daten nicht unbedingt sinnvoll weitergereicht werden können. So werden die Normalen mit Hilfe der Tiefentextur und Texturkoordinaten berechnet, statt in den ursprünglichen Weltkoordinaten.

Die einzelnen Abschnitte des Deferred Shadings nennt man Paths. Jeder Path enthält im Grunde genommen das Anbinden eines Framebuffer Objects und eines Shaderprograms, das Anbinden benötigter Uniform-Variablen an die benötigen Shader und das Zeichnen einer Geometrie. Für das Zeichnen von Texturen wird ein sog. ScreenQuad verwendet, das nur die Eckpunkte des Bildschirms als Vertices nutzt.

Arbeitsschritte

Die einzelnen Arbeitsschritte für das Surface Rendering bestanden darin, den Deferred Shader zu entwickeln, die Beleuchtung vorzubereiten, sie auszuführen und letztendlich aus der fertigen Szene und dem Wasser ein finales Bild zu erzeugen.

Das Ergebnis kann in einem kleinen Video begutachtet werden.

4.2 Surface Rendering

Die Aufgabenstellung der Wasserdarstellungsgruppe war es aus den von der Simulationsgruppe gelieferten Partikelinformationen eine Wasseroberfläche in OpenGL mithilfe verschiedener Prozesse zu generieren. Man könnte den Vorgang folgendermaßen grob zusammenfassen.

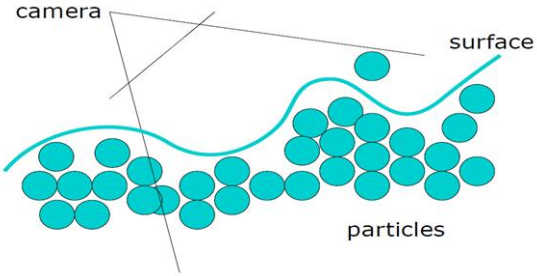
Die Simulation liefert die Partikelpositionen in Form von Vektoren

$$\vec{v_i} = (x_i, y_i, z_i)^T$$

mit drei Komponenten

$$x_i, y_i, z_i$$

Ausgehend von diesen Partikelpositionen werden eine „Depthtexture“ und eine „Thicknesstexture“ erstellt. Danach werden auf die Texturen verschiedene Glättungsmethoden angewandt und anhand der zuvor berechneten Normalen die Beleuchtungseffekte berechnet. In einem finalen Schritt werden alle Texturen und der Untergrund zusammengefügt.



4.3 Rendering Particle Spheres

Bevor man rendert muss die vorderste Wasseroberfläche, ausgehend von der Kameraposition, bestimmt werden. Dazu werden die Partikel als Kugeln gerendert und mithilfe des in OpenGL vorhandenen Tiefentest nur der Wert angezeigt der der Kamera am nächsten liegt.

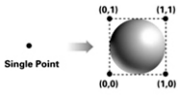
Um die Partikel als Kugeln zu rendern muss man diese zunächst als point sprites(ab OpenGL 1.5) rendern. Gleichzeitig wird im Vertexshader eine w-Komponente der Partikelposition hinzugefügt, die später die Tiefe repräsentieren soll. Mit einer if-Abfrage im Fragmentshader und dem discard Befehl kann man die Pixel außerhalb eines bestimmten Radius um die Partikelposition nun entfernen, z.B.:

```
if(length(gl_PointCoord - vec2(0.5,0.5)) > 0.5) discard;
```

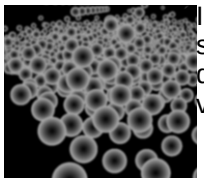
Um es performanter zu machen bietet sich an die „aufwändige“ Methode length zu umgehen:

```
if(dot(gl_PointCoord - vec2(0.5,0.5), gl_PointCoord - vec2(0.5,0.5)) > 0.25)
discard;
```

In unserer Implementation wurde dieser Befehl zudem in mehrere Befehle aufgeteilt, damit z.B. gl_PointCoord - vec2(0.5, 0.5) nur einmal berechnet werden muss, um so die Performance noch weiter zu steigern. Nach diesem Befehl erhält man letztlich Kreise, die die Partikel repräsentieren.



Mithilfe der Tiefe soll nun die Kugel simuliert werden. Dafür wird in dem Fertexshader in Abhängigkeit des Abstands der Partikelposition zu einem Punkt auf dem Kreis, der diesen Partikel repräsentiert, und in Abhängigkeit von frei wählbaren Skalierungsfaktoren ein Tiefenwert berechnet. Die x_i-,y_i- und z_i-Komponenten des Vektors v_i beinhalten weiterhin nur die Position des Partikels.



In diesem Beispiel wurde die Tiefe als Farbe ausgegeben und angezeigt. Die Skalierung wurde so gewählt, dass man die Simulation der Kugel besser erkennt. Schwarz bedeutet, dass an dieser Stelle die „Kugel“ am nächsten zur Kamera zeigt und die weißen Stellen immer weiter von der Kamera entfernt. Diese Textur wurde z.B. für die Berechnung der Normalen verwendet.

Letztlich mussten mehrere Versionen der Depthtexture erstellt werden, da die weiteren Schritte auf verschiedene Skalierungen, Auflösungen u.Ä. angewiesen sind. Diese Textur ist für den Tiefentext nötig.

4.4 Smoothing

Es ist nicht wünschenswert, dass man die Kugeln, aus denen das Wasser erstellt wird, auch als solche erkennt. Das Ziel ist eine feine und glatte Wasseroberfläche. Dies kann man erreichen, in dem man statt den ursprünglichen Kugeln nur Annäherungen an die Partikelpositionen nimmt. Dabei verfälscht man zwar die eigentlichen, exakten Werte aber das Ergebnis ist eine deutlich wasserähnlichere Oberfläche. Diese Annäherungen werden in unserer Implementation von den „Glättungsmethoden“ ermittelt. Die Glättungsmethoden werden also dazu benutzt die Übergänge zwischen den einzelnen Kugeln weicher zu machen um somit letztlich aus einer „Kugelmasse“ eine möglichst realistisch aussehende Wasseroberfläche zu generieren. Die von uns angewandten Methoden waren der im Folgenden erklärte Gaussian blur filter und die Interpolation.

- Gaussian blur filter
- Interpolation

4.4.1 Gaussian Blur Filter

Dieser Filter kann sehr grob so beschrieben werden: Man ordnet ein Pixel dabei in seinen Kontext ein. Anders ausgedrückt man berücksichtigt in seinen Berechnungen additiv auch die Umliegenden Pixel bzw. deren Farbe. Diese umliegenden Pixelfarben werden je nach Abstand zu dem im Moment berechneten Pixel anders gewichtet. Diese Stelle bietet sich sehr an um verschiedene Parameter bzw. Skalierungsfaktoren auszuprobieren um einen möglichst realistischen Effekt zu erzeugen. Man geht also Schrittweise die umliegenden Pixel durch um den dazugehörigen Farbwert zu gewichten und auf den finalen Farbwert zu addieren. Dieser Vorgang kann in zwei Schritte unterteilt werden um performanter zu arbeiten. Die beiden Schritte wären, zunächst das horizontale und danach das vertikale blurren durchzuführen. Hier ein Beispiel für den vertikalen Blur (also in y_i-Richtung):

```
vec4 sum = vec4(0.0);
sum += texture2D(RTScene, vec2(vTexCoord.x - 4.0 * blurSize, vTexCoord.y)) * 0.05;
sum += texture2D(RTScene, vec2(vTexCoord.x - 3.0 * blurSize, vTexCoord.y)) * 0.09;
sum += texture2D(RTScene, vec2(vTexCoord.x - 2.0 * blurSize, vTexCoord.y)) * 0.12;
sum += texture2D(RTScene, vec2(vTexCoord.x -       blurSize, vTexCoord.y)) * 0.15;
sum += texture2D(RTScene, vec2(vTexCoord.x                 , vTexCoord.y)) * 0.16;
sum += texture2D(RTScene, vec2(vTexCoord.x +       blurSize, vTexCoord.y)) * 0.15;
sum += texture2D(RTScene, vec2(vTexCoord.x + 2.0 * blurSize, vTexCoord.y)) * 0.12;
sum += texture2D(RTScene, vec2(vTexCoord.x + 3.0 * blurSize, vTexCoord.y)) * 0.09;
sum += texture2D(RTScene, vec2(vTexCoord.x + 4.0 * blurSize, vTexCoord.y)) * 0.05;
finalColor = sum;

// dabei gilt: const float blurSize = 1.0 / [Breite von RTScene]
```

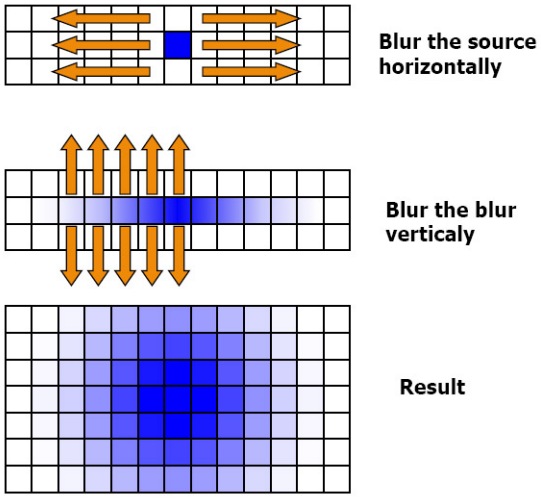
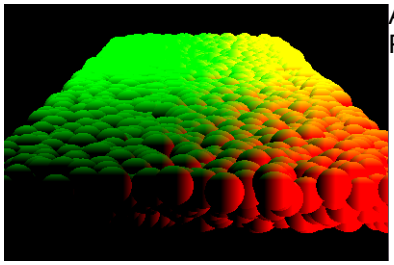


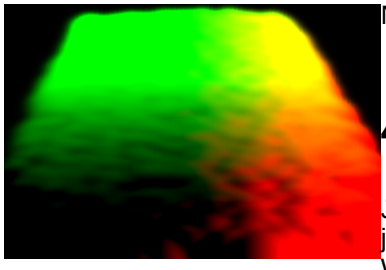
Image taken from ATI's presentation

Dieses Bild veranschaulicht wie der blur filter funktioniert. Im ersten Schritt werden die Farben der horizontal liegenden Nachbarpixel in die neue Farbe einberechnet. Im zweiten Schritt werden die Pixel vertikal geblurt.

Gleichzeitig verdeutlicht dieses Beispielbild auch die Nachteile dieser Art des blurens. Dabei werden nämlich alle Nachbarpixel berücksichtigt. Das bedeutet aber auch, dass am Rand der finalen Struktur die Partikel auch mit dem eigentlichen Hintergrund vermischt werden und nicht nur mit den nächstliegenden Wasserpartikeln, was letztendlich zu Artefakten im finalen Bild führt. Abhilfe schaffen abgeänderte Versionen dieses Filters, z.B. der Bileteral Gaussian filter, der extreme Farbunterschiede (z.B. zwischen Partikel und Hintergrundfarbe) erkennt und bei Bedarf das Pixel nicht blurt. Dieser Filter wurde jedoch nicht in unserer Implementierung benutzt und soll somit hier nicht genauer dargelegt werden.



Auf diesem Bild sieht man die Depthtexture, bei der die Partikelposition als Farbe ausgegeben wurde.



Nun wurde dieselbe Textur mit unserem Gaussian blur filter bearbeitet.

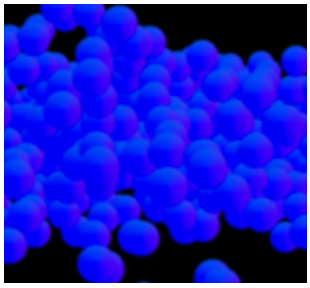
4.4.2 Interpolation

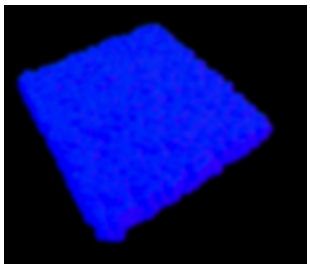
Je nach Größe der darzustellenden Wassermenge und damit einhergehend je nach Anzahl der Wasserpartikel kann es, vor allem bei sich bewegendem Wasser, zu eine solchen Menge von Berechnungen führen, dass der Programmfluss ins stocken gerät. Eine Möglichkeit die Performance zu steigern, wäre die Interpolation, bei der die Bildqualität zugunsten der Performanz sinkt. Bei der Interpolation werden nämlich in Abhängigkeit von dem Abstand der Kamera zu dem Wasser verschieden Aufgelöste Texturen benutzt. Ist die Kamera weit vom Wasser, so soll die hochaufgelöste Textur angezeigt werden. Kommt die Kamera näher, so kann es passieren, dass dann die Kugeln, aus denen das Wasser besteht, immer deutlicher zu sehen sind. Um dies zu verhindern wird bei Annäherung der Kamera an das Wasser eine niedrigaufgelöste Textur angezeigt die weniger Performance verbraucht, aber gleichzeitigt oft den Effekt mitbringt, dass die „verpixelte“ Oberfläche mehr nach Wasser aussieht. Würde man aber die Textur mit der niedrigen Auflösung auch benutzten, wenn die Kamera weit weg ist, würden die in Artefakte, die zuvor angesprochen werden, den Übergang zum Hintergrund viel mehr beschädigen als es bei der hochauflösenden Textur der Fall wäre. Es werden also zwei Versionen der Textur

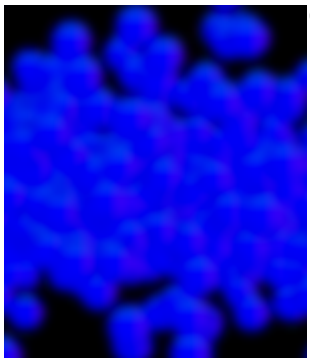
erstellt, die interpoliert werden soll. Die eine Textur soll dabei eine hohe Auflösung haben die andere eine etwa halb- oder sogar ¼ so große Auflösung. Beide Texturen werden bei Bedarf vorher geblurt.

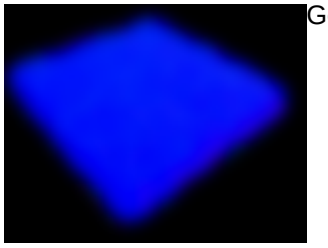
Besonders aufpassen muss man, wenn man bei seinen Berechnungen (z.B. bei der Berechnung des Kugelradius) die Auflösung einbezogen hat. An solchen Stellen muss man den Quelltext an die verschiedenen Auflösungen anpassen (zum Beispiel in Form von Skalierungsfaktoren).

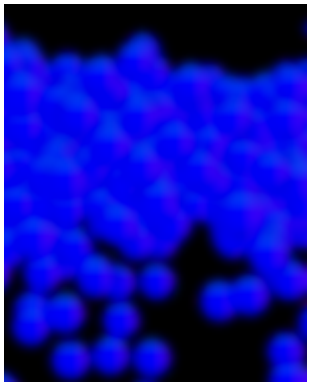
Beide Texturen werden an den Fragmentshader übergeben, der für die Interpolation zuständig ist. In diesem wird nun die finale Farbe in Abhängigkeit des Abstands der Kamera zu dem Punkt berechnet.

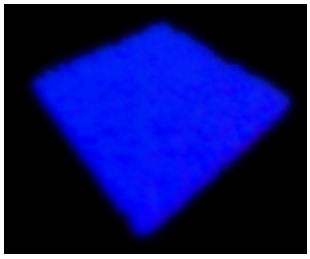

Geblurte, hoch aufgelöste Normalen aus der Nähe.


Geblurte, hoch aufgelöste Normalen aus der Ferne.


Geblurte, niedrig aufgelöste Normalen aus der Nähe.


Geblurte, niedrig aufgelöste Normalen aus der Ferne.


Geblurte, interpolierte Normalen aus der Nähe.



Geblurte, interpolierte Normalen aus der Ferne.

4.5 Normalen

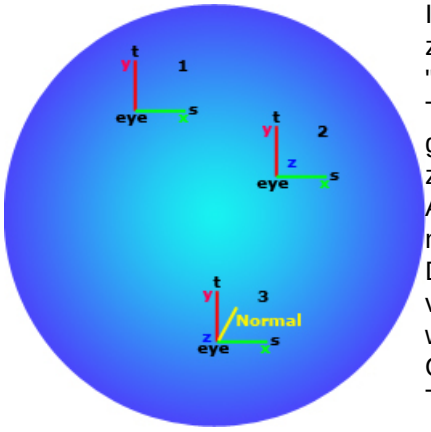
Motivation

Eine wichtige Grundlage für die Beleuchtung einer Szene sind die Normalen. Eine Normale steht senkrecht auf einer Oberfläche und ermöglicht so die Berechnung von Reflexionswinkeln, Winkeln zwischen Sichtlinie und Oberfläche und anderen wichtigen Faktoren. So können unter anderem die Farbe der beleuchteten Oberfläche (z.B. durch Reflexion in eine Cubemap) oder die Lichtintensität (i.d.R. durch Winkelgröße zwischen Lichtquelle und Oberfläche, siehe Beleuchtung) bestimmt werden.

Berechnung

Gewöhnlich werden Normalen einer Oberfläche mit dem normalisierten Kreuzprodukt zwischen zwei Vektoren auf der Oberfläche berechnet:

$$\vec{n} = \frac{\vec{x} \times \vec{y}}{|\vec{x} \times \vec{y}|}$$



Sinnbildliche Darstellung der Normalen

In der Kugelform bestimmen wir diese zwei Vektoren durch die Differenz zwischen dem Ortsvektor des betrachteten Punktes (in der Graphik links "eye") und je einem Punkt in x(s)- bzw. y(t)-Richtung (Welt- bzw. Texturkoordinaten) vom betrachteten Punkt aus. Damit eine möglichst genaue Annäherung an die Kugelform erreicht wird, wird der Abstand zwischen den betrachteten Punkten klein gewählt, z.B. ein Pixel. Die Abweichungen in s- und t-Richtung allein (siehe Graphik links: 1) reichen noch nicht aus, um Normalen einer dreidimensionalen Figur zu bestimmen. Durch die Berechnung auf Texturebene stehen allerdings nur diese in Form von Texturkoordinaten zur Verfügung. Um eine z-Koordinate zu simulieren wird deshalb die Tiefe aus der Tiefentextur hinzugezogen (2). Die Ortsvektoren s, t und eye können jetzt mit Hilfe der z-Koordinaten aus der Tiefe zu zwei Vektoren verrechnet werden:

$$\vec{x} = \vec{s} - e\vec{y}e$$
$$\vec{y} = \vec{t} - e\vec{y}e$$

Um Fehler an den Kanten der Kugel zu vermeiden werden hier die Vektoren in die entgegengesetze Richtung verwendet (eye - s bzw. eye - t). Die beiden so entstandenen Vektoren x und y können wie oben gezeigt für die Berechnung der Normalen (3) benutzt werden.

Die entstandenen Normalen müssen angepasst werden, da die Koordinatensysteme (Texturkoordinaten, Weltkoordinaten) den Ursprung an unterschiedlichen Stellen haben. Deshalb werden der X und der Y Wert der entstandenen Normale negiert. Im letzten Schritt werden die Normalen als Farbwerte in eine Textur geschrieben. Hierfür reicht sogar eine kleine 8-Bit-Textur, da alle Werte durch die Normalisierung zwischen 0 und 1 liegen.

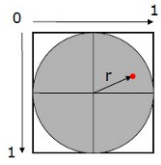
Kommentiertes Codebeispiel

Dies ist ein FragmentShader in der OpenGL Shading Language (GLSL), der die oben erklärte Rechnung durchführt.

```
#version 330

uniform sampler2D depthTex;
uniform float texSize;

in vec2 texCoord;
out vec4 normal;

float offset = 1.0 / texSize;

// Berechnet den Ortsvektor mit Texturkoordinaten,
// Offset und der passenden Tiefe.
vec3 eyePos(vec2 offset, sampler2D tex) {
        vec2 newCoord = texCoord.xy + offset;
        float depth = texture(tex, newCoord).w;
        return vec3(newCoord, depth);
}

void main(void){
        // Fragments die keine Tiefe haben werden verworfen
        if(texture(depthTex, texCoord).w <= 0) discard;

        // Ortsvektor des aktuellen Fragments
        vec3 eye = eyePos(vec2(0), depthTex);

        // Bestimmung des x-Vektors
        vec3 ddx  = eyePos(vec2( offset, 0), depthTex) - eye;
        // Alternativer x-Vektor
        vec3 ddx2 = eye - eyePos(vec2(-offset, 0), depthTex);
        // Wahl des korrekten x-Vektors
        if(abs(ddx.z) > abs(ddx2.z)) ddx = ddx2;

        // Bestimmung des y-Vektors
        vec3 ddy  = eyePos(vec2(0,  offset), depthTex) - eye;
        // Alternativer y-Vektor
        vec3 ddy2 = eye - eyePos(vec2(0, -offset), depthTex);
        // Wahl des korrekten y-Vektors
        if(abs(ddy2.z) < abs(ddy.z)) ddy = ddy2;

        // Kreuzprodukt
        vec3 newNormal = -cross(ddx, ddy);
        // Korrektur aufgrund der Koordinatensysteme
        newNormal.z = -newNormal.z;
        // normalisieren und zurÃ¼ckgeben
        normal = normalize(vec4(newNormal, 0));
}
```

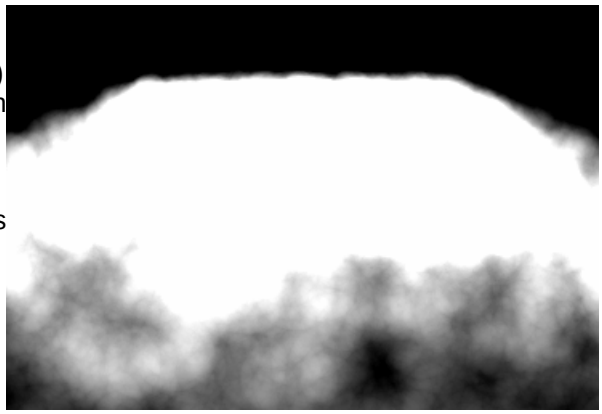
4.6 Thickness Shading

Die Thickness-Textur soll angeben, wie viele Wasser-Partikel an jeder Stelle übereinanderliegen und stellt somit die Wassertiefe aus Betrachtersicht dar.



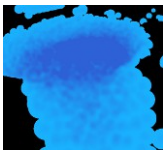
Zunächst werden aus den **`glPoints`**, die der VertexShader erhält, ähnlich wie bei der Tiefentextur Kugeln berechnet. Dazu wird als erstes im Vertex Shader die Kugelgröße anhand des Abstands des Punktes zur Kamera berechnet. Dies dient dazu, dass Kugeln, die näher am Betrachter sind, größer sind und immer kleiner werden, je weiter die Kamera sich entfernt. Da die **`glPoints`** jedoch Quadrate sind, müssen wir noch dafür sorgen, dass wir Kreise bekommen, da man mit Quadraten wohl kaum realistisch Wasser darstellen kann. Der FragmentShader erhält dann die Position des Punktes und die **`glPointCoords`**, die die Position des Fragments innerhalb des **`glPoints`** angeben. Mithilfe der **`glPointCoords`** kann man zunächst den Abstand des aktuellen Fragments zum Mittelpunkt seines **`glPoints`** berechnen und dann alle Fragments discarden, deren Abstand größer als 1 ist. Dadurch werden sozusagen alle Fragments in den Ecken des **`glPoints`** discarded und man behält nur noch einen Kreis. Da wir Kugeln verwenden wollen um das Wasser zu simulieren, besteht nun noch das Problem, dass eine Kugel in der Mitte wesentlich dicker ist als an den Rändern. Dies versuchen wir dadurch anzunähern, dass abhängig vom Abstand zum Mittelpunkt die Thickness eines Fragments zum Rand der Kugel hin geringer wird.

Das Berechnen der Thickness kann man OpenGL übernehmen lassen indem man den Tiefentest deaktiviert, Blending aktiviert und **`glBlendFunc(GL_ONE, GL_ONE)`** als Blending-Funktion angibt. Dadurch werden immer einfach die Farben der übereinanderliegenden Fragments aufeinander addiert. Wenn man beispielsweise in die z-Komponente der Farbe für jedes Fragment eine 0.1 schreibt, würde man bei 2 übereinanderliegenden Fragments einen Wert von 0.2 erhalten. Dadurch ist die maximale Thickness von 1 bei 10 übereinanderliegenden Wasserpartikeln erreicht.



4.7 Beleuchtung

Wasserfarbe



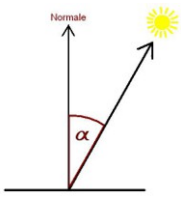
Aufgrund der Lichtabsorption des Wassers erscheint uns tieferes Wasser in einem dunkleren Blau, während flaches Wasser einen wesentlich helleren Blauton hat. Deshalb berechnen wir die Wasserfarbe anhand der thickness. Dazu definieren wir ein dunkles Blau **`(0.2, 0.3, 0.8)`** für thickness=1 und ein helles Blau **`(0.1, 0.7, 1.0)`** für thickness=0 und interpolieren die restlichen Werte dazwischen.

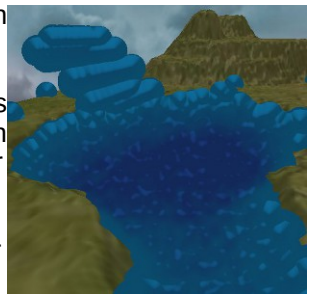
Phong

Ambientes Licht

Das ambiente Licht ist eine Grundhelligkeit, die unabhängig von der Beleuchtung an jeder Stelle vorhanden ist. Wir verwenden zur Berechnung des ambienten Lichts die anhand der thickness berechnete Farbe als ambiente Farbe. Wir haben den ambienten Koeffizienten jedoch ziemlich gering gewählt, da die Cubemap-/Sphere-Reflection auch für eine Art Grundhelligkeit sorgt und zudem einen realistischeren Wassereindruck schafft.

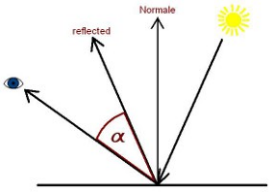
Diffuse Beleuchtung





Bei der diffusen Beleuchtung sind der Lichtquelle zugewandte Flächen heller und der Lichtquelle abgewandte Flächen dunkler.
Für die diffuse Beleuchtung verwenden wir ebenfalls unsere berechnete Wasserfarbe als diffuse Farbe. Um die Helligkeitan einers Fragments zu bestimmen, berechnen wir das Skalarprodukt zwischen dem Vektor von der Position des Fragments zum Betrachter und der Oberflächennormale, was dem Kosinus des Winkels zwischen den Vektoren entspricht. Dieser Wert beträgt eins wenn der Betrachterstandpunkt von der Fragmentposition aus genau in Lichtrichtung liegt und wird kleiner je größer der Winkel zwischen diesen Vektoren wird. Ist der Winkel 90° oder größer erhalten wir null, weil auf diese Stellen kein Licht trifft, da sie dem Licht abgewandt sind.
Berechnet man nun hiermit das diffuse Licht, entstehen schwarzen Flecken auf der Wasseroberfläche, weil einige Normalen nach unten zeigen, was bei den vielen einzelnen Kugeln durchaus passieren kann. Da bei echtem Wasser jedoch wohl kaum die Wasseroberfläche an einer Stelle nach unten zeigt und das Wasser dort schwarz ist, haben wir dieses Skalarprodukt mit der glsl-Funktion **`max`** auf ein Minimum von 0.7 begrenzt.

Spekulare Beleuchtung

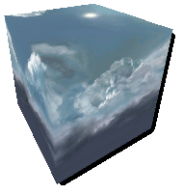


Zur Berechnung des spekularen Lichts berechnet man als erstes die Richtung des an der Oberfläche reflektierten Lichts. Dazu nutzt man die glsl_Funktion **`reflect`** und wendet diese auf den Vektor von der Lichtquelle zur Objektposition an. Nun berechnet man das Skalarprodukt dieses Vektors mit dem Vektor von der Objektposition zum Betrachter. Wie schon beim diffusen Licht gilt auch hier, je kleiner der Winkel zwischen den beiden Vektoren umso näher an 1 liegt das Skalarprodukt. Und somit gibt es dann spekulares Licht, wenn das reflektierte Licht ins Auge des Betrachters fällt. Mit einem relativ hohen Exponenten wird dafür gesorgt, dass es nur einen kleinen Reflexions-Punkt gibt und nicht die halbe Kugel zu leuchten beginnt.

Cubemap-/Skydome-Reflection

Einer der wichtigsten Bestandteile eines realistischen Wassereindrucks, sind Spiegelungen im Wasser, also hauptsächlich die Reflexion des Himmels auf der Wasseroberfläche. Für diese Reflexion gibt es zwei beliebte Möglichkeiten: Die Verwendung einer Cubemap oder das Implementieren eines Skydomes, die hier erklärt werden.

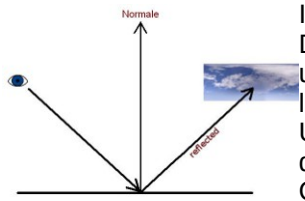
Bei unserem Himmel herrscht leider ein kleines Durcheinander: Zunächst hatten wir die Reflexionen mit Hilfe einer Cubemap implementiert, mussten dann aber alles auf einen Skydome umstellen, da der Himmel der anderen Dastellungs-Gruppe ein Skydome ist und wir diesen eigentlich hätten verwenden sollen. Das hat am Ende jedoch leider nicht mehr geklappt. Deshalb verwenden wir die Cubemap für die Darstellung des Himmels und die Sphere für die Reflexionen auf dem Wasser, da wir nicht wieder alles zur Cubemap zurückändern wollten. Deshalb erklären wir an dieser Stelle beide Methoden.



Cubemap

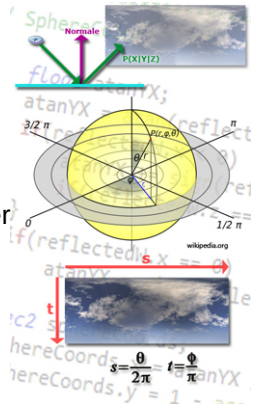
Eine Cubemap kann man als spezielle Textur an OpenGL übergeben. Dafür muss man sechs verschiedene Texturen (eine für jede Würfel-Seite) an unterschiedliche Targets wie z.B. **`GL_TEXTURE_CUBE_MAP_POSITIVE_X`** anbinden und natürlich alle anderen für eine normale Texturanbindung erforderlichen Befehle aufrufen. Man kann sich die Cubemap nun vorstellen wie einen Würfel, auf dem die sechs Himmelstexturen liegen. Diese Cubemap kann man nun wie jede andere Textur als uniform

an einen Shader übergeben, man muss nur im Shader darauf achten, dass es ein **`samplerCube`** und kein **`sampler2D`** ist.



Im Shader kann man nun mit einem **`vec3`** eine Farbe aus der Cubemap auslesen. Der angegebene Vektor wird als vom Mittelpunkt des Würfels ausgehend betrachtet und dann der Farbpixel ausgegeben, der in Richtung des Vektors auf dem Würfel liegt.
Um die Wasserreflexionen aus der Cubemap auszulesen, berechnet man nun zunächst diesen Vektor. Dafür wendet man **`reflect`** auf den Vektor vom Betrachter zur Objektposition und die Normale an und erhält somit den an der Oberfläche reflektierten Vektor, der genau in die Richtung zeigt, in der man den Himmel sehen würde.

Skydome und Kugelkoordinaten



Reflexionswinkel

Die Normalen können benutzt werden, um von der Augenposition (Kameraposition) auf die passende reflektierte Farbe des Himmels zu schließen. Der Vektor vom Auge zum betrachteten Punkt wird an der Normale reflektiert. Der so entstandene Vektor zeigt auf die Stelle des Skydomes, aus der die Farbe ausgelesen wird. Dies ist vergleichbar mit der Reflexion in der Cubemap. Unterschiedlich ist die weitere Verarbeitung.

Vektor und Texturkoordinaten

Da der Skydome aus einer zweidimensionalen Textur besteht, die zur Anzeige um eine Kugel gelegt wird, muss der dreidimensionale Reflexionsvektor in zweidimensionale Texturkoordinaten umgerechnet werden. Dazu werden die Vektorkoordinaten `(x, y, z)` zunächst in Kugelkoordinaten `(r, , )` umgerechnet. Das geschieht mit unten stehender Formel. Üblicherweise werden die Komponenten in anderer Art verwendet (y und z ausgetauscht, siehe Graphik unten). Unser Projekt nutzt jedoch die hier angewandten Konventionen für die Kugelberechnung.

$$\phi = atan2(z,x) = \begin{cases} \arctan(\frac{z}{x}) & , \text{if } x > 0, \\ \arctan(\frac{z}{x}) + \pi & , \text{if } x < 0 \wedge z \geq 0, \\ \arctan(\frac{z}{x}) - \pi & , \text{if } x < 0 \wedge z < 0, \\ sgn(y)\frac{\pi}{2} & , \text{if } x = 0 \end{cases}$$

$$\theta = \arccos \frac{y}{\sqrt{x^2 + y^2 + z^2}}$$

$$r = \sqrt{x^2 + y^2 + z^2}$$

Der Radius `r` muss nicht weiter berücksichtigt werden, da der Skydome auf der Kugel liegt und lediglich die Richtung entscheidend ist. und werden anschließend in Texturkoordinaten `(s,t)` umgerechnet.

$$s = \frac{\theta}{2\pi}$$

$$t = \frac{\phi}{\pi}$$

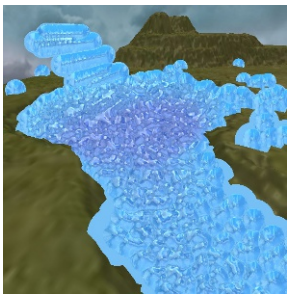
Die Farbwerte können nun aus der Textur ausgelesen werden und als Reflexionsfarbe benutzt werden.

Codebeispiel in OpenGL Shading Language

```
// ...
```

```
const float pi = 3.14159265358979;
// ...

// *************************
// SphereColor

        float atanYX;
        atanYX = atan(reflectedW.z / reflectedW.x);
        if(reflectedW.x < 0) {
                atanYX += sign(reflectedW.z) * pi;
                if(reflectedW.z == 0) atanYX += pi;
        }
        if(reflectedW.x == 0) atanYX = sign(reflectedW.z) * pi / 2.0;

        vec2 sphereCoords;
        float theta = atanYX;
        float phi = acos(reflectedW.y / length(reflectedW));
        sphereCoords.s = theta / (pi * 2.0);
        sphereCoords.t = phi / pi;

        vec3 sphereColor = texture(skyTex, sphereCoords).xyz;

// ...
```

Gesamtbeleuchtung



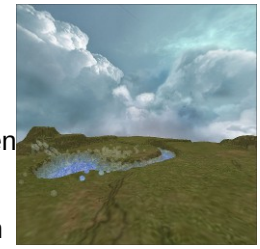
Wenn wir nun die Phong-Beleuchtung und die Reflexionen aufeinanderaddieren, müssen wir zunächst die Reflexionen etwas runterskalieren, da man sonst nichts anderes vom Wasser sehen kann und es auch eher unrealistisch wirkt, wir haben hierfür einen Faktor von 0.7 gewählt. Nun sieht unser fertig beleuchtetes und reflektierendes Wasser jedoch noch nicht aus wie Wasser, denn Wasser ist durchsichtig wenn es flacher ist und man sieht den Untergrund anstelle der Wasserfarbe. Diese Berechnung muss noch für das finale Bild durchgeführt werden.

Vorheriges Kapitel: Thickness Shading **Übersicht** **Nächstes Kapitel: Finales Bild**

4.8 Finales Bild

Als letzten Schritt müssen wir nun noch das Wasser in eine Umgebung integrieren, die wir als fertig beleuchtete Textur erhalten Die folgenden Berechnungen werden gemeinsam mit den Berechnungen für die Beleuchtung alle im selben FragmentShader ausgeführt, um so durch weniger benötigte Texturen eine etwas bessere Performance zu erzielen.

Test-Umgebung



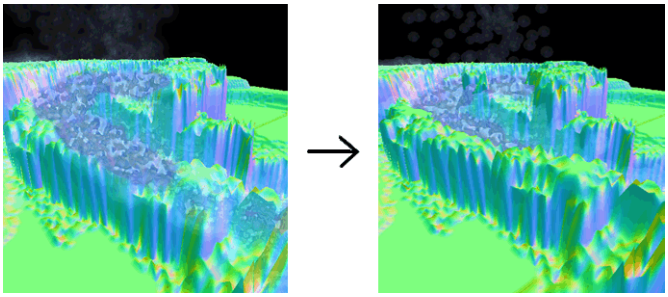
Da wir am Ende des Praktikums leider doch nicht mehr mit den anderen Gruppen mergen konnten (Ausnahme: Simulation), haben wir unser bisheriges Test-Gelände noch etwas verschönert. Dafür haben wir zunächst eine einfache Gras-Textur über den Untergrund gelegt und das Gelände nach dem Phong-Modell beleuchtet. Zusätzlich haben wir noch einen riesigen Würfel um unser Gelände herum erstellt und mithilfe einer Cubemap einen Himmel darauf abgebildet und eine Bewegung der Lichtquelle eingefügt, wodurch auch ein Tag-Nacht-Wechsel

erzeugt wird. Wir haben dann von dem beleuchteten Gelände und vom Himmel jeweils eine Textur erstellt und diese dann an den Beleuchtungs-FragmentShader des Wassers übergeben.

Thickness

Das Durchscheinen des Untergrunds durch flaches Wasser bildet einen der wichtigsten Aspekte der Wasseroptik und wird mit Hilfe der Thickness-Textur berechnet. Hierfür wird die Untergrundfarbe mit **`(1-thickness)`** und die Wasserfarbe mit **`thickness`** multipliziert und dann werden beide aufeinander addiert.

Tiefentest



Ein großes Problem war auch, dass das Wasser einfach über das Gelände "drübergelegt" wurde und man es somit auch durch Berge oder hohe Flussufer hindurch sehen konnte, obwohl das Gelände es hätte verdecken müssen. Um dieses Problem zu lösen haben wir einen manuellen Tiefentest implementiert. Dazu mussten wir zunächst bei der Erstellung der Gelände-Textur zusätzlich zur Farbe die Tiefe berechnen und in der w-Komponente abspeichern. Dadurch kann man dann im Beleuchtungs-Shader die Tiefentexturen von Wasser und Gelände vergleichen und somit das Wasser ausblenden, wenn es vom Gelände verdeckt wird. Verwendet man für dieses Verfahren jedoch die normale Tiefentextur des Wassers, bleiben trotzdem noch verschwommene Ränder vom Wasser übrig, die durchs Gelände durchscheinen. Das liegt daran, dass die Beleuchtung basierend auf den geblurrten Normalen berechnet wird und da beim Blurren der Normalen an den Kanten Ränder entstehen, sieht man diese dann auch noch beim beleuchteten Wasser. Man kann diese Ränder jedoch nicht einfach abschneiden, da dadurch an den Ufern des Wassers der Blur-Effekt fast verschwindet und man wieder nur einzelne Kugeln sieht. Um dieses Problem zu umgehen, haben wir eigens für den Tiefentest eine weitere Tiefentextur erstellt, in der wir den Radius jeder Kugel vervierfacht haben. Dadurch funktioniert der Tiefentest einwandfrei, es bleiben keine Artefakte, die durchs Gelände durchscheinen, und die Ränder des Wassers sehen weiterhin schön geblurrt aus.

4.9 Wasservideo

Download: water_video.mp4[5] (ca. 6.0 MB)

5 http://media2mult.uos.de/pmwiki/fields/cgp12/m2m.d/Wasserdarstellung.Wasservideo/media/video/water_video.mp4

4.10 Wasservideo (Handy-Version)

Download: water_video_small.mp4[6] (ca. 3.2 MB)

6 http://media2mult.uos.de/pmwiki/fields/cgp12/m2m.d/Wasserdarstellung.Wasservideo-klein/media/video/water_video_small.mp4

5. Quellen der Darstellungsgruppen

Online-Quellen (Stand: 26.08.2012)

http://en.wikipedia.org/wiki/Deferred_shading

http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html

http://www.shawnhargreaves.com/DeferredShading.pdf

http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model

http://en.wikipedia.org/wiki/High_dynamic_range_rendering

http://en.wikipedia.org/wiki/Tone_mapping

http://en.wikipedia.org/wiki/Bloom_(shader_effect)

http://en.wikipedia.org/wiki/Gaussian_filter

http://en.wikipedia.org/wiki/Shadow_map

http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html

Arbeiten

High Dynamic Range Rendering in OpenGL, Fabien Houlmann, Stéphane Metz:

```
http://transporter-game.googlecode.com/files/HDRRenderingInOpenGL.pdf
```

Literatur

OpenGL SuperBible: Comprehensive Tutorial and Reference (4th Edition), Addison-Wesley, 2007, Richard S. Wright, Benjamin Lipchak, Nicholas Haemel

```
9 Advanced Buffers: Beyond the Basics
```

OpenGL 4.0 Shading Language Cookbook, Packt Publishing, 2011, David Wolff:

```
Chapter 7: Shadows
```

Abbildungen

Abbildung 1: Framebufferobjekte eines Deferred Shaders (23.08.2012)

```
http ://upload.wikimedia.org/wikipedia/commons/e/e5/Deferred_Shading_FBOs.jpg
```

Alle anderen Abbildungen wurden selbst erstellt.

Zurück

6. Simulation

Wassersimulation

Autoren: Artem Krukow, Andreas Rinas, Benjamin Graf, Oliver Tschesche

- Einleitung
- Physik
- Grid
- Endergebnisse

Gesamtübersicht

6.1 Einleitung

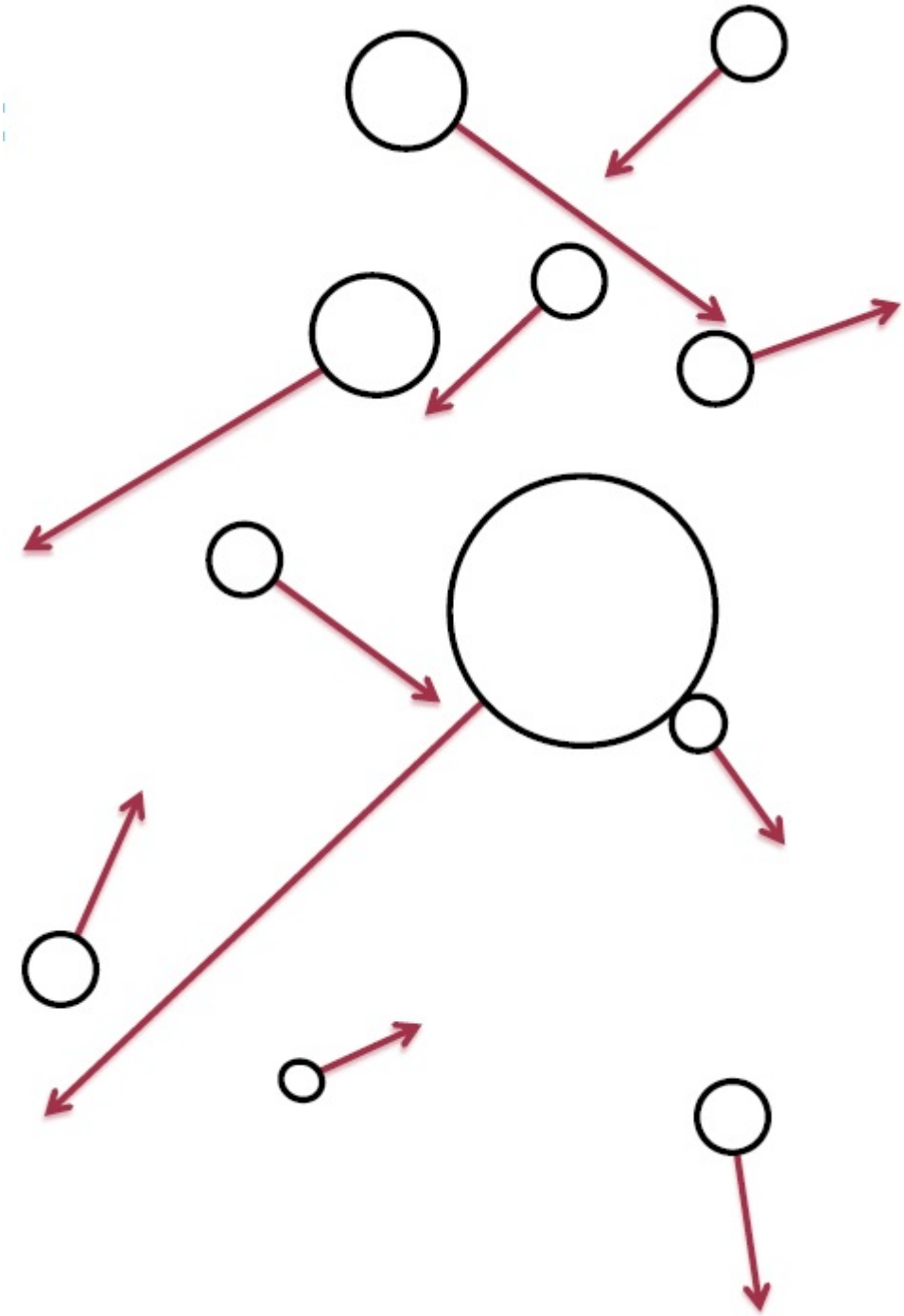
Einleitung

Hintergrund

Wassersimulation ist schon sehr lange ein wichtiger Aspekt in der Computergrafik. Wo zu beginn nur eine simple 2D-Textur auf der Oberfläche angezeigt wurde, war schon einige Zeit später die Wasserqualität das Maß der Dinge, wenn man die Grafik beurteilen wollte. So gab es schon früh Ansätze, die Wasseroberfläche als Dynamische Textur zu gestalten, die in der Höhe variabel ist, um somit Wellengang zu simulieren. Der große Nachteil dieser Methode ist, dass man nur Ebene Oberflächen simulieren kann. Realistisches Wasser-Fließverhalten ist damit nicht möglich. Da die Hardware zur Grafikdarstellung mittlerweile so schnell ist, dass auch bessere Wassersimulationen möglich sind, ist das Bestreben nach selbigen natürlich nahe liegend. Aufgrund der noch recht jungen Geschichte der Flüssigkeitssimulationen gibt es noch recht wenige Ansätze diese zu implementieren. Die dabei gebräuchlichste und vielversprechendste Variante ist dabei das Wasser als Partikelsystem zu betrachten, was der Realität wohl auch am nächsten kommt.

Allgemeines

Die Aufgabe der Arbeitsgruppe Simulation bestand darin, das Fließverhalten von Wasser in Echtzeit zu simulieren. Implementiert wurde das Ganze mit Hilfe eines Partikelsystems in OpenCL. Zu Beginn stellt sich die Frage was man überhaupt unter einem Partikelsystem versteht. Ein Partikelsystem besteht aus einer Menge einzelner Partikeln. Jedes dieser Partikel wird in der Regel durch seine Position, Geschwindigkeit und Ausdehnung repräsentiert.



| **Assoziation eines Partikelsystems** | *Quelle: Vorlesungsskript von Henning Wenke, Computergrafik 2012*

Die zeitlichen Änderungen der Position und Geschwindigkeit werden dabei mit Hilfe des letzten Zeitschrittes berechnet. Die Partikel interagieren mit der Umgebung und oftmals auch untereinander. Ein Partikelsystem besteht meistens aus mehreren tausend Partikeln, die einzeln berechnet werden müssen. Da dies unabhängig voneinander geschieht, lässt sich der Algorithmus parallelisieren. Das geschieht mit Hilfe von OpenCL.

OpenCL (Open Compute Language) ist eine API für einheitliche parallele Programmierung heterogener Systeme, also zum Beispiel für CPUs, GPUs, usw. und für Kombinationen daraus. Die zugehörige Programmiersprache ist OpenCL C . Ursprünglich wurde das Ganze von Apple entwickelt und zusammen mit AMD, IBM, Intel und Nvidia ausgearbeitet und schließlich am 8. Dezember 2008 veröffentlicht. Seit dem wird OpenCL von der Khronos Group betreut. Die zugehörigen Programme werden Kernel genannt. Das Konzept ist Plattform, Betriebssystem und Sprachunabhängig. Es arbeitet direkt mit OpenGL zusammen.

In unserer Aufgabe repräsentiert also jedes Partikel ein Wasserteilchen und das ganze System voluminöses Wasser. Schnell ließen sich naive Konzepte ausarbeiten und implementieren. Diese waren aber sowohl von

der Performance als auch von der Darstellung nicht zufriedenstellend. Am Anfang des Projektes haben wir uns in 2 Gruppen geteilt. Eine Gruppe hat sich mit der Erstellung eines Probeterrains beschäftigt, während die andere Gruppe das Partikelsystem implementiert hat. Anschließend hat sich eine Gruppe mit der Implementierung der Interaktion mit den Partikeln untereinander beschäftigt, während bei der anderen Gruppe die Interaktion mit dem Terrain im Vordergrund stand. Die Interaktion mit dem Terrain geschah am Anfang auf naive Art und Weise. Zuerst wurde mit Hilfe einer Heightmap die Höheninformation an der Stelle des jeweiligen Partikels bestimmt und dann mit der Höhe des Partikels verglichen um zu erfahren, ob sich das Partikel auf dem Boden befindet und mit dem Terrain interagiert oder nicht. Falls eine Interaktion stattfinden sollte wurden die benachbarten Punkte dieses Partikels miteinander verglichen und der tiefste dieser Punkte wurde zur Berechnung der neuen Geschwindigkeit genommen. So fließen die Partikel zum tiefsten Punkt des Terrains. Der größte Nachteil dieser Implementation besteht darin, dass sich am Ende alle Partikel in einem Punkt sammeln und sich dadurch kein Volumen bildet. Eine Optimierung dieser Implementation bestand darin, nicht nur einen Punkt zu betrachten, sondern alle Nachbarpunkte zur Berechnung mit einzubeziehen. Insgesamt gab es keine großen Probleme die Interaktion mit dem Boden zu implementieren.

Größere Schwierigkeiten ergaben sich bei dem Versuch die Interaktion der Partikel untereinander zu implementieren. In einer ersten Implementation wurde die Collide-Methode aus der Computergrafik-Vorlesung verwendet.

/** Diese Methode berechnet aus der Interaktion der Partikel die Geschwindigkeitsänderung

```
  * @param n ist der Vektor, der sich aus der Substraktion der eigenen Position und der Position des
 Partikels ergibt, mit dem man sich vergleicht
  * @param vi & vj sind die Geschwindigkeitsvektoren von sich selbst und dem Partikel mit dem man sich
 vergleichen will
  * @param distance ist die länge des Vektors n
  * @return Geschwindigkeitsänderung des Partikels
  */
```

float4 collide(float4 n, float4 vi, float4 vj, float distance) {

```
    float4 norm = normalize(n);
    float4 relVelo = vj - vi;
    float d = dot(norm, relVelo);
    float4 tanVelo = relVelo - d*norm;
    float s = -SPRING*(SPHERE_RADIUS + SPHERE_RADIUS - distance);
    return SHEAR*tanVelo + DAMPING*relVelo + s*norm;
}
```

Besondere Schwierigkeiten stellte dabei die Optimierung der Rechenzeit dar, denn in einer naiven Implementation würde man jedes Partikel mit jedem Vergleichen, da man nicht weiss welche Partikel sich in der nähe des Vergleichsobjektes befinden und somit Einfluss auf die Geschwindigkeitsänderung nehmen. Das führt zu einer $O(n^2)$ Laufzeit und ist somit sehr ineffizient und schon bei relativ wenigen Partikeln bricht die Framerate stark ein. Um die Performance zu steigern war es nötig ein System zu finden, was es uns ermöglichte nicht mehr alle Partikel untereinander zu vergleichen, sondern nur noch die in Betracht zu ziehen, die sich auch in der Nähe des zu Vergleichenden Objektes befinden. Um das zu erreichen muss man jedes Partikel in ein so genantes GRID einordnen. Was das genau ist und wie es von uns implementiert wurde, wird **hier** erläutert. Dadurch verringert sich die Laufzeit enorm, was man sofort an der Framerate erkennt. Letztendlich haben wir uns für eine Implementation entschieden, möglichst physikalisch Korrekt ist, das so genannte (**SPH**) (Smoothed Particle Hydrodynamics).

Übersicht

6.2 Physik

Physik

Grundlegendes

Wie schon erwähnt, bilden viele Partikel das zu simulierende Wasser. Um ein realistisches Fließverhalten zu simulieren, müssen die Partikel zum einen mit ihrer Umgebung, zum anderen untereinander interagieren können. Bei der Interaktion von Partikeln spielen äußere Kräfte, wie die Schwerkraft oder der Auftrieb (bei Gasen wichtig), und innere Kräfte, wie Druck und Viskosität eine wichtige Rolle. Diese implementierten wir mit dem Ansatz der SPH-Methode (Smoothed Particle Hydrodynamics). Da die Aufgabe darin bestand Wasser zu simuliere, brauchten wir die Auftribskraft nicht einzubeziehen. Die Kräfte, die in unserer Simulation eine Rolle spielen sind: Die Schwerkraft, als einzige äußere Kraft, und der Druck, die Viskosität und die Oberflächenspannung, welche bei Flüssigkeiten eine wichtige Rolle spielt, als innere Kräfte.

Smoothed Particle Hydrodynamics

Ursprünglich wurde das Vorgehen "Smoothed Particle Hydrodynamics"(SPH) für Simulationen von astrophysischen Problemen entwickelt **[The Eurographics Association 2003]**[7]. Die Methode ist aber allgemein genug gehalten, um Flüssigkeiten, wie zum Beispiel Wasser zu simulieren. Da SPH eine Interpolationsmethode für die Partikel ist, werden die berechneten Kräfte, die auf ein Partikel wirken, über einen Glättungskernel mit der Funktion W(**r** - $\mathbf{r}_j$, h) skaliert, **r** ist dabei die Position des betrachteten Partikels und rj die Postion der Partikel, die auf den betrachteten wirken. h ist der Glättungsradius, der bestimmt wie stark der Einfluss der berechneten Kräfte auf das betrachtete Partikel ist. Insgesamt ergibt sich für die physikalische Berechnung des Wassers folgende Gleichung:

$$A_s(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, h)$$

,

wobei j über alle Partikel iteriert, die auf das betrachtete Partikel wirken, Aj die Feldquantität dieser Partikeln ist, also die Kräfte, die auf die Partikel wirken, m_j die Masse der Partikel ist und ρ_j die Dichte der Partikel ist. Diese Gleichung bildet den Ausgang für unsere Berechnung und ist somit auch in der Literatur die im Zusammenhang mit SPH am häufigsten erwähnte Gleichung (s. z.B.**[The Eurographics Association 2003]**[8]. Für das A_j werden entsprechend die Ausdrücke, die für die einzelnen Kräfte wichtig sind, eingesetzt. Des Weiteren sind die erste und Ableitung der Ausgangsgleichung wichtig, in SPH wirken sich diese allerdings lediglich auf den Glättungskernel W aus:

$$\nabla A_s(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(r - r_j, h)$$

und

$$\nabla^2 A_s(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(r - r_j, h)$$

.

Es ist jedoch wichtig, dass das Herausleiten der Gleichungen für Wasser aus dem SPH einige Probleme liefert **[The Eurographics Association 2003]**[9]. Streng genommen stellt SPH nicht sicher, dass physikalische Phänomene wie Kräftesymmetrie und Massen- und Impulserhaltung, gelten.

7 http://fluid3dsjtu.googlecode.com/svn-history/r32/trunk/Paper/sph.pdf
8 http://fluid3dsjtu.googlecode.com/svn-history/r32/trunk/Paper/sph.pdf
9 http://fluid3dsjtu.googlecode.com/svn-history/r32/trunk/Paper/sph.pdf

Lösen der Probleme

Da wir das Wasser mit Hilfe eine Partikelsystems simulieren, in dem die Partikel die gleiche Masse besitzen und die Anzahl der Partikel konstant bleibt, ist das Massenerhaltungsproblem implizit damit gelöst und spielt in unserer Programmierung daher keine entscheidende Rolle. Wichtiger ist die Impulserhaltung. Für die Impulserhaltung in Flüssigkeiten verwenden wir die von Navier und Stokes entwickelte und nach ihnen benannte Gleichung **[The Eurographics Association 2003]**[10]:

$$\rho(\frac{\delta v}{\delta t} + v \cdot \nabla v) = -\nabla p + \rho g + \mu \nabla^2 v$$

, dies ist die vereinfachte Form der Gleichung, in der angenommen wird, die Flüssigkeit kann nicht zusammengedrückt werden, was bei Wasser nahezu der Fall ist. Deshalb können wir diese Form für unsere Wassersimulation ohne Bedenken nutzen. Dabei stellt **g** die äußeren Kräfte dar (in unserem Fall nur die Schwerkraft), **v** ist die Geschwindigkeit der Partikel als Vektor,

$$\mu$$

ein Parameter für die Viskosität und *p* ist das Druckfeld. Hier bekommen wir wieder als Resultat, dass wir ein Partikelsystem benutzen eine Vereinfachung der oben genannten Gleichung als „Geschenk“. Der Ausdruck

$$\frac{\delta v}{\delta t} + v \cdot \nabla v$$

auf der linken Seite der Gleichung darf durch die Zeitableitung der Geschwindigkeit der einzelnen Partikel ersetzt werden, also: D**v**/Dt

Damit bleibt nur noch auf der rechten Seite die Berechnung der aufgeführten Kraftfelder. Die Schwerkraft ist einfach Senkrecht nach unten gerichtet und bleibt immer Konstant. Mathematisch berechnet werden müssen damit nur noch der Druck, die Viskosität und, obwohl die Oberflächenspannung nicht in der Gleichung auftaucht, wird auch diese noch für den Fall für Wasser hinzugenommen. Sie wird aber etwas anders als die übrigen Kräfte berechnet.

Die einzelnen Kräfte

Druck

Der Druck ist die Kraft mit dem größten Einfluss in der gesamten Berechnung. Die Formel für den ist so aufgebaut, dass die Partikel sich von Bereichen mit großem Druck in Richtung der Bereiche mit kleinem Druck begeben. Sind die Partikel also sehr nahe zusammengekommen, entsteht ein hoher Druck und die Partikel stoßen sich ab (abstoßender Druck). Ist um die Partikel herum wenig Druck, ziehen diese sich an (anziehender Druck). Wie schon bereits erwähnt, wird für die Berechnung des Druckes die SPH-Gleichung durch Einsetzen für den Druck in der Navier-Stokes-Gleichung wie folgt umgewandelt:

$$F_i^{Druck} = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(r_i - r_j, h)$$

. Das Problem hierbei ist, dass die Kraftsymmetrie, wie schon vorher erwähnt, nicht garantiert wird. Um dies zu beheben, schlägt **[The Eurographics Association 2003]**[11] vor, die Gleichung in folgende Form zu bringen:

$$F_i^{Druck} = -\sum_j m_j \frac{p_i + p_j}{2 \cdot \rho_j} \nabla W(r_i - r_j, h)$$

. Es gibt auch weitere Ansätze, die Kräftesymmetrie zu erreichen, dies ist der einfachste von ihnen und er genügt unserem Vorhaben. Damit diese Formel berechnet werden kann, muss vorher der Druck p berechnet werden. Dieser setzt sich zusammen aus p =

$$k \cdot \rho$$

, wobei k eine Gaskonstante, also ein Parameter ist. Da der Computer aus die Formeln in jedem Zeitschritt numerisch bestimmt, wird die Berechnung des Druckes aus Stabilitätsgründen in p =

$$k(\rho - \rho_0)$$

, mit

10 http://fluid3dsjtu.googlecode.com/svn-history/r32/trunk/Paper/sph.pdf
11 http://fluid3dsjtu.googlecode.com/svn-history/r32/trunk/Paper/sph.pdf

$$\rho_0$$

als Restdruck (immer konstant), umgewandelt. Im Programmcode sieht dies folgendermaßen aus:

```
...

for (int m = 0; m < num; m++) {
    int other_gid = grid.cells[cellid*max+m];
    float4 other_pos = position[other_gid];
    if(length(mypos.s012-other_pos.s012) <= h)
    {
        sum_neighbours += MASS*W((float4)(mypos.s012,0)-(float4)(other_pos.s012,0),h);

    }
...
    float mass_density = sum_neighbours ;
    float pressure = K_CONSTANT *( mass_density - REST_DENS);
...
```

Natürlich wurden in dem Codeabschnitt nur p und

$$\rho$$

bisher bestimmt. Für

$$F_i^{Druck}$$

gilt:

```
...
f_pressure += (my_pressure + other_pressure) / (2*other_massdens)
     * MASS
     * nW_pres((float4)(mypos.s012,0)-(float4)(other_pos.s012,0),h)
...
```

Viskosität

Die Viskosität bestimmt das Fließverhalten einer Flüssigkeit. Je höher die Viskosität ist, desto zähflüssiger ist damit die Flüssigkeit. Berechnet wird die Viskosität ähnlich wie der Druck, nur dass diese von der Geschwindigkeit der Partikel abhängt. Die fertig abgeleitete Gleichung mit Einbeziehung der Kräftesymmetrie sieht wie folgt aus:

$$F_i^{Vis} = \mu \sum_j m_j \frac{v_i + v_j}{\rho_j} \nabla W^2(r_i - r_j, h)$$

.

So wie die Formeln der Druck- und Viskositätskräfte sich ähneln, so ähnelt sich auch die Implementierung von ihnen,

```
f_viscos += MU_CONSTANT * ((other_vel.s0123-myvel.s0123) / other_massdens)
      * MASS
      * lW_visc(mypos.s0123-other_pos.s0123,h);
```

Oberflächenspannung

Die Oberflächenspannung ist eine Kraft, die die Partikel zusammenhält. Sie ist als Normale der äußeren nach innen zur Flüssigkeit gerichtet und hört bei den inneren Partikeln auf zu wirken. Weiterhin hängt die Stärke der Oberflächenspannung von der Anordnung der Partikel ab. Sind sie im gleichen Abstand im Kreis um das Zentrum der Flüssigkeit angeordnet, so ist die Oberflächenspannung ausbalanciert.

Bei verschiedenen Abständen zu den inneren Partikeln ändert sich die Oberflächenspannung. Bei positiver Krümmung wird die Oberflächenspannung stärker, bei negativer Krümmung wird sie schwächer, die Richtung bleibt unverändert und zeigt zu den inneren Partikeln der Flüssigkeit. Um dieses simulieren zu können, wird

eine weitere Feldgröße eingeführt, die in der Literatur als „Colorfield“ bezeichnet wird **[The Eurographics Association 2003]**[12]. Die Berechnung dafür im wurde auch aus der Ausgangsgleichung von SPH übernommen:

$$c_S(r) = \sum_j m_j \frac{1}{\rho_j} W(r - r_j, h)$$

. Der Gradient von c_S: **n** =

$$\nabla c_S$$

bestimmt die Richtung der Oberflächenspannung zu den inneren Partikeln. Dieser ist also Code sehr leicht zu implementieren:

```
…
gcolor_field = (MASS / other_massdens) * nW((float4)(mypos.s012,0)-(float4)(other_pos.s012,0),h);
…
```

Die Divergenz bestimmt die Krümmung bei der Anordnung der Partikel:

$$\kappa = \frac{\nabla^2 c_S}{|n|}$$

In der Implementierung folgendermaßen umgesetzt:

```
…
lcolor_field = (MASS / other_massdens) * lW((float4)(mypos.s012,0)-(float4)(other_pos.s012,0),h);
…
```

Werden die Gleichungen zusammengesetzt, so wird die Kraft der Oberflächenspannung mit

$$F^{Surface} = -\sigma \nabla^2 c_S \frac{n}{|n|}$$

beschrieben. Für sehr kleine Werte für |n| gibt existiert das Risiko von numerischen Fehlern. Deshalb sollte ein Grenzwert bestimmt werden, dessen Wert |n| erreichen muss, um die Kraft für die Oberflächenspannung zu erreichen. In unserer Implementation liegt der Grenzwert bei 0,75* Partikelradius. Beim

$$\sigma$$

handelt es sich wieder um einen Parameter.

```
…
float  threshold = 0.75 * radius;
float gradient_length = length(gcolor_field);

     if(gradient_length >= threshold)
     {
      f_surface = -SURFACE_TENS * lcolor_field * gcolor_field / gradient_length;
     } else
       {
        f_surface = (float4)(0);
       }
…
```

Das Zusammenrechnen

Zum Schluss werden alle wirkenden Kräfte zusammenaddiert. Für die Beschleunigung der Partikel gilt dann: $\mathbf{a}_i$ = (Summe der inneren Kräfte/ i) *dt + Schwerkraft.

Zu der aktuellen Geschwindigkeit wird im nächsten Zeitschritt die Geschwindigkeitsänderung, die durch die Beschleunigung entstanden ist, hinzugerechnet:
$d\mathbf{v}_i = \mathbf{a}_i * dt$ $\mathbf{v}_i = \mathbf{v}_i + d\mathbf{v}_i * dt$

Die Position des Partikels wird in abhängig seiner Geschwindigkeit im nächsten Zeitschritt aktualisiert:
$\mathbf{r}_i = \mathbf{r}_i + \mathbf{v}_i * dt$

12 http://fluid3dsjtu.googlecode.com/svn-history/r32/trunk/Paper/sph.pdf

Letzte Anmerkungen

Die SPH-Methode ist von der Idee nicht allzu schwer zu implementieren, aber was Probleme bereitet, ist die Wahl der richtigen Parameter (**Endergebnisse**). Die Funktionen der Glättungskernel sind in der SPH-Literatur zahlreich aufgeführt (**[The Eurographics Association 2003]**[13], **[Kelager06]**[14]). Es muss beachtet werden, dass je nach Kraft, die man berechnet, ein anderer Glättungskernel benutzt wird. Welche Glättungskernel die besten Ergebnisse liefern, kann nicht generell gesagt werden, da ist, wie bei den Parametern, viel Probieren notwendig. Zusätzlich sieht man anhand der ganzen Berechnungen, dass SPH mit einem gewaltigen Rechenaufwand verbunden ist. Diese Rechnung wird für jedes Partikel durchgeführt, was bei hohen Partikelzahlen auch aktuelle Rechner schnell an ihre Leistungsgrenzen stoßen lässt. Wie die Effizienz bei der Berechnung gesteigert wurde, wird im nächsten Kapitel behandelt.

< EinleitungÜbersicht

Grid >

6.3 Grid

Grid

Allgemein

Der naive Ansatz zur Berechnung der Interaktionen der Partikel untereinander erfordert einen zur Anzahl der Partikel quadratischen Zeitaufwand, da jedes Partikel mit jedem anderen Partikel die Möglichkeit zur Interaktion bekommt. Diese Möglichkeit ist jedoch unnötig, da der einfache Kollisionsansatz Partikel genau dann aufeinander einwirken lässt, wenn sie sich berühren, das heißt, wenn der Abstand zwischen den Mittelpunkten kleiner ist als die addierten Radien beider Partikel. Werden die Berechnungen der Smoothed-Particle-Hydrodynamics genutzt erledigt diese Aufgabe der Smoothing-Kernel, der in der Regel so eingerichtet ist, dass eine Interaktion zu weit von einander entfernter Teilchen mit dem Faktor Null versehen wird, und somit nicht zur Summe der Interaktionen beiträgt. Aus diesem Grund ist es möglich, die Interaktionen von Partikeln, die sich nicht in räumlicher Nähe zueinander befinden von vorneherein auszuschließen um somit den quadratischen Zeitaufwand der Interaktionsberechnung zu verkleinern.

Um dies zu erreichen haben wir uns an einem Artikel von Simon Green (NVidia) (pdf[15]) orientiert, und das dort geschilderte Uniform Grid in seiner einfachsten Form implementiert. Der Ansatz besteht darin, den Raum in dem sich die Partikel bewegen können in gleichgroße Zellen zu zerlegen. So kann man bei einer Interaktionsberechnung die Zelle des aktuellen Partikels, sowie die 26 Nachbarzellen betrachten und den Großteil der Partikel von der Berechnung ausschließen. Zu bedenken ist hierbei, dass die Größe einer Gridzelle nicht zu klein gewählt werden darf, da sonst die 26 umliegenden Zellen eventuell nicht alle Partikel enthalten, die der Smoothing-Kernel in die Berechnung einfließen lassen würde. Wählt man die Größe zu groß werden wieder vermehrt unnötige Berechnungen durchgeführt. So entspricht also das Grid mit genau einer Zelle dem oben geschilderten naiven Ansatz mit quadratischer Laufzeit. Die zulässige Anzahl an Partikeln in einer Gridzelle ist ebenso von der gewählten Größe der Zellen abhängig. Je größer die Zelle im Vergleich zum Radius der Partikel, desto mehr Speicherplatz muss für eine Zelle vorgehalten werden um die Partikel darin aufzunehmen. Sinnvoll ist es wenn Zellengröße und Durchmesser der beinhalteten Partikel ungefähr gleichgroß gewählt werden.

Implementation

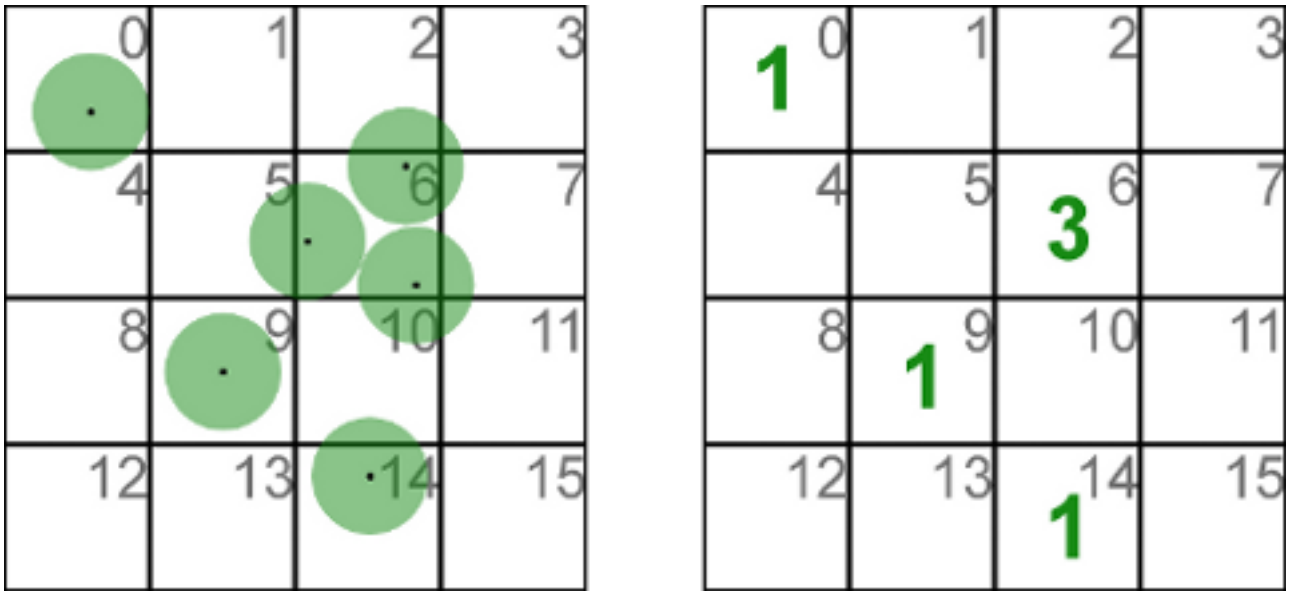
Die Implementation erfolgt konkret über zwei Grids. Das Zählergrid enthält für jede Zelle genau eine nicht-negative Integerzahl, welche die Anzahl der Partikel in dieser Zelle repräsentiert. Das Partikelgrid hält in jeder Zelle die globalen IDs der beinhalteten Partikel. Jeder Partikel wird in den folgenden OpenCL-Kerneln als einzelnes Work-Item bzw Thread gehandhabt, insofern entspricht die globale Partikel-ID der globalen

13 http://fluid3dsjtu.googlecode.com/svn-history/r32/trunk/Paper/sph.pdf
14 http://image.diku.dk/projects/media/kelager.06.pdf
15 http://www.dps.uibk.ac.at/~cosenza/teaching/gpu/nv_particles.pdf

OpenCL-ThreadID. Da sich die Positionen der Partikel ständig verändern, muss das Zählergrid immer zu Beginn einer neuen Berechnung bzw. eines Frames auf Null gesetzt werden. Dazu haben wir einen eigenständigen OpenCL-Kernel implementiert, da es außer über zeitliches Nacheinanderaufrufen von Kerneln keine Möglichkeit zur globalen Synchronisation in OpenCL gibt. Diese ist aber nötig, da die Positionen der Partikel im Positionsspeicher nicht mit den Positionen im Simulationsraum zusammenhängen. Wenn das Zählergrid auf Null zurückgesetzt ist, wird ein weiterer Kernel aufgerufen, welcher die Partikel anhand ihrer Position im Raum in das Zählergrid und anschließend in das Partikelgrid einträgt. Damit jedes Partikel einen Speicherplatz für seine globale ID im Partikelgrid finden kann, wird dazu der jeweilige Zählerstand aus der Zelle des Zählergrids als Offset im Partikelgrid verwendet. Dies erfordert, dass der Zugriff auf - und das Inkrementieren des Zählers nicht simultan von zwei Threads durchgeführt wird. Um dies zu gewährleisten haben wir auf eine OpenCL-Funktion namens *atomic_inc(*pointer_to_variable)* zurückgegriffen, welche simultane Zugriffe nacheinander abarbeitet und somit verhindert, dass mehrere Partikel die selbe Offset im Partikelgrid zugewiesen bekommen.

```
// Funktion fuegt ein Partikel an der Position 'pos' in das Grid 'g' ein
void dg_add_particle(dgrid_t *g, float4 pos)
{
    int cid = dg_cell_id(g, pos, (int4)(0));         // Ermittle Zellen-ID
    if (cid != -1)
    {
        // valid id
        int mp = g->gmax;                            // maximale Anzahl an Partikeln pro Zelle
        int old_num = atomic_inc(g->counter+cid);    // Zahl inkrementieren (Threadsafe)
        if (old_num < mp)
        {
            // Es ist noch Platz im Partikelgrid, fuege globale ID hinzu.
            g->cells[cid*mp+old_num] = get_global_id(0);
        }
    }
}
```

Wenn der Kernel das hinzufügen aller Partikel beendet hat, wird die eigentliche Berechnung in einem weiteren Kernel durchgeführt. Dieser ermittelt anhand der Position des aktuell zu bearbeitenden Partikels die Position im Grid und damit auch die Positionen der umliegenden 26 Zellen, welche dann in einer Schleife durchlaufen werden. In jedem Schleifendurchlauf wird die Anzahl der Partikel aus dem Zählergrid gelesen und die gelesene Anzahl Partikel aus dem Partikelgrid verarbeitet.



Schematische Darstellung des Raums (links) mit zugehörigem Zählergrid (rechts)

Speicherzugriffsprobleme

Zu Beginn der Implementation des Grids, gegen Ende der ersten Woche des Praktikums, wurde das Eintragen in das Grid sowie die anschließende Kollisionsberechnung im selben Kernel durchgeführt. Das kann zu folgendem Ablauf führen: Thread A trägt sein Partikel in das Zählergrid ein. Thread B liest die neue Anzahl aus dem

Zählergrid und anschließend die dazugehörigen Partikel aus dem Partikelgrid. Die globale ID zum letzten Partikel wurde zu diesem Zeitpunkt von Thread A allerdings noch nicht gesetzt. Daher kann die ID entweder eine falsche sein, oder vollkommen außerhalb des Speicherbereichs liegen, in dem die Partikel gespeichert sind. Dieser Fehler wurde erst nach dem Umzug auf einen neuen Rechner bemerkt, da die Grafikkarten der Rechner, an denen dieser falsche Ansatz implementiert wurde kein Problem mit Zugriff auf falsche Speicherstellen hatten. In der Simulation selbst war durch betrachten einzelner Partikel und ihrer Interaktionen der Fehler nicht ersichtlich. Die Beseitigung des Fehlers kostete einige Zeit und Nerven, da die Möglichkeiten zum Debugging auf einer Grafikkarte geringer sind als in einer Umgebung mit Ein- und Ausgabeströmen für Text. Es müssen Bufferobjekte an die Kernel übergeben werden, dort gefüllt und später in der aufrufenden Java-Applikation zurückgelesen und ausgegeben werden. Desto größer war jedoch die Freude als das Grid dann funktionierte wie es sollte.

Erweiterung

Der Raum der durch das Grid abgedeckt wird bestimmt, wo Interaktionen der Partikel stattfinden können. Die erste Implementation ging davon aus, dass sich der Raum auf einen Würfel im positiven Quadranten beschränkt, mit Punktkoordinaten

$$(x, y, z) \quad x, y, z \in [0, 1]$$

Im weiteren Verlauf des Praktikums sollte die Simulation der Partikel in das größere Terrain der anderen Gruppen eingebettet werden. Daher wurde das Grid erweitert, sodass ein Grid-Ursprung sowie eine Grid-Kantenlänge in der Java-Applikation eingestellt werden kann. Diese Daten werden an die OpenCL-Kernel übergeben und finden Eingang in die Funktion zur Berechnung der Gridposition anhand einer Partikelposition im Raum.

Fazit und Ausblick

Die Anwendung eines Grids für die Beschleunigung der Berechnungen ist in der Theorie eine einfache Sache und zeigt gute Ergebnisse. Schwierigkeiten traten bei einem für uns neuen Ansatz der parallelen Datenverarbeitung und den, mit herkömmlichen Mitteln schwierig zugänglichen, Grafikkarten auf. Die Programmierung auf Grafikkarten benötigt schon für einfache Anwendungen ein spezifischeres Verständnis der vorliegenden Hardware und ihrer Eigenschaften. So sind die Grafikkarten der verschiedenen Hersteller, aber auch schon verschiedene Serien eines Herstellers untereinander sehr verschieden im Bezug auf ihre Befehlssätze und Reaktionen auf falsche Programmierung. Es erfordert also eine gewisse Zeit sich in diese Eigenheiten einzuarbeiten. Obwohl die Implementierung des Grids zwei neue Kernel erforderte, einen zum Zurücksetzen des Zählergrids und einen um die Partikel in das Grid einzutragen, war die Geschwindigkeitssteigerung der Berechnung enorm. Die Steigerung ist bereits bei kleinen Partikelanzahlen deutlich sichtbar. Allerdings geht die Methode mit einem großen Speicheraufwand einher, und kann an dieser Stelle noch weiter verbessert werden. Beispielsweise gibt es die Möglichkeit die Position des Grids im Raum dynamisch an die Position der äußersten Partikel anzupassen, was wiederrum einen Nachteil hat, wenn die Partikel über ein weites Gelände verteilt sind. Alle weiteren dynamischen Anpassungen erfordern außerdem mehr Rechenaufwand pro Frame und würden sich daher erst bei größeren Partikelzahlen deutlich bemerkbar machen. Für unsere Zwecke wäre dies also nicht unbedingt ein Vorteil gewesen.

< PhysikÜbersicht

Endergebnisse >

6.4 Endergebnisse

Energebnisse

- Vorgehensweise
- Probleme
- Ergebnis
- Zukünftige Arbeit

< GridÜbersicht

6.4.1 Vorgehensweise

Vorgehensweise

Für unsere Wassersimulation haben wir uns für ein System entschieden, indem jedes Partikel eine Position und eine Geschwindigkeit hat. Zudem interagieren die Partikel untereinander, wenn sie einander nahe genug sind. Die Ausmaße der Partikel sind identisch. Somit ist der Speicherbedarf für die Partikeldaten minimal.

Des Weiteren besitzt jedes Partikel eine Lebenszeit und eine Initialposition in der w-Komponente von Position bzw. Geschwindigkeit gespeichert. Die Lebenszeit ist auf einen Negativwert initialisiert und erhöht sich in jedem Schritt sukzessiv. Sobald diese positiv wird, erscheint das Partikel. Anhand der Initialposition wird das Partikel an der Vordefinierten stelle im Terrain platziert. Durch die festgelegte Indizierung bei Erzeugung der Partikel, wird sichergestellt, dass die Partikel geordnet nacheinander auftauchen, und nicht am selben Ort zwei Partikel zur selben Zeit auftauchen. Letzteres würde eine sehr starke, unrealistische Abstoßung verursachen.

Unsere Implementierung sollte Ursprünglich einen Wasserfall darstellen. Aufgrund von Parametertuning-Problem und selten nachvollziehbaren Verhalten der Partikel untereinander, ist der Abstand des Nachflusses zu lang, so dass der Eindruck von Balken entsteht.

In der Luft wirkt permanent eine nach unten gerichtete Kraft auf die Partikel ein, die die Schwerkraft simuliert. Sobald die Partikel die Oberfläche erreichen, werden sie an selbiger Reflektiert abgeleitet und in der Geschwindigkeit gebremst.

Nähern sich die Partikel einander zunahe an, interagieren sie Untereinander nach der vorgestellten **SPH**-Methode, die ein realistisches Fließverhalten simuliert. Zur Bestimmung des Abstandes der Partikel wird das vorgestellte **Grid** benutzt, welches die Partikel-Partikel vergleiche auf eine Bruchteil der Gesamtmenge reduziert, indem es nur nahe liegende Partikel in Betracht nimmt.

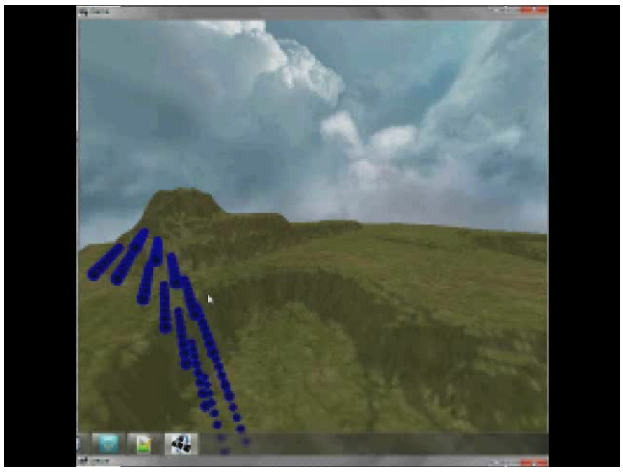
Erreicht die Lebenszeit eines Partikels einen gewissen Schwellenwert, so ‚stirbt' es und wird zurück zur Initialsposition verschoben.

Zurück

6.4.2 Probleme

Probleme

Aufgrund des äußerst fragilen Physiksystems der SPH-Methode, ist es eine sehr schwierige und langwierige Aufgabe, die passenden Parametereinstellungen für die Partikel-Partikel Interaktion zu finden. Schon kleinste Abweichungen können dazu führen, dass die Partikel sich entweder zu stark anziehen, bzw. zu schwach abstoßen, sodass sich diese entweder ‚clustern', also mehrere Partikel auf einem Punkt anhäufen und ineinander verschmelzen, oder nur auf der Oberfläche des Terrains verteilen, ohne ein echtes Volumen zu bilden. Das füllen von Becken zum Beispiel ist dann nicht möglich.



Hier sieht man die Auswirkungen schon sehr kleiner Abweichungen nach unten von den optimalen Parametereinstellungen der Abstoßungskräfte. Man sieht deutlich, wie die Partikel in einander übergehen und auf einer Ebene bleiben, ohne sich voneinander abzustoßen. Somit bildet sich kein Volumen und das Wasser fließt nur sehr schwerfällig (und würde schließlich an der kleinsten Erhebung stehen bleiben).

Andererseits können sich die Partikel sehr leicht auch zu stark abstoßen, sodass sie bei jeglicher Kollision förmlich ‚explodieren' oder aber ‚lediglich' permanent ‚zittern', wodurch das Wasser viel zu unruhig wirkt.

Bei diesem Video sind alle Parameter mit der optimalen Version identisch. Der einziger unterschied besteht in der Erzeugungszeit der Partikel, sodass sie zu schnell hintereinander erscheinen. Dies führt dazu, dass die Abstoßungskräfte zu beginn sehr groß sind und für diesem ‚sprühenden' (bei noch kürzerem Abstand sogar ‚explodierenden') Effekt sorgen.

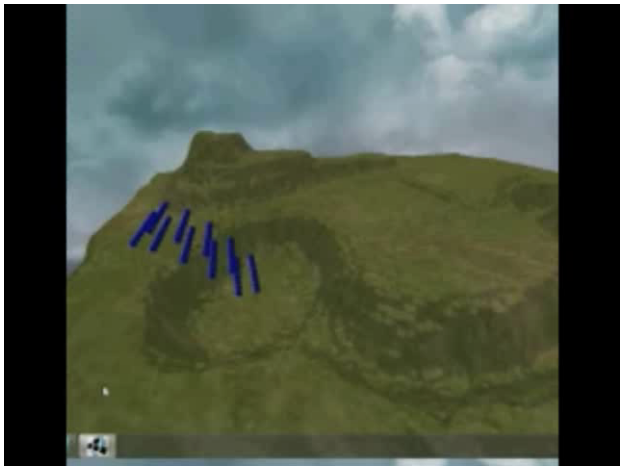
Des Weiteren ist die Kollision mit dem Untergrund problematisch. Hier besteht nur ein kleiner Grad zwischen einem ‚Gummiball- Verhalten' der Partikel, also ein sehr abprallendes Erscheinungsbild, und einer zu hohen Dämpfung, die jegliches Fließverhalten zu Nichte macht. Eine Aufspaltung der vertikalen und horizontalen Komponenten kann hier helfen, ist jedoch für schiefe Umgebungen, wie es bei Landschaften meist der Fall ist, eher ungeeignet. Daher setzen die meisten Implementierungen auf sehr gradlinige Umgebungen, da diese leichter umzusetzen sind.

Zurück

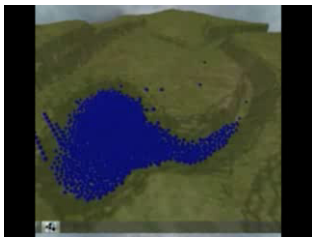
6.4.3 Ergebnis

Ergebnis

Das Ergebnis unserer Implementierung ist eine Wassersimulation, die sich für Wasser in größerem Kontext, sprich in großem Gelände, durchaus eignet. Sie ist flexibel für jede Form von Untergrund und weist ein realistisches Fließverhalten auf (siehe Videos). Dabei ist sie losgelöst von Hilfsprogrammen, wie etwa Blender, welche eine solche Funktionalität bereitstellen.



Man sieht hier, wie die Partikel das Becken füllen und dabei ein Volumen bilden, welches die Wassermassen darstellt. Dabei passt es sich dynamisch der Umgebung an. So kann es für jedes Terrain genutzt werden.



Hier sieht man die (zugegebener Weise noch recht träge) Fließbewegung des Wassers. Dabei füllt sich der Fluss nach und nach und der Wasserspiegel erhöht sich kontinuierlich.

Für feinere Animationen und Umgebungen, wie in etwa das befüllen eines Wasserglases, ist die aktuelle Implementierung noch ungeeignet. Weiteres Parametertuning würde aber auch hier Abhilfe verschaffen. Wir erreichen selbst bei hoher Partikelzahl eine in Echtzeit laufende Animation und können somit auch größere Areale mit Wasser füllen oder feine Animationen sehr detailliert darstellen. Auf Grund des schweren Parametertunings ist das Fliessverhalten noch recht zäh, so dass die Wellenbildung und ähnliches nicht richtig zu Stande kommt. Das fällt jedoch bei ‚normalem' Blickwinkel nicht auf, da man das Terrain normalerweise mit etwas Abstand betrachtet. Ein realistisches Fließverhalten würde einen kleineren Radius der Partikel voraussetzen, was jedoch wiederum bei gleich bleibendem Volumen im Terrain eine deutlich höhere Partikelzahl erfordern würde. Da die Anwendung jedoch in Echtzeit laufen soll, ist in diesem Projekt unsere Lösung wohl der beste Kompromiss.

Zurück

6.4.4 Zukünftige Arbeit

Zukünftige Arbeit

Wasserdarstellung ist und bleibt ein beliebtes Thema in der Computergrafik. Ein gutes Fließverhalten ist die Voraussetzung für einen realistischen Eindruck der Darstellung. Um dies noch besser gewährleisten zu können sind mehrere Ansätze möglich: Zum einen kann man die Partikelzahl weiter erhöhen. Um das System aber weiterhin ich Echtzeit berechnen zu lassen, sind hier noch schnellere, effizientere Berechnungs- und Sortieralgorithmen von Nöten. So kann man z.B. die Partikel innerhalb des Buffers so zu sortieren, dass nahe liegende Partikel auf den selben ‚local Memory' zugreifen können, um somit die Speicherzugriffe zu optimieren.

In der Abhandlung [**Chentanez & Müller 2011**[16]] benutzen die Autoren ab einer gewissen Wassertiefe so genannte ‚tall cells' die den Rechenaufwand dadurch verringern, indem tief liegende Wasserpartikel zu einem länglichen Partikel zusammengefasst werden. Dadurch werden ‚unnötige' Berechnungen, die weit unter der Wasseroberfläche stattfinden und somit kein sichtbaren Einfluss auf das Ergebnis haben, auf ein Minimum reduziert.

Des Weiteren fehlt uns noch die Interaktion mit anderen Objekten als der Terrain-Oberfläche. [**Kelager 2006**[17]] liefert hiefür eine simple Vorgehensweise, die leicht zu implementieren ist.

Zurück

16 http://www.matthiasmueller.info/publications/tallCells.pdf
17 http://image.diku.dk/projects/media/kelager.06.pdf