Fachhochschule Wiener Neustadt
> Technische Studiengänge

Master Thesis

Erstellung eines 3D isometrischen Terraingenerators für Smartphones in Unity3D auf Basis eines 2D isometrischen Terraingenerators

FH-Masterstudiengang
Informatik

Softwarearchitektur und -design

Felber Patrick
Verfasser

26.08.2015
Datum

Fachhochschule Wiener Neustadt
> Technische Studiengänge

Titel der Master Thesis:

Erstellung eines 3D isometrischen Terraingenerators für Smartphones in Unity3D auf Basis eines 2D isometrischen Terraingenerators

Eingereicht von: Patrick Felber BSc

Matrikelnummer: 12102770071

am: **Fachhochschul-Masterstudiengang Informatik**

Vertiefung: Softwarearchitektur und -design

Begutachter: DI(FH) Markus Safar MSc MBA

Wiener Neustadt: 26.08.2015

Ich versichere,

dass ich die Master Thesis selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe und diese Master Thesis bisher weder im In- noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt habe. Die von mir eingereichte schriftliche Version stimmt mit der digitalen Version der Arbeit überein.

Datum	Unterschrift

Kurzzusammenfassung:

In dieser Master-Thesis wird ein 2D isometrischer Terraingenerator erstellt, welcher als Basis für die Erstellung eines 3D isometrischem Terraingenerator dient. Beide Terraingeneratoren basieren dadurch auf denselben Anforderungen und Aussehen. Der Fokus wird dabei auf die Erstellung des 3D Terraingenerators und dessen Leistungsoptimierung gelegt. Herauszufinden ist ob in Unity ein 3D isometrisches Terrain über die selber oder bessere Leistung verfügen kann, als ein 2D isometrisches Terrain. Dazu werden die Terrains auf Android Smartphones getestet.

Schlagwörter:
Terrain, 2D, 3D, Isometrisch, Leistung

Abstract:

This master-thesis describes the development of a 2D isometric terrain generator, which is used as the foundation for the development of a 3D isometric terrain generator. Both terrain generators have the same requirements and appearance. The focus of the thesis is the development of the 3D terrain generator and its optimizations. The reason to do this is to figure out if a 3D terrain can reach the same or better performance than a 2D terrain in Unity. Both terrains will be tested on Android smartphones.

Keywords:
Terrain, 2D, 3D Isometric, Performance

Inhaltsverzeichnis

1. Einleitung **1**
- 1.1. Begriffserklärung . . . 1
- 1.2. Ausgangssituation . . . 1
- 1.3. Forschungsfragen . . . 2
- 1.4. Zielgruppe . . . 3
- 1.5. State of the Art . . . 3
- 1.6. Aufbau der Arbeit . . . 4

2. Umsetzung der Terraingeneratoren **5**
- 2.1. Anforderungen . . . 5
  - 2.1.1. Aussehen . . . 5
  - 2.1.2. Funktionalität . . . 6
  - 2.1.3. Laden von Information . . . 6
- 2.2. 2D Terraingenerator . . . 8
  - 2.2.1. Isometrie und Tiles . . . 8
  - 2.2.2. Erstellung des Terrains . . . 8
  - 2.2.3. Ändern des Terrains . . . 12
- 2.3. 3D Terraingenerator . . . 15
  - 2.3.1. Erstellung eines Meshes in Unity . . . 15
  - 2.3.2. Raster erstellen . . . 18
  - 2.3.3. Felder drehen . . . 21
  - 2.3.4. Chunks . . . 23
  - 2.3.5. Klicken . . . 27
  - 2.3.6. Terrain ändern . . . 29

3. Leistungsmessung **33**
- 3.1. Berechnung der Testergebnisse . . . 33
- 3.2. Errechnung der Felder . . . 33
- 3.3. Einstellungen in Unity . . . 35
- 3.4. Tests . . . 35

3.4.1. Chunks 36
3.4.2. Anzeigen von Bereichen 36
3.4.3. Erhöhungsdauer 37
3.4.4. Generierungsgeschwindigkeit 38
3.4.5. Conclusion 38

4. Zusammenfassung und Ausblick **42**

A. Terraindaten mit 240x240 Feldern **43**

B. Umgesetzter Code des 2D Terraingenerators **83**

C. Umgesetzter Code des 3D Terraingenerators **95**

Abbildungsverzeichnis **109**

Tabellenverzeichnis **110**

Listings **111**

Literaturverzeichnis **111**

Verwendete Abkürzungen

2D	Zweidimensional
3D	Dreidimensional
CSV	Comma-separated values
FPS	Frames per second, Bilder pro Sekunde

Kapitel 1

Einleitung

In diesem Kapitel werden die grundlegenden Punkte behandelt, die zur Erstellung eines 3D isometrischen Terraingenerators benötigt werden.

1.1. Begriffserklärung

Die verwendeten Begriffe in der Master-Thesis sind folgende:

- Axonometrie - Ein Verfahren zur Darstellung von 3D Objekten im 2D Raum.
- Isometrie - Ein spezielles Verfahren der Axonometrie, bei dem die Achsen den selben Abstand zueinander haben.
- Punkt - Ein Punkt ist eine Position im Terrain an dem sich mehrere Eckpunkte eines Feldes treffen.

1.2. Ausgangssituation

Smartphones sind aus dem heutigen Zeitalter nicht mehr wegzudenken. Nahezu jede Person besitzt heutzutage ein Smartphone mit spieletauglicher Hardware. Durch den Lebenszyklus eines Smartphones werden Benutzer nahezu dazu gezwungen diese durch bessere zu ersetzen. Dieser Leistungsfortschritt ermöglicht es Spiele- und App-Entwicklern die Anwendung von immer komplexeren und aufwendigeren Methoden. Wenn man den aktuellen Stand von Smartphone-Spielen im Android Appstore in Tabelle 1.1 betrachtet, setzen trotz Fortschritt der Leistung noch viele Spieleentwickler auf 2D Spiele. Diese nutzen den selben Vorteil wie schon zu Beginn

Names des Spiels	Entwickler	Dimensionen	Kamera Perspektive
Clash of Kings	Elex Wireless	2D	isometrisch
Clash of Clans	Supercell	2D	isometrisch
Game of War - Fire Age	Machine Zone, Inc	2D	isometrisch
Boom Beach	Supercell	2D/3D	isometrisch
Invasion	tap4fun	3D	vollständig 3D
Castle Clash	IGG.COM	2D	isomtrisch
Bloons TD 5	ninja kiwi	2D	top-down
Empires and Allies	Zynga	3D	gesperrter Winkel
Clash of Lords	IGG.COM	2D	isometrisch
DomiNations	NEXON M Inc.	2D	isometrisch

Tabelle 1.1.: Top 10 Stragiespiele im Android Play Store vom 17.07.2015

der 80er Jahre. Sie verwenden die isometrische Perspektive, um dem Benutzer ein 3D Spiel vorzutäuschen.

Zu Beginn der Spieleentwicklung wurden Spiele auf einer eigens entwickelten Spiele-Engine entwickelt. Diese wurden abgelöst durch Engines von Firmen wie Unreal oder Unity, welche die Sicht auf Spiele-Engines revolutioniert haben. Mit Hilfe solcher Spiele-Engines wird jedem die Möglichkeit geboten ein Spiel zu entwickeln, ohne sich dabei mit dem Detailwissen über Grafikkarten und dessen Funktionsweise auseinandersetzen zu müssen.

Dies sind gute Voraussetzungen um 3D Spiele für Smartphones zu entwickeln, da eine wesentliche Hürde wegfällt. Bei der Entwicklung eines Aufbaustrategiespieles für ein Smartphones, ist es relevant zu wissen ob eine Umsetzung in 3D möglich ist und ob diese einfacher und besser ist.

1.3. Forschungsfragen

Die Hauptforschungsfrage dieser Master-Thesis ist es herauszufinden, ob es möglich ist, einen 3D Terraingenerator zu entwickeln, welcher Terrain erstellt mit denselben Anforderungen, Aussehen und mindestens selber Leistung eines 2D isometrischen Terrains. Dazu muss das 3D Terrain in den selben Spielen eingesetzt werden können, wie ein 2D isometrische Terrain. Ist eine solche Umsetzung möglich, stellen sich die Unterfragen:

- Welche Leistung wird benötigt um das generierte Terrain darstellen zu können?
- Wie kann diese Leistung des 3D Terrains verbessert werden?

1.4. Zielgruppe

Diese Master-Thesis soll Spieleentwickler helfen, die vor der Entscheidung stehen, ob sie ein 3D oder 2D Aufbaustrategiespiel entwickeln wollen. Dazu werden in dieser Master-Thesis beide Varianten der Terrains nachvollziehbar implementiert. Die Vor- und Nachteile beider Umsetzungen werden mittels Tests demonstriert, um bei dieser Entscheidung zu helfen.

1.5. State of the Art

Diese Master-Thesis setzt sich mit der Erstellung von Terrains im zweidimensionalen und dreidimensionalen Raum auseinander. Dazu zählen 2D und 3D isometrische Terraingeneratoren.

Thomas Schuster hat sich bereits mit einem ähnlichen Thema auseinandergesetzt und beschreibt in seiner Arbeit [1] die Erstellung einer isometrischen Grafik-Engine die ebenfalls ein 3D Terrain in isometrischer Perspektive generiert. Die Erstellung des Terrains wird ebenfalls wie in dieser Master-Thesis beschrieben. Im Vergleich zu seiner Arbeit, liegt der Schwerpunkt dieser Master-Thesis bei der Veränderung des Terrains. Ebenfalls wurde in seiner Arbeit nicht auf die Möglichkeit eines isometrischen 2D Terrains eingegangen, welches in dieser Master-Thesis vollständig umgesetzt wird. Die Umsetzung des Terrains ist in beiden Arbeiten ähnlich. Die Einheitsbezeichnungen des Terrains wurden anders gewählt. Punkte sind Nodes, Felder sind Tiles und Chunks sind Pages.

Die Verwendung von prozeduraler Geometrie in Unity wird im Buch von Alan Thorn [2][S. 81] beschrieben. Prozedurale Geometrie wird beschrieben als die Veränderung oder Verformung von 3D Objekten zur Laufzeit. Diese Information ist notwendig um prozedural Terrain zu erstellen.

Auf das Thema 2D isometrische Terrains wird im Buch von Charles Kelly [3][S. 301-326] detailliert eingegangenen. Im Kapitel 10 werden Tiled Games erklärt und die isometrische Projektion mit Code Beispielen in C++ beschrieben. In Kapitel 10.8 wird die Erstellung eines 2D isometrischen Terrains erklärt, welche als Basisinformation für den 2D Terraingenerator in dieser Master-Thesis herangezogen wird.

1.6. Aufbau der Arbeit

Um eine Antwort auf die Forschungsfrage in Punkt 1.3 zu finden wird in Punkt 1.5 der aktuelle Stand der Technik für die Entwicklung von 2D und 3D Terraingeneratoren beschrieben. Mit Hilfe dieser Information werden in Punkt 2 die Erstellung der Terraingeneratoren beschrieben. Dazu wird mit der Definition der Anforderungen begonnen. Auf Basis dieser Anforderungen wird in Punkt 2.2 ein 2D isometrischer Terraingenerator in Unity entwickelt. Dazu wird die isometrische Perspektive und Tiles grundlegend beschrieben. In Punkt 2.2.2 wird die Erstellung des 2D isometrischen Terrains beschrieben. In Punkt 2.2.3 wird beschrieben, wie das 2D Terrain dynamisch verändert werden kann.

Der nächste Punkt 2.3 beschreibt die Erstellung des 3D Terraingenerators. Dazu wird mit der Erstellung eines einfachen Meshes begonnen, welches weitergeführt wird zur Erstellung eines Rasters. Dieser Raster wird in Punkt 2.3.4 verwendet um mittels Chunks ein beliebig großes Terrain zu generieren. Die folgenden Punkte in Kapitel 2 beschäftigen sich mit der dynamischen Veränderung des 3D Terrains.

In Punkt 3 werden beide Terraingeneratoren und die Verbesserungen des 3D Terraingenerators Leistungstests unterzogen.

Der vollständige umgesetzte Code der Terraingeneratoren sind im Anhang B und C zu finden.

Kapitel 2

Umsetzung der Terraingeneratoren

In diesem Kapitel wird beschrieben wie der 2D Generator und der 3D Generator umgesetzt werden. Begonnen wird mit den Anforderungen der Terraingeneratoren.

2.1. Anforderungen

Damit die Terraingeneratoren vergleichbar sind ist es notwendig, dass die gleichen Anforderungen an beide Generatoren gestellt werden. Diese setzen sich aus Aussehen und Funktionalität zusammen.

2.1.1. Aussehen

Das Aussehen des Terrains basiert auf alten isometrischen Spielen, wie zum Beispiel Holiday Island von SunFlower. Ein Auszug dieses Spieles wird in Abbildung 2.1 gezeigt. Darin ist zu erkennen, dass für das Terrain ein Raster als Basis verwendet wird. Der Raster besteht aus quadratischen Feldern. Jedes Feld besteht aus zwei Dreiecken. Die Dreiecke werden benötigt um Eckpunkte eines Feldes anzuheben oder abzusenken zu können, ohne dass die anderen drei Eckpunkte mit angehoben oder abgesenkt werden. Jeder Eckpunkt eines Feldes wird als Punkt bezeichnet. Ein Punkt hat vier Nachbarpunkte. In der Abbildung 2.2 sind die vier Nachbarpunkte für den schwarzen Punkt in rot eingezeichnet. Der Höhenunterschied von einem Punkt zu seinen vier Nachbarpunkten kann maximal eine Stufe betragen. Die Größe eines Feldes beträgt 1x1m. Die maximale Stufe beträgt 0,25m.

2.1.2. Funktionalität

Die Funktionalität besteht aus den Basisanforderungen, die das Terrain erfüllen muss, um in Spielen eingesetzt werden zu können. Eine dieser Basisanforderungen ist es zu ermitteln, auf welchen Punkt geklickt wurde. Das ermöglicht dem Benutzer das Terrain anzuheben oder abzusenken. Beim Anheben oder Absenken eines Punktes darf der maximale Stufenunterschied zu den Nachbarpunkten nicht überschritten werden, wodurch die betroffenen Nachbarpunkte mit angehoben oder abgesenkt werden müssen. Weiters muss das Terrain ermitteln, welche Höhe ein Punkt am Terrain hat. Dieser Höhenwert wird in Spielen dazu verwendet um Objekte auf dem Terrain zu platzieren.

Abbildung 2.1.: Screenshot aus dem Spiel Holiday Island von Sunflower

2.1.3. Laden von Information

Um einheitliche Tests in Kapitel 3 durchführen zu können, müssen beide Generatoren auf Basis derselben Daten mit demselben Ladeprozess ein Terrain generieren. Dazu

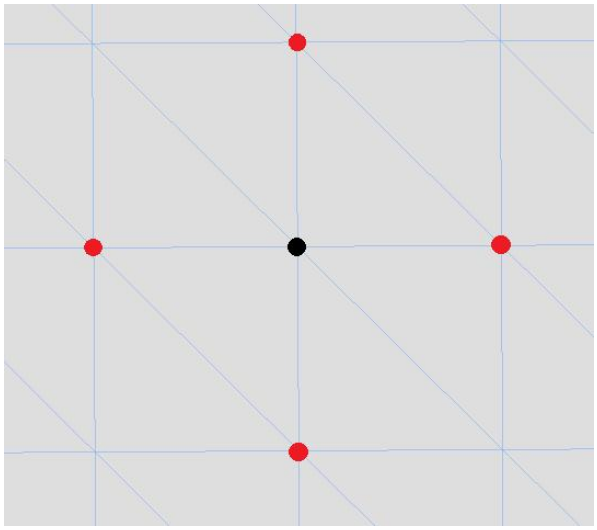

Abbildung 2.2.: Nachbarpunkte eines Punktes

wird den Terraingeneratoren eine CSV Datei zur Verfügung gestellt. Die CSV Datei enthält die Höheninformationen jedes Punktes des 240x240 Felder großen Terrains. Diese CSV Datei wird bei beiden Generatoren mit der Funktion LoadTerrainFile im Codeschnipsel 2.1 eingelesen und in einem Array gespeichert. Die CSV Datei ist im Anhang A zu finden.

```
private void LoadTerrainFile()
{
    TextAsset txt = Resources.Load("Terrain") as TextAsset;
    string[] linesFromfile = txt.text.Split("\n"[0]);

    for (int y = 0; y < sizeY; y++)
    {
        string[] s = linesFromfile[y].Split(';');
        for (int x = 0; x < sizeX; x++)
        {
            terrain[x, y] = float.Parse(s[x]);
        }
    }
}
```

Listing 2.1: Laden der Terrain CSV Datei

2.2. 2D Terraingenerator

In diesem Abschnitt wird die Erstellung des 2D isometrischen Terraingenerators beschrieben. Es wird von Grund auf erklärt wie ein 2D Terrain mit Hilfe der isometrischen Perspektive umgesetzt wird. Dazu werden die Anforderungen aus Punkt 2.1 mit Hilfe des Buches [3][S. 315] umgesetzt.

2.2.1. Isometrie und Tiles

In vielen 2D Spielen werden Tiles eingesetzt. Dazu zählen Spiele wie zum Beispiel side scroller, top-down und isometrische Spiele [3][s. 303]. Tiles sind Bilder mit bestimmter Auflösung die in einem Raster zusammengefügt werden können um ein ganzes Bild zu generieren. Dadurch kann mit wenigen Tiles ein größeres Bild erzeugt werden, wie zum Beispiel ein dynamisches Terrain.

Zur Erstellung solcher Tiles wird die Axonometrie verwendet, welche räumliche Objekte auf einer 2D Fläche darstellen zu können [4]. Eine der häufigsten verwendeten Formen davon ist die Isometrie. Bei der Isometrie besitzen alle drei Achsen denselben Winkel von 120° wodurch die Abmessungen innerhalb des Bildes im selben Verhältnis zueinander stehen [1][S. 7]. Dies ist die Voraussetzung damit nebeneinander angeordnete Tiles als Gesamtbild einen 3D Effekt vortäuschen können, wie zum Beispiel bei einem Terrain.

2.2.2. Erstellung des Terrains

Um ein Terrain in der isometrischen Perspektive darzustellen, wird ein Tile Set wie in Abbildung 2.3 benötigt. Dieses Tile Set enthält jede mögliche Variation eines Feldes. Ein Tile Set in isometrischer Perspektive kann manuell gezeichnet oder mit Generatoren und Editoren erstellt werden [3][S. 305]. Damit aus diesen Tiles ein Terrain erstellt werden kann, müssen die Positionen der Felder berechnet werden. Zur Berechnung der Weltkoordinaten X und Y werden die Formeln 2.1 und 2.2 verwendet, welche aus dem Buch [3][S. 312] adaptiert wurden. In den Formeln werden die Reihe row und die Spalte col des Feldes verwendet. Die Variable TexSize enthält die Anzahl der Pixel der Breite oder Höhe des Tiles. Ein Tile im Tile Set aus Abbildung 2.3 hat 64x64 Pixel, wodurch die TexSize 64 ist. Die Variablen OffsetX und OffsetY sind zum Verschieben des Terrains. Zusätzlich zur errechneten Y Position muss die Erhöhung aus den geladenen Terraindaten aus Punkt 2.1.3 hinzugefügt werden. Das Ergebnis ist die finale Y Position des Feldes [3][S. 319].

Um dies in Unity umzusetzen wird die Funktion LoadTiles in der Klasse Terraingenerator im Listing 2.2 verwendet. Diese Funktion erstellt für jedes Feld im Terrain ein GameObject und fügt ihm einen SpriteRenderer hinzu. Jedes Gameobject wird in einem zwei dimensionalen Array für einen schnellen Zugriff bei späterer Verwendung gespeichert. Um den Sprites eine Position und ein Aussehen zu geben, wird die Funktion UpdateField in der Klasse Terraingenerator im Listing 2.3 verwendet. Diese Funktion speichert die Höheninformation der vier Eckpunkte des aktuellen Feldes. Durch das Vergleichen der vier Höhenpunkte wird erkannt welche Tiles aus dem Tile Set für das aktuelle Feld verwendet wird. Ein Beispiel dafür ist, wenn alle vier Eckpunkte gleich hoch sind, muss das Tile mit einer waagerechten Oberfläche verwendet werden. Die TexId entspricht dabei der Position des Tiles im Tile Set. Die Variable additionalRaise wird benötigt, da bestimmte Tiles im Tile Set eine zu hohe Position oder Offset haben und mit dieser Variable ausgeglichen werden können. Am Ende der Funktion wird die Position des Feldes und die zusätzliche Höhen/Offset berechnet und dem GameObject die Position und das Sprite zugewiesen. Das Gesamtbild der Felder ergibt ein Terrain wie in Abbildung 2.4.

Abbildung 2.3.: Map tile set

$$x = OffsetX - \frac{row \cdot TexSize}{2} + \frac{col \cdot TexSize}{2} \tag{2.1}$$

$$y = OffsetY + \frac{row \cdot TexSize}{4} + \frac{col \cdot TexSize}{4} \tag{2.2}$$

```
private void LoadTiles()
{
    test.StartLoadingTimer();
    for (int row = 0; row < sizeX - 1; row++)
    {
        for (int col = 0; col < sizeY - 1; col++)
        {
            GameObject go = new GameObject(row.ToString() + "," + col.ToString());
            go.transform.parent = this.transform;

            SpriteRenderer sprite = go.AddComponent<SpriteRenderer>();

            gameObjects[col, row] = go;
```

```
            UpdateField(col, row);
        }
    }
    test.StopLoadingTimer();
}
```

Listing 2.2: Erstellen der Tiles

```
private void UpdateField(int col, int row)
{
    float leftBottom = terrain[col, row];
    float leftTop = terrain[col, row + 1];
    float rightBottom = terrain[col + 1, row];
    float rightTop = terrain[col + 1, row + 1];
    float additionallRaise = 0;

    int TexId = 0;

    if (leftBottom < rightTop && leftBottom < rightBottom && leftTop < rightTop &&
        leftTop < rightBottom)
    {
        TexId = 3;
    }
    else if (leftBottom < rightTop && leftBottom < leftTop && rightBottom <
        rightTop && rightBottom < leftTop)
    {
        TexId = 9;
    }
    else if (leftBottom < leftTop && leftBottom < rightTop && leftBottom <
        rightBottom)
    {
        if (rightTop > leftTop)
        {
            TexId = 16;
        }
        else
        {
            TexId = 11;
        }
    }
    else if (rightTop < leftTop && rightTop < leftBottom && rightTop < rightBottom)
    {
        if (leftBottom > leftTop && leftBottom > rightBottom)
        {
            TexId = 18;
            additionallRaise--;
            additionallRaise--;
        }
```

```
    else
    {
        TexId = 14;
        additionallRaise--;
    }
}
else if (rightTop < leftTop && rightTop < leftBottom && rightBottom < leftTop
    && rightBottom < leftBottom)
{
    TexId = 12;
    additionallRaise--;
}
else if (rightTop < leftBottom && rightTop < rightBottom && leftTop <
    leftBottom && leftTop < rightBottom)
{
    TexId = 6;
    additionallRaise--;
}
else if (leftTop < rightTop && leftTop < leftBottom && leftTop < rightBottom)
{
    if (rightBottom > leftBottom)
    {
        TexId = 19;
        additionallRaise--;
    }
    else
    {
        TexId = 7;
        additionallRaise--;
    }
}
else if (rightBottom < rightTop && rightBottom < leftBottom && rightBottom <
    leftTop)
{
    if (leftTop > leftBottom)
    {
        TexId = 17;
        additionallRaise--;
    }
    else
    {
        TexId = 13;
        additionallRaise--;
    }
}
else if (rightTop > leftBottom && rightTop > rightBottom && rightTop > leftTop)
{
    TexId = 1;
}
```

```
    else if (leftBottom > leftTop && leftBottom > rightBottom && leftBottom >
        rightTop)
    {
        TexId = 4;
        additionallRaise--;
    }
    else if (rightBottom > leftBottom && rightBottom > leftTop && rightBottom >
        rightTop)
    {
        TexId = 2;
    }
    else if (leftTop > leftBottom && leftTop > rightBottom && leftTop > rightTop)
    {
        TexId = 8;
    }
    else if (rightTop < leftTop && leftBottom < rightBottom)
    {
        TexId = 10;
    }
    else if (rightTop > leftTop && leftBottom > rightBottom)
    {
        TexId = 5;
        additionallRaise--;
    }
    SpriteRenderer sprite = gameObjects[col, row].GetComponent<SpriteRenderer>();
    sprite.sprite = sprites[TexId];

    float x = ((float)(OFFSET_X - (row * TEXTURE_SIZE / 2) + (col * TEXTURE_SIZE /
        2)));
    float y = (float)(OFFSET_Y + (row * TEXTURE_SIZE / 4) + (col * TEXTURE_SIZE /
        4));

    float yOffSet = terrain[col, row];
    gameObjects[col, row].transform.localPosition = new Vector3(x, y +
        (additionallRaise + yOffSet) * (TEXTURE_SIZE/8));
}
```

Listing 2.3: Setzen der Textur und Positionierung der Tiles

2.2.3. Ändern des Terrains

Um das Terrain anheben oder absenken zu können, muss die Position erkannt werden an der die Änderung durchgeführt werden soll. Dazu wird die Zeile und Spalte eines Feldes mit der Formel 2.3 und 2.4 errechnet. Diese wurden durch das Umformen der Formeln 2.1 und 2.2 ermittelt. Die Variablen x und y entsprechen den Weltkoordinaten des Feldes und die Variable TexSize der Pixelbreite oder -höhe

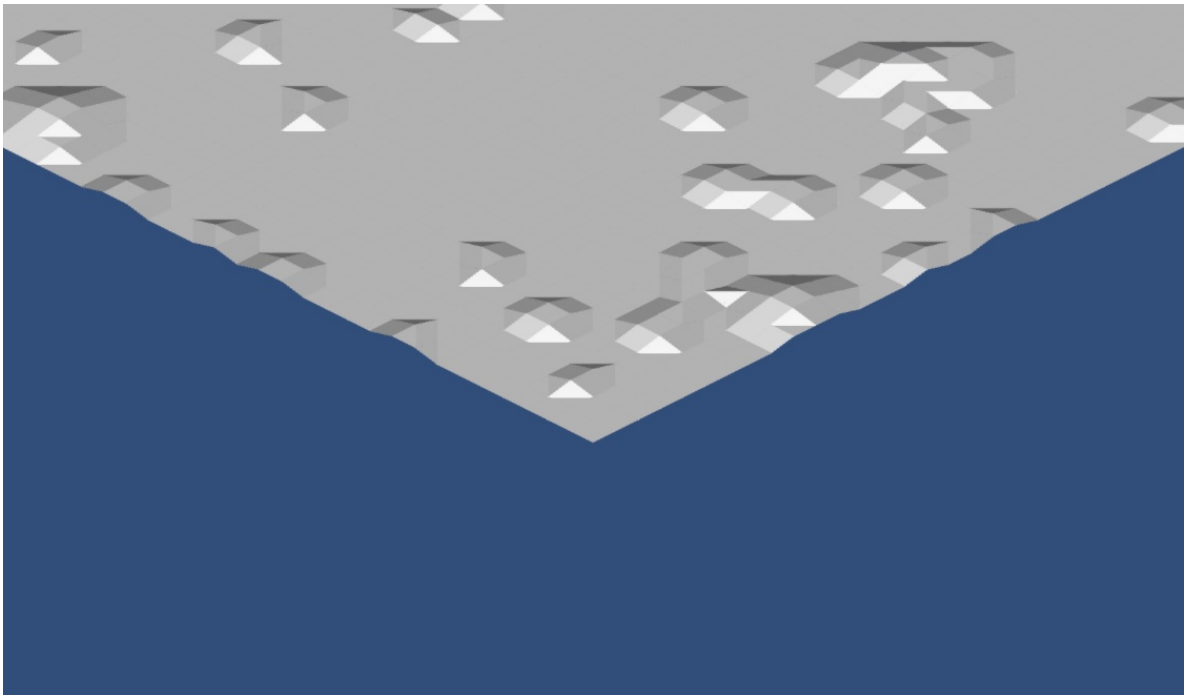

Abbildung 2.4.: 2D isometrisches Terrain nach der Generierung

des Tiles. Wie in den Anforderungen in Punkt 2.1.2 definiert, muss nach Änderung der Höhe eines Punktes die Nachbarpunkte rekursiv mitgeändert werden, damit der maximale Höhenunterschied zu den Nachbarpunkten nicht überschritten wird. Diese Nachbarpunkte müssen beim Anheben nur mit angehoben werden, wenn sie bereits niedriger sind als der anzuhebende Punkt. Beim Absenken müssen Nachbarpunkte nur mitabgesenkt werden, wenn sie höher sind als der anzuhebende Punkt sind. Dies wird in der Funktion ChangeHeight im Listing 2.5 durchgeführt. Dazu wird der Funktion die Zeile und Spalte des zu verändernden Punktes übergeben. Weiters wird der Funktion übergeben, ob der Punkt angehoben oder abgesenkt werden soll. Die Variable count zählt die Aufrufe der Funktion bei der Rekursion mit, für spätere Testzwecke. Nach der Änderung des Terrain Arrays, welches die Höheninformationen enthält, müssen die vier am geänderten Punkt anliegenden Tiles angepasst werden. Dies geschieht in der Funktion UpdateField im Listing 2.3, welche in 2.2.2 erklärt wurde.

$$row = \frac{2y - 2OffsetY - x + OffsetX}{TexSize} \tag{2.3}$$

$$col = \frac{2y - 2OffsetY + x - OffsetX}{TexSize} \tag{2.4}$$

```
Vector3 mousePos = Camera.main.ScreenToWorldPoint(new
    Vector3(Input.mousePosition.x, Input.mousePosition.y,
    Camera.main.transform.localPosition.z));

int row = (int)Mathf.Round((2 * mousePos.y - 2 * OFFSET_Y - mousePos.x + OFFSET_X)
    / TEXTURE_SIZE);
int col = (int)Mathf.Round((2 * mousePos.y - 2 * OFFSET_Y + mousePos.x - OFFSET_X)
    / TEXTURE_SIZE);

int count = ChangeHeight(col, row, true, 0);
```

Listing 2.4: Umrechnung der Mauskoordinaten in Weltkoordinaten

```
private int ChangeHeight(int col, int row, bool increase, int count)
{
    count++;
    if (increase)
    {
        if (terrain[col - 1, row] < terrain[col, row])
        {
            count = ChangeHeight(col - 1, row, increase, count);
        }
        if (terrain[col + 1, row] < terrain[col, row])
        {
            count = ChangeHeight(col + 1, row, increase, count);
        }
        if (terrain[col, row - 1] < terrain[col, row])
        {
            count = ChangeHeight(col, row - 1, increase, count);
        }
        if (terrain[col, row + 1] < terrain[col, row])
        {
            count = ChangeHeight(col, row + 1, increase, count);
        }
        terrain[col, row] = terrain[col, row] + 1;
    }
    else
    {
        if (terrain[col - 1, row] > terrain[col, row])
        {
            count = ChangeHeight(col - 1, row, increase, count);
        }
        if (terrain[col + 1, row] > terrain[col, row])
        {
```

```
            count = ChangeHeight(col + 1, row, increase, count);
        }
        if (terrain[col, row - 1] > terrain[col, row])
        {
            count = ChangeHeight(col, row - 1, increase, count);
        }
        if (terrain[col, row + 1] > terrain[col, row])
        {
            count = ChangeHeight(col, row + 1, increase, count);
        }
        terrain[col, row] = terrain[col, row] - 1;
    }

    UpdateField(col, row);
    UpdateField(col-1, row);
    UpdateField(col, row-1);
    UpdateField(col-1, row-1);

    return count;
}
```

Listing 2.5: Höhenänderung eines Punktes im Terrain

2.3. 3D Terraingenerator

In diesem Kapitel wird die Erstellung des 3D Generators zu den Anforderungen aus Punkt 2.1 beschrieben. Dazu wird zuerst erläutert wie ein Mesh funktioniert und erstellt wird.

2.3.1. Erstellung eines Meshes in Unity

Ein Mesh oder auch Polygonnetz ist ein Netz aus Polygonen [5]. In der Computergrafik werden dazu meist Dreiecks- oder Viereckснetze verwendet. Die Daten eines Meshes bestehen aus zwei Teilen. Die Knotenliste oder auch Vertizes List genannt und die Kantenliste auch Edge List genannt. Die Knotenliste enthält die Positionen aller Punkte. Die Kantenliste enthält die Reihenfolge in der die Punkte verbunden sind.

Um ein Mesh in Unity prozedural zu erstellen, wird die Mesh Klasse verwendet [6]. Ein Mesh benötigt mindestens eine Knotenliste und eine Dreiecksliste [2, S. 81 f.][S. 81]. Um ein Feld prozedural zu erstellen wird im Listing 2.6 ein gebogenes viereckiges Feld, wie in Abbildung 2.5, generiert. Dazu wird die Knotenliste mit vier

Punkten befüllt. Diese repräsentieren die vier Eckpunkte des Feldes. Danach wird in der Dreiecksliste definiert, in welcher Reihenfolge die Punkte miteinander verbunden sind, wodurch zwei Dreiecke gebildet werden. Diese beiden Listen werden einem neu erstellten GameObject mit einem Mesh zugewiesen.

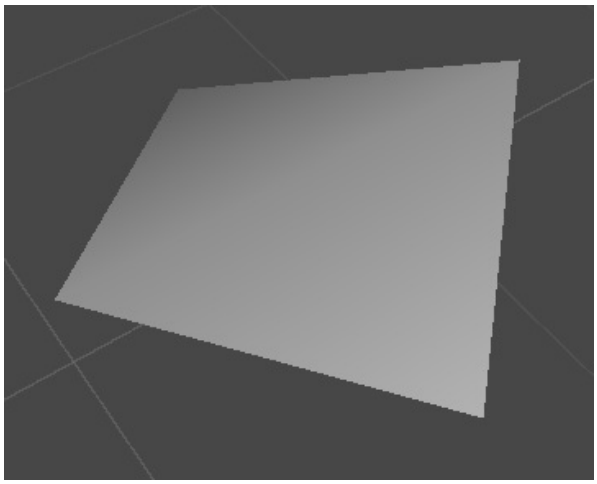

Abbildung 2.5.: Ein Feld mit 4 Vertizes

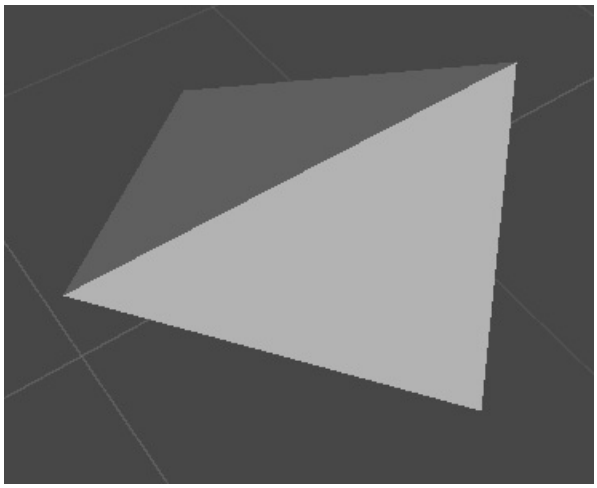

Abbildung 2.6.: Ein Feld mit 6 Vertizes

```
Vector3[] verts = new Vector3[4];
verts[0] = new Vector3(0, 0, 1); //Punkt 1
verts[1] = new Vector3(1, 0.25f, 1); //Punkt 2
verts[2] = new Vector3(0, 0.25f, 0); //Punkt 3
verts[3] = new Vector3(1, 0, 0); //Punkt 4

int[] triangles = new int[6] { 0, 1, 2, 2, 1, 3 };
```

```
GameObject newGo = new GameObject();
MeshRenderer mr = newGo.AddComponent<MeshRenderer>();
mr.material = new Material(Shader.Find("Diffuse"));
MeshFilter meshFilter = newGo.AddComponent<MeshFilter>();
Mesh mesh = new Mesh();
meshFilter.mesh = mesh;
mesh.vertices = verts;
mesh.triangles = triangles;
mesh.RecalculateNormals();
mesh.RecalculateBounds();
```

Listing 2.6: Generierung eines Feldes mit vier Vertizes

Damit dieses Feld dem Aussehen entspricht, welches in Punkt 2.1.1 definiert wurde, müssen die Dreiecke deutlich erkennbar sein. Dies ist möglich indem die Dreiecke getrennt werden. Daher besitzt jedes Dreieck seine eigenen drei Punkte. Dies bezweckt, dass jedes Dreieck individuell beleuchtet wird und daher eine erkennbare Kante entsteht, wie in Abbildung 2.6 zu erkennen ist. Weitere Information zum Thema Belichtung in Unity ist im Buch [7][S. 198] mit dem Thema Rendering Paths zu finden. Um eine solche Kante zu erhalten wird im Listing 2.7 ein Feld mit sechs Punkten umgesetzt. Die Reihenfolge der Punkte die dem Vertizes Array zugewiesen werden ist in der Abbildung 2.9 rot eingezeichnet.

```
Vector3[] verts = new Vector3[6];
verts[0] = new Vector3(0, 0, 1); //Punkt 1
verts[1] = new Vector3(1, 0.25f, 1); //Punkt 2
verts[2] = new Vector3(0, 0.25f, 0); //Punkt 3
verts[3] = new Vector3(1, 0, 0); //Punkt 4
verts[4] = new Vector3(0, 0.25f, 0); //Punkt 5
verts[5] = new Vector3(1, 0.25f, 1); //Punkt 6

int[] triangles = new int[6] {0, 1, 2, 3, 4, 5};

GameObject newGo = new GameObject();
MeshRenderer mr = newGo.AddComponent<MeshRenderer>();
mr.material = new Material(Shader.Find("Diffuse"));
MeshFilter meshFilter = newGo.AddComponent<MeshFilter>();
Mesh mesh = new Mesh();
meshFilter.mesh = mesh;
mesh.vertices = verts;
mesh.triangles = triangles;
mesh.RecalculateNormals();
mesh.RecalculateBounds();
```

Listing 2.7: Generierung eines Feldes mit sechs Vertizes

2.3.2. Raster erstellen

Wie bereits in Abschnitt 2.1.1 als Anforderung definiert wurde, besteht das Terrain aus einem Raster. Zur Erstellung eines Rasters wird im Listing 2.8 ein Mesh wie in Abschnitt 2.3.1 aus mehreren Feldern erstellt. Zuerst werden dazu die Größen definiert und das GameObject zum Zeichnen des Rasters angelegt und vorbereitet. Anschließend wird die Höheninformation wie in 2.1.3 beschrieben geladen. Danach wird errechnet wie viele Eckpunkte und Dreieckspunkte im Raster sind. Die Variablen fieldCountH und fieldCountV geben die Anzahl an Feldern horizontal und vertikal im Raster an. Die Variablen vertCountH und vertCountV, repräsentieren die Anzahl horizontaler und vertikaler Punkte. Diese sind um Eins größer, da zum Beispiel ein Raster aus 10x10 Feldern, 11x11 eindeutige Eckpunkte enthält. Die Erstellung des Rasters geschieht in der Funktion CreateGrid, indem Feld für Feld die Vertizes und Triangles erstellt werden. Abschließend werden die Daten des Meshes aktualisiert.

```
public int fieldCountH = 10;
public int fieldCountV = 10;

public Vector3[] vertices;
public int[] triangles;

public float scaleFieldH = 1.0f;
public float scaleFieldV = 1.0f;
public float stepSize = 0.25f;

MeshFilter meshFilter;
Mesh m;

float[,] terrain;

void Start ()
{
    terrain = new float[fieldCountH, fieldCountV];
    LoadTerrainFile();

    GameObject g = new GameObject("Raster");
    g.isStatic = true;
    Transform transform = (Transform)Component.FindObjectOfType(typeof(Transform));

    MeshRenderer meshRenderer = (MeshRenderer)g.AddComponent(typeof(MeshRenderer));
    meshRenderer.sharedMaterial = new Material(Shader.Find("Diffuse")); ;
    meshFilter = (MeshFilter)g.AddComponent(typeof(MeshFilter));

    m = new Mesh();
    m.Clear();
```

```
    int numTriangles = fieldCountH * fieldCountV * 6;
    int numVertices = fieldCountH * fieldCountV * 6;

    vertices = new Vector3[numVertices];
    triangles = new int[numTriangles];

    CreateGrid();
}
public void CreateGrid()
{
    for (int y = 0; y < fieldCountV; y++)
    {
        for (int x = 0; x < fieldCountH; x++)
        {
            CreateVertices(x, y);
            CreateTriangleField(x, y);
        }
    }
    m.vertices = vertices;
    m.triangles = triangles;

    m.RecalculateNormals();
    m.RecalculateBounds();
    meshFilter.mesh = m;
}
```

Listing 2.8: Code zum Erstellen des Rasters

Zur Erstellung der Vertizes wird im Listing 2.9 die erste Position des Vertizes-Arrays des aktuell zu erstellenden Feldes berechnet. Diese wird mit Hilfe der Koordinaten x und y des aktuellen Feldes und der Größe des Rasters errechnet. Danach werden die Höheninforationen der vier Eckpunkte des Feldes aus dem Höheninformations-Array ausgelesen und gespeichert. Am Ende der Funktion werden die sechs Vertizes angelegt. Das Ergebnis eines Aufrufes dieser Funktion ist ein Feld wie in Abbildung 2.9, welches aus einem linken oberen Dreieck und einen rechten unteren Dreieck besteht. Rot nummeriert ist die Reihenfolge der Vertizes.

```
public void CreateVertices(int x, int y)
{
    int index = (x + (y * (vertCountH))) * 6;

    float heightLeftBot = 0;
    float heightLeftTop = 0;
    float heightRightBot = 0;
    float heightRightTop = 0;

    heightLeftBot = terrain[x, y] * stepSize;
    if (y + 1 < fieldCountV)
```

```
    {
        heightLeftTop = terrain[x, y + 1] * stepSize;
    }
    if (x + 1 < fieldCountH)
    {
        heightRightBot = terrain[ x + 1, y] * stepSize;
    }
    if (y + 1 < fieldCountV && x + 1 < fieldCountH)
    {
        heightRightTop = terrain[x + 1, y + 1] * stepSize;
    }

    vertices[index] = new Vector3(x * scaleFieldH, heightLeftTop, y * scaleFieldV +
        scaleFieldV);
    vertices[index + 1] = new Vector3(x * scaleFieldH + scaleFieldH,
        heightRightTop, y * scaleFieldV + scaleFieldV);
    vertices[index + 2] = new Vector3(x * scaleFieldH, heightLeftBot, y *
        scaleFieldV);

    vertices[index + 3] = new Vector3(x * scaleFieldH + scaleFieldH,
        heightRightBot, y * scaleFieldV);
    vertices[index + 4] = new Vector3(x * scaleFieldH, heightLeftBot, y *
        scaleFieldV);
    vertices[index + 5] = new Vector3(x * scaleFieldH + scaleFieldH,
        heightRightTop, y * scaleFieldV + scaleFieldV);
}
```

Listing 2.9: Funktion zur Erstellung der Punkte des Rasters

Zur Erstellung der Dreiecke wird im Listing 2.10 die Anfangsposition im Dreiecks-Array errechnet. Diese Position ist ebenfalls die Anfangsposition des ersten Vertizes des Feldes. Danach werden die zwei Dreiecke angelegt.

```
public void CreateTriangleField(int x, int y)
{
    int index = (x + (y * fieldCountH)) * 6;

    triangles[index + 0] = index + 0;
    triangles[index + 1] = index + 1;
    triangles[index + 2] = index + 2;

    triangles[index + 3] = index + 3;
    triangles[index + 4] = index + 4;
    triangles[index + 5] = index + 5;
}
```

Listing 2.10: Funktion zur Erstellung der Dreiecke

Das Ergebnis ist ein Raster aus Feldern die jeweils aus 6 Vertices und 2 Dreiecken bestehen. Ein Beispiel dafür ist in Abbildung 2.7 dargestellt, welches 10x10 Felder enthält. Das Endergebnis in 3D mit Sicht auf den Ausgangspunkt ist in Abbildung 2.8 dargestellt.

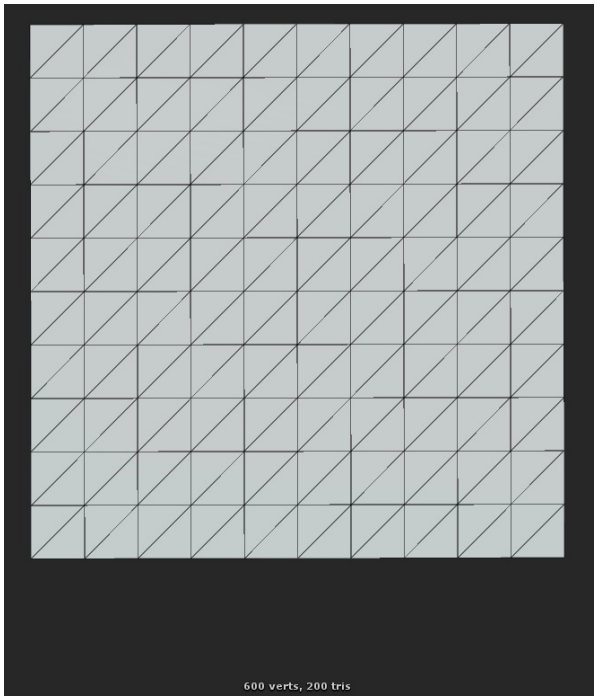

Abbildung 2.7.: Ein Raster mit 10*10 Feldern

2.3.3. Felder drehen

Im erstellten Raster in Abbildung 2.8 und 2.7 sind die Hypotenusen aller Dreiecke parallel. In isometrischen Terrains sind die Hypotenusen nicht immer parallel, welches in Abbildung 2.4 erkennbar ist. Dies kann behoben werden, indem die betroffenen Felder und deren Dreiecke um 90 Grad gedreht werden. Die Ausgangsposition des Feldes, wie in Abbildung 2.9, ist, dass die Hypotenusen beider Dreiecke vom linken unteren Punkt des Feldes zum rechten oberen Punkt des Feldes verlaufen. Im gedrehte Feld, wie in Abbildung 2.10, verlaufen die Hypotenusen der Dreiecke von links oben nach rechts unten. Dies wird im Code realisiert indem die sechs Vertizes in der Funktion CreateVertices im Listing 2.10 in einer anderen Reihenfolge angelegt werden. Die roten Zahlen in den Abbildungen 2.9 und 2.10 repräsentieren die Reihenfolge der Vertizes. Durch den Wert orientationLeft im Listing 2.11 werden die Vertizes in gedrehter Version erstellt. Felder müssen gedreht werden wenn der

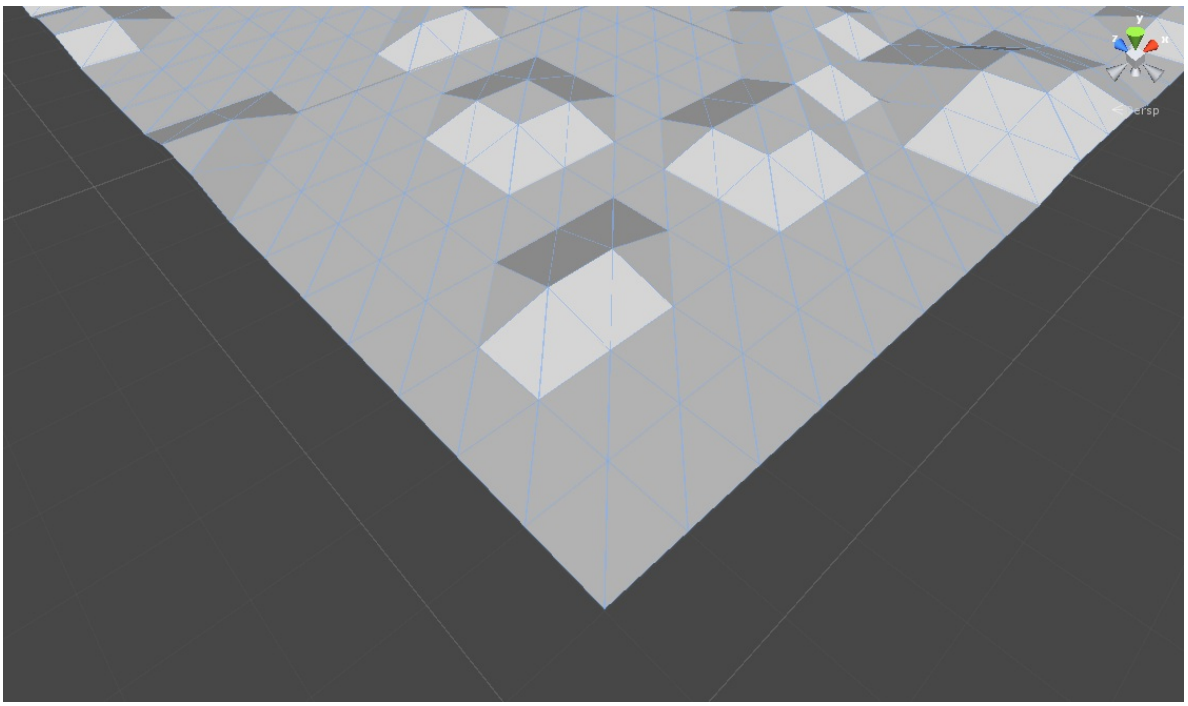

Abbildung 2.8.: Terrain nach dem Erstellen des Rasters und Laden der Höheninformationen

linke untere oder rechte obere Eckpunkt eines Feldes eine niedrigere oder höhere y Koordinate hat als der linke obere und der rechte untere Punkt.

```
bool orientationLeft = false;
if (heightLeftBot < heightLeftTop && heightLeftBot < heightRightBot ||
    heightLeftBot > heightLeftTop && heightLeftBot > heightRightBot)
{
    orientationLeft = true;
}
if (heightRightTop < heightLeftTop && heightRightTop < heightRightBot ||
    heightRightTop > heightLeftTop && heightRightTop > heightRightBot)
{
    orientationLeft = true;
}
if (orientationLeft)
{
    vertices[index] = new Vector3(x * scaleFieldH, heightLeftBot, y * scaleFieldV);
    vertices[index + 1] = new Vector3(x * scaleFieldH, heightLeftTop, y *
        scaleFieldV + scaleFieldV);
    vertices[index + 2] = new Vector3(x * scaleFieldH + scaleFieldH,
        heightRightBot, y * scaleFieldV);
```

```
    vertices[index + 3] = new Vector3(x * scaleFieldH + scaleFieldH,
        heightRightTop, y * scaleFieldV + scaleFieldV);
    vertices[index + 4] = new Vector3(x * scaleFieldH + scaleFieldH,
        heightRightBot, y * scaleFieldV);
    vertices[index + 5] = new Vector3(x * scaleFieldH, heightLeftTop, y *
        scaleFieldV + scaleFieldV);
}
else
{
    vertices[index] = new Vector3(x * scaleFieldH, heightLeftTop, y * scaleFieldV +
        scaleFieldV);
    vertices[index + 1] = new Vector3(x * scaleFieldH + scaleFieldH,
        heightRightTop, y * scaleFieldV + scaleFieldV);
    vertices[index + 2] = new Vector3(x * scaleFieldH, heightLeftBot, y *
        scaleFieldV);

    vertices[index + 3] = new Vector3(x * scaleFieldH + scaleFieldH,
        heightRightBot, y * scaleFieldV);
    vertices[index + 4] = new Vector3(x * scaleFieldH, heightLeftBot, y *
        scaleFieldV);
    vertices[index + 5] = new Vector3(x * scaleFieldH + scaleFieldH,
        heightRightTop, y * scaleFieldV + scaleFieldV);
}
```

Listing 2.11: Funktion zur Erstellung der gedrehten Punkte

Nach dem Drehen der bestimmten Felder ist das Aussehen des 3D Terrains in Abbildung 2.11 dem isometrischen Terrain in Abbildung 2.4 ähnlich. Bei Verwendung einer orthografischen Kamera mit den selben isometrischen Winkeln wie im isometrischen Terrain von x=30° und y=45° und der Belichtung mit denselben Winkeln, ist das Aussehen des 3D Terrains in Abbildung 2.12 dem isometrischen Terrain in Abbildung 2.4 identisch.

2.3.4. Chunks

Mit Hilfe von Chunks wird das Terrain in gleichgroße Teile unterteilt. Dies hat zwei Vorteile. Der erste Vorteil ist, dass das Terrainmesh dadurch nicht die Vertizes-Limitierung erreichen kann. In Unity kann ein Mesh nicht mehr als 65535 Vertizes enthalten. Der Zweite Vorteil ist, dass Frustrum Culling eingesetzt werden kann. Mit Frustrum Culling werden nur Elemente gerendert, welche sich in dem Sichtfeld der Kamera befinden [8, S. 448 f.]. Frustrum Culling wird in Unity standardmäßig eingesetzt [9].

Mit Frustrum Culling wird Leistung eingespart, indem nur sichtbare Chunks des Terrains gerendert werden. Für die Generierung von Chunks wurden zwei Scripts

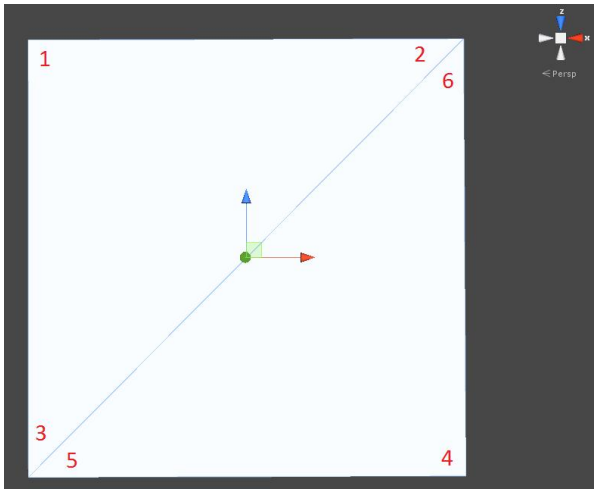

Abbildung 2.9.: Ein Feld in der Ausgangsposition

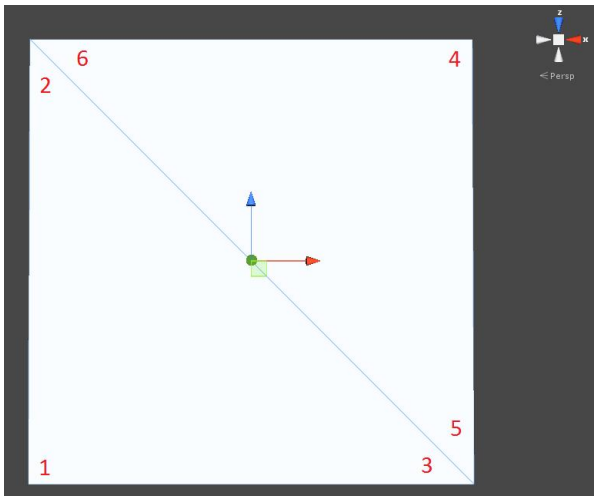

Abbildung 2.10.: Ein Feld nach dem Rotieren

entwickelt. Das TerrainCreator Script erstellt die Chunks, welche das Chunk Script ausführen, und weist ihnen ihre Größen und Position zu. Jeder Chunk erstellt anschließend für seinen Bereich ein Raster, wie im Punkt 2.3.2 definiert ist. Das vollständige TerrainCreator und Chunk Script ist im Anhang C zu finden.

Begonnen wird damit im TerrainCreator, welcher die Variablen für die Größe des Terrains und deren Chunks definiert und die dazu benötigen Arrays anlegt. Nach dem Laden der Höheninformation aus Punkt 2.1.3 werden die Chunk Gamobjects im Listing 2.12 angelegt. Die Variablen chunkCountH und chunkCountV definieren, wieviele Chunks das Terrain beinhalten soll. Die Variablen FieldCountV und FieldCountH definieren, wieviele Felder ein Chunk vertikal und horizontal enthält. Die

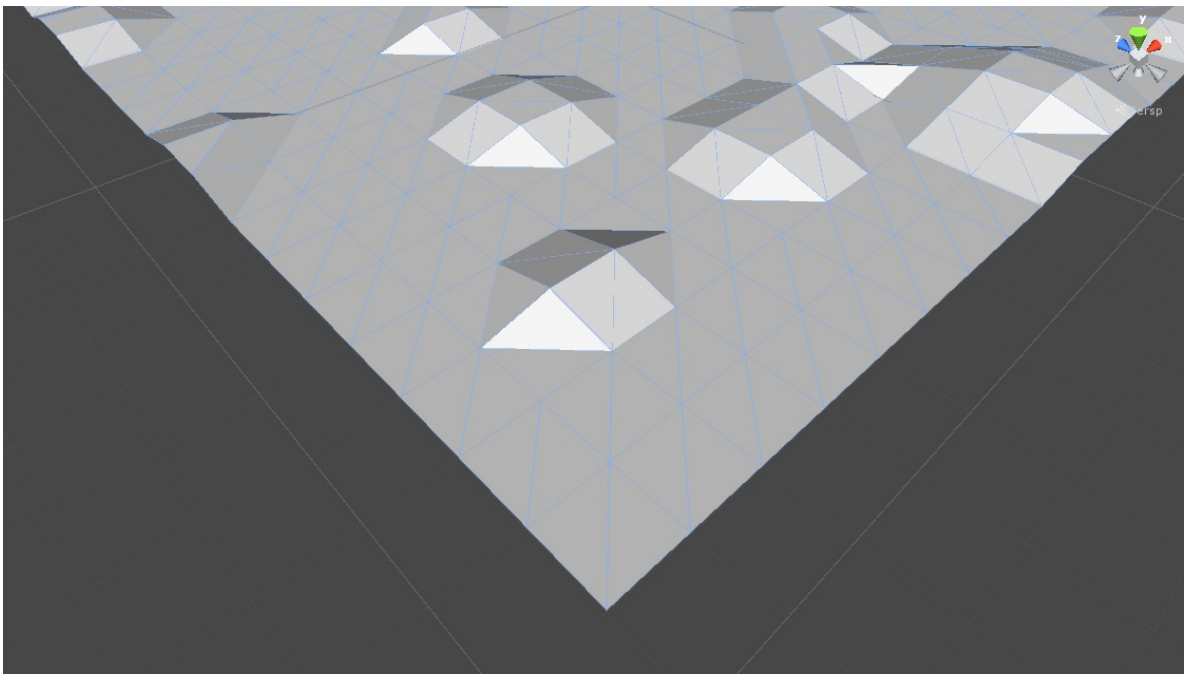

Abbildung 2.11.: 3D Terrain nach dem Drehen der Felder

Variablen scaleX und scaleY sind Multiplikator, welche im Normalfall Eins sind. Das Hinzufügen des Chunks Scripts zum GameObject bewirkt, dass die Start Funktion im Chunk ausgeführt wird. Abschließend wird der Chunk für späteren Zugriff in einem 2D Array gespeichert.

```
public void CreateTerrain()
{
        for(int y = 0; y < chunkCountV;y++)
    {
            for(int x = 0; x < chunkCountH;x++)
            {
                    GameObject g = new
                        GameObject("Chunk:"+x.ToString()+","+y.ToString());
                    g.transform.parent = this.transform;
                    gameObjects1[x,y] = g;
            g.transform.position = new Vector3(x * FieldCountV * scaleX, 0, y *
                FieldCountH * scaleY);
                    Chunk c = (Chunk)g.AddComponent(typeof(Chunk));

                    chunks[x,y] = c;
            }
    }
```

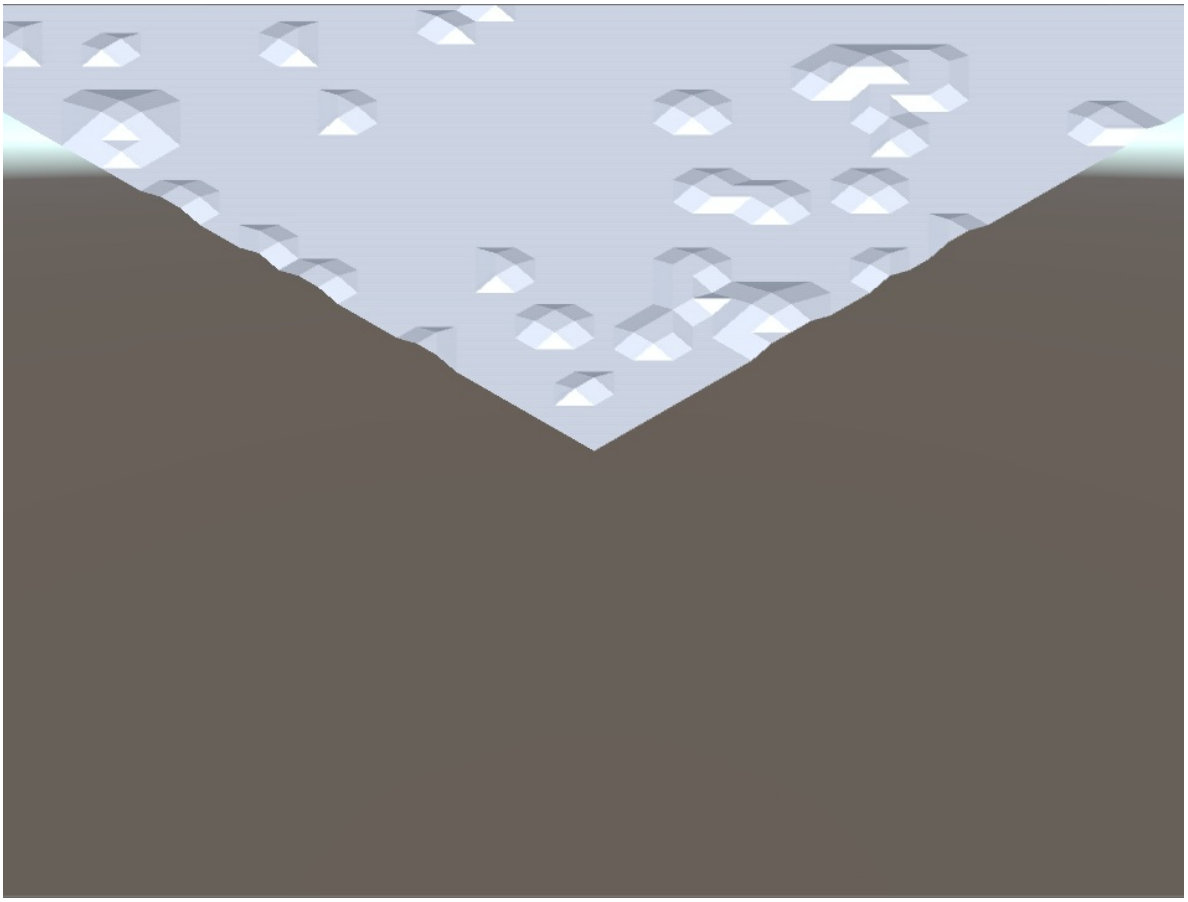

Abbildung 2.12.: 3D Terrain nach dem Drehen der Felder und Verwendung orthografischer Kamera

```
}
```

Listing 2.12: Funktion zur Erstellung der Chunks des Terrains

Das Vorgehen der Start Funktion im Chunk ist das selbe wie beim Erstellen eines Rasters in Punkt 2.7. Der einzige Unterschied ist, dass die Größe und Position vom Terraingenerator vorgegeben werden. In der Funktion CreateVertices im Listing 2.13 werden dazu die Array Positionen und die maximal zulässigen Positionen ermittelt. Diese werden verwendet, um die Höheninformation aus dem eingelesenen Terrain-Array auszulesen. Die zwei Werte im Array chunkPos geben den X und Y Wert an des aktuellen Chunks. Die Erstellung der Vertizes und Triangles funktioniert gleich wie beim Raster erstellen in Punkt 2.7 im Listing 2.11 und im Listing 2.10.

```
public void CreateVertices(int x, int y)
```

```
{
    int index = (x + (y * (fieldCountH))) * 6;
    float heightLeftBot = 0;
    float heightLeftTop = 0;
    float heightRightBot = 0;
    float heightRightTop = 0;

    int arrayPosX = chunkPos[0] * fieldCountH;
    int arrayPosY = chunkPos[1] * fieldCountV;

    int posXMax = creator.maxFieldCountH;
    int posYMax = creator.maxFieldCountV;

    if (arrayPosX + x < posXMax - 1 && arrayPosY + y < posYMax - 1)
    {
        heightLeftBot = creator.terrain[arrayPosX + x, arrayPosY + y] * stepSize;
        if (arrayPosY + y + 1 < posYMax)
        {
            heightLeftTop = creator.terrain[arrayPosX + x, arrayPosY + y + 1] *
                stepSize;
        }
        if (arrayPosX + x + 1 < posXMax)
        {
            heightRightBot = creator.terrain[arrayPosX + x + 1, arrayPosY + y] *
                stepSize;
        }
        if (arrayPosY + y + 1 < posYMax && arrayPosX + x + 1 < posXMax)
        {
            heightRightTop = creator.terrain[arrayPosX + x + 1, arrayPosY + y + 1] *
                stepSize;
        }

        \\ Erstellung der Vertizes
```

Listing 2.13: Funktion zur Erstellung der Dreiecke

2.3.5. Klicken

In Unity wird die Ermittlung eines Punkte mittels Raycasting [10] mit Hilfe der Physics Klasse durchgeführt [11][S. 131]. Dazu müssen Collider zum Mesh hinzugefügt werden, damit die Engine Kollisionen erkennen kann. Um einen exakten Collider auf dem Terrain zu erhalten, muss einen Mesh Collider verwendet werden. Einfache Collider sind in diesem Anwendungsfall nicht geeignet. Mesh Collider sind prozessor-intensiver als einfache Collider [12]. Diese Collider wurden nur für den Test in Punkt 3.4.3 eingefügt. Der Collider kann nach der Erstellung des Gameobjects im Listing 2.12 mit dem Code aus dem Listing 2.14 hinzugefügt werden. Nach der

Erstellung des Rasters im Listing 2.8 muss der Collider aktualisiert werden. Diese Aktualisierung geschieht mit dem Code aus dem Listing 2.15. Dabei werden dem Mesh Collider die Daten des Meshes zugewiesen.

```
g.AddComponent(typeof(MeshCollider));
```

Listing 2.14: Code zur Erstellung des Colliders

```
MeshCollider meshCollider = GetComponent(typeof(MeshCollider)) as MeshCollider;
meshCollider.sharedMesh = meshFilter.mesh;
```

Listing 2.15: Code zum Aktualisieren des Colliders

Eine weitere Möglichkeit, das Klicken zu realisieren ist mittels eigen entwickeltem Raycasting. Das Ziel ist es, aus den 2D Bildschirmkoordinaten den Punkt am Terrain zu finden. Dies wurde in der Funktion GetClickPosition im Listing 2.16 umgesetzt. Dazu wird ein Ray vom Ausgangspunkt des Klicks in die Richtung der Kamera erstellt. Dieser Ray bewegt sich in jeder Iteration um eine Einheit in die Richtung des Terrains. Bei jeder Iteration wird überprüft ob der Ray das Terrain durchdrungen hat, indem die Höhe des Rays mit der Höhe desselben Punktes am Terrain verglichen wird. Dieser Punkt ist eine Annäherung an den echten Punkt, mit der maximalen Abweichung einer Schrittgröße. Um die Genauigkeit zu erhöhen kann die Schrittgröße jumpSize verringert werden, welches mehr Rechenschritte benötigt. Eine weitere Möglichkeit um die Genauigkeit zu erhöhen ist es, nach dem Durchdringen des Terrains mit dem Ray einen Schritt zuvor die Schrittgröße zu verringern. In diesem Terraingenerator wurde nur die einfache Schrittgröße umgesetzt, da die Genauigkeit für diese Anforderung, den nächsten Punkt zu finden, ausreicht. Im Falle, dass der Ray außerhalb der maximalen oder minimalen Höhe ist, wird eine Fehlerposition zurückgegeben.

```
public Vector3 GetClickPosition()
{
    Vector2 clickVector = new Vector2(Input.mousePosition.x, Input.mousePosition.y);
    Ray ray1 = Camera.main.ScreenPointToRay(clickVector);
    Vector3 jump = ray1.origin;

    float jumpSize = 0.1f;

    while (jump.y > lowestPossibleHeight && jump.y < 1000)
    {
        jump = jump + ray1.direction * jumpSize;

        if ((int)jump.x < maxFieldCountH && (int)jump.z < maxFieldCountV &&
            (int)jump.x > 0 && (int)jump.z > 0)
        {
            float currentHeight = terrain[(int)jump.x, (int)jump.z] * stepSize;
```

```
            if (jump.y < currentHeight)
            {
                return jump.RoundUp();
            }
        }
    }
    return Helper.Vector3Error;
}
```

Listing 2.16: Funktion zur Ermittlung des geklickten Punktes

2.3.6. Terrain ändern

Um das Terrain an einem bestimmten Punkt anheben oder absenken zu können, muss das Terrain-Array, welches die Höheninformationen des Terrains enthält, verändert werden. Wie in den Anforderungen 2.1 definiert wurde, darf der Höhenunterschied zwischen einem Punkte und seinem Nachbarpunkten nicht höher als eine Schritthöhe sein, wodurch die Nachbarpunkte rekursiv mit angehoben oder abgesenkt werden müssen. Diese Rekursion geschieht in der Funktion ChangeHeightRecursive im Listing 2.18, welche in der Funktion ChangeHeight im Listing 2.17 zu Beginn ausgeführt wird. Dabei wird jeder Nachbarpunkt des übergebenen Punktes überprüft, ob er beim Absenken niedriger oder beim Anheben höher ist. Ist dies der Fall, wird die Funktion für diese Nachbarpunkte ausgeführt. Nach Überprüfung der Nachbarpunkte wird der Punkt im Terrain-Array erhöht oder abgesenkt. Um die Änderung des Terrains sichtbar zu machen, müssen die Vertizes des betroffenen Chunks angepasst werden. Dazu wird die Position des betroffenen Chunks aus der Position der Punkte errechnet und in cx und cy gespeichert. Die betroffenen Chunks werden in einer Liste chunksThatNeedsUpdates vorgemerkt. Änderungen am Rand eines Chunks betreffen auch den Rand des Nachbarchunks, wodurch diese auch vorgemerkt werden müssen. Für die vorgemerkten Chunks werden nach der Rekursion in der Funktion ChangeHeight im Listing 2.17 die Raster neu erstellt.

```
public void ChangeHeight(int x, int y, bool increase)
{
    SetHeightRecursive(x, y, increase);
    foreach (Chunk c in chunksThatNeedsUpdates)
    {
        c.CreateRaster();
    }
    chunksThatNeedsUpdates.Clear();
}
```

Listing 2.17: Funktion zur Erhöhung oder Absenkung eines Punktes

```
public void ChangeHeightRecursive(int x, int y, bool increase)
{
    if (increase)
    {
        int xAround = x + 1;
        int yAround = y;
        if (xAround < maxFieldCountH)
        {
            if (terrain[x, y] > terrain[xAround, yAround])
            {
                ChangeHeightRecursive(xAround, yAround, increase);
            }
        }

        xAround = x - 1;
        yAround = y;
        if (xAround >= 0)
        {
            if (terrain[x, y] > terrain[xAround, yAround])
            {
                ChangeHeightRecursive(xAround, yAround, increase);
            }
        }

        xAround = x;
        yAround = y - 1;
        if (yAround >= 0)
        {
            if (terrain[x, y] > terrain[xAround, yAround])
            {
                ChangeHeightRecursive(xAround, yAround, increase);
            }
        }

        xAround = x;
        yAround = y + 1;
        if (yAround < maxFieldCountV)
        {
            if (terrain[x, y] > terrain[xAround, yAround])
            {
                ChangeHeightRecursive(xAround, yAround, increase);
            }
        }
        terrain[x, y] += 1;
    }
    else
    {
        int xAround = x + 1;
        int yAround = y;
        if (terrain[x, y] < terrain[xAround, yAround])
```

```
    {
        ChangeHeightRecursive(xAround, yAround, increase);
    }

    xAround = x - 1;
    yAround = y;
    if (terrain[x, y] < terrain[xAround, yAround])
    {
        ChangeHeightRecursive(xAround, yAround, increase);
    }

    xAround = x;
    yAround = y - 1;
    if (terrain[x, y] < terrain[xAround, yAround])
    {
        ChangeHeightRecursive(xAround, yAround, increase);
    }

    xAround = x;
    yAround = y + 1;
    if (terrain[x, y] < terrain[xAround, yAround])
    {
        ChangeHeightRecursive(xAround, yAround, increase);
    }

    terrain[x, y] -= 1;
}

int cx = (int)(x / vertCountH);
int cy = (int)(y / vertCountV);
if (!chunksThatNeedsUpdates.Contains(chunks[cx, cy]))
{
    chunksThatNeedsUpdates.Add(chunks[cx, cy]);
}

cx = (int)((x - 1) / vertCountH);
cy = (int)(y / vertCountV);
if (cx >= 0)
{
    if (!chunksThatNeedsUpdates.Contains(chunks[cx, cy]))
    {
        chunksThatNeedsUpdates.Add(chunks[cx, cy]);
    }
}
cx = (int)(x / vertCountH);
cy = (int)((y - 1) / vertCountV);
if (cy >= 0)
{
    if (!chunksThatNeedsUpdates.Contains(chunks[cx, cy]))
    {
```

```
            chunksThatNeedsUpdates.Add(chunks[cx, cy]);
        }
    }
    cx = (int)((x - 1) / vertCountH);
    cy = (int)((y - 1) / vertCountV);
    if (cx >= 0 && cy >= 0)
    {
        if (!chunksThatNeedsUpdates.Contains(chunks[cx, cy]))
        {
            chunksThatNeedsUpdates.Add(chunks[cx, cy]);
        }
    }
}
```

Listing 2.18: Rekursive Funktion zur Erhöhung oder Absenkung eines Punktes

Kapitel 3

Leistungsmessung

In diesem Kapitel wird beschrieben, welche Strategien eingesetzt werden um die Leistung des 3D Terraingenerator zu messen. Anschließend werden diesen Strategien Leistungstests unterzogen und dokumentiert.

3.1. Berechnung der Testergebnisse

Um die Leistung eines Spieles messen zu können, wird eine Messgröße benötigt. Die zentrale Messgröße der Geschwindigkeits-Optimierung ist die Bildgenerierrate (frame rate). Diese gibt an wie viele Bilder pro Sekunde (frames per second, FPS) generiert werden können [8][s. 472]. In Unity3D wird von der Klasse Time die Information DeltaTime [14] zur Verfügung gestellt, welche angibt wie lange die Erstellung des letzten Bildes gedauert hat. Daraus wird in der Update Funktion der Test Klasse die Bildgenerrierrate errechnet, wie im Listing 3.1.

```
fps = 1.0f / Time.deltaTime;
```

Listing 3.1: Berrechnung der Bilder pro Sekunde in der Test Klasse

3.2. Errechnung der Felder

Damit Tests auf beiden Terrains durchgeführt werden können, muss die Testsituationen gleich gestaltet werden. Da beide Terraingeneratoren auf verschiedenen Methoden und Dimensionen basieren, muss für beide Terrains die Möglichkeit existieren, die selbe Menge an Felder anzuzeigen. Dazu werden bei jedem Test beim

Ladeprozess der Terraindaten aus Abschnitt 2.1.3 die notwendige Daten eingelesen.

Da in Spielen meist nur ein Teil des Terrains betrachtet wird, müssen auch Tests für diesen Anwendungsfall durchgeführt werden. In manchen Spielen wird die Funktionalität des Zoomens auf einen maximalen Wert beschränkt oder das Zoomen deaktiviert. Um sicher zu stellen, dass sich beim Anzeigen von Teilen des Terrains, sowohl in 2D als auch in 3D gleich viele Felder am Display befinden, muss diese Anzahl berechnet werden. Dies wurde in beiden Terraingeneratoren in der Test Klasse mit der Funktion CalcActiveFields umgesetzt. Der Code für den 3D Generator ist im Listing 3.2 und der Code für den 2D Generator ist im Listing 3.3. Bei beiden Umsetzungen wird jedes Feld des Terrains überprüft, ob es im Sichtfeld der Kamera ist. Dazu wird die von der Unity Kamera zur Verfügung gestellten Funktion WorldToViewportPoint [13] verwendet. Diese liefert einen Vektor zwischen 0 und 1 zurück, sofern der übergebene Punkt im Sichtfeld der Kamera ist. Bei der Umsetzung für den 2D Generator muss zuvor noch die Zeile und Spalte in Weltkoordinaten umgerechnet werden. Dies geschieht in der Funktion GetWorldPosFromRowCol, welche die Weltkoordinaten X und Y wie in Punkt 2.2.2 errechnet.

```
public void CalcActiveFields()
{
    currentActiveFields = 0;
    for (int x = 0; x < creator.maxFieldCountH; x++)
    {
        for (int y = 0; y < creator.maxFieldCountV; y++)
        {
            Vector3 v = Camera.main.WorldToViewportPoint(new Vector3(x,
                creator.terrain[x, y], y));
            if (v.x <= 1 && v.y <= 1 && v.x >= 0 && v.y >= 0)
            {
                currentActiveFields++;
            }
        }
    }
}
```

Listing 3.2: Funktion zu Errechnung der Anzahl der angezeigten Felder des 3D Terrains

```
public void CalcActiveFields()
{
    currentActiveFields = 0;
    for (int x = 0; x < terrainGenerator.sizeX - 1; x++)
    {
        for (int y = 0; y < terrainGenerator.sizeX - 1; y++)
        {
```

Hersteller	Name	Prozessortaktung	RAM	Auflösung	Android
OnePlus	One	4 x 2,5 Ghz	4 GB	1080x1920	5.1.1
Samsung	Galaxy Note	2 x 1,4 Ghz	1 GB	800x1280	4.1.2
Samsung	Galaxy S2	2 x 1,2 Ghz	1 GB	480x800	4.4.4
Samsung	Galaxy S3 Mini	1 Ghz	1 GB	480x640	4.1.3

Tabelle 3.1.: Testgeräte

```
            Vector2 v1 = terrainGenerator.GetWorldPosFromRowCol(x, y);
            Vector3 v = Camera.main.WorldToViewportPoint(v1);
            if (v.x <= 1 && v.y <= 1 && v.x >= 0 && v.y >= 0)
            {
                currentActiveFields++;
            }
        }
    }
}
```

Listing 3.3: Funktion zu Errechnung der Anzahl der angezeigten Felder des 2D Terrains

Zur Messung von Zeitspannen wurde für alle Tests die Klasse Stopwatch [15] aus dem Mono Framework 2.0 verwendet.

3.3. Einstellungen in Unity

Für die Tests wurde Unity 5.0.1p1 Personal verwendet. Die Einstellungen in den Player Settings in Unity wurden nicht verändert. In den Quality Settings wurden Schatten deaktiviert, damit die Schattenberechnung keinen Einfluss auf die Tests hat. Alle weiteren Quality Settings wurden nicht verändert. Bei allen Tests wurde eine orthographische Kamera verwendet.

3.4. Tests

Zur Durchführung der Tests wurden die Android Smartphones aus der Tabelle 3.1 verwendet.

3.4.1. Chunks

In Punkt 2.3.4 wurden Chunks umgesetzt um ein beliebig großes Terrain generieren zu können. Die Auswirkungen der implementierten Chunks wird nun getestet. Dazu wird ein volles Terrain mit 100x100 Feldern auf den Testgeräten angezeigt und die durchschnittliche FPS gemessen. Um einen Vergleichswert zu einem Terrain ohne Chunks zu haben wird repräsentativ ein Terrain mit einem Chunk mit 100x100 Feldern verwendet. Die Größe 100x100 Felder wurde gewählt, weil ein Mesh in Unity maximal ein 65000 großes Vertizes-Array haben kann. Dies ergibt bei 6 Vertizes pro Feld ein Maximum von 10833 Felder, welches ein Maximum von 104x104 Felder pro Chunk ergibt.

Um die Leistungsauswirkung bei Verwendung von Chunks zu testen wurden Terrains getestet, welche die gleiche Anzahl von Felder und verschiedene Anzahl von Chunks haben. Folgende Verhältnisse wurden getestet:

- 20x20 Chunks mit 5x5 Feldern
- 10x10 Chunks mit 10x10 Felder
- 5x5 Chunks mit 20x20 Felder
- 1 Chunks mit 100x100 Felder

Im Ergebnis ist in Abbildung 3.1 zu erkennen, dass die Verwendung von mehreren Chunks zu weniger FPS führen.

Um einen Vergleichswert mit dem 2D Terraingenerator zu erhalten, wurde dieser mit 100x100 Felder getestet. Dieser Test zeigt, dass ein Terrain mit 100x100 Felder des 2D Terraingenerators mehr Leistung benötigt als das des 3D Terraingenerator.

3.4.2. Anzeigen von Bereichen

Da ein Terrain in Spielen nicht vollständig angezeigt werden muss, wurden die FPS des Terrains des 2D und 3D Generatores, beim Anzeigen von 10, 100, 1000, 5000, 10000 und 57600 Felder gemessen. 57600 Felder ist das Maximum der geladenen Daten des 240x240 großen Terrains aus dem Punkt 2.1.3. Für das 3D Terrain wurden 16x16 Chunks mit 15x15 Feldern verwendet. Für das 2D Terrain wurden 240x240 Felder verwendet.

Das Ergebnis in Abbildung 3.2 zeigt, dass beim Test des 3D Terrains alle 4 Testgeräte beim Anzeigen von 10 bis 10000 Feldern ausreichend Leistung besitzen. Das 2D Terrain weist dagegen bei 1000 Felder bereits Leistungsprobleme auf.

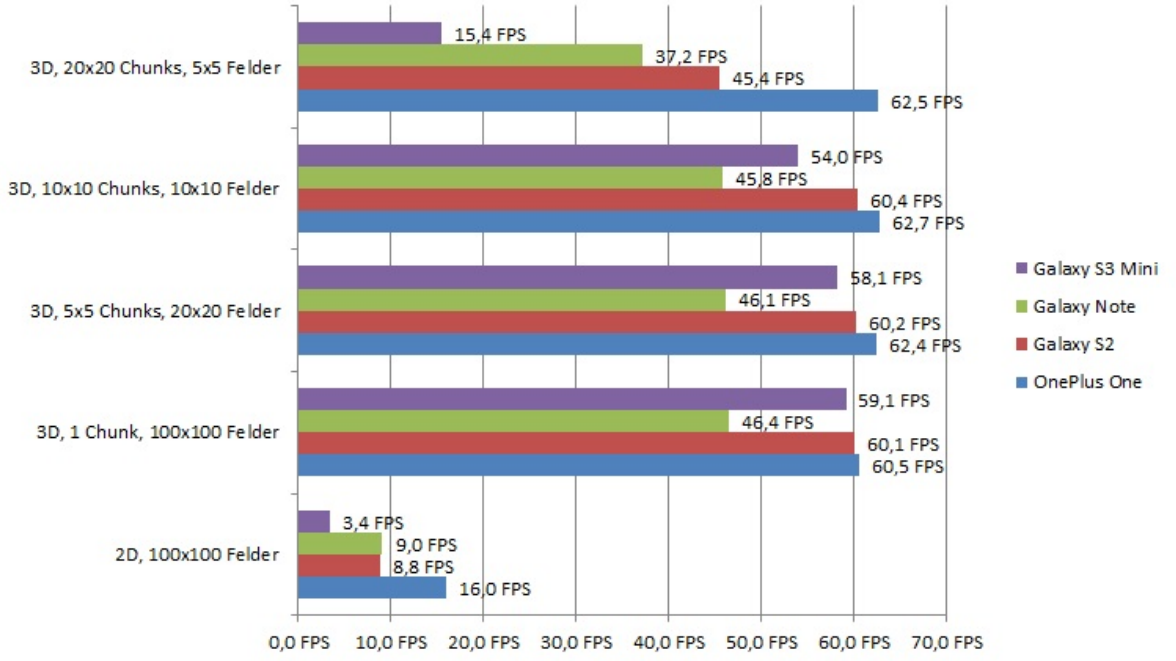

Abbildung 3.1.: Testergebnis für 100000 Felder. Werte in Bilder pro Sekunde

3.4.3. Erhöhungsdauer

Um die Leistung bei der Veränderung des Terrains zu messen, wird der Zeitunterschied zwischen Beginn und Ende einer Veränderung am Terrains gemessen. Diese Zeit wurde beim rekursiven Anheben von 1000 und 10000 Feldern am 2D und 3D Terrain gemessen.

Das Ergebnis in Abbildung 3.3 zeigt, dass das Verändern des 3D Terrains im Durchschnitt doppelt so lange dauert wie beim 2D Terrain. Beide Terrains können 1000 Felder in unter 52ms anheben, welches für User keine spürbare Verzögerung gibt. Für den User passieren Vorgänge sofort, sofern sie unter 0,1 Sekunde passieren [16][S. 181-188]. Bei 10000 Feldern ist dieser Wert bei beiden Terrains überschritten.

In Punkt 2.3.5 sind zwei Umsetzungen beschrieben, wie das Terrain geklickt werden kann. In diesem Test wird gemessen welche Leistung das Verwenden des implementierten Raycasting gegenüber der Standardmethode durch die Unity Engine aufweist. Dazu wurden die Collider getestet, welche bei Verwendung der Standardmethode notwendig sind. Collider müssen nach jeder Änderung des Terrains neu erstellt werden. Deshalb wurde die Zeit gemessen, die benötigt wird, um im Terrain 1000 und 10000 Felder mit und ohne Collider anzuheben. Das Ergebnis in Abbildung 3.4

zeigt, dass die Erhöhung mit Collider im Durchschnitt doppelt so lange dauert, wie die Erhöhung ohne Collider. Daraus schließend ist die Verwendung des Klickens ohne Collider die leistungsfähigere Methode.

3.4.4. Generierungsgeschwindigkeit

In diesem Test wird gemessen wieviel Zeit die Generierung eines Terrains mit 100x100 und 240x240 Feldern in Anspruch nimmt. Dabei wurde die benötige Zeit für das Laden der Höheninformation aus Punkt 2.1.3 ausgeschlossen. Diese Tests wurden auf dem 2D und 3D Terraingenerator durchgeführt. Das Ergebnis in Abbildung 3.5 zeigt, dass der 2D Generator für die Erstellung eines Terrains im Durchschnitt doppelt so viel Zeit benötigt wie der 3D Generator.

3.4.5. Conclusion

In den Tests wurde gezeigt dass in Unity ein 3D isometrisches Terrain erstellt werden kann, welches die selbe oder bessere Leistung bringt wie ein einfaches 2D isometrisches Terrain. Dies wurde demonstriert, indem bestimmte Anzahlen an Felder angezeigt wurden, wobei das 3D Terrain in allen Tests mehr Bilder pro Sekunde generieren konnte.

Der Versuch das Terrain mittels Frustum Culling durch Chunks zu optimieren, brachte keine Verbesserung. Dessen Einsatz zeigte, dass bei Verwendung der minimalen Anzahl an Chunks die FPS gleich blieben. Bei Verwendung von vielen Chunks verringerten sich die FPS. Es behob das Problem, welches die Größe des Terrains limitierte.

Um die Leistung bei der Veränderung des Terrains zu verbessern, wurde auf Collider verzichtet und ein eigenes Raycasting implementiert. Dies resultierte in einer schnelleren Veränderung des Terrains. Trotz der dieser Verbesserung ist das 2D Terrain schneller beim Verändern als das 3D Terrain.

Der Test der Generierungsgeschwindigkeit zeigte, dass der 3D Terraingenerator schneller Terrain erstellt, als der 2D Terraingenerator. Dieser Unterschied äußert sich bei großen Terrains deutlicher.

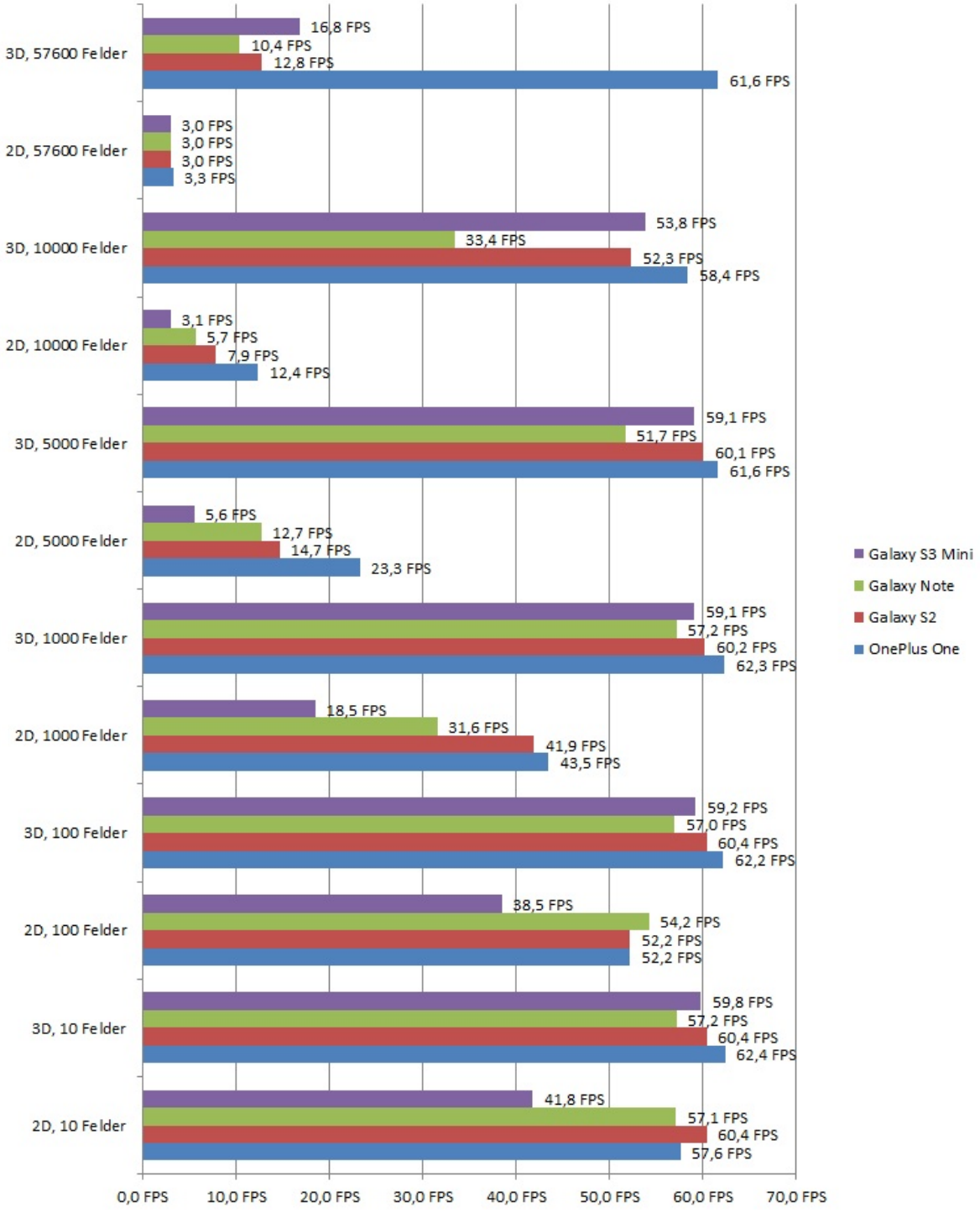

Abbildung 3.2.: Testergebnis für das Anzeigen von Bereichen. Werte in Bilder pro Sekunde

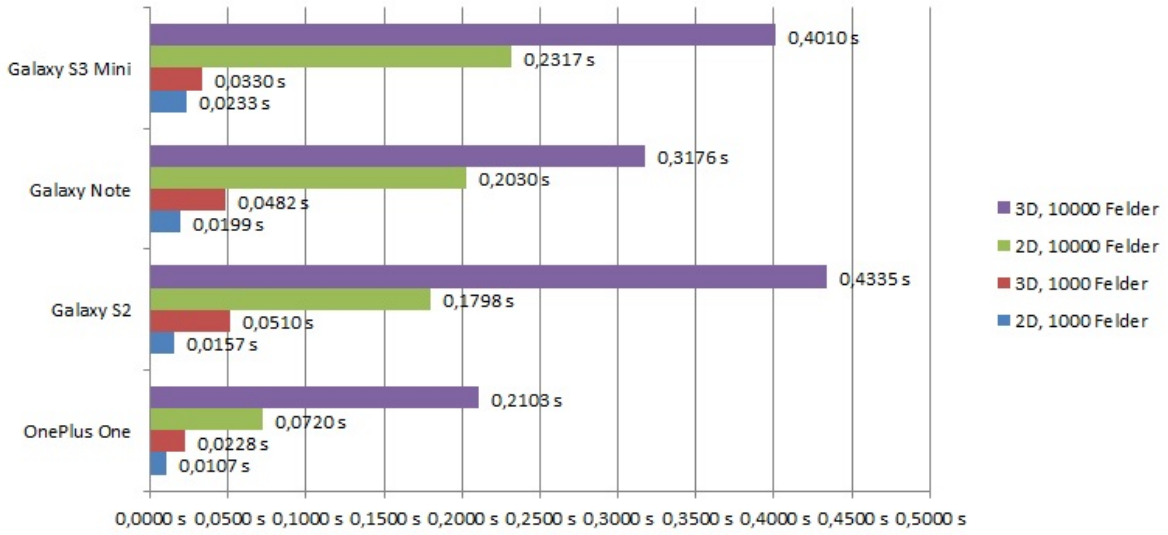

Abbildung 3.3.: Testergebnis für die Erhöhungsdauer. Werte in Sekunden

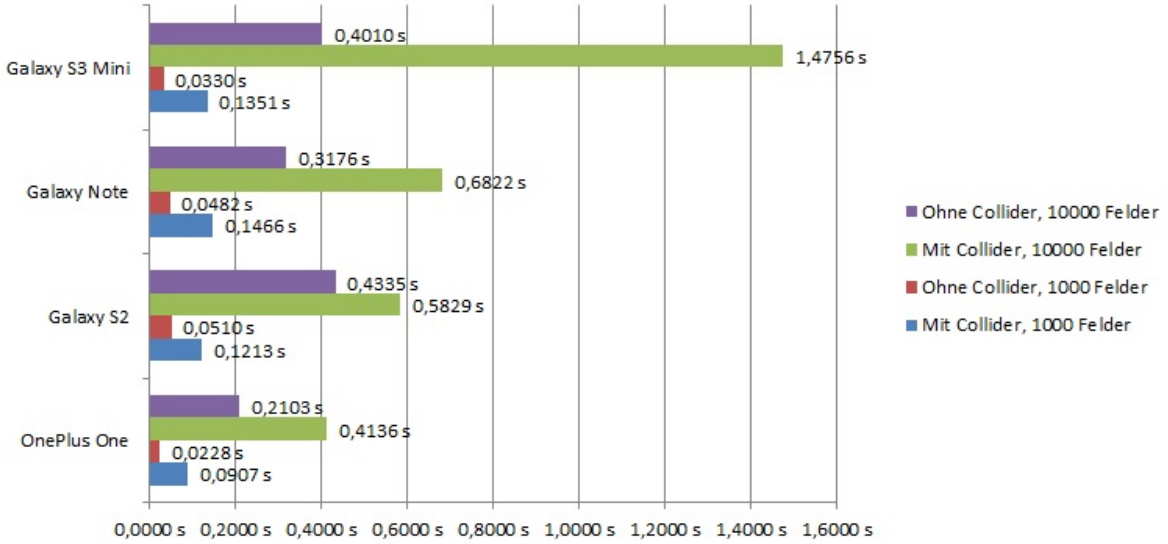

Abbildung 3.4.: Testergebnis für die Erhöhungsdauer mit Collidern am 3D Terrain. Werte in Sekunden

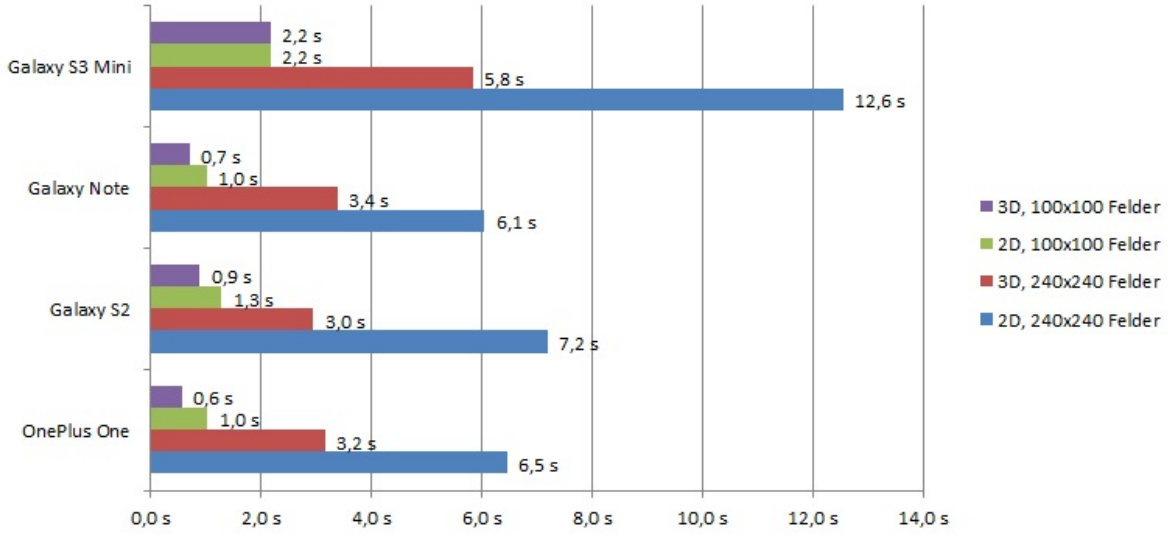

Abbildung 3.5.: Testergebnis für die Generierungsgeschwindigkeit. Werte in Sekunden

Kapitel 4

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein 2D isometrischer Terraingenerator und ein 3D Terraingenerator in Unity erstellt. Der 3D Terraingenerator wurde optimiert, indem auf Collider verzichtet wurde und ein eigenes Raycasting für das Klicken implementiert wurde. Dies brachte dem Terrain eine Leistungsverbesserung beim rekursiven Erhöhen von Feldern.

Durch den Einsatz von Chunks wurde das Problem gelöst, bei dem die Grenze der Terraingröße erreicht wurde. Die Tests der Chunks ergaben, dass die maximale Anzahl von Feldern für Chunks die beste Leistung in Unity erzielen.

Um eine Antwort auf die Forschungsfrage, welche Leistung die Terrains benötigen, zu bekommen, wurde die Leistung beider Terrains beim Anzeigen verschieden großer Daten getestet. Dabei stellte sich heraus, dass bei Verwendung der selben Hardware das 3D Terrain zehn mal mehr Felder anzeigen kann als das 2D Terrain, um die selbe Anzahl an Bildern pro Sekunde zu erhalten.

Beim Test der Erhöhungsgeschwindigkeit wurde deutlich, dass in diesem Gebiet das 2D Terrain dem 3D Terrain überlegen ist. Der Grund dafür ist, dass diese nur Texturen der Felder verändern müssen, wo hingegen beim 3D Terrain das gesamte Vertizes-Array aktualisiert werden muss.

Die Erkenntnis die aus den Tests geschlossen werden kann, ist das Aufbaustrategiespiele für Smartphones nicht nur in 2D umgesetzt werden müssen. Die 3D und Hybrid Lösungen können eine Konkurrenz stellen, welche eigene Vor- und Nachteile bieten.

Der Schwerpunkt dieser Arbeit war die Erstellung des 3D Terraingenerators. Deshalb wurde lediglich eine einfache Version eines 2D Terraingenerators umgesetzt. In weiterer Folge ist zu untersuchen wie der 2D Terraingenerator in Unity optimiert werden kann und welcher Leistungsgewinn dadurch erzielt werden kann.

Anhang A

Terraindaten mit 240x240 Feldern

Anhang B

Umgesetzter Code des 2D Terraingenerators

```
using UnityEngine;
using System.Collections;
using System.IO;

public class TerrainGenerator : MonoBehaviour {

    public int sizeX = 240;
    public int sizeY = 240;

    int BOARD_X = 0;
    int BOARD_Y= 0;
    float TEXTURE_SIZE = 0.64f;

    float[,] terrain;
    GameObject[,] gameObjects;
    Vector2 OldMouse = new Vector2(-999999,-999999);
    float OldDist = -99999;

    Test test;
    float stepSize = 0.08f;

    Sprite[] sprites;

    bool loading = true;

        // Use this for initialization
        void Start ()
    {
        test = (Test)GameObject.Find("Test").GetComponent(typeof(Test));
        terrain = new float[sizeX, sizeY];
        gameObjects = new GameObject[sizeX, sizeY];
```

```
    sprites = Resources.LoadAll<Sprite>("IsoMap_Tiles");

    LoadTerrainFile();

    LoadTiles();

    }
private void LoadTiles()
{
    test.StartLoadingTimer();
    for (int row = 0; row < sizeX - 1; row++)
    {
        for (int col = 0; col < sizeY - 1; col++)
        {
            GameObject go = new GameObject(row.ToString() + "," + col.ToString());
            go.transform.parent = this.transform;

            SpriteRenderer sprite = go.AddComponent<SpriteRenderer>();

            gameObjects[col, row] = go;

            UpdateField(col, row);
        }
    }
    test.StopLoadingTimer();
}
private void LoadTerrainFile()
{
    TextAsset txt = Resources.Load("Terrain") as TextAsset;
    string[] linesFromfile = txt.text.Split("\n"[0]);

    for (int y = 0; y < sizeY; y++ )
    {
        string[] s = linesFromfile[y].Split(';');
        for (int x = 0; x < sizeX; x++)
        {
            terrain[x, y] = float.Parse(s[x]);
        }
    }
}
public Vector2 GetWorldPosFromRowCol(int row, int col)
{
    float x, y;
    x = ((float)(BOARD_X - (row * TEXTURE_SIZE / 2) + (col * TEXTURE_SIZE / 2)));
    y = (float)(BOARD_Y + (row * TEXTURE_SIZE / 4) + (col * TEXTURE_SIZE / 4));

    float leftBottom = terrain[col, row];
    float leftTop = terrain[col, row + 1];
    float rightBottom = terrain[col + 1, row];
```

```
float rightTop = terrain[col + 1, row + 1];
float additionallRaise = 0;

if (rightTop < leftTop && rightTop < leftBottom && rightTop < rightBottom)
{
    if (leftBottom > leftTop && leftBottom > rightBottom)
    {
        additionallRaise--;
        additionallRaise--;
    }
    else
    {
        additionallRaise--;
    }
}
else if (rightTop < leftTop && rightTop < leftBottom && rightBottom <
    leftTop && rightBottom < leftBottom)
{
    additionallRaise--;
}
else if (rightTop < leftBottom && rightTop < rightBottom && leftTop <
    leftBottom && leftTop < rightBottom)
{
    additionallRaise--;
}
else if (leftTop < rightTop && leftTop < leftBottom && leftTop < rightBottom)
{
    if (rightBottom > leftBottom)
    {
        additionallRaise--;
    }
    else
    {
        additionallRaise--;
    }
}
else if (rightBottom < rightTop && rightBottom < leftBottom && rightBottom <
    leftTop)
{
    if (leftTop > leftBottom)
    {
        additionallRaise--;
    }
    else
    {
        additionallRaise--;
    }
}
```

```
    else if (leftBottom > leftTop && leftBottom > rightBottom && leftBottom >
        rightTop)
    {
        additionallRaise--;
    }

    else if (rightTop > leftTop && leftBottom > rightBottom)
    {
        additionallRaise--;
    }
    float yOffSet = terrain[col, row];
    return new Vector2(x, y + (additionallRaise + yOffSet) * 0.08f);
}
// Update is called once per frame
void Update()
{
    if (Input.GetMouseButtonUp(0) || Input.GetMouseButtonUp(1))
    {
        test.StartClickTimer();
        Vector3 mousePos = Camera.main.ScreenToWorldPoint(new
            Vector3(Input.mousePosition.x, Input.mousePosition.y,
            Camera.main.transform.localPosition.z));

        int col = (int)Mathf.Round(mousePos.y / (TEXTURE_SIZE / 2) + mousePos.x
            / TEXTURE_SIZE);
        int row = (int)Mathf.Round(mousePos.y / (TEXTURE_SIZE / 2) - mousePos.x
            / TEXTURE_SIZE);
        test.StopClickTimer();
        test.StartRaiseTimer();
        int count = 0;
        if (Input.GetMouseButtonUp(0))
        {
            count = ChangeHeight(col, row, true, 0);
        }
        else
        {
            count = ChangeHeight(col, row, false, 0);
        }
        test.StopRaiseTimer(count);
        //gameObjects[col,row].GetComponent<SpriteRenderer>().color = new
            Color(255, 0, 0);
    }


    if (Input.touchCount > 1)
    {
        float dist = Vector2.Distance(Input.touches[0].position,
            Input.touches[1].position);
        if (OldDist == -99999)
        {
```

```
            OldDist = dist;
        }
        else
        {
            Camera.main.orthographicSize += (dist - OldDist)/1000;
        }
    }
    else
    {
        OldDist = -99999;
    }
    }
private int ChangeHeight(int col, int row, bool increase, int count)
{
    count++;

    if (increase)
    {
        if (terrain[col - 1, row] < terrain[col, row])
        {
            count = ChangeHeight(col - 1, row, increase, count);
        }
        if (terrain[col + 1, row] < terrain[col, row])
        {
            count = ChangeHeight(col + 1, row, increase, count);
        }
        if (terrain[col, row - 1] < terrain[col, row])
        {
            count = ChangeHeight(col, row - 1, increase, count);
        }
        if (terrain[col, row + 1] < terrain[col, row])
        {
            count = ChangeHeight(col, row + 1, increase, count);
        }
        terrain[col, row] = terrain[col, row] + 1;
    }
    else
    {
        if (terrain[col - 1, row] > terrain[col, row])
        {
            count = ChangeHeight(col - 1, row, increase, count);
        }
        if (terrain[col + 1, row] > terrain[col, row])
        {
            count = ChangeHeight(col + 1, row, increase, count);
        }
        if (terrain[col, row - 1] > terrain[col, row])
        {
            count = ChangeHeight(col, row - 1, increase, count);
        }
```

```
        if (terrain[col, row + 1] > terrain[col, row])
        {
            count = ChangeHeight(col, row + 1, increase, count);
        }
        terrain[col, row] = terrain[col, row] - 1;
    }

    UpdateField(col, row);
    UpdateField(col-1, row);
    UpdateField(col, row-1);
    UpdateField(col-1, row-1);

    return count;
}
private void UpdateField(int col, int row)
{
    float leftBottom = terrain[col, row];
    float leftTop = terrain[col, row + 1];
    float rightBottom = terrain[col + 1, row];
    float rightTop = terrain[col + 1, row + 1];
    float additionallRaise = 0;

    int TexId = 0;

    if (leftBottom < rightTop && leftBottom < rightBottom && leftTop < rightTop
        && leftTop < rightBottom)
    {
        TexId = 3;
    }
    else if (leftBottom < rightTop && leftBottom < leftTop && rightBottom <
        rightTop && rightBottom < leftTop)
    {
        TexId = 9;
    }
    else if (leftBottom < leftTop && leftBottom < rightTop && leftBottom <
        rightBottom)
    {
        if (rightTop > leftTop)
        {
            TexId = 16;
        }
        else
        {
            TexId = 11;
        }
    }
    else if (rightTop < leftTop && rightTop < leftBottom && rightTop <
        rightBottom)
    {
        if (leftBottom > leftTop && leftBottom > rightBottom)
```

```
    {
        TexId = 18;
        additionallRaise--;
        additionallRaise--;
    }
    else
    {
        TexId = 14;
        additionallRaise--;
    }
}
else if (rightTop < leftTop && rightTop < leftBottom && rightBottom <
    leftTop && rightBottom < leftBottom)
{
    TexId = 12;
    additionallRaise--;
}
else if (rightTop < leftBottom && rightTop < rightBottom && leftTop <
    leftBottom && leftTop < rightBottom)
{
    TexId = 6;
    additionallRaise--;
}
else if (leftTop < rightTop && leftTop < leftBottom && leftTop < rightBottom)
{
    if (rightBottom > leftBottom)
    {
        TexId = 19;
        additionallRaise--;
    }
    else
    {
        TexId = 7;
        additionallRaise--;
    }
}
else if (rightBottom < rightTop && rightBottom < leftBottom && rightBottom <
    leftTop)
{
    if (leftTop > leftBottom)
    {
        TexId = 17;
        additionallRaise--;
    }
    else
    {
        TexId = 13;
        additionallRaise--;
    }
}
```

```
        else if (rightTop > leftBottom && rightTop > rightBottom && rightTop >
            leftTop)
        {
            TexId = 1;
        }
        else if (leftBottom > leftTop && leftBottom > rightBottom && leftBottom >
            rightTop)
        {
            TexId = 4;
            additionallRaise--;
        }
        else if (rightBottom > leftBottom && rightBottom > leftTop && rightBottom >
            rightTop)
        {
            TexId = 2;
        }
        else if (leftTop > leftBottom && leftTop > rightBottom && leftTop > rightTop)
        {
            TexId = 8;
        }
        else if (rightTop < leftTop && leftBottom < rightBottom)
        {
            TexId = 10;
        }
        else if (rightTop > leftTop && leftBottom > rightBottom)
        {
            TexId = 5;
            additionallRaise--;
        }
        SpriteRenderer sprite = gameObjects[col, row].GetComponent<SpriteRenderer>();
        sprite.sprite = sprites[TexId];

        float x = ((float)(BOARD_X - (row * TEXTURE_SIZE / 2) + (col * TEXTURE_SIZE
            / 2)));
        float y = (float)(BOARD_Y + (row * TEXTURE_SIZE / 4) + (col * TEXTURE_SIZE /
            4));

        float yOffSet = terrain[col, row];
        gameObjects[col, row].transform.localPosition = new Vector3(x, y +
            (additionallRaise + yOffSet) * (TEXTURE_SIZE/8));
    }
}
```

Listing B.1: TerrainGenerator Klasse des 2D Terraingenerators

```
using UnityEngine;
using System.Collections;
using System.Diagnostics;
```

```
public class Test : MonoBehaviour {

    TerrainGenerator terrainGenerator;
    public int currentActiveFields = 0;

    float deltaTime = 0.0f;

    float countSeconds = 0;

    float lowestFPS = 99999;
    float highestFPS;

    float averageFPSSum = 0;
    int averageFPSCount = 0;

    float lastAverageFPS = 0;
    float lastHighFPS = 0;
    float lastLowFPS = 0;

    float tempCameraSize = 0;

    Stopwatch loadingWatch = new Stopwatch();
    Stopwatch clickWatch = new Stopwatch();
    Stopwatch raiseWatch = new Stopwatch();

    int raiseCount = 0;

        void Start () {
        terrainGenerator =
            (TerrainGenerator)GameObject.Find("TerrainGenerator").GetComponent(typeof(TerrainGenerat
        }

        void Update ()
    {
        deltaTime += (Time.deltaTime - deltaTime) * 0.1f;
        float msec = deltaTime * 1000.0f;
        float fps = 1.0f / deltaTime;
        countSeconds += Time.deltaTime;
        if (fps < lowestFPS)
        {
            lowestFPS = fps;
        }
        if (fps > highestFPS)
        {
            highestFPS = fps;
        }
        averageFPSSum += fps;
        averageFPSCount++;

        if (countSeconds > 10)
```

```
    {
        lastAverageFPS = averageFPSSum / (float)averageFPSCount;
        lastHighFPS = highestFPS;
        lastLowFPS = lowestFPS;
        averageFPSCount = 0;
        averageFPSSum = 0;
        lowestFPS = 99999;
        highestFPS = 0;
        countSeconds = 0;

    }
    if (Camera.main.orthographicSize != tempCameraSize)
    {
        tempCameraSize = Camera.main.orthographicSize;
        CalcActiveFields();
    }
    }
public void CalcActiveFields()
{
    currentActiveFields = 0;
    for (int x = 0; x < terrainGenerator.sizeX-1; x++)
    {
        for (int y = 0; y < terrainGenerator.sizeX - 1; y++)
        {
            Vector2 v1 = terrainGenerator.GetWorldPosFromRowCol(x, y);
            Vector3 v = Camera.main.WorldToViewportPoint(v1);
            if (v.x <= 1 && v.y <= 1 && v.x >= 0 && v.y >= 0)
            {
                currentActiveFields++;
            }
        }
    }
}
void OnGUI()
{
    int w = Screen.width;
    int h = Screen.height;

    GUIStyle style = new GUIStyle();
    style.alignment = TextAnchor.UpperLeft;
    style.fontSize = h * 2 / 100;
    style.normal.textColor = new Color(1.0f, 0.0f, 0.0f, 1.0f);

    float textheight = h * 2 / 100;

    Rect rectTitle = new Rect(0, 0, w, textheight);
    Rect rectFps = new Rect(0, textheight, w, textheight);
    Rect rectActiveFields = new Rect(0, textheight * 2, w, textheight);
    Rect rectLoadTimer = new Rect(0, textheight * 3, w, textheight);
    Rect rectClickTimer = new Rect(0, textheight * 4, w, textheight);
```

```
    Rect rectRaiseTimer = new Rect(0, textheight * 5, w, textheight);
    Rect rectRaiseCounter = new Rect(0, textheight * 6, w, textheight);

    float msec = deltaTime * 1000.0f;
    float fps = 1.0f / deltaTime;

    string textTitle = "2D_" + terrainGenerator.sizeX + "f_" +
        terrainGenerator.sizeX + "f";
    GUI.Label(rectTitle, textTitle, style);
    string text = string.Format("Low:{0:0.0} Av:{1:0.0} High:{2:0.0}
        curr:{3:0.0}", lastLowFPS, lastAverageFPS, lastHighFPS, fps);
    GUI.Label(rectFps, text, style);
    string text2 = string.Format("Fields:{0:0}", currentActiveFields);
    GUI.Label(rectActiveFields, text2, style);
    string text3 = string.Format("Loading Time: " + loadingWatch.Elapsed);
    GUI.Label(rectLoadTimer, text3, style);
    string text4 = string.Format("Click Time: " + clickWatch.Elapsed);
    GUI.Label(rectClickTimer, text4, style);
    string text5 = string.Format("Raise Time: " + raiseWatch.Elapsed);
    GUI.Label(rectRaiseTimer, text5, style);
    string text6 = string.Format("Raise Count:{0}", raiseCount);
    GUI.Label(rectRaiseCounter, text6, style);
}
public void StartLoadingTimer()
{
    loadingWatch = new Stopwatch();
    loadingWatch.Start();
}
public void StopLoadingTimer()
{
    loadingWatch.Stop();
}
public void StartClickTimer()
{
    clickWatch = new Stopwatch();
    clickWatch.Start();
}
public void StopClickTimer()
{
    clickWatch.Stop();
}
public void StartRaiseTimer()
{
    raiseWatch = new Stopwatch();
    raiseWatch.Start();
}
public void StopRaiseTimer(int raiseCount)
{
    raiseWatch.Stop();
    this.raiseCount = raiseCount;
```

```
    }
}
```

Listing B.2: Test Klasse des 2D Terraingenerators

Anhang C

Umgesetzter Code des 3D Terraingenerators

```
using UnityEngine;
using System.Collections;
using System;
using System.Collections.Generic;
using System.IO;

public class TerrainCreator : MonoBehaviour
{
    public int chunkCountH = 16;
    public int chunkCountV = 16;

    public int vertCountH = 16;
    public int vertCountV = 16;

    public int fieldCountH = 15;
    public int fieldCountV = 15;

    public float scaleX = 1.0f;
    public float scaleY = 1.0f;
    public float stepSize = 0.25f;

    public float lowestPossibleHeight = -10;
    public float highestPossibleHeight = 30;

    public Chunk[,] chunks;
    public float[,] terrain;

    List<Chunk> chunksThatNeedsUpdates = new List<Chunk>();

    public int maxFieldCountH = 0;
    public int maxFieldCountV = 0;
```

```
Tests test;
public int ChunksLoaded = 0;

bool loading = true;


public static Vector3 Vector3Error = new Vector3(-99999, -99999, -99999);

private void Start()
{
    test = (Tests)GameObject.Find("Tests").GetComponent(typeof(Tests));

    vertCountH = fieldCountH + 1;
    vertCountV = fieldCountV + 1;

    maxFieldCountH = chunkCountH * fieldCountH;
    maxFieldCountV = chunkCountV * fieldCountV;

    terrain = new float[maxFieldCountH, maxFieldCountV];
    chunks = new Chunk[chunkCountH, chunkCountV];

    LoadTerrainFile();



}
private void LoadTerrainFile()
{
    TextAsset txt = Resources.Load("Terrain") as TextAsset;
    string[] linesFromfile = txt.text.Split("\n"[0]);

    for (int y = 0; y < maxFieldCountV; y++)
    {
        string[] s = linesFromfile[y].Split(';');
        for (int x = 0; x < maxFieldCountH; x++)
        {
            terrain[x, y] = float.Parse(s[x]);
        }
    }
}
public void CreateTerrain()
{
    for (int y = 0; y < chunkCountV; y++)
    {
        for (int x = 0; x < chunkCountH; x++)
        {
            GameObject g = new GameObject("Chunk:" + x.ToString() + "," +
                y.ToString());
            //g.AddComponent(typeof(MeshCollider));
```

```
            g.transform.parent = this.transform;
            g.transform.position = new Vector3(x * fieldCountH * scaleX, 0, y *
                fieldCountV * scaleY);
            Chunk c = (Chunk)g.AddComponent(typeof(Chunk));
            chunks[x, y] = c;
        }
    }
}
void Update()
{
    if (loading)
    {
        test.StartLoadingTimer();
        CreateTerrain();
        loading = false;
    }
    if (ChunksLoaded >= chunkCountH * chunkCountV)
    {
        test.StopLoadingTimer();
    }
    else
    {
        test.loadTimeElapsed += Time.deltaTime;
    }
    if (Input.GetMouseButtonUp(0))
    {
        test.StartClickTimer();
        Vector3 pos = GetClickPosition();
        test.StopClickTimer();

        int count = 0;
        if (pos != Vector3Error)
        {
            test.StartRaiseTimer();
            count = ChangeHeight((int)pos.x, (int)pos.z, true);
            test.StopRaiseTimer(count);
        }
    }
    if (Input.GetMouseButtonUp(1))
    {
        test.StartClickTimer();
        Vector3 pos = GetClickPosition();
        test.StopClickTimer();

        int count = 0;
        if (pos != Vector3Error)
        {
            test.StartRaiseTimer();
            count = ChangeHeight((int)pos.x, (int)pos.z, false);
            test.StopRaiseTimer(count);
```

```
        }
    }
}
public int[] GetChunkPos(Chunk c)
{
    for (int y = 0; y < chunkCountV; y++)
    {
        for (int x = 0; x < chunkCountH; x++)
        {
            if (chunks[x, y] == c)
            {
                return new int[2] { x, y };
            }
        }
    }
    return null;
}
public int ChangeHeight(int x, int y, bool increase)
{
    int count = ChangeHeightRecursive(x, y, increase, 0);
    foreach (Chunk c in chunksThatNeedsUpdates)
    {
        c.CreateGrid();
    }
    chunksThatNeedsUpdates.Clear();
    return count;
}
public int ChangeHeightRecursive(int x, int y, bool increase, int count)
{
    if (increase)
    {
        int xAround = x + 1;
        int yAround = y;
        if (xAround < maxFieldCountH)
        {
            if (terrain[x, y] > terrain[xAround, yAround])
            {
                count = ChangeHeightRecursive(xAround, yAround, increase, count);
            }
        }

        xAround = x - 1;
        yAround = y;
        if (xAround >= 0)
        {
            if (terrain[x, y] > terrain[xAround, yAround])
            {
                count = ChangeHeightRecursive(xAround, yAround, increase, count);
            }
        }
```

```
    xAround = x;
    yAround = y - 1;
    if (yAround >= 0)
    {
        if (terrain[x, y] > terrain[xAround, yAround])
        {
            count = ChangeHeightRecursive(xAround, yAround, increase, count);
        }
    }

    xAround = x;
    yAround = y + 1;
    if (yAround < maxFieldCountV)
    {
        if (terrain[x, y] > terrain[xAround, yAround])
        {
            count = ChangeHeightRecursive(xAround, yAround, increase, count);
        }
    }
    terrain[x, y] += 1;
}
else
{
    int xAround = x + 1;
    int yAround = y;
    if (terrain[x, y] < terrain[xAround, yAround])
    {
        count = ChangeHeightRecursive(xAround, yAround, increase, count);
    }

    xAround = x - 1;
    yAround = y;
    if (terrain[x, y] < terrain[xAround, yAround])
    {
        count = ChangeHeightRecursive(xAround, yAround, increase, count);
    }

    xAround = x;
    yAround = y - 1;
    if (terrain[x, y] < terrain[xAround, yAround])
    {
        count = ChangeHeightRecursive(xAround, yAround, increase, count);
    }

    xAround = x;
    yAround = y + 1;
    if (terrain[x, y] < terrain[xAround, yAround])
    {
        count = ChangeHeightRecursive(xAround, yAround, increase, count);
```

```
        }

        terrain[x, y] -= 1;
    }

    int cx = (int)(x / fieldCountH);
    int cy = (int)(y / fieldCountV);
    if (!chunksThatNeedsUpdates.Contains(chunks[cx, cy]))
    {
        chunksThatNeedsUpdates.Add(chunks[cx, cy]);
    }

    cx = (int)((x - 1) / fieldCountH);
    cy = (int)(y / fieldCountV);
    if (cx >= 0)
    {
        if (!chunksThatNeedsUpdates.Contains(chunks[cx, cy]))
        {
            chunksThatNeedsUpdates.Add(chunks[cx, cy]);
        }
    }
    cx = (int)(x / fieldCountH);
    cy = (int)((y - 1) / fieldCountV);
    if (cy >= 0)
    {
        if (!chunksThatNeedsUpdates.Contains(chunks[cx, cy]))
        {
            chunksThatNeedsUpdates.Add(chunks[cx, cy]);
        }
    }
    cx = (int)((x - 1) / fieldCountH);
    cy = (int)((y - 1) / fieldCountV);
    if (cx >= 0 && cy >= 0)
    {
        if (!chunksThatNeedsUpdates.Contains(chunks[cx, cy]))
        {
            chunksThatNeedsUpdates.Add(chunks[cx, cy]);
        }
    }
    return count + 1;
}

public Chunk GetChunkAtPos(int x, int y)
{
    if (x < chunkCountH && x >= 0 && y < chunkCountV && y >= 0)
    {
        return chunks[x, y];
    }
    else
    {
```

```
            return null;
        }
    }
    public Vector3 GetClickPosition()
    {
        Vector2 clickVector = new Vector2(Input.mousePosition.x,
            Input.mousePosition.y);
        Ray ray1 = Camera.main.ScreenPointToRay(clickVector);

        Vector3 jump = ray1.origin;

        float jumpSize = 0.1f;

        while (jump.y > lowestPossibleHeight && jump.y < 1000)
        {
            jump = jump + ray1.direction * jumpSize;

            if ((int)jump.x < maxFieldCountH && (int)jump.z < maxFieldCountV &&
                (int)jump.x > 0 && (int)jump.z > 0)
            {
                float currentHeight = terrain[(int)jump.x, (int)jump.z] * stepSize;
                if (jump.y < currentHeight)
                {
                    return new Vector3((float)Math.Round(jump.x), jump.y,
                        (float)Math.Round(jump.z));
                }
            }
        }
        return Vector3Error;
    }
}
```

Listing C.1: TerrainCreator Klasse des 3D Terraingenerators

```
using UnityEngine;
using System.Collections;
using System;
using System.Collections.Generic;

public class Chunk : MonoBehaviour
{
    public int fieldCountH = 50;
    public int fieldCountV = 50;

    int vertCountH = 0;
    int vertCountV = 0;

    private float scaleFieldH = 1.0f;
    private float scaleFieldV = 1.0f;
```

```
private float stepSize = 0.25f;

private TerrainCreator creator;

public int[] chunkPos;

public Vector3[] vertices;
public Vector2[] uvs;
public int[] triangles;

Mesh m;
MeshFilter meshFilter;

void Start()
{
    creator =
        (TerrainCreator)GameObject.Find("TerrainCreator").GetComponent(typeof(TerrainCreator));

    this.fieldCountV = creator.fieldCountH;
    this.fieldCountH = creator.fieldCountV;
    this.stepSize = creator.stepSize;
    this.scaleFieldH = creator.scaleX;
    this.scaleFieldV = creator.scaleY;

    chunkPos = creator.GetChunkPos(this);
    GameObject plane = GameObject.Find("Chunk:" + chunkPos[0].ToString() + "," +
        chunkPos[1].ToString());
    plane.layer = 1;
    plane.isStatic = true;

    MeshRenderer meshRenderer =
        (MeshRenderer)plane.AddComponent(typeof(MeshRenderer));
    meshRenderer.sharedMaterial =
        (Material)Resources.Load("Materials/Materials/TerrainMaterial");
    meshRenderer.castShadows = false;
    meshFilter = (MeshFilter)plane.AddComponent(typeof(MeshFilter));

    m = new Mesh();
    m.name = name + "Mesh";
    m.Clear();

    vertCountH = fieldCountH + 1;
    vertCountV = fieldCountV + 1;

    int numTriangles = fieldCountH * fieldCountV * 6;
    int numVertices = fieldCountH * fieldCountV * 6;
    vertices = new Vector3[numVertices];
    uvs = new Vector2[numVertices];
    triangles = new int[numTriangles];
```

```
    CreateGrid();
}

public void CreateGrid()
{
    for (int y = 0; y < fieldCountV; y++)
    {
        for (int x = 0; x < fieldCountH; x++)
        {
            CreateVertices(x, y);
            CreateTriangleField(x, y);
        }
    }
    UpdateMesh();
    creator.ChunksLoaded++;
}
public void CreateVertices(int x, int y)
{
    int index = (x + (y * (fieldCountH))) * 6;
    float heightLeftBot = 0;
    float heightLeftTop = 0;
    float heightRightBot = 0;
    float heightRightTop = 0;

    int arrayPosX = chunkPos[0] * fieldCountH;
    int arrayPosY = chunkPos[1] * fieldCountV;

    int posXMax = creator.maxFieldCountH;
    int posYMax = creator.maxFieldCountV;

    if (arrayPosX + x < posXMax - 1 && arrayPosY + y < posYMax - 1)
    {
        heightLeftBot = creator.terrain[arrayPosX + x, arrayPosY + y] * stepSize;
        if (arrayPosY + y + 1 < posYMax)
        {
            heightLeftTop = creator.terrain[arrayPosX + x, arrayPosY + y + 1] *
                stepSize;
        }
        if (arrayPosX + x + 1 < posXMax)
        {
            heightRightBot = creator.terrain[arrayPosX + x + 1, arrayPosY + y] *
                stepSize;
        }
        if (arrayPosY + y + 1 < posYMax && arrayPosX + x + 1 < posXMax)
        {
            heightRightTop = creator.terrain[arrayPosX + x + 1, arrayPosY + y +
                1] * stepSize;
        }

        bool orientationLeft = false;
```

```
        if (heightLeftBot < heightLeftTop && heightLeftBot < heightRightBot ||
            heightLeftBot > heightLeftTop && heightLeftBot > heightRightBot)
        {
            orientationLeft = true;
        }
        if (heightRightTop < heightLeftTop && heightRightTop < heightRightBot ||
            heightRightTop > heightLeftTop && heightRightTop > heightRightBot)
        {
            orientationLeft = true;
        }
        if (orientationLeft)
        {
            vertices[index] = new Vector3(x * scaleFieldH, heightLeftBot, y *
                scaleFieldV);
            vertices[index + 1] = new Vector3(x * scaleFieldH, heightLeftTop, y *
                scaleFieldV + scaleFieldV);
            vertices[index + 2] = new Vector3(x * scaleFieldH + scaleFieldH,
                heightRightBot, y * scaleFieldV);

            vertices[index + 3] = new Vector3(x * scaleFieldH + scaleFieldH,
                heightRightTop, y * scaleFieldV + scaleFieldV);
            vertices[index + 4] = new Vector3(x * scaleFieldH + scaleFieldH,
                heightRightBot, y * scaleFieldV);
            vertices[index + 5] = new Vector3(x * scaleFieldH, heightLeftTop, y *
                scaleFieldV + scaleFieldV);
        }
        else
        {
            vertices[index] = new Vector3(x * scaleFieldH, heightLeftTop, y *
                scaleFieldV + scaleFieldV);
            vertices[index + 1] = new Vector3(x * scaleFieldH + scaleFieldH,
                heightRightTop, y * scaleFieldV + scaleFieldV);
            vertices[index + 2] = new Vector3(x * scaleFieldH, heightLeftBot, y *
                scaleFieldV);

            vertices[index + 3] = new Vector3(x * scaleFieldH + scaleFieldH,
                heightRightBot, y * scaleFieldV);
            vertices[index + 4] = new Vector3(x * scaleFieldH, heightLeftBot, y *
                scaleFieldV);
            vertices[index + 5] = new Vector3(x * scaleFieldH + scaleFieldH,
                heightRightTop, y * scaleFieldV + scaleFieldV);
        }
    }
}
public void CreateTriangleField(int x, int y)
{
    int index = (x + (y * fieldCountH)) * 6;

    triangles[index + 0] = index + 0;
    triangles[index + 1] = index + 1;
```

```
        triangles[index + 2] = index + 2;

        triangles[index + 3] = index + 3;
        triangles[index + 4] = index + 4;
        triangles[index + 5] = index + 5;
    }
    public void UpdateMesh()
    {
        m.vertices = vertices;
        m.uv = uvs;
        m.triangles = triangles;

        m.RecalculateNormals();
        m.RecalculateBounds();
        meshFilter.mesh = m;

        //MeshCollider meshCollider = GetComponent(typeof(MeshCollider)) as
            MeshCollider;
        //meshCollider.sharedMesh = meshFilter.mesh;
    }
}
```

Listing C.2: Chunk Klasse des 3D Terraingenerators

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

public class Tests : MonoBehaviour {

    TerrainCreator creator;
    public int currentActiveFields = 0;

    float deltaTime = 0.0f;
    float fps = 0;

    float countSeconds = 0;

    float lowestFPS = 99999;
    float highestFPS;

    float averageFPSSum = 0;
    int averageFPSCount = 0;

    float lastAverageFPS = 0;
    float lastHighFPS = 0;
    float lastLowFPS = 0;
```

```
float tempCameraSize = 0;

Stopwatch loadingWatch = new Stopwatch();
Stopwatch clickWatch = new Stopwatch();
Stopwatch raiseWatch = new Stopwatch();

int raiseCount = 0;
public float loadTimeElapsed = 0;

    void Start () {
    creator =
        (TerrainCreator)GameObject.Find("TerrainCreator").GetComponent(typeof(TerrainCreator));
    loadingWatch = new Stopwatch();
    }
void Update()
{
    fps = 1.0f / Time.deltaTime;
    countSeconds += Time.deltaTime;
    if (fps < lowestFPS)
    {
        lowestFPS = fps;
    }
    if (fps > highestFPS)
    {
        highestFPS = fps;
    }
    averageFPSSum += fps;
    averageFPSCount++;

    if (countSeconds > 10)
    {
        lastAverageFPS = averageFPSSum / (float)averageFPSCount;
        lastHighFPS = highestFPS;
        lastLowFPS = lowestFPS;
        averageFPSCount = 0;
        averageFPSSum = 0;
        lowestFPS = 99999;
        highestFPS = 0;
        countSeconds = 0;
    }

    if (Camera.main.orthographicSize != tempCameraSize)
    {
        tempCameraSize = Camera.main.orthographicSize;
        CalcActiveFields();
    }
}
public void CalcActiveFields()
{
    currentActiveFields = 0;
```

```
    for (int x = 0; x < creator.maxFieldCountH; x++)
    {
        for (int y = 0; y < creator.maxFieldCountV; y++)
        {
            Vector3 v = Camera.main.WorldToViewportPoint(new Vector3(x,
                creator.terrain[x, y], y));
            if (v.x <= 1 && v.y <= 1 && v.x >= 0 && v.y >= 0)
            {
                currentActiveFields++;
            }
        }
    }
}
void OnGUI()
{
    int w = Screen.width;
    int h = Screen.height;

    GUIStyle style = new GUIStyle();
    style.alignment = TextAnchor.UpperLeft;
    style.fontSize = h * 2 / 100;
    style.normal.textColor = new Color(1.0f, 0.0f, 0.0f, 1.0f);

    float textheight = h * 2 / 100;

    Rect rectTitle = new Rect(0, 0, w, textheight);
    Rect rectFps = new Rect(0, textheight, w, textheight);
    Rect rectActiveFields = new Rect(0, textheight*2, w, textheight);
    Rect rectLoadTimer = new Rect(0, textheight * 3, w, textheight);
    Rect rectClickTimer = new Rect(0, textheight * 4, w, textheight);
    Rect rectRaiseTimer = new Rect(0, textheight * 5, w, textheight);
    Rect rectRaiseCounter = new Rect(0, textheight * 6, w, textheight);

    string textTitle = "3D_" + creator.chunkCountH + "c_" + creator.fieldCountH
        + "f";
    GUI.Label(rectTitle, textTitle, style);
    string text = string.Format("Low:{0:0.0} Av:{1:0.0} High:{2:0.0}
        curr:{3:0.0}", lastLowFPS, lastAverageFPS, lastHighFPS, fps);
    GUI.Label(rectFps, text, style);
    string text2 = string.Format("Fields:{0:0}", currentActiveFields);
    GUI.Label(rectActiveFields, text2, style);
    string text3 = string.Format("Loading Time: " + loadingWatch.Elapsed);
    GUI.Label(rectLoadTimer, text3, style);
    string text4 = string.Format("Click Time: " + clickWatch.Elapsed);
    GUI.Label(rectClickTimer, text4, style);
    string text5 = string.Format("Raise Time: " + raiseWatch.Elapsed);
    GUI.Label(rectRaiseTimer, text5, style);
    string text6 = string.Format("Raise Count:{0}", raiseCount);
    GUI.Label(rectRaiseCounter, text6, style);
}
```

```
    public void StartLoadingTimer()
    {
        loadingWatch = new Stopwatch();
        loadingWatch.Start();
    }
    public void StopLoadingTimer()
    {
        loadingWatch.Stop();
    }
    public void StartClickTimer()
    {
        clickWatch = new Stopwatch();
        clickWatch.Start();
    }
    public void StopClickTimer()
    {
        clickWatch.Stop();
    }
    public void StartRaiseTimer()
    {
        raiseWatch = new Stopwatch();
        raiseWatch.Start();
    }
    public void StopRaiseTimer(int raiseCount)
    {
        raiseWatch.Stop();
        this.raiseCount = raiseCount;
    }
}
```

Listing C.3: Test Klasse des 3D Terraingenerators

Abbildungsverzeichnis

2.1. Screenshot aus dem Spiel Holiday Island von Sunflower 6
2.2. Nachbarpunkte eines Punktes 7
2.3. Map tile set 9
2.4. 2D isometrisches Terrain nach der Generierung 13
2.5. Ein Feld mit 4 Vertizes 16
2.6. Ein Feld mit 6 Vertizes 16
2.7. Ein Raster mit 10*10 Feldern 21
2.8. Terrain nach dem Erstellen des Rasters und Laden der Höheninformationen 22
2.9. Ein Feld in der Ausgangsposition 24
2.10. Ein Feld nach dem Rotieren 24
2.11. 3D Terrain nach dem Drehen der Felder 25
2.12. 3D Terrain nach dem Drehen der Felder und Verwendung orthografischer Kamera 26

3.1. Testergebnis für 100000 Felder. Werte in Bilder pro Sekunde 37
3.2. Testergebnis für das Anzeigen von Bereichen. Werte in Bilder pro Sekunde 39
3.3. Testergebnis für die Erhöhungsdauer. Werte in Sekunden 40
3.4. Testergebnis für die Erhöhungsdauer mit Collidern am 3D Terrain. Werte in Sekunden 40
3.5. Testergebnis für die Generierungsgeschwindigkeit. Werte in Sekunden 41

Tabellenverzeichnis

1.1. Top 10 Stragiespiele im Android Play Store vom 17.07.2015 2

3.1. Testgeräte 35

Listings

2.1. Laden der Terrain CSV Datei 7
2.2. Erstellen der Tiles 9
2.3. Setzen der Textur und Positionierung der Tiles 10
2.4. Umrechnung der Mauskoordinaten in Weltkoordinaten 14
2.5. Höhenänderung eines Punktes im Terrain 14
2.6. Generierung eines Feldes mit vier Vertizes 16
2.7. Generierung eines Feldes mit sechs Vertizes 17
2.8. Code zum Erstellen des Rasters 18
2.9. Funktion zur Erstellung der Punkte des Rasters 19
2.10. Funktion zur Erstellung der Dreiecke 20
2.11. Funktion zur Erstellung der gedrehten Punkte 22
2.12. Funktion zur Erstellung der Chunks des Terrains 25
2.13. Funktion zur Erstellung der Dreiecke 26
2.14. Code zur Erstellung des Colliders 28
2.15. Code zum Aktualisieren des Colliders 28
2.16. Funktion zur Ermittlung des geklickten Punktes 28
2.17. Funktion zur Erhöhung oder Absenkung eines Punktes 29
2.18. Rekursive Funktion zur Erhöhung oder Absenkung eines Punktes . . 30

3.1. Berrechnung der Bilder pro Sekunde in der Test Klasse 33
3.2. Funktion zu Errechnung der Anzahl der angezeigten Felder des 3D Terrains 34
3.3. Funktion zu Errechnung der Anzahl der angezeigten Felder des 2D Terrains 34

B.1. TerrainGenerator Klasse des 2D Terraingenerators 83
B.2. Test Klasse des 2D Terraingenerators 90

C.1. TerrainCreator Klasse des 3D Terraingenerators 95
C.2. Chunk Klasse des 3D Terraingenerators 101
C.3. Test Klasse des 3D Terraingenerators 105

Literaturverzeichnis

[1] T. Schuster, *Entwicklung einer isometrischen Graphik-Engine in Java*. Universitaet Koblenz Landau, 8 2004.

[2] A. Thorn, *How to Cheat in Unity 5: Tips and Tricks for Game Development*. Focal Press, 7 2015.

[3] C. Kelly, *Programming 2D Games*. Taylor and Francis Inc, 7 2012.

[4] "Axonometrie." `https://de.wikipedia.org/wiki/Axonometrie`. Online, accessed: 15 Oktober 2015.

[5] "Polygonnetz." `https://de.wikipedia.org/wiki/Polygonnetz`. Online, accessed: 16 August 2015.

[6] "Mesh." `http://docs.unity3d.com/ScriptReference/Mesh.html`. Online, accessed: 10 August 2015.

[7] C. Seifert, *Spiele entwickeln mit Unity 5: 2D- und 3D-Games mit Unity und C-Sharp fuer Desktop, Web und Mobile, 2. Auflage*. Carl Hanser Verlag, 7 2015.

[8] P. H. G. S. Alfred Nischwitz, Max Fischer, *Computergrafik und Bildverarbeitung: Band I: Computergrafik, Ausgabe 3*. Springer-Verlag, 7 2012.

[9] "Camera." `http://docs.unity3d.com/Manual/class-Camera.html`. Online, accessed: 10 August 2015.

[10] "Physics.raycast." `http://docs.unity3d.com/ScriptReference/Physics.Raycast.html`. Online, accessed: 16 August 2015.

[11] P. de Byl, *Holistic Mobile Game Development with Unity*. CRC Press, 2014.

[12] "Colliders." `http://docs.unity3d.com/Manual/CollidersOverview.html`. Online, accessed: 16 August 2015.

[13] "Camera.worldtoviewportpoint." http://docs.unity3d.com/ScriptReference/Camera.WorldToViewportPoint.html. Online, accessed: 15 Oktober 2015.

[14] "Time.deltatime." http://docs.unity3d.com/ScriptReference/Time-deltaTime.html. Online, accessed: 13 August 2015.

[15] "Stopwatch-klasse." https://msdn.microsoft.com/de-de/library/system.diagnostics.stopwatch(v=vs.110).aspx. Online, accessed: 15 Oktober 2015.

[16] J. Nielsen, *Usability Engineering*. 1993.