entwickler.press
shortcuts

Erfolgreiche Spiele-entwicklung

Minecraft-Welten erschaffen

Alexander Rudolph

Alexander Rudolph

Erfolgreiche Spieleentwicklung

Minecraft-Welten erschaffen

ISBN: 978-3-86802-549-1

Ein Imprint der Software & Support Media GmbH

1 Entwicklung eines Minecraft-Klons – Grundlegende Spielmechaniken

Der Hype um den Open-World-Megaseller Minecraft ist nach wie vor ungebrochen. Was aber verbirgt sich unter der Motorhaube eines solchen Spiels? Bevor wir uns zu einem späteren Zeitpunkt mit fortgeschrittenen Themen wie der Landschaftsgenerierung oder der Implementierung von komplexen Beleuchtungsmodellen widmen werden, müssen wir uns zunächst mit den grundlegenden Spielmechaniken auseinandersetzen. Am Beispiel einer „einfachen“ Demoanwendung erörtern wir die wichtigsten Programmabläufe, befassen uns mit einem effizienten Speichermanagement und zeigen, wie sich blockbasierte Spielewelten in Echtzeit modifizieren lassen.

Als am 17. Mai 2009 die erste Version von Minecraft (Classic) das Licht der Welt erblickte, hatte wohl noch niemand zu träumen gewagt, dass sich Microsoft fünf Jahre später die für die Entwicklung verantwortliche Spieleschmiede Mojang für sage und schreibe 2,5 Milliarden Dollar einverleiben würde. Ganz im Gegenteil, nicht wenige Spieler und Entwickler hatten das Spiel geradezu belächelt, denn im Vergleich zu den detailreichen Umgebungen, wie man sie aus zahlreichen Triple-A-Spielen kennt, wirkten die blockbasierten Landschaften anfangs noch ein wenig – drücken wir es mal freundlich aus – gewöhnungsbedürftig (**Abb. 1.1** und **1.2**).

Wer sich mit der puristisch wirkenden Darstellung nicht zufriedengeben möchte, dem steht es heutzutage zwar frei, die Spielewelt mittels diverser High-Resolution-Texture-Packs oder Shader-Modifikationen (Beispiel: Sonic Ether’s Unbelievable

Shaders) optisch aufzuwerten, natürlich wirkende Formen lassen sich auf diese Weise jedoch nicht ins Spiel integrieren.

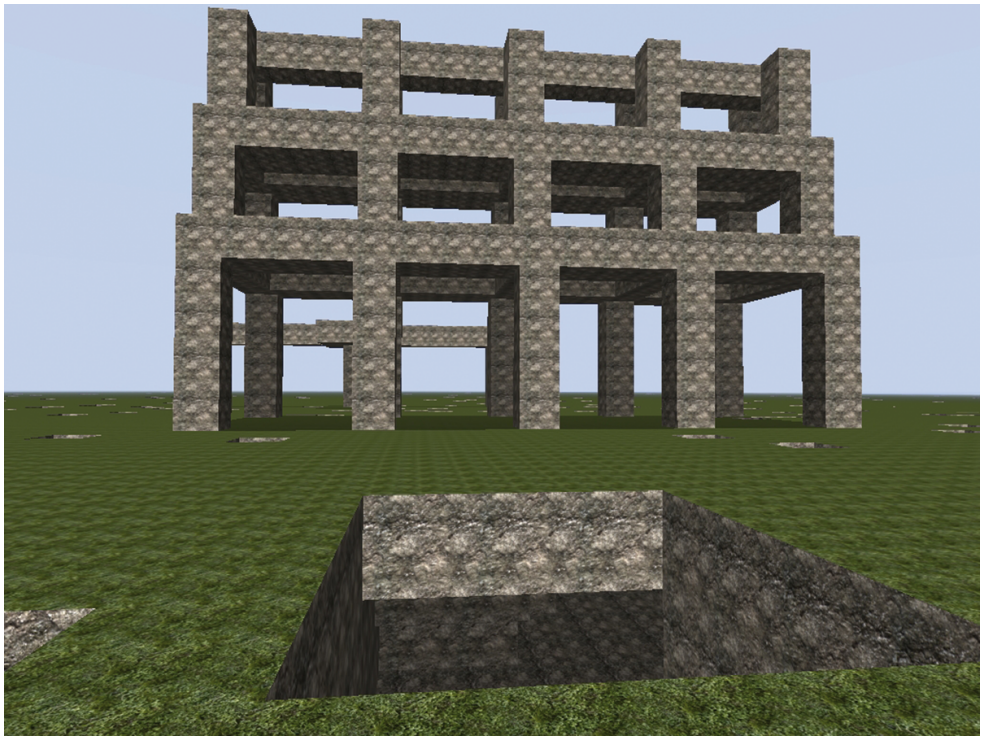

Abbildung 1.1: Blockbasierte Spielewelt (frühe Demoprogrammversion)

Abbildung 1.2: Ein weiteres Beispiel für eine blockbasierte Spielewelt, entnommen aus einer frühen Demoprogrammversion

Aber warum eigentlich sollte man dies überhaupt wollen? Selbst die schönste Umgebung wirkt auf Dauer eintönig und langweilig, wenn man sie nicht nach den eigenen Vorstellungen formen und verändern kann. Andererseits: Stören sich Kinder etwa an der begrenzten Formenvielfalt bei Bauklötzen und Legosteinen? Oder hält es sie gar vom Spielen ab? Nein, natürlich nicht, ganz im Gegenteil! Zu viele Formen hemmen die Kreativität und bremsen ein jedes Bauvorhaben aus. Anstatt einfach mal drauf loszubauen, denkt man viel zu häufig darüber nach, ob man lieber den einen oder doch lieber den anderen Typ Stein verwenden sollte.

Aber gilt das zuvor Gesagte denn ausschließlich für Kinder? Zugegeben, auf den ersten Blick scheint Minecraft kein Spiel für Erwachsene zu sein. Es ist weder ein Sport-, Kriegs-, Handels- oder Strategiespiel, geschweige denn ein Egoshooter. Doch der Wettkampfgedanke wird in vielen Spieletiteln zuweilen mächtig überstrapaziert. Gleiches gilt für die Actionsequenzen. Es muss nicht jedes Mal gleich alles explodieren. Nicht immer bedarf es eines Gegners, den es zu besiegen gilt. Es soll auch schon mal vorkommen, dass ein Spieler auf der Suche nach Entspannung ist. Und am Beispiel von Minecraft wird mehr als deutlich, dass Entspannung keinesfalls mit Langeweile gleichzusetzen ist. Spiele ohne übergeordnetes Spielziel können gleichzeitig entspannend wie auch abwechslungsreich sein. An einem Tag möchte man als Spieler vielleicht lediglich ein bislang unbekanntes Höhlensystem erkunden, während an anderen Tagen der Nachbau einer kompletten Stadt oder eines Fantasy-Settings (beispielsweise Mittelerde) auf dem Programm steht.

Minecraft aus der Perspektive eines Programmierers

An der Tatsache, dass ein Spiel wie Minecraft eine große Spielerschaft über einen langen Zeitraum hinweg in den Bann

zu ziehen vermag, bestand wohl spätestens einige Monate nach der Veröffentlichung der ersten, noch unfertigen Version keinerlei Zweifel mehr. Das Minecraft-Virus befällt jedoch keinesfalls nur Gamer. Nicht zuletzt durch die Vielzahl von YouTube-Videos, die einem das Potenzial dieses Spiels offenbaren, wird man als Programmierer geradezu angestachelt, sich auch einmal mit den technischen Aspekten des Programms zu befassen. Insbesondere für angehende Spieleprogrammierer stellt Minecraft ein interessantes Anschauungsobjekt dar, zeigt es doch, was ein Soloentwickler so alles auf die Beine stellen kann (zur Erklärung, am Anfang der Entwicklung handelte es sich zunächst um ein „nerdiges“ Ein-Mann-Projekt).

Die Fertigkeiten der Spieleentwicklung erlernt man am besten anhand von praktischen Beispielen. Weltraumspiele gelten in diesem Zusammenhang gemeinhin als besonders anfängerfreundliche Projekte, um sich mit den Grundlagen der Spielmechanik und der 3-D-Programmierung vertraut zu machen. Die wirklichen Herausforderungen, die etwa in der Darstellung von detaillierten, abwechslungsreichen Planetenoberflächen, weitläufigen Asteroidenfeldern oder volumetrischen Nebelwolken bestehen, kann man sich hingegen für einen späteren Zeitpunkt aufsparen. Doch ist diese Vorgehensweise wirklich so klug? Die Gefahr, dass man sich auf diese Weise in eine Sackgasse programmiert, ist jedenfalls nicht von der Hand zu weisen. Dinge, wie eine performante Szenendarstellung oder ein effizientes Speichermanagement sollten nicht auf die lange Bank geschoben werden. Gerade in diesen Bereichen kann man bei der Entwicklung eines Minecraft-Klons eine Menge lernen. Betrachten wir zum Einstieg einmal eine relativ kleine blockbasierte Welt mit einer Länge und Breite von jeweils 2 000 und einer Höhe von 256 Metern. Sofern nun die einzelnen Blöcke eine Kantenlänge von jeweils einem Meter besitzen, bietet unsere Spielewelt genügend Raum für 1 024 Milliarden

Blöcke (2 000*2 000*256). Würde man nun beispielsweise für die Beschreibung der jeweiligen Blockeigenschaften (hierzu zählen selbstverständlich auch Luft- und Wasserblöcke) eine einzelne 1-Byte-Variable vom Datentyp *unsigned __int8* verwenden (die Position eines Blocks in der Spielewelt ergibt sich aus der Position im zugehörigen *unsigned __int8*-Array), so kämen wir in der Summe auf einen Gesamtspeicherbedarf von 1 024 GB. Es ist daher abzusehen, dass wir bei deutlich größeren Welten unweigerlich mit einem Speicherproblem zu kämpfen hätten. Für die Lösung dieses Dilemmas bieten sich nun zwei Vorgehensweisen an, die selbstverständlich auch miteinander kombiniert werden können. Beim Datenstreaming lädt man lediglich die Daten der sichtbaren Bereiche der Spielewelt von der Festplatte in den Hauptspeicher, während man die nicht mehr benötigten Daten im Gegenzug aus dem Hauptspeicher entfernt. Um die hierbei anfallenden Datenmengen sowohl auf der Festplatte als auch im Hauptspeicher zu begrenzen, bietet sich darüber hinaus der Einsatz eines geeigneten Kompressionsverfahrens an.

Eine möglichst performante (ruckelfreie) Darstellung der blockbasierten Spielewelt ist ebenfalls alles andere als trivial und ohne Einbeziehung moderner Rendering-Techniken so gut wie unmöglich. In diesem Zusammenhang sind der Einsatz von Geometry Instancing zur Verringerung der Anzahl der erforderlichen Render-Aufrufe (Draw Calls) oder die Verwendung von Texture Arrays zur Vermeidung unnötiger Texturwechsel praktisch alternativlos. Auch die vorausgehenden Sichtbarkeitsberechnungen sind nicht ganz ohne. Im ersten Schritt müssen wir zunächst die Oberflächenblöcke identifizieren, die im Unterschied zu den von allen sechs Seiten vollständig verdeckten inneren Blöcken zumindest potenziell sichtbar sind. Im Anschluss daran können wir die jeweils sichtbaren Flächen (es können unter keinen Umständen alle sechs Flächen gleichzeitig sichtbar sein) der Oberflächenblöcke ermitteln und darstellen.

Darüber hinaus bietet einem die Entwicklung eines Minecraft-Klons die Möglichkeit, sich mit einer Vielzahl von weiteren Aspekten der Spieleentwicklung auseinanderzusetzen. Fassen wir zusammen:

- Das Ziel der Entwicklung besteht in der Erschaffung einer mittels Ray Casting dynamisch veränderbaren, blockbasierten Spielewelt mit dynamischen Tag-Nacht-Wechseln, unterschiedlichen Wetterverhältnissen sowie einer realistischen Himmelsdarstellung samt volumetrischer (dreidimensionaler) Wolken.
- Die Generierung der Landschaften, Höhlensysteme und Vegetationszonen erfolgt mithilfe unterschiedlicher prozeduraler Techniken.
- Der Spieler hingegen ist für alles Architektonische verantwortlich – von der Errichtung des ersten Gebäudes bis hin zur Planung und dem Bau ganzer Städte.
- Damit sich eine derart komplexe Spielewelt ruckelfrei darstellen lässt, verteilen wir die hierfür erforderlichen Berechnungen (Datenstreaming, Datenkompression, Sichtbarkeitstests usw.) auf mehrere Worker-Threads (Stichwort: Parallelisierung).
- Darüber hinaus nutzen wir unser Demoprogramm im weiteren Verlauf als eine Art Testumgebung für die Implementierung verschiedener komplexer Beleuchtungsmodelle.

Datenkompression – der Schlüssel zur Effizienz

Kennen Sie eventuell den im Jahr 1999 erschienenen Science-Fiction-Film *The 13th Floor* oder vielleicht die im Jahr 1964 veröffentlichte Romanvorlage *Simulacron-3* des Autors Daniel F. Galouye? Im Film erschaffen Computerwissenschaftler eine Simulation der Stadt Los Angeles, die so lebensecht wirkt, dass selbst die Bewohner denken, sie seien real und würden im Jahr 1937 leben. Was anfangs noch als großer Triumph der

Wissenschaft gefeiert wurde, entwickelt sich für die Wissenschaftler im weiteren Verlauf des Films zu einem regelrechten Albtraum, denn auch sie müssen erkennen, dass ihre eigene Welt und sie selbst ebenfalls nur als Simulationen in einem Computer existieren – der, wie am Ende des Films offenbart wird, lediglich ein Teil einer noch größeren Simulation ist.

Die Frage, ob wir selbst nun real sind oder nicht, kann wahrscheinlich nur Gott oder ein göttlicher Programmierer beantworten. Es ist jedoch ganz interessant, einmal darüber nachzudenken, auf welchem Wege die Wissenschaftler im Film die komplexe Speicherproblematik wohl gelöst haben könnten. Denn auch wir müssen uns tagtäglich mit der Frage auseinandersetzen, wie sich die stetig anwachsenden Datenmengen der heutigen Spiele verwalten und handhaben lassen. Betrachten wir in diesem Zusammenhang einmal – als eine Art Extrembeispiel – eine grenzenlos wirkende, vollkommen flache, blockbasierte Spielewelt und stellen die Frage, wie viel Speicher man für die Beschreibung dieser Welt benötigt, sofern der Untergrund lediglich aus einheitlichen Blöcken besteht. Die Antwort ist einfach, man benötigt lediglich Speicherplatz für vier simple Zahlenwerte:

- die Description-ID (mit anderen Worten: der Blocktyp) des Bodenblocks sowie die Anzahl der Bodenblöcke
- die Description-ID des Luftblocks sowie die Anzahl der Luftblöcke

Das in unserem Beispiel verwendete Kompressionsverfahren ist unter der Bezeichnung Lauflängenkodierung (englische Bezeichnung: **R**un-**L**ength **E**ncoding, kurz **RLE**) bekannt und stellt die einfachste Möglichkeit dar, einen Datensatz verlustfrei zu komprimieren. Trotz seiner Schlichtheit ist dieser Algorithmus für die Lösung unseres Speicherproblems bestens geeignet, da unsere Spielewelten größtenteils aus Luft-, Wasser- beziehungsweise unterhalb der Oberfläche aus gleichartigen

Grundgesteinsblöcken bestehen. Wie man anhand der nachfolgenden Beispiele gut erkennen kann, vergrößert sich die Komprimierungsrate, wenn sich gleichartige Blöcke in zunehmendem Maße aneinanderreihen (große Anzahl von Wiederholungen):

- **Beispiel 1** – minimale Komprimierungsrate. Block-Description-IDs (unkomprimiert): 0, 1, 0, 2, 1, … Block-Description-IDs (komprimiert/lauflängenkodiert): 0, 1, 1, 1, 0, 1, 2, 1, 1, 1, …
- **Beispiel 2** – vergleichsweise hohe Komprimierungsrate. Block-Description-IDs (unkomprimiert): 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 2, 2, 2, 2, 2, 2, … Block-Description-IDs (komprimiert/lauflängenkodiert): 0, 9, 1, 4, 0, 3, 2, 6, …

Für den praktischen Einsatz, etwa bei der Durchführung von Sichtbarkeitsberechnungen, sind längenkodierte Daten eher ungeeignet, da man lediglich direkten Zugriff auf die Block-Description-IDs von zwei der sechs benachbarten Blöcke hat. Im Rahmen eines vollständigen Sichtbarkeitstests müssen wir jedoch für alle sechs Seiten eines Blocks überprüfen können, ob sie durch benachbarte Blöcke verdeckt werden oder eben nicht. Da es jedoch, wie wir bereits im vorangegangenen Abschnitt festgestellt haben, aufgrund der Speicherproblematik ebenfalls nicht infrage kommt, sämtliche Daten auf unkomprimierte Art und Weise zu verwalten, bleibt uns nur der Kompromiss:

- Die Gesamtheit aller Daten der Spielewelt speichern wir sowohl auf der Festplatte als auch im Hauptspeicher in längenkodierter Form.
- Lediglich den sichtbaren Ausschnitt der Spielewelt speichern wir unkomprimiert in einem Array vom Typ *unsigned __int8* ab, das, in Abhängigkeit von der Bewegung des Spielers, kontinuierlich aktualisiert wird (wie genau das funktioniert, erörtern wir im nachfolgenden Abschnitt). Zugriff auf das aktualisierte Array, das wir im Demoprogramm als

LocalBlockArray bzw. Umgebungs-Array bezeichnen, erhält man mithilfe der Zeigervariable *BlockID_Int* pLocalBlockArray* (Hinweis: *typedef unsigned __int8 BlockID_Int*).

Unter Berücksichtigung der Ausdehnung des sichtbaren Bereichs der Spielewelt in x-, y- und z-Richtung (*iLocalRangeX*, *iMaxHeight* und *iLocalRangeZ*) sowie der Spieleweltposition eines Blocks (*ix*, *iy* und *iz*) lässt sich die korrespondierende Position innerhalb des Umgebungsarrays (*PosID*) mithilfe der folgenden einfachen Formel berechnen:

```
PosID = ix + iz*iLocalRangeX + iy*iLocalRangeX*iLocalRangeZ
```

bzw.

```
PosID = ix + iz*iLocalRangeX + iy*NumLocalBlocksXZPlane
```

Hinweis: Bei der Angabe der Blockpositionen verwenden wir ausschließlich Integer-Variablen (zu Erkennen am Namenspräfix *i*), da wir hierbei aufgrund der von uns gewählten Blockskalierung von 1 ohne Nachkommastellen auskommen. Darüber hinaus ersparen wir uns bei der Berechnung der Array-Indizes (*PosID*) eine Vielzahl von Typumwandlungen, wie sie bei alternativer Verwendung von Fließkommavariablen erforderlich wären.

Um nun die Array-Indizes (*PosID2*) der benachbarten Blöcke berechnen zu können, müssen wir die vorangegangene Gleichung in Abhängigkeit von der jeweiligen Lage der einzelnen Blöcke, wie nachfolgend gezeigt, modifizieren:

- Zugriff auf die benachbarten Blöcke in x-Richtung:

```
PosID2 = (ix-1) + iz*iLocalRangeX + iy*NumLocalBlocksXZPlane
PosID2 = (ix+1) + iz*iLocalRangeX + iy*NumLocalBlocksXZPlane
```

- Zugriff auf die benachbarten Blöcke in y-Richtung:

```
PosID2 = ix + iz*iLocalRangeX + (iy-1)*NumLocalBlocksXZPlane
PosID2 = ix + iz*iLocalRangeX + (iy+1)*NumLocalBlocksXZPlane
```

- Zugriff auf die benachbarten Blöcke in z-Richtung:

```
PosID2 = ix + (iz-1)*iLocalRangeX + iy*NumLocalBlocksXZPlane
PosID2 = ix + (iz+1)*iLocalRangeX + iy*NumLocalBlocksXZPlane
```

Möchte man jetzt beispielsweise im Zuge der Sichtbarkeitsberechnungen nacheinander auf alle im Array gespeicherten Blöcke zugreifen, bietet sich hierfür die nachfolgend gezeigte Vorgehensweise an:

```
for(iy = iMaxHeight-1; iy > -1; iy--)
{
  for(iz = izMin; iz < izMax; iz++)
  {
    for(ix = ixMin; ix < ixMax; ix++)
    {
      PosID = ix + iz*iLocalRangeX + iy*NumLocalBlocksXZPlane;
      [Block auf Sichtbarkeit testen etc.]
}}}
```

Chunk-basiertes Datenmanagement

Der zuvor gefundene Kompromiss hilft uns zwar dabei, einerseits die Performance bei der Durchführung der Sichtbarkeitstests, beim Rendering sowie bei den Beleuchtungsberechnungen zu maximieren, er hat jedoch auch den gravierenden Nachteil, dass bei größeren Spielewelten die Zeitdauer, die für die Aktualisierung des *LocalBlockArrays* (zur Erinnerung, dieses Array repräsentiert den sichtbaren Ausschnitt der Spielewelt) erforderlich ist, mitunter mehrere Sekunden beträgt. Um verstehen zu können, zu welchen Problemen das führen kann, müssen Sie zunächst verstehen, wie die Aktualisierung des *LocalBlockArrays* im Rahmen unseres Demoprogramms überhaupt vonstatten geht.

Im Prinzip orientieren wir uns hierbei an der Funktionsweise des beim Rendering zum Einsatz kommenden Double Bufferings. Um jedoch einerseits Missverständnissen vorzubeugen und um andererseits der Tatsache Rechnung zu tragen, dass die Aktualisierung der Buffer bzw. Arrays parallel zum Hauptprogramm in einem separaten Thread durchgeführt wird, bezeichnen wir das von uns verwendete Verfahren im weiteren Verlauf hingegen als Parallel Buffering.

- Im ersten Schritt initialisieren wir zunächst zwei *LocalBlockArrays* und speichern den sichtbaren Ausschnitt der Spielewelt in *LocalBlockArray1*. Hierbei gehen wir so vor, dass sich die x- und z-Koordinaten der Spielerposition zu Beginn des Aktualisierungsvorgangs genau im Zentrum des sichtbaren Bereichs befinden:

```
LocalBlockArray1 = new BlockID_Int[NumLocalBlocks];
LocalBlockArray2 = new BlockID_Int[NumLocalBlocks];
[sichtbaren Ausschnitt der Spielewelt zunächst in LocalBlockArray1 speichern]
```

- Nachdem wir sämtliche Daten aller potenziell sichtbaren Blöcke ins *LocalBlockArray1* übertragen haben, speichern wir die Adresse des aktualisierten Arrays zur weiteren Verwendung in der Zeigervariable *pLocalBlockArray*:

```
pLocalBlockArray = &LocalBlockArray1[0];
```

- Während man nun mittels des *pLocalBlockArray*-Zeigers unter anderem beim Rendering oder bei der Durchführung der Sichtbarkeitstests auf die Daten von *LocalBlockArray1* zugreifen kann, werden parallel dazu, in einem separaten Thread, die im *LocalBlockArray2* gespeicherten Daten aktualisiert. Wiederum gilt, dass sich die x- und z-Koordinaten der Spielerposition zu Beginn des Aktualisierungsvorgangs genau im Zentrum des neu abgegrenzten sichtbaren Bereichs befinden.
- Nach Abschluss der Aktualisierung müssen wir lediglich die im *pLocalBlockArray*-Zeiger gespeicherte Adresse ändern, damit im Verlauf des erneuten Updates von *LocalBlockArray1* auf die Daten von *LocalBlockArray2* zugegriffen werden kann:

```
pLocalBlockArray = &LocalBlockArray2[0];
```

Wenn nun ein Spieler die Grenzen des sichtbaren Bereichs der Spielewelt erreicht, ohne dass die Aktualisierung des zweiten *LocalBlockArrays* abgeschlossen ist, blickt er buchstäblich ins Nichts. Die Welt scheint einfach zu enden – jedenfalls für den