Performancevergleich einiger Speicherstrukturen und Mesh-Generations-Algorithmen von Voxel Terrains mit blockartigem Erscheinungsbild

Luis Rothenhäusler (20202459)

Brandenburg, der 26. Januar 2022

Inhaltsverzeichnis

1 Einleitung

Voxelbasierte Welten, die in Blöcken repräsentiert werden, sind in Videospielen heutzutage keine Seltenheit mehr. Hierfür wird für jeden Block in der Welt eine ID gespeichert. Die sich daraus ergebenden Daten werden dann in ein Mesh überführt, welches angezeigt wird. Doch welche Speicherstrukturen eignen sich für die Darstellung solcher Welten, und wie performant sind diese? Welche Möglichkeiten zur Generierung von Meshes ergeben sich daraus? Wie lassen sich diese miteinander vergleichen?

Das Ziel dieser Arbeit ist es, diese Fragen zu beantworten. Hierfür werden zunächst gewisse Rahmenbedingungen gesetzt. Es wird von einer endlichen Welt ausgegangen. Weiterhin erfolgt die Einfärbung des Meshs über Vertex-Farben, nicht mittels Texturen. Zum Performancevergleich werden einige Variablen gesammelt, unter anderem der benötigte Speicherplatz, Zugriffszeiten, die Dauer der Mesh-Generierung sowie Anzahl der Vertices und Dreiecke im resultierenden Mesh.

2 Grundlagen

Es wird die Unity-Engine genutzt, um diese Tests durchzuführen. Daher wird die Implementation der Speicherstrukturen und Meshgenerierungsalgorithmen in C# erfolgen. Weiterhin wird Rücksicht darauf genommen, das Unity Job-System sowie den Burst-Kompilierer zu nutzen. Beide diese Technologien sind Teil von Unitys Data-Oriented-Technology-Stack (DOTS). Die Unity Engine orientiert sich aktuell intern um. Hierbei ist das Ziel, alle bestehenden Features in DOTS neu umzusetzen. Das Job System ist hierbei für Multithreading zuständig. Es soll ermöglichen, Multithreaded Code zu schreiben, ohne sich um Race Conditions sorgen zu müssen. Der Burst-Kompiler soll C# in performanteren Maschinencode zu übersetzen.

Da es nicht möglich ist, eine ganze voxelbasierte Spielwelt in einem Stück zu speichern, wird diese üblicherweise in Chunks aufgeteilt. Die Speicherstrukturen und die Mesh-Generation erfolgt jeweils pro Chunk. Wie groß die Chunks sind ist durch eine Konstante, die Kantenlänge des Chunks, gegeben. Chunks könnten hierbei jede Form annehmen. In dieser Arbeit wird davon ausgegangen, dass die Chunks Würfel mit fester Kantenlänge sind und die Kantenlänge eine gerade Zahl ist. Weiterhin wird in den Chunks für jeden Voxel nur eine ID gespeichert. Diese ist eine ganzahlige Zahl. Mit einem Höchstwert von 65535 ist unsigned Short hierfür ausreichend und spart im Vergleich zu Integer die Hälfte an Bits.

3 Speicherstrukturen

Im Folgenden werden die einzelnen Speicherstrukturen beschrieben. Weiterhin wird auf die Kriterien eingegangen, welche zur Bewertung herangezogen werden.

3.1 Strukturen

Die Speicherstrukturen müssen es ermöglichen, die Voxel-ID an bestimmten Koordinaten zu finden und zu setzen. Weiterhin müssen sie als Structs implementiert sein, um im Job-System als Parameter genutzt werden zu können.

3D-Array Diese Repräsentation ist der tatsächlichen Anordnung der Voxel am Nächsten. Im dreidimensionalen Array entsprechen die drei Indices des Zugriffes jeweils den x-, y- und z-Koordinaten des entsprechenden Voxels. Da für jeden Voxel immer eine ID gespeichert wird, ist der Speicherbedarf des 3D-Arrays immer gleich.

1D-Array In dem eindimensionalem Array wird eine dreidimensionale Repräsentation in ein einzelnes, langes Array überführt. Hierzu werden die Koordinaten mit einer Formel in einen Index übersetzt und andersherum. Der Vorteil gegenüber der dreidimensionalen Repräsentation ist der schnellere Speicherzugriff. Jedoch kommt hier die Umrechnung von Koordinaten hinzu. Vielleicht kann an manchen Stellen hier noch mit cleveren Indexoperationen gearbeitet werden. Wie auch bei dem 3D-Array ist der Speicherbedarf des 1D-Arrays immer gleich.

Octree Ein Octree ist eine dynamischere Repräsentation der Daten im Raum. Wie bei einem Baum üblich, gibt es hier einen einzelnen Ursprungsknotenpunkt. Jeder Knotenpunkt beinhaltet entweder eine Voxel-ID (Blattknoten) oder acht weitere Knoten (Astknoten). Die acht weiteren Knoten repräsentieren jeweils ein Achtel des Raumes des Elternknotens. In diesem Beispiel wird davon ausgegangen, dass der Octree den Raum gleichmäßig teilt. Weiterhin enthält jeder Knoten Informationen über seine Stufe im Baum. Damit können Rückschlüsse auf die Größe des Knotens gezogen werden.

Der Octree ist in seinem Speicherbedarf abhängig von der Komplexität des darzustellenden Chunks. Für Leseoperationen muss der Baum durchschritten werden bis der entsprechende Blattknoten gefunden wird. Bei Schreiboperationen müssen gegebenenfalls Teile des Baumes umgeschrieben werden. Dafür ist der Speicherbedarf potentiell geringer als bei den Arrays. Außerdem ist es im Schnitt wahrscheinlich einfacher Meshes mit weniger Vertices und Dreiecken zu erstellen, da ein Blatt in vielen Fällen größer als nur ein Voxel ist.

3.2 PerformanceiKriterien

Es gibt einige Kriterien, die für die Performance der Speicherstrukturen genutzt werden. Die Lese- und Schreibzeit ist in vielen Fällen wahrscheinlich wichtiger als der Speicherbedarf. In Bereichen, in denen es selten zu Lese- und Schreiboperationen kommt, wird der Speicherbedarf relevanter.

Speicherbedarf Wie viel Speicherplatz verwendet die entsprechende Speicherstruktur?

Lesezeit Wie lange dauert es die ID eines Voxels an spezifischen Koordinaten zu lesen?

Schreibzeit Wie lange dauert es die ID eines Voxels an spezifischen Koordinaten zu schreiben?

4 Mesh-Generierung

Das Generieren eines Meshes beschreibt, in dieser Arbeit, das Überführen der Speicherstruktur in ein Modell aus Vertices und Dreiecken, welches in der Welt angezeigt werden kann. Hierbei sollte versucht werden die Zahl der Vertices und Dreiecke möglichst gering zu halten, um die Grafikkarte zu entlasten. Um dies zu erreichen werden die verschiedensten Techniken angewandt. Diese können sich je nach Speicherstruktur in der Implementation unterscheiden. Generell wird jedoch versucht, nur Voxelseiten darzustellen welche nicht von anderen verdeckt sind. Außerdem werden Vertices an der selben Position oft zusammengefasst. Da wir jedoch mit Vertex-Coloring und Flat-Shading arbeiten, muss hierbei beachtet werden, dass nur, weil zwei Vertices an der selben Stelle liegen, diese nicht unbedingt zusammengefasst werden können. Hierbei können potentiell weitere Vertices zusammengefasst werden, wenn der Flat-Shading Effekt durch einen Shader erreicht werden kann.

4.1 Algorithmen

Algorithmen, welche die Chunks Voxel für Voxel durchschreiten, können für alle Speicherstrukturen verwendet werden. Der Octree kann hierbei potentiell Arbeit einsparen, da nur für jeden Blattknoten ein Mesh erstellt werden muss.

Generell kann auch überlegt werden für jeden Chunk sechs Meshes zu generieren, eins für jede Seite. So kann der Chunk von der Kamera abgewandte Seiten deaktivieren und somit für die Grafikkarte weitere Vertices und Dreiecke einsparen.

Der einfachste Ansatz Es wird der Chunk Voxel für Voxel durchlaufen. Soll ein Voxel dargestellt werden, wird für jede seiner Seiten zunächst geprüft ob diese verdeckt ist. Dafür wird geprüft, ob es einen benachbarten Voxel gibt, welcher diese Seite verdeckt. Ist das nicht der Fall, werden die vier Eckpunkte der Seite festgehalten und die entsprechenden Dreiecke aufgespannt.

Seitenweise Vertices zusammenfassen Anders als bei dem ersten Algorithmus werden hier Vertices mit gleicher Position, Farbe und Normalenvektor Zusammengefasst.

Blattweise Dieser Algorithmus funktioniert nur bei Octrees. Hierbei wird jeder Blattknoten des Octrees betrachtet. Bei der Überprüfung der Nachbarn kann hier auf die Stufe des Blattknotens im eigenen Baum, sowie die Höhe des Nachbarblatts Bezug genommen werden. Sollte das Nachbarblatt auf der selben Stufe oder höher im Baum sein, so muss beispielsweise nur ein einziger Check gemacht werden um zu wissen, ob die Seite des

Blattes angezeigt werden muss oder nicht. Weiterhin werden für eine Seite eines Blattes immer vier Vertices benötigt, egal wie groß der Bereich ist, welcher von dem Blatt abgedeckt wird.

4.2 Performance-Kriterien

Welche Kriterien hier wie wichtig sind, kommt darauf an, wie das Leistungsverhältnis zwischen CPU und GPU ist. Es muss beispielsweise nicht viel Zeit verwendet werden ein effektives Mesh zu generieren, wenn die Grafikkarte auch mit einem komplexeren Mesh nicht ausgelastet ist.

Generationszeit Wie lange braucht der ensprechende Algorithmus ein Mesh zu erstellen?

Anzahl der Vertices Wie viele Vertices hat das resultierende Mesh?

Anzahl der Dreiecke Wie viele Dreiecke besitzt das resultierende Mesh?

5 Test-Setup

Um die Speicherstrukturen und Algorithmen miteinander vergleichen zu können, werden zufällig generierte Welten genutzt. Diese Welten basieren auf jeweils einem Seed, welcher bei gleichem Input die gleiche Welt erzeugt. Damit sind die einzelnen Kriterien miteinander vergleichbar. Die zeitbasierten Kriterien werden anhand der Systemzeit ermittelt. Auch sollte darauf geachtet werden immer die selbe Performance zur Verfügung zu stellen.

6 Erwartungen

Ich erwarte, dass das 1D-Array sich, aufgrund seiner schnellen Lese- und Schreibzugriffe, als besonders nützlich erweist in Bereichen, welche sich oft verändern. Der Octree ist hingegen für Chunks geeignet, welche lange unverändert bleiben und nicht zu komplex sind. Sollte es bei der Mesh-Generierung große Unterschiede in der benötigten Zeit geben, bietet es sich hier wahrscheinlich an nach einer Voxeländerung zunächst so schnell wie möglich ein Mesh zu errechnen. Nachdem sich der entsprechende Chunk einige Zeit nicht geändert hat, kann hier im Hintergrund ein effektiveres Mesh errechnet werden. Weiterhin erwarte ich, dass sich im Laufe der Entwicklungszeit noch mehr Möglichkeiten für Meshgenerierungsalgorithmen ergeben werden.