Digital Entertainment Technologies

Projektbericht

von Tammam Nader und Daniel Wolf

Inhalt:

- Problemstellung
  Welche Themen behandelt unser Projekt?
- Motivation
  Warum haben wir uns für diese Themen entschieden?
- Features
  Welche Inhalte bringt unser Projekt mit?
- Herangehensweise und technische Umsetzung im Wechsel
  Wie haben wir unsere Ideen technisch umgesetzt?
- Fazit

Für dieses Projekt wurde hauptsächlich die Videoreihe von b3agz „Code a Game like Minecraft in Unity“ verwendet.[1]

1 GitHub Repository: https://github.com/b3agz/Code-A-Game-Like-Minecraft-In-Unity und YouTube Tutorial: https://www.youtube.com/playlist?list=PLVsTSlfj0qsWEJ-5eMtXsYp03Y9yF1dEn
[zuletzt abgerufen am: 31.01.2020]

Problemstellung

Unsere Problemstellung umfasste die prozedurale Generierung einer Voxel-Welt wie man sie aus Minecraft kennt in Verbindung mit einem Kampfsystem.

Motivation

Die Möglichkeit Welten prozedural zu generieren und somit abwechslungsreiche, einfach veränderbare Welten zu generieren hat von vorn herein unser Interesse geweckt.
Jedoch wollten wir auch ein spielbares Erlebnis und haben uns so dazu entschieden ein Kampfsystem einzubauen, weswegen das Projekt im Folgenden „Spiel“ genannt wird.
Die Idee das man die Welt und ihre Bestandteile im Kampf nutzt, existierte schon vorher. Inspiration gab die Serie „Avatar – Der Herr der Elemente“ / „Avatar: The Last Airbender“

Features

Features und Projektideen, die es auch in das Spiel geschafft haben sind:

- Die Bildung einer Insel, welche von Wasser umgeben ist. So haben wir die Grenzen unserer Welt gebildet, ohne dass der Spieler einfach nur am Rand der Welt herunterfällt oder man auf unüberwindbare Mauern zurückgreifen muss.
- Die Unterteilung der Welt in Biome, welche verschiedene Materialien und Flora aufweisen und fließend ineinander übergehen.
- Die Interaktion mit Blöcken, sprich das Abbauen und Setzen von Blöcken. Dazu gehört auch die Nutzung der Blöcke als Projektile im Kampf.
- Die Lagerung der abgebauten Blöcke in einem Inventar. Man muss natürlich die Möglichkeit haben auf abgebaute Blöcke zugreifen zu können, um sie zu setzen oder im Kampf zu nutzen.
- Das Vorkommen von Gegnern in der Welt, die den Spieler angreifen, wenn er in der Nähe ist.
- Verschiedene Blöcke haben verschiedene Eigenschaften im Kampf. Wenn sie als Projektil verwendet werden, haben sie unterschiedliche Wirkungen auf den getroffenen Gegner. Dabei lässt sich nicht jeder Block als Projektil verwenden, da wir der Meinung sind, dass einen Sandblock oder Laubblock abzuschießen wenig Sinn ergibt.
- Das Abfeuern von Blöcken kostet Energie. Dadurch wird sichergestellt, dass man nicht einfach alle Blöcke, die man besitzt, in kürzester Zeit abfeuern kann und somit praktisch unbesiegbar wird. Dazu gehört auch, dass der Spieler eine Lebensanzeige hat.
- Eine GameOver Szene, die erreicht wird, wenn der Spieler besiegt wird. Von dort aus kann man das Spiel beenden oder zurück ins Hauptmenü gelangen.
- Ein Hauptmenü, in dem das Spiel gestartet, Änderungen an den Optionen vollzogen und ein Seed eingegeben werden kann.

Texturen und 3D Objekte sind weitestgehend selbst erstellt. Uns war wichtig, dass auch die Grafik Eindruck macht, daher haben wir auf die Minecraft Standardtexturen verzichtet.
Features die es leider nicht geschafft haben sind zum einen, dass das Wasser auch eine Wasserphysik hat. Außerdem fehlen leider die Spieleranimationen, da es beim Animieren zu Problemen mit den in Blender hergestellten Objekten kam. Unsere geplante Höhlengenerierung konnte nicht umgesetzt werden.

Herangehensweise und technische Umsetzung

Unser Spiel bedient sich der Voxel-Engine aus dem Youtube Tutorial von b3agz.

Weltgenerierung
Wie bei den meisten besteht unsere Welt aus Chunks, welche aus Voxeln bestehen.
Damit es bei einer Welt von z.B. 100x100 Chunks nicht zu Performanzproblemen kommt, haben wir eingebaut, dass nur eine gewisse Anzahl an Chunks um den Spieler herum geladen wird.
Diese *ViewDistance* kann auch im Hauptmenü verändert werden, damit man sie an die Leistungsfähigkeit seines PC anpassen kann.

Erreicht wird das durch die Methode *CheckViewDistance()*, welche in der *Update()*-Methode unserer *World*-Klasse aufgerufen wird, wenn der Spieler einen neuen Chunk betritt. Dort werden dann die Chunk-Koordinaten anhand der Spielerposition in eine *activeChunksList* hinzugefügt.
Update() stellt außerdem sicher, dass diese Chunks in eine *chunksToUpdateList* kommen.
Die Inhalte dieser Liste, also die Chunk-Koordinaten, werden von der Methode *UpdateChunks()* aktualisiert. *UpdateChunks()* arbeitet zur Verbesserung der Performanz mit Threads.
Dabei wird auch die MeshData und die Belichtung der jeweiligen Chunks berechnet.
Im Folgenden werden diese aktualisierten Chunks dann einer *chunksToDrawList* hinzugefügt, die wie der Name vermuten lässt die Chunks beinhaltet, die erstellt werden. Diese Liste wird jedoch nicht mit Threads, sondern bei jedem Frame überprüft.

Inselform und Biome
Der Plan war zuerst, dass die Form der Insel durch eine 2D-Noisemap erstellt wird. Da wir was Noise angeht wenig Erfahrung hatten(und auch immer noch haben), scheiterten wir an diesem Vorhaben. Wir haben uns also dazu entschlossen, dass die Form der Insel immer ein Kreis ist.

Das wird umgesetzt, indem der Mittelpunkt der Welt ermittelt wird. Von dort aus wird eine Distanz erstellt, die an die Weltgröße angepasst ist. Diese Distanz ist dann der Radius für unsere Insel.
Bei der Zuteilung der Werte und Eigenschaften an jeden Voxel wird in der *GetVoxel()* Methode einem Block, der nicht innerhalb dieses Radius liegt, das Wasserbiom zugeschrieben.

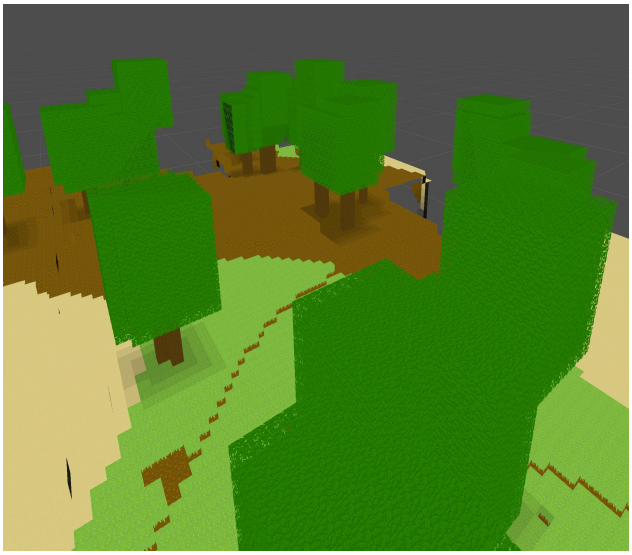
Des Weiteren sollte die Welt aus verschiedenen Biomen bestehen, welche verschiedene Blöcke und Charakteristika besitzen. Es kommt das Desert-Biom, bestehend aus Sandblöcken, das Grasslands-Biom, bestehend aus Gras- und Erdblöcken, das Forest-Biom, bestehend aus Gras- und Erdblöcken sowie das Water-Biom bestehend aus Wasserblöcken vor.
Die Welt weist auch eine hügelige Landschaft auf, die von Biom zu Biom variiert.

Die Umsetzung findet größten Teils in der *GetVoxel()* Methode statt. Hier wirkt aber auch die Klasse *BiomAttributes*, die alle Bestandteile eines Bioms aufführt. Dazu gehören unter anderem der SurfaceBlock, also der Block, der die Oberfläche bildet, sowie der SubsurfaceBlock welcher die vier darunterliegenden Schichten bildet. Da *BiomeAttributes* ein scriptable Object ist, können die Biomattribute einfach und schnell im Unity Editor verändert werden.
In *GetVoxel()* werden den verschiedenen Biomen mit Hilfe vom Methodenaufruf *Noise.Get2DPerlin(Vector2(pos.x, pos.z), offset, scale))* je ein Wert zugeteilt. Anhand dieses Wertes wird ausgewählt welches Biom gerade generiert wird.
Die Hügel werden ebenfalls mit Hilfe von 2D Perlin Noise erstellt. So wird das Attribut *terrainHeight*, das jedes Biom hat, angepasst. Damit der Übergang zwischen den Biomen nicht durch Klippen oder ähnliches unterbrochen wird, wird die Höhe der einzelnen Biome dort angepasst.

Dazu wird die Summe aller Höhen(*sumOfHeights*) berechnet und durch die Anzahl der Biome geteilt. Dann wird *terrainHeight* gerundet durch den Aufruf *terrainHeight = Mathf.FloorToInt(sumOfHeights + solidGroundHeight).* *solidGroundHeight* beschreibt die Mindesthöhe des Bioms.

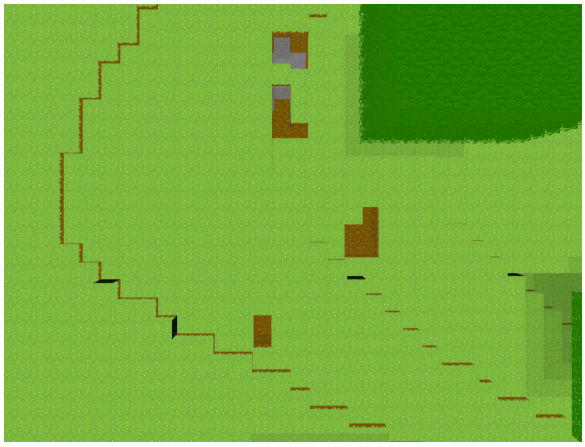
Die Inselform bei 10x10 Chunks

Fließender Übergang zwischen Biomen



Eine weitere Eigenschaft der Welt und der Biome ist sowohl das Vorkommen von „Klecksen“ (im Folgenden lodes genannt), sowie Flora. Diese lodes sind einfach Vorkommen eines anderen Materials innerhalb eines Bioms. Beispielsweise sieht man im obigen Bild mit der Inselform grüne Flecken auf dem braunen Erdboden. Hierbei handelt es sich um Grasblöcke die den *surfaceBlock* des Forest-Bioms -Erde- unterbrechen. So kommen auch Höhlen (Vorkommen von Luftblöcken) und Erzadern zustande. Die lodes werden mit Hilfe von 3D Perlin Noise erstellt und sind flexibel variierbar, da sie ein Attribut der *BiomAttribute* Objekte darstellen.
Die Flora des Bioms wird ebenfalls mit Hilfe von Perlin Noise gesetzt.

Höhleneingang (obere Hälfte des Bildes)



Interaktion mit Blöcken: Bewegung, Abbauen und Setzen

Für eine bessere Performanz wurden *MeshCollider* für unsere Voxel weggelassen. Wir haben das Kollidieren des Spielers mit den Blöcken durch einen den Raycasts ähnlichen Ansatz berechnet.

Die Methode *CheckForVoxel()* wird in der Klasse Player aufgerufen, wo sie eine Position als Argument erhält und dann *true* für einen soliden und *false* für einen nicht soliden(Luft) Voxel zurückgibt. Mit ihrer Hilfe schaut die Methode *CheckDownSpeed(),* ob unter dem Spieler ein Voxel ist. *CheckUpSpeed()* übernimmt über dem Spieler liegende Voxel und die Methoden *front, back, left* und *right* sind für den Rest verantwortlich. Die Methoden schauen sich dabei die Ecken des Colliders des Spielers in Beziehung zu den Ecken der Voxel an.

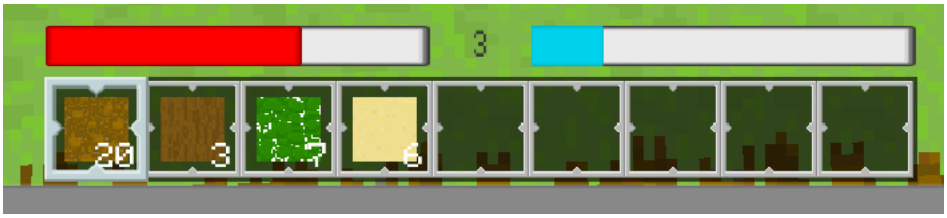
Das Abbauen und Setzen von Blöcken funktioniert mit Hilfe eines Highlight-Blockes. Ist das Fadenkreuz auf einen Voxel gerichtet, der eine bestimmte Entfernung zum Spieler hat, wird auf diesem Voxel ein Highlight-Block erstellt. Beim Setzen befindet sich dieser auf dem Voxel, für das Abbauen wird der Voxel von ihm umfasst. Dieser Highlight-Block kann unsichtbar sein, es bietet sich aber auch an ihn als leichte Umrandung darzustellen, wie man es aus Minecraft kennt.
Da jeder Chunk eine *voxelMap* (3D-Array) besitzt, in der die Informationen über alle Voxel innerhalb des jeweiligen Chunks hinterlegt sind, wird bei Links-Klick der Voxel an der Position des Highlight-Blocks in der *voxelMap* durch einen Air-Block ersetzt. Danach müssen die umgebenen Voxel aktualisiert werden, da aus Performanzgründen nur die Blockseiten geladen sind, die der Spieler auch sehen kann.
Der Chunk kommt dann in die *chunksToUpdateList,* wo dann das Prozedere, das weiter oben Beschrieben ist, abgehandelt wird.
Bei Rechts-Klick wird an der Position des Highlight-Blocks nun der Voxel in der *voxelMap* durch den im Inventar ausgewählten Block ersetzt. Das geht natürlich nur, wenn dort ein Air-Block ist.

Zu erwähnen ist hier auch, dass beim Abbauen eines Blockes an dessen Position ein Prefab des Selbigen erstellt wird, welches mittels der *AddForce()*-Methode und einem negativen Parameter zum Spieler hinfliegt und dann zerstört wird. Das erzeugt den Eindruck, dass der abgebaute Block vom Spieler angezogen wird.

Interaktion mit Blöcken: Lagerung in einem Inventar

Aufgenommene Blöcke werden in einem toolbelt mit neun Slots gelagert.
Dieser toolbelt wird durch neun *ItemFrames* und einem *HighlightFrame* auf dem Canvas dargestellt. Jeder *ItemFrame* besitzt einen ItemStack und einen *ItemSlot*. Dabei besteht ein *ItemStack* aus der BlockID(byte) und einer Anzahl, die als kleine Zahl in jedem *ItemFrame* angezeigt wird.
Wird ein Block aufgenommen, wird in der Methode *addToToolbelt()*, die die ID des aufgenommenen Blocks übergeben bekommt, geschaut, ob es bereits einen ItemSlot gibt, in dem dieser Block gelagert wird. Ist das der Fall, wird dort die Anzahl erhöht(maximal 64). Ansonsten wird der nächste freie *ItemSlot* gesucht. Das Setzen von Blöcken ist nur möglich, wenn der ausgewählte *ItemSlot*, angezeigt durch den *HighlightFrame*, eine Anzahl >0 aufweist.
Der *slotIndex* verändert die Position des *HighlightFrames* und so auch den ausgewählten Block im Inventar. Gesteuert wird mit dem Mausrad.

Toolbelt mit ausgewähltem Erdblock. Zu sehen ist auch die Healthbar und Energybar sowie Score.



Interaktion mit Blöcken: Schießen

Manche aufgenommene Blöcke können als Projektile verwendet werden. Hierbei generieren wir normale Unity Cubes als Projektile anstelle der selbstgemachten Voxel Cubes, da der Umgang mit ihnen in Bezug auf das Instanziieren und Kollidieren einfacher war.

Die Methode *shoot()* schaut sich den ausgewählten Block im toolbelt anhand seiner ID mit Hilfe des Aufrufs *toolbelt.slots[toolbelt.slotIndex].itemSlot.stack.id an*. Dort ruft ein *switch(voxelID)* für den richtigen Block die Methode *initiateAndShootBullet() auf*, die anhand der *voxelID* das richtige GameObject übergeben bekommt. Hier wird nun dem Spieler über die Energybar Energie abgezogen. Wenn genügend Energie vorhanden war, wird ein Prefab-Projektil an der Position des FirePoints in der Hand des Spielers instanziiert. Der Abschuss erfolgt dann über die *AddForce()*-Methode.
Berührt ein Projektil einen Gegner oder einen Voxel, wird es zerstört.

Zwei Erdblöcke wurden hintereinander abgefeuert. Energie wurde dementsprechend abgezogen.

Blöcke und ihre Eigenschaften

Damit das Kampfsystem Abwechslung bietet, haben verschiedene Blöcke verschiedene Eigenschaften. So macht ein Steinblock mehr Schaden als ein Erdblock und ein Eisblock friert den Gegner für eine kurze Zeit ein. Ein Feuerblock macht am meisten Grundschaden, während ein Säureblock weniger, aber dafür Schaden über Zeit macht.

In der Klasse Bullet wird jedem Block sein Schadenswert zugeteilt.
Kollidiert nun ein Eisblock mit einem Gegner, so wird die Methode *freezeEnemy()* aufgerufen, welche mit Hilfe einer IEnumerator Coroutine den Gegner für ein paar Sekunden bewegungsunfähig macht.
Kollidiert ein Säureblock mit einem Gegner wird die Methode *takeDamageContinously()* aufgerufen. Auch sie startet eine IEnumerator Coroutine, in der dem Gegner Schaden zugefügt wird, dann einen Moment gewartet wird und wieder Schaden zugefügt wird usw.

Gegner: Spawning

Wichtig für ein Kampfsystem sind natürlich auch Gegner. Diese bewegen sich halbwegs intelligent durch unsere Welt nachdem sie gespawnt sind. Ist man in ihrer Nähe, verfolgen sie den Spieler und greifen ihn an.

Die Klasse *EnemySpawner* überprüft alle drei Sekunden, ob noch ein Gegner existiert, mit Hilfe der simplen Methode *EnemyIsAlive()*. Existiert kein Gegner mehr, wird mit einem WaveSpawner eine neue Gegnerwelle gespawnt. Die Anzahl der Gegner sowie deren Stärke hängt dabei vom Spieler-Score ab. Dieser wird erhöht sobald man einen Gegner vernichtet.
In der IEnumerator Routine *SpawnWave()* und *SpawnEnemy()* wird anhand der Spielerposition der oder die Gegner 25 Voxel entfernt initialisiert. Dabei handelt es sich um ein Model aus dem Assetstore. Die *SpawnStates* SPAWNING, WAITING, und COUNTING unterstützen diese Funktionen.

Gegner: Fortbewegung

Die Fortbewegung der Gegner funktioniert in unserer Voxel-Welt ohne *MeshCollider* mit Hilfe von einem *NavMesh*. Durch die beiden Komponenten *LocalNavMeshBuilder* und *NavMeshSurface Components* wird ein GameObject erstellt, das die gesamte Welt umgibt.
Beim Erstellen eines Chunks wird dann ein *NavMeshSourceTag Component* eingefügt, der Veränderungen im Chunk beobachtet, wie zum Beispiel das fehlen eines soliden Blocks durch abbauen. Dann muss nämlich auch das *NavMesh* aktualisiert werden.
Dem Gegner-Objekt wird ein *NavMeshAgent Component* beigefügt. Dieser passt an, wie sich der Gegner bewegen muss, um zu seinem Ziel zu kommen. Dabei müssen natürlich die Hindernisse, die die Welt auftut, überwunden werden.
Dazu muss gesagt werden, dass wir die volle Funktionsweise von *NavMesh* und all seinen Komponenten nicht vollkommen verstanden haben.

Gegner: Verhalten

Wenn die Entfernung zwischen einem Gegner und dem Spieler weniger als 12 Voxel beträgt, wird mit Hilfe der Methodenaufrufe *NavMeshAgent.SetDestination()* und *ThirdPersonCharacter.move()* die Verfolgung des Spielers aufgenommen.
Liegt eine größere Distanz vor, läuft der Gegner zu einem zufälligen Ziel nach Ablauf eines Timers. Sollte nun die Distanz auf mehr als 70 Voxel steigen wird der Gegner zerstört.
Dadurch wird die Performanz nicht durch unnötige Gegner-Objekte in Mitleidenschaft gezogen.

Ist der Gegner nahe genug am Spieler, wird die Schlaganimation abgespielt. Dabei wird in bestimmten Frames der Animation geprüft, ob der Hand-Collider des Gegners mit dem Collider des Spielers kollidiert. Ist das der Fall wird ein Event ausgelöst und dem Spieler wird Schaden zugefügt.

Fazit

Da wir beide kaum Erfahrung mit Unity und der Entwicklung eines Spiels als zeitlich begrenztes Projekt hatten, haben wir uns leicht übernommen bezüglich des Einbaus einiger Features. So haben wir im Endeffekt zu viel Zeit damit verbracht die Form der Insel natürlich zu gestalten, was wir dann auch verworfen haben.
Das von uns gewünschte „realistische“ Höhlensystem einzubauen hat auch eine Menge Zeit gekostet. Wir haben uns erst über Perlin Worms informiert, diese aber nicht verstanden. Danach sind wir auf einen Minecraft-Klon namens ClassiCube gestoßen, der eine interessante und verständlichere Höhlengenerierung mit sich brachte. Leider konnten wir diese Methode zur Höhlengenerierung nicht mit unserer *GetVoxel()* Methode vereinbaren.
Weiterhin haben wir für die Shader die Implementierung von b3agz genutzt, welche noch nicht optimal ist. Aus diesem Grund(so denken wir) hat das von uns angefertigte HD-Texture Pack, welches man im Menü auswählen können sollte, Probleme bereitet.
Auch das Animieren der eigens angefertigten 3D Objekte, wie dem Arm, hat aus Gründen mangelnder Erfahrung mit Blender nicht so richtig funktioniert.

Trotzdem haben wir die meisten Features einbauen können und sind froh dieses Projekt gemacht zu haben. Die gewonnenen Erfahrungen mit Unity, Blender etc. sind uns sehr wichtig, vor allem in Anbetracht dessen, dass einer von uns den Beruf des Spielentwicklers anstrebt.
Zudem bereichern uns auch die Erfahrungen, die wir im Bereich der Projektgestaltung, Planung und Bearbeitung im Team gemacht haben.