Entwicklung eines Spiels mit Unity

Niels Niederwieser (4g)
betreut von Lukas Fässler

Maturitätsarbeit
MNG Rämibühl Zürich
3.1.2020

Inhaltsverzeichnis

1 Abstract

In dieser Maturarbeit wird die Entwicklung eines Computerspieles mit dem 3-D-Programm Unity beschrieben. Ziel war in beschränkter Zeit (ca. 8 Monate) ein Spiel selbständig zu entwickeln. Dies ist mir einigermassen gelungen, ich würde aber gerne das Spiel noch weiter verbessern.

This Matura thesis describes the development of a computer game with the 3-D program Unity. The goal was to develop a game independently in a limited time (about 8 months). I succeeded to a certain extent, but I would like to improve the game even further.

2 Vorwort

Als ich mir die Frage stellte, was ich für ein Thema für meine Maturarbeit wählen sollte, war mir sofort klar dass ich eine Informatik-Arbeit machen will. Programmieren hat mir seit langer Zeit gefallen, wie zum Beispiel mit *Scratch* (eine visuell basierte Programmiersprache) oder im Informatik Unterricht mit *Java (eine bekannte Programmiersprache)*. Ich hatte mehreren Matura-Arbeitsvorträgen zugehört, welche *Unity* benutzten, ebenso hatte ich mir kürzlich YouTube Videos über *Unity* angesehen. Ausserdem gehören Computerspiele zu meinen Hobbys unter anderem auch das Verstehen deren Mechanik, weshalb ich mich schlussendlich entschieden habe, ein Spiel mit *Unity* zu programmieren.

Dankesworte gehen an meinen Vater Marc Niederwieser und an meinen Betreuer Lukas Fässler. Zusätzlich inspirierten mich viele YouTuber insbesondere Brackeys und Blackthornprod.

3 Einleitung

Wie im Vorwort erwähnt wollte ich ein Spiel programmieren. Ich kannte viele Spiele und lies mich stark von einem Spiel namens *Galaxy on fire* inspirieren. Ich wusste, dass ich zu wenig Zeit und Wissen hatte, um ein marktfähiges Spiel zu programmieren, weshalb ich mich auf ein einfacheres Spiel konzentriert habe. Als Umgebung habe ich den Weltraum gewählt, weil der Weltraum relativ leer ist und nicht viele Objekte dargestellt werden müssen, um realistisch zu wirken. Dazu wollte ich ein Raumschiff, das durch den Spieler kontrolliert werden kann. Mir war wichtig, dass ich die volle Kontrolle über das Raumschiff ermögliche, d.h. dass man die Richtung und Geschwindigkeit verändern kann. Zusätzlich sollte man Schüsse abfeuern können, um Asteroiden zu beschiessen, um so eine freie Flugbahn zu haben. Die Asteroiden sollten sich in einem Asteroidenfeld befinden. Um den Spieler zu limitieren wollte ich eine verfügbare Energie, welche sich mit der Zeit regeneriert. Verbraucht wird diese durch Beschleunigen oder Bremsen des Raumschiffes und durch Abfeuern von Schüssen. Das Ziel des Spieles sollte sein, so viele Asteroiden wie möglich zu zerstören. Dabei habe ich mir überlegt, dass Asteroidenteile, welche keinen Effekt mehr haben, nach einer gewissen Zeit verschwinden sollten, um Rechenleistung einzusparen. Damit es nie zu wenig Asteroiden hat, kam mir die Idee Asteroiden nach und nach wieder neu zu erzeugen, aber nur wenn der Spieler nichts davon merkt.

Ein Spiel kann man von Grund auf programmieren oder man benutzt eine *Spiel-Engine*, das letztere ist heute üblich. Eine Spiel-Engine ist ein Tool (Programmierwerkzeug), welches das Programmieren erleichtert. Um dies zu verstehen, muss man als erstes wissen, wie Computerspiele aufgebaut sind. Meistens bestehen Spiele aus mehreren Komponenten, wie zum Beispiel einem Teil, der alles Visuelle zeichnet, einem der Physikberechnungen wie Kollisionen macht und einem weiteren, durch den spezifische Spielmechaniken definiert werden. Eine Spiel-Engine kümmert sich vor allem um den visuellen Teil und im Fall von *Unity* auch um Physikberechnungen.

Nun musste ich mich für eine Spiel-Engine entscheiden. Es gibt viele anfängerfreundliche, darunter die beiden bekanntesten: *Unity* und *Unreal Engine*. Ich entschied mich für *Unity*, weil ich viel positives darüber gehört hatte.

Unity ist sehr benutzerfreundlich und für nicht kommerzielle Zwecke auch kostenlos. Ausserdem kann man es für 2D, aber auch für 3D Spiele benutzen.

4 Vorgehen und Planung

Ich wusste nicht viel über *Unity*, weshalb ich als erstes eines der eingebauten Tutorials durcharbeitete. Leider waren damals diese Tutorials ziemlich oberflächlich, darum habe ich auf YouTube nach einem Einsteiger-Tutorial gesucht. Schlussendlich bin ich auf ein Tutorial von YouTuber Brackeys gestossen[1]. Es ist sehr anfängerfreundlich und erklärt gleichzeitig viele nützliche Dinge. Es dauert insgesamt 100 Minuten und um es zu absolvieren und zu

1 Brackeys, 2017: *How to make a Video Game in Unity*

verstehen habe ich ungefähr vier Stunden gebraucht. Danach habe ich mit der Programmierung meines Spieles gestartet. Ich fing ganz begeistert an zu programmieren und versuchte meine Ideen kreativ zu verwirklichen. Manche Ideen konnte ich jedoch nicht verwirklichen, weil sie zu schwierig waren oder weil ich nicht so vielen Ideen gleichzeitig nachgehen konnte.

5 Programmieren und Entwicklung des Spieles

In Unity wird alles mit Objekten, den sogenannten *GameObjects* berechnet. Dabei ist jeder Gegenstand ein *GameObject*. Diese haben Komponenten, darunter als Beispiel die *Transform* Komponente für die Position, Rotation und Skalierung. Zusätzlich kann man *Skripte* schreiben, um mehr zu definieren. Ein *Skript* ist ein kleines Programm. Ein Programm besteht aus verschiedenen Befehlen, ein wichtiges Element ist eine Variable. Dabei kann eine Variable verschiedene Werte speichern abhängig vom Variabel-Typ. Ich habe unter anderem folgende Variabel-Typen gebraucht: *Integer* (eine ganze Zahl zwischen -2^{31} und 2^{31}), *Float* (eine Gleitkommazahl) und *Boolean* (zwei Werte *true* oder *false*). In den nächsten Unterkapiteln stelle ich die wichtigsten Objekte vor, in der Reihenfolge in der ich sie programmiert habe.

Ein solches Objekt hat in den meisten Fällen ein Bild (2D) oder ein 3D-Modell (3D). Ein 3D-Modell ist ist ein Objekt, welches eine 3D-Figur festlegt. Die am häufigsten genutzte Variante ist durch *Vertices* (Ecken), *Edges* (Kanten) und *Faces* (Flächen) definiert. Dabei ist ein *Vertex (Einzahl von Vertices)* ein Punkt in einem dreidimensionalen Koordinatensystem, eine *Edge* eine Verbindung von 2 *Vertices* und ein *Face* aus mehreren zusammengehängten *Edges*, welche am Ende mit dem Ursprungspunkt verbunden sind, im einfachstem Falle eine Dreiecksfläche. Man bezeichnet dieses Konstrukt auch als Geometrie. Je mehr *Vertices* ein Modell hat, desto rechenintensiver ist es. Man spricht auch von der Geometriedichte. Ein solches 3D-Modell ist leicht mit dem Gratisprogramm *Blender* herstellbar.

5.1 Raumschiff

Für mein Raumschiff habe ich zuerst ein Skript geschrieben, welches sich *Movement* nennt. Es ist verantwortlich für die Bewegung des Raumschiffes. Damit sich das Raumschiff realistisch verhält, habe ich das eingebaute Skript namens *Rigidbody*[2] benutzt, welches sich um Physik spezifische Sachen wie Kollisionen kümmert. Dabei kann man die Geschwindigkeit eines Objektes entweder durch Kräfte verändern oder direkt. Mit dem Befehl *AddTorque()* kann man eine Drehkraft auf das Objekt einwirken lassen. Dies habe ich benutzt, um die Richtung des Raumschiffes zu ändern. Mit dem Befehl *AddForce()* kann man einen Kraftvektor auf das Objekt einwirken lassen. Dies habe ich für die Geschwindigkeit des Raumschiffes benutzt.

2 Rigidbody = ein Skript, das ein Objekt als starren Körper simuliert

Mit einem weiterem Skript namens *Variables* werden verschiedene Attribute des Raumschiffes festgelegt, wie z.B. das Leben des Raumschiffes. Zusätzlich kümmert sich dieses Skript um die Anzahl Punkte, welche durch das Zerstören von Asteroiden erhöht wird.

Ausserdem wollte ich mit meinem Raumschiff schiessen können. Dafür habe ich auch ein Skript geschrieben, welches unter dem Kapitel Schuss zu finden ist.

Danach habe ich mich um das 3D-Modell des Raumschiffes gekümmert, welches ich mit dem kostenlosen Programm *Blender* erstellt habe.

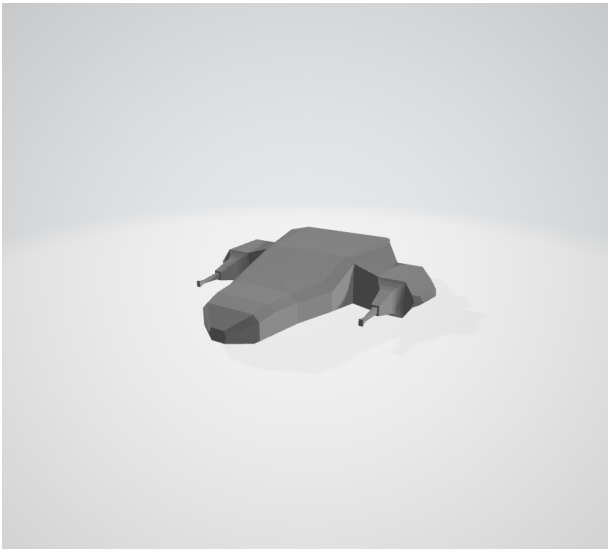

Bild 1: 3D-Modell Raumschiff

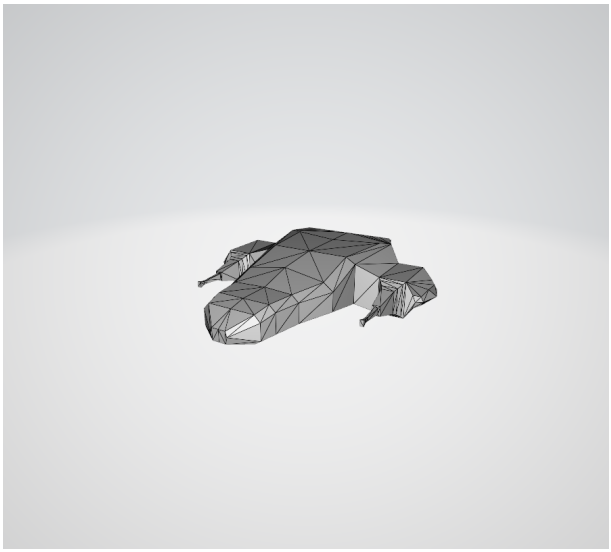

Bild 2: 3D-Modell Raumschiff Wireframe

5.2 Kamera

In meiner ersten Version sah man das Raumschiff während es flog immer konstant in der Mitte des Bildschirmes und bei Richtungsänderungen bewegte sich der Weltraum darum herum.

Dies überzeugte mich nicht, weil man so die Bewegungen des Raumschiffes bei Richtungsänderungen nicht sehen konnte. Deshalb baute ich mit einem Skript eine schwache verzögerte Verfolgung der Kamera ein und nun sieht die Ansicht viel realistischer aus. Dafür benötigte ich ein Skript, bei dem die Kamera dem Spieler mit einer schwachen Verzögerung folgt. Zuerst hatte ich keine Ahnung, wie so etwas gemacht wird. Zum Glück fand ich Hilfe im Internet vor allem durch das Video *Smooth FollowCam*[3] auf Youtube.

Dabei wird zuerst die Rotation des Raumschiffes mit einem Vektor multipliziert. Der Vektor dient dazu, den Standardabstand zwischen der Kamera und dem Raumschiff zu definieren. Dieser wird mit der Rotation multipliziert, um die richtige Richtung des Vektors zu berechnen. Dabei musste ich beachten, dass man mit *Unity* nur eine Rotation mit einem Vektor

3 BurgZerg Arcade (2016): *Smooth FollowCam*

multiplizieren kann, aber nicht umgekehrt. Danach muss man die Position des Raumschiffes dazu addieren, um das Ziel zu bekommen. Mit einer eingebauten Funktion namens *Slerp*, welche zwei Punkte interpolieren kann, lässt sich leicht ein Zwischenpunkt berechnen, um eine Verzögerung zu visualisieren. Mit der Funktion *LookRotation* kann ich leicht eine Ziel-Rotation berechnen und wieder wie vorhin einen Zwischenwert berechnen.

Zusätzlich wollte ich einen Weltraum-Hintergrund haben. Dafür habe ich eine sogenannte *Skybox* benutzt. Man kann sich dies als Würfel vorstellen, bei welchem man eine Textur den sechs Innenseiten zuordnen kann. Schliesslich habe ich mit einem Bildbearbeitungsprogramm ein schwarzes Bild mit weissen Punkten, welche die Sterne darstellen sollen, erstellt. Dieses Bild habe ich dann für alle sechs Seiten benutzt.

5.3 Schuss

Damit mein Raumschiff schiessen kann musste ich zunächst einen einzelnen Schuss programmieren. Dazu habe ich das Skript *ShotManager* erstellt, das sich um das Abfeuern von Schüssen vom Raumschiff aus kümmert und ein weiteres Skript namens *Shot*, welches das Projektil als Objekt verwaltet.

Mit dem Befehl *Instantiate()* kann man sogenannte *prefabs* erzeugen.

Ein *prefab* ist ein *GameObject* welches man mehrfach benutzen kann, aber jedes hat das *prefab* als Ursprung. Auf den Schuss bezogen kann ich diesen zu einem *prefab* deklarieren. Dadurch kann ich einen Schuss erzeugen und wenn ich daran etwas ändern möchte, so wird diese direkt bei allen Verwendungen angewendet.

Im Skript *Shot* sind verschiedene Attribute definiert, wie die Lebenszeit eines Schusses. Die Lebenszeit legt fest, wie lange ein Schuss existiert. Man könnte sich fragen, wieso ich eine Lebenszeit einbaue, wenn ich den Schuss entfernen kann, sobald er mit etwas kollidiert. Der Grund dafür ist, dass ein Schuss in den schwarzen Raum abgefeuert und dann weit weg fliegen könnte und dann nur noch unnötige Rechenleistung verbrauchen würde. Mit dem Befehl *Destroy(gameObject, Livetime)* kann ich den Schuss, nachdem die Lebenszeit abgelaufen ist, zerstören. Mit dem Ausdruck «*gameObject»* ist das Objekt auf dem sich das Skript referenziert (in diesem falle der Schuss) und *Livetime* ist die Variable, welche die Lebenszeit definiert.

Zusätzlich zur Lebenszeit sind folgende Variablen definiert:*SummonVelocity*, *ShotSpeed, Damage*, *DamagePlayer*. *SummonVelocity* ist die Geschwindigkeit des Erzeugers, welche dem Schuss beim Erzeugen gesetzt werden muss. *ShotSpeed* ist ein Wert, welcher zur Geschwindigkeit dazu gezählt wird. *Damage* ist eine Variable, welche vom Erzeuger gesetzt werden muss und bei einer Kollision abgerufen wird. Sie dient dazu dem Schuss einen Schadenswert zuzuordnen, welcher später dem Empfänger beim Einschlag von seinen Lebenspunkten abgezogen wird. *DamagePlayer* ist ein sogenannter *Boolean*. Dies ist ein Variablen-Typ, welcher nur zwei Werte [true, false] speichern kann. In diesem Fall benutzte ich diese Variable, um zu merken, ob der Schuss vom Raumschiff schon abgefeuert worden ist oder nicht, da ansonsten, weil der Schuss beim Schiessen das Raumschiff selber direkt berührt, es als Treffer mit dem Raumschiff angesehen würde.

Für meinen Schuss wollte ich eine Kugel als 3D-Modell. Es gibt verschiedene Methoden, um Kugeln durch 3D-Modelle darzustellen. Die häufigsten zwei nennen sich *UV-sphere* und *Icosphere*.

Eine *UV-sphere* gleicht dem Koordinatensystem der Erde. Dabei wird eine Kugel aus horizontalen und vertikalen Kreisen erstellt (siehe Bild).
Eine *UV-sphere* ist sehr praktisch, um einer Kugel eine Textur zu geben, da man die Geometrie leicht zu einem Rechteck auffalten kann. Eine rechteckige Weltkarte ist somit die Textur der Erde.

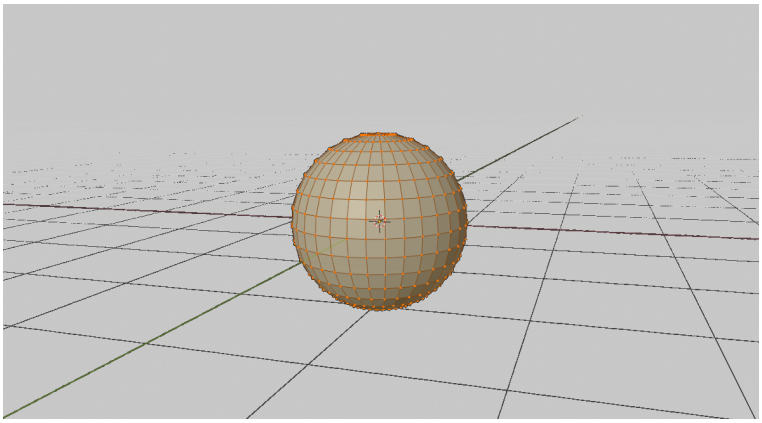

Bild 3: UV-spehre Wireframe

Bei einer *Icosphere* besteht das Modell aus gleichgrossen Dreiecken (siehe Bild).
Der Vorteil gegenüber der *UV-sphere* ist, dass die Dichte der Geometrie an jeder Stelle gleich gross ist.

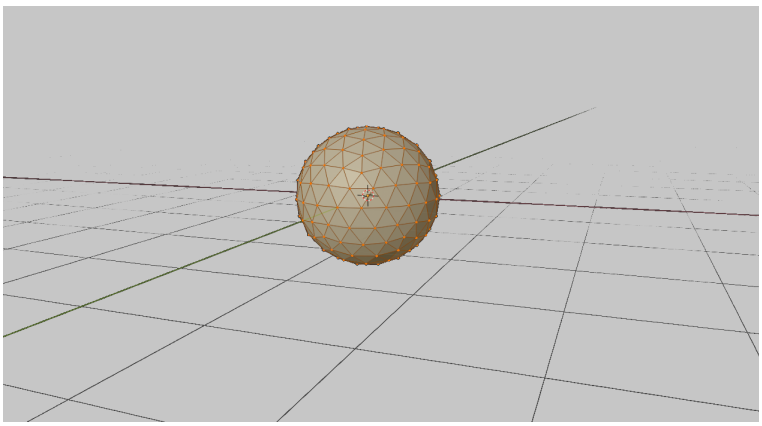

Bild 4: Icospehere Wireframe

Für das 3D-Modell des Schusses habe ich mich für eine UV-sphere entschieden, weil es vor allem wichtig ist, dass er rund wirkt und nicht, dass die Geometriedichte gleich sein muss.

5.4 KeyManager

Ich erstellte ein Skript bei dem alle Tastatur- und Maus-Eingaben eindeutig Funktionen zugeordnet werden. Somit kann ich, wenn ich eine Taste ändern will, dies zentral an einer Stelle bewerkstelligen. Da dieses Skript nicht an ein *GameObject* gekoppelt ist, musste ich beachten, dass ich die standard-generierte Funktion *MonoBehaviour* entfernte. Mit dem Befehl *Input.GetKey("w")* kann man testen, ob ein Buchstabe in diesem Fall «w» gedrückt ist. Mit dem Befehl *Input.GetMouseButtonDown(0)* kann man testen, ob eine Maustaste gedrückt ist (0 steht für die linke, 1 für die rechte und 2 für die mittlere Taste).

Folgende Tastenbelegung für die Steuerung des Raumschiffes habe ich benutzt:

Nach oben steuern:	«s»
Nach unten steuern:	«w»
Nach rechts steuern:	«d»
Nach links steuern:	«a»
Schiessen:	«Maus linksklick»
Beschleunigen/bremsen:	«Mausrad»
Beenden:	«Esc»

5.5 Asteroid

Für den Asteroiden brauchte ich ein Skript, um ihn zerstörbar zu machen. Dafür habe ich eine Variable namens *health* definiert, welche die Lebenspunkte symbolisiert. Sobald der Wert 0 erreicht wird, wird der Asteroid zerstört. Um das Auseinanderbrechen des Asteroiden zu visualisieren, muss zuerst das Asteroid-Objekt mit den zusammen gesetzten Bruchstücken an der gleichen Position und mit der gleichen Rotation erzeugt werden und dann das vorherige Asteroid-Objekt mit dem Befehl *Destroy(gameObject)* entfernt werden. Ausserdem wird die Punktanzahl erhöht.

Zusätzlich brauche ich einen Kollisionstester, welcher bei einer Kollision mit einem Schuss das Leben reduziert und bei einer Kollision mit dem Raumschiff sie auf 0 setzt. Die Funktion *OnCollisionEnter()* wird aufgerufen, sobald das Objekt mit etwas kollidiert. Danach kann ich überprüfen, ob es mit dem Raumschiff oder mit einem Schuss kollidiert ist.

Für mein Asteroiden-3D-Modell habe ich als Basis eine *Icosphere* genommen. Um einen länglichen Asteroiden zu machen, habe ich es an einer Achse skaliert. Durch verschiedene Werkzeuge im *Sculptmodus* (Ein Modus in *Blender*) habe ich die finale Form modelliert. Ich wollte meinen Asteroiden aber auch während des Spiels zerstören können, weshalb ich ihn mit einem *Addon* (*Cell Fracture*) für *Blender* in kleine Teile geteilt habe. Um das ganze und das zerstörtes Asteroiden-Modell nun zu Unity zu exportieren, musste ich darauf achten, dass ich die zerstörte Version mit allen Teilen exportierte.

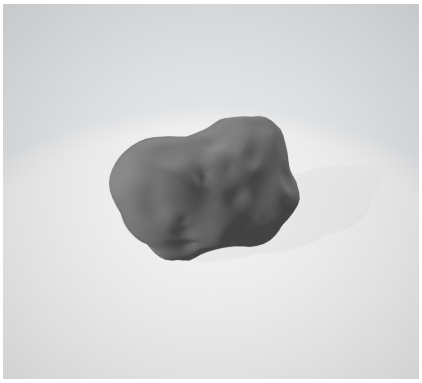

Bild 5: 3D-Modell Asteroid

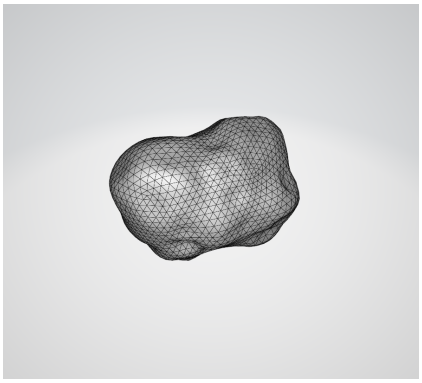

Bild 6: 3D-Modell Asteroid Wireframe

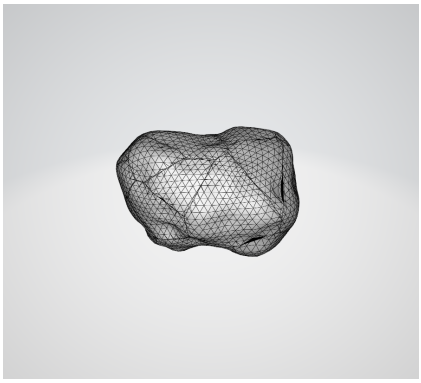

Bild 7: 3D-Modell kaputter Asteroid Wireframe

Später habe ich ein sogenanntes *Level of Detail (LOD)* System eingebaut. Das Idee eines LOD ist, dass man nicht so viel Detail darstellen muss, wenn man etwas von weit weg anschaut. Es erhöhte die Performance stark, als ich dies auf das 3D-Modell anwendete.

5.6 Asteroidenfeld

Ich wollte ein Asteroidenfeld kreieren und musste mir zuerst überlegen wie. Wenn ich die Positionen von Asteroiden einfach mit einem simplen Zufallsfaktor erstelle, so werden diese einfach gleichmässig im gesamten Raum verteilt, was überhaupt nicht einem kompakten Asteroidenfeld entspricht.

Meine Idee war die Positionen (drei Dimensionen) von mehreren Asteroiden mit Hilfe der Gaussschen Normalverteilung zu erzeugen und damit zu steuern, dass die Asteroiden alle in der Nähe eines Mittelpunktes des Feldes zu liegen kommen.

Durch langes Nachforschen im Internet habe ich eine Formel der Normalverteilung programmieren können, aber ich bemerkte erst spät, dass mir die Formel nicht viel bringt. Ich konnte zwar die Wahrscheinlichkeit an einem Punkt berechnen, aber ich hatte keine Zufallszahl, welche normalverteilt war. Irgendwann bin ich auf eine nützliche Website[4] gestossen, die mir die Polar-Methode von George Marsaglia erklärte.

4 Zucconi, Alan (2015): *How to generate Gaussian distributed numbers*

Wikipedia Polar-Methode:

Man geht von zufälligen Punkten in der Ebene aus, die im Einheitskreis gleichverteilt sind. Aus deren Koordinaten werden jeweils zwei standardnormalverteilte Zufallszahlen erzeugt:

1. Erzeuge zwei voneinander unabhängige, gleichverteilte Zufallszahlen u, v im Intervall $[-1,1]$
2. Berechne $q = u^2 + v^2$. Falls $q = 0$ oder $q \geq 1$, gehe zurück zu Schritt 1.
3. Berechne $p = \sqrt{\frac{-2 \cdot \ln(q)}{q}}$
4. $x_1 = v \cdot p$ und $x_2 = v \cdot p$ sind nun zwei voneinander unabhängige, standardnormalverteilte Zufallszahlen.

Der Punkt u, v muss im Einheitskreis liegen $q < 1$, und es muss $q > 0$ gelten, da in den reellen Zahlen der Logarithmus von Null und die Division durch Null nicht definiert sind. Anderenfalls müssen zwei neue Zahlen u und v erzeugt werden.

Durch lineare Transformation lassen sich hieraus beliebige normalverteilte Zufallszahlen erzeugen: Die generierten Werte x_i sind $N(0,1)$ -verteilt, somit liefert $a \cdot x_i + b$ Werte, die $N(b, a^2)$ -verteilt sind.

Nachdem ich diese Methode implementiert hatte, generierte ich die Grösse des Asteroiden (*Scale*) auch mit einer Normalverteilung. Für die Rotation habe ich eine simple Zufallszahl zwischen 0 und 359 für jede Achse genommen. Jetzt konnte ich eine bestimmte Anzahl Asteroiden normalverteilt erzeugen, um eine realistisches Asteroidenfeld nachzubilden.

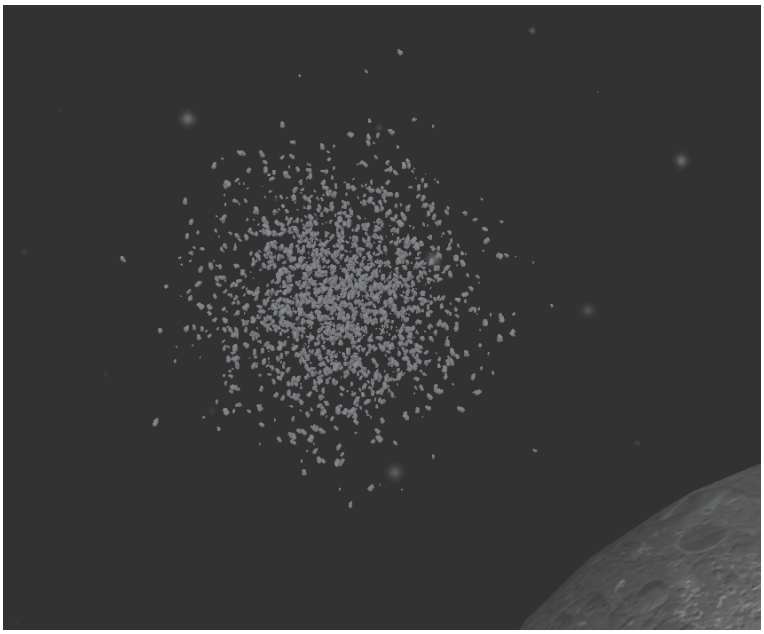

Bild 8: Asteoridenfeld

5.7 Planeten

Zusätzlich wollte ich, dass einige Planeten sichtbar sind, um den Weltraum spannender wirken zu lassen. Als erstes habe ich simple Kugeln benutzt. Dabei habe ich eine Icosphere als 3D-Modell benutzt, weil die Geometriedichte an allen Stellen eines Planeten gleich sein sollte. Zu Beginn hatten meine Planeten nur eine eintönige Farbe.

Durch das Video *Create A Realistic Moon*[5] bin ich auf die Idee gekommen den realen Mond in mein Spiel als Dekoration einzubauen. Leider hatte ich Probleme dies in *Unity* zu importieren.

Daher habe ich das im Video benutzte Bild von der *Website von Ernie Wright*[6] mit einem Bildbearbeitungsprogramm in zwei Teile geteilt und der Import der zwei Teile hat dann erfolgreich funktioniert. Ich verstehe leider immer noch nicht, warum es so funktioniert und anders aber nicht.

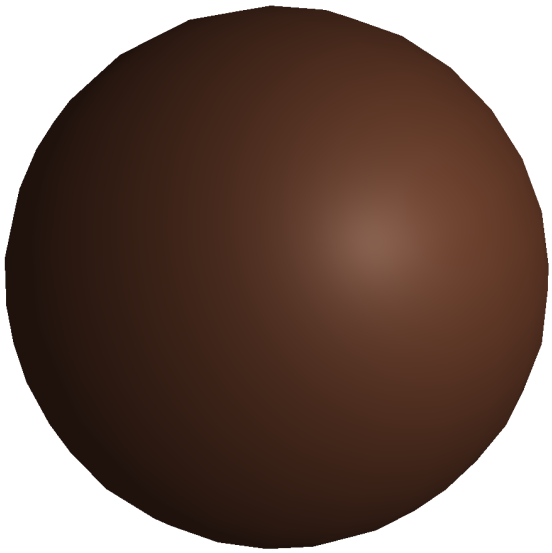

Bild 9: Planet einfarbig

Bild 10: Planet mit Mond Textur

5 CGMatter (2019): *Create A Realistic Moon*
6 Ernie Wright (2019): *CGI Moon Ki*

5.8 UI (U*ser Interface*)

UI (Abkürzung für U*ser Interface*) steht für den Text und ähnliche Dinge. In meinem Spiel gehören die verschiedenen Balken [Lebens, Energie, Schutzschild] und die Punkte dazu. Dabei zeigt der rote Balken das Leben, der hellblaue die Energie und der dunkelblaue das Schutzschild in an (siehe Bild). Für die Balken habe ich einem Tutorial benutzt[7]. Die Punkte werden durch eine simple Zahl in orange angezeigt.

Bild 11: Punkt Anzahl

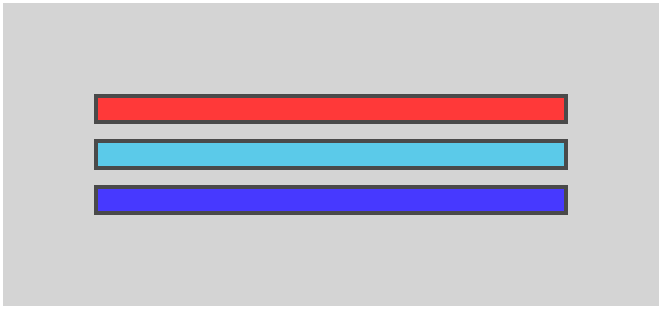

Bild 12: Balken

Ursprünglich wollte ich ein Start-Menü und ein Pause-Menü einbauen, aber ich bin leider nicht mehr dazugekommen. Wahrscheinlich werde ich es später implementieren.

5.9 Verwaltung und Struktur des Sourcecodes

Um mein Spiel zu speichern habe ich *GitHub* benutzt. *GitHub (github.com)* ist eine Website für Sourcecode. Ein Sourcecode ist der für den Menschen lesbare Teil eines Programms. Ausserdem kann man bei *GitHub* leicht ein *Repository* einrichten. Mit einem *Repository* kann man mehrere Versionen eines Ordners speicher. Dies ist sehr hilfreich mit GitHub, da man so den ganzen Entwicklungsprozess speichern kann. Dadurch konnte ich zu jeder Zeit jede Version meines Spieles herunterladen und somit auf verschiedenen Computer arbeiten.

Zusätzlich musste ich mir überlegen wie ich meine Ordnerstruktur erstelle. *Unity* generiert automatisch einen *Assets* Ordner indem ich alle Files gespeichert habe. Dabei habe ich es nach den Typen geordnet z.B 3D-Modells, Scenes und Prefabs.

Man kann sein GitHub Projekt privat oder öffentlich setzten. Ich habe es öffentlich gesetzt d.h. jeder kann alles anschauen und downloaden. Mehr Infos siehe Kapitel Ergebnisse.

5.10 Erstellen eines ausführbaren Programms

Bisher konnte ich mein Spiel im Unityeditor ausführen. Um es ausserhalb von Unity zu spielen, muss man einen sogenannten *Build* kreiren. Im Prinzip ist das ein ausführbares Programm wie man es im Alltag kennt. Dazu muss man im Unityeditor im Menü *Build Settings* aussuchen, welche *Scenes* man haben möchte.

7 Code Monkey (2018): *How to make a Health Bar*

Eine *Scene* ist ein Zusammenschluss von *GameObjects*. Dies ist vor allem wichtig für Spiele mit verschiedenen Levels, bei denen jede Level eine eigene *Scene* hat. Ausserdem ist es praktisch das Start-Menü als einzelne *Scene* zu haben. Wenn die *Start-Menü-Scene* geladen ist, werden nur deren Objekte geladen. Wenn alle anderen Objekte in der gleichen *Scene* hätte, würden Rechenleistungen für diese gebraucht. In meinem Fall hatte ich noch kein Start-Menü, weshalb ich nur eine Szene gebraucht habe.

Danach kann man auswählen für was für ein Betriebssystem es exportiert werden soll. *Unity* hat sehr viele unterstützte Betriebssysteme, aber ich brauche nur Windows, Linux und Mac OS X, da mein Spiel für Computer optimiert ist. Dann hat man einen Ordner, der alle nötigen Dateien beinhaltet. Durch ausführen des *.exe* Programms führt man es aus. Viel geholfen hat mir ein Video von Brackeys[8].

8 Brackeys (2017): *How to BUILD / EXPORT your Game in Unity*

5.11 Protokoll

Datum	Zeitaufwand [Min.]	Tätigkeit
2019-03-16	240	Brackeys Tutorial
2019-04-01	180	3D-Modell Raumschiff, Bewegungs-Controller vom Raumschiff,Schuss Skript[während 3 Wochen]
2019-04-10	180	Verzögerte Kamera folgen des Spielers
2019-04-26	30	Schuss Abgabe Klonen (Instantiate)
2019-04-26	60	Einführung und Speicherung des Projektes auf GitHub mit Vater
2019-04-29	60	Schuss an zwei Orten des Raumschiffes erzeugen
2019-05-02	30	3D Modell Space Station
2019-05-02	15	Schuss nur nach einer Zeit abfeuerbar
2019-05-03	60	Kurz anhaltender Boost
2019-05-10	60	Problem 1 versucht durch Google zu lösen
2019-05-12	60	Problem 1 vereinfacht probiert zu lösen
2019-05-12	30	Mit slomo Kamera versucht zu verstehen
2019-05-27	30	Problem 1 bei Unity Forum beschrieben
2019-06-14	30	Problem 1 gelöst und Forum Eintrag aktualisiert
2019-07-04	60	Spielplanung auf Blattpapier
2019-07-16	30	Lebenszeit einer Kugel
2019-07-24	30	Variablen des Spielers
2019-07-26	60	Menü angefangen
2019-07-27	60	Ostions Menü angefangen
2019-07-28	15	Raumschiffvariablen Energie/Max
2019-07-28	15	Variablen public -> [SerializeField] private
2019-07-31	45	Regeneration von Energie + Energie zu Schild
2019-08-04	15	Scrollwheel Input
2019-08-04	60	Scrollwheel kontrollierbare Geschwindigkeit des Schiffes
2019-08-05	30	Tasteninput neu geordnet
2019-08-06	15	Elementen Materials gegeben (=> Farbe)
2019-08-07	60	Kontrollierbare Geschwindigkeit optimiert
2019-08-08	60	Abzug von Schiffsenergie bei Beschleunigung / Schildenergie Erhöhung / Schuss
2019-08-10	60	Lebens- usw. Leiste angefangen
2019-08-11	60	Leiste fertig gemacht
2019-08-12	90	Asteroid 3D Modell gemacht
2019-08-12	45	Asteroid Bruchstücke Teile mit Blender Addon Cell Fracture

2019-08-13	45	Asteroid + Bruchstück-Version in Unity importieren
2019-08-28	30	Asteroid Leben
2019-09-01	30	Asteroid zerstören
2019-09-26	30	Gausssche Formel gesucht
2019-09-26	30	Gausssche Formel Bedeutungen wie „mean“ versucht zu lernen
2019-09-26	30	Gausssche Formel implementiert
2019-09-27	15	Bemerken, dass Gauss Formel nicht normalverteilte Zufallsz. ergibt
2019-09-27	60	Gausssche Zufallszahl recherchiert
2019-09-27	30	Normalverteilte Zahlen mit der Polar-Methode von George Marsaglia implementiert
2019-09-27	30	Asteroidenspawner programmiert (noch verbesserungswürdig)
2019-10-08	60	Asteroidengrösse zufällig gemacht.
2019-10-25	420	Versuch Spaceship controler zu verbessern [während 2 Wochen]
2019-11-24	60	Mond in Blender gemacht
2019-11-30	60	Versuch den Mond in Unity zu implementieren
2019-11-30	60	Masterarbeit Dokumentation Teil Struktur
2019-12-01	60	Versuch ein cube mit Texturen von Blender nach Unity zu exp.
2019-12-01	30	Vorwort Maturarbeit
2019-12-02	30	Versuch ein Material von Blender nach Unity zu exportieren
2019-12-02	30	Einleitung 3D-Modells + Asteroidmodell Maturarbeit
2019-12-04	60	Geschützturm der den Spieler treffen soll angefangen
2019-12-04	30	Einleitung UV-sphere + Icosphere Maturarbeit
2019-12-05	30	Geschützturm Algoritmus um Spieler zu treffen

6 Ergebnisse

Es ist mir gelungen in dieser begrenzten Zeit ein ansprechendes Spiel zu entwickeln.

Das Spiel Silent Space kann über folgenden Link ausprobiert werden:
https://github.com/nniederw/MaturarbeitUnitySpiel/releases
Dazu muss man das *.zip File* downloaden und dann entpacken. Und schon kann man *SilentSpace.exe* ausführen!

Einige Personen, z.B. meine Brüder, meine Eltern und Kollegen und Kolleginnen durften mein Spiel ausprobieren. Die Rückmeldungen waren für mich nicht überraschend. Allen machte das Spielen Spass und sie fanden den Anblick des Asteroidenfeldes sehr gelungen. Als Kritikpunkte wurde am häufigsten erwähnt, das das Spiel noch kein richtiges Ziel hatte, woran ich noch am arbeiten bin. Ausserdem wurde angemerkt, dass das Beschleunigen mit dem Mausrad nicht so praktisch ist.

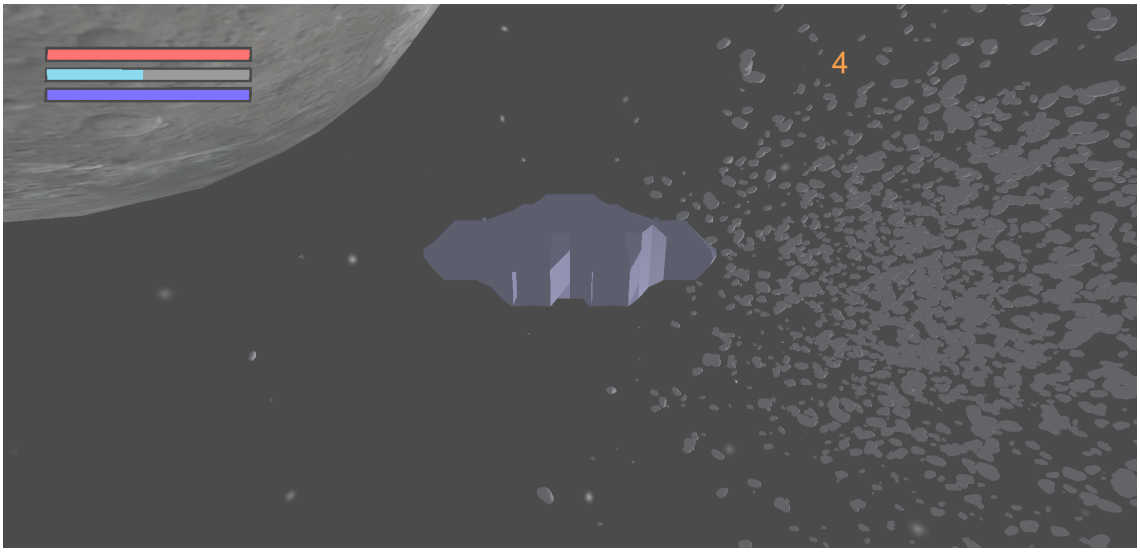

Bild 13: Foto während des Spielverlaufs

7 Reflexion

Durch Beschäftigung meiner Maturitätsarbeit habe von Grund auf *Unity* kennen gelernt. Wie von mir erhofft, eignet sich *Unity* sehr gut für Anfänger, da man viele Erklärungen auf dem Internet findet. Anfangs nutze ich vor allem YouTube-Tutorials, mit der Zeit habe ich aber mehr die offizielle, von *Unity* bereitgestellte Dokumentation benutzt und ich kann das auch so weiter empfehlen. Neben vielen kleineren Problemen, die ich rasch lösen konnte, beschäftigte

und ärgerte mich vor allem ein grosses. Bei höherem Tempo begann mein Raumschiff zu zittern. Ich versuchte das Problem einzugrenzen. Dafür habe ich zunächst ein neues Projekt gemacht mit einem Würfel. Auch hier zeigte sich das gleiche Problem. Dann habe ich mich mit einem Eintrag an das offizielle *Unity* Forum gewendet. Leider half mir dies nicht weiter. Irgendwann habe ich die Einstellungen in dem eingebautem Skript *Rigidbody* durchgeschaut und habe im Menü *Interpolate* den Punkt *Interpolate* statt *None* ausgewählt. Dies hat das Problem auf einfache Weise gelöst. Ich war sehr erleichtert darüber. Die Lösung dieses Problems hat mich mindestens 5 Stunden Arbeitszeit gekostet. Soweit ich es verstehe berechnet *Unity* Bewegungen in grossen Schritten um wahrscheinlich Rechenleistung einzusparen, aber wenn man die Interpolation anschaltet werden Zwischenwerte berechnet um es flüssiger zu machen.

Am aufwändigsten war für mich der Versuch den Controller des Raumschiffes zu verbessern. Das Raumschiff hat eine Variable, die seine Geschwindigkeit definiert. Die Geschwindigkeit der *Rigidbody* wird immer auf diese Variable des Raumschiffes gesetzt. Dadurch funktioniert die eingebaute Physik-Engine nicht richtig. Das merkt man z.B. wenn das Raumschiff mit etwas kollidiert. Es wird nicht abgebremst, sondern fliegt in eine andere Richtung mit derselben Geschwindigkeit weiter. Ich habe 2 Wochen intensiv daran gearbeitet dieses Problem zu lösen und vieles ausprobiert. Doch leider ist es mir nicht gelungen und ich habe es schlussendlich aufgegeben.

Eigentlich wollte ich noch verschiedene andere Ideen verfolgen, wie z.B. eine Raumstation, oder die Bewegung des Raumschiffes verbessern, doch leider funktionierte nicht alles sofort oder ich müsste viel mehr Zeit dafür investieren.

Wenn ich jetzt zurück denke, würde ich heute vieles anders angehen. Das liegt vor allem daran, dass ich jetzt viel mehr Erfahrung besitze. Meine Programmierfertigkeit hat sich stark erhöht, weshalb ich jetzt anders programmiere als zu Beginn. Ich werde in Zukunft mehr objektorientiert arbeiten und mit mehr Funktionen.

Ausserdem habe ich gelernt, dass die Namen der Skripte immer mit Grossbuchstaben beginnen sollten. Dies hatte ich nicht immer berücksichtigt.

8 Literaturverzeichnis

- Landolt, Robin (2019): «Unity» als Spiel-Engine für ein zug-basiertes Role Playing Game. Maturitätsarbeit MNG Rämibühl, Zürich.

8.1 Internetquellen

- Brackeys (2017): How to make a Video Game in Unity: https://www.youtube.com/playlist?list=PLZiMaaLtfkR7zz9qebmvGMNHAbZkUKYpR
- Brackeys (2017): SHATTER / DESTRUCTION in Unity: https://www.youtube.com/watch?v=EgNV0PWVaS8&
-
- Brackeys (2017): How to BUILD / EXPORT your Game in Unity: https://www.youtube.com/watch?v=7nxKAtxGSn8&
- BurgZerg Arcade (2016): Smooth FollowCam: https://www.youtube.com/watch?v=pNPuMMR5cSk
- Code Monkey (2018): How to make a Health Bar: https://www.youtube.com/watch?v=Gtw7VyuMdDc
- CGMatter (2019): Create A Realistic Moon: https://www.youtube.com/watch?v=UAOy42TR_Rk
- Ernie Wright (2019): CGI Moon Kit: https://svs.gsfc.nasa.gov/4720
- Zucconi, Alan (2015): How to generate Gaussian distributed numbers: https://www.alanzucconi.com/2015/09/16/how-to-sample-from-a-gaussian-distribution/

9 Bildverzeichnis

10 Anhang

Auf dem beigelegten USB-Stick sind alle wichtigen Daten aufgeführt.
Inhalt des USB-Sticks:

-README_(Anleitung).txt	[Anleitung zu den Files]
-SilentSpace.exe	[Spiel]
-Skripts.odt	[Alle Skripts]
-Links	[Alle Links]
-SilentSpace	[Projektordner (kann man mit Unity öffnen)]

11 Eigenständigkeitserklärung

Der Unterzeichnende bestätigt mit Unterschrift, dass die Arbeit selbständig verfasst und in schriftliche Form gebracht worden ist, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind.

Pfaffhausen, 3.1.2020 _____