

GENERELLE STRUKTUR

Die Basisklasse jedes Objektes, das im 3D Raum existiert ist `Entity`. `Entity` hat eine Transformations-Matrix, die über diverse Funktionen manipuliert und gelesen werden kann und kann einen Pointer auf ein Parent-Entity und mehrere Pointer auf Child-Entities haben. Alles, was nicht als Objekt gerendert werden muss (Lichtquellen, Kamera, etc.) erbt von `Entity` direkt, alles was eine Geometrie hat, die gerendert werden soll, erbt von `Object`, einer Unterklasse von `Entity`, die Geometrie-Daten speichert und dementsprechende Funktionen anbietet. In der main Funktion werden Strecke und Raumschiff des Spielers, sowie mehrere NPCs erstellt und eine Instanz der `Game` Klasse bekommt eine Liste aller erzeugten Entities, die kein Parent Entity haben. Die GameLoop ist noch in der main Funktion und wird nicht ins `Game` Objekt ausgelagert. Das `Game` Objekt ist lediglich für die einzelnen Frames und die Zeitrechnungen zwischen diesen verantwortlich. In der Frame Funktion von `Game` wird durch alle Top-Level-Entities durchiteriert, ein rekursives Update und anschließend ein iteratives Rendering ausgeführt.

FEATURES

PFADE

Die Strecken bestehen aus Kreissegmenten und geraden Pfadstücken und sind „gehardcoded“. Auf- und Abfahrten sind mit Catmull-Rom-Splines realisiert.

Generell speichern Pfade Richtung, Vor- und Nachgänger, nebenläufige Pfade sowie von welchen Teilen des Pfades aus diese mit welchem Richtungsinput erreicht werden können und natürlich alle notwendigen Infos, die den Pfad definieren. Die Geometrie der Pfade wird beim Instanziiieren generiert.

Alle Funktionen jedes Pfades, die mit bestimmten Punkten auf dem Pfad zu tun haben arbeiten nicht mit einem Skalar, der von 0 bis 1 läuft, sondern mit der Entfernung des Punktes auf der Strecke zum Startpunkt des Pfades. Ein Laufparameter läuft also von 0 bis `distance()`, das macht viele Berechnungen (besonders in den `Ship` Klassen) einfacher. Von jedem Punkt auf dem Pfad kann man sich seine Position, die Richtung des Pfades an dieser Position, den Up-Vektor an der Position, den Normalvektor an der Position und alle Nachbar Pfade, die von der Position aus erreicht werden können, ausgeben lassen. Außerdem lässt sich von einem beliebigen Positions-Vektor auf den Pfad abbilden, sodass die kürzest mögliche Strecke zurückgelegt wird. Das Ermöglicht es Schiffen von jedem Punkt eines Pfades stets den kürzesten und besten Weg zum Nachbarpfad zu finden. Dieses Feature ist für den Catmull-Rom-Pfad nicht implementiert, daher gibt es auf der Strecke auch keinen Catmull-Rom-Pfad mit Nachbar-Pfad. Catmull-Rom-Pfade haben jedoch andere exklusive Funktionalitäten, die es ihnen ermöglichen zwei beliebige andere Pfade (die nicht direkt aneinanderhängen) zu verbinden. Daher werden sie auf der Strecke für die Auf- und Abfahrten genutzt.

SCHIFFE

Schiffe können auf Pfaden sowohl in als auch gegen die Richtung fahren, wobei das Fahren gegen die Richtung dem Spieler vorbehalten ist. Hat eine Fahrbahn Nebenfahrbahnen so kann ein Schiff zwischen diesen wechseln. Das Spieler-Schiff kann außerdem springen.

Die NPCs sind Schiffe, die zufällig eine Fahrtgeschwindigkeit und die Fahrbahn wechseln können. Dabei dürfen sie aber nicht wie der Spieler, die Fahrtrichtung wechseln, sondern fahren immer nur in die vorgegebene Richtung der Fahrbahn. Die Schiffe versuchen nicht aufeinander zu prallen, in dem sie ihre Geschwindigkeit an die anderen NPCs anpassen. Das schaffen sie jedoch nicht immer.

STEUERUNG

Das Schiff lässt sich momentan via W, A, S, D steuern. W und S lässt das Schiff schneller bzw. langsamer fahren, A und D sind für Spurwechsel nach links bzw. rechts da. Mit Space kann der Spieler springen.

Die Kamera lässt sich über die Pfeiltasten und das Mausrad steuern. Sie bleibt generell hinter dem Raumschiff, man kann aber ihren Winkel und Abstand kontrollieren.

F1 schaltet Wireframe an/aus (dafür sollte Bloom deaktiviert werden)
F2 schaltet Backface-Culling an/aus
P schaltet Partikel an/aus
O schaltet Bloom an/aus
I schaltet Specular Maps an/aus

DEBUG SETTINGS

Um das Spiel zu testen führten wir einige Settings in der settings.ini ein, die es leichter machen gewisse Features herauszuheben. Diese sind unter `[debug]` zu finden. `fps_counter_in_console` gibt jeden Frame vergangene Zeit seit dem letzten Frame, die vergangene Zeit seit Spiel Start und die aktuellen FPS in die Konsole aus. `aerial_view` setzt die Kamera direkt über das Schiff und stellt einen großen Abstand ein. Ist `interchange_wrap_around` auf `true` gesetzt so wechselt man nicht automatisch auf die nächste Fahrbahn, falls die aktuelle endet. Stattdessen springt man zurück an den Anfang der aktuellen Fahrbahn. Das erleichtert das Testen spezieller Fahrbahnabschnitte. `free_camera` sorgt dafür, dass die Kamera nicht mehr an das Schiff gebunden ist. Man kann von nun an mit Left Mouse Drag um den Fokus rotieren, mit Right Mouse Drag den Fokus verschieben und mit Scrollen den Abstand der Kamera zum Fokus ändern. Der Fokus wird schneller verschoben, je weiter die Kamera vom Fokus entfernt ist. `test_scene` lädt eine andere Szene, in der wir unterschiedliche Features getestet haben. Ist dies aktiviert werden die Settings `aerial_view`, `interchange_wrap_around` und `free_camera` ignoriert. Die Kamera ist hier immer im freien Modus.

VISUALS

Eine der wichtigsten Klassen ist die `Shader` Klasse, mit der man selbst definierte Werte wie Vektoren, Matrizen etc. in den Shader laden kann. Generell ist die Klasse dazu da, den eigenen Shader in OpenGL zu initialisieren und verwenden zu können. Im aktuellen Fragment-Shader können mehrere Lichtquellen unterschiedlicher Art (ambient, direktional, punktuell, Scheinwerfer) definiert werden. Es wird das Phong-Beleuchtungsmodell verwendet. Ebenfalls können im Shader Texturen definiert werden, wobei bis jetzt nur diffuse Texturen berücksichtigt werden. Texturen können über die `Texture` Klasse mit Angabe des Pfades zur Texturdatei in OpenGL initialisiert werden.

Für die unterschiedlichen Lichter gibt es eine Klasse `Light` (ambient), welche von `DirectionalLight` und `PointLight` erweitert wird. `PointLight` wird noch von `SpotLight` erweitert. Für `DirectionalLight`, `PointLight` und `SpotLight` gibt es noch Klassen, die mehrere Lichter in einer Liste managen.

Grundsätzlich kann jedes Mesh mit der `Geometry` Klasse in OpenGL geladen werden, hier werden alle geometrischen Daten pro Vertex (Position, Normale, UV-Koordinate) und die Indizes für die Faces gespeichert. Ebenso werden in `Geometry` auch das `Material` des Mesh geladen und gespeichert. Ein `Material` kann mehrere Texturen haben und die Definition von Konstanten, die Einfluss auf das ambiente, diffuse und spiegelnde Beleuchtung des Materials im Shader haben.

Um komplexe Modelle zu laden gibt es die `Model` Klasse. In dieser können mithilfe der assimp Bibliothek 3D Daten von Modellen, gespeichert in unterschiedlichsten Formaten, ins Projekt und OpenGL geladen werden. Ein `Model` besteht grundsätzlich aus einer Liste aus `Geometry` Objekten.

IN DER ABGABE VORHANDEN

Gameplay:

- Playable
- 3D Geometry
- Loose Condition (Wenn man mit NPC kollidiert)
- Intuitive Controls

- Intuitive Camera
- Textures
- Moving Objects
- Documentation
- Adjustable Parameters

Effekte

- CPU Particle System (Raketenantrieb der Schiffe)
- Vertex Shader Animation (Wenn Schiffe aufeinanderprallen)
- Specular Map (am auffälligsten auf Gebäude)
- Bloom

FEHLERHAFT ODER NUR TEILS IMPLEMENTIERT

- View Frustum Culling
- Shadow Mapping

BEKANNTE PROBLEME

- Kreissegment- und Catmull-Rom-Pfade verbinden nicht immer ganz zum nächsten Pfadelement (wir sind leider aus Zeitgründen nicht dazu gekommen das zu fixen)
- unerwünschte Sprünge der Kamera beim Durchfahren mancher Catmull-Rom-Pfade
- Bloom hat einen Offset wenn nicht im Vollbild Modus

BELEUCHTUNG

Alle gerenderten Objekte werden mit einem Directional-Light beleuchtet und haben Ambient Light Faktor. Eine Ausnahme stellen Partikel dar, die nur das Ambient Light berücksichtigen.

BIBLIOTHEKEN, EXTERNER CODE UND ASSETS

- Easing Equations von <https://github.com/jesusgollonet/ofpennereasing> übernommen
- Model Loader (<http://www.assimp.org/>)
- mehrere kleinere Code stücke, wobei die URL jeweils über dem Code angegeben ist
- 3D Model des Schiffes von Rudolf Getel über Turbosquid erhalten
<https://www.turbosquid.com/es/3d-models/corsair-fighter-lwo-free/215828>
- 3D Model des Gebäudes von namk07a3 über Turbosquid erhalten
<https://www.turbosquid.com/3d-models/raffles-city-tower-max-free/988483>

EFFEKT IMPLEMENTIERUNG

VERTEX SHADER ANIMATION

Dieser Effekt war ziemlich einfach zu Implementieren. Ziel war, dass sich eine Art Schockwelle ausbreiten soll, die alle Pfad Elemente in einem bestimmten Radius kurzzeitig anhebt, sobald ein Schiff explodiert. Um diesen Effekt überhaupt möglich zu machen musste erst gewährleistet werden, dass jeder Pfad genug Vertices hat, die animiert werden können. Bis zum Zeitpunkt der Implementierung dieses Features hatten alle gerade verlaufenden Pfade genau 8 Vertices (4 am Anfang des Pfades und 4 am Ende). Daher haben wir die Klasse `StraightPath`, wie auch die anderen beiden Klassen `ArcPath` und `CatmullPath` von `MultisegmentPath` erben lassen, der bei der Erzeugung der Geometrie immer gleich große Segmente erzeugt. Anschließend haben wir im Vertex-Shader das neue Struct `Explosion` erstellt, das den Ursprung und die vergangene Zeit seit der Explosion abspeichert. Andere Konstanten halfen dabei den Effekt möglichst gut aussehen zu lassen. Der uniform Array `explosions` nimmt nun die Daten vom Programm entgegen und wird in der `main` Funktion durchiteriert. Die Zeit, die seit der Explosion vergangen ist,

bestimmt den Radius in wessen Nähe die Vertices angehoben werden sollen. Schlussendlich wird mit der easing-Funktion `easeInOutQuat` eine schön gekurvte Erhebung erzeugt.

CPU-PARTICLE SYSTEM

Ziel war es den Raketenantrieb der Raumschiffe zu visualisieren. Dazu haben wir die drei Klassen `ParticleSystem`, `ParticleSpawner` und `Particle` erstellt. Da viele unterschiedliche Parameter für das Erstellen eines Partikelsystems notwendig sind und die Klassen zum Teil dieselben Parameter benötigen haben wir uns dazu entschieden zusätzlich das Struct `ParticleDTO` zu erstellen, welches all diese Parameter zusammenfasst und default Werte anbietet, damit nicht immer alle Parameter angegeben werden müssen. Generell wird vorausgesetzt, dass das `ParticleSystem` immer im Weltursprung bleibt, damit die Partikel, die als Children generiert werden nicht verschoben sind. Der `ParticleSpawner` kann hingegen frei verschoben werden und auch parent-Elemente haben, die sich bewegen. Das `ParticleSystem` ruft jeden Frame in `ParticleSpawner` die Methode `spawn` auf und teilt ihm mit wie viele er maximal erzeugen darf, um nicht die maximal erlaubten Partikel zu überschreiten. Der Spawner überprüft ob er nahe genug zur Kamera ist, um Partikel zu erzeugen, berechnet anhand der vergangenen Zeit und der `spawnRate` wie viele Partikel er erzeugen soll und liefert eine Liste an Positionen und Richtungen, die das `ParticleSystem` verwendet, um neue Particles zu generieren.

Die Particles selbst updaten jeden Frame ihre Position anhand ihrer Velocity, sowie die Transparenz und ihre Größe, zusätzlich orientieren sie sich so, dass sie immer Richtung Kamera schauen. Um das Verhalten der Partikel so flexibel wie möglich zu machen kann man von außerhalb einen Function-Pointer auf easing-Functions definieren, die verwendet werden sollen, um Größe und Transparenz zu bestimmen. Sobald ein Partikel sein Alterslimit überschritten hat wird es vom `ParticleSystem` entfernt.

Um die Transparenz der Partikel untereinander und die Transparenz in Kombination mit der Transparenz der Pfade richtig darzustellen sammeln wir vor dem Rendern alle Objekte der Szenen-Hierarchie in einer Liste, die wir der Entfernung zur Kamera nach ordnen. Da besonders bei Pfaden, die aus großen Kreissegmenten oder geschwungenen Catmull-Kurven bestehen die Entfernung zum Objektsprung viel zu ungenau ist verwenden wir hier die Methode `distEstimateTowards` die den Pfad in größeren Intervallen abfährt und den geringsten Abstand eines Punktes auf dem Pfad zur Kamera zurück gibt. Ein neues Problem wird dadurch jedoch verursacht: Die kürzeste Distanz zur Kamera ist meist geringer als der Abstand der meisten Partikel zur Kamera. Das kommt daher, dass die Kamera Diagonal von oben auf das Schiff hinabblickt. Der nächste Punkt des Pfades ist direkt unter der Kamera, während die Partikel zwar über dem Pfad sind, aber auf der xz-Ebene eine größere Entfernung haben. Um das zu beheben haben wir etwas geschummelt und die Distanz-Berechnung der Partikel verändert. Anstatt der korrekten Distanz wird $(1 - 1 / \text{dist})$ zurückgegeben. Das garantiert quasi, dass die Partikel als aller letztes gerendert werden, hält aber immer noch die Reihenfolge, die die Partikel untereinander haben intakt. Eine unerwünschte, aber vernachlässigbare Folge daraus ist, dass Partikel nun nicht mehr durch die Unterseite eines Pfades gesehen werden können, da sie von seinem Depth Buffer verdeckt werden.

SPECULAR MAP

Die Specular Map haben wir hauptsächlich deshalb gewählt, um das Gebäude Modell, das wir im Spiel platzieren, attraktiver aussehen zu lassen, wir haben uns später dazu entschieden auch Elemente des Pfades mit einer Specular Map hervorzuheben. Die Texturen an sich haben wir mit Paint.net / Photoshop auf Basis der diffusen Texturen erzeugt. Neben dem Sampler für diffuse Texturen haben wir einen Sampler für Specular Maps eingeführt. Falls eine Specular Map vorhanden ist und Specular Maps aktiviert sind wird statt dem specular wert, den das Material standardmäßig verwendet der rot-wert aus der Specular Map verwendet, hierbei werden dieselben TexCoordinates wie für die diffuse Textur verwendet.

BLOOM EFFEKT

Für den Bloom Effekt haben wir im ersten Schritt die aktuelle Szene in zwei unterschiedlichen Framebuffers rendern lassen. In einem Framebuffer wird die Szene mit dem normalen phong.frag Fragmentshader gerendert, in dem anderen Framebuffer wird der Hintergrund der Szene auf schwarz gestellt und die helleren Farben werden mit dem bloomFilterShader.frag Fragmentshader herausgefiltert. Die Texturen der Framebuffers werden über ein einfaches Rechteck gerendert.

Das gefilterte Bild wird jetzt nach dem Ping-Pong Prinzip mit einem gaussian Filter geblurt. Hier werden zwei unterschiedliche Framebuffer erstellt, die das gefilterte Bild in Abwechslung immer weiter bluren. Dabei wird im Fragment Shader vom gaussian Filter immer alternierend horizontal und vertikal geblurt, da das Performance spart.

Wenn nach einer bestimmten Anzahl an Iteration das gefilterte Bild fertig geblurt wurde, wird dieses jetzt mit der Textur des Framebuffers, welches die Szene ganz normal gerendert hat, mit einem Blending shader zusammengefügt. Der Fragment Shader für das Blending addiert die Farbwerte der beiden Texturen dann miteinander.

SHADOW MAPPING

Die Entwicklung von Shadow Mapping haben wir nach einiger Zeit abgebrochen, da kaum etwas weiter gegangen ist und der Zeitdruck immer stärker geworden ist. Generell hat das Rendern auf einen anderen Frame Buffer funktioniert, die gerenderte Shadow Map war jedoch sehr stark verschoben und die Schatten waren deshalb an den Falschen stellen. Die Ursache für die Verschiebung konnten wir nicht finden und es war nicht gewährleistet, dass der restliche Code funktioniert, sobald dieses Problem beseitigt war.

FRUSTUM CULLING

Das Frustum Culling haben wir bei uns geometrisch implementiert, in dem wir über die Kameraposition, den Fokus und den FOV die einzelnen Planes des Frustums ausrechnen. Um zu überprüfen, dass ein Objekt im Frustum ist, berechnen wir für die einzelnen Geometries in unserem Code eine Bounding Box. Leider ist bei der Überprüfung der Bounding Boxes, ob diese im Frustum sind, oder vielleicht auch wo anders ein Bug, den wir bis jetzt nicht finden konnten.

VERWENDETE TOOLS

- Visual Studio (Coding, git)
- Blender (Modell Bearbeitung / Konvertierung)
- Photoshop (Texturbearbeitung)
- Paint.net (Texturbearbeitung)
- OneNote (Skizzen, Notizen)
- Discord (Kommunikation)
- Chrome (Recherche)
- Word (Dokumentation)