Programmierung einer Engine für erleichterte 3D-Spiele-Entwicklung

FM3D

Flying Mind 3D Engine

Besondere Lernleistung

Technikwissenschaft & Mathematik
Berufliches Gymnasium Groß-Gerau

Autoren: Felix Lösing, Max Schmitt
Wohnorte: Groß-Gerau, Trebur

Prüfer: Uwe Homm, Friedhelm Ernst
Schulleiter: Martin Gonnermann

Abgabedatum: 30.3.2017

Inhaltsverzeichnis

Abkürzungsverzeichnis

CLR Common Language Runtime

FBO Framebuffer Object

FM3D Flying Mind Engine

GPU Graphics Processing Unit

IBO Index Buffer Object

PBR Physically Based Rendering

VBO Vertex Buffer Object

VAO Vertex Array Object

WPF Windows Presentation Foundation

XAML Extensible Application Markup Language

XML Extensible Markup Language

SLERP Spherical Linear Interpolation

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einführung

1.1. Warum haben wir dieses Themenfeld gewählt?

Uns ist aufgefallen, dass man bei größeren Schulaufgaben, bei denen man auf sich alleine gestellt ist, mehr mit einer Programmiersprache ausprobieren und experimentieren kann. Das Erfolgserlebnis beim Fertigstellen einer Software ist immens.

Da ein Großteil der Schüler in der heutigen Zeit mit dem Medium „Videospiele“ konfrontiert ist, kann es für sie ein höchst interessanter Prozess sein, ein eigenes Videospiel zu entwickeln. Dies haben wir auch praktisch in unserem eigenen Unterricht positiv zu spüren bekommen. Leider ist die Entwicklung eines Spieles ein sehr komplexer Prozess, welcher bei großen Industrien sogar in vier bis fünf Arbeitsbereiche unterteilt wird: (Produzent), Programmierer, Grafiker, Spiele-Designer und Sound-Designer. [1]

Für die Entwicklung eines Spieles ist außerdem einige höhere Fachkompetenz von Nöten, die nicht jeder Schüler besitzt und die nicht von Grund auf bei jedem vorhanden ist. Zudem ist die Entwicklung „from scratch"bzw. vom Nullpunkt eines Spieles sehr zeitaufwendig und nicht jeder Lehrer kann so viel Zeit für ein Projekt zur Verfügung stellen. So könnte man nun also eine Engine verwenden, um ein Spiel zu entwickeln. Hier tritt aber ein weiteres Problem auf: Die meisten Engines sind sehr kostspielig oder verfügen über eine eigene Programmier-/ Skriptsprache. Diese Engines sind rein zum Spiele-Entwickeln konzipiert, um Zeit zu sparen und damit kommerzielle Ziele zu erreichen. Sie wurden nicht entwickelt, um die Struktur einer Programmiersprache zu verstehen und zu erlernen.

Darum wurde die Flying Mind Engine (FM3D) Engine entwickelt. Die FM3D Engine besitzt die wichtigsten Komponenten, um ein Spiel zu entwickeln: Ein Mathsystem, ein Memorysystem, ein Filesystem, ein Grafiksystem und einige Funktionen, die es vereinfachen, ein Spiel zu entwickeln. Zudem wurde auch ein „Designer“ entwickelt, in dem man die 3D-Modelle laden und die Grundstruktur des Spiels zu einem VisualStudio 2015 Projekt exportieren kann. Der Lehrer kann dem Schüler zum Beispiel die Aufgabe geben, einen kleinen Roboter durch ein Labyrinth zu einer Fahne hindurch zu manövrieren. Der Schüler kann nun alle Einstellungen im Designer tätigen, sich ein funktionsfähiges Projekt exportieren lassen und sich vollkommen auf die Programmierung der Steuerung des 3D-Modells des Roboters fokussieren.

Natürlich können die Schüler mit der Engine auch größere Projekte programmieren und sie auch für private Projekte nutzen. Aber die Engine ist nicht nur für Schüler entwickelt worden. Auch Hobby-Entwickler können mit ihr einfach und simpel eigene Spiele entwickeln, ohne dabei den Überblick im Code zu verlieren.

Die FM3D-Engine kann auch für Programme mit dreidimensionaler grafischer Oberfläche verwendet werden, welche nicht in den Anwendungsbereich *Spiele* fallen. Darunter zählen zum Beispiel Lehrsoftware, Simulationen oder mehr.

1.2. Begriffsdefinitionen

1.2.1. Spiele

Bevor man eine Spiele-Engine programmiert, sollte man sich erst einmal die Frage stellen, was genau ein *Spiel* ist. Wir alle können durch unseren Alltag etwas mit dem Begriff „Spiel“ verbinden. Doch müssen wir für das allgemeine Verständnis auch hier noch einmal definieren, was genau ein Spiel ist; Brettspiele wie Schach, Dame, Mühle; Glücksspiele wie Poker, Roulette, Lotto; Kinderspiele wie Topfschlagen; und natürlich die Video-Spiele. Dies sind für uns alle Spiele, doch warum? Schlägt man den Begriff „Spiele“ in dem Nachschlagewerk „Der Brockhaus“ nach, so bekommt man die folgende Definition:

> „Spiel das, 1) jede Tätigkeit, die aus Freude an dieser selbst geschieht, im Ggs. zur zweckbestimmten Arbeit. [...]“ (Aus [2])

Spiele dienen also der Unterhaltung und dem Zeitvertreib des Spielers. Diese Beschreibung trifft auch auf Videospiele zu, denn auch sie sind größtenteils nicht zweckbestimmt und dienen nur zur Unterhaltung des Spielers. Raph Koster definiert den Begriff „Spiel“ in dem Buch „A Theory of Fun for Game Design“ wie folgt:

> „Ein Spiel ist eine interaktive Erfahrung, die dem Spieler ermöglicht, steigende Herausforderungen von Mustern, welche der Spieler im Laufe des Spieles lernt, zu meistern.“ ([3], Zitat aus dem Englischen eigene Übersetzung)

1.2.2. Video-Spiele

In der Videospielindustrie ist ein Spiel eine Software, die Bilder oder virtuelle dreidimensionale Objekte, die durch Spieler-Eingaben beeinflusst werden können, um ein vom Entwickler (oder genauer vom Spieldesigner) festgelegtes Ziel zu erreichen. Im Normalfall wird vom Spieler ein menschen- oder tierähnliches Wesen oder Fahrzeug durch ein Eingabegerät fortbewegt. Betont wird hier der Normalfall, da es verschiedene Genres oder Videospiel-Formen gibt. Hier ist die Kreativität des Entwicklers gefragt.

Gregory definiert in dem Buch „Game Engine Architecture“ ein Video-Spiel aus technischer Sicht als „soft real-time interactive agent-based computer simulations“ [1] (interaktive agentbasierte computergestützte Echtzeitsimulationen). Doch wie kommt man zu diesem Begriff?

Die meisten Spiele bilden eine Simulation einer realen oder fiktiven Welt ab, die mathematisch modelliert wird. In einer Agent basierten Simulation interagieren verschiedene „Entities“ miteinander, weshalb auch objektorientierte Programmiersprachen (wie z.B. C++ oder Java) verwendet werden. Solche interaktiven Videospiele sind zeitlich abhängige Simulationen, da sich Eigenschaften, wie zum Beispiel das Aussehen, in dieser fiktiven Welt mit der Zeit verändern. Außerdem sollte das Programm in Echtzeit auf die Eingaben des Spielers reagieren.

1.2.3. Game-Engine

Wenn man Spiele entwickeln möchte, so taucht früher oder später das Wort „Game-Engine“ auf. Einige Entwicklungsfirmen verweisen auch auf ihre Engine, um technisch besser da zu stehen und ihrem Spiel ein besseres Ansehen zu verschaffen. Heißt das aber, mit einer besseren Game-Engine kann man ein besseres Spiel entwickeln? Um dies herauszufinden, sollte man sich erst einmal anschauen, wie solche Engines funktionieren und aufgebaut sind.

Frühere Spiele wie „Tetris“, „PacMan“ oder „Space-Invaders“ wurden immer „from scratch“ entwickelt.[1] Die Entwickler besaßen also so gut wie gar kein Gerüst, bevor sie dieses Spiel programmierten. Da die meisten Spiele aber einen ähnlichen Aufbau besitzen, fiel auf, dass man immer einen ähnlichen Programm-Code verwenden könnte, um die Grundstruktur eines Spieles zu entwickeln. Vereinfacht verfügt ein Spiel über die Zentralbereiche der Grafik und die der Logik. Die Soundausgabe ist bei Spielen optional, da auch Spiele ohne Musik und Sound-Effekte existieren. Dennoch bieten die meisten Engines auch hierfür ein Grundgerüst.

Eine Game-Engine bietet dem Entwickler also diese Rahmenfunktionen, die seiner Software das Grundgerüst liefert. Dies heißt in der Softwareentwicklung „Framework“. Moderne Game-Engines sind auf ein bestimmtes Genre oder auf bestimmte Aufgabenbereiche spezialisiert. So gibt es Game-Engines sowohl für 2D- als auch 3D-Spiele. Der „GameMaker“ von „YoyoGames“ ist zum Beispiel auf 2D-Spiele spezialisiert. Doch kann man auch mit größerem Aufwand Spiele in 3D mit ihm erstellen.[**?**] Die „CryEngine“ hingegen ist hauptsächlich für 3D-Spiele konzipiert.[4] (Für weitere Details zu diesem Thema siehe 1.4.)

1.3. Geschichte

1.3.1. Vergangenheit bis heute

Die ersten Konsolen auf dem Markt besaßen geringen Speicher. So mussten die Entwickler die Spiele immer speicherkalkuliert entwickeln, sodass sie möglichst klein und speichersparend waren.[5] Da man also für jedes Spiel einen optimierten Code benötigte, waren Game-Engines von Drittanbietern nicht nötig.

Erst mit den 3D-Spielen und verbesserter Hardware wurden die Game-Engines von externen Unternehmen populär. Konsolen und PCs verfügten nun über genug Speicher und man musste nicht mehr auf jedes Byte achten. Die „Angst“, man könne Speicher verschwenden oder ein zu großes Programm entwickeln, wurde mit der Zeit immer mehr zurück gedrängt. Da die Entwickler nun auch Spiele herausbringen wollten, welche auf dem Markt besser ankommen und technisch fortschrittlicher sind, wurde der Code der Software um einiges komplexer.

So kam man schnell auf die Idee, ein System zu entwickeln, das für die Hauptbereiche eines Spiels ein Grundgerüst baut, die Software in Betrieb hält und das immer wieder verwendbar ist. Mit einer Game-Engine konnte man nun Zeit und Aufwand beim Programmieren sparen, mehr Zeit in die Optimierung des Spieles stecken und sich mehr Zeit für die Spielinhalte nehmen.

1.3.2. Zukunft der Game-Engine

Auch in der Zukunft werden Game-Engines stark von Bedeutung sein. Wie man schon in der Vergangenheit bemerkt hat: mit der Hardware, die sich mit der Zeit immer weiter verbesserte und komplexer wurde und immer noch wird, verändert sich auch die Komplexität der Syntax einer Programmiersprache. Wenn also hochwertigere *neue* Spiele in kurzer Zeit produziert werden sollen, so wird es immer ineffizienter, den Code „from scratch“ zu programmieren. Game-Engines oder auch andere Engines werden somit immer wichtiger. Auch die Produktionskosten werden dadurch verringert.

Um eine nutzerfreundliche Entwicklungsumgebung zu schaffen, sollte man sich mit anderen Engines und deren GUI vertraut machen. In den folgenden Kapiteln analysieren wir einige populäre Game-Engines und deren Features, um deren Funktionsweise zu verstehen und herauszufinden, was sie unterscheidet.

1.4. Engines

1.4.1. GameMaker - YoyoGames

Die zuerst für 2D-Animationen konzipierte Game-Engine „GameMaker“ (In den Anfängen *Animo*) wurde von Overmars erstmals im Jahre 1999 veröffentlicht und ist nun als „GameMaker:Studio“ bekannt. Zunächst hat man sich bei der Entwicklung sehr stark auf die 2D-Spiele Entwicklung konzentriert. Mittlerweile kann man mit ihr sogar 3D-Spiele entwickeln.

Das Programm ist momentan in drei verschiedenen Versionen erhältlich (Stand 2017). Darunter auch eine kostenlose Version mit eingeschränkten Funktionen. Die Benutzer-Oberfläche ist sehr simpel gehalten und so konzipiert, dass sogar ein Laie den Programmablauf eines Spieles durch einfaches „Drag’n’Drop“ von Schaltflächen schnell erstellen und strukturieren kann. Zudem besitzt der „GameMaker“ eine eigene Skript-Sprache, welche einige der höheren Programmiersprachen (zum Beispiel Pascal, Java und C) als Vorbild hat.

Die mit der „GameMaker“-Engine entwickelten Spiele werden mit verschiedenen Ressourcen entwickelt; Sprites, Sounds, Backgrounds, Paths, Scripts, Fonts, Timelines, Objects und Rooms. Auch wir wollten dies so ähnlich in unserem Designer handhaben und haben uns von dem Prinzip der Ressourcen inspirieren lassen (siehe Abschnitt 3.4).

Die Spiele können für diverse gängige Betriebssysteme (Windows, Android, iOS u.a.), aber auch für populäre Spielekonsolen (PlayStation, XBox u.a.) kompiliert werden.

1.4.2. Unity3D - Unity Technologies

„Unity3D“ ist eine Game-Engine der Firma *Unity Technologies*, welche erstmals im Jahre 2005 veröffentlicht wurde. Sie ist für die 3D-Spiele Entwicklung konzipiert, besitzt eine eigene 2D-Engine, ist weitaus komplexer als der „GameMaker“ und bietet somit auch mehr Funktionen. Mit ihr kann man Spiele für Computer, aber auch Spielkonsolen, mobile Geräte und Webbrowser entwickeln. Sie ist für Windows, Linux (nur Beta) und OSX in vier verschiedenen Versionen erhältlich, unter denen sich auch eine kostenlose Version befindet. Das Programm beinhaltet zudem noch einen 3D-Terrain-Modellierer, Baum- und Pflanzen-Modell-Editor, Werkzeuge für Partikeleffekte und ein Tool für Bewegungssteuerung für Charaktere.

Die grafische Entwicklungsumgebung ähnelt sehr modernen 3D-Animationsprogrammen. Sie stellt im Hauptfenster eine Spiel-Szene dar und verfügt zudem über eine sogenannte *Timeline*, in der der Zeitablauf des Spieles dargestellt wird. In Menüs kann man Parameter des Spieles verändern und sie für die Strukturierung des Spieles verwenden.

Das Spiel-Scripting basiert auf dem *Mono*-Compiler und bietet verschiedene Scriptsprachen, darunter auch C# und eine vom Entwicklerteam eigens entwickelte Sprache „UnityScript“. Sie ist hauptsächlich für *Cross-Platform* Programme gedacht. Das heißt, dass alle Scripte sofort für verschiedene Geräte exportiert werden können, ohne davor den Code an das Gerät anpassen zu müssen.

Eine aktuelle Szene besteht aus *Game-Objects*, denen man verschiedene Eigenschaften (Materialien, Klänge, physikalische Eigenschaften, Skripte) zuordnen kann. Dies haben wir auch bei unserem Designer mittels Komponenten realisiert (siehe Abschnitt 3.4 oder Abschnitt 3.2.4).

Aufgrund dieser Komplexität ist die „Unity3D-Engine“ eher für Fortgeschrittene geeignet und braucht weitaus mehr Einarbeitungszeit als der „GameMaker“, sie bietet jedoch auch mehr Möglichkeiten. Bei der Auswahl einer geeigneten Engine muss man abwägen, ob eine einfach zu bedienende Engine auch allen erforderlichen Entwicklungsmöglichkeiten bietet.

2. Theorie

2.1. Rendering

Im folgenden Abschnitt werden die gewonnenen Erkenntnisse aus mehreren Online-Tutorials und Nutzerhandbüchern zusammengefasst. Verwendet wurden die Materialien der Khronos Group, E.Meiri, ThinMatrix, T.Cherno, OpenGL Tutorials.[6, 7, 8, 9, 10]

2.1.1. OpenGL Grundlagen

Die FM3D-Engine verwendet für die Darstellung von dreidimensionalen Szenen die Grafikbibliothek OpenGL. Mit OpenGL ist es möglich, verschiedene kleine grafische Objekte zu rendern. Diese entsprechen drei geometrischen Grundobjekten (*Primitives*): Punkte, Linien und Dreiecke. Zudem gibt es verschiedene Möglichkeiten diese aneinander zu reihen, wie in Abbildung 1 dargestellt ist. Sie können alle einzeln, aber auch aneinanderhängend gerendert werden. Bei letzterem kann Speicher bei den angrenzenden Eckpunkten gespart werden. Diese *Primitives* werden durch ihre Eckpunkte, oder auch *Vertices* genannt, definiert. Sie beschreiben eine Position in einem dreidimensionalen Raum. OpenGL arbeitet mit einem orthografischem Koordinatensystem. Das heißt: alle drei Achsen sind orthogonal zueinander und reichen von den Werten -1 bis 1.

Frame-Buffer

Das aktuelle Bild wird in mehreren Buffern gespeichert. Die Größe der Buffer ist direkt proportional zu der Pixelanzahl des *Viewports*, also zu dem Bereich, in welchem das ge-

renderte Bild dargestellt wird. Am relevantesten ist der Color-Buffer, der die Farbe jedes Pixels in vier Floats speichert: Jeweils einen für Rot, Grün, Blau und einen Alpha-Wert, welcher die Transparenz beschreibt. Des Weiteren gibt es den Depth-Buffer. Dieser ist selten auch als Z-Buffer in der Literatur zu finden. Er beschreibt den Abstand zwischen dem aktuell gerenderten Pixel und der Kameraebene. Diese Information wird als Farbinformation in dem Depth-Buffer gespeichert. Jede Textur besteht aus verschiedenen Farbkomponenten. Diese nennt man in der Fachliteratur „Channels“. Man benötigt nur einen Channel, da nur ein einziger Zahlenwert gespeichert werden muss. So ist, wenn man den Buffer anzeigt, nur ein Schwarz-Weiß-Bild zu sehen. Je heller der Pixel ist, desto weiter weg befindet er sich. Die Größe des Depth-Buffer ist einstellbar. Je größer er ist, desto größer ist auch die Präzision, wobei der Depth-Buffer immer eine viel größere Präzision in der Nähe der Kamera besitzt und nach weiter hinten an Präzision verliert. Dies ist von Vorteil, wenn Objekte sehr nah an der Kamera gerendert werden, da dort eine sehr hohe Präzision erforderlich ist. Die Präzision ist aber nicht sehr relevant, wenn das Objekt weiter entfernt ist. Der Depth-Buffer ist optional. Wird er nicht verwendet, so kann die räumliche Anordnung der Primitives nicht ermittelt werden. Welcher Pixel am Ende angezeigt wird, ist von der Reihenfolge, in der die Primitives gerendert werden, abhängig. Wobei das zuletzt gerenderte Primitive ganz vorne zu sehen ist. Diese Buffer sind in Abbildung 2 dargestellt.

Zudem gibt es noch den *Stencil-Buffer*. Dieser ist ebenfalls optional und ordnet jedem Pixel einen bestimmten Wert zu. Er kann verwendet werden, um bei bestimmten Pixeln das Rendern zu verhindern. Wie er dies umsetzt, ist einstellbar. Es ist möglich verschiedene Operationen durchzuführen, wenn ein Pixel gerendert wird. Es ist zum Beispiel möglich, dass der Stencil-Wert auf 1 gesetzt oder um 1 erhöht wird. Zudem ist es möglich den Stencil-Test einzustellen. Dieser wird bei jedem Renderprozess eines Pixels ausgeführt. So kann man zum Beispiel erreichen, dass nur dort, wo der Stencil-Buffer den Wert 1 besitzt, ein Pixel gerendert wird. Dadurch können *Schablonen* erstellt werden, die festlegen welche Teile eines Objekts gerendert werden sollen.

Alle Buffer zusammen werden in einem Framebuffer Object (FBO) gespeichert. Dieses ermöglicht das gleichzeitige Aktivieren aller zugehörigen Buffer, sowie das Anzeigen des Color-Buffers auf dem Bildschirm. Es muss nicht speziell ein FBO erstellt werden. Wenn keines erstellt wird, so wird standardmäßig direkt auf den *Screen-Buffer* geren-

dert. Ein weiterer Vorteil ist, dass man ein FBO sowohl als Output (dies ist die häufigere Verwendung) aber auch auch als Input verwenden kann. So kann man zum Beispiel eine gerenderte 3D-Szene in einer anderen 3D- oder 2D-Szene einfügen.

Buffer Objects

Will man einen oder mehrere dieser *Primitives* rendern, so laufen die Daten, welche diese *Primitives* definieren, durch verschiedene Schritte. Diese werden in der Literatur auch oft als die „Rendering-Pipeline“ bezeichnet. Bei Verwendung dieser Pipeline kann immer nur eine Art von Primitives gleichzeitig gerendert werden. Die Daten werden dabei als Buffer auf der Graphics Processing Unit (GPU) gespeichert. Verwendet werden dazu die Vertex Buffer Objects (VBOs), welche als Byte-Arrays vorliegen. Bei ihnen muss manuell eingestellt werden, wie diese Bytes interpretiert werden sollen. Dafür beschreibt man verschiedene Attribute mit Byte-Anzahl und Größe sowie DatenTyp. Zum Beispiel beschreiben die ersten 4 Bytes einen Float und die darauffolgenden 12 einen 3D-Float-Vektor. Es können auch mehrere VBOs in einem Vertex Array Object (VAO) zusammengefasst werden. VAOs speichern den Zustand der enthaltenen VBOs und die Information, welche Attribute verwendet werden. Zusätzlich zu einem oder mehreren VBOs kann ein Index Buffer Object (IBO) verwendet werden. Dieser besteht aus einem Array von ganz-rationalen Zahlen und wird ebenfalls auf der GPU gespeichert. Er beschreibt, welche Vertices für welche Primitives verwendet werden sollen, wobei Vertices mehrfach verwendet werden können. Der Index-Buffer wird nacheinander durchgegangen und jeder darin gespeicherte Index steht für einen Wert im VBO. Wenn man also viele Primitives hat, die sich einen gemeinsamen Punkt teilen, ist es effizienter, einen Index-Buffer zu verwenden, da dieser gemeinsame Vertex nur einmal gespeichert werden muss. In der Regel ist die Größe eines Vertex weitaus größer, als die Größe eines Index. Zum Beispiel können mit dem IBO 0, 1, 2, 2, 1, 3 zwei Dreiecke gerendert werden. Für die gleiche Darstellung werden aber nur 4 Vertices benötig (0,1, 2, 3)[8].

Pipeline

Ein VBO oder ein VAO entsprechen dem Input der Pipeline, wobei in modernen Programmen immer VAOs verwendet werden, da sie zusätzliche Informationen speichern können.

In der Pipeline bestehen mehrere Schritte aus Shadern. Diese Shader sind kleinere Programme, welche auf der Grafikkarte ausgeführt werden. Sie werden in der Programmiersprache GLSL programmiert. Dies ist eine spezielle Sprache, die für Shader von OpenGL verwendet wird. Sie ähnelt stark C und besitzt einige bereits zur Verfügung gestellte Funktionen, um linear algebraische Rechnungen durchzuführen. Sie unterscheidet sich stark in den einzelnen OpenGL-Versionen, da ständig neue Features hinzugefügt werden. Die FM3D-Engine besitzt daher auch teilweise für verschiedene OpenGL-Versionen verschiedene Shader-Implementationen.

Ein grober Überblick über die Rendering-Pipeline wird in Abbildung 3 gegeben. Der erste Schritt ist das Erstellen der Daten und das Angeben der Attribute. Diese sind der Input für den *Vertex-Shader*.

Im *Vertex-Shader* werden die Positionen einzelner Vertices festgelegt. Daher wird er für jeden *Vertex* einmal ausgeführt. Es können zum Beispiel Operationen wie Verschiebungen durchgeführt werden. Der einfachste mögliche Vertex-Shader liest eine Position aus dem VBO und verwendet diese als Vertex-Position.

Der Ausgang des Vertex-Shader ist der Eingang des *Tessellation-Shader*. Dieser ist optional und erst seit OpenGL-Version 4.0 verfügbar. Er wird in der FM3D-Engine nicht verwendet. Daher wird nicht weiter auf ihn eingegangen.

Als nächster Schritt folgt der *Geometry-Shader*. Er wird für jedes *Primitive* einmal ausgeführt und bekommt dieses auch als Eingabe. Zusätzlich erhält er noch die Ausgabe des Tessellation-Shader bzw. des Vertex-Shader. Es können Operationen ausgeführt werden, die für jedes Primitive ausgeführt werden müssen. Es ist auch möglich die Art des Primitive zu ändern: zum Beispiel könnte man ein Dreieck in drei Linien umwandeln. Der Geometry-Shader ist ebenfalls optional.

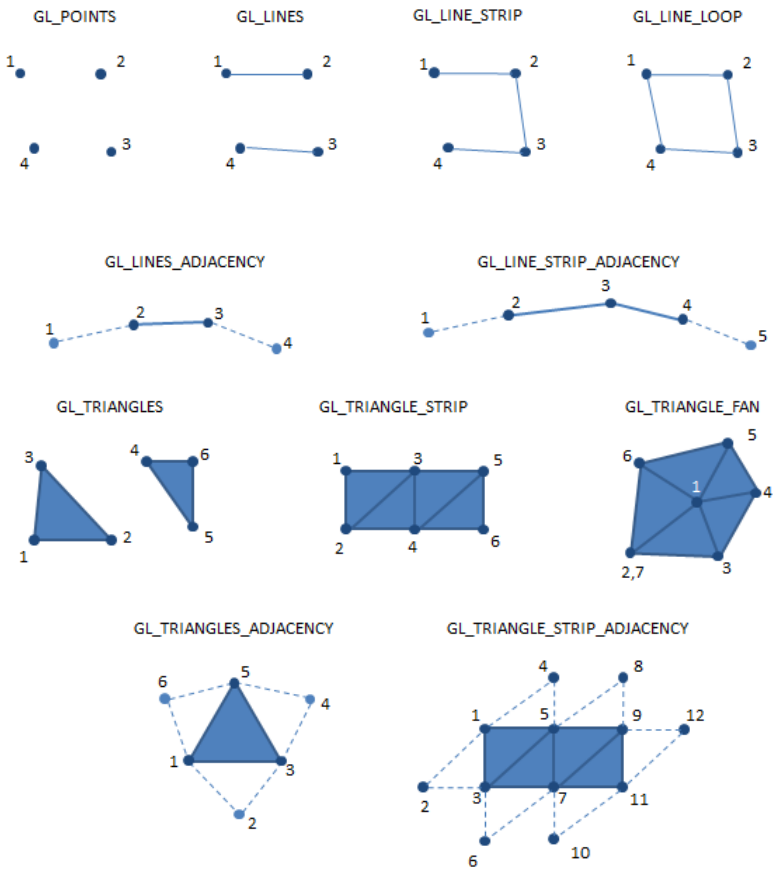
Nach diesen drei Shadern werden einige nicht veränderbare Operationen mit den Daten durchgeführt, auch bekannt als *Fixed Function Processing*. Primitives, die sich nicht mehr in dem Bereich von -1 bis 1 befinden, werden entfernt und nicht gerendert. Primitives, die sich genau auf der Grenze befinden, werden geteilt, so dass nur der Teil im erlaubten Bereich gerendert wird. Dieses Verfahren bezeichnet man als *Clipping*.

Der nächste Schritt ist abhängig von der Art von Primitive, die man gewählt hat. Wenn man eine zusammenhängende Reihe von Primitives rendert, wird diese aufgelöst und in einzelne Primitves aufgeteilt. Dies geschieht so, dass folgende Operationen immer auf ganze Primitves ausgeführt werden und nicht nur auf einen Stream von Vertices. Es wird auch eine Operation namens *Face Culling* durchgeführt. Diese Operation wird nur für Dreiecke ausgeführt. Alle Dreiecke, die von der Kamera weg zeigen, werden ignoriert ohne sie zu rendern. So kann verhindert werden, dass bei Objekten, die eine geschlossene Oberfläche aus Dreiecken aufweisen, bei denen die Rückseite eines Dreiecks sowieso nie zu sehen ist, unnötige Dreiecke gerendert werden. Es kann eingestellt werden, ob diese Operation durchgeführt werden soll oder nicht. Für transparente Objekte kann sie zum Beispiel nicht verwendet werden, da dort die Rückseite trotzdem zu sehen ist.

Danach folgt der Schritt *Rasterization*. Er beschreibt die Umwandlung von Primitives in *Fragments*, also Pixel auf dem Bildschirm. Es werden bei diesem Schritt weitere Optimisierungsvorgänge durchgeführt, um keine unnötigen *Fragments* zu rendern.

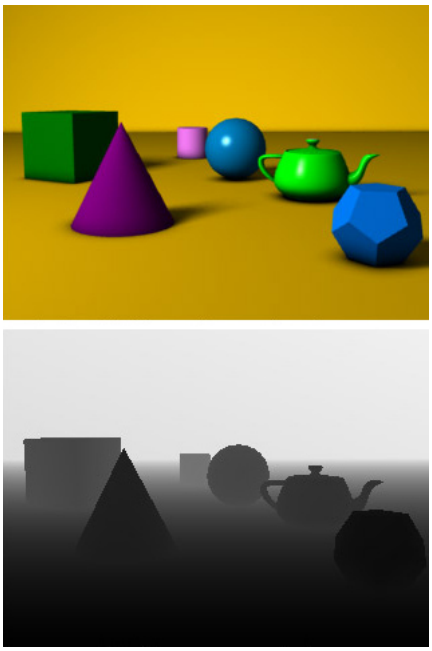
Der vorletzte Schritt ist ein Shaderprogramm. Der *Fragment-Shader* wird für jedes Fragment einmal ausgeführt und bestimmt die Farbe der Pixel. Als Input bekommt er den Output des davor ausgeführten Shaders, wobei die Werte für jedes Fragment interpoliert werden müssen und so eine Mischung aus den Daten jedes Vertex generiert wird. Diese verteilen sich linear über das Primitive. Der Fragment-Shader ist der am häufigsten ausgeführte Shader. Daher sollten alle nicht unbedingt benötigten Berechnungen in vorherigen Shadern ausgeführt werden.

Als finalen Schritt werden verschiedene Tests für das Fragment ausgeführt, wie der Depth- und der Stencil-Test, um zu bestimmen, ob das Fragment wirklich angezeigt werden soll. Wenn alle Tests positiv sind, wird das Fragment auf dem Buffer gespeichert. (vgl. [11])



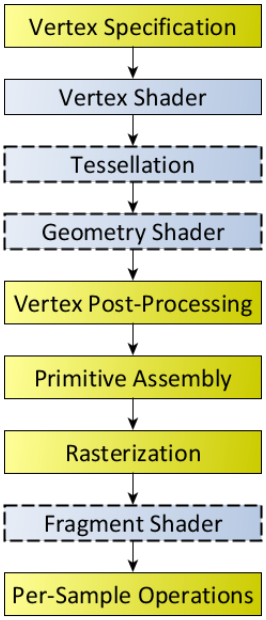
Quelle: http://www.lighthouse3d.com/tutorials/glsl-tutorial/primitive-assembly/

Abbildung 1.: OpenGL Primitives



Oben: Color, Unten: Depth
Quelle: https://de.wikipedia.org/wiki/Datei:Z-buffer_no_text.jpg

Abbildung 2.: Color- und Depth-Buffer



Quelle: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

Abbildung 3.: OpenGL Pipeline

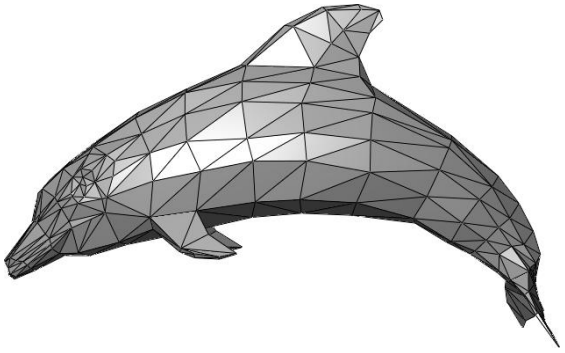


Abbildung 4.: Mesh eines Delfins

2.1.2. Physically Based Rendering

In einer 3D-Szene eines Videospiels will man natürlich komplexere Objekte als simple Primitives darstellen. Diese Objekte bestehen aus vielen Primitives, in diesem Falle meist Dreiecke. Die Art der Zusammensetzung ist unterschiedlich. Manchmal kann es effizienter sein, aneinander hängende Dreiecke zu verwenden, jedoch werden meistens nur einzelne Dreiecke gerendert. Einige Programme verwenden Polygone, welche aber auch aus Dreiecken bestehen. Daher ist dies irrelevant. Ein Objekt, gebildet aus Dreiecken, nennt man *Mesh*. Als Beispiel ist das Mesh eines Delfins in Abbildung 4 dargestellt. Die Farbe bzw. die „Haut“ eines Objektes wird aus einem zweidimensionalem Bild gelesen. Genannt wird dies *Textur*.

Dafür besitzt jeder Vertex zusätzlich zu seiner Position eine zweidimensionale Position auf der Textur. So kann ermittelt werden, welche Pixel der Textur verwendet werden sollen. Mit Mesh und Textur ist es möglich, ein Objekt in einer dreidimensionalen Szene anzuzeigen. In einem Videospiel würde dies jedoch nicht überzeugen. Es wird Physically Based Rendering (PBR) verwendet. PBR ist eine Möglichkeit, eine realistischere dreidimensionale Szene zu erzeugen, indem physikalische Phänomene wie z.B. Licht berücksichtigt werden. Wie ein Objekt auf Licht reagiert, hängt von vier verschiedenen Größen ab: der eigentlichen Farbe des Objekts, der Oberflächenstruktur, der Farbe und der Richtung des einfallenden Lichts. Alle Eigenschaften eines Objekts, die sich auf die Farbe auswirken, werden in einem *Material* zusammengefasst. Ein Objekt, welches gerendert werden soll, besitzt beides: ein *Mesh* und ein *Material*. Das ganze

wird zusammengefasst *Model* genannt.

Es gibt zwei Arten von Lichtquellen in der FM3D-Engine: *Directional Light* und *Point Lights*. Ein Directional Light besitzt eine Richtung, aber keine Position. Diese Art kann verwendet werden, um zum Beispiel eine Sonne zu simulieren, die so weit von der Erde entfernt ist, dass die Position irrelevant wird. Die Richtung der Strahlen hingegen ist wichtig und von der Tageszeit abhängig.

Point Lights sind das genaue Gegenteil. Sie besitzen keine Lichtrichtung, sondern scheinen in alle Richtungen gleich. Dafür besitzen sie eine genau festgelegte Position, die relevant ist, da die Lichtstärke mit zunehmendem Abstand kleiner wird. Point Lights können verwendet werden, um die meisten Lichtquellen darzustellen (als Beispiel: Laternen oder Fackeln).

Man kann aber nicht nur zwischen Lichtquellen unterscheiden, sondern auch zwischen verschiedenen Arten des ausgesendeten Lichts. Die FM3D-Engine verwendet ein Lichtmodell genannt „Ambient/Diffuse/Specular“. *Ambient Light* ist das Licht, das man jeden Tag sieht, auch wenn gerade keine Sonne scheint oder man sich nicht in direkter Nähe einer Lichtquelle befindet. Es entsteht dadurch, dass Licht von allen Objekten wieder teilweise reflektiert wird und so eine schwache und gleichmäßige Beleuchtung entsteht. Ohne das Ambient Light wäre es hinter einem Haus, welches die Sonne verdeckt, komplett finster.

Die zweite Lichtart ist *Diffuse Light*, welches abhängig von dem Auftrittswinkel des Lichtstrahls ist. Betrachten wir einen Würfel, so sehen wir: die dem Licht zugewandte Seite ist heller, als die nur teilweise zugewandte Seite und die abgewandte Seite erfährt gar kein *Diffuse Light*.

Specular light modelliert die Lichtstrahlen, welche von einem Objekt reflektiert und in die Linse der Kamera bzw. in die Augen des Betrachters gelangen. Dies wird als blendendes, helles Licht wahrgenommen und ist oft auf metallischen Oberflächen zu erkennen. Diese drei Lichtarten sind in Abbildung 5 dargestellt.

Alle Lichtberechnungen müssen für jedes Pixel ausgeführt werden und laufen daher im Fragment-Shader ab. Um die Farbe eines gerenderten Pixels zu bestimmen, benötigt

man einige Informationen: die Position, die Farbe, den Normalenvektor und Specular-Factor des Pixels. Diese vier Informationen reichen für alle Lichtberechnungen aus. Die Phase, in der sie erstellt bzw. berechnet werden, ist unterschiedlich. Ein Teil wird außerhalb des Programms in externen Programmen erstellt und als Vertex im Mesh oder als Information im Material gespeichert. Dabei unterscheidet man zwischen:

- Der Information, die sich für verschiedene Objekte unterscheidet, aber nicht innerhalb des Objekts
- Information, die für jeden Vertex anders ist aber für jeden Pixel nur linear interpoliert werden muss
- Information, die für jedes Pixel anders ist

Erstere kann einfach im Material gespeichert werden, da jedes Objekt ein eigenes Material haben kann und es anders als ein Mesh nicht viel Speicher benötigt. Die Vertexinformationen werden im VBO des Mesh gespeichert und automatisch linear interpoliert, wenn sie an den Fragment-Shader weitergegeben werden. Pixelinformationen müssen einzeln in einer Textur gespeichert werden. Diese sind dann im Material enthalten, wobei diese nur referenziert werden, damit verschiedene Materials die gleichen Texturen verwenden können. In den Vertex-Informationen müssen hierfür zusätzlich Textur-Koordinaten gespeichert werden. Daraus ergeben sich die folgenden Werte, die in Tabelle 1 zu sehen sind.

Die FM3D-Engine verwendet *Deferred Rendering*. Dies bedeutet, dass zuerst alle Objekte einer Szene gerendert und die Ergebnisse daraus in mehreren Buffern gespeichert werden, zusammengefasst *G-Buffer* genannt. Danach wird für jede Lichtquelle ein weiterer Renderprozess ausgeführt, wobei die vorher genannten Buffer als Input dienen. Das Ergebnis ist in der Abbildung 5 zu sehen. Der Vorteil hierbei ist, dass keine unnötige Lichtberechnungen ausgeführt werden muss, da alle Pixel später zu sehen sind und es keine Begrenzung für die Anzahl der Lichtquellen gibt. Der Aufbau des G-Buffers ist in Tabelle 3 zu sehen.

Im ersten Renderdurchgang wird nur in den G-Buffer die Informationen Position, Normalenvektor und Farbe des jeweiligen Fragments gespeichert. Der zweite Renderdurchgang ist abhängig von der Lichtquelle. Zusätzlich zum G-Buffer benötigt der Shader die

Tabelle 1.: Vertex Aufbau

Datentyp	Name	Beschreibung
3D-Vektor	Position	Position des Vertex
2D-Vektor	Textur Koordinate	Position des Vertex auf der Textur
3D-Vektor	Normal	Normalenvektor zum Dreieck
32-Bit Farbe	Color	Farbe des Vertex (optional, Standard ist weiß)
3D-Vektor	Tangent	Tangentenvektor des Dreiecks. Genaueres in Abschnitt 2.1.3

Quelle: Eigene Darstellung

Position der Kamera und die Farbinformationen der jeweiligen Lichtquelle. Die Farben für die drei Lichtarten werden dann nacheinander berechnet. Die Berechnung für Directional Light lautet folgendermaßen:

Gegeben:
Lichtinformationen: Richtung: $\overrightarrow{r}$ Farbe: $\overrightarrow{L_c}$ Ambient und Diffuse Intensität: L_a, L_d
GBufer: $\overrightarrow{G_P}, \overrightarrow{G_N}, \overrightarrow{G_C}$
Kameraposition: $\overrightarrow{K}$
Resultierende Farben der drei Lichtarten: $\overrightarrow{C_A}, \overrightarrow{C_D}, \overrightarrow{C_S}$
Finale Farbe: $\overrightarrow{C_F}$

$\overrightarrow{C_A} = \overrightarrow{L_c} \cdot L_a$
$\overrightarrow{C_D} = \overrightarrow{L_c} \cdot L_d \cdot (\overrightarrow{G_N} \cdot -\overrightarrow{r})$
$\overrightarrow{x} = \overrightarrow{r} - 2 \cdot (\overrightarrow{G_N} \cdot \overrightarrow{r}) \cdot \overrightarrow{G_N}$
$\overrightarrow{C_S} = \overrightarrow{L_c} \cdot \overrightarrow{G_{Calpha}} \cdot ((\overrightarrow{K} - \overrightarrow{G_P}) \cdot \overrightarrow{x})^{16}$

Die Berechnung für ein Point Light ist gleich, nur ist die Farbe von der Position des Lichts abhängig und die Lichtrichtung von der Position des Fragments.

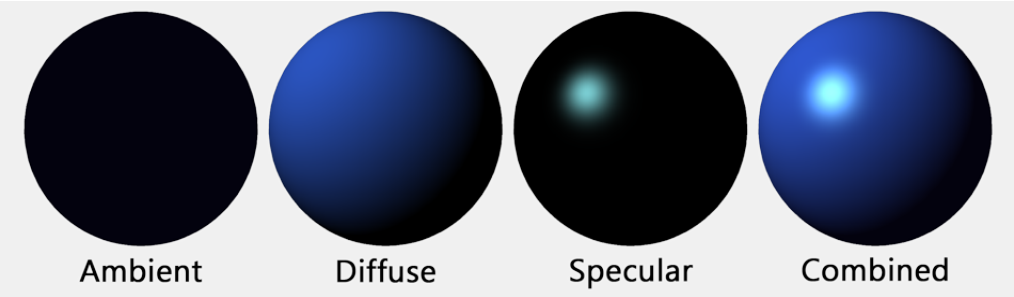
2.1.3. Normalmapping

Um die visuelle Qualität eines Models zu erhöhen, muss detailliertere Modellierung erfolgen aus der eine höhere Anzahl von Dreiecken resultiert. Damit erhöht sich aber

Tabelle 2.: Material Aufbau

Datentyp	Name	Beschreibung
32-Bit Farbe	Color	Farbe des gesamten Objekts
Textur	Color Texture	Gibt die Farbe jedes Pixels des Objektes an
Textur	Normal map	Gibt den Normalenvektor jedes Pixels an. Genaueres in Abschnitt 2.1.3
Float	Specular factor	Faktor für das resultierende Specular light
Textur	Specular map	Specular factor für jeden Pixel. Der Faktor des ganzen Objekts wird weiterhin verwendet
Boolean	UseWireframe	Gibt an ob ganze Dreiecke gerendert werden sollen oder nur die Kanten. (Nützlich für Debugging)

Quelle: Eigene Darstellung



Quelle: https://clara.io/img/pub/amb_diff_spec.png

Abbildung 5.: Lights

Tabelle 3.: G-Buffer Aufbau

Größe	Name	Channels	Beschreibung
96 Bit	Position-Buffer	RGB mit je 32-Bit float	Position im Worldspace.
128 Bit	Color-Buffer	RGBA mit je 32-Bit float	Farbe jedes Pixels in den Werten RGB un
96 Bit	Normal-Buffer	RGB mit je 32-Bit float	Normalenvektor im Worldspace
32 Bit	Depth-Stencil-Buffer	24 Bit Depth, 8 Bit Stencil	Depth- und Stencil-Wert des resultierende
128 Bit	Final-Buffer	RGBA mit je 32-Bit float	Resultierende Farbe jedes Pixels mit Alph

auch der Rechenaufwand für dieses Model. Ab einer bestimmten Grenze ist die gewonnene Qualität nur sehr gering, der Rechenaufwand aber um so größer. Um ein Model trotzdem mit einer höheren Auflösung darzustellen, wird ein Vorgang namens *Normalmapping* verwendet. Dieser baut auf Lichtberechnungen auf. Ohne Licht gibt es keine zu erkennende verbesserte Qualität. Normalerweise hat jeder Vertex einen Normalenvektor und für einen Pixel wird er zwischen den drei Vertices interpoliert. Mit Normalmapping weist man jedem Pixel einen eigenen Normalenvektor zu. So ist es möglich, die Oberfläche so aussehen zu lassen, als würde sie rau mit kleinen Unebenheiten sein. Somit wird kein Rechenaufwand durch das Rendern vieler Dreiecke eines Meshes unnötig erzeugt. Den Unterschied sieht man in Abbildung 6. Die Normalenvektoren werden in einer Textur gespeichert, sodass sie mit den normalen Textur-Koordinaten verwendet werden können. Diese Textur wird im Material abgespeichert.

In einer Textur können die drei Werte rot, grün und blau, jeweils mit einem Wert von 0 bis 1, abgespeichert werden. Ein Normalenvektor hat drei Komponenten mit jeweils Werten von -1 bis 1. Daher muss dieser erst berechnet werden:

$$\overrightarrow{N} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = 2 \cdot \overrightarrow{C} - 1 = \begin{pmatrix} 2 \cdot r - 1 \\ 2 \cdot g - 1 \\ 2 \cdot b - 1 \end{pmatrix}$$

Der Normalenvektor $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ zeigt direkt vom Model weg, daher sind Normalmaps auch immer sehr bläulich. Der meiste Anteil des Normalenvektors liegt in der Z-Komponente. Ein Beispiel einer Normalmap ist in Abbildung 7 zu sehen. Man kann diesen Vektor nicht direkt verwenden, da er im *Tangent Space* definiert ist. Dies bedeutet, dass er relativ zu dem zugehörigen Dreieck steht. Man benötigt zudem einen Normalenvektor im *Object Space*, also relativ zu dem Model. Zur Umwandlung wird eine 3x3 Matrix benötigt, die aus drei verschiedenen Vektoren des Tangent-Space gebildet wird. Es können hierbei drei beliebige nicht linear abhängige Vektoren verwendet werden. Man bemerkt schnell, dass, wenn alle drei orthogonal zueinander verlaufen, es einfacher und effizienter ist, die Matrix zu erstellen. Der erste Vektor ist der Standardnormalenvektor jedes Vertex. Hinzu kommt noch ein Tangentenvektor jedes Vertex. Dieser befindet sich ebenfalls im VBO des Meshes, wie in Tabelle 1 abgebildet. Der dritte Vektor ist der Bitan-

gentenvektor und kann mittels Kreuzprodukt aus den Standardnormalenvektor und dem Tangentenvektor berechnet werden. Es ist nötig alle drei Vektoren zu normalisieren, bevor die Matrix erstellt wird.

Gegeben ist der Normalenvektor $\overrightarrow{N}$ und der Tangentenvektor $\overrightarrow{T}$

Die normalisierten Vektoren: $\overrightarrow{N_0} = \frac{\overrightarrow{N}}{|\overrightarrow{N}|} \qquad \overrightarrow{T_0} = \frac{\overrightarrow{T}}{|\overrightarrow{T}|}$

Der Bitangentenvektor: $\overrightarrow{B} = \overrightarrow{N_0} \times \overrightarrow{T_0} \qquad \overrightarrow{B_0} = \frac{\overrightarrow{B}}{|\overrightarrow{B}|}$

Die Matrix: $M = \begin{pmatrix} T_{0x} & B_{0x} & N_{0x} \\ T_{0y} & B_{0y} & N_{0y} \\ T_{0z} & B_{0z} & N_{0z} \end{pmatrix}$

Diese Matrix wird für erhöhte Leistung im Vertex-Shader erstellt und daraufhin dem Fragment-Shader übergeben. In diesem muss der Normalenvektor aus der Normalmap geladen werden. Dafür werden die gleichen Texturkoordinaten (wie bei der Color-Texture) verwendet und mit dieser Matrix multipliziert.

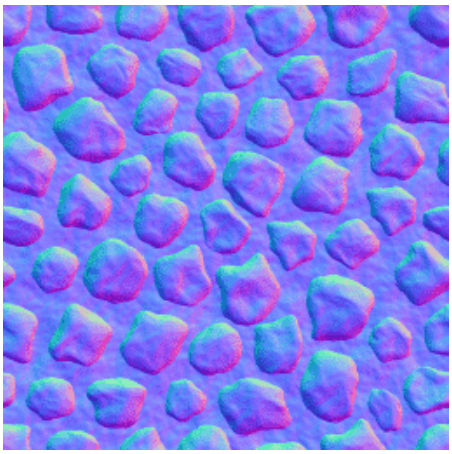
2.1.4. Animation

Für eine lebendige Szene in einem Spiel reicht es nicht aus, nur statische Models zu verschieben und zu drehen. Wenn man zum Beispiel ein Pferdmodel im Spiel einbindet und es nur nach vorne bewegt, wenn es laufen soll, so wirkt das Pferd nicht lebendig und es sähe so aus, als würde man eine Statue verschieben. Man muss in diesem Fall also ein Modell animieren.

Die FM3D-Engine verwendet eine Animationsart namens *Skeletal-Animation*. Hierbei bekommt jedes animierte Mesh ein Skelett zugewiesen, wobei sich mehrere verschiedene Meshes das gleiche Skelett teilen können. Zum Beispiel könnte es in einem Spiel ein Mesh für einen Ritter und eines für einen Bauern geben. Diese sehen unterschiedlich aus, aber beide könnten das gleiche Skelett und somit die gleichen Animationen haben.

Oben: mit Normalmapping, Unten: ohne Normalmapping
Model: Allosaurus aus dem Spiel „Ark: Survival Evolved"

Abbildung 6.: Normalmapping Beispiel



Eine Beispiel Normalmap
Quelle: http://www.bencloward.com/images/tutorial_normals07.gif

Abbildung 7.: Normalmap Beispiel

Ein Skelett besteht aus mehreren Knochen, die jeweils an weiteren Knochen „hängen“. Wenn sich ein Knochen bewegt, bewegt er alle an ihm hängenden Knochen und die wiederum alle an ihm hängenden. Dies hat den Vorteil, dass, wenn sich der Knochen des linken Armes bewegt, sich auch gleichzeitig alle Fingerknochen bewegen würden. Jeder Knochen besitzt unabhängig von den anderen eine Ausgangsposition und eine ID. Die ID wird verwendet, um herauszufinden, welcher Knochen sich auf welchen Vertex auswirkt und welche Knochen nicht verwendet werden. Jeder Vertex kann von bis zu vier Knochen transformiert werden. Dazu besitzt jeder Vertex vier Knochen-IDs und vier Floats, die angeben, wie stark sich ein Knochen auf den Vertex auswirkt (siehe Tabelle 1). Diese Daten werden in einem externen Programm beim Erstellen des Meshes festgelegt. Dieser Vorgang wird *Weight Painting* genannt, da die Zuweisung durch eine Art Malen in den Modellierungsprogrammen geschieht. Ein Beispiel aus dem Programm *Blender* ist in Abbildung 8 zu sehen.

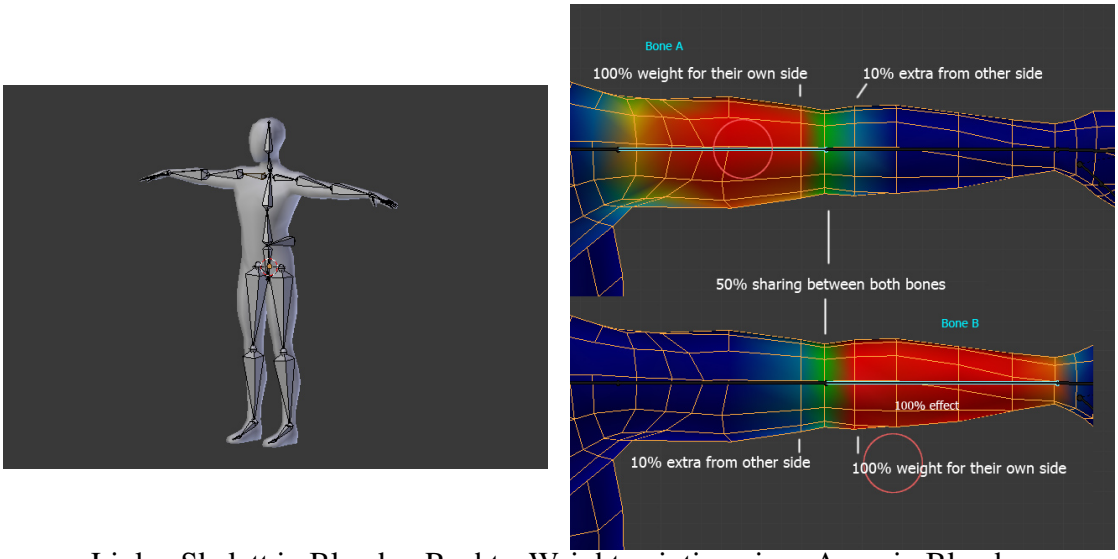
Eine Animation besitzt zu verschiedenen Zeitpunkten eine Position, Rotation und Skalierung für einen Knochen, genannt *Keyframe*. Dabei ist es nicht nötig, dass alle Knochen die gleiche Anzahl an Keyframes haben oder mehrere Keyframes an der gleichen Zeitposition existieren. Ein Keyframe muss nicht immer Position, Rotation und Skalierung auf einmal beinhalten, sondern nur eines oder zwei davon reichen aus. Die fehlenden Werte werden bei Bedarf aus anderen Keyframes ergänzt. Um den Zustand aller Knochen zu einem bestimmten Zeitpunkt herauszufinden, beginnt man am obersten Knochen der Hierarchie und arbeitet sich dann schrittweise nach unten, da die unteren Knochen von den oberen abhängig sind. Falls der Zustand zu diesem Zeitpunkt nicht durch die Keyframes genau definiert ist, muss zwischen den zwei am nächsten liegenden Keyframes interpoliert werden.

Für die Position und Skalierung kann zwischen den Keyframes linear interpoliert werden:

$t_0 \leq t \leq t_1 \qquad t_0 < t_1 \qquad$ Keyframes: $\overrightarrow{P_0}, \overrightarrow{P_1}$

$$f = \frac{t - t_0}{t_1 - t_0}$$

$$\overrightarrow{P} = ((\overrightarrow{P_0} \cdot (1 - f)) + (\overrightarrow{P_1} \cdot f)$$



Links: Skelett in Blender, Rechts: Weight painting eines Arms in Blender
Quelle: https://cgi.tutsplus.com/tutorials/building-a-basic-low-poly-character-rig-in-blender–cg-16955

Abbildung 8.: Blender Skelett

Für die Rotation muss Spherical Linear Interpolation (SLERP) [12] angewendet werden, dadurch bleibt die Kreisgeschwindigkeit über die Zeit konstant. Die Berechnung erfolgt für zwei Quaternionen, die eine Rotation repräsentieren (Quaternionen erweitern den Zahlenbereich der reelen Zahlen). $t_0 \leq t \leq t_1 \qquad t_0 < t_1 \qquad$ Keyframes: Q_0, Q_1

$$f = \frac{t - t_0}{t_1 - t_0}$$

$$\phi = Q_0 \cdot Q_1$$

$$\theta = \arccos(\phi) \cdot f$$

$$Q_2 = Q_1 - Q_0 \cdot \phi$$

$$Q = Q_0 \cdot \cos(\theta) + Q_2 \cdot \sin(\theta)$$

2.1.5. Projection Matrix

Eine Projektionsmatrix wird meistens verwendet um ein dreidimensionales Koordinatensystem auf eine Ebene zu projizieren, sie kann aber auch dafür verwendet werden um ein dreidimensionales Koordinatensystem in ein anderes dreidimensionales Koordinatensystem zu projizieren, was hier Anwendung findet. OpenGL arbeitet mit einem orthografischen Koordinatensystem, ein Koordinatensystem bei dem alle Achsen orthogonal zueinander sind. Alle Achsen haben Werte von -1 bis 1. Wenn man ein anderes Koordinatensystem verwenden will benötigt man eine Projektionsmatrix. Diese projiziert das gewünschte Koordinatensystem in das von OpenGL verwendete. Dabei unterscheidet man zwischen zwei Arten von Projektionsmatrizen: orthografische und perspektivische. Diese projizieren das jeweilige Koordinatensystem in das OpenGL-Koordinatensystem (siehe Abbildung 9).

Eine orthografische Projektionsmatrix ist nur eine Skalierungsmatrix, da der Raum in jede Achse nur so gestreckt oder gestaucht werden muss, dass die Begrenzungen -1 und 1 ergeben. Dabei wird der gesamte Raum nicht verformt. Daher ist die Berechnung dieser Matrix auch sehr einfach. Für ein Koordinatensystem mit den Begrenzungen l & r (links & rechts) an der X-Achse, t & b (top & bottom) an der Y-Achse und n & f (near & far) an der Z-Achse:

$$M_o = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{b-t} \\ 0 & 0 & \frac{2}{n-f} & \frac{f+n}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

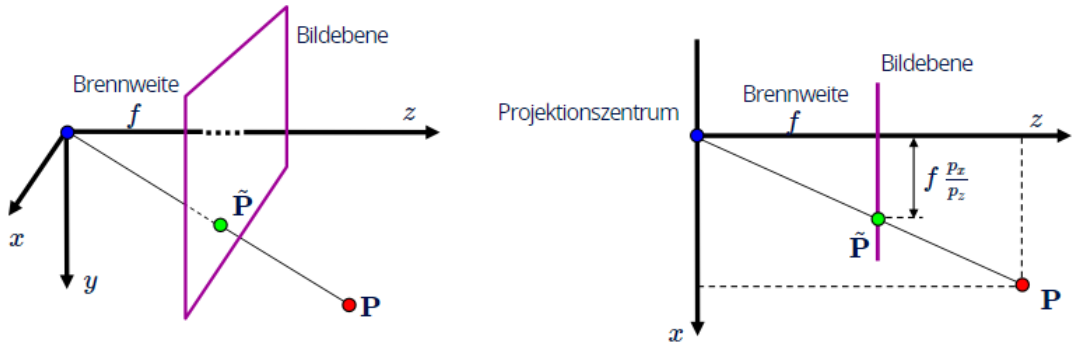
Eine perspektivische Projektionsmatrix projiziert eine perspektivisches Koordinatensystem in das OpenGL-Koordinatensystem. In einem perspektivischen Koordinatensystem verlaufen in einer Achsenrichtung alle parallele Geraden in diese Richtung zu genau einem Punkt, dem Fluchtpunkt. Ein bekanntes Beispiel dafür sind Schienen, die so aussehen als würden sie am Horizont zusammenlaufen. So nehmen wir die Welt war. Idealer weise wäre dieser Punkt unendlich weit weg. Da so eine Berechnung im Programm

nicht möglich ist, wird der Raum in die Ferne durch die *far plane* begrenzt. Für ein Koordinatensystem mit den Begrenzungen l & r (links & rechts) an der X-Achse, t & b (top & bottom) an der Y-Achse und n & f (near & far) an der Z-Achse:

$$M_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

(vgl. Songho's Webseite [13])

Der Unterschied zwischen dem near und far Wert sollte möglichst gering sein, da sonst Präzisionsprobleme im Depth-Buffer auftreten können, da dieser eine höhere Präzision in der Nähe der Kamera besitzt und zusätzlich mit Fließkommazahlen arbeitet, welche ebenfalls noch einmal eine höhere Präzision gegen 0 haben. Die Wahl dieser Werte ist dem User der Engine überlassen, aber es sollte immer der kleinste akzeptierbare Wert für die Szene sein. (vgl. Khronos Website [14])



Quelle: `http://www.songho.ca/opengl/gl_projectionmatrix.html` 14.01.2017

Abbildung 9.: Projektions Koordinatensysteme

2.2. Extensible Markup Language

Dieser Abschnitt basiert auf den Quellen [15][16][17]. Im Designer wird für das Abspeichern der Projektdatei die Extensible Markup Language (XML) verwendet. Diese *erweiterbare Auszeichnungssprache* wird im FM3D-Designer gebraucht, um die Projektdatei in hierarchisch strukturierten Daten in Textdateien zu speichern. XML beschreibt eine Baumstruktur von Daten, die hierarchisch angeordnet sind. Ein XML-Dokument besteht aus Elementen, Attributen, Verarbeitungsanweisungen und Kommentaren. Ein Element kann mittels einem Paar aus Start-Tag (zB. <*Tag*>) und End-Tag (zB. <*/Tag*>) oder einem leeren Tag (zB. <*Tag/*>) erfolgen. Diesen Elementen kann man nun beliebig viele Attribute zuordnen, welche wiederum Werte besitzen. Die Werte der Attribute sind standardmäßig in Form des Datentyps *String* angegeben. Dieser muss zunächst in den benötigten Datentyp umgewandelt werden. Als Beispiel wird nun das Dokument in Abbildung 10 verwendet. Dort erkennt man zunächst einen Verarbeitungshinweis, welcher die XML-Version und Zeichenkodierung spezifiziert. Das darauffolgende Element <*Wald*> besitzt das Unterelement <*Baum*>. Dieses Element besitzt das Attribut *name* mit dem Wert *Eiche*. Dieses Element <*Baum*> enthält weitere drei Unterelemente als leere Tags. Sie besitzen keine weiteren Unterelemente. Das Element <*Stamm*> besitzt das Attribut *LängeInMeter* mit dem Wert *„3“*. Möchte man diesen Wert als Integer in C# verwenden, so muss man diesen String erst in einen solchen Typ konvertieren. Des weiteren besitzt das Element <*Baum*> noch zwei Unterelemente mit dem Namen <*Blatt*>. Beide besitzen das Attribut *Farbe*. Auch hier liegt der Wert als String vor. In Abbildung 11 wird ein kommentiertes FM3D-Projekt dargestellt. Die Kommentare werden nicht in der Datei gespeichert und sind nur für die Veranschaulichung abgebildet.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Wald>
    <Baum name="Eiche">
        <Stamm LängeInMeter="3">
        <Blatt Farbe="Grün"/>
        <Blatt Farbe="Gelb"/>
    </Baum>
</Wald>
```

Abbildung 10.: XML-Beispiel
Quelle: Eigene Darstellung

```xml
<!--FM3D-Projekt Datei-->
<?xml version="1.0" encoding="utf-8"?>
<Project name="BOB">
    <!--Ordnerstruktur des Projektes-->
    <ProjectFiles name="ProjectFiles">
        <!--Ordner-->
        <Directory name="Entities">
            <!--Entity-Datei-->
            <EntityFile name="Baum.ent"/>
        </Directory>
        <Directory name="Models" >
            <!--Mesh-Datei-->
            <MeshFile name="Baum.fm_mesh"/>
        </Directory>
        <Directory name="Textures" />
    </ProjectFiles>
    <!--Daten für die Kommunikation mit der Extension via Pipe-->
    <CPlusPlus>
        <FM3D_File name="fm3d.xml" />
        <Solution name="GameProject.sln" />
    </CPlusPlus>
</Project>
```

Abbildung 11.: FM3D-Projektdatei
Quelle: Eigene Darstellung

3. Unser Programm

3.1. Engine - Allgemeiner Aufbau

Die Engine ist eine Visual C++ Laufzeitbibliothek (Dynamic-Link-Library; DLL). Diese DLL enthält den gesamten Code, welcher zum Erstellen eines Spieles erforderlich ist. Jason Gregory beschreibt in seinem Buch den Aufbau einer Game-Engine als eine Struktur aus Systemen und Schichten. Hierbei können mehrere Schichten übereinander liegen. Ideal verwenden die höher gelegenen Schichten bereitgestellte Funktionalitäten der unteren Schichten, was aber nie umgekehrt geschehen soll. Dies ermöglicht die Abschirmung von hardwareabhängigen und spielnahen Klassen. Ein System übernimmt eine bestimmte Aufgabe und kann sich über mehrere Schichten ausbreiten. [1] Die FM3D-Klasse hat diese Idee in bestimmten Teilen übernommen. Im Code ist die System-Struktur als Ordner-Struktur wiederzufinden. Diese Systeme sind teilweise auch durch einen *Namespace* oder Präfix vor jedem Namen zu erkennen. Das Schichtenmodell ist in der Engine als mehrere Untersysteme (auch Subsystems genannt) erkennbar. Diese Untersysteme sind genauso zu erkennen und verhalten sich wie normale Systeme. Der einzige Unterschied ist, dass sie sich bereits in einem System befinden.

3.2. Engine - Systeme

3.2.1. Math System

Das Math-System ist für sämtliche mathematischen Berechnungen zuständig. Dazu gehören sehr einfache Rechnungen wie Umwandlung von Grad in Bogenmaß, aber auch

komplexere Rechnungen wie zum Beispiel Matrix-Rechnungen. Das System besteht aus Klassen, aber auch einigen C-Style Funktionen, welche kleine Berechnungen ausführen, wie das Umwandeln von Grad in Bogenmaß. Alle Klassen sind Templateklassen, damit sie so flexibel wie möglich sind und für verschiedene Zahl-Datentypen verwendet werden können. Auf Vererbung wurde komplett verzichtet, da es wichtig ist, alle Berechnungen so schnell wie möglich auszuführen. Statt Vererbung wird daher Template-Spezialisierung verwendet. Dies hat zwar den Nachteil, dass einiger Code mehrfach geschrieben bzw. kopiert werden musste, aber dafür werden keine virtuellen Methoden verwendet und somit ist die Größe eines Objektes genau definiert. Dies ist wichtig, da einige Objekt in großer Anzahl in einem Buffer verwendet werden, in welchem alle Objekte im Speicher direkt aneinander liegen. Alle Klassen und Funktionen befinden sich im Namespace FM3D::Math, alle Typdefinitionen nur im Namespace FM3D für leichteren Zugriff.

Die zwei Grund Klassen sind *Vector* und *Matrix*. Sie besitzen jeweils einen Templateparameter für den Zahl-Datentyp der intern verwendet wird. Dazu besitzen sie Ganzzahltemplateparameter, welche die Größe bzw. die Dimension angeben. Die Klasse Vector kann für alle Dimensionen sämtliche mathematischen Grundoperationen und zusätzlich einige Operationen, welche mathematisch nicht möglich, aber im Programm nützlich sind (Im Beispiel Kursiv markiert), durchführen:

- Vektor-Addition $\quad \overrightarrow{a} + \overrightarrow{b} = \begin{pmatrix} a_0 + b_0 \\ a_1 + b_1 \\ \vdots \end{pmatrix} = \overrightarrow{c}$
- Vektor-Subtraktion $\quad \overrightarrow{a} - \overrightarrow{b} = \begin{pmatrix} a_0 - b_0 \\ a_1 - b_1 \\ \vdots \end{pmatrix} = \overrightarrow{c}$
- *Vektor-Multiplikation* $\quad \overrightarrow{a} \cdot \overrightarrow{b} = \begin{pmatrix} a_0 \cdot b_0 \\ a_1 \cdot b_1 \\ \vdots \end{pmatrix} = \overrightarrow{c}$
- *Vektor-Division* $\quad \frac{\overrightarrow{a}}{\overrightarrow{b}} = \begin{pmatrix} a_0 / b_0 \\ a_1 / b_1 \\ \vdots \end{pmatrix} = \overrightarrow{c}$

- *Skalar-Addition* $\quad \overrightarrow{a} + b = \begin{pmatrix} a_0 + b \\ a_1 + b \\ \vdots \end{pmatrix} = \overrightarrow{c}$
- *Skalar-Subtraktion* $\quad \overrightarrow{a} - b = \begin{pmatrix} a_0 - b \\ a_1 - b \\ \vdots \end{pmatrix} = \overrightarrow{c}$
- Skalar-Multiplikation $\quad \overrightarrow{a} \cdot b = \begin{pmatrix} a_0 \cdot b \\ a_1 \cdot b \\ \vdots \end{pmatrix} = \overrightarrow{c}$
- Skalar-Division $\quad \dfrac{\overrightarrow{a}}{b} = \begin{pmatrix} a_0/b \\ a_1/b \\ \vdots \end{pmatrix} = \overrightarrow{c}$
- Vektor-Produkt $\quad \overrightarrow{a} \cdot \overrightarrow{b} = a_0 b_0 + a_1 b_1 + \cdots = c$
- Länge $\quad |\overrightarrow{a}| = \sqrt{a_0^2 + a_1^2 + \ldots} = c$
- Normalisieren $\quad \dfrac{\overrightarrow{a}}{|\overrightarrow{a}|} = \overrightarrow{a}$
- Quadrierte Länge $\quad |\overrightarrow{a}|^2 = a_0^2 + a_1^2 + \cdots = c$

Diese Operationen sind als Methoden implementiert, bei denen das Objekt, welches die Methode ausführt, am Ende dem neuen Vektor entspricht, also $\overrightarrow{a} = \overrightarrow{c}$. Zurückgegeben wird eine Referenz auf das Objekt, damit die Methoden aneinander gekettet werden können. Außerdem sind die Operationen als statische Methoden implementiert, welche den Vektor $\overrightarrow{a}$ und $\overrightarrow{b}$ bzw. b als Parameter annehmen und ein neues Objekt zurück geben, ohne die Argumente zu verändern. Zusätzlich ist für jede Methode der entsprechende Operator als *inline* Methode bzw. *inline friend* Funktion implementiert. Dieser ruft einfach nur die Methode auf, ist aber in der Entwicklung übersichtlicher.

Für Vektoren mit der Dimension zwei, drei und vier gibt es jeweils eine Spezialisierung der Template-Klasse. Das hat den Vorteil, dass die Member-Variablen richtige Namen habe, und somit keine For-Schleifen verwendet werden müssen und die Klasse besitzt Methoden, welche nur für diese Dimension Anwendung finden. Dies sind zum Beispiel statische Methoden für die Koordinatenachsen oder das Kreuzprodukt für einen 3D-

Vektor.

Die Matrix-Klasse verhält sich ähnlich wie die Vector-Klasse. Alle Elemente werden in einem Array gespeichert. Sie werden Reihe für Reihe gespeichert, was die folgenden Indices ergibt:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

Die Klasse kann die folgenden Operationen ausführen:

- Matrix-Multiplikation
- Matrix-Addition
- Skalar-Multiplikation
- Vektor-Multiplikation

Sie sind wie bei der Vector-Klasse als Methoden und Operatoren implementiert.

Es gibt zwei verschiedene Matrix-Spezialisierungen: Eine 2x2 Matrix und 4x4 Matrix. Diese Klassen besitzen zusätzlich statische Methoden, um spezielle Matrizen zu erstellen. Die 2x2 Matrix kann eine Rotationsmatrix für 2D-Vektoren erstellen, die 4x4 Matrix verschiedene Transformationsmatrizen für 3D-Vektoren:

- Projektionsmatrix — Siehe Abschnitt 2.1.5
- Translationsmatrix — Verschiebung für $\overrightarrow{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ $M = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- Rotationsmatrix — Berechnung nach [18]
- Skalierungsmatrix — Faktoren: x, y, z $M = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Bei der Multiplikation mit einem 3D-Vektor wird angenommen, dass die 4. Komponente des Vektors, welche nicht vorhanden ist, 1 beträgt. Dadurch ist es möglich Translationen abzubilden. Zusätzlich kann die 4x4 Matrix invertiert und transponiert werden.

Damit nicht immer der volle Klassenname mit Templateargumenten ausgeschrieben werden muss, gibt es einige Typdefinitionen. Diese sind folgendermaßen aufgebaut:
„Vector“ + *Dimension* + *Abkürzung des Datentyps*
„Matrix“ + *Anzahl Reihen* + *Anzahl Spalten* + *Abkürzung des Datentyps*
Für quadratische: „Matrix“ + *Anzahl Reihen* + *Abkürzung des Datentyps*
„Quaternion“ + *Abkürzung des Datentyps*
Zum Beispiel einen dreistelligen Vektor für Float:
Vector3f. Zusätzlich gibt es noch Typdefinitionen für Farben. Diese sind nur eine andere Schreibweise eines Vektors bei der „Vector“ mit „Color“ ersetzt wird.

3.2.2. File System

Um Dateien verschiedener Formate in der FM3D-Engine verwenden zu können, wird die Klasse *ExternFileManager* verwendet. Die *ExternFileManager* Klasse besitzt Methoden um Schriftarten, Bilder und Modelle zu laden. Der Methode *ReadFontFile*, um die Schriftarten zu, laden übergibt man eine Referenz zu dem Namen der Datei. Zudem gibt man die Größe und die Skalierung der Schriftart an. Um die Textur auch in einem RenderSystem darzustellen, benötigt die Methode auch eine Referenz zu einem existierenden Objekt der *RenderSystem*-Klasse. Die Methode benötigt weiterhin eine doppelte Referenz zu einem bereits erstelltes Objekt der Klasse *Font*. Die Methode *ReadTextureFile* gibt eine Referenz zu einem Objekt der Klasse *Texture* zurück. Als Übergabeparameter benötigt diese Methode zunächst das Render-System, in dem die Textur gerendert werden soll. Außerdem benötigt diese Methode noch die Enumerationen vom Typen *FilterMode*, *WrapMode* und *MipMapMode*, welche in der Klasse *Textur* stehen. Alle dieser Enumerationen besitzen bereits in dieser Methode einen Standardwert und müssen nicht explizit angegeben werden. Zur Erläuterung dieser Enumerationen siehe den Abschnitt 3.2.3. Die Methode *ReadModelFile* benötigt als Übergabeparameter den Dateinamen, das Render-System, in dem das Model geladen werden soll. Ein boolescher Übergabeparameter gibt an, ob *Instancing* verwendet werden soll. Der darauffolgende

boolescher Wert gibt an, ob eine Animation in dem Model verwendet werden soll. Eine vierstellige Matrix *Matrix4f* gibt die Skalierung des Models an. Zur Initialisierung wird die Methode *Initialize* verwendet.

3.2.3. Graphic System

RenderSystem

Um in diesem Fenster nun zu rendern, benötigt man ein Objekt der Klasse RenderSystem. Das RenderSystem erstellt die Objekte der Grafikklassen je nachdem welche OpenGL Version man verwendet.

Font-Klasse

Objekte der Font-Klasse repräsentieren Schriftarten, welche man im Programm rendern kann.

Texture-Klasse

Die Klasse besitzt die Enumerationen *FilterMode*, *WrapMode* und *MipMapMode*. In *FilterMode* gibt verschiedene Filtermodi an, in denen die Textur gerendert wird. Wenn verschiedene nebenstehende Pixel auf der Textur nicht auf den Bildschirm „passen“ bzw. aufgrund der Transformation nicht parallel nebeneinander gerendert werden können, muss zwischen den beiden Modi NEAREST und LINEAR auswählen werden. Der Modus LINEAR bildet eine Mischung der nebenstehenden Pixel. Im Gegensatz zu LINEAR bildet der Modus NEAREST harte Kanten der gerenderten Textur. Die Enumeration *WrapMode* gibt an, wie die Textur auf das Modell gerendert werden soll und *MipMap* gibt an wie die verschieden MipMaps, bzw. verschiedenen

bei der erstelllung Linear verrechnet werden soll

TODO!

Möchte man eine Textur für OpenGL erstellen, so verwendet man dafür die Klasse GL3Texture, welche von der Klasse Texture erbt. Diese Klasse beschreibt eine Textur, wie sie in OpenGL verwendet wird.

Model-Klasse

Die Klasse *Model* besitzt alle Eigenschaften, die auch jedes andere Model besitzt. Darunter fallen ein Skelett, ein boolescher Wert, der angibt, ob Instancing verwendet werden soll und mehrere Teile des Models.

Der Begriff Instancing beschreibt das Folgende: Nehmen wir an, man rendert eine Szene mit einer Vielzahl an Modellen, welche eine Menge gleicher Vertices-Sätzen besitzen, aber eine andere Transformation haben. Als Beispiel stellen wir uns einen zu rendernden Baum vor. Dieser Baum besitzt hunderte Blätter, welche alle gleich modelliert sind. Würde man ein einzelnes Blatt rendern, so würde dies sehr schnell verarbeitet werden. Aber die ganzen Render-Aufrufe verlangsamen das Programm um ein Vielfaches. Bei Instancing werden nun verschiedene Instanzen dieses Blattes gerendert. Dies spart eine Menge Zeit und lässt das Programm wesentlich schneller laufen. Die Klasse *Animated-Model* erbt von der Klasse Model und beschreibt ein Model, welches animiert wird.

Material-Klasse

Um ein Modell zu rendern, muss man ihm ein Material zuweisen. Jedes Model besteht aus verschiedenen Materialien, welche von der Klasse Material beschrieben werden. So kann man verschiedene Objekten die gleichen Materialien zuweisen. Möchte man zum Beispiel einen Tisch und einen Stuhl aus Holz rendern, so könnte man ein Material *Holz* mit einer Textur, die *Holz* abbildet erstellen. Dieses Material könnte man sowohl dem Tisch als auch dem Stuhl zuweisen.

3.2.4. Entity System

Bevor das Entity-System erläutert wird, muss erst geklärt werden, was ein Entity bzw. eine Entität überhaupt ist. Die Verwendung von Entities in der Engine werden im folgenden erläutert. Peter Dr. Chen, welcher das Entity-Relationship-Model in den Jahren 1970 bis 1976 entwickelte, definiert eine Entität als folgende:

> ([...] Eine Entität ist ein „Ding“, welche deutlich unterschieden werden kann. Als Beispiel für eine Entität kann zB. eine spezifische Person, Firma oder ein Event betrachtet werden. [...]) ([19], Zitat aus dem Englischen übersetzt)

Die FM3D-Engine verwendet ein *Entity-Component-System*, um Objekte eines Spieles zu verwalten. Im Gegensatz zu einem vererbungs-basierten Entity-System ist dieses sehr flexibel. Die Grundidee besteht darin, die *Daten* und *Logik* eines Entities aufzuteilen. Hierfür werden alle Daten eines Entities (alle Attribute eines Entities) in Komponenten geschrieben. Die Logik (Methoden, die das Entity ausführen soll) wird in eine EntityLogic-Klasse geschrieben. Ein Entity kann beliebig viele verschiedene Komponenten besitzen und sie können während der Laufzeit beliebig hinzugefügt oder entfernt werden. Jedoch kann ein Entity immer nur einen Komponente eines Typs enthalten. Der Manager führt dann bei jedem Update oder bei bestimmten Events eine Methode, das die vom Manager benötigten Komponenten enthält, für jedes Entity aus.

Alle Entities werden in einer *Entity-Collection* gespeichert. Diese ist eine Sammlung von Entities, welche dafür zuständig ist, neue Entities zu erstellen und alte zu löschen. Wenn ein Entity zerstört wird, so wird dieses nicht aus dem Speicher gelöscht. Es bleibt weiterhin in der Entity-Collection gespeichert. Wenn ein neues Entity erstellt wird, muss nicht neuer Speicher angefragt werden und das bereits vorhandene aber „gelöschte“ Entity kann nun wieder verwendet werden. Dies ermöglicht es effizienter eine hohe Anzahl von Entities zu erstellen und wieder zu löschen. Mit Komponenten verhält es sich genauso. In der *Entity-Collection* werden alle zerstörten Komponenten gespeichert. Diese werden solange gespeichert, bis eine neue Komponente des gleichen Typs erstellt wird. Das gesamte Entity-System, befindet sich im Namespace *EntitySystem* um es vom restlichen Code abzutrennen.

Bevor wir auf die wichtigsten Klassen des Entity-Systems eingehen, müssen erst einige Hilfsklassen erläutert werden. Da es standardmäßig in C++ keine Events gibt, anders als in C#, besitzt die FM3D-Engine ihre eigene Event-Klasse. Ein Objekt der Event-Klasse beschreibt die Eventquelle. Sie speichert alle Funktionen, die aufgerufen werden sollen, wenn das Event aktiviert wird. Als Vorbild wurden die Events in C# genommen. Mit den Operatoren += bzw. -= ist es möglich, neue Funktionen hinzuzufügen. Mit dem ()-Operator wird das Event aktiviert. Dabei werden die Argumente direkt an die gespeicherten Funktionen weitergegeben. Zurückgegeben wird ein Vektor mit den Rückgabewerten jeder Funktion, es sei denn der Rückgabewert ist *void*. Um die Funktionen aufzurufen, wird eine statische Hilfsklasse *Invoker* verwendet. Diese hat eine Template-Methode, um die Funktionen aufzurufen und eine Spezialisierung für den Rückgabewert *void*. Die Event-Klasse ist Multithread-sicher, da intern ein *Mutex* verwendet wird, um den Zugriff auf die Threads zu regeln. Dies ermöglicht ein unkompliziertes Verwenden im Entitysystem.

Wenn man auf alle Entities mit bestimmten Komponenten zugreifen will, kann die Klasse Group verwendet werden. Diese wird mit einer EntityCollection erstellt und speichert alle Entities mit bestimmten Bedingungen. Welche Entities diese Bedingungen erfüllen, wird mit der Klasse *Matcher* entschieden. Diese speichert die Komponenten, welche ein Entity besitzen müssen, welche es nicht besitzen darf und von welchen es mindestens eines enthalten muss. Mit diesen Informationen wird dann bestimmt, ob das Entity zur Group gehört oder nicht. Die Group updatet sich mit Hilfe von Events automatisch, wenn sich ein Entity verändert oder neues erstellt wird.

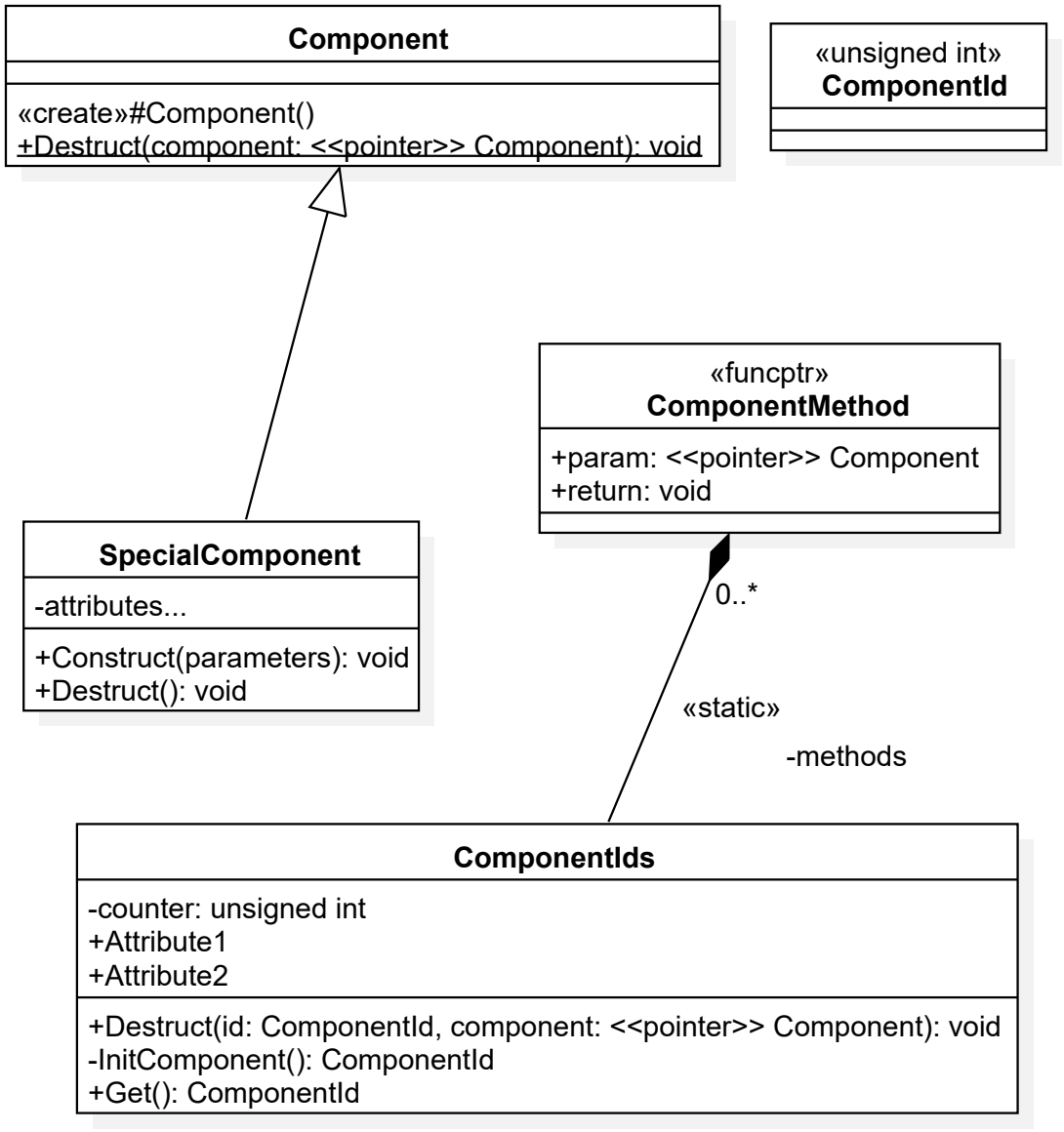
Dies wird im Code durch die folgende Klassenstruktur ermöglicht. Die Hauptklasse Entity besitzt einen Container mit Komponenten, die nach ihrer ID sortiert sind. Um auf eine Komponente zuzugreifen, benötigt man nur die ID in Form eines 32-Bit Integers. Auf diese IDs kann man mit der Hilfsklasse *ComponentIds* zugreifen. Diese Klasse enthält eine statische Variable, die jedes Mal bei einer neuen ID erhöht wird. Auf die ID kann dann mit der statischen Template-Methode Get() zugegriffen werden.

Um ein Entity zu erstellen, benötigt man ein Objekt der Klasse *EntityCollection*. Mit der Entity-Klasse selbst kann kein Objekt erstellt werden. Die *EntityCollection* enthält eine *Map* mit bereits gelöschten Komponenten, um diese wiederverwenden zu können. Alle

Komponenten müssen nach ihrer ID sortiert werden, da nur Komponenten, die mit dem neuen Typ übereinstimmen und daher die gleiche Speichergröße und Variablen haben, für den neuen Komponenten verwendet werden können.

Möchte man eine neue Komponente erstellen, so muss man eine Klasse erstellen, die von der Klasse Component erbt. Die Component-Klasse ist eine leere Klasse und dient nur dazu, verschiedene Komponenten auf eine allgemeine Weise zu speichern. Wenn eine neue Komponente erstellt wird und nicht unbedingt der Konstruktor aufgerufen wird, so muss jede Klasse, die von Component erbt, die Methode Construct mit beliebigen Parametern enthalten, sowie die Methode Destruct ohne Parameter. Da Vererbung in diesem Fall einen großen Geschwindigkeitsverlust bewirken würde, muss die *Construct*-Methode mit Hilfe von Templates aufgerufen werden. Bei der *Destruct*-Methode ist dies leider nicht so einfach möglich. Da der Datentyp zum Zeitpunkt der Zerstörung nicht mehr bekannt ist, muss ein Funktions-Pointer für jeden Komponenten-Typ gespeichert werden. Dieser zeigt auf eine Funktion, welche einen *Component*-Pointer erst zu dem spezifischen Komponenten-Pointer casted und dann die Destruct-Methode aufruft. Diese Funktion ist eine statische Template-Methode in der Component-Klasse. Mit dem Template-Parameter ist es möglich diese Methode für verschiedene Komponenten zu verwenden. Der Pointer wird in einem Objekt der Klasse *EntityIds* gespeichert und kann verwendet werden, indem die statische Methode *Destruct* aufgerufen wird, der sowohl die Komponente, als auch die ID übergeben wird. Er wird beim erstmaligen aufrufen der Methode Get() für jeden Komponent-Typ mit der Methode InitComponent() erstellt. Diese Klassen sind in Abbildung 12 dargestellt.

Die Logik wird in einer Klasse, welche von *EntityLogic* erbt, implementiert. EntityLogic ist eine abstrakte Klasse, die eine abstrakte Execute-Methode enthält. Diese wird für jedes Entity einmal ausgeführt. Eine Subklasse soll diese Methode überschreiben und damit die Logik für bestimmte Entities darstellen. Die Klasse enthält eine Group, welche alle Entities enthält für die die Logik angewandt werden soll. Im Konstruktor wird daher ein Matcher übergeben.



Die Klasse SpecialComponent ist eine Beispiel-Klasse für einen benutzererstellten Komponenten. Hierbei ist attributes für tatsächliche Variablen auszutauschen und parameters für die Parameter, die der Komponent benötigt um initialisiert zu werden. Es können gegebenenfalls auch mehrere Überladungen der Construct-Methode erstellt werden.

Abbildung 12.: Component-Klassen

3.2.5. Window-System

Fenster

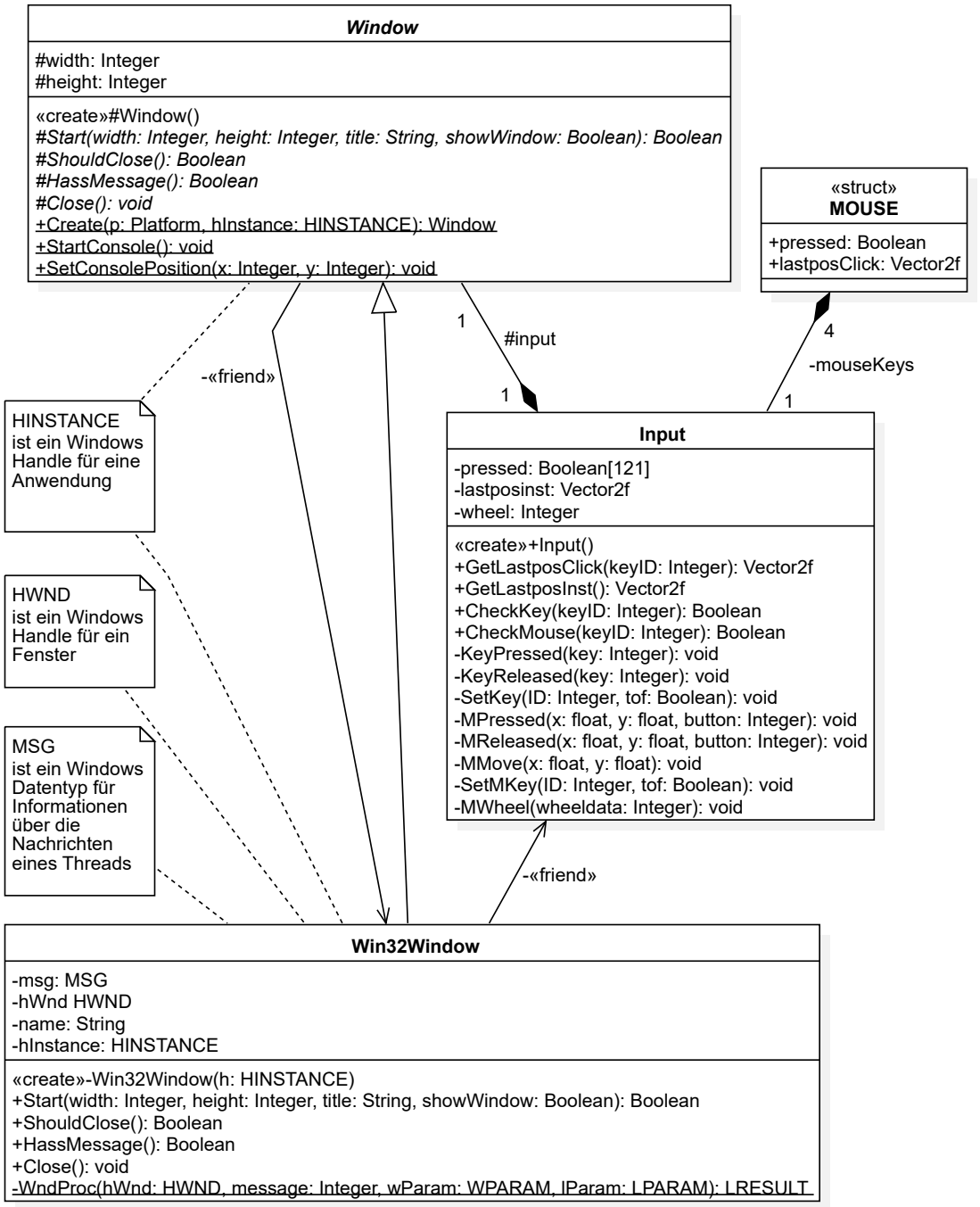
Ein Spiel *„passiert“* in unserem heutigen Zeitalter wie jedes andere GUI in einem *Fenster*. Um ein solches zu erstellen, wurden in der FM3D-Engine die Klassen *Window* und *Win32Window* implementiert. Window ist die Basisklasse, von welcher alle Fenster-Typen erben sollen. Die Klasse besitzt die Attribute *width* und *heigth* vom Typ Integer, welche die Breite und Höhe eines Fensters darstellen. Es existieren ein Konstruktor und verschiedene Methoden, die die Interaktion mit einem Fenster ermöglichen. Alles erwähnte ist *protected*. *Start* startet unter Angabe der Parameter, welche die Breite, Höhe, den Fenster Titel, und die Sichtbarkeit des Fensters angeben, das bereits erstellte Fenster. Die Methode *HasMessage* gibt an ob das Fenster eine Message hat. Shouldclose gibt zurück, ob das Fenster geschlossen werden soll. Die Methode *Close()* schließt ein Fenster. Mit der Methode *Create()* erstellt man ein neues Fenster mit den Übergabeparametern der Plattform, auf welcher das Spiel bzw. die GUI laufen soll und einem Objekt der Klasse HINSTANCE, welches ein Handle für Windows-Anwendungen ist. Die Methode *StartConsole()* startet eine Konsole für eventuelle Fehlerdiagnosen oder ähnlichem. Mit der Methode *SetConsolePosition* kann man mit Übergabeparametern X und Y vom Typ Integer, die die Koordinaten auf dem Bildschirm beschreiben, die Position der Konsole auf dem Bildschirm setzen. Jedes Window-Objekt besitzt ein Objekt der Klasse Input, doch dazu später mehr. Die Basisklasse *Window* hat den Vorteil, dass man verschiedene Fensterklassen für verschiedene Betriebssysteme implementieren könnte. In der aktuellen Version der Engine existiert nur eine Fensterklasse für Windows32 die von *Window* erbt. Dies ist die Klasse *Win32Window*. Die Klasse Windows besitzt zudem noch einige *Get*- und *Set*-Methoden, wie Größe und Position des Fensters auf dem Bildschirm.

Wie schon erwähnt, erbt die Klasse *Windows32Window* von der Klasse *Window* und die Klasse *Window* steht zu *Win32Window* in einer *friend* Beziehung. *Win32Window* besitzt die Attribute *msg* vom Typ *MSG*, *hWnd* vom Typ *HWND*, *hInstance* vom Typ *HINSTANCE* und einen String der den Namen des Fensters beschreibt. HWND ist ein Windows-Handle für Fenster und *MSG* ein Windows-Datentyp für Informationen über

die Nachrichten eines Threads. Die Klasse besitzt zudem einen Konstruktor, dem man ein Objekt der Klasse *HINSTANCE* übergibt, vier *public* Methoden und eine *private* Methode. Die Methode *Start* startet das Fenster und benötigt als Übergabeparameter die Höhe, die Breite, den Titel des Fensters und einen boolschen Wert, der angibt, ob das Fenster angezeigt werden soll, oder nicht. Auch hier gibt die Methode HasMessage an, ob das Fenster eine Message hat und Shouldclose gibt zurück, ob das Fenster geschlossen werden soll. Nur sind diese Methoden auf Windows32 zugeschnitten und verwenden unteranderem auch die Methoden aus der Klasse Window. *Close* schließt das Fenster. (für weitere Informationen Siehe Abbildung 13)

Input-Klasse

Natürlich benötigt ein Spiel bzw. auch die meisten grafischen Nutzer-Oberflächen auch ein sicheres Eingabe-System, welches sowohl die Position der Maus ermittelt, aber auch die Abfrage jeder einzelnen Taste auf der Tastatur regelt. Dafür wurde in der FM3D-Engine eine eigene Klasse implementiert, welche für diesen Aufgabenbereich zuständig ist. Um den Nutzer der FM3D-Engine davor zu bewahren, nicht jeden einzelnen ASCII-Code jeder Taste auf der Tastatur nachzuschlagen, sind alle Tasten in Form von Makros definiert. In dem Namespace FM3D befindet sich die Klasse *Input* und jedes Objekt eines Fensters bzw. der Klasse *Window* besitzt ein Objekt dieser Klasse. Die Klasse *Input* beinhaltet außerdem die Enumeration *KEYCLICK*, welches den aktuellen Zustand der Maustasten beschreibt. Diese können entweder gedrückt, nicht gedrückt oder losgelassen werden und sind in dieser Enumeration definiert. Eine Struktur *MOUSE* enthält die Enumeration *KEYCLICK* und einen zweidimensionalen Vektor des Typen *Float*, welcher die zweidimensionale Position im Fenster in OpenGL Koordinaten angibt, an welcher eine Taste der Maus gedrückt wurde. Diese Struktur wird in der Klasse Input als ein vier-Felder Array verwendet, welches jede Taste einer durchschnittlichen Maus beschreibt. Diese sind die linke, die rechte und die mittlere Maustaste. Das letzte Feld ist für ein zusätzliches Feld einer beliebigen Taste der Maus reserviert, da es immer unterschiedliche Maus-Typen gibt. Zudem besitzt die Klasse *Input* einen Vektor *lastposinst*, welcher ununterbrochen die Position der Maus ermittelt. Dies ist in den Spielen erforderlich in denen die Maus ununterbrochen die Kamera steuert.



Es wurde auf die Angabe von Get- und Set-Methoden sowie Events verzichtet.

Abbildung 13.: Window-System

Für die Tastenabfrage wurde zunächst ein Integer Wert verwendet, in welchen der aktuell gedrückte ASCII-Code gespeichert wurde. Dies hat aber den Nachteil, dass nur eine Taste gedrückt sein bzw. abgefragt werden darf. Möchte man nun zum Beispiel einen Spiele-Charakter schräg durch einen Raum mit gedrückter Links- **und** Rechts-Taste bewegen, so wäre hiermit nicht möglich. Deswegen beschreibt nun ein 121-Felder Array aus booleschen Werten die komplette Tastatur. Jede Feldnummer ist äquivalent zu dem zugeordneten ASCII-Code der Taste. Die Felder [1] bis [4] beschreiben die Maustasten. Einige Felder des Arrays sind nicht von einem ASCII Code auf der Tastatur *belegt* und sind somit „*frei*“. Dies hat den Vorteil, dass später simpel weitere Eingabegeräte in die Klasse Input eingebunden werden können. Die Klasse besitzt Methoden, welche in der Klasse Win32Window.h (Siehe Doxygen Dokumentation), die ein GUI-Fenster abbildet, die die verschiedenen Werte der Arrays und des Vektors setzen. Zudem besitzt die Klasse Methoden, die diese Werte zurück geben. (Für die Verwendung dieser Klasse siehe Abschnitt 4.3.4)

3.3. DesignerLib - Allgemeiner Aufbau

Die DesignerLib ist eine Bibliothek, welche vom Designer verwendet wird. Sie ist in der Programmiersprache C++ mit Common Language Runtime (CLR)-Kompatibilität geschrieben. Dies ermöglicht es Native C++-Code zusammen mit .Net-Code zu verwenden. Alle Klassen der Bibliothek befinden sich im Namespace DesignerLib. Die Bibliothek verwendet die FM3D-Engine als Bibliothek. Da in der Engine einige Klassen, die nicht mit CLR kompatibel sind, verwendet werden wie *std::mutex*, muss öfters eine Zwischenklasse erstellt werden ohne CLR-Kompatibilität. Diese enthält einen Pointer zu der FM3D-Klasse und wird in einer Managed-Klasse verwendet.

Die Designerlib bildet eine Schnittstelle zwischen der Engine und dem Designer. Sie enthält Klassen zum Rendern eines Meshes, damit dieses im Designer angezeigt werden kann und auch Klassen, welche die verschiedenen Ressourcen repräsentieren. Diese sind dafür zuständig das externe Dateien geladen und umgewandelt werden, genauso wie das Laden und Speichern in eigene Dateiformate.

3.4. Designer - Allgemeiner Aufbau

Der *Designer* ist ein Tool, um die Programmierung mit der Engine zu erleichtern. Er generiert über eine sogenannte *Pipe*, die zu der Kommunikation zwischen dem Designer und der Entwicklungsumgebung *VisualStudio2015* dient, um Code auf den Einstellungen des Users basierend zu generieren. Der Designer ist fähig ein volles funktionsfähiges Grundgerüst für ein Spiel zu generieren. So wird dem Nutzer des Designers viel Schreibarbeit gespart und er kann sich so den gewünschten Funktionen seines Spieles widmen. Dieses eben genannte Grundgerüst besteht aus renderbaren Scenen mit den zugehörigen Entities3.2.4, welche mit der *Pipe* in ein VisualStudio-Projekt hinzugefügt werden. Die Verwendung des Designers wird in dem folgenden Kapitel 4.2 erklärt. Es wird im folgenden nur auf den Programmaufbau eingegangen. Mit dem Designer können alle Projekte zwischengespeichert werden. Sowohl die erstellten Projektdateien mit den Pfaden zu hinzugefügten Dateien und den Einstellungen, aber auch die verschiedenen Scenen und Entity-Presets können gesichert werden. Dies hat den Vorteil, dass man Entities oder Scenen aus alten Projekten einfach in neuen verwenden kann. Zudem werden die Texturen für Modelle mit dem Designer in ein Format konvertiert, welches für die Engine einfacher zu verarbeiten ist, als herkömmliche Bilddateien. Der Designer arbeitet mit eigens für ihn entwickelten Dateiformaten. Die meisten sind in Form einer .xml Datei (Extensible Markup Language Abschnitt 2.2) auffindbar, dennoch werden verschiedene Endungen verwendet, um unterschiedliche Datentypen von anderen zu trennen. Im folgenden werden einzelne ausschlaggebende Funktionen erklärt.

3.5. GUI-Fenster

Der Designer verfügt im allgemeinen über drei Arten, wie Seiten der GUI im Designer dargestellt werden können. Das Programm besitzt ein Hauptfenster, welches das Design von MahApps.Metro verwendet. (Siehe Abschnitt 3.7.4, Zum besseren Verständnis der Seitenarten siehe Abbildung 14) Dieses Hauptfenster kann nun in verschiedene Reiter unterteilt werden. Dies wurde wir mit der GUI Bibliothek DotNetBar entwickelt (Siehe Abschnitt 3.7.3). Diese Reiter sind im Designer sogenannte *Layouts*. Jeder der

verschiedenen Layoutklassen steht im Namespace *FM3D_Designer.src.WindowLayouts* und erbt von der Klasse *WindowLayout* im Namespace *FM3D_Designer.src*. Diese Klasse beinhaltet Attribute, die jedes der Layouts verfügt. Die Klasse WindowLayout, welche wiederum von der Klasse *DockWindow* der DotNetBar-Bibliothek erbt, sorgt dafür, dass die verschiedenen Seiten als Reiter und Unterfenster (oder auch child-windows) behandelt werden. In jedem Layout ist es möglich verschiedene kleinere Fenster aufzurufen. Der User kann diese Fenster innerhalb der Layouts oder des Desktops andocken. Jedes dieser *ToolWindows* ist der Strukturierung wegen nochmal in einen weiteren Namespace unterteilt. Alle dieser *ToolWindows* erben von der Klasse *ToolWindow*, welche eine Initialisierungsmethode und eine Aggregation zu einem *WindowLayout* verfügt. Auch die Klasse *ToolWindow* erbt von der Klasse *DockWindows*. Neben diesen Reitern und Unterfenstern verwendet der Designer noch sogenannte *Dialoge*. All diese Dialoge erben von der Klasse *DialogBase*, welche die Grundkomponenten der Dialoge beinhaltet. Diese Klasse erbt wiederum von *BaseMetroDialog* welche sich in der MahApps.Metro Bibliothek befindet.

3.5.1. Layouts

StartLayout

Das Start-Layout des Designers bietet dem Nutzer die Option ein Projekt in den Designer zu laden oder ein neues zu erstellen. Möchte man nun eines erzeugen, so wird man auf einen neuen Reiter geleitet, in dem man nun Einstellungen bezüglich des zu erstellenden Projektes tätigen kann. Rechts davon befindet sich ein Text mit einer kurzen Info zu dem Programm. Darunter ist ein Internet-Browser eingebunden, welcher die letzten Änderungen des Programmes anzeigt.

CreateProjekt

Nachdem man auf einen Nebenreiter des Startlayouts geleitet wurde, kann der User einen Pfad bestimmen, in dem das Projekt geladen werden soll. Im vom User angegebenen Pfad werden jetzt zwei Ordner angelegt: einen für die Projektdateien und

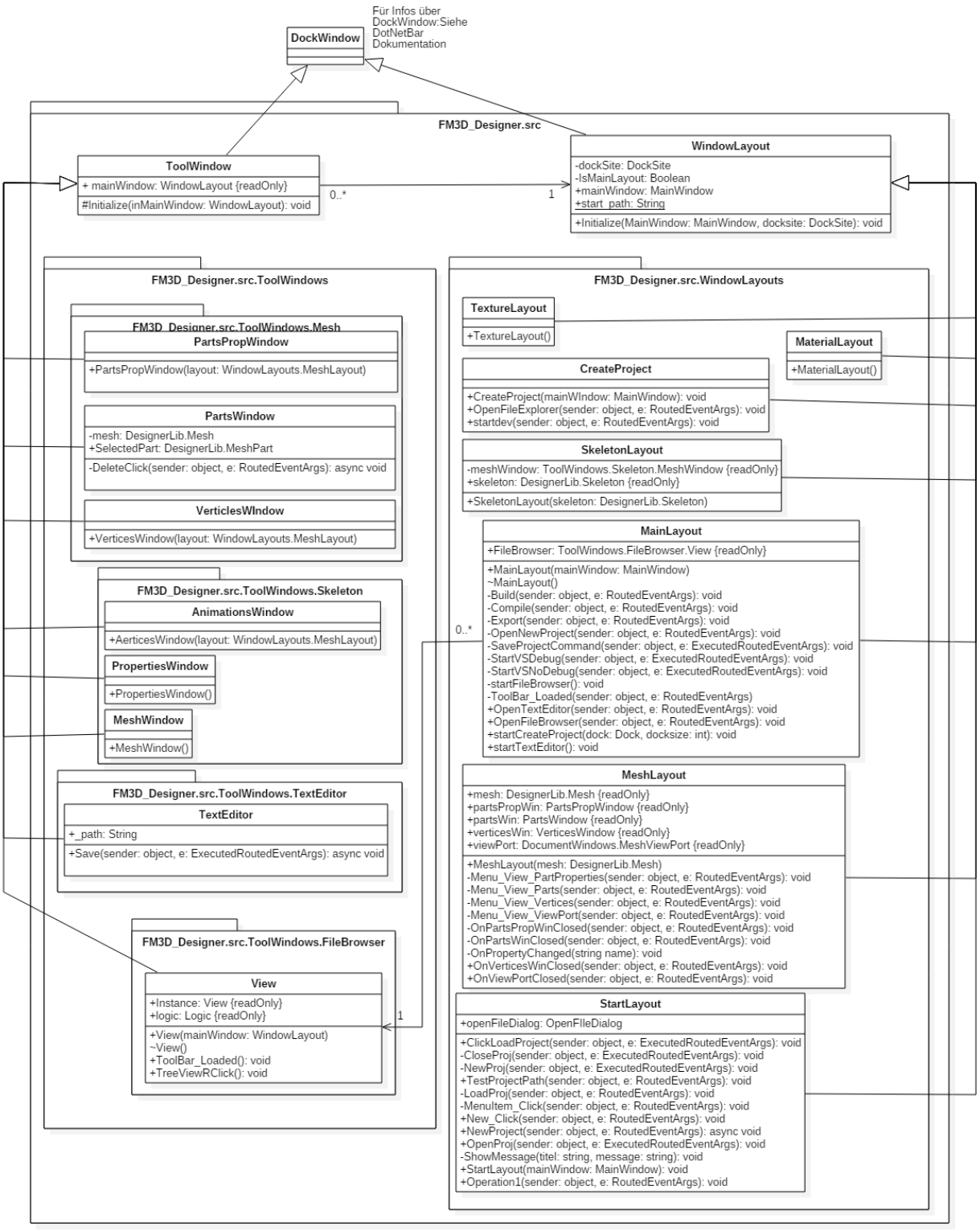


Abbildung 14.: Fenster-Klassen

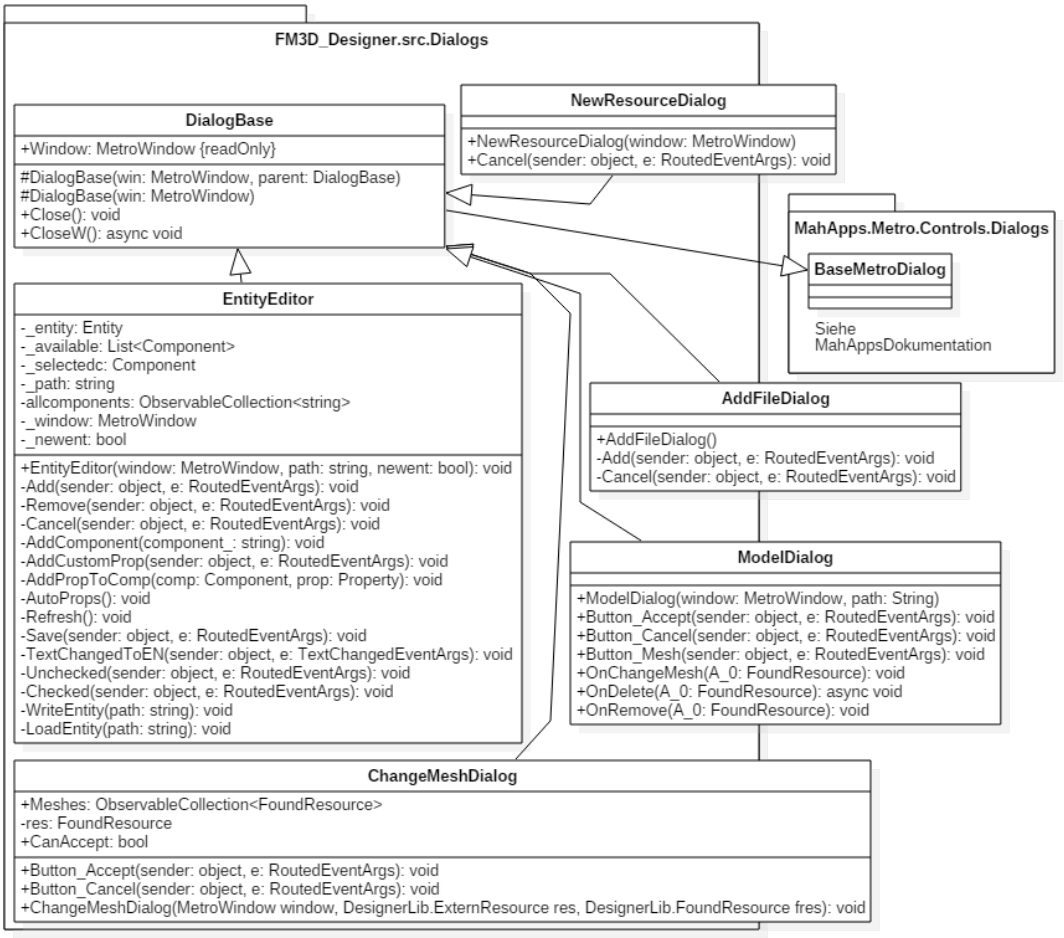


Abbildung 15.: Dialog-Klassen

einen für die C++ Dateien. In den Ordner der C++ Dateien wird nun ein VisualStudio-Projekt-Template erstellt, welches ein Projekt für das Spiel abbildet. Zu diesem wird eine „fm3D.xml“ Datei in dem gleichen Ordner generiert. In dieser stehen Informationen, welche später für die Pipe 3.6 benötigt werden. Darunter der Dateiname, der Projektmappenname und die Pipe-ID. Nun wird die Projektdatei in Form von einer XML-Datei mit der Endung „.fmproj“ erzeugt. Diese wird beim Start der FM3D-Designer-Entwicklungsumgebung geladen und später in der *TreeView* des *File-Browsers* dargestellt.

MainLayout

Nachdem nun ein Projekt erstellt oder geöffnet und geladen wurde, wird der User auf das „Main-Layout“ geleitet. Dort werden mit dem Start ein Texteditor und ein File-Browser in andockbaren Childfenstern geöffnet. In dem Kontextmenü des Programmes kann der User nun zwischen vier verschiedenen Reitern auswählen. Neben jeder Option werden zudem -falls verfügbar- Tastenkürzel angezeigt. In dem ersten Reiter „File“ kann der User entweder ein neues Projekt erstellen oder das aktuelle speichern. Im zweiten Kontextmenü-Item sind die Operationen aufrufbar, welche für die Kommunikation mit der Extension (Abschnitt 3.6) über die Pipe(Abschnitt 3.6) benötigt werden. Diese Optionen sind auch in Icons unter dem Kontextmenü abgebildet, um dem User ein schnelles arbeiten zu ermöglichen. Im dritten Kontextmenü-Item kann man nun die verschiedenen andockbaren Fenster auswählen. Im vierten und letzten sind Informationen zum Entwicklerteam und über dieses Projekt zu finden.

MeshLayout

Das MeshLayout wird verwendet, um ein geöffnetes Mesh anzuzeigen. Dabei verwendetet es eine einfache Menüleiste, die es erlaubt, verschiedene Toolwindows zu öffnen oder in den Vordergrund zu holen und das geladene Mesh zu speichern. Im MeshLayout ist das Mesh gespeichert und die Mesh-Toolwindows greifen über diese Klasse darauf zu, genauso wie auf andere Toolwindows. Zum Speichern werden die Methoden der DesignerLib verwendet.

3.5.2. ToolWindows

FileBrowser

Im File-Browser wird die Ordnerstruktur des FM3D-Projektes in Form der *Item*-Klasse (Siehe Abschnitt 3.6) abgebildet. Der Nutzer kann durch diesen File-Browser nun entweder Daten erstellen oder sich diese darstellen lassen. Entity-Dateien können durch den Entity-Editor grafisch betrachtet und editiert werden. Alle abgebildeten Dateien können zudem in einem im Designer implementierten Text-Editor mit den nötigsten Funktionen bearbeitet werden.

TextEditor

Der Texteditor ist ein Tool, um Texte zu editieren. Es verfügt über die wichtigsten Funktionen, die ein Text-Editor verfügen sollte. Jede dieser Funktionen sind über Tastenkürzel aufrufbar.

- Speichern
- Undo/Redo (Rückgängig/Wiederherstellen)
- Copy/Cut (Kopieren/Ausschneiden)
- Paste (Einfügen)
- Delete (Löschen)

Mesh-Parts-Window

Ein Mesh besteht aus mehreren Parts, welche alle in einer Listbox angezeigt werden. Andere Toolwindows verwenden dieses, um herauszufinden, welches Part ausgewählt ist und demnach genauere Informationen anzuzeigen. Wenn ein neuer Part ausgewählt wird, updaten sich automatisch alle anderen Windows durch die Verwendung des *INotifyPropertyChanged-Interface*. Es ist möglich mit einem Rechtsklick ein Part zu löschen, umzubenennen oder auszublenden.

Mesh-Vertices-Window

In diesem Fenster werden in einer Tabelle alle Vertices des aktuell ausgewählten Mesh-Parts angezeigt. Dafür müssen alle Vertices in Strings umgewandelt werden, was sehr Zeit aufwendig ist und viel Speicher verbraucht. Daher ist es empfohlen dieses Toolwindow nur für kleinere Parts zu verwenden.

Mesh-Parts-Property-Window

Dieses Toolwindow zeigt alle Eigenschaften eines Mesh-Parts. Zum Anzeigen wird ein PropertyGrid aus der DotNetBar-Bibliothek genutzt, da dieses automatisch alle Properties eines Objektes in die Tabelle einträgt. Um auszuwählen, welche Properties angezeigt werden sollen und mit welcher Beschreibung besitzen alle Properties der Designerlib.MeshPart-Klasse dafür zuständige Attribute.

3.5.3. Dialogs

Model-Loader | Add Resource-Dialog

Der Model-Loader ist dafür gedacht, um simpel Modelle in den Designer zu laden, um sie dann in der Engine weiter verwenden zu können. Die Modeldatei wird analysiert und die Daten werden in eine Datei in der Ordnerstruktur des Designers gespeichert.

Entity-Editor

Der Entity-Editor wurde implementiert, um dem Nutzer das Erstellen von Entity-Presets zu erleichtern. (siehe: 3.2.4). Der Nutzer gibt im Entity-Editor zunächst die Komponenten ein, die das zu erstellende Entity besitzen soll. Zu diesen Komponenten werden nun optional Standart-Properties bereitgestellt, die automatisch hinzugefügt werden können. Der User kann außerdem noch weitere benutzerspezifische Properties zu den Entities

hinzufügen. Diese Daten werden nun in Form einer .ent Datei im Projektordner gespeichert. Diese kann sich der Nutzer sowohl mit einem externen oder mit dem implementierten Texteditor im FM3D-Designer anschauen und wenn es ihm beliebt manuell verändern.

3.6. Logik

Neben den Klassen, in denen die Fenster beschrieben werden, existieren auch Klassen in denen die Logik definiert ist.

Project-Klasse

Die *Project*-Klasse stellt ein FM3D-Projekt dar. Sie besitzt Methoden, um Projekte aus XML-Dateien zu laden, in XML-Dateien zu speichern und den Projektnamen zu setzen. Zudem besitzt es eine Instanz zu dem gerade aktiven Projekt, den Projektnamen und das Verzeichnis in Form des Datentyps *String*. Auf die Unterverzeichnisse kann man durch ein Objekt der *RootDirectory*-KlasseAbschnitt 3.6 zugreifen.

RootDirectory-Klasse

Diese Klasse erbt von der Klasse *Directory* (Siehe Abschnitt 3.6) und stellt das sogenannte Wurzelverzeichnis dar. Sie besitzt ein Boolean, mit dem dargestellt wird, ob das Verzeichnis im FileBrowser (Siehe Abschnitt 3.5.2) angezeigt werden kann.

Directory-Klasse

Die *Diretory*-Klasse besitzt drei *ObservableCollections* in Form von Objekten der Klassen *FileObject*, *File* und *Directory*. Dadurch, dass die Directory-Klasse *ObservableCollections* des eigenen Typen hat, kann eine simple Baumstruktur von Ordner gebildet werden. Sie erbt von der *FileObject*-Klasse. Abschnitt 3.6

File-Klasse

Diese Klasse erbt auch von der *FileObject*-Klasse. Sie wird in der *Directory*-Klasse verwendet, um die Projektdateien zu beschreiben.

FileObject-Klasse

Diese Klasse beschreibt die Grundstruktur eines Objektes, welches in einer Ordnerstruktur bzw. in einem FM3D-Projekt dargestellt wird. Der Dateiname und Dateipfad werden durch den Datentyp String beschrieben.

Filebrowser View-Logic

Die Klasse *Logic*, welche im Namespace „*FM3D_Designer.src.ToolWindows.FileBrowser*“ steht, erbt von dem Interface *INotifyPropertyChanged*. *INotifyPropertyChanged* ist ein Interface, womit Events aktiviert werden können, wenn sich eine bestimmte Eigenschaft ändert, sodass andere Klassen wie z.B. ein Fenster darauf reagieren können. Die Klasse enthält eine Enumeration, welche die Anzeigeeinstellung des File-Browser beschreibt. Eine in der Klasse *Logic* stehende *ObservableCollection* enthält alle Wurzelverzeichnisse, welche mit der Klasse *Item* beschrieben werden. In einem Objekt der Klasse *Item* namens *_CurrentDirectory* und der Eigenschaft *CurrentDirectory*, welche das *Item*-Objekt von *_CurrentDirectory* zurück gibt, wird das aktuelle . Ein Interface vom Typ *IList* gibt den Inhalt der aktuell ausgewählten *Directory* im File-Browser (Siehe Abschnitt 3.5.2) zurück. Zudem besitzt diese Klasse verschiedene Methoden zur Interaktion mit dem File-Browser.

Item

Objekte der Klasse *Item* sind dafür , im *File-Browser* angezeigt zu werden. Die Klasse erbt von dem Interface *INotifyPropertyChanged*, welches schon in Abschnitt 3.6 erläutert wurde. Ein *Item* besitzt eine Enumeration namens *ItemState*, welche angibt, ob das Item im Projekt eingebunden, nicht eingebunden wurde oder nicht in der Ordnerstruktur

gefunden wird. Es besitzt zudem ein Objekt der Klasse ItemType, um den Typ der Datei anzugeben, und ein Objekt der Klasse Logic. So kann auch hier, ähnlich wie bei der Klasse *Directory*, eine Baumstruktur gebildet werden. Ein rekursives Objekt der Klasse Item gibt das *Parent*-Item an. Dies ist das *Item*, welches das übergeordnete Verzeichnis angibt. Ein Item besitzt zudem ein *ContextMenu*, welches bei einem Rechtsklick auf das Item geöffnet wird. Des weiteren wurden Methoden implementiert, welche verschiedene Dialoge und Fenster öffnen. Man kann durch sie den *Text-Editor*, den *Entity-Editor* und den *AddNewResource*-Dialog öffnen, die sofort mit diesem *Item* interagieren. Die Methode *CreateFile* kann mit Angabe eines Dateinamen, einem *ItemType* und dem *Item*, das gerade angeklickt ist, eine neue Datei erstellen. Dies geschieht im Programm über das *ContextMenu* durch den *AddResource*-Dialog.

ItemType

Diese Klasse gibt den Typ eines Items an. Sie besitzt einen Container in Form eines *Dictionary*. Als Schlüsselwert/Key wird ein String verwendet. Der Wert / die Value ist ein rekursives Objekt der selbigen Klasse *ItemType*. In ItemType stehen statische einzelne rekursive Objekte der selbigen Klasse für jeden Dateien-Typ den es in dem FM3D-Projekt existiert. Jeder Typ besitzt Pfade zu den Icons, welche im File-Browser angezeigt werden. Es existieren insgesamt acht verschiedene ItemTypen:

- Directory
- UnknownFile
- EntityFile
- MaterialFile
- SkeletonFile
- MeshFile
- TextureFile
- ModelFile

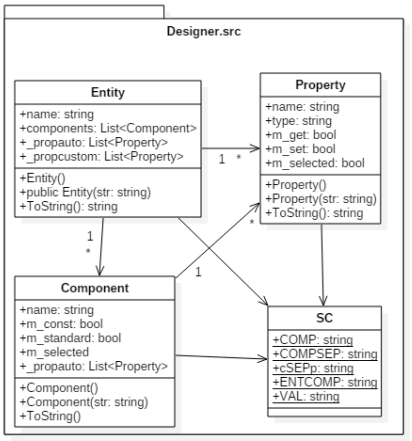


Abbildung 16.: Entity-Klassen im Designer

Entity-Klassen

Sowohl im Designer als auch in der Extension gibt es jeweils die selben drei Klassen, welche für die Entities zuständig sind: *Entity*, *Component* und *Property*. Die *Entity* Klasse beschreibt das Entity, so wie sie später im Code erstellt werden sollen. Ein Entity-Objekt besitzt einen Namen und drei Container des Typen *List*. Die erste Liste besitzt Objekte der Klasse *Component*, die zweite und dritte Liste sind Objekte der Klasse *Property* und sind jeweils für die automatisch generierten und für die benutzerdefinierten Properties. Jeder dieser Klassen besitzt zudem eine überladene *ToString()* Methode, welche dafür sorgt, dass alle Daten in ein String geschrieben werden, um sie dann später per Pipe zu verschicken. Um die einheitliche Trennung der Daten durch verschiedene *Chars* in diesem zu versendenden Stringdatensatz zu gewährleisten, wurde die statische Klasse *SC* implementiert, welche die Trennungszeichen beinhaltet. Jeder dieser drei Klassen - die statische Klasse *SC* zählt nicht dazu, da sie nur eine Sammlung von Zeichen zur Trennung sind - enthalten noch einen Konstruktor, welcher einen umgewandelten Datensatz vom Typ String wieder in eines dieser Objekte umwandelt. So können, nachdem ein Entity in Form eines Datensatzes vom Typ String per Pipe an die VisualStudio-Extension geschickt wurde, die Daten dort weiterverwendet werden. Die Klassen werden in Abbildung 16 dargestellt.

Creator

Der Creator ist dafür zuständig dem Filebrowser alle Items zu übermitteln, sodass alle Items anzeigbar sind und die Daten korrekt geladen werden. Die Klasse CreateFile erstellt eine neue Datei und fügt dieses neue Item dann dem FileBrowser hinzu.

FM3DPropertyFile-Klasse

Diese Klasse ist dafür zuständig, Daten aus der Datei *fm3d.xml* zu lesen. Diese Datei beinhaltet Daten für die Kommunikation mit der Extension per Pipe. Sowohl im Designer als auch in der Extension sind zwei von der Grundstruktur sich ähnelnde Klassen der *FM3DPropertyFile*-Klasse.

PipeSystem

Grob betrachtet ist die Pipe eine Klasse, welche die Kommunikation zwischen zwei Programmen ermöglicht. In unserem Fall kommunizieren der FM3D-Designer und die dazugehörige VisualStudio-Extension3.6. Das ganze läuft ähnlich wie bei Netzwerkkommunikation ab. Sowohl im Designer als auch in der Extension existieren von der Grundstruktur sich ähnelnde Klassen des PipeSystems.

FM3D-Extension

Die FM3D-Extension ist eine von uns entwickelte Erweiterung für VisualStudio. Sie ist dafür zuständig, gesendete Entities3.6 in C++ Code umzuwandeln. In Abbildung 17 wird der Aufbau der Extension dargestellt.

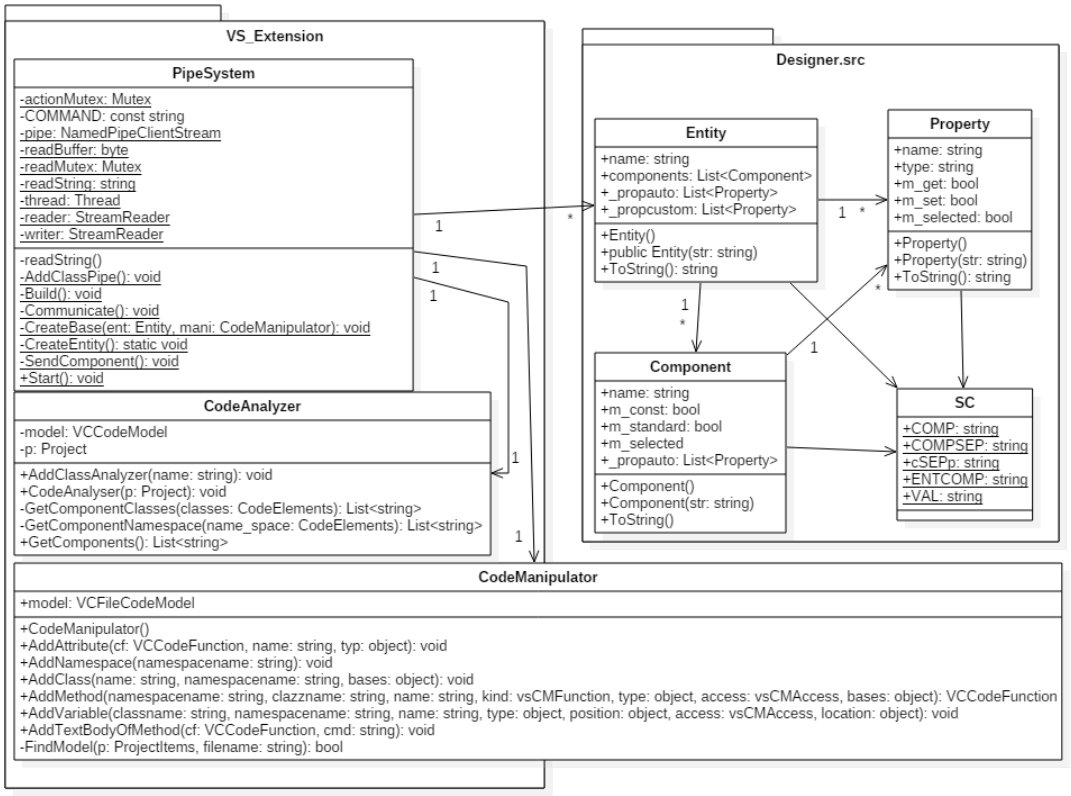


Abbildung 17.: VisualStudio-Extension

3.7. Externe Bibliotheken

3.7.1. OpenGL

Beschreibung der OpenGL

Zum Darstellen von Grafiken wird von der FM3D-Engine OpenGL (Open Graphics Library, deutsch: Offene Grafikbibliothek)verwendet. Diese Bibliothek ist eine Schnittstelle für Grafikanwendungen, die es ermöglicht auf Funktionen der Grafikkarte zugreifen zu können. Geschrieben ist sie in C, was ein problemloses Einbinden in C++ ermöglicht.

Da Grafikkarten immer innovativer werden und somit mehr Leistung bekommen, wird OpenGL immer weiter entwickelt. So entstehen immer wieder neuere Versionen der Bibliothek. Die aktuellste Version ist OpenGL 4.5 (Stand 2016).

Die verschiedenen Versionen werden grundsätzlich in zwei Gruppen unterteilen: *Legacy OpenGL* und *Modern OpenGL*. *Modern OpenGL* beschreibt hier alle Versionen ab 3.0. und *Legacy OpenGL* alle Versionen darunter.

Die FM3D-Engine verwendet ausschließlich modern OpenGL 3.3. und höher. Dies bietet weitaus mehr Flexibilität und bessere Performance. Eine frühere Version von OpenGL wäre obsolet. Rechner dessen Grafikkarte höchstens *Legacy OpenGL* verwenden, wären nicht im Stande modernes 3D-Rendering zu unterstützen.

Erwähnenswert sind auch die Erweiterungen von OpenGL. Da OpenGL nicht nur von einer einzigen Firma entwickelt wird, sondern viele Firmen ein Mitspracherecht haben, werden neue Funktionen meist speziell von einem Hersteller entwickelt. Funktionen dieser Erweiterungen erhalten einem zum Hersteller gehörenden Suffix. Wenn sich mehrere Hersteller für eine Erweiterung zusammenschließen, so bekommt sie den Suffix „EXT“. Wenn das Architecture Review Board ARB beschließt, eine Erweiterung hinzuzufügen, so bekommt sie den Suffix „ARB“ und wird in der nächsten Version zum Core-Library hinzugefügt.

In der FM3D-Engine wird versucht diese Erweiterungen möglichst zu vermeiden. Herstellerspezifische Erweiterungen werden gar nicht verwendet, damit die Engine auf möglichst vielen Systemen verwendet werden kann. Leider sind einige der für die FM3D-Engine wichtigen Funktionen, die verwendet werden noch nicht in der Core-Library. Daher müssen Erweiterungen verwendet werden.

Funktionsweise

Mit OpenGL kann man nicht direkt z.B. einen Delphin oder einen Baum rendern. Da OpenGL nur Dreiecke rendert, benötigt man um diese Abbildungen darzustellen ein 3D-Modell. Dieses besteht aus Dreieckigen Flächen, die aus "Verticles"(Punkte) gebildet werden. (Siehe Abbildung 4)

Neben den Dreiecken ist es zudem möglich einzelne Linien oder Punkte zu rendern. Dies folgt dem gleichen Prinzip, nur werden hierbei weitaus weniger Punkte zur eindeutigen Definition benötigt. Das ist aber bei der Darstellung von den 3D-Modellen nicht wirklich relevant. Möchte man also nun einen Delphin rendern, so braucht man ein 3D-Modell, welches einen Delphin darstellt und aus den besagten Dreiecken besteht. Solche Modelle kann man mit verschiedenen Modellierungsprogrammen wie zum Beispiel „Blender“ oder „3DS-MAX“ erstellen.

Früher nutzte OpenGL eine Fixed *function pipeline*. Damit war gemeint, dass fest in der Grafikkarte definiert war, wie die Dreiecke verarbeitet und gerendert werden. Man konnte jedem Punkt des Dreiecks einen feste Koordinate zuweisen und optionale weitere Attribute wie Textur-Koordinaten und Farbe hinzufügen. Mit diesen Werten wurde das Dreieck dann gerendert. Heute gibt es sogenannte *Shader*. Diese bestimmen wie ein Objekt gerendert wird.

3.7.2. Windows Presentation Foundation

Windows Presentation Foundation Windows Presentation Foundation (WPF) ist eine 2006 eingeführte Bibliothek. [20] „Sie vereint die Vorteile von DirectX, Windows Forms, Adobe Flash, HTML und CSS.“[21] Das Aussehen der Anwendungen werden mit der

Extensible Application Markup Language Extensible Application Markup Language (XAML) deklarativ beschrieben. Die Logik wird mittels C# implementiert. So sind Arbeitsschritte besser zu unterteilen und zu strukturieren. Zudem bieten C# und WPF einen übersichtlichen Syntax zur Programmierung der GUI-Komponente. Auch die VisualStudio Extension wurde in C# geschrieben. Des Weiteren sind die GUI-Bibliotheken nur in C# und WPF implementierbar. Zwar ist der Rechenaufwand aufgrund des enthaltenen bi-direktionalen *Beobachters* relativ höher, dennoch ist dies für eine so kleine Anwendung nicht bedeutungsvoll. Ein *Beobachter* ist in der Softwareentwicklung Entwurfsmuster, welches zu der Weitergabe von Änderungen an einem Objekt an von diesem Objekt abhängige Strukturen dient.

3.7.3. DotNetBar

DotNetBar ist eine Bibliothek von DevComponents, die ähnliche Funktionen, wie sie in den Office Paketen vorhanden sind, verspricht. Wir verwenden diese Bibliothek, um uns an das Design von VisualStudio an zu nähern und *Child-Windows* von *Main-Windows* erstellen zu können, die dem Nutzer eine übersichtlichere Arbeitsumgebung liefert. Hierbei handelt es sich um Fenster, die einem *Hauptfenster* untergeordnet sind. Diese verwenden wir für die einzelnen Unteroptionen des Designers. Solche *Child*-Windows haben den Vorteil, dass der User seine Arbeitsumgebung beliebig vom Aussehen anpassen kann.

3.7.4. MahApps

MahApps.Metro ist ein kostenlose open source Framework für C# (WPF), welches der Anwendung das Aussehen einer Metro basierten Anwendung verleiht. *Metro* ist eine Design-Sprache, welche Typografie- und Geometrie-fokussiert ist. Sie kommt aus dem Hause Microsoft und wurde mit dem Windows 7 Phone eingeführt.[22]

4. Verwendung der FM3D-Engine

Die Flying-Mind 3D Engine ist eine Spiel-Engine für die Entwicklung von Computerspielen. Die Flying-Mind 3D Engine besteht aus zwei Teilen: aus einer dynamischen C++ Bibliothek und dem Designer, einer grafische Nutzeroberfläche. Der Spieleprogrammierer verwendet die Bibliothek, um die Logik des Spiels zu beschreiben und den Designer, um Entities zu erstellen und Ressourcen des Spiels zu verwalten, wie Modelle und Texturen. Die Nutzung des Designers ist komplett optional. Es ist möglich ein vollständiges Spiel ohne den Designer zu programmieren. Dies wird aber nicht empfohlen.

4.1. Mindestvorraussetzungen...

...für den Computer, auf welchem später das Spiel gespielt wird:

- Windows 32-Bit/64-Bit
- OpenGL 3.3 oder höher

...für den Computer, auf welchem der Designer ausgeführt wird:

- Windows 32-Bit/64-Bit
- OpenGL 3.3 oder höher
- .Net 4.5

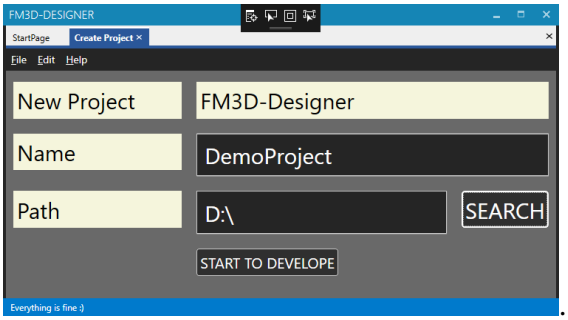


Abbildung 18.: Create-Dialog

- Visual Studio 15
- (Für einen Debug des Spiels müssen die Voraussetzungen „ ...für den Computer, auf welchem später das Spiel gespielt wird“ erfüllt sein)

4.2. Designer

4.2.1. Neues Projekt erstellen

Wenn Sie ein neues Spiel entwickeln wollen, so sollten Sie zunächst ein Projekt mit dem Designer erstellen. Wenn bereits ein Projekt erstellt wurde, können Sie zu Abschnitt 4.2.2 springen. (Siehe Abbildung 18) Natürlich ist der Designer optional und Sie können ein Spiel vollkommen ohne ihn erstellen (Informationen zu der Syntax finden Sie in der Doxygen Dokumentation, welche sich im Anhang befindet und in Abschnitt 4.3). Um nun ein Projekt zu erstellen, starten Sie den FM3D-Designer. Es öffnet sich nun ein Start-Layout (siehe Abschnitt 3.5.1). Klicken Sie nun auf die Schaltfläche *New Project*. Daraufhin wird sich ein weiterer Dialog öffnen, in dem Sie nun den Namen und Pfad ihres Projektes angeben können. Durch die Schaltfläche *Search* wird ein Explorer-Dialog geöffnet, in dem Sie ihre Ordnerstruktur nach einem geeigneten Pfad durchsuchen können. Wenn Sie nun ihre Einstellungen getätigt haben, drücken Sie auf die Schaltfläche *Start To Develope*. Sie werden nun auf den Haupt-Arbeitsplatz des FM3D-Designers weitergeleitet.

4.2.2. Altes Projekt laden

Falls Sie nun ein Projekt erstellt haben und es laden wollen, drücken Sie auf die Schaltfläche *Load* rechts neben der Textbox. Es öffnet sich nun ein Explorer-Dialog, in dem Sie nun eine *.fmproj* Datei auswählen können. Wenn Sie nun eine solche ausgewählt haben, drücken Sie auf *Start*. Sie werden nun auf den Haupt-Arbeitsplatz des FM3D-Designers weitergeleitet. Der Pfad des letzten geladenen Projektes kann durch den ContextMenü Punkt *Last Project* in die Pfadleiste geladen und dann sofort geöffnet werden.

4.2.3. Hauptarbeitsplatz

Wenn Sie nun in das Main-Layout (Siehe Abschnitt 3.5.1) gelangt sind, befindet sich rechts ein File-Browser (siehe Abschnitt 3.5.2). In diesem werden ihnen nun drei Ordner angezeigt. In diesen können Sie nun beliebig viele Entities erstellen und Ressourcen in das Projekt laden. Öffnen Sie nun ein Entity, um verschiedene Komponente hinzuzufügen. Auch können Sie Textdateien mit jeder beliebigen Endung zum Projekt hinzufügen. Die exportierten Ressourcen müssen Sie manuell in das Cpp Projekt einbinden. Wenn Sie weitere Ordner in das Projekt einfügen wollen, speichern Sie zunächst das Projekt ab und schließen den FM3D-Designer. Gehen Sie dann in Ihren Dateiexplorer, öffnen Sie den Projektpfad und erstellen an der beliebigen Stelle ein Verzeichnis. Laden Sie das Projekt neu über den Designer und klicken Sie mit der rechten Maustaste auf das Projekt-Item (Siehe Abschnitt 3.6). Ein *ContextMenü* öffnet sich nun. Betätigen Sie den Schalter *Include* und die Datei wird zum Projekt hinzugefügt.

4.2.4. Extension

Stellen Sie sicher, dass die FM3D-Extension installiert ist und starten Sie darauf hin das VisualStudio-Projekt über dem Designer. Klicken Sie, falls Sie es nicht schon getan haben, hierfür auf das Visual-Studio Icon im Designer ([1] in Abbildung 20). Bestätigen Sie die *MessageBox* und warten Sie bis das Projekt geladen wurde. Wenn es

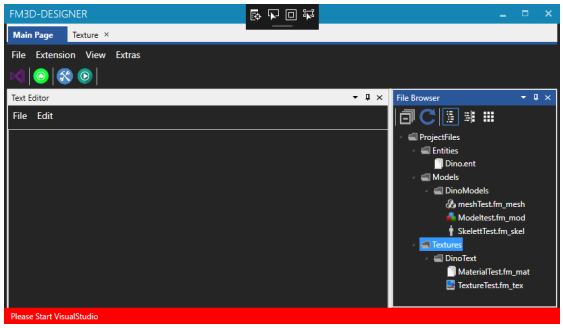


Abbildung 19.: Haupt-Fenster des FM3D-Designers

Abbildung 20.: Extension Menü

nun geladen wurde, überprüfen Sie nochmal Ihre Entities im File-Browser. Falls alle ihren Vorstellungen übereinstimmen, gehen Sie nun auf den zweiten Schalter und betätigen Sie diesen ([2] in Abbildung 20). Nun werden die Entity-Preset Klassen in dem VisualStudio-Projekt generiert. Speichern Sie ihr Projekt anschließend. Wenn Sie Ihr Spiel fertig programmiert haben, so können Sie es über das Programm mit den zwei letzten Icons Debugen und Compilen ([3] und [4] in Abbildung 20), falls Sie noch gerade im Designer beschäftigt sind. Diese Funktion ist optional.

4.3. Engine

Wenn Sie die FM3D-Engine in Kombination mit dem FM3D-Designer verwenden, so wird Ihnen bei der Projekterstellung ein funktionsfähiges VisualStudio C++ Projekt generiert, welches in der VisualStudio-Solution *GameProject.sln* zu finden ist. Es wird Ihnen geraten, nichts an dem generierten Code zu ändern. In der Datei *„presets.h“* werden die Entity-Presets generiert, welche Sie im Designer erstellen können. Sie können dem Projekt auch neue Dateien hinzufügen und unabhängig vom generierten Code programmieren. Weitere Dateien, welche ursprünglich nicht zu dem generierten Projekt

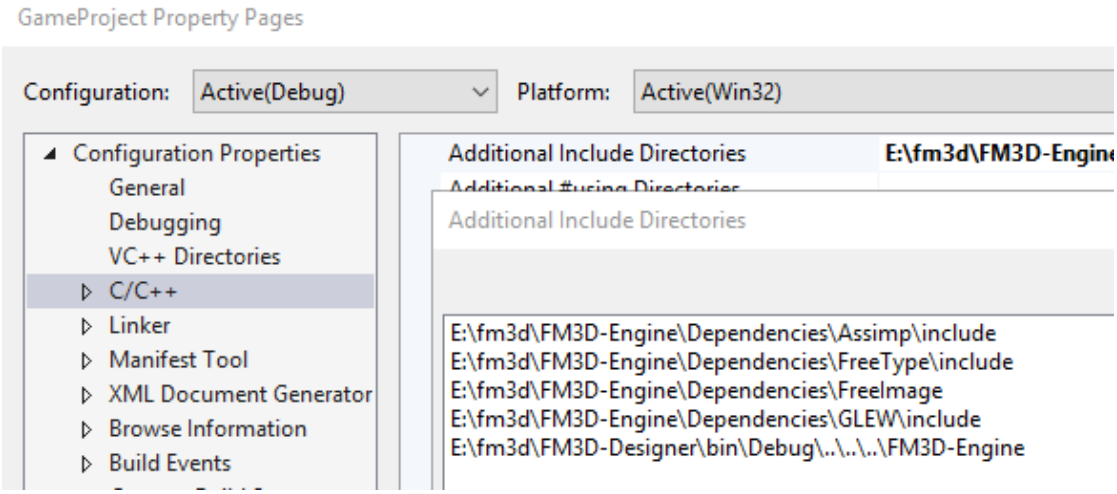


Abbildung 21.: Include Verzeichnisse

gehört haben, sollten keinen Einfluss auf die Funktionalität des generierten Codes haben. Das Spiel an sich steht in der *Main.cpp* Datei, welche im Projekt bereits vorhanden ist.

4.3.1. Voreinstellungen

Falls Sie die Engine in Kombination mit dem FM3D-Designer verwenden, so werden die Verzeichnisse automatisch eingebunden. Falls nicht, so müssen Sie Zunächst die Bibliotheken der FM3D-Engine, OpenGL, FreeImage, FreeType und Assimp in das Projekt manuell einbinden. Fügen Sie die Verzeichnisse in die zugehörige Option hinzu. Das ganze sollte so in ihren Einstellungen aussehen:

$$ConfigurationProperties->C/C++->AdditionalIncludeDirectories$$

Abbildung 21

$$ConfigurationProperties->Linker->AdditionalIncludeDirectories$$

Abbildung 22

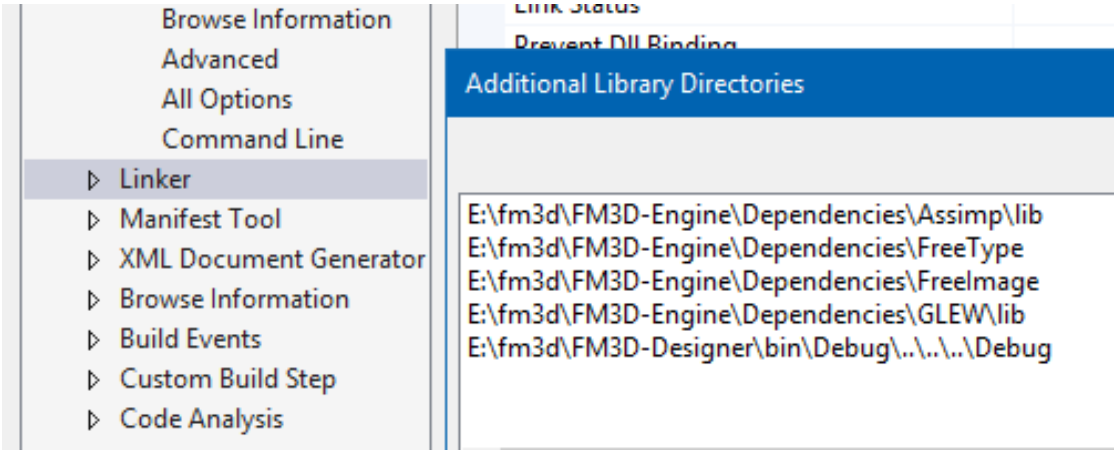


Abbildung 22.: Lib-Verzeichnisse

4.3.2. Kamera

Die Kamera ist in dem generierten Projekt bereits vorhanden. Ihnen wird empfohlen die Werte der Position, Rotation, Zoom usw. der Kamera im Bereich der *GameLogic* zu verändern. Auch dieser Bereich ist eindeutig mit Kommentaren kenntlich gemacht. In dem erstellten Projekt existiert bereits ein Objekt der Klasse *Camera*. Die *Get*-Methoden geben eine Referenz zu dem jeweiligen Parameter der Kamera. Diese können somit sofort manipuliert werden. Als Beispiel:

```
1 // Create Camera Object
2 Camera camera(Vector3f(0.3f,0.4f,0.0f));
3 // Positionattribute Manipulation
4 camera.GetPosition().x += 0.5f;
5 camera.GetPosition().y -= 0.1f;
```

Für weitere Infos über die Syntax der Kamera wird empfohlen, in der DoxyGen-Dokumentation die Klasse „Camera“ nachzuschlagen.

4.3.3. Entities

Erstellen Sie zunächst die Entities an der eindeutig kommentierten Stelle *“Create Entities here„*. Erstellen Sie beliebig viele Preset-Objekte Ihrer erstellten Entity-Preset-Klassen. Sie können nun bei jedem beliebigen Objekt „Preset-Einstellungen“ tätigen, in dem Sie dem Entity-Preset-Objekt Standardwerte zuweisen. Gehen wir davon aus, es gäbe ein Entitypreset Namens BaumPreset. Diesem wurde mittels Designer das RenderableComponent hinzugefügt.

```
1 BaumPreset baumpreset1;
```

Um nun dem Preset-Objekt ein Standartmodel zuzuweisen, verwendet man die folgende Syntax:

Erstellen wir zunächst ein Referenz-Objekt vom Typ Model.

```
2 Model* baummodel;
```

Um nun ein Model zu laden, wird die Klasse *ExternFileManager* wie folgt verwendet:

```
3 ExternFileManager::
4 ReadModelFile("res/baum.dae", rendersystem,
5 &baummodel, false, true);
```

Dies sagt nun Folgendes aus: wir übergeben den Dateipfad „res/baum.dae“ und ein Objekt des Render-Systems, in dem dann das Model gerendert werden soll. Daraufhin wird eine Adresse zu dem Objekt der Klasse *Model* übergeben, in der das Model gespeichert wird. Der darauffolgende boolesche Wert gibt an, ob Instancing verwendet werden soll und der folgende, ob das Modell animierbar sein soll. Sie können nun dem Objekt Baum dieses Standartmodel zuweisen. Dies geschieht wie folgt:

```
6 baumpreset1.SetModel(&baummodel);
```

Initialisieren Sie nun einen Entity-Pointer (Klasse: *EntityPtr*) und erstellen Sie das Entity in der bereits initialisierten *EntityCollection* namens *scene*. Als Beispiel:

```
7 EntityCollection scene;
8 EntityPtr baum = scene.CreateEntity();
```

Ein Objekt der Klasse **EntityCollection** beinhaltet alle Entities, die in dem Spiel verwendet werden. Jedoch können auch mehrere EntityCollection erstellt und verwendet werden. Um die Presets auf einen Entity-Pointer und somit auf ein Entity in einer EntityCollection anzuwenden, wird die folgende Syntax verwendet:

```
9 baumpreset1.SetComponents(baum);
```

Nun wird dem EntityPointer alle Komponenten, mit den zuvor zugewiesenen Standartwerten, zugewiesen.

Wenn ein Entity das Komponent *RenderableComponent* besitzt, so kann es in dem eindeutig kommentierten Abschnitt „Submit objects here to renderer“ dem Renderer übergeben werden. Das heißt nun, dass das Model des Entities gerendert werden kann.

```
10 /// ############################
11   //
12   //  Submit objects here to renderer!
13   //
14   renderer3D->Submit(baum.get());
15 /// ############################
```

Bevor Sie den Entities die Modelle zuweisen, vergessen Sie nicht die Modelle in die Projektmappe von VisualStudio zu laden.

4.3.4. Inputsystem

Hierfür wird die Klasse *Input* verwendet. Möchte man nun Tasten abfragen, so muss man zunächst über das Fenster auf das Objekt der Klasse Input zugreifen. Nun kann eine Methode aus dieser Klasse verwendet werden. Möchten Sie nun abfragen, ob z.B. die Taste F5 auf der Tastatur gedrückt wurde, so schreiben Sie dies so in den Code:

```
1 win->GetInput().CheckKey(KEY_F5);
```

Möchten Sie nun überprüfen, ob die Linke Maustaste gedrückt wurde, so tun Sie dies folgendermaßen:

```
2 win->GetInput().CheckMouse(MOUSE_LEFT);
```

Möchte man nun die Position des letzten Klicks der linken Maustaste ermitteln, benutzt man diese Methode:

```
3 win->GetInput().GetLastposClick(MOUSE_LEFT);
```

Diese gibt einen zweidimensionalen Vektor vom Typ Float zurück, welcher die Position vom letzten Klick der Maus mit einer bestimmten Maustaste beschreibt. Die folgende Methode gibt Daten in Form eines zweidimensionalen Vektors mit der aktuellen Position der Maus zurück:

```
4 win->GetInput().GetLastposInst();
```

Alle Tasten der Tastatur und Maus können über Makros angesprochen werden. Auch können Sie die ASCII-Codes der einzelnen Tasten verwenden. Die Makros, welche die Tasten der Tastatur beschreiben, starten mit *„KEY_“*. Die Makros, die für die Maus verwendet werden, starten mit *„MAUS_“*. (Für weitere Informationen: Siehe DoxyGen Dokumentation)

5. Resumé

5.1. Verbesserungsfähiges

Wie jedes Entwicklerteam hatten auch wir einige, nicht wenige Schwierigkeiten bei der Entwicklung unserer Software. Die wichtigsten Fehler und verbesserbaren Punkte haben wir hier dokumentiert.

5.1.1. Software

Der ursprüngliche Grund, warum wir den Designer entwickeln wollten, war das räumliche Darstellen von dreidimensional-renderbaren Entities und verschiedener Szenen vor dem Export in ein C++ Projekt. Handhaben wollten wir es ähnlich wie bei WPF: Eine Designer-Ansicht sollte die Übersicht und Optionen auf Entities liefern, welche mit einem Szenen-Editor editierbar wäre. Diese Ansicht sollte dem Nutzer die Ausgangslage des Spieles anzeigen. Der Szenen-Editor sollte dem Nutzer ermöglichen, verschiedene Entity-Presets (siehe: Abschnitt 3.2.4) in eine Szene zu setzen. Die Szene würde dann mit XML- und Designer-spezifischen Skripten beschrieben und später wie Entities via Pipe (sieheAbschnitt 3.6) per Extension in Code umgewandelt werden. Der Nutzer der Engine könnte dadurch selbständig ein komplett funktionsfähiges Programm generieren, ohne selbst programmieren zu müssen. Wenn man nun einen solchen Skript-gesteuerten Szenen-Editor implementieren will, so müsste man auch einen komplexeren Text-Editor mit mehreren Funktionen implementieren. Er sollte an die Befehle für Skripte und XML angepasst werden und über automatische Wortvervollständigung für Befehle verfügen. Der Code müsste direkt bei der Eingabe auf Richtigkeit

geprüft und Fehler sollten dem Nutzer angezeigt werden. Um dem Nutzer ein besseres Spiel-Entwicklungserlebnis zu garantieren, könnte man den Entity-Editor in andockbare Fenster umlagern. So wäre es dem Nutzer möglich verschiedene Entities parallel zu bearbeiten. Vieles der geplanten Ziele mussten aus Zeitmangel heraus gekürzt werden. (Grund: Siehe Abschnitt 5.1.2)

5.1.2. Team und Workflow

Planung

Uns wurde schon relativ früh bewusst, dass die Planung bei einem größeren Projekt das A und O ist. Die Arbeitsschritte müssen am Anfang klar definiert sein, sodass man die Arbeit zum einen aufteilen und zum anderen planmäßig in einer festen Zeit fertigstellen kann. Bei unserem Projekt war die Planung gegen Anfang recht ausgefeilt, doch zählt nicht nur die Planung am Anfang des Projektes, sondern auch die Planung und Strukturierung während das Projekt am Laufen ist und die Kommunikation 5.1.2 währenddessen. Es kam des öfteren vor, dass während der Entwicklung des Programmes bessere und effizientere Lösungsvorschläge in Frage kamen, als ursprünglich geplant war. Dies hatte oft auch eine Umstrukturierung des Projektes zur Folge. Als Beispiel nehme man das Entity-System(Siehe: 3.2.4). Zu Anfang wollten wir die ganzen Objekte aus Klassen erstellen, doch erwies sich dies als ungeeignet, da sonst verschiedene Objekte nur schwer miteinander interagieren könnten und außerdem müsste man jede Klasse einzeln definieren, was sehr viel Zeit beansprucht. Als wir das Entity-System nun eingeführt hatten, mussten wir natürlich das komplette Projekt dem Entity-System anpassen. Dort hatten wir einen Planungsdefizit.

Kommunikation

Kommunikation ist das **wichtigste** bei einem komplexeren Projekt, bei dem mehrere Personen involviert sind. Um einen ununterbrochenen Arbeitsfluss zu garantieren, muss auch eine flüssige, detaillierte und verständliche Kommunikation vorhanden sein um Missverständnisse zu verhindern. Man muss immer bedenken, dass Mitarbeiter nicht

in den Kopf anderer schauen können. So muss man seine Mitarbeiter immer über den eigenen Arbeitsfortschritt am laufenden halten.

Klein anfangen

Eines unserer größten Probleme war die Unterschätzung der von uns benötigten Zeit und die Überschätzung von uns selbst. So kam es, dass wir uns zu viele Funktionen überlegt hatten, die wir in unsere Lernleistung verarbeiten wollten, welche wir später aber des Zeitdruckes wegen wieder herausstreichen mussten. Wir haben uns des öfteren an mehreren Teilen des Programmes gleichzeitig aufgehalten. Dies hatte zwar zum einen den Vorteil, dass man immer am Arbeiten war und man so durch kleinere Erfolgserlebnisse die Motivation am Arbeiten nicht verlieren konnte. Dennoch hatte es den großen Nachteil, dass so wichtige Teile und Funktionen eines Programmes immer weiter bis zum Ende aufgeschoben wurden. Anstatt sich sofort diesen wichtigen Grundfunktionen zu widmen, steckten wir die Zeit so in Nebenfunktionen, die nicht essentiell wichtig für das Programm waren. So wurde uns im Laufe des Projektes klar, dass es besser ist, mit einem kleiner konzipierten Programm anzufangen, bei dem die Grundfunktionen alle vollständig funktionsfähig sind und erst später, wenn das Grundgerüst und die Grundfunktionen eines Programmes stehen, man es ausbauen und verbessern kann.

5.2. Haben wir unser Ziel erreicht?

Auch wenn viele der Funktionen, die wir uns zusätzlich vorgenommen hatten, der Zeit wegen aus dem Programm gefallen sind, haben wir dennoch unser Ziel erreicht. Wir haben uns vorgenommen eine Engine zu entwickeln, die dem Nutzer das Programmieren von Spielen erleichtert. Zudem wollten wir Tools entwickeln, die das Arbeiten mit der Engine vereinfachen und die Funktionen zur Unterstützung bieten. Genau dies haben wir auch erreicht: Wir haben eine funktionsfähige Engine „*from Scratch*“ geschaffen, mit der man ein komplettes Spiel, aber auch andere 3D-GUI-lastige Anwendungen

programmieren kann. Zudem haben wir ein sehr nützliches Tool für die Programm-Entwicklung entwickelt, welches dem Entwickler eine Menge Code-Schreibarbeit abnimmt.

5.3. Was haben wir gelernt?

Rückblickend können wir sagen, dass wir eine Menge durch diese besondere Lernleistung gelernt und auch verinnerlicht haben. Dies reicht von theoretischem bis hin zu praktischem Wissen. Die analytische Auseinandersetzung verschiedener Game-Engines (siehe Abschnitt 1.4) und das Einlesen in Fachbücher zu diesem Thema hat uns die Tür in die Spiele- und Engine-Entwicklung geöffnet. Um nun 3D-Grafiken darzustellen mussten wir uns mit dem OpenGL Syntax(siehe Abschnitt 3.7.1) sowie Projektions- und Licht- Berechnungen (siehe Abschnitt 2.1) auseinander setzen. Die Entwicklung des FM3D-Designers hat uns tiefe Einblicke in den C-Sharp Syntax ermöglicht. So konnten wir uns eine weitere Programmiersprache aneignen. Für das Projekte- und Dateien-Speichern, so wie es in unserem Designer geschieht, mussten wir uns den Umgang mit XML-Dateien (siehe Abschnitt 2.2) und den dazugehörigen Bibliotheken aneignen. Auch wurde unser Umgang mit dem C++ Syntax gefestigt. Zudem haben wir gelernt, wie man eine Extension für VisualStudio entwickelt und mit ihr kommunizieren kann.

A. Anhang

Literaturverzeichnis

[1] J. Gregory, *Game Engine Architecture*. United Kingdom: Taylor and Francis Ltd., 2014, second Edition.

[2] H. Kahnt, *Brockhaus - In eine Band*. Mannheim: Brockhaus GMBH, 1994, 6. Auflage.

[3] B. Koster, *A Theory of Fun for Game Design*. United States: O'Reilly and Associates, 6. November 2004, 2. Auflage.

[4] CryTek, "Cryengine," 2017, [accessed 15-März-2017]. [Online]. Available: https://www.cryengine.com/

[5] Wikipedia, "Gameboy, details," 2017, [accessed 27-Januar-2017]. [Online]. Available: https://de.wikipedia.org/wiki/Game_Boy#Details

[6] Khronos-Group, "Opengl® 4.5 reference pages," 2017, [accessed 28-Februar-2017]. [Online]. Available: https://www.khronos.org/registry/OpenGL-Refpages/gl4/

[7] E. Meiri, "Ogl dev modern opengl tutorials," 2017, [accessed 28-Februar-2017]. [Online]. Available: http://ogldev.atspace.co.uk/index.html

[8] ThinMatrix, "Opengl 3d game tutorial," Juli 2014, [accessed 28-Februar-2017]. [Online]. Available: https://www.youtube.com/watch?v=VS8wlS9hF8E&list=PLRIWtICgwaX0u7Rf9zkZhLoLuZVfUksDP

[9] Cherno, "How to make a game engine," März 2015, [accessed 28-Februar-2017]. [Online]. Available: https://www.youtube.com/watch?v=vWU8EltWTfM&list=PLlrATfBNZ98fqE45g3jZA_hLGUrD4bo6_&index=1

[10] opengl tutorials, “Opengl-tutorial,” 2017, [accessed 28-Februar-2017]. [Online]. Available: http://www.opengl-tutorial.org/

[11] Khronos-Group, “Rendering pipeline overview,” 2017, [accessed 28-Februar-2017]. [Online]. Available: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[12] Wikipedia, “Slerp,” 2017, [accessed 8-März-2017]. [Online]. Available: https://en.wikipedia.org/wiki/Slerp

[13] Songho, “Opengl projection matrix,” 2017, [accessed 10-März-2017]. [Online]. Available: http://www.songho.ca/opengl/gl_projectionmatrix.html

[14] Khronos-Group, “Depth buffer precision,” 2017, [accessed 10-März-2017]. [Online]. Available: https://www.khronos.org/opengl/wiki/Depth_Buffer_Precision

[15] Wikipedia, “Extensible markup language,” 2017, [accessed 1-März-2017]. [Online]. Available: https://de.wikipedia.org/wiki/Extensible_Markup_Language

[16] net tutorials.com, “Introduction to xml with c-sharp,” 2017, [accessed 1-März-2017]. [Online]. Available: http://csharp.net-tutorials.com/xml/introduction/

[17] Microsoft, “Microsoft api- und referenzkatalog,” 2017, [accessed 27-Januar-2017]. [Online]. Available: https://msdn.microsoft.com/de-de/library

[18] Wikipedia, “Conversion between quaternions and euler angles,” 2017, [accessed 10-März-2017]. [Online]. Available: https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles

[19] P. Dr. Chen, *The Entity-Relationship Model – Toward a Unified View of Data*. Massachusetts Institute of Technology, 1976.

[20] Wikipedia, “Windows presentation foundation — wikipedia, the free encyclopedia,” 2017, [accessed 14-Januar-2017]. [Online]. Available: https://de.wikipedia.org/wiki/Windows_Presentation_Foundation

[21] T. Theis, *Einstieg in WPF 4.5. Grundlagen und Praxis*. Bonn: Galileo Press, 2013, 2. Auflage, S 15.

[22] Wikipedia, “Metro (design language) — wikipedia, the free encyclopedia,” 2017, [accessed 6-März-2017]. [Online]. Available: https://de.wikipedia.org/wiki/Metro_(design_lenguage)