Max-Eyth-Schule Kassel
Berufliches Gymnasium
Schwerpunkt : Datenverarbeitung
Besondere Lernleistung
Leitung : Herr Andreas John, Herr Christian Dippel Q4 2013

Die Entwicklung eines 2D-Computerspiels mit einer selbstentwickelten Engine

22.03.2013

Julien Saevecke
Hellweg 26
34292 Ahnatal
Email: Julien.Saevecke@gmx.de

Johannes Winkler
Frettholz 12
34376 Immenhausen
Email: JW.Artery22@web.de

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Vorwort

1.1. Vorwort von Johannes Winkler

Ich habe mich schon immer für Computerspiele begeistert und spiele sehr gerne solche. Doch nicht einfach nur das Spielen interessiert mich, es reizt mich selbst kreativ zu werden und eigene Inhalte und Aspekte zu erschaffen. Im Jahre 2008 fing ich an, mit dem Map-Editor[1] von Warcraft3[2] eigene kleine Karten und Spielvariationen zu kreieren. Ich lernte in einer gewissen Weise das Programmieren mit einer GUI[3] die der Editor von Warcraft3 bot. Davon war ich derart fasziniert, dass ich mich in der Mapping-Szene von Warcraft3 und später von Starcraft2[4] engagierte und mitwirkte, in dem ich begann eigene Spiele zu „programmieren".

Die Faszination an der Programmierung und dem Erschaffen von Spielen brachte mich zu dem Entschluss die Max-Eyth-Schule in Kassel zu besuchen, um richtige Programmierung zu erlernen. Meine Schulzeit dort führte mir nochmals deutlich vor Augen, dass ich später einmal in der PC-Spiele Entwicklungsbranche als Programmierer arbeiten will.

Ein eigenes, richtiges Spiel selbst zu programmieren war ein Wunsch, den ich mir mangels Kenntnisse, lange nicht erfüllen konnte. Im Verlauf der 12. Klasse wurde mir klar, dass ich definitiv eine besondere Lernleistung (BLL) in Datenverarbeitungstechnik für das Abitur einbringen wollte, ob nun Spiel oder nicht war mir egal.
Mein Freund und Mitschüler Julien wollte genau wie ich auch ein PC-Spiel programmieren und hatte sogar schon mit einem Freund zusammen damit angefangen. Als wir uns irgendwann einmal im Zug über PC-Spiele und die BLL unterhielten, beschlossen wir, sein angefangenes Spiel zusammen zu programmieren und als BLL ins Abitur einzubringen.
Was nach einigen Gesprächen mit Herr John und Herr Semper auch möglich gemacht wurde, denn es ist das erste Mal, dass Schüler zu zweit eine BLL anfertigen und ins Abitur einbringen.

Aufgrund meiner Erfahrung in der Mapping-Szene von Warcraft3 und Starcraft2 war mir von Anfang an klar, dass ich gerne die GUI, sowie die Künstliche Intelligenz (KI) des Spiels programmieren möchte. Die GUI ist der Hauptteil meiner Arbeit, die KI ist aufgrund der Zeit und des Anspruchs nur sehr kurz ausgefallen.

[1] Ein Programm zum Erstellen von eigenen Karten(Maps) und Modifikationen(Mods).
[2] Warcraft3 ist ein Strategiespiel von Blizzard Entertainment aus dem Jahre 2002.
[3] Engl. Graphical User Interface – Dieser Begriff bezeichnet eine grafische Benutzeroberfläche, in diesem Fall ist eine mögliche „Programmiersprache" des Editors von Warcraft3 beziehungsweise (bzw.) Starcraft2 gemeint.
[4] Starcraft2 ist ein Strategiespiel von Blizzard Entertainment aus dem Jahre 2010

1.2. Vorwort von Julien Saevecke

Von klein auf habe ich mich mit Computerspielen beschäftigt und habe immer mit begeisterten Blick auf die Spiele geschaut, es steigerte in mir immer mehr das Verlangen selber in einem Computerspiel mit zu wirken. Mit 15/16 fing ich dann an mich zum ersten Mal an einer Programmiersprache zu versuchen und ich wollte unbedingt mehr und mehr über die Entwicklung von Spielen erfahren. Um das Programmieren besser zu „beherrschen" suchte ich mir nach der 10ten Klasse eine Schule, welche zumindest Informatik im praktischen Bereich anbietet, ich stieß auf die Max-Eyth-Schule, weil sie den Schwerpunkt Datenverarbeitungstechnik anbietet. Ich konnte neue Kenntnisse erlangen und festigen.

Die individuellen Projekte, welche wir fast jedes Jahr machten, gaben mir die Chance ernsthaft an kleine Spiele zu arbeiten. Ich programmierte im ersten Projekt „Snake[5]" auf der Konsole und im zweiten Projekt „Brick Breaker[6]" als MFC Anwendung, denn mir gefällt es nicht langweilige normale Programme zur Berechnung von irgendetwas zu programmieren.

Das was mich bei der Spiele-Programmierung reizt ist die Freiheit Ideen zu schaffen, zu planen und auszuführen. Jedes Spiel ist auf seine Art und Weise anders, auch wenn sie zur selben Art gehören. Man kann neue kreative Ideen rein bringen und somit seine „Fantasiewelt" schaffen, mit seinen eigenen Regeln.

Ich merkte schnell, dass mir das Programmieren von Spielen Spaß macht und so kam es mir gelegen diese besondere Lernleistung zu machen. Es motiviert mich endlich ein komplett eigenes Grundgerüst für das Spiel zu entwickeln, es ist nicht mehr das kleine Schlangen-Spiel, sondern etwas Großes, etwas was man bei seinen Bewerbungen stolz vorzeigen kann.

Ich entschied mich in dieser besonderen Lernleistung die „Grund"-Engine zu programmieren bzw. alles was im Hintergrund läuft, um das Spiel stabil zu halten. Es ist mir gelungen eine kleine, beschränkt modifizierbare Engine zu schaffen, mit einem soliden Design. Es gibt jedoch noch viele Verbesserungen, welche ich in der Zukunft durchführen möchte, auch am Design. Es ist nämlich sehr schwer, alle kleinen Zahnräder mit einander zu verbinden und laufen zu lassen. Ich musste sehr oft, die Engine umschreiben bzw. das Design verändern und es sind immer noch Verbesserungen in Sicht.

Ich möchte zum Schluss mich noch einmal bedanken, an alle die mich Unterstützt haben und Verständnis dafür hatten, dass ich Zeit so wenig für sie hatte.

[5] Es ist ein Spiel in dem eine Schlange in einem beschränkten Bereich gesteuert wird und diese wird beim fressen immer größer, das Ziel ist es so viel wie möglich zu fressen ohne dabei den beschränkten Bereich zu verlassen

[6] Es ist ein Spiel in dem Blöcke zerstört werden müssen mit Hilfe einer Plattform und einem Ball. Das Ziel ist es die Blöcke zu zerstören ohne das der Ball hinter die Plattform gerät

2. Ehrenwörtliche Erklärung

„Wir erklären hiermit, dass wir die vorgelegte Facharbeit ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt haben. Wir bestätigen ausdrücklich, Zitate und Quellenangaben mit größter Sorgfalt und Redlichkeit in der vorgeschriebenen Art und Weise kenntlich gemacht zu haben."

Julien Saevecke Johannes Winkler

3. Einleitung

Unser Thema ist die Entwicklung eines 2D-Computerspiels mit einer selbst entwickelten Engine. Wir haben uns dazu entschieden, die Aufteilung des Projektes modular erfolgen zu lassen. Denn das Projekt lässt sich nicht in klar getrennte Arbeitsbereiche aufteilen, z.B. Software und Hardware. Wir beide arbeiteten zusammen an dem Code, doch jeder hatte seine eigenen „Arbeitsbereiche“. Die Aufteilung erfolgte folgendermaßen:

Julien Saevecke: Die Programmierung der Spielzustände, der Ressourcenverwaltung, des Physiksystems, der Objekte und der Karten im Spiel.

Johannes Winkler: Die Programmierung der Benutzeroberfläche, Eingabe-Events, Auslöser, Missions-System und der Künstlichen Intelligenz (KI).

Des Weiteren gehört zu diesem Projekt noch ein dritter Mann Waldemar Hoppe. Er ist unser Grafiker und erheblich am Game-Design beteiligt. Er selbst ist ein Freund von Julien und studiert Informatik an der Universität Kassel. Von ihm stammen die qualitativ hochwertigen Grafiken der Menüs, er ist aber nicht an der Programmierung beteiligt gewesen.

Im Folgenden werden die Teile, die Julien Saevecke geschrieben hat mit J.S. in der Überschrift gekennzeichnet. Alle weiteren Punkte wurden von ihm verfasst, bis J.W. hinter einer Überschrift steht. Alle folgenden Teile wurden dann von Johannes Winkler verfasst. Die Einleitung und die Punkte 3. bis 5.2 wurden von uns beiden zusammen verfasst.

3.1. Ziel

Unser Ziel war es, dieses 2D-Computerspiel soweit zu entwickeln, wie der für ein solches Thema recht kurze Zeitrahmen (Anfang September 2012 bis Ende März 2013) es ermöglichte. Natürlich war es nicht möglich, ein komplett fertiges Spiel in dem Umfang zu entwickeln, den wir uns vorstellten. Denn es handelt sich hierbei um ein sehr komplexes und aufwändiges Projekt. Allein in diesem Stadium beinhaltet das Projekt ca. 25.000 Zeilen Code (mit Leerzeilen und Kommentaren), sowie 17.370 Zeilen an reinem Quellcode. Gute Computerspiele haben im Durchschnitt eine Entwicklungszeit von zwei bis fünf Jahren und an diesen arbeiten teilweise über hundert Personen.

Damit ist das Ergebnis unserer Aufgabenstellung ein 2D-Computerspiel, welches alle wichtigen Systeme besitzt, aber noch nicht über Spielinhalt (Gamecontent) verfügt. Dies wurde auch bis zu der Abgabe dieser Dokumentation erreicht. In der folgenden Zeit, die uns noch bis zur Präsentation bleibt, werden wir noch etwas weiterarbeiten. Dies beinhaltet hauptsächlich das Programmieren und Hinzufügen von spielrelevanten Inhalten.

3.2. Konzept/Ideen

Das Konzept des Spiels ist, dass der Spieler ein Raumschiff besitzt, welches modular aufrüst- und veränderbar ist. Dies bedeutet, dass der Spieler einen gewissen Raumschifftyp besitzt und diesen mit verschiedenen Modulen wie z.B. Raketenwerfern, Schilden oder Triebwerken ausrüsten kann.
Das Spiel konzentriert sich immer auf eine Raumstation und ein Sternensystem in dem sich der Spieler befindet. In diesem Sternensystem ist die Raumstation der Dreh- und Angelpunkt für den Spieler. Dieser kann dort sein Schiff modifizieren, Missionen annehmen usw. In dem Sternensystem kann sich der Spieler mit seinem Raumschiff frei bewegen und die vorgegebenen Missionen der Hauptstory[7] sowie Nebenmissionen erfüllen.

Es gibt einige Spiele die als Inspiration für das Game-Design, den Stil des Spiels, dem Gesamtkonzept und einigen programmierten Systemen gedient haben.
Inspirationsquellen für das Game-Design waren hauptsächlich Freelancer[8] und SPAZ[9]. Bei einigen Systemen hat sich Johannes teilweise von der Funktionalität an Systemen von Warcraft3 und Starcraft2 orientiert. Aber auch Spiele wie Metro 2033[10] und Dawn of War 2[11] inspirierten uns in gewissen Bereichen.

3.3. Name des Spiels und der Engine

Als Arbeitsnamen verwendeten wir durchgängig Outerspace. Diesen hatte sich Julien überlegt, schon bevor Johannes am Projekt mitwirkte. Aber als endgültiger Name für das Spiel war er uns zu unspektakulär und zu wenig aussagekräftig. Wir konnten uns bisher noch auf keinen anderen Namen einigen, was aber auch daran liegt, dass das Spiel momentan noch keinen Gamecontent besitzt; der Name sollte sich auch darauf beziehen.
Auf dem Pad im Hauptmenü lässt sich zwar der Name „Harbinger of Eternity“[12] lesen, doch dieser war nur ein Platzhalter und resultierte aus dem damaligen Stand des Projektes. Damals steckte unsere Engine noch in den Kinderschuhen und die damals aktuelle Version war eben nur der „Vorbote“ unserer Engine. Diese trägt nämlich den Namen „Eternity-Engine“. Diesen Name fanden wir für die Engine interessant klingend und passend.

[7] Die Story oder Geschichte bezeichnet den Inhalt bzw. den inhaltlichen Hergang eines Spiels
[8] Entwickelt von Digital Anvil und publiziert durch Microsoft Game Studios, veröffentlicht im März 2003
[9] Entwickelt und publiziert von MinMax Games, veröffentlicht im August 2011
[10] Entwickelt von 4A Games und publiziert durch THQ, veröffentlicht im März 2010
[11] Entwickelt von Relic Entertainment und publiziert durch THQ, veröffentlicht im Februar 2009
[12] Englisch für „Vorbote der Ewigkeit“

4. Bibliotheken

Die Eternity-Engine ist eine Spiele-Engine, das bedeutet für unser Projekt, dass diese nicht von Anfang an programmiert wurde. Sie benutzt eine Vielzahl von freien und starken Bibliotheken. Diese braucht sie um ihre Arbeit zu vollrichten. Insgesamt benutzt die Eternity-Engine fünf Bibliotheken: die SFML[13]-, STL-, Boost[14]-, Lua[15]- und die Luabind[16]-Bibliothek. Sie erleichtern uns die Arbeit und sie erweitern sogar unsere Möglichkeiten.

4.1. SFML

Die „Simple and Fast Multimedia Library" ist eine freie, plattformunabhängige, in C++ geschriebene, Multimedia-Bibliothek. Sie steht unter der „zlib/libpng"-Lizenz[17] und wird immer noch von Laurent Gomila entwickelt. Der Quellcode ist im objektorientierten Design programmiert worden und ist für jeden einsehbar und ebenso in Kooperation mit dem Entwickler erweiterbar und veränderbar. Sie ist für 2D-Programme geeignet und beherbergt eine 3D-Schnittstelle zu OpenGL. Sie ist neben ihrer Struktur, auch die Leistungsfähigste 2D-Bibliothek zurzeit. SFML ist in Abschnitte aufgeteilt: System, Fenster, Grafiken, Audio und Netzwerk. Die Eternity-Engine benötigt folgende Aspekte dieser Abschnitte:

System:
- Zeitmessung
- Threads und Mutexes

Fenster:
- Erstellung des Fensters mit allen möglichen Eigenschaften
- Echtzeit-Eingabe von Maus und Tastatur und damit auch das Event Handling

Grafiken:
- Das Laden und Speichern von Grafiken für die Formate JPEG, JPG, PNG
- Darstellung von Grafiken auf den Bildschirm
- Translation, Rotation, Spiegelung, Skalierung und Einfärbung der Grafiken
- Textdarstellung und Laden von Schriftarten
- Erstellung von Geometrischen Figuren, wie Kreise, Linien und Rechtecke
- Die Kameraklasse „View" zum automatischen skalieren unserer Grafiken und dem Benutzer der Eternity-Engine eine flexible Ansicht auf 2D-Bereiche zu gewähren.

Audio:
- Laden von Musik-Dateien
- Laden von Sound-Dateien
- Bearbeitung der Lautstärken: Sound-Lautstärke, Musik-Lautstärke und der globalen Lautstärke
- Das Abspielen von Audio-Dateien

[13] http://www.sfml-dev.org/
[14] http://www.boost.org/
[15] http://www.lua.org/
[16] http://www.rasterbar.com/products/luabind.html
[17] http://opensource.org/licenses/Zlib

4.2. STL

Die STL(Standard Template Library) ist eine große in C++ geschriebene Bibliothek, welche aus weiteren kleinen Bibliotheken besteht. Sie ermöglicht uns eine angenehme Verwaltung über Listen[18], Vektoren[19] und Maps[20]. Sie ist ein fester Bestandteil der Programmiersprache C++ und ist demnach auch fast in jedem Compiler, welcher C++ unterstützt, verfügbar.

4.3. Boost

Die Boost –Bibliothek ist eine freie C++ Bibliothek. Sie besteht aus einer großen Anzahl von Unterbibliotheken und bügelt Schwächen der Programmiersprache C++ aus. Neben dieser Ausbesserung bietet sie gängige Algorithmen um Probleme zu lösen. In der Eternity-Engine findet Boost in der Speicherverwaltung den Einzug. Sie bietet uns „intelligente"-Zeiger, wenn diese mit einer Reservierung des Speichers initialisiert werden, wird dieser reservierte Speicher automatisch wieder freigegen, sobald zum Beispiel der „intelligente" Zeiger zerstört wird oder alle auf den Speicherbereich verweisende Referenzen nicht mehr existieren. Diese Bibliothek wird jedoch noch nicht im gesamten Projekt benutzt, da die Bibliothek erst im späteren Verlauf der Entwicklung dazu gekommen ist.

4.4. Lua

Lua ist eine in C geschriebene freie, schnelle und starke Script[21]-Sprache. Mit ihrer Hilfe ist es möglich Quellcode von außen zu modifizieren ohne am Quellcode etwas zu verändern. Sie ist leicht zu lesen und somit auch benutzbar für Menschen ohne große Programmiererfahrung. In unserer Engine wird sie für die Künstliche Intelligenz, das Quest-System, das Trigger-System und für alle möglichen Initialisierungen benutzt, zum Beispiel für die Registrierung der zu ladenden Ressourcen und deren Pfade. Dadurch, dass Lua in C geschrieben ist, ist es schwer möglich objektorientiertes Design mit Lua zu verbinden. Um diese Schwäche auszugleichen haben wir Luabind noch mit in das Projekt genommen. Es ist möglich mit solch einer Skript-Sprache während der Laufzeit des Programmes Änderungen am Programm vorzunehmen, wenn man die Skripte zur Laufzeit ändert und diese auch im Programm immer wieder aufgerufen wird.

4.4.1. Luabind

Luabind erweitert Lua dahingehen, dass Klassen, dessen Attribute und dessen Methoden im Script, relativ zu einem Objekt, benutzt werden können. Es ist mit der Erweiterung Luabind erst möglich die Eternity-Engine modifizierbar zu machen, aufgrund ihrer Unterstützung zum objektorientieren Design. Es gibt noch mehr Sachen die Luabind erweitert jedoch benutzen wir diese im Projekt nicht.

[18] http://www.cplusplus.com/reference/list/list/
[19] http://www.cplusplus.com/reference/vector/vector/
[20] http://www.cplusplus.com/reference/map/map/
[21] Ein Script ist eine Datei, mit ihr ist es möglich das Programm während der ihrer Laufzeit zu beeinflussen

5. Vereinbarungen mit dem Projekt-Partner

Um ein solches Projekt ordentlich zu bearbeiten und um einen geregelten Ablauf zu gewähren, haben wir einige Absprachen und organisatorische Dinge miteinander beschlossen und abgeklärt. Diese betreffen den Programmierstil, die Organisation und Verwaltung des Codes, sowie Absprachen bezüglich des Projektes an sich.

Da wir ein Team aus drei Leuten (Julien, Johannes und Waldemar Hoppe) sind, haben wir oft Gespräche über das Projekt geführt, teilweise über Skype und teilweise direkt bei einem von uns zuhause. Dabei haben wir unter anderem die Dinge festgelegt, die im Folgenden erklärt werden.

5.1. Programmierstil

Wir legten schon in der Schule viel Wert auf einen ordentlichen Code und Programmierstil, doch unsere Eigenheiten unterschieden sich teilweise voneinander. Für dieses Projekt war es uns sehr wichtig, dass wir einen ordentlich organisierten, übersichtlichen und möglichst selbsterklärenden Code schreiben.
Dies umfasst die Namensgebung von Objekttypen, Variablen-, Funktionsnamen und Aufbau des Codes. Die erste Entscheidung war, dass wir alle Namen im gesamten Code in Englisch schreiben, sodass jeder der Englisch kann die Namen aller Variablen, Klassen und Typen verstehen kann.

Als zweites war uns wichtig, dass man die einzelnen Namespaces unterscheiden kann, denn wir benutzen gleich fünf verschiedene. Wir finden es ist übersichtlicher, wenn man sieht welche Klassen und Typen zu welchem Namespace gehören. Deswegen beschlossen wir weder „using namespace“ noch Defines für die Datentypen bzw. Namespaces zu erstellen. Namespaces die wir benutzen sind:

- ety (Namespace unserer Engine)
- sf (Namespace von SFML)
- std (Namespace der Standardbibliothek von C++)
- lua (Namespace der Script-Sprache LUA)
- boost (Namespace der Boost-Bibliothek für C++)

Was auch zur Folge hatte, dass Variablen immer diesen Namespace im Namen tragen, außer beim Namespace std.

Die Namensgebung bei Klassen und Typen:

- Klassen werden immer mit einem großen „C“ gekennzeichnet, z.B. CEternityEngine.

- Enums sollen immer mit „en_“ beginnen z.B. en_etyEventType. Des Weiteren bekommen Enums noch einen eigenen abgegrenzten Namespace innerhalb des Namespaces ety.

Variablen tragen immer eine Abkürzung des Datentyps in ihrem Namen, den Namespace, egal ob es ein Zeiger, eine Klasse, eine Struktur, ein Enum oder eine Membervariable ist.

Beispiele:

• bool	bActive	Eine einfache Bool-Variable.
• std::string	strCustomID	Eine Stringvariable.
• ety::Dialog	c_etyDialogBackground	Eine Klasse.
• ety::Dialog*	p_c_etyDialogBackground	Ein Zeiger auf eine Klasse.
• sf::Sprite	m_c_sfSpriteBackground	Eine Klasse von SFML als Membervariable.
• sf::Sprite*	mp_c_sfSpriteBackground	Das gleiche nur noch mal als Zeiger.

Bei manchen Variablen wurde auf einen expliziten Namen verzichtet, da dieser in dem Fall unnötig ist, da allein der Teil der den Typ der Variable beschreibt auch letztlich die Variable beschreibt. Was auch oft bei Enums gemacht wurde Beispiel: Anchor::en_etyAnchor en_etyAnchor.

Die wichtigsten Datentyp Abkürzungen sind:

- bool: „b" - bActive
- float „f" - fValue
- int „i" - iValue
- unsigned int „ui" - uiValue
- std::string „str" - strName

5.2. Verwaltung und Organisation des Projektes

Anfangs hat jeder seine Hauptbestandteile (Julien die Engine und Johannes das Dialogsystem) in einem eigenen Projekt programmiert. Später, als wir beide Teile in ein Projekt zusammenfügten, mussten wir eine gute Lösung finden, wie wir unseren Quellcode immer auf dem aktuellen Stand halten konnten und wie wir an den Partner immer unsere aktuellen Projektdateien schnell und einfach weitergeben konnten. Dazu benutzten/benutzen wir zwei wichtige Programme: Dropbox[22] und TortoiseSVN[23].

TortoiseSVN ist ein sehr mächtiges und nützliches Tool auf das uns Niels Becker, ein Mitschüler von uns, hingewiesen hat. Mit diesem Programm kann man Dateien mit einer Online-Datenbank wie z.B. Sourceforge[24] verbinden. Das Programm markiert veraltete oder veränderte Dateien. Des Weiteren kann man die aktuellen Projektdateien hoch- und runterladen, um den Partner und sich selbst auf dem neusten Stand zu halten. Der große Vorteil von SVN ist, dass wir beide an ein und derselben Datei arbeiten können und SVN setzt später beide Dateien richtig zusammen. Sollte es einmal zu Komplikationen kommen, speichert SVN beide Versionen der Dateien. Darüber hinaus bietet SVN noch viel mehr Möglichkeiten, z.B. auch das Wiederherstellen von alten Dateiverläufen.

Das zweite Programm was ebenfalls sehr nützlich ist, ist Dropbox. Dieses Programm speichert Daten in einer Cloud und verschiedene befugte Benutzer können dann auf die Dateien zugreifen. Wir haben die Dropbox deswegen eigerichtet, um unsere Projektdaten angemessen zu sichern, denn diese befinden sich in der Cloud und gleichzeitig auf jeden Computer der auch auf diese Dateien zugreifen kann. Aber der eigentliche Hauptgrund war, dass wir schnell und einfach alle möglichen Dateien hin und her tauschen konnten. Dadurch konnte unser Grafiker seine neuen Grafiken und Modelle einfach in die Dropbox legen und wir hatten alle Zugriff darauf, anders herum hatte er Zugriff auf die aktuellste Version des Projektes.

Wir ließen uns von Nils Rohde, einem weiteren Mitschüler, eine Website[25] einrichten. Auf diese schrieben wir den aktuellen Projektstand, an was wir gerade arbeiteten und wie es aufgebaut war. Diese Plattform nutzte hauptsächlich Johannes, aber auch nur unregelmäßig.

Ohne die beiden genannten Programme wäre wahrscheinlich ein reibungsloser Ablauf des Projektes so gut wie unmöglich gewesen. Ein wichtiges weiteres Programm war Skype[26]. Über dieses Programm haben wir zahlreiche Gespräche und Konferenzen geführt, sowie zusammen gearbeitet. Dabei war die Möglichkeit der Bildschirmübertragung sehr hilfreich.

[22] https://www.dropbox.com/
[23] http://tortoisesvn.net/
[24] http://sourceforge.net/
[25] http://outerspace.dead-man-rockin.de/
[26] http://www.skype.com/de/

6. Die Eternity-Engine (J. S.)

Die Eternity-Engine ist das Herz unseres Spieles. Die Eternity-Engine kümmert sich um alles, um das Spiel stabil und am Laufen zu halten. Sie besteht daher aus vielen Modulen, welche so programmiert wurden um miteinander zu arbeiten. Einige von den Modulen bzw. viele von ihnen sind so programmiert worden, dass sie auch als einzelnes Modul in anderen Projekten benutzt werden können. Solche Module sind:

- Das Gamestate-System
- Das Dialog-System
- Das Ressourcen-System
- Das Entity-System
- Die Application-Klasse

Unsere Engine ist eine Spiele-Engine, welche sich rein auf den 2-Dimensionalen Bereich konzentriert und die im Kapitel 4. gezeigten Bibliotheken benutzt. Das Ziel für die Eternity-Engine ist es, sie für 2-Dimensionale Weltraumspiele, mit einer hohen Modifizierbarkeit, erstellen zu können. Es ist uns wichtig, dass unser Spiel „Outerspace“, komplett modifizierbar ist. Dieses Ziel kann in dem jetzigen Zustand der Engine nur bedingt erfüllt werden. Das Programmieren einer Engine ist auch ein sehr schwieriges Unterfangen. Es musste „Refactoring[27]“ mit einer sehr hohen Rate betrieben werden, ganze Systeme wurden umgeschrieben und immer noch sind sehr viele Verbesserungen in Sicht. In Moment ist folgendes modifizierbar[28]:

- Das Laden der Ressourcen
- Das Laden der Spieleinstellungen
- Die Erstellung und das Hinzufügen der Karten
- Spielobjekte definieren und ihnen Eigenschaften zuweisen
- Ausrüstungsgegenstände definieren und ihnen Eigenschaften zuweisen
- Das Erstellen von Aufgaben und deren Eigenschaften auf den Karten
- Das Erstellen von Auslösern und deren Eigenschaften auf den Karten

Weitere Ziele der Eternity-Engine, in Bezug zur Modifizierbarkeit des Spieles, sind:

- Die Initialisierungen der Spielzustände
- Das Erstellen neuer Spielzustände
- Das Dialog-System
- Das Entity-System

[27] Es ist das Umschreiben/Verbessern einer Klasse oder eines Systems
[28] Veränderungen am Programm vornehmen ohne den Quellcode zu „berühren“

Neben den Systemen die bereits programmiert worden sind, gibt es noch weitere System die später hinzugefügt werden, diese werden aber nicht mehr bis zum Abschluss der besonderen Lernleistung hinzugefügt.

- Ein Animations-System
- Ein Partikel[29]-System
- Weitere Kollisionserkennungen
- Ein Shader[30]-System

Wir unterscheiden bei der Eternity-Engine zwischen den Engine-Elementen und den Spiel-Elementen. Bei den Spiel-Elementen beziehen wir uns auf Systeme/Klassen, welche nur ausdrücklich für das Spiel „Outerspace“ geschrieben wurden. Die Engine-Elemente sind die allgemeinen Systeme/Klassen, welche auch in anderen Projekten benutzt werden können. In dem folgenden UML-Diagramm[31] geht es um eine grobe Übersicht der Engine-Elemente. Aus Platz Gründen werden bei den unteren 3 Systemen nicht alle Klassen bzw. Beziehungen aufgezeigt:

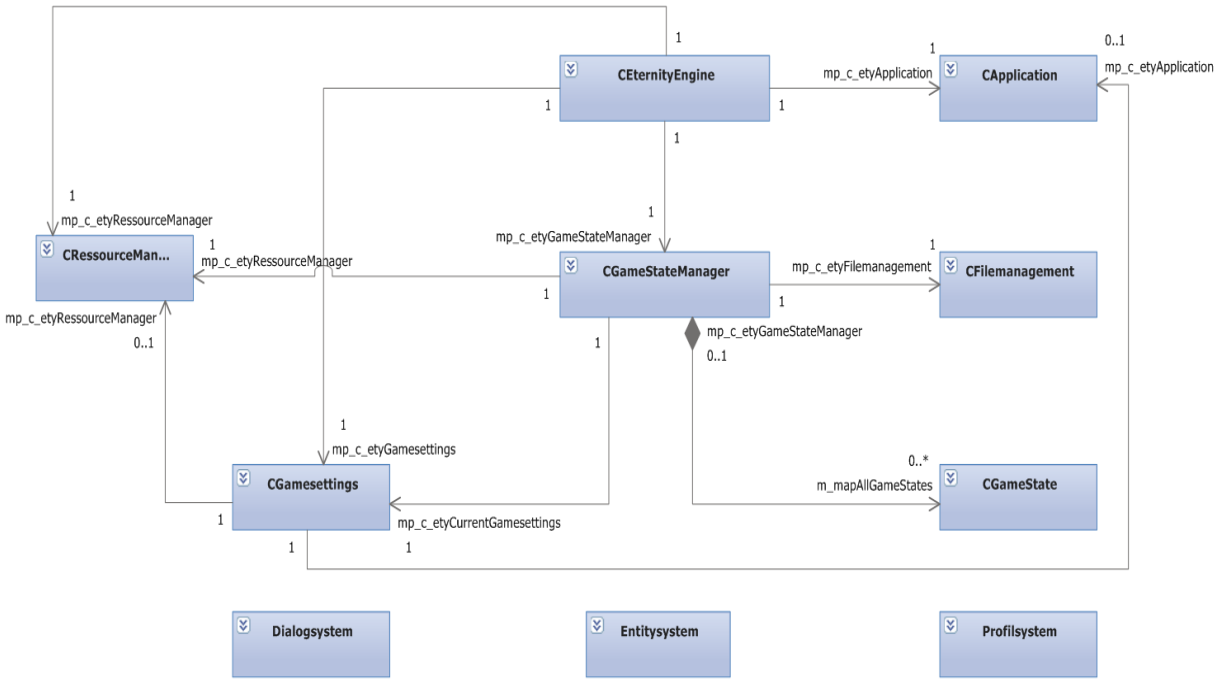


Abbildung 1 - UML-Diagramm: Grobe Übersicht auf die Eternity-Engine

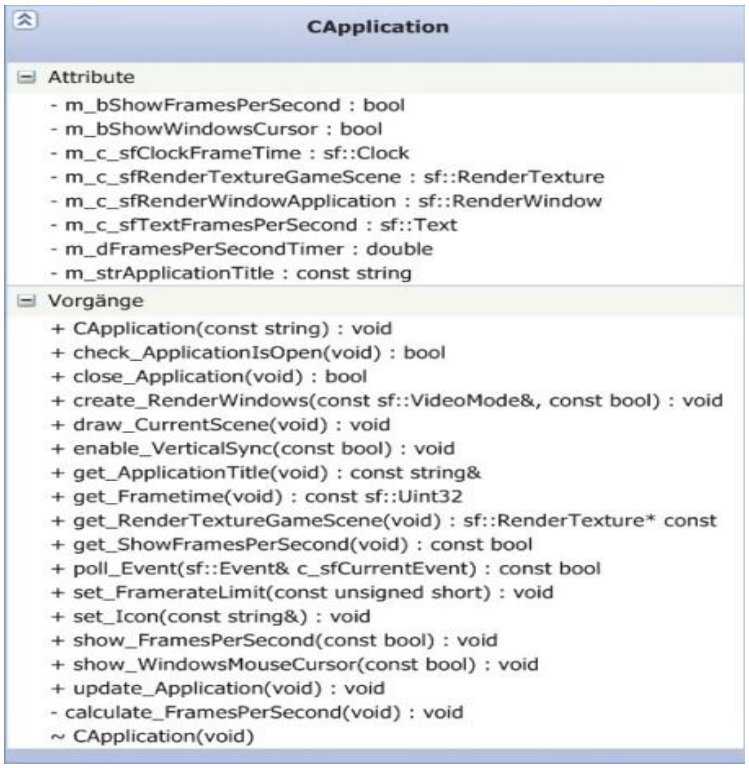
In den folgenden Kapiteln werden die einzelnen Systeme genauer erklärt. Es wird dargestellt, wofür die einzelnen Strukturen da sind, warum diese Strukturen gewählt wurden und wie sie endgültig implementiert worden sind. Danach geht es weiter mit dem Thema wie man unsere Eternity-Engine eigentlich benutzt werden kann, aber zu allererst wird die Fenster-Klasse erklärt.

[29] System, welches viele kleine Grafiken „aussprüht“, um zum Beispiel Feuer darzustellen
[30] System, welches eine bestehende Grafik mit zum Beispiel einem Leucht-Effekt belegt
[31] Die Unified Modeling Language ist ein standardisierte Modellierungssprache

6.1. CApplication

Die Klasse „CApplication“ ist unsere Fenster-Klasse, sie stellt das gesamte Fenster dar.
Mit ihr kann man den Titel des Fensters, das Icon, die Größe des Fensters, die Darstellung der maximalen Anzahl von Bildern in einer Sekunde, die vertikale Synchronisation und den Fenstermodus(z.B. Vollbildmodus) setzen. Mit ihr kann man ein individuelles Fenster erstellen und sie kümmert sich zusammen mit der „CEternityEngine“-Klasse darum, dass das Fenster erhalten bleibt und reagiert. Die Fenster-Klasse ist aber nicht nur zum Erstellen und Erhalten des Fensters da, sie zeichnet auch die aktuelle Szene des Spieles und speichert die Interaktion des Benutzers mit der Maus und Tastatur, um sie später in anderen Klassen weiterverarbeiten zu können. Eine wichtige Berechnung, um die Leistung des Programmes bzw. des PCs zu messen vollführt sie ebenfalls, diese kann im Spiel selber auch angezeigt werden, es ist die Berechnung der Bilder[32] in einer Sekunde, welche der PC schafft. Es ist ein Maß, um die Voraussetzungen des PCs zu messen oder um das Programm an bestimmten Stellen von der Leistung her zu verbessern. Wir richten uns nach 60 Bildern in der Sekunde als Minimum für das Programm, schafft das Programm unter 60 Bilder in der Sekunde schadet es der Flüssigkeit des Programmes. Der Grund dafür ist, dass nicht nur die Zeit des Aufbaus eines Bildes berechnet wird sondern zusätzlich alle Aktualisierungen und Interaktionen des Benutzers mit in diese Berechnung einfließen und somit heißt dies, dass bei einer geringeren Bilder in der Sekunde Anzahl, auch die Aktualisierungen und eventuell Interaktionen des Benutzers zu lange dauern. Diese Klasse kann in anderen Projekten, aufgrund ihrer unabhängigen Programmierung verwendet werden.

Abbildung 2 - Klassendiagramm: CApplication



[32] Auch Frames Per Second (kurz: „FPS“)

Das obige Klassendiagramm zeigt die Klasse „CApplication“ komplett. Schon in den Attributen erkennt man Datentypen im Namensbereich „sf“. Sie sind Datentypen bzw. Klassen von der SFML: sf::Clock[33], sf::RenderWindow[34], sf::RenderTexture[35] und sf::Text[36]. Zum Berechnen der Bilder in der Sekunde und um die vergangene Zeit seit dem letzten Bild[37], wird das Attribut „m_c_sfClockFrameTime“ von der Klasse sf::Clock benötigt. Die „Frametime“ wird zum Aktualisieren aller Spielzustände benötigt, da manche Instanzen die vergangene Zeit brauchen um zum Beispiel nach einer Sekunde etwas auszulösen. Die „sf::RenderTexture“ hat einen wichtigen Stellenwert, diese wird jedem Spielzustand ‚zum Zeichnen ihrer relevanten Szene, übergeben. Normalerweise wird die „sf::RenderWindow“-Instanz direkt benutzt zum Zeichnen der Szene, jedoch ist dieses Zeichnen direkt, das bedeutet selbst wenn einige Pixel von Grafiken sich überlappen werden alle Pixel gezeichnet, um dies zu verhindern werden alle Grafiken erst in die „sf::RenderTexture“-Instanz kopiert, was weitaus schneller geht als das Zeichnen und wird dann über die „sf::RenderWindow“-Instanz direkt gezeichnet. Somit wird verhindert dass unnötige Pixel gezeichnet werden.

Neben den Attributen befinden sich auch nennenswerte Methoden in der Klasse:

- void create_RenderWindow(const sf::VideoMode[38]&, const bool)
  - Mit dieser Methode wird das Fenster erstellt.
  - Der erste Parameter ist die gewünschte Auflösung, sie wird auch Kompatibilität mit dem Bildschirm überprüft bevor sie übernommen wird, sonst kann es zu ungewollten Fehlern kommen
  - Der zweite Parameter gibt an, ob das Fenster im Fenstermodus oder im Vollbildmodus erstellt werden soll

- void draw_CurrentScene(void)
  - Diese Methode zeichnet die aktuelle gespeicherte Szene, welche im Attribut derselben Klasse „m_c_sfRenderTextureGameScene“ gespeichert ist.

- const bool poll_Event(sf::Event[39]& c_sfCurrentEvent)
  - Ohne diese Methode, würde das Fenster auf keine Interaktion des Benutzers reagieren.
  - Die aktuelle Interaktion wird in der Referenz „c_sfCurrentEvent“ der Klasse „sf::Event“ gespeichert, um sie weiter an die Spielzustände geben zu können, diese werten die Interaktion aus, wenn sie denn relevant ist.

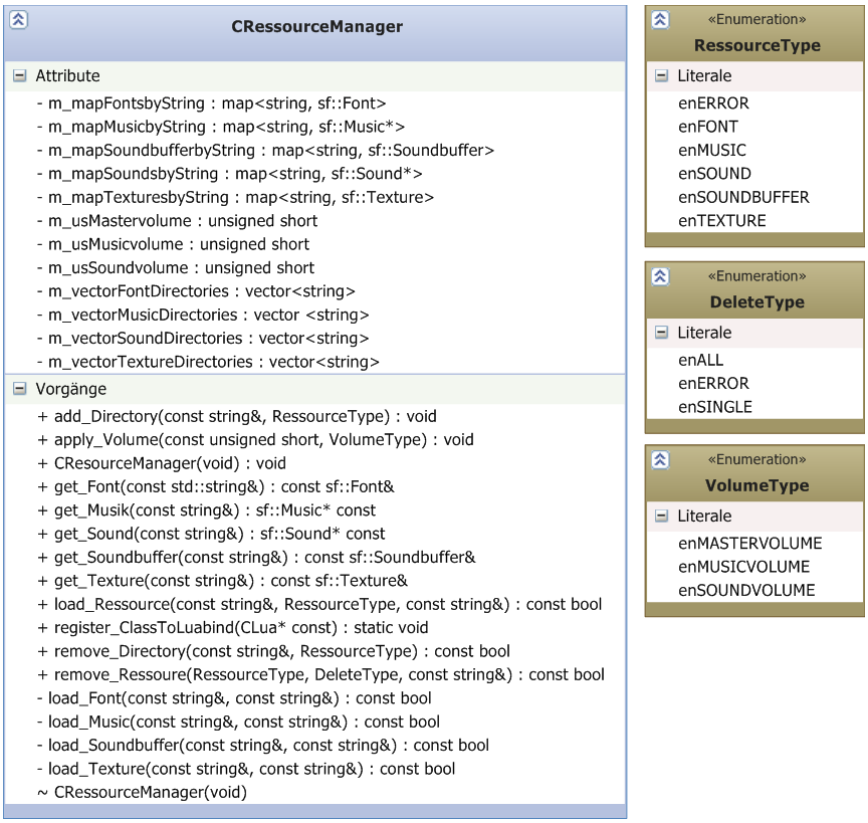
Als nächstes wird die Verwaltung der Ressourcen unter die Lupe genommen.

[33] Klasse der SFML: Sie speichert die vergangene Zeit
[34] Klasse der SFML: Sie stellt das Fenster dar und wird zum Zeichnen von Grafiken benötigt
[35] Klasse der SFML: Sie ist im Prinzip eine große Grafik in der man andere Grafiken reinkopieren kann
[36] Klasse der SFML: Sie stellt einen Text dar, welcher gezeichnet werden kann
[37] Im Englischen wird es die „frametime“ genannt, der Begriff wird ab diesen Punkt verwendet
[38] Klasse der SFML: In ihr wird die Auflösung gespeichert
[39] Klasse der SFML: Sie speichert die Interaktion des Benutzers mit der Maus und Tastatur

6.2. CRessourcenManager

Die Klasse „CRessourceManager“ verwaltet, lädt und speichert alle Ressourcen, darunter fallen Audio-Dateien, Bild-Dateien und Schriftarten. Nachdem man einige Ressourcen geladen hat, ist es dann möglich Ressourcen für die weitere Verwendung anzufordern. Dadurch dass Audio-Dateien eine Lautstärke besitzen und die Lautstärke kollektiv geändert werden muss, ist es auch möglich über diese Klasse die Lautstärken zu ändern. Die Ressourcen-Verwaltung benutzt ein Pfad-System, welches das Finden von Ressourcen schneller ausführen lässt. Es ist möglich dem Ressourcen-Manager Pfade zu übergeben bzw. zu registrieren und diese werden ebenfalls alle durchsucht beim Suchen der Ressourcen. Das hat den Nachteil, dass die Ressourcen-Verwaltung mehre Male versucht die Ressourcen an Stellen zu laden, wo sie nicht existieren, um das zu kompensieren wird ein „Ordner-Schlüssel“ benötigt. Dieser „Ordner-Schlüssel“ muss ein Teil des Pfades sein in dem die Ressource sich befindet. Der Ressourcen-Manager such die registrierten Pfade nach diesem „Ordner-Schlüssel“ ab und wenn ein passender Pfad gefunden wurde oder auch mehrere, werden nur in den Pfaden versucht die Ressource zu laden. Auch diese Klasse ist in jedem anderen Projekt ohne Bedenken benutzbar. Im folgenden Diagramm sieht man die volle Struktur der „CRessourceManager“-Klasse.

Abbildung 3 - Klassendiagramm: CRessourceManager



Die „CRessourceManager“-Klasse ist an sich nicht besonders spektakulär, sie besitzt sehr viele Container zum Verwalten der Ressourcen und speichert die Lautstärke der Audio-Ressourcen. Zum Anfordern der einzelnen Ressourcen müssen die einzelnen „Get“-Methoden aufgerufen werden. Neben den „Get“-Methoden, gibt es die Methoden zum Laden der Ressourcen und zum Hinzufügen der Pfade, ebenso um die Ressourcen oder Pfade wieder zu löschen.

Wie das Laden eigentlich funktioniert, zeigt das folgende Struktogramm am Beispiel der Funktion „load_Texture“:

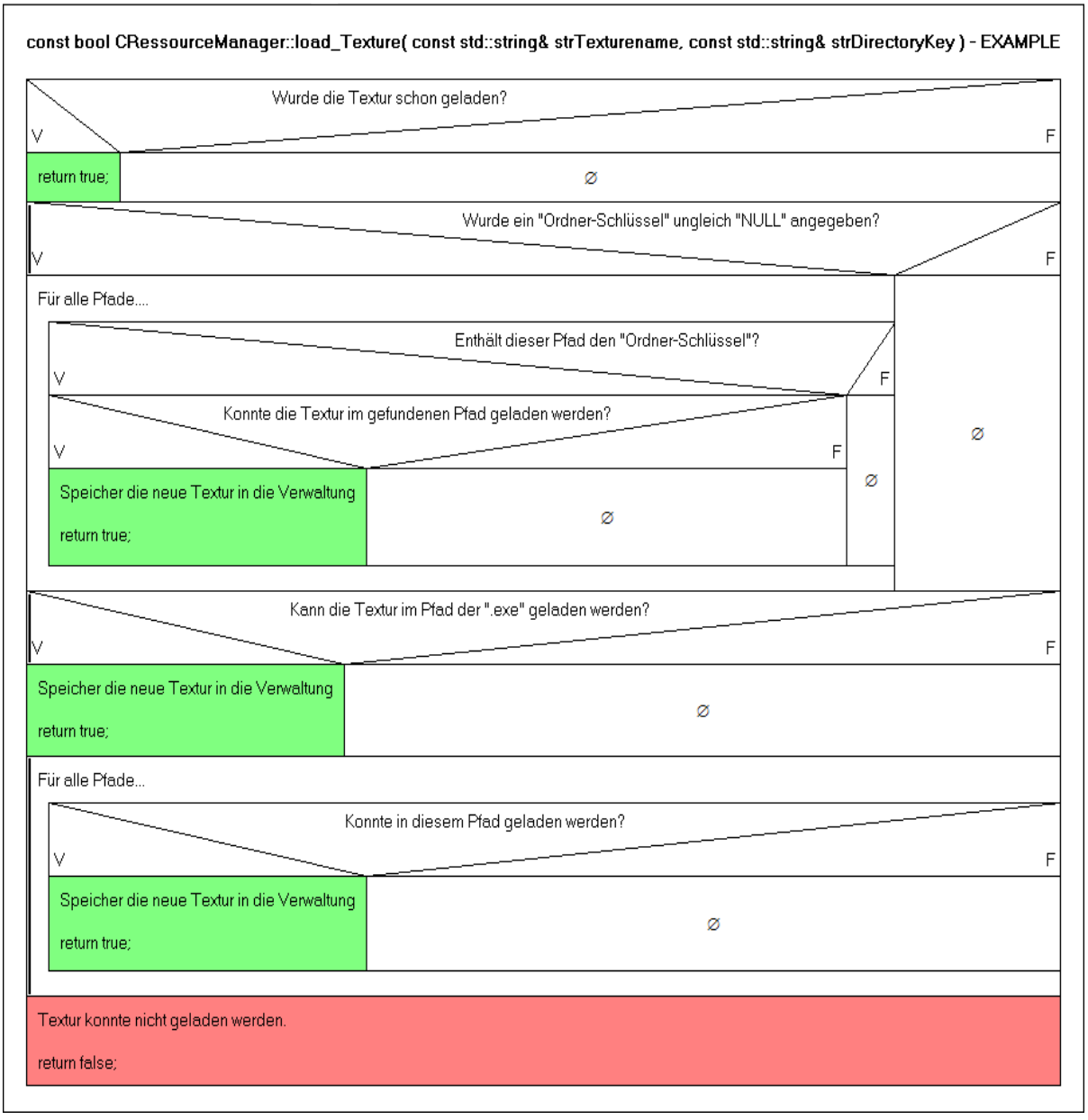


Abbildung 4 - Das Laden von Ressourcen (Struktogramm)

Als erstes werden alle Pfade, welche den „Ordner-Schlüssel“ enthalten, durchsucht und es wird überprüft ob die Ressource in den Pfaden geladen werden kann, scheitert dies wird der Pfad des Programms untersucht auf die Ressource. Findet die Ressourcen-Verwaltung auch dort die gesuchte Ressource nicht, werden alle Pfade auf die Ressource durchsucht. Neben der Besonderheit des Ladens, ist die Klasse „CRessourceManager“ mit Lua verbunden. Dies erkennt man an der statischen Methode „register_ClassToLuabind“. Diese Methode registriert die Klasse und ihre Methoden in einem Script, in welchem man nun die Methoden relativ zu einer Referenz dieser Klasse aufrufen kann. Wie das genauer funktioniert sehen wir in der nächsten Abbildung.

```
void ety::CRessourceManager::register_ClassToLuabind( ety::CLua* const p_c_etyLuaScript )
{
    luabind::module(p_c_etyLuaScript->get_LuaState() )
    [
        luabind::class_< CRessourceManager >("CRessourceManager")
        .def( "get_Texture"         , &CRessourceManager::get_Texture       )
        .def( "get_Font"            , &CRessourceManager::get_Font          )
        .def( "get_Music"           , &CRessourceManager::get_Music         )
        .def( "get_Sound"           , &CRessourceManager::get_Sound         )
        .def( "get_Soundbuffer"     , &CRessourceManager::get_Soundbuffer   )
        .def( "add_Directory"       , &CRessourceManager::add_Directory     )
    ];
}
```

Abbildung 5 - Registrierung einer Klasse mit Luabind (CRessourceManager.cpp)

Um diese Methode nutzen zu können, muss ein Zeiger auf eine Instanz eines Lua-Scriptes[40] übergeben werden. Mit Hilfe der Luabind-Bibliothek kann nun die Klasse dem übergebenen Script registriert werden. Die Luabind-Funktion „luabind::class_<>()“ ist für die Bekanntmachung der Klasse da, jede der nachfolgenden „.def()“-Funktionen, sind für die Bekanntmachung der Methoden dieser Klasse zuständig. Es ist somit möglich nur bestimmte Methoden oder Attribute dem Script kenntlich zu machen. Die nächsten zwei Abbildungen zeigen, wie der Aufruf des Scriptes ausschaut und wie die Script-Datei aussehen kann.

```
const   bool                        ety::CEternityEngine::init_Engine               ( void )
{
    //Initialisieren von LUA und LUABIND.
    CLua c_etyLuaScript;
    c_etyLuaScript.init_Lua();

    //Registriere die Klassen mit Hilfe von LUABIND.
    //Diese Klassen können dann im Script benutzt werden.
    ety::CRessourceManager::register_ClassToLuabind( &c_etyLuaScript );
    ety::CFilemanagement::register_ClassToLuabind( &c_etyLuaScript );

    //An LUA die gewünschten Objekte zum bearbeiten weiter geben.
    luabind::globals( c_etyLuaScript.get_LuaState() )["c_etyRessourceManager"   ]   = mp_c_etyRessourceManager;
    luabind::globals( c_etyLuaScript.get_LuaState() )["c_etyFilemanager"        ]   = mp_c_etyGameStatemanager->get_Filemanagement();

    //Starten des LUA-Scriptes.
    c_etyLuaScript.start_Script( "Initialisation/Gamestates/RessourceManagement.lua" );

    //Schließen von LUA.
    c_etyLuaScript.close_Lua();
```

Abbildung 6 - Beispiel Aufruf eines Scripts

In der Abbildung 6 ist zu erkennen wie die Lua-Instanz initialisiert wird, es ist essentiell für die Verwendung von einem Script, denn es werden alle wichtigen Vorbereitungen

[40] In diesem Fall stellt die selbst erstellte Klasse „CLua“ unser Script dar

der Lua-Bibliothek für die Verwendung von einem Script getroffen. Nachdem man die Initialisierung abgeschlossen hat, geht es weiter mit der Registrierung der benötigten Klassen. Meist möchte man in einem Script bereits bestehende Instanzen von Klassen weiter bearbeiten bzw. verwenden, im nächsten Schritt werden diese Instanzen dem Script bekannt gemacht. Zu allerletzt wird nur noch das Script gestartet und wieder geschlossen zur weiteren Verwendung. In der nächsten Abbildung wird die Verwendung im Script selber gezeigt.

```
if c_etyRessourceManager ~= nil then

    c_etyRessourceManager:add_Directory ( "Media/Textures/Dialog/Backgrounds/"          ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Dialog/Buttons/"              ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Dialog/Checkboxes/"           ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Dialog/ComboBoxes/"           ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Dialog/EditControls/"         ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Dialog/PictureControls/"      ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Dialog/Progressbar/"          ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Dialog/Scrollbars/"           ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Dialog/SlideControls/"        ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Effects/Shots/"               ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Cursor/"                      ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Entities/Planets/"            ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Entities/Spaceships/"         ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Entities/Meteors/"            ,   0   );
    c_etyRessourceManager:add_Directory ( "Media/Textures/Entities/Weapons/"            ,   0   );

    c_etyRessourceManager:add_Directory ( "Media/Fonts/"                               ,   1   );

    c_etyRessourceManager:add_Directory ( "Media/Audio/Music/"                         ,   3   );

    c_etyRessourceManager:add_Directory ( "Media/Audio/Sounds/"                        ,   4   );

end
```

Abbildung 7 - Erster Teil des Scripts der Ressourcen-Verwaltung(RessourceManagement.lua)

In dieser Abbildung ist die Verwendung der bekannt gemachten Instanz der Klasse „CRessourceManager“ im Script zu sehen. Dieses Script ist relativ unkompliziert, denn hier werden nur Pfade zum Laden der Ressourcen hinzugefügt. Man kann an diesem Beispiel jedoch sehr gut erkennen wie das Laden von außen beeinflusst werden kann. Auf die Feinheiten der Scriptsprache „Lua“ wird nicht weiterhin eingegangen. Es ist aber wichtig zu erkennen, dass dieselbe Instanz wie im richtigen Code[41] auch im Script benutzt wird und ebenso dieselben Methoden wie das Klassendiagramm der Klasse „CRessourceManager“ zeigt.

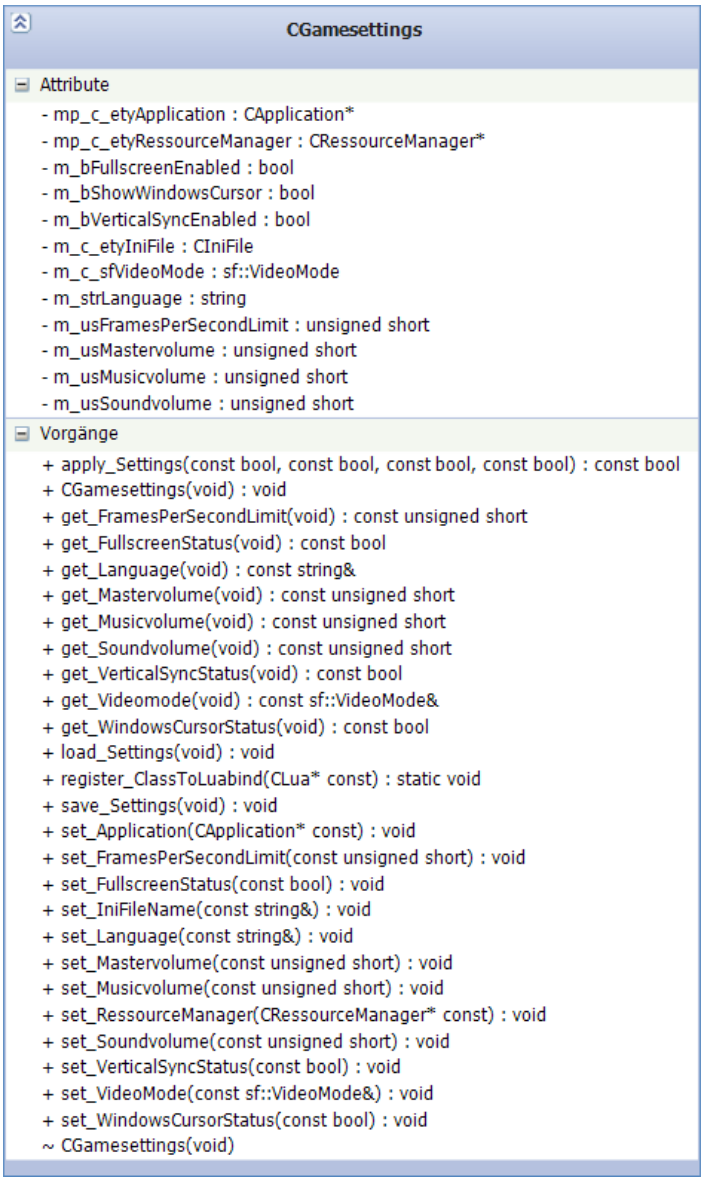
Als nächstes wird genauer erklärt wie Einstellungen gespeichert werden mit Hilfe der CGamesettings-Klasse.

[41] Damit ist der richtige Programmiercode gemeint, nicht die Script-Dateien

6.3. CGamesettings

Die Klasse „CGamesettings“ spiegelt die aktuellen ausgewählten Spieleinstellungen wieder. Mit ihrer Hilfe kann man Spieleinstellungen setzen und übernehmen, sie laden oder speichern. Das Setzen von Einstellungen kann auf zwei verschiedene Methoden vollbracht werden, entweder über die „Set-Methoden“ oder über eine INI-Datei[42]. Wird kein Pfad für eine INI-Datei angegeben, ist es nicht möglich Einstellungen zu laden oder zu speichern. Diese Klasse ist nicht allein benutzbar, sie benötigt Referenzen zu den Klassen „CApplication“ und zum „CRessourceManager“, um seine Einstellungen(z.B. Auflösung oder die Lautstärke von Audio-Dateien) übernehmen zu können. Sind diese beiden Referenzen nicht gegeben, kann man Einstellungen zwar setzen, aber sie können nicht übernommen werden.

Abbildung 8 - Klassendiagramm: CGamesettings



[42] Damit ist eine Initialisierungsdatei gemeint, welche in einem speziellen Format Wertepaare beherbergt

Wie im Klassendiagramm zu sehen ist, sind die meisten Attribute und ihre Bedeutung her einfach zu verstehen:

- „m_bFullscreenEnabled“
  - Bei dem Wert „true“ – Vollbildmodus an
- „m_bShowWindowsCursor“
  - Bei dem Wert „true“ – Windows-Mauszeiger wird angezeigt
- „m_bVerticalSynchEnabled“
  - Bei dem Wert „true“ – Die vertikale Synchronisation[43] wird angeschaltet
- „m_c_sfVideoMode“
  - Speichert die Auflösung
- „m_strLanguage“
  - Speichert die verwendete Sprache
- „m_usFramesPerSecondLimit“
  - Die Begrenzung der gezeigten Bilder in einer Sekunde
- „m_isMusicvolume“
  - Speichert die Musik-Lautstärke
- „m_isSoundvolume“
  - Speichert die Effekt-Lautstärke
- „m_isMastervolume“
  - Speichert die allgemeine Lautstärke

Zu jeden dieser Attribute gibt es jeweils eine „Set“- und „Get“-Methode. Die einzige zu erwähnende Methode ist die „apply_Settings“-Methode. Bei der Veränderung der Einstellungen ist etwas Wichtiges zu beachten, sie werden nicht übernommen bzw. das Programm reagiert nicht auf die Veränderungen, erst mit dem Aufruf der „apply_Settings“-Methode. Sie bringt das Programm dazu die Einstellungen zu übernehmen, es ist mit Hilfe dieser Methode und ihren vier boolean-Übergabewerten auch möglich nur einzelne Einstellungen zu übernehmen.

Möchte man die aktuellen Einstellungen speichern oder Einstellungen laden, muss der Pfad zur INI-Datei mit Hilfe der „set_IniFileName“-Methode gesetzt werden. Die INI-Datei selber muss so aussehen:

[43] Verhindert das aktualisieren der Daten des Bildes, während die Grafikkarte das vorherige Bild noch zeichnet

Abbildung 9 - Die INI-Datei für die Spieleinstellungen

```
[DISPLAY]
iScreenWidth=800
iScreenHeight=600
iFramesPerSecondLimit=60
bVerticalSync=0
bWindowsCursor=0
bFullscreenMode=0
[SOUND]
iMasterVolume=100
iMusicVolume=100
iSoundvolume=100
[LANGUAGE]
strLanguage=English
```

Die INI-Datei ist in Sektionen unterteilt, eine von ihnen ist „DISPLAY“. In diesem Fall ist die „DISPLAY“-Sektion für das Fenster zuständig, die „SOUND“-Sektion für die Lautstärken und die letzte Sektion „LANGUAGE“ für die Sprache. Mit folgendem Code werden diese Einstellungen geladen, das Speichern sieht ähnlich aus:

```
void                ety::CGamesettings::load_Settings           ( void )
{
    //Sektion [DISPLAY]...
    set_Videomode           ( sf::VideoMode(    m_c_etyIniFile.read_Integer( "DISPLAY"  , "iScreenWidth"    , sf::VideoMode::getFullscreenModes()[0].width ),
                                                m_c_etyIniFile.read_Integer( "DISPLAY"  , "iScreenHeight"   , sf::VideoMode::getFullscreenModes()[0].height ) ) );

    set_FullscreenStatus    ( m_c_etyIniFile.read_Boolean( "DISPLAY"    , "bFullscreenMode"     , true      ) );
    set_VerticalSyncStatus  ( m_c_etyIniFile.read_Boolean( "DISPLAY"    , "bVerticalSync"       , false     ) );
    set_WindowsCursorStatus ( m_c_etyIniFile.read_Boolean( "DISPLAY"    , "bWindowsCursor"      , false     ) );
    set_FramesPerSecondLimit( m_c_etyIniFile.read_Integer( "DISPLAY"    , "iFramesPerSecondLimit"   , 60    ) );

    //Sektion [SOUND]...
    set_Mastervolume        ( m_c_etyIniFile.read_Integer( "SOUND"  , "iMasterVolume"           , 100   ) );
    set_Musicvolume         ( m_c_etyIniFile.read_Integer( "SOUND"  , "iMusicVolume"            , 100   ) );
    set_Soundvolume         ( m_c_etyIniFile.read_Integer( "SOUND"  , "iSoundvolume"            , 100   ) );

    //Sektion [LANGUAGE]...
    set_Language            ( m_c_etyIniFile.read_String( "LANGUAGE", "strLanguage"             , "English" ) );
}
```

Abbildung 10 - Laden der Spieleinstellungen

Die Einstellungen werden, mit der Hilfe einer selbst geschriebenen INI-Klasse(„CIniFile“), zum Auslesen und zum Schreiben in eine INI-Datei, gespeichert oder geladen. Die „write“- und „read“-Methoden der INI-Klasse sind leicht zu verstehen. Als erstes übergibt man die Sektion, dann der Schlüssel aus welchen Ausgelesen werden soll bzw. in welchen geschrieben werden soll. Der letzte Übergabeparameter ist beim Lesen der Standard-Wert welcher zurückgegeben werden soll, wenn das Auslesen schief läuft, beim Schreiben ist es der Wert, der in den Schlüssel reingeschrieben werden soll.

Das nächste Kapitel erklärt wie die Eternity-Engine die einzelnen Spielzustände des Spiels verwaltet.

6.4. GameState-System

Zu Beginn der Entwicklung der Eternity-Engine bestand das Problem, wie verwaltet man übersichtlich und leicht erweiterbar, das Spiel an sich selber. Die Entscheidung fiel auf ein System, welches das Programm in verschiedene ausführbare Spielzustände unterteilt. Die Idee hierbei ist, diese ausführbaren Spielzustände in ein übergeordnetes System zu registrieren und nur dann auszuführen, wenn dieser Spielzustand gebraucht wird. Somit ist es möglich das Spiel in mehrere Teile aufzuteilen und diese Teile nach „Aufgabe“ oder „Funktion“ zu unterteilen. Somit kann ein Spielzustand für das Hauptmenü zuständig sein und ein anderer ist zuständig für das Starten und Ausführen des Spieles. So ein Spielzustand nennt sich in der Eternity-Engine „CGameState“ und das übergeordnete System „CGameStateManager“.

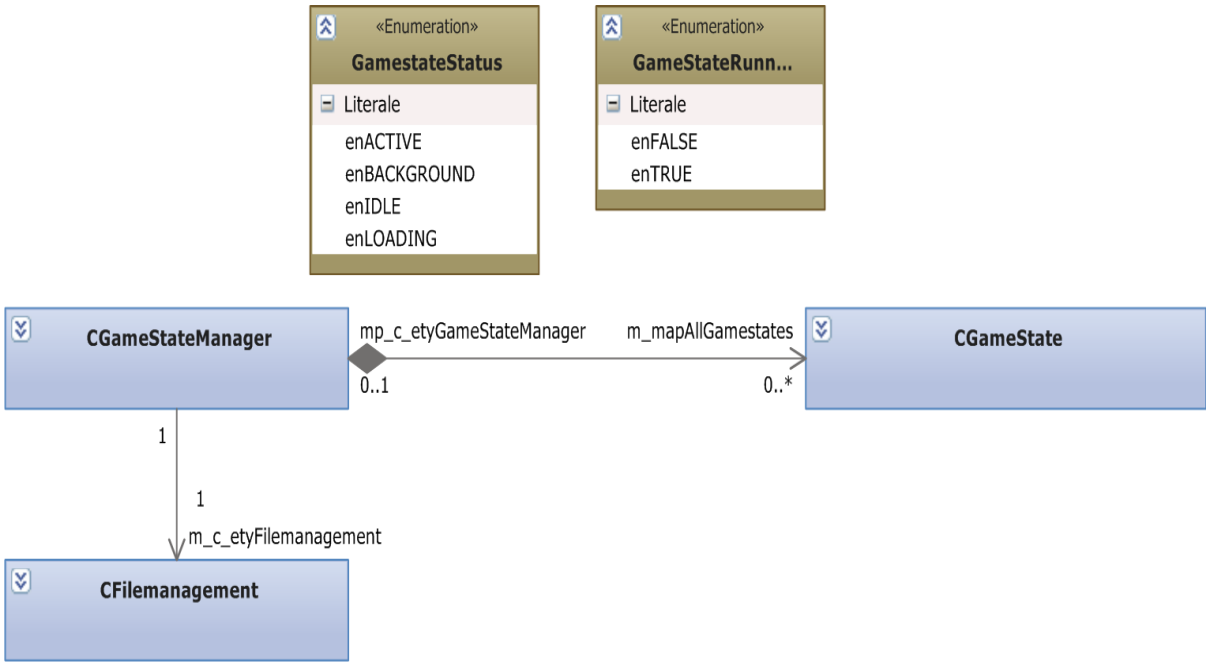


Abbildung 11 - UML-Diagramm: Gamestate-System

Wie man in der Abbildung 11 erkennt ist nicht nur das Zusammenspiel der Klassen „CGameState“ und „CGameStateManager“ wichtig. Die Klasse „CFilemanagement“ kümmert sich darum, dass die benötigten Ressourcen für die Spielzustände auch verfügbar sind. Die Klasse „CFilemanagement“ ist eine etwas komplexere Klasse und wird daher erst am Schluss dieses Systems erklärt. Als erstes wird die Klasse „CGameState“ genauer erklärt, denn sie ist recht leicht zu verstehen und ist dennoch sehr wichtig für das System.

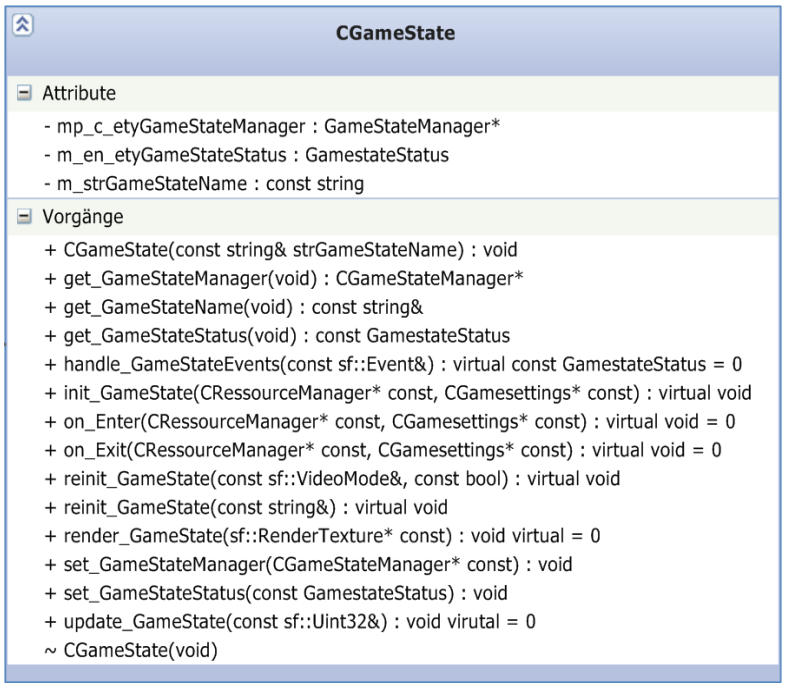
6.4.1. CGameState

Die „CGameState“-Klasse stellt einen Spielzustand dar, solch ein Spielzustand kann im Prinzip jede Funktion/Arbeit annehmen. Diese Klasse wurde so programmiert, dass sie für alle Spielzustände des Spieles als Basisklasse fungiert. Sie beherbergt eine große Palette an reinen abstrakten Methoden[44], um ihre Arbeit zu gewährleisten. Sie ist demnach eine fast leere Klasse, welche unbedingt abgeleitet werden muss, um ihren Zweck zu erfüllen. Alle unsere Spielzustände sind von dieser Klasse abgeleitet und füllen die reinen abstrakten Methoden der Basisklasse „CGameState“ aus. Diese Klasse ist nur mit dem Gamestate-Manager richtig benutzbar. Auch wenn diese Klasse fast nichts beherbergt, muss jeder Spielzustand von einander unterscheidbar sein, beim Erstellen eines solchen Spielzustandes muss darauf geachtet werden, dass dieser einen einzigarten Namen besitzt, denn die Spielzustände werden im Gamestate-Manager über ihren Namen angesprochen. Neben dem Namen kann ein Spielzustand selber noch einen Zustand annehmen. Dieser Zustand ist aber keine Instanz von „CGameState“. Ein Spielzustand kann einen der folgenden Zustände annehmen:

- „enBACKGROUND“
- „enLOADING“
- „enIDLE“
- „enACTIVE“

Diese Zustände sind nur im Zusammenhang mit dem Gamestate-Manager relevant, da dieser je nach Zustand des Spielzustandes anders agiert. Sie werden deshalb erst im Kapitel 5.4.2. erklärt. In der folgenden Abbildung wird das Klassendiagramm aufgezeigt:

Abbildung 12 - Klassendiagramm: CGameState



[44] Methoden bzw. Funktionen einer Klasse, welche nach einer gezwungenen Ableitung unbedingt implementiert werden müssen.

In der Abbildung 12 ist es deutlich zu sehen, dass die Klasse „CGameState“ fast nur aus reinen abstrakten Methoden besteht. Der Grund für diese Struktur ist die vielseitige Benutzung von einem Spielzustand. Der Nachteil dieser Struktur ist, dass es eine große Arbeit sein kann, einen neuen Spielzustand zu erstellen, weil alle Methoden komplett neu implementiert werden müssen. Die wichtigsten Methoden werden nun erklärt:

- „virtual void init_GameState(CRessourceManager* const, CGamesettings* const)”
  - Sie ist keine reine virtuelle Methode, weil die Basis-Methode ein Script zum initialisieren aufruft, dieser Script trägt denselben Namen wie der Spielzustand.
  - Die Methode wird nach dem Laden der Ressourcen für alle registrierten Spielzustände aufgerufen

- „virtual void on_Enter(CRessourceManager* const, CGamesettings* const) = 0“
  - Zum Initialisieren des Spielzustandes, wenn dieser aktiviert wird

- „virtual void on_Exit(CRessourceManager* const, CGamesettings* const) = 0“
  - Zum Freigeben oder Zurücksetzen von Instanzen des Spielzustandes, wenn dieser deaktiviert wird

- „virtual const GameStateStatus handle_GameStateEvents(const sf::Event&) = 0“
  - Diese Methode kümmert sich um die Interaktionen des Benutzers mit dem Programm
  - Wird vom Gamestate-Manager bei jedem Schleifen-Durchlauf als erstes aufgerufen

- „virtual void update_GameState(const sf::Uint32&) = 0”
  - Sie aktualisiert alle relevanten Instanzen im Spielzustand
  - Wird vom Gamestate-Manager bei jedem Schleifen-Durchlauf nach der Methode „handle_GameStateEvents“ aufgerufen

- „virtual void render_GameState(sf::RenderTexture* const) = 0”
  - Sie zeichnet alle relevanten Instanzen im Spielzustand
  - Wird vom Gamestate-Manager bei jedem Schleifen-Durchlauf nach der Methode „update_GameStateEvents“ aufgerufen

Die oberen Methoden sind für den reibungslosen Ablauf des Spielzustandes zuständig. Dadurch dass sie aber vom Programmierer selbst implementiert werden müssen, ist darauf zu achten, die Methoden richtig für ihren Zweck zu nutzen. Es wurde nun häufig der Gamestate-Manager erwähnt, um diesen geht es im nächsten Abschnitt.

6.4.2. CGameStateManager

Die „CGameStateManager“-Klasse verwaltet die Spielzustände. In ihr werden Spielzustände registriert und aufgerufen. Durch die Zusammenarbeit von „CGameState“-Instanzen und der Instanz der „CGameStateManager“-Klasse entsteht erst das Programm. Ohne einen registrierten Spielzustand, wird die Eternity-Engine nicht lange das Fenster aufrechterhalten und das Programm beenden. Der Gamestate-Manager kümmert sich im Grunde um die Aktualisierung des Spielzustandes, das Zeichnen der relevanten Spielzustands-Objekte und die Interaktion des Benutzers mit dem Spielzustand. Ist mindestens ein Spielzustand registriert, kann dieser aufgerufen werden, ohne einen Aufruf eines Spielzustandes, wird das Programm sich ebenfalls beenden. Es ist möglich im Gamestate-Manager mehrere Spielzustände gleichzeitig laufen zu lassen. Im vorherigen Abschnitt wurden die Zustände eines Spielzustandes erwähnt, diese Zustände haben Einfluss auf die Arbeit des Gamestate-Managers. Je nach Zustand wird die Arbeit beeinträchtigt oder erweitert.

Trägt ein Spielzustand den Zustand:

- „enACTIVE“
  - Mit diesem Spielzustand kann der Benutzer interagieren
  - Dieser Spielzustand wird aktualisiert
  - Die Szene dieses Spielzustandes wird gezeichnet

- „enBACKGROUND“
  - Dieser Spielzustand wird aktualisiert
  - Die Szene dieses Spielzustandes wird gezeichnet

- „enIDLE“
  - Dieser Zustand wird nicht mehr verwaltet

- „enLOADING“
  - Dieser Zustand sagt aus, dass die Ressourcen vorerst geladen werden müssen, mit Hilfe des Ressourcen-Systems
  - Ist das Laden beendet wird dieser Spielzustand sofort aktiv gesetzt

Jedoch ist folgendes zu beachten:

- Nur ein Spielzustand kann aktiv sein (Er trägt den Zustand „enACTIVE“)

- Sobald ein Spielzustand aktiv ist, können alle anderen nur noch die Zustände „enIDLE“ oder „enBACKGROUND“ annehmen

- Besitzt ein Spielzustand den Zustand „enIDLE“, wird er aus der Liste zur Verwaltung rausgenommen, er ist aber immer noch registriert und kann dementsprechend aktiviert werden

Weiterhin ist beim Aktivieren/Deaktivieren eines Spielzustandes folgendes zu beachten:

- Einen Spielzustand aktivieren
  - Ist bereits ein Spielzustand aktiv, kann dieser den Status „enIDLE“ oder „enBACKGROUND“ annehmen, erst dann wird der gewünschte Spielzustand aktiv

- Den aktiven Spielzustand wieder deaktivieren
  - Sobald noch ein Spielzustand in der Liste der Verwaltung steht - dieser trägt den Zustand „enBACKGROUND“ - wird dieser seinen Zustand zu „enACTIVE“ wechseln

In der Abbildung 13 wir das Klassendiagramm des Gamestate-Managers aufgezeigt.

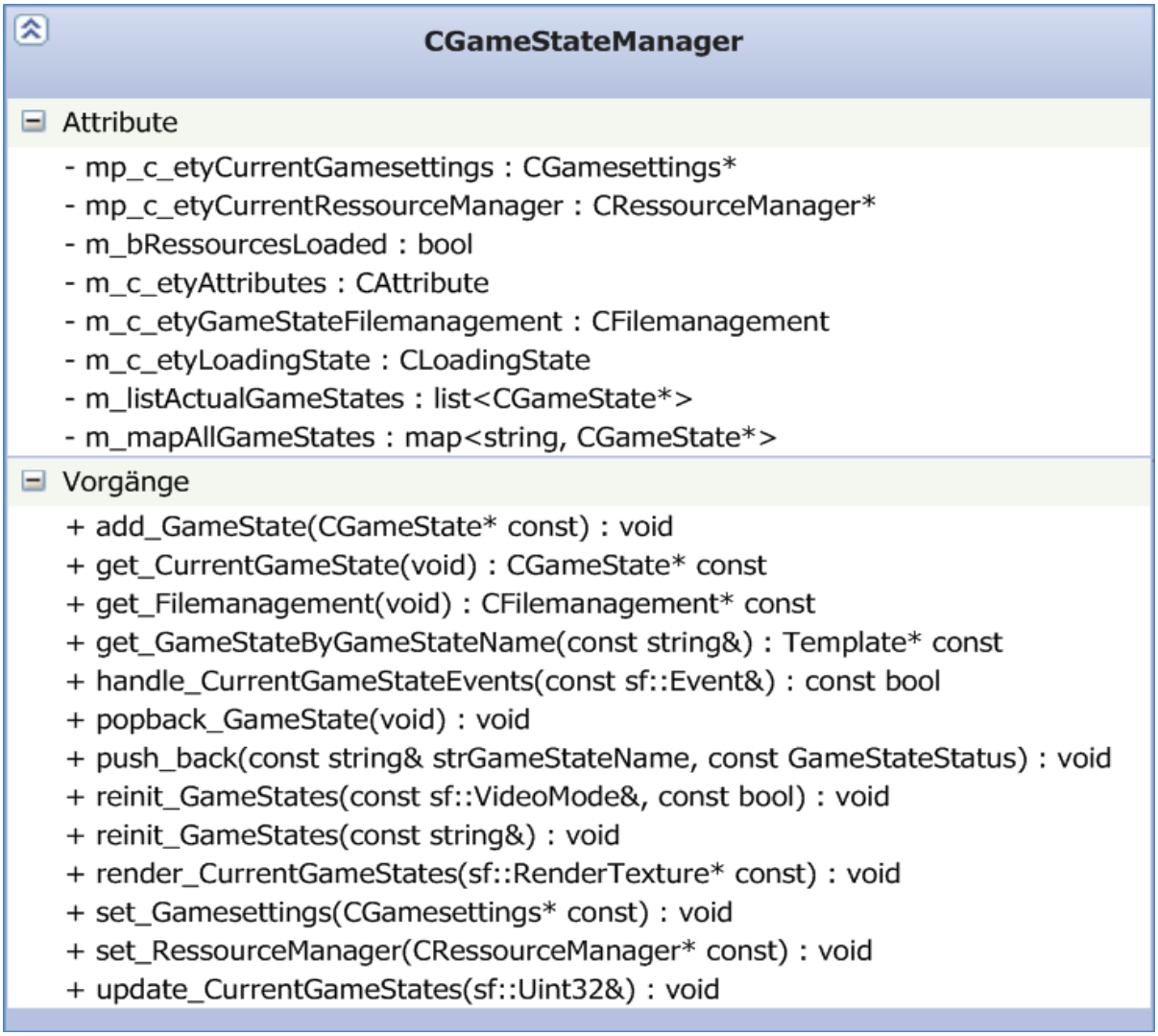


Abbildung 13 - Klassendiagramm: CGameStateManager

Die Klasse „CGameStateManager“ ist schon ein wenig umfangreicher als die „CGameState“-Klasse, aber als erstes wird die Verwaltung der „CGameState“-Instanzen mehr erläutert, bevor die einzelnen Methoden der Klasse „CGameStateManager“ erklärt werden.

Wie in der Abbildung 13 zu erkennen ist, gibt es zwei STL-Container für die Verwaltung der Spielzustände. Das Attribut „m_mapAllGameStates“ beherbergt alle registrierten Spielzustände unabhängig von ihren derzeitigen Zustand. Die Liste „m_listActualGametates“ ist gefüllt mit dem aktiven oder den im Hintergrund laufenden Spielzustände. In der nächsten Abbildung wird es deutlicher:

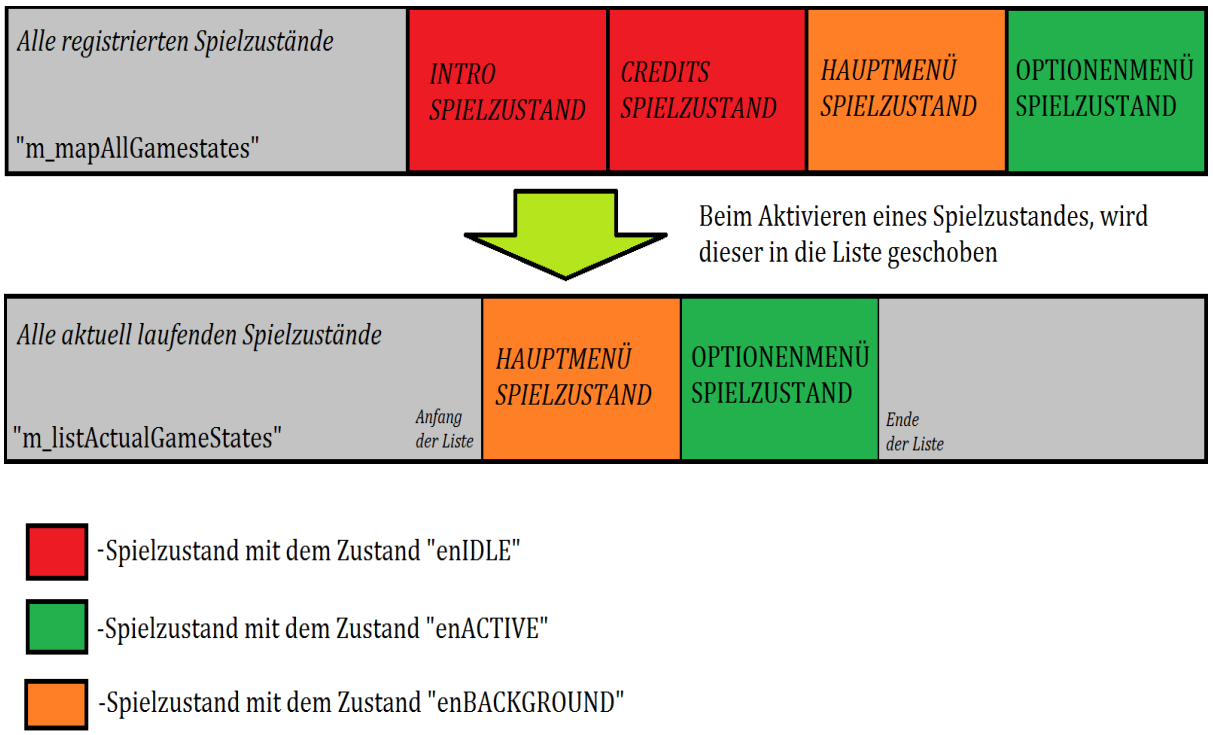


Abbildung 14 - Verwaltung der Spielzustände

Es kann oben aus der Abbildung abgenommen werden, dass der aktive Spielzustand immer als letztes in der Liste der aktuell laufenden Spielzustände steht. Alle anderen Spielzustände werden vorne nach Reihenfolge ihrer Aktivierung mit dem Zustand „enBACKGROUND“ eingereiht, wenn es denn gewünscht ist. Die Methoden zum Aktivieren/Deaktivieren und zum Registrieren der Spielzustände werden im Folgenden erklärt:

- „void add_GameState(CGameState* const)“
  - Diese Methode registriert einen Spielzustand
  - Übergabe ist ein Zeiger auf eine bestehende „CGameState“-Instanz

- „void pushback_GameState(const string&, const GameStateStatus)”
  - Sie aktiviert einen Spielzustand
  - Erster Parameter: Der Name des zu aktivierenden Spielzustandes
  - Zweiter Parameter: Der Zustand des bestehenden aktiven Spielzustandes

- „void popback_GameState(void)”
  - Methode zum Deaktivieren des aktiven Spielzustandes und aktiviert den nächsten in der Liste liegenden Spielzustand, wenn einer vorhanden ist

Neben den Methoden zum Verwalten der Spielzustände gibt es noch eine Besonderheit in der "CGameStateManager"-Klasse. Während der Entwicklung der Eternity-Engine, wurde die Möglichkeit gebraucht zwischen Spielzuständen Informationen auszutauschen. Die Instanz der Klasse „CAttribute" kann dafür ausgenutzt werden. Sie ist einer unserer vielen Unterstützungsklassen, aber auch eine relativ komplexe Klasse. Die nächste Abbildung zeigt das Klassendiagramm der Klasse „CAttribute":

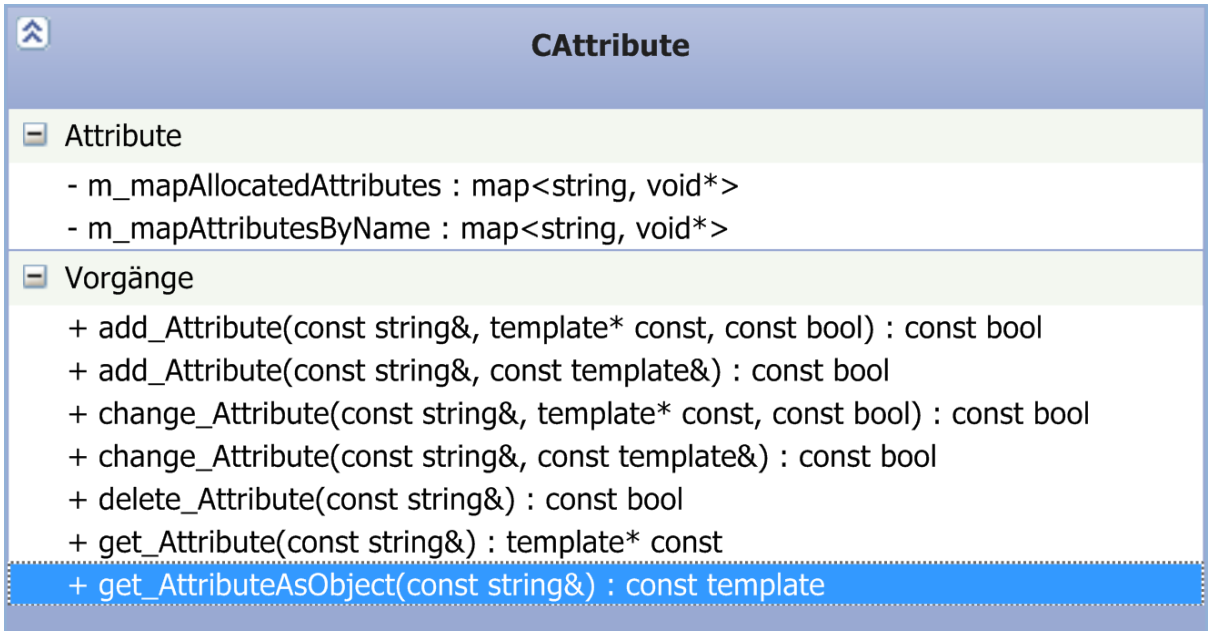


Abbildung 15 - Klassendiagramm: CAttribute

Die Klasse „CAttribute" kann jeden, aufgrund ihrer Struktur, Datentyp speichern. Sie sollte aber mit Bedacht eingesetzt werden, da diese Klasse noch überarbeitet werden muss. Jeder dieser Methoden ist eine Template[45]-Methode. Ist es der Wunsch eine Information zu speichern ruft man die Methode „add_Attribute" auf und übergibt ihr den „Namen" der Information, ebenso der Datentyp muss angegeben werden. Über den Namen der Information, kann der Wert dieser wieder aufgerufen („get_Attribute"), geändert („change_Attribute") oder gelöscht werden („delete_Attribute"). Dadurch dass jeder Spielzustand Zugriff auf seinen Verwalter hat („CGameStateManager") und die Instanz der Klasse „CAttribute" öffentlich zugänglich ist, kann jeder Spielzustand Informationen speichern oder verändern.

Das System verfügt, wie weiter oben erwähnt, noch über eine Ressourcenverwaltung für die Spielzustände. Im nächsten Abschnitt geht es um die Klasse „CFilemanagement".

[45] Aus dem Englischen „Schablone" – Möglichkeit Programmiercode generisch zu gestalten

6.4.3. CFilemanagement

Die Klasse „CFilemanagement“ kümmert sich um das Laden der Ressourcen, bevor die Spielzustände initialisiert werden. Die Registrierung der Ressourcen, welche geladen werden sollen, wird durch ein Script vollführt. Mit Hilfe der des Ressourcen-Managers werden diese registrierten Ressourcen geladen. Das Laden findet in einem zusätzlichen Thread[46] statt, um das Fenster weiterhin auf Interaktionen des Benutzers reagieren lassen zu können. Das Laden der Ressourcen wird nur einmal beim Start des Programms vollführt und die Freigabe der Ressourcen findet auch nur beim beenden des Programms statt. Die Klasse „CFilemanagement“ ist eine recht komplexe Klasse, wie im folgenden Klassendiagramm zu sehen ist:

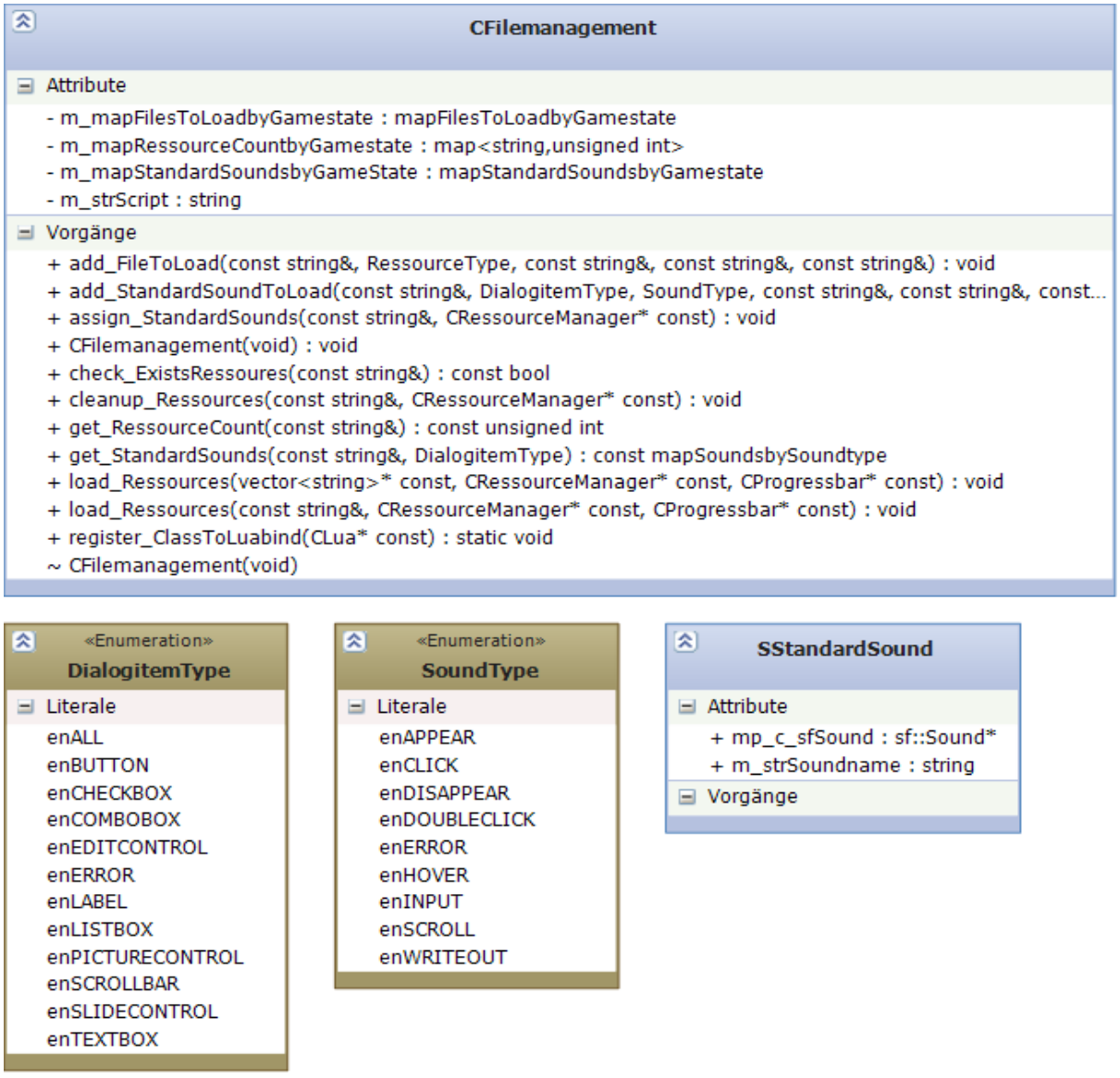


Abbildung 16 - Klassendiagramm: CFilemanagement

[46] Ein quasi-parallel laufender Prozess

Der Datei-Manager besteht aus vielen STL-Containern zum Verwalten der zu ladenden Ressourcen:

- „m_mapRessourceCountbyGamestate“
  - Diese „Map“ beherbergt die Anzahl der Ressourcen, welche geladen werden müssen für jeden Spielzustand
  - Es werden in diese „Map“ nur Spielzustände eingetragen, welche mindestens eine Ressource benötigen

- „m_mapFilesToLoadbyGamestate“
  - In dieser „Map“ stehen zu jeden eingetragenen Spielzustand die Namen und der Typ der Ressourcen, die geladen werden müssen

- „m_mapStandardSoundsbyGamestate“
  - Diese „Map“ dient ausschließlich dem Dialog-System
  - Sie speichert Audio-Dateien, welche geladen werden müssen
  - Zusätzlich zu jeder Audio-Datei wird der Typ des Steuerelements gespeichert, für diesen Typ dient diese Audio-Datei („DialogitemType“)
  - Es wird ebenfalls der Zweck dieser Audio-Datei gespeichert („SoundType“)
  - Diese Audio-Dateien gelten demnach als Standard Audio-Dateien für ihr jeweiligen Zweck und Steuerelementtyp

Bevor das Script genauer betrachtet wird, werden die zum Verständnis benötigten Methoden erklärt:

- „void add_FileToLoad(const string&, const RessourceType, const string&, const string&)“
  - Diese Methode fügt eine Ressource zum Laden relativ zu einem Spielzustand hinzu
  - Erster Parameter: Der Name des Spielzustandes
  - Zweiter Parameter: Der Typ der Ressource
    - Siehe „Enumeration“ in der Abbildung 3
  - Dritter Parameter: Der Name der Ressource
  - Vierter Parameter: Der Ordnername in der sich die Datei befindet
    - Siehe Stichwort „Ordner-Schlüssel“ im Kapitel 5.2.

- „void add_StandardSoundToLoad(const string&, DialogitemType, SoundType, const string&, const string&, const bool)“
  - Diese Methode fügt eine Audio-Datei für einen Zweck und einem Steuerelement hinzu
  - Erster Parameter: Der Name des Spielzustandes
  - Zweiter Parameter: Steuerelementtyp
  - Dritter Parameter: Der Zweck der Audio-Datei
  - Vierter Parameter: Der Name der Audio-Datei
  - Fünfter Parameter: Der Ordnername in der sich die Datei befindet

- Sechster Parameter: Ob die Audio-Datei geladen oder einfach nur für einen Zweck und Steuerelement registriert werden soll

Diese zwei Methoden sind im Script der Ressourcen-Verwaltung verfügbar. Der erste Teil des Scripts wurde im Kapitel 5.2. in der Abbildung 7 gezeigt. In der folgenden Abbildung sehen wir, aufgrund ihrer Länge, einen kleinen hinteren Teil dieser Script-Datei:

```
if c_etyFilemanager ~= nil then

    --|||||||||||||||||||||||||||||||||||||||||||||| INGAME RESSOURCES

    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "Crosshair.png"                , "Cursor"            );

    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "Alpha-Centauri-System.jpg"    , "Backgrounds"       );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "Beta-Centauri-System.jpg"     , "Backgrounds"       );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "Minimap.jpg"                  , "Backgrounds"       );

    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet1.png"                  , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet2.png"                  , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet3.png"                  , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet4.png"                  , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet5.png"                  , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet6.png"                  , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet7.png"                  , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet10.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet11.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet12.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet13.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet14.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet15.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet16.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet17.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet18.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet19.png"                 , "Planets"           );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "planet20.png"                 , "Planets"           );

    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "AlienCarrier.png"             , "Spaceships"        );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "AlienCruiser.png"             , "Spaceships"        );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "AlienFighter.png"             , "Spaceships"        );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "AlienScorpion.png"            , "Spaceships"        );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "HumanCarrier.png"             , "Spaceships"        );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "HumanCruiser.png"             , "Spaceships"        );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "HumanFighter.png"             , "Spaceships"        );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "HumanPhalanx.png"             , "Spaceships"        );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "HumanSpacestation.png"        , "Spaceships"        );
    c_etyFilemanager:add_FileToLoad             ( "IngameState"          , 0     , "HumanDrone.png"               , "Spaceships"        );
```

Abbildung 17 - Zweiter Teil des Scripts der Ressourcen-Verwaltung(RessourceManagement.lua)

Die Ressourcen-Verwaltung ist leicht zu benutzen. Ist es der Wunsch eine neue Ressource hinzuzufügen, schiebt man sie in den gewünschten Ordner und gibt im Script den Pfad dazu an. Die benötigte Methode dafür ist „add_Directory“ der Klasse „CRessourceManager“. Der nächste Schritt wäre die gewünschte Ressource zum Laden zu registrieren mit Hilfe der Methode „add_FileToLoad“ der Klasse „CFilemanagement“. Sofern die Ressource zu einem registrierten Spielzustand zugewiesen wurde, wird diese auch umgehend beim Start des Programmes geladen. Neben den zwei wichtigen Methoden, welche im Script benutzt werden, gibt es noch Methoden zum Freigeben von Ressourcen, zum Laden der Ressourcen über einen Thread und eine Methode zum Überprüfen ob ein Spielzustand Ressourcen benötigt.

Das Gamestate-System alleine lässt ein Programm nicht vollwertig erscheinen um das Programm Leben einzuhauchen, werden Benutzer-Schnittstellen benötigt um mit dem Programm zu interagieren. Im nächsten Kapitel geht es um das Dialog-System.

6.5. Dialog-System (J. W.)

Das Dialogsystem ist die GUI (Grafische Benutzer Oberfläche) von unserem Spiel. Sie verwaltet die angezeigten Steuerelemente, gewährt die Interaktion mit dem Benutzer und die damit verbundene Funktionalität aller Menüs. Es ist der Hauptteil meiner Arbeit an diesem Projekt, da es sehr umfangreich ist und auch die meiste Zeit in Anspruch genommen hat. Des Weiteren ist die GUI unabhängig vom Spiel, zusammen mit der Engine kann man jede beliebige Anwendung einfach programmieren. Wenn ich mehr Zeit gehabt hätte, dann hätte ich diese GUI mit Visual Studios verbunden, um es ähnlich wie MFC zu nutzen. Ich denke, dass es bei einfachen Anwendungen einfacher, schneller und übersichtlicher ist mit meiner GUI zu programmieren.

Das Dialogsystem ist eine Mischung aus MFC-Aspekten und dem Dialogsystem aus Starcraft2. Der Grundaufbau des Systems ist stark an das von Starcraft2 angelehnt. Die Dialogitems (Steuerelemente) sind von ihrem Aufbau und Funktionalität an den Steuerelementen von MFC orientiert.

Mein Einstieg in die Programmierung hatte ich durch den Editor von Warcraft3 (2008) und war dort in der Mapping-Szene aktiv. Als im Jahre 2010 Starcraft2 erschien, stieg ich auf den weitaus besseren Editor um. Für eine Map programmierte ich die Verwaltung des Dialogsystems in eine an C++ angelehnte Klassenstruktur um. Als ich später dann mit Julien die Lernleistung zusammen anfing, wollte ich definitiv die Benutzeroberfläche programmieren. Da ich durch die vorangegangene Programmierung schon etwas Erfahrung hatte und mir das Dialogsystem von Starcraft2 gefiel, beschloss ich unsere GUI ähnlich meines vorher programmierten Systems aufzubauen.

Anfangs war das Dialogsystem etwas sehr umständlich über Strukturen und void-Zeiger aufgebaut, was an teilweise fehlenden Kenntnissen über Polymorphie lag. Das System wurde aber von mir mehrfach stark überarbeitet und verbessert.
Der Grundaufbau des Dialogsystems ist folgender: Jeder DialogGameState besitzt einen Dialogmanager. Der Dialogmanager ist die Schnittstelle zwischen dem zugehörigen DialogGameState-System und dem Dialogsystem.
Der Dialogmanager besitzt eine Liste von Dialogen. Dialoge sind Fenster die mehrere Dialogitems besitzen und verwalten. Erst die Dialogitems an sich sind die einzelnen Steuerelemente, mit denen der Benutzer interagiert.
Die Verwaltung läuft hierarchisch ab. Der Dialogmanager gibt die Befehle, z.B. das Updaten des Dialogsystems an die Dialoge weiter, welche ihre eigenen Update-Funktion aufrufen und in dieser wieder die Update-Funktionen der einzelnen Dialogitems aufrufen.

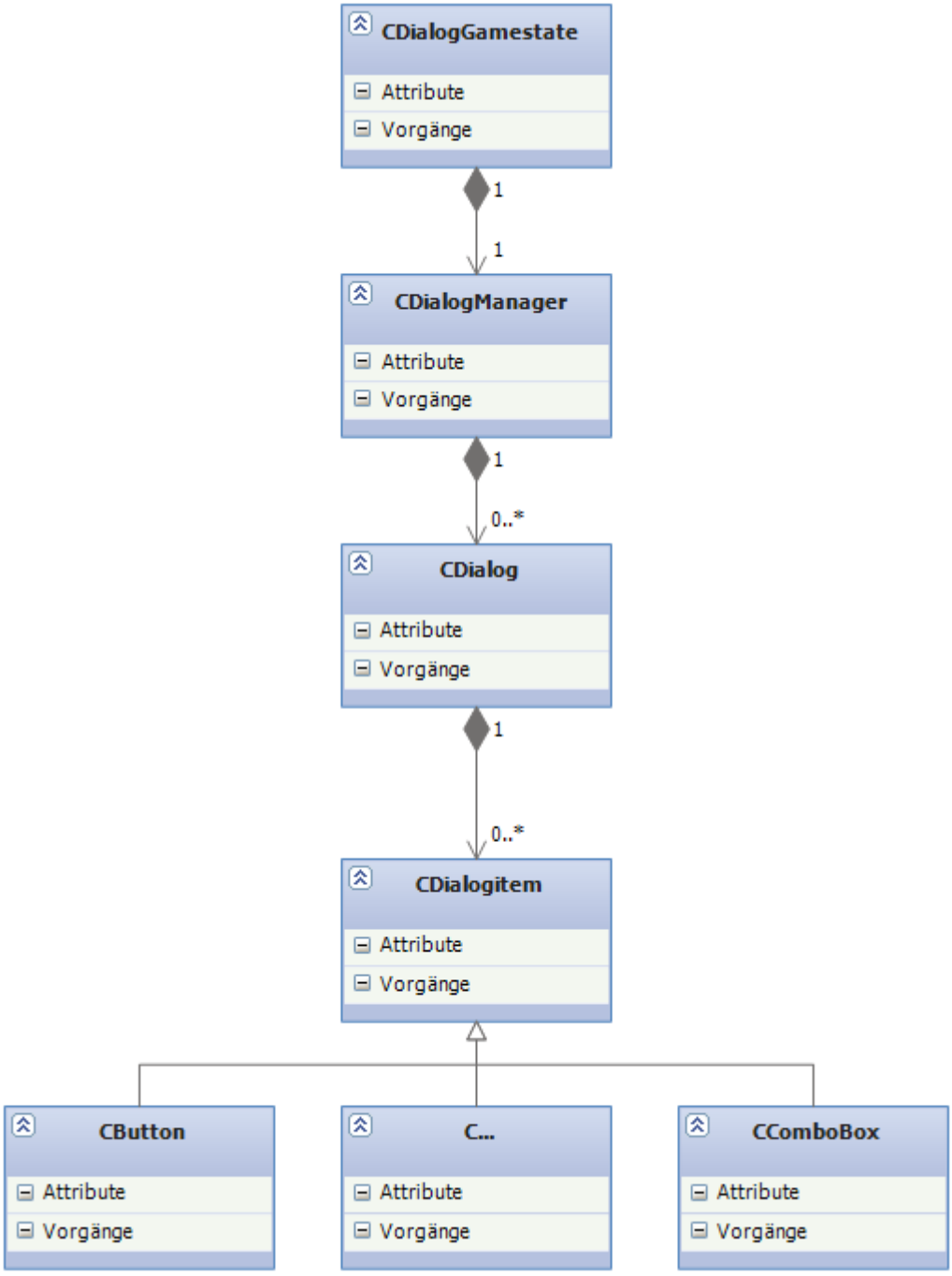


Abbildung 18 - Übersicht Dialogsystem

Es gibt noch mehr Dialogitems als die abgebildet, was durch „C…“ verdeutlicht werden sollte. CDialogGamestate gehört nicht direkt zum Dialogsystem, doch es soll an dieser Stelle die hierarische Struktur verdeutlichen.

6.5.1. Dialogmanager

Der DialogManager ist der Dreh- und Angelpunkt der Verwaltung und Funktionalität des Dialogsystems. Seine Aufgaben sind: Updaten und Aufrufen der draw-Funktionen der Dialoge/Dialogitems, das Weitergeben von Änderungen, wie z.B. der Auflösung oder der Sprache. Des Weiteren ist er für das Verwalten der Events zuständig, des Focus, sowie die Modalität von Dialogen. Zu diesen Aspekten in den Funktionen mehr.

Der Dialogmanager besitzt eine Vielzahl von Funktionen, doch ich möchte in diesem Abschnitt nur auf die wichtigsten eingehen. Der Rest sind hauptsächlich Get- und Set-Methoden. Eine Übersicht befindet sich im Anhang (11.1.1) bzw. können im Quellcode des Projektes direkt eingesehen werden.

```
bool        add_Dialog                ( CDialog* const p_c_etyDialogNewDialog );

void        change_FocusedDialogitem  ( ety::CDialogitem* const p_c_etyDialogitemNewFocusedDialogitem );
void        change_ModalDialog        ( bool bDialogModal, CDialog* const p_c_etyDialogModalDialog,
                                        sf::Vector3f c_sfVector3fFactors );
bool        check_IsDialogitemActive  ( CDialogitem* const p_c_etyDialogitem );
void        cleanup_Dialogs           ( void );

const bool  delete_Dialog             ( ety::CDialog* const p_c_etyDialog );
void        draw_Dialogs              ( sf::RenderTexture* const p_c_sfRenderTextureGameScene );

void        handle_Events             ( sf::Event& c_sfEvent );

void        reinit_Dialogs            ( const sf::VideoMode& c_sfNewVideoMode );

void        update_Dialogs            ( const sf::Uint32& uiFrameTime );
void        update_Language           ( const std::string strNewLanguage, CLua* const mp_c_etyLuaScript );
```

Abbildung 19 - Wichtige Funktionen des Dialogmanagers

- bool add_Dialog (CDialog* const p_c_etyDialogNewDialog)
  - Fügt einen neu erstellten Dialog dem Dialogmanager hinzu
  - Ordnet ihn in die richtige Position der Liste ein
  - Dialog werde direkt erstellt und nicht wie Dialogitems über Funktionen

Die Dialoge sind in der Liste nach ihrer DrawPosition[47] sortiert. Alle Dialoge und Dialogitems haben eine Anordnung, in welcher Reihenfolge diese gezeichnet werden sollen, damit z.B. Hintergrundbilder nicht über wichtigen Steuerelementen liegen oder Dialogitems im Hintergrund eine aufgeklappte ComboBox überdecken. Für die Reihenfolge des Zeichnens habe ich das Affinity[48]-System entworfen, welches aus zwei Variablen besteht. Zum einen die Affinity als Enum und zum anderen die DrawPosition als unsigned int. Die Affinity ist in drei Bereiche unterteilt, enBACKGROUND, en STANDARD und enFOREGROUND. Die Dialoge und Dialogitems werden zuerst nach diesem Enum sortiert, sodass Dialoge/Dialogitems, die enFOREGROUND als Affinity haben, immer über anderen mit der Affinity enSTANDARD oder enBACKGROUND gezeichnet werden. Nach dieser Unterteilung bestimmt die DrawPosition die Anordnung innerhalb der Affinity. Da die Liste beim Zeichnen von vorne nach hinten durchgegangen wird, stehen die Dialoge/Dialogitems mit der Affinity enBACKGROUND ganz oben in der Liste, aufsteigend sortiert nach ihrer DrawPosition.

[47] Bedeutet Zeichenposition, in diesem Fall benutze ich es als Eigenname
[48] Ist Englisch und bedeutet Affinität oder Neigung

Danach folgen die mit der Affinity enSTANDARD, welche auch aufsteigend sortiert sind. Am Ende der Liste stehen die mit der Affinity enFOREGROUND, welche aber nach ihrer DrawPosition absteigend geordnet sind.
Damit ist gewährleistet, dass Dialog/Dialogitems mit der Affinity enBACKGROUND und der DrawPosition = 0 immer sehr weit im Hintergrund liegen und die mit der Affinity enFOREGROUND und der DrawPosition = 0 möglichst weit im Vordergrund liegen.

- void change_FocusedDialogitem (CDialogitem* const p_c_etyDialogitemNewFocusedDialogitem)
    - Setzt den Focus auf ein neues oder gar kein Dialogitem
    - Der Focus ist das letzte angeklickte Dialogitem
    - Der Focus ist dialogübergreifend, dies bedeutet, dass nur maximal ein Dialogitem im ganzen State im Focus liegen kann

- void change_ModalDialog (bool bDialogModal, CDialog* const p_c_etyDialogModalDialog)
    - Setzt einen Dialog modal
    - Es kann immer nur ein Dialog im State modal sein
    - Wenn ein Dialog modal ist, kann nur mit diesem interagiert werden

- bool check_IsDialogitemActive (CDialogitem* const p_c_etyDialogitem)
    - Überprüft ob ein Dialogitem momentan auf Events reagieren kann, Dies ist nur für Modalität von Dialog wichtig
    - Es gibt zwei Variablen die angeben, ob das Dialogitems auf Events reagieren kann
    - Diese sind m_bInternActive und m_bActive
    - Die erste gibt an, ob das Dialogitem allgemein auf Events reagieren kann, die zweite gibt an, ob das Dialogitem in diesem Moment auf Events reagieren kann
    - Sollte ein Dialog modal sein, dann können zwar Dialogitems m_bInternActive gleich true sein, aber wenn sie nicht zu dem modalen Dialog gehören, können sie auch nicht auf Events reagieren und sind damit m_bActive = false

- void draw_Dialogs (sf::RenderTexture* const p_c_sfRenderTextureGameScene)
    - Die Funktion ruft die draw-Funktionen der Dialoge auf
    - Welche den Dialog und seine Dialogitems zeichnen

- void handle_Events (sf::Event& c_sfEvent)
    - Diese Funktion verwaltet eingehende Events und gibt sie an die entsprechenden Dialogitems weiter (Näheres unter 6.5.4. Event-System)

- void reinit_Dialogs (const sf::VideoMode& c_sfNewVideoMode)

  - Diese Funktion passt die Dialoge auf die neue Auflösung an
  - Dazu wird die adjust_SizeToResolution-Funktion der Dialoge aufgerufen
  - Diese wiederrum rufen die gleiche Funktion ihrer Dialogitems auf
  - Zur Anpassung werden Position und Größe der Dialoge und Dialogitems neu berechnet
  - Beim Erstellen werden alle Größen- und Positionsangaben so gewählt wie sie bei einer Auflösung von 1680x1050 (16:10) sind
  - Erst im Konstruktor der Dialoge und Dialogitems wird dieser Wert an die aktuelle Auflösung umgerechnet

- void update_Dialogs (const sf::Uint32& uiFrameTime)

  - Aktualisiert die Dialoge über deren update-Funktion
  - Diese rufen die update-Funktionen ihrer Dialogitems auf
  - Die update-Funktion der Dialoge gibt gegeben falls einen Zeiger auf ein Dialogitem zurück, wenn der Cursor so über diesem liegt, dass es von ihm angesprochen werden könnte und auf Events reagieren könnte (Näheres unter 6.5.4. Event-System)

- void update_Language (const std::string strNewLanguage, CLua* const mp_c_etyLuaScript)

  - Aktualisiert die Sprache die Texte und Beschriftungen der Dialogitems
  - Dazu ruft diese Funktion die update_Language-Funktion der Dialoge auf, welche wiederrum die gleiche Funktion der Dialogitems aufrufen

6.5.2. Dialog

Ein Dialog ist ein Fenster, aber nicht zwangsläufig in dem Sinne wie es unter MFC-Anwendungen oder in Windows bekannt ist, es kann aber diese Funktionalität erfüllen. Es kann auch als Menü-Hintergrund und als Verwaltungsbereich für Dialogitems eingesetzt werden.
Jeder Dialog besitzt und verwaltet mehrere Dialogitems, welche immer zu genau einem Dialog gehören und ihre Zugehörigkeit nicht verändern können. Des Weiteren reagieren nur Dialogitems innerhalb des Dialogs auf Events, könnten aber auch außerhalb dieses gezeichnet werden.

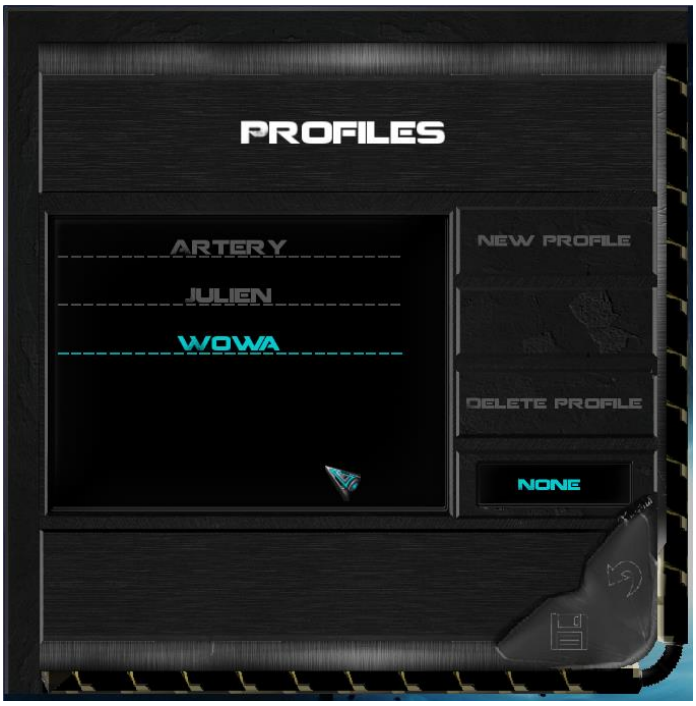


Abbildung 20 - Beispiel Dialog

Ein typischer Dialog mit verschiedenen Dialogitems, z.B. Buttons, TextButtons, einer ListBox und zwei Labeln.

Abbildung 21 - Leerer Dialog

Der leere Dialog im Vergleich dazu.

Dialog und Dialogitems besitzen eine CustomID, welche einzigartig ist und über die die Dialoge und Dialogitems eindeutig identifizierbar sind. Die Benennung der CustomIDs wird beim Erstellen der Objekte festgelegt. Hierbei wählt der Programmierer diese, was bedeutet, dass es keine Verwaltung und Überprüfung der IDs gibt z.B. ob eine solche ID bereits besteht. Dies wäre sinnlos und ein unnötiger Aufwand, weswegen der Programmierer aufpassen muss wie die CustomIDs gewählt werden. Da dies aber immer nach einem gewissen Schema passiert, sollte es zu keinen Komplikationen kommen.
Bei Dialogen wird die CustomID zuerst mit dem Statenamen begonnen, dann kommt die Funktionalität und zum Schluss wird sie ggf. noch mit „Menu“ versehen. Z.B. der Dialog der aufgerufen wird, wenn man das Spiel im Hauptmenü verlassen will, heißt „MainMenuExitMenu“.

Die Dialoge sind wie die Dialogitems mit dem Affinity-System angeordnet. Darüber hinaus nutzen sie, genau wie die Dialogitems, das Anchor-System. Dieses System ermöglicht es, die Dialoge an bestimmte Ankerpunkte zu hängen und sie erst dann durch Positionsangaben zu verschieben. Das System besteht aus zwei Variablen: Zum einen dem Anchor als Enum und zum anderen einer Bool-Variable „m_bAnchorCentered“. Das Enum gibt den Ankerpunkt direkt an, z.B. enMID oder enTOPLEFT. Als Positionsursprung für Dialoge wird immer die linke, obere Ecke des Bildschirms gewählt, bei Dialogitems ist es stattdessen die linke, obere Ecke des zugehörigen Dialogs. Die Bool-Variable gibt dann zusätzlich an, ob der Dialog/Dialogitem auf dem Ankerpunkt zentriert werden soll. Beispielsweise hat ein Dialog den Anchor enMID, der Ankerpunkt soll zentriert sein und die Position ist (0,0), dann liegt dieser Dialog genau in der Mitte des Bildschirms und zwar zentriert.
Ein Dialog kann auch modal sein dies bedeutet, dass nur dieser Dialog auf Events reagieren kann, dies wird durch den Dialogmanager verwaltet. Des Weiteren verwaltet der Dialog seine Dialogitems in einer Liste, genau wie der Dialogmanager seine Dialoge, und gibt die Befehle des Dialogmanagers an seine Dialogitems weiter.

Wichtige Funktionen:

- bool check_MouseCoordinates (void)
    - Diese Funktion gibt an, ob sich der Cursor über bzw. in dem Dialog befindet
    - Eine solche Funktion findet sich auch in allen Dialogitems
- bool add_ChildrenDialogitem (CDialogitem* const p_c_etyDialogitemChildren)
    - Fügt dem Dialog Dialogitems hinzu, die nicht direkt über den Dialog erstellt wurden
    - Dies betrifft ChildrenDialogitems, welche ein fester Bestandteil eines anderen Dialogitems sind, z.B. die Scrollbar einer ComboBox
    - Damit aber die Scrollbar auf Events reagieren kann, muss sie dem Dialog hinzugefügt werden

- void modificate_ColorFilter (sf::Vector3f c_sfVector3fFactors)
    - Verändert die Farbeigenschaften des Dialog und seiner Dialogitems um gewisse Faktoren
    - Durch diese Funktion kann man z.B. Dialoge ganz einfach aufhellen oder verdunkeln

- template <typename typeDialogitemType> typeDialogitemType get_DialogitemByCustomID (const std::string& strCustomID)
    - Dieses Template einen Zeiger auf Dialogitem über seine CustomID zurück
    - Das Besondere ist, dass dieses Template den Zeiger in dem gewünschten Datentyp konvertiert
    - Dadurch kann mit einer einzigen Funktion jedes beliebe Dialogitem angefordert werden
    - Bsp.: CComboBox* p_c_etyComboBoxTmp = p_c_etyDialogGraphicOptions-> get_DialogitemByCustomID<CComboBox*>(„GraphicOptionsResolutionComboBox„);

```
template                <typename typeDialogitemType>
typeDialogitemType      get_DialogitemByCustomID        ( const std::string& strCustomID)
{
    for(std::list<CDialogitem*>::iterator itDialogitem = m_listDialogitems.begin() ;
        itDialogitem != m_listDialogitems.end(); itDialogitem++)
    {
        //Es wird die CustomID des Dialogitems überprüft und wenn sie überein stimmt wird das
        //Dialogitem zurückgegeben
        if((*itDialogitem)->get_CustomID() == strCustomID)
        {
            typeDialogitemType typeDialogitemTypeRequestedDialogitem =
                reinterpret_cast<typeDialogitemType>((*itDialogitem));

            return typeDialogitemTypeRequestedDialogitem;
        }
    }
    return NULL;
}
```

Abbildung 22 - get_DialogitemByCustomID Funktions-Template

Ein Dialog wird erst über einen Zeiger angelegt und dann in dessen Konstruktor dem Dialogmanager des DialogGameState hinzugefügt.

```
ety::CDialog* p_c_etyDialogBackground = new CDialog(
    "MainMenuBackground", p_c_etyCurrentRessourceManager->get_Texture( "MainMenuBackground.jpg" ),
    1680.f, 1050.f, sf::Vector2f( 0, 0 ), sf::IntRect(sf::Vector2i( 0, 0 ), sf::Vector2i( 1280, 1024 )),
    mp_c_etyDialogManager, c_sfCurrentVideomode
    );

p_c_etyDialogBackground->set_Affinity(Affinity::enBACKGROUND);
```

Abbildung 23 - Anlegen eines neuen Dialogs

Die Dialogitems eines Dialogs werden direkt über den Dialog erstellt, in dem die passende create-Funktion des Dialogs aufgerufen wird. In dieser Funktion wird ein neues Objekt des jeweiligen Dialogitems angelegt und mit den übergebenen Parametern ausgestattet und letztlich richtig in die Dialogitemliste eingefügt.

```
p_c_etyDialogExitMenu->createDialogitem_Button(
    "ExitYesButton", p_c_etyCurrentRessourceManager->get_Texture( "transparentBackground.png" ),
    137.f, 74.f, sf::Vector2f(161.f, -205.f), sf::IntRect(sf::Vector2i(0,0), sf::Vector2i(0,0))
    );
```

Abbildung 24 - Anlegen eines neuen Buttons

```
void    ety::CDialog::createDialogitem_Button   (
                                                   std::string strCustomID, const sf::Texture& c_sfTextureButtonSet,
                                                   float fWidth, float fHeight, sf::Vector2f c_sfVector2fPosition,
                                                   sf::IntRect c_sfIntRectButtonSubRect
                                                 )
{
    //Ein neuer Button wird angelegt
    CButton* p_c_etyNewButton                   = NULL;
    p_c_etyNewButton                            = new CButton (strCustomID, c_sfTextureButtonSet, fWidth, fHeight,
                                                               c_sfVector2fPosition, c_sfIntRectButtonSubRect, NULL, this);

    //Und dem Dialog hinzugefügt
    add_Dialogitem(p_c_etyNewButton, true);
}
```

Abbildung 25 - Funktion zum Erstellen eines neuen Buttons

Die restlichen Funktionen sind hauptsächlich Get- und Set-Methoden, die hier keine wichtigere Rolle spielen, aber bei Bedarf können sie direkt im Code angesehen werden. Eine Übersicht befindet sich auch im Anhang (11.1.2).

6.5.3. Dialogitem

Ein Dialogitem ist die Basisklasse eines Steuerelements und kann über Polymorphie ein beliebiges Dialogitem darstellen. Die Basisklasse CDialogitem besitzt alle wichtigen Grundattributen und Funktionen, welche alle virtuell sind, die Dialogitem-Klasse an sich ist aber nicht rein virtuell, da oftmals einfach nur mit Zeigern auf Dialogitems gearbeitet wird. Alle Dialogitems werden von dieser Klasse public abgeleitet.
Ich habe mich dagegen entschieden spezifische Dialogitems von SFML Klassen oder anderen Dialogitems Klassen abzuleiten. Stattdessen habe immer die jeweiligen Objekte in die Klasse gesetzt. Diese Entscheidung rührt daher, dass ich mehr Übersicht erhalten wollte, denn sonst würden alle Attribute und Funktionen der Klasse mit in dieser stehen. Des Weiteren empfinde ich Ableitungen so, dass die neue Klasse ein Typ der Ableitung ist und z.B. sf::Sprites[49] oder ety::Labels nur Komponenten eines Dialogitems sind.
Das Grundgerüst des Dialogitems besteht aus Angaben über die Sichtbarkeit und Aktivität, die Positionsangaben, die Größen und einem Zeiger auf den ParentDialog. Nähere Information stehen im Klassendiagramm, sowie in den Header- und Quelldateien.

[49] Klasse der SFML: Sie speichert und stellt einen Teil einer Textur als Bild dar

Es gibt eine Vielzahl von Dialogitems, die aber von ihrer Funktionalität her genauso oder ähnlich wie die Standard-Windows-Steuerelemente funktionieren.
Dialogitems können Sounds bei bestimmten Events abspielen. Dafür besitzen sie eine Map in denen Zeiger auf die Sounds und ein Enum für das jeweilige Event gespeichert werden. Es besteht die Möglichkeit, die Dialogitems mit StandardSounds zu belegen. In der RessourcenManagment.lua Datei, über die die Sounds geladen werden, kann man pro State einstellen, welche Dialogitems welche StandardSounds besitzen sollen. Diese können dann über die passende Variable im Konstruktor des Dialogitems hinzugefügt werden.
Eine weitere Möglichkeit die die Dialogitems bieten ist, dass sich der Ankerpunkt des Dialogitems nicht, wie standardmäßig, sich auf den Dialog bezieht, sondern auf ein anderes Dialogitem. Somit ist es möglich leicht Dialogitems zu formatieren und auszurichten.
Allein, wenn man sich die Anzahl der Steuerelemente in MFC ansieht ist diese sehr groß. Natürlich macht es nicht unbedingt Sinn, alle Steuerelemente von Windows nach zu programmieren und abzukupfern. Für unser Projekt habe ich erst mal die wichtigsten Steuerelemente programmiert und diese auch nicht eins zu eins von MFC oder Starcraft2 übernommen. Natürlich sind sie von der Funktionalität ähnlich, was auch logisch ist, und teilweise an den Aufbau angelehnt.
Fast alle Dialogitems sind bildbasiert, dies heißt, dass sie über ein Bild, ein Sprite gezeichnet und dargestellt werden, wie z.B. ein Button. Die Möglichkeit zum Zeichnen läuft über SFML, denn jedes Dialogitem besitzt mindestens ein Objekt von sf::Drawable. Meistens sind dies sf::Sprite, also Bilder. Es können aber auch sf::Text[50] Objekte sein. Diese Objekte liegen meist nicht direkt in den Klassen, sondern werden nochmal durch ein Dialogitem verwaltet, wobei es zwei Ausnahmen gibt.
Wenn ein Dialogitem durch ein Bild dargestellt wird, besitzt es immer ein Objekt von CPictureControl Dies ist eine Bild-Klasse und verwaltet ein Objekt von sf::Sprite. Doch Button und Checkbox tragen keine PictureControls in sich, sondern jeweils ein Sprite. Was verschiedene Gründe hat (Siehe Checkbox).
Wenn ein Dialogitem Schrift besitzen soll, trägt es meist ein Objekt von CLabel in sich, welches sf::Text und somit die Schrift eines Dialogitems verwaltet.
Es gibt bisher folgende Dialogitems: CButton, CCheckbox, CComboBox, CEditControl, CLabel, CListBox, CScrollbar und CSlideControl.
Dies sind die wichtigsten Dialogitems für unser Projekt und mit diesen können wir auch so gut wie alles Wichtige realisieren. Hauptsächlich aus Zeitgründen wurde erst einmal auf gewisse Steuerelemente verzichtet. Diese könnten aber jederzeit einfach programmiert und in das Dialogsystem eingebettet werden. Dialogitems die geplant waren aber noch nicht umgesetzt wurden, sind „RadioButtonBox“ und „TextBox“. Das erste soll eine Gruppierungsbox für RadioButtons bilden, welche an sich aus normalen CheckBoxes bestehen. Bisher gab es noch keine Verwendung für RadioButtons und bei Bedarf kann diese Funktionalität auch noch über Events realisiert werden, ohne das Dialogitem zu erstellen.
Die TextBox ist eine Box, die mit formatiertem Text gefüllt sein soll. Des Weiteren soll sie um eine Scrollbar verfügen, mit der sich der Text hoch und runterscrollen lässt. Dieses Dialogitem ist zwar ziemlich wichtig, doch aus Zeitgründen muss erst einmal darauf verzichtet werden. Da dieses Dialogitem erst bei z.B. Missionsbeschreibungen

[50] Klasse der SFML: Stellt Text dar, welcher über diese Klasse auf modifizierbar ist

etc. Verwendung finden würde. Dies fällt eher in den Content-Teil, daher hat es nicht so hohe Priorität wie andere Aspekte des Projekts.
Eine gesamt Übersicht über die Funktionen und Attribute befindet sich im Anhang (11.1.3). An dieser Stelle möchte ich nur auf die wichtigsten Funktionen eingehen.

```
virtual void    calculate_Positions         (bool bRealPosition);
/* ... */
virtual void    add_EventSound              (enum SoundType::en_etySoundType en_etySoundType, 
virtual void    adjust_SizeToResolution     ( const sf::VideoMode& c_sfVideoModeNewSettings, c

/* ... */
virtual bool    check_MouseCoordinates      (void);
/* ... */
virtual bool    draw_Dialogitem             (sf::RenderTexture* p_c_sfRenderTextureScene);
/* ... */
virtual void    handle_InternEvents         (const sf::Event& c_sfEvent);
/* ... */
virtual void    modificate_Colorfilter      (sf::Vector3f c_sfVector3fFactors);
/* ... */
virtual bool    remove_EventSound           (enum SoundType::en_etySoundType en_etySoundType);
/* ... */
virtual void    reset_FocusedDialogitem     (void);
/* ... */
virtual void    update_Dialogitem           (const sf::Uint32& uiFrameTime);
```

Abbildung 26 - Wichtige Funktionen des Dialogitems

- void calculate_Positions (bool bRealPosition)
    - Die beim Erstellen eines Dialogitems angegebene Position wird mit dieser Funktion auf die aktuelle Auflösung angepasst
- void adjust_SizeToResolution (const sf::VideoMode& c_sfVideoModeNewSettings, const sf::VideoMode& c_sfVideoModeOldSettings)
    - Diese Funktion passt die Größe und Position des Dialogitems an die aktuelle Auflösung an
    - Diese Funktion muss in jedem abgeleiteten Dialogitem spezifiziert werden
- bool check_MouseCoordinates (void)
    - Diese Funktion überprüft, ob der Cursor auf bzw. in dem Dialogitem liegt
    - Standardmäßig wird das Dialogitem als Rechteck gesehen
    - Bei Bedarf kann diese Funktion in anderen abgeleiteten Dialogitems spezifiziert werden
- bool draw_Dialogitem (sf::RenderTexture* p_c_sfRenderTextureScene)
    - Diese Funktion zeichnet das Dialogitem
    - Sie muss in jeder abgeleiteten Klasse spezifiziert werden

- void handle_InternEvents (const sf::Event& c_sfEvent)
    - Diese Funktion verwaltet Events die zur Funktionalität des Dialogitems benötigt werden
    - Z.B. dass ein Button bei einem MouseHoverEvent sein HoverSprite anzeigt

- void modificate_ColorFilter (sf::Vector3f c_sfVector3fFactors)
    - Funktioniert genau wie beim Dialog
    - Verändert den Farbfilter eines Dialogitems um gewisse Faktoren

6.5.3.1. Button

Der Button ist eine Schaltfläche und nimmt die Aufgabe eines konventionellen Buttons oder „Drückers“ ein. Dies bedeutet konkret, dass er aus einem Bild, dem Background, welches seine Schaltfläche ist, besteht.
Wenn man mit dem Mauszeiger über den Button fährt, erscheint, so fern gewünscht und vorhanden, das Hover-Bild des Buttons. Dies signalisiert, dass der Button im Fokus der Maus liegt.
Das HoverSprite ist standardmäßig genauso groß wie der Background, doch es ist möglich, dieses größer anzeigen zu lassen. Was z.B. dann Sinn macht, wenn das HoverSprite ein Leuchten oder Schein besitzt.

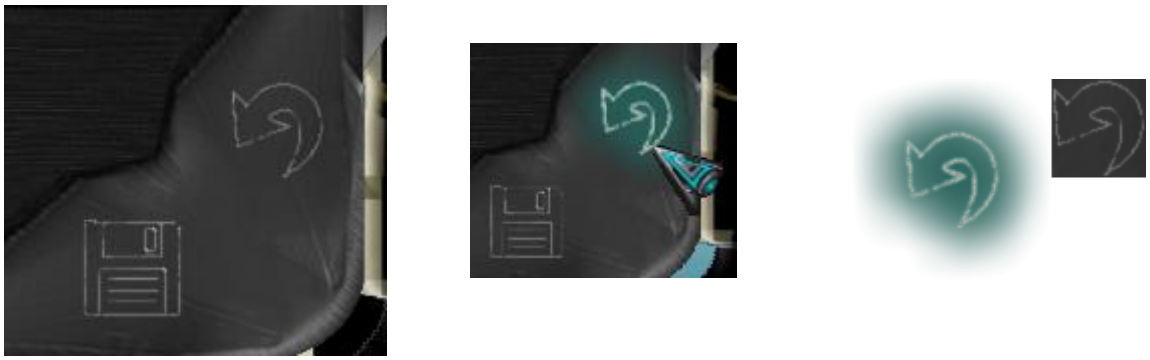


Abbildung 27 - Beispielbilder für einen Button

Buttons können auf verschiedene Arten angeklickt werden und können auf verschiedene Events reagieren. Z.B. wie eben schon angesprochen auf Hover-Events, dass sich das Aussehen des Buttons ändert, wenn man mit dem Mauszeiger drüber fährt. Aber auch durch Anklicken oder durch Doppelklicks.
Des Weiteren besitzt der Button noch eine Caption, also eine Beschriftung. Dies ist aber nicht explizit erforderlich, sondern optional. Die Caption besteht aus einem weiteren Dialogitem, dem Label (in Abschnitt Label genauer beschrieben).
Ein Button muss aber nicht zwangsläufig eine Schaltfläche besitzen, es ist auch möglich einen Textbutton zu erstellen. Der Textbutton besteht nur aus einer Caption, welche angeklickt werden kann und dann die Schaltfläche ersetzt.

Abbildung 28 - Beispiele für verschiedene Buttons

Der Textbutton reagiert nur dann, wenn der Text im Focus der Maus liegt. Das Anlegen eines Text- oder normalen Buttons läuft über zwei unterschiedliche Funktionen, welche unterschiedliche Konstruktoren aufrufen. Zum einen über die „create_Button"-Funktion für einen standardmäßigen und „create_TextButton" für einen Textbutton.
Die Caption kann auch bei einem Hover-Event verändert werden, in dem sie „gehighlighted" (hervorgehoben) wird und dadurch ihre Farbe wechselt, noch dazu ist es möglich die Schrift bei einem solchen Event zu vergrößern.
Buttons sind der Hauptbestandteil jeder GUI und wir setzen sie auch oft in unserem Projekt ein. Ich denke, dass die Umsetzung dieses Steuerelements ist mir gut gelungen.

Hier noch ein paar der wichtigsten Funktion mit Erklärungen:

```
        void    change_SpriteStyle          (bool bHoverSprite);
        void    set_CaptionHighlighted      (bool bCaptionHighlighted);

virtual void    adjust_SizeToResolution     ( const sf::VideoMode& c_sfVideoModeNewSettings,
                                              const sf::VideoMode& c_sfVideoModeOldSettings );

virtual bool    check_MouseCoordinates      (void);

        bool    create_CaptionLabel         (
                                                ety::Color c_etyColorCaptionColor_highlighted,
                                                std::string strContent, float fCharactersize,
                                                ety::Color c_etyColorTextcolor
                                            );

        bool    delete_CaptionLabel         (void);
virtual bool    draw_Dialogitem             (sf::RenderTexture* p_c_sfRenderTextureScene);
virtual void    handle_InternEvents         (const sf::Event& c_sfEvent);
virtual void    modificate_Colorfilter      (sf::Vector3f c_sfVector3fFactors);

        void    swap_HighlightInformation   (void);
```

Abbildung 29 - Wichtige Funktionen des Buttons

Eine gesamt Übersicht über die Funktionen und Attribute befindet sich im Anhang (11.1.4)

- bool create_CaptionLabel ([…])
    - Erstellt eine Caption für den Button, sofern noch keine vorhanden ist
    - Standardmäßig besitzt der Button keine Caption
    - Bei TextButtons ist dies nicht nötig, da sie selbst nur aus eine Caption bestehen
- void swap_HighlightInformation (void)
    - Wechselt die Farbinformationen und Schriftgrößeninformationen der Caption, um sie „gehighlighted“ oder normal anzuzeigen
    - Im Button werden nur die Informationen für den „gehighlightedten“-Zustand gespeichert
    - Beim Aufruf dieser Funktion werden die Informationen des Labels mit denen im Button vertauscht, um alle Informationen zu erhalten
    - Diese Funktion ist private

- void change_SpriteStyle (bool bHoverSprite)
    - Passt das Sprite des Buttons so an, dass es entweder den Background oder das HoverSprite anzeigt
    - Dazu wird das SubRect [51]auf der Textur verschoben
    - Des Weiteren wird ggf. die Größe des Sprite angepasst
    - Diese Funktion ist private

[51] In einem Sprite wird ein Zeiger auf eine Textur gespeichert und das SubRect ist ein rechteckiger Bereich auf der Textur welchen das Sprite darstellt

6.5.3.2. Checkbox

Das Dialogitem Checkbox ist eines der einfachsten Dialogitems des System, aber für eine GUI und unser Projekt nicht minder wichtig. Die Checkbox ist von ihrer Funktionalität her denkbar einfach. Sie besteht nur aus einem Sprite, welches die Checkbox darstellt.
Die Checkbox kann man durch Anklicken in den Zustand check gleich true oder false setzen, was anzeigt, ob sie angewählt ist oder nicht. Die Checkbox zeigt je nach Zustand unterschiedliche Bilder an, um diesen Zustand klar unterscheiden zu können.

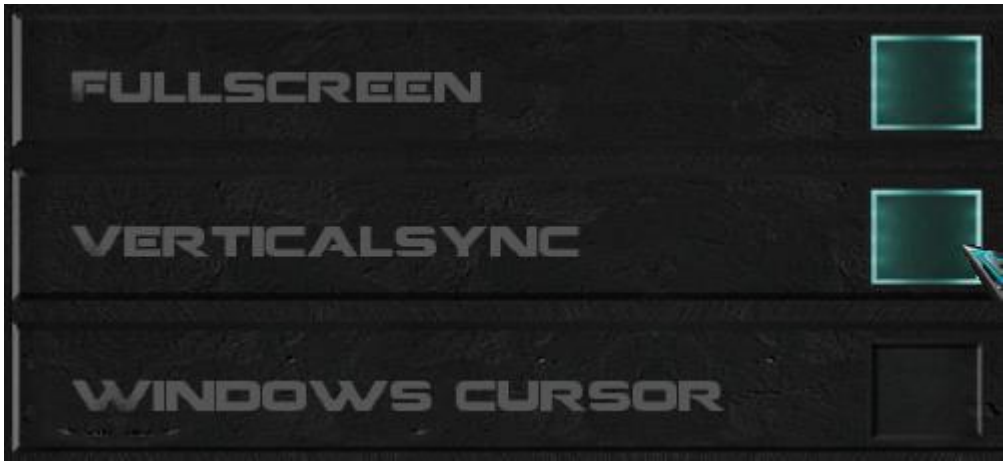


Abbildung 30 - Checkbox Beispiele


Abbildung 31 - Beispiel für eine Checkboxtextur

Die Checkbox und der Button sind die einzigen beiden Dialogitems, welche direkt ein Sprite besitzen, anstatt dies durch die Klasse CPictureControl verwalten zulassen. Dies liegt daran, dass die ersten erstellten Dialogitems der Button und die Checkbox waren, das PictureControl entstand erst später. Es wäre natürlich möglich dies zu ändern, aber ich entschied mich dagegen, da es im Endeffekt keine wirkliche Rolle spielt und die Funktionalität nicht eingeschränkt wird und es letztlich nur unnötigen Aufwand birgt.
Die Checkbox besitzt keine weiteren besonderen Funktionen, die hier aufgegriffen werden sollten, da es sich nur um vererbte Funktionen und Get- und Set-Methoden handelt.
Allein, wenn man sich die Anzahl der Attribute anguckt, die die Checkbox von sich aus besitzt, wird deutlich, dass die Checkbox ein doch sehr einfaches Dialogitem ist. Denn im Vergleich besitzt die Checkbox nur vier eigene Attribute und z.B. die ComboBox über 20 eigene Attribute.

Eine gesamt Übersicht über die Funktionen und Attribute befindet sich im Anhang (11.1.5)

6.5.3.3. ComboBox

Die Klasse ComboBox ist das umfangreichste Dialogitem des Systems und an diesem arbeitete ich die meiste Zeit, im Vergleich zu den anderen Dialogitems. Das besondere an der ComboBox ist, dass sie im Endeffekt nicht ein Dialogitem ist. Denn sie kann zwei Arten von Zuständen annehmen, einmal ist es möglich eine konventionelle ComboBox zu erstellen, also eine Box die sich aufklappen lässt, in der Einträge stehen und man einen davon auswählen kann. Die andere Möglichkeit ist es eine ListBox zu erstellen, diese zeigt die Einträge in einer Liste an, ohne sie vorher aufklappen zu müssen. Aber auch in ihr kann ein Eintrag ausgewählt werden. Ich entschied mich dazu, beide Dialogitems in dieselbe Klasse zu schreiben, da sie viel Code gemeinsam haben, nur bei Events und den Updates unterscheiden sie sich sehr. Dadurch gibt es natürlich Variablen, die in einem Zustand der Klasse gebraucht werden und in dem anderen unnütz sind. Natürlich hätte die Möglichkeit bestanden einfach ein paar Funktionen virtuell zu gestalten, eine neue Klasse ListBox anzulegen und dann von ComboBox abzuleiten. Doch auf diese Möglichkeit bzw. Idee bin ich erst später gekommen, wo beides schon fertig war und uns leider schon die Zeit zu sehr im Nacken hing.
Im Folgenden werde ich die konventionelle ComboBox erklären und an den entsprechenden Stellen angeben, worin und wie sich die ListBox unterscheidet.
Wie schon erwähnt besteht die ComboBox in ihrem Grundzustand einfach aus einer Box, einem Button. Dieser Button bildet den Kopf einer ComboBox, er zeigt das aktuell angewählte Element an und kann durch Anklicken die ComboBox aufklappen und ihre Einträge anzeigen lassen.

Abbildung 32 - Beispiele für ComboBoxen

Die Einträge wiederrum bestehen auch aus Buttons die angeklickt werden können, um sie auszuwählen. Wird dies getan wird der Eintrag oben im Kopf der ComboBox angezeigt. Darüber hinaus besitzt die ComboBox noch eine Scrollbar, um sie mit beliebig vielen Einträgen zu füllen und zwischen diesen auswählen zu können.

Eine ListBox unterscheidet sich an dieser Stelle dahingehend, dass sie alle Einträge anzeigt, den ausgewählten Eintrag farblich hinterlegt und ihn damit als ausgewählt kennzeichnet. Es kann in der ComboBox wie auch in der ListBox immer nur ein Eintrag gleichzeitig ausgewählt sein. Darüber hinaus besitzt die ListBox noch einen Background als PictureControl.

Abbildung 33 - Beispiel für eine ListBox

Die Einträge von beiden Dialogitems werden in einer Liste gespeichert und zwar auf Zeiger auf CButton.
Es ist eigentlich so geregelt, dass Dialogitems die nicht eigenständig sind sondern ein Teil eines anderen Dialogitems sind (ChildrenDialogitems), nicht in der Dialogitemliste des ParentDialog gespeichert werden. Bei der ComboBox aber liegen die Einträge nicht direkt auf oder innerhalb der Grenzen des Dialogitem, sondern darunter und können im aufgeklappten Zustand auch andere Dialogitems überlappen. Deswegen müssen sie extern über das Event-System angesprochen werden können. Dazu müssen die Einträge und die Scrollbar in die Dialogitemliste des zugehörigen Dialogs gespeichert werden. Die Einträge und die Scrollbar reagieren nun auf Events, als wären es eigenständige Dialogitems.

Die Funktionalität scheint eigentlich recht einfach und nicht all zu aufwendig umsetzbar zu sein, doch der Schein trügt. Ich habe sehr lange gebraucht, um diese Klasse zu programmieren. Dies lag unter anderem daran, dass ich den Code an ComboBox und ListBox anpassen musste, so wie an der Scrollbar. Denn die Scrollbar an sich ist ein sehr einfach Dialogitem, erst durch das Einbetten einer Scrollbar in ein Dialogitem oder Dialog wird die Funktionalität kompliziert und aufwendig zu programmieren.

Diese Klasse ist eine sehr umfangreiche und wichtige Klasse, die in vielen Anwendungsgebieten unabdingbar ist. Aber ich finde diese Klasse ist leider nicht optimal realisiert worden. Was zum einen daran liegt, dass ComboBox und ListBox in einer Klasse sind und dass diese Klasse sehr „schwer“ ist. D.h. sie beinhaltet viel Code und eine große Datenmenge, was sich z.B. beim Erstellen klar deutlich macht im Vergleich zu einem Label.
Es ist die gleiche Aussage wie immer: „Die Zeit reicht einfach nicht“. Hätte ich mehr Zeit gehabt, hätte ich diese Klasse noch um einiges besser umsetzen können. Nichts desto trotz bin ich stolz auf dieses Dialogitem, da es viel Arbeit gekostet hat und im Endeffekt sehr großen Nutzen beinhaltet. Aber leider nicht völlig perfekt, aber gut umgesetzt ist.

Eine gesamt Übersicht über die Funktionen und Attribute befindet sich im Anhang (11.1.6)

6.5.3.4. EditControl

Das EditControl ist ein Eingabefeld für Buchstaben und Zahlen. Es funktioniert wie die allgemein bekannten Eingabefelder. Man wählt es aus und gibt Text ein, dabei kann man mit den Pfeiltasten und der Maus zwischen den schon eingegebenen Buchstaben hin und her navigieren. Das EditControl besteht aus einem Background und einem Label, welches den eingegebenen Text darstellt.

Abbildung 34 - Beispiel für ein EditControl

Das EditControl bietet verschiedene Eingabemöglichkeiten. Zum einen kann man auswählen, ob der Inhalt des EditControls beim ersten Mal anklicken gelöscht werden soll oder der TextCursor an die angeklickte Stelle springen soll. Zum anderen sind verschiedene Eingabetypen möglich. Dies sind enSTRING, enFLOAT, enINT und enBYTE, die Namen hierbei sollten selbsterklärend sein.
Der TextCursor des EditControls markiert die Stelle an der die neue Eingabe eingefügt wird, es ist der blinkende Balken, den man aus Eingabefeldern kennt.

Es ist möglich mit verschiedenen Tasten in dem EditControl zu navigieren, z.B. die Tasten „Pos1“ und „Ende“. Aber besonders die Pfeiltasten spielen eine entscheidende Rolle. Insbesondere dann, wenn man mehr Text in das EditControl eingegeben hat, als es anzeigen kann. Denn der Text wird dann nicht einfach über die eigentliche Breite des EditControls geschrieben, sondern der Text verschiebt sich. Diese Verschiebung war anfangs nicht einfach umzusetzen und ich habe lange gebraucht, bis ich dann eine gute und auch banal einfache Lösung gefunden hatte.

Aber auch, wie bereits erwähnt, kann mit der Maus durch das EditControl navigiert werden. Dies funktioniert aber nicht ganz so gut, wie man das von anderen Eingabefeldern kennt. Das Navigieren funktioniert so, dass ich die einzelnen Breiten der Buchstaben des eingebebenen Textes aufaddiere, bis ich die Mausposition erreicht habe. Zum Glück bietet SFML dazu Funktionen, um z.B. an die Breite der Buchstaben heran zu kommen, sonst wäre dies wahrscheinlich nicht möglich gewesen.

Eine weitere Sache ist beim EditControl noch zu beachten. Um normalerweise an den Text von Dialogitems heran zukommen, fragt man dies einfach über das Label ab. Denn durch den Textwriteout „enEDITCONTROLCURSORBLINK“, den das Label benutzt um den Cursor blinken zulassen, wird der Inhalt des EditControls modifiziert. Aber man möchte nur den wirklich eingegebenen Text zurück haben. Deswegen muss man in diesem Fall die Funktion „get_EditControlContent“ benutzen.

6.5.3.5. Label

Ein Label ist eine Beschriftung oder ein Text der angezeigt wird.
Das Label basiert auf dem SFML Objekt sf::Text, welches das Zeichnen und Anzeigen von Text ermöglicht. Dieses Textobjekt wird durch eine Font (Schriftart), Textfarbe, Textstyle, Charactersize (Buchstabengröße) und dem Inhalt definiert. Die Verwaltung dafür und für weitere Funktionalitäten übernimmt das Label, doch die Funktionalitäten bietet die Klasse sf::Text selber. Darüber hinaus ist es möglich den Inhalt des Labels durch verschiedene Textwriteouts zu animieren. Bisher gibt es vier Arten von Textwriteouts: „enSTANDARD“, „enCOD“, „enEDITCONTROLCURSORBLINK“ und „enNONE“. Das erste sorgt dafür, dass die einzelnen Buchstaben des Textes nacheinander in einer vorgegebenen Zeit erscheinen. „enCOD“ zeigt alle Buchstaben direkt an, doch sie werden durch zufällige, ständig wechselnde Zeichen ersetzt. Im Verlauf der vorgegebenen Zeit kristallisieren sich die richtigen Buchstaben nach und nach heraus.

(„Profiles“)

Abbildung 35 - Verlauf des TextwriteoutStyles 'COD'

Der Name kommt daher, dass es in dem Video-Spiel „Call of Duty – Modern Warfare“ einen ähnlichen TextwriteoutStyle gibt, welcher mich zu diesem inspiriert hat. Der dritte TextwriteoutStyle ist genau genommen kein richtiger, denn er sorgt nur dafür, dass bei einem EditControl der Textcursor blinkt und ausschließlich dafür gedacht ist.

Abbildung 36 - Beispiel für den TextwriteoutStyle 'EditControlCursorBlink'

Eine Problematik von sf::Text ist, dass die Charactersize des Objekts nur die Größe in Y-Richtung der Buchstaben verändert, wodurch Texte in unterschiedlichen Auflösungen unterschiedlich breit sind. Dies wurde dadurch gelöst, dass die Charactersize immer konstant bleibt und die Buchstaben einfach auf die passenden Werte skaliert wurde. Eine weitere Problematik von sf::Text ist, dass wenn gewisse Funktion, z.B. das Zeichnen in Threads ausgelagert werden, dass in dem Text-Objekt Buchstaben verschwinden.

Der TextStyle des Labels wird über einen sf::Uint32[52] Wert angegeben. Es gibt drei Arten von TextStyles, diese können beliebig kombiniert werden: „Bold“, „Italic“ und „Underlined“. Dies Kombinationen ergeben sich aus der Summe des TextStyles

- TextStyles:
  - 0 = Regular
  - 1 = Bold
  - 2 = Italic
  - 3 = Bold&Italic
  - 4 = Underlined
  - 5 = Bold&Underlined
  - 6 = Italic&Underlined
  - 7 = Bold&Italic&Underlined

Abbildung 37 - Beispiele für Textstyles

(Bold, Italic, Underlined und alle drei zusammen)

Die Nachteile des Labels und allgemein von sf::Text sind die Formatierung. Es besteht bei sf::Text nicht die Möglichkeit den Text innerhalb des Objektes zu formatieren. Dies ist auch beim Label so, wobei es sich nur um mehrzeiligen Text handelt, einzeiliger Text kann einfach über die Position des Textes formatiert werden.

[52] Datentyp der SFML: In diesen Datentyp kann eine ein Byte große Zahl gespeichert werden

Das Label ist einer der Grundbausteine der GUI. Texte und Beschriftungen werden oft gebraucht und verwendet. Aber auch in anderen Dialogitems wie z.B. bei Buttons oder EditControls.

Eine gesamt Übersicht über die Funktionen und Attribute befindet sich im Anhang (11.1.8)

6.5.3.6. PictureControl

Die Klasse PictureControl ist das einfachste Dialogitem, da es im Endeffekt nur ein Objekt von sf::Sprite verwaltet, die gesamte Funktionalität bietet die Klasse von SFML. Diese Klasse beinhaltet auch nur insgesamt vier eigene Variablen.

Dennoch ist diese Klasse Grundbaustein und essentiell für viele Dialogitems. Aber diese Klasse dient nicht nur als Grundbaustein für neue Dialogitems, sondern auch einfach dazu um Bilder anzuzeigen.

Z.B. die Lampe des Pads im Hauptmenü, die sich je nach angewähltem Button verändert ist ein PictureControl:

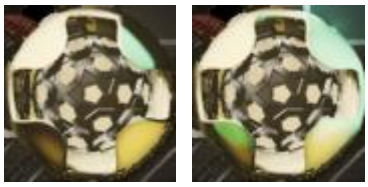


Abbildung 38 - Beispiele für PictureControls

Des Weiteren ist es aber auch möglich PictureControls um einen gewünschten Winkel zu drehen.
Es ist zwar theoretisch möglich, dass PictureControls auch auf Events reagieren können, doch dies wurde nicht implementiert, da diese Klasse einfach nur dazu da sein soll Bilder anzuzeigen und zu verwalten. Diese Klasse verwaltet nur die ganzen Funktionalitäten die die Klasse sf::Sprite wie von SFML bietet.

Eine gesamt Übersicht über die Funktionen und Attribute befindet sich im Anhang (11.1.9)

6.5.3.7. Scrollbar

Eine Scrollbar ist, wie der Name schon sagt, eine Leiste über die „gescrollt“/“geblättert“ werden kann. Sie dient dazu durch gewisse Bereiche zu „blättern“, um ihren gesamten Inhalt anzusehen, der zu groß ist um ihn komplett bzw. gleichzeitig darzustellen.

Das Dialogitem Scrollbar besteht aus zwei Elementen. Zum einen dem Background welcher ein PictureControl ist und zum anderen aus dem Slider, dieser wurde durch einen Button realisiert.

Es war nicht einfach sich zu überlegen, wie die Scrollbar nun genau funktionieren soll. Ich habe es so gelöst, dass die Scrollbar in Blocks unterteilt ist. Mit dem Slider kann nun jeder einzelne Block durch Ziehen oder Scrollen mit dem Mausrad angewählt werden.
Bei den Blocks wird zwischen „available“-Blocks und „current“-Block unterschieden. Das erste gibt an in wie viele Teile die Scrollbar unterteil ist und das zweite, welcher der aktuell angewählte Block ist. Durch das Block-System gibt es keinen reibungslosen Übergang zwischen den Inhalten, was z.B. bei der ComboBox deutlich wird. Es wäre aber möglich einen solchen Übergang zu erzeugen. Bisher wurde die Scrollbar nur in der Klasse ComboBox eingesetzt und dort war der blockweise Übergang völlig ausreichend.

Abbildung 39 - Beispiel für eine Scrollbar

Die Scrollbar übernimmt genau genommen gar keine Verwaltung des anzuzeigenden Bereichs. Was genau und wie bei welchem Block angezeigt werden soll entscheidet das Dialogitem bzw. die Klasse in die die Scrollbar eingebettet ist. Hierdurch wird auch ein reibungsfreier Übergang der Inhalte leicht implementiert, da dazu die Scrollbar an sich nicht verändert werden muss. Womit die Scrollbar nur zur Darstellung zuständig ist.

Es gibt bestimmt noch bessere Möglichkeiten eine Scrollbar zu realisieren, aber ich denke meine Variante ist für unsere Anforderungen völlig ausreichend. Es war auch gar nicht so leicht sich dieses System auszudenken und es ordentlich umzusetzen. Es hat sehr lange gedauert die Scrollbar und die ComobBox so mit einander zu verbinden, dass sie richtig und gut funktionieren.

Eine gesamt Übersicht über die Funktionen und Attribute befindet sich im Anhang (11.1.10)

6.5.3.8. SlideControl

Das SlideControl ist ein Schieberegler mit dem vorgegebene Werte eingestellt werden können. Beispielsweise die Lautstärke der Musik des Spiels, diese kann z.B. zwischen 0% und 100% liegen. Über den Schieberegler lässt sich nun der gewünschte Wert, über das Bewegen des Reglers, einstellen.

Das SlideControl besteht aus 3 Elementen. Zum einen den Background als PictureControl, zum anderen dem Regler, der als ‚Bar' realisiert wurde und ebenfalls aus einem PictureControl besteht. Das dritte Element ist das ValueDisplay, welches den aktuellen Wert des SlideControls anzeigen kann, dieses besteht aus einem Label.

Abbildung 40 - Beispiel für eine SlideControl Textur

Der Background des SlideControls ist immer statisch. Durch Anklicken und Bewegen der Maus lässt sich nun der gewünschte Wert einstellen. Dabei bewegt sich die Bar und wird, je nach Richtung in der Horizontalen länger oder kürzer.

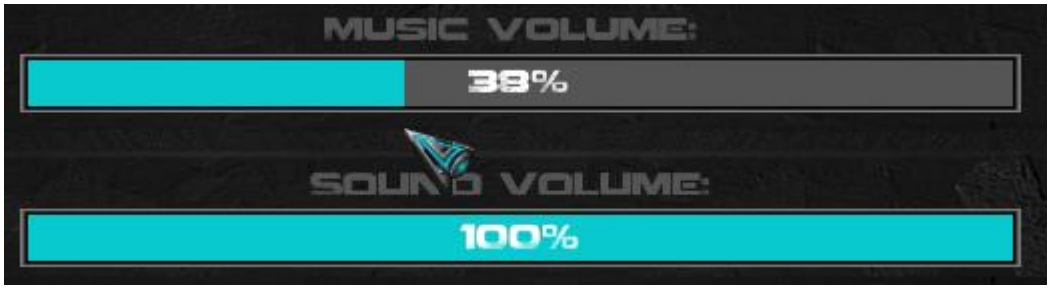


Abbildung 41 - Beispiele für SlideControls

Wenn ein ValueDisplay angelegt wurde wird der aktuelle Wert des SlideControls angezeigt. Dieses Label ist frei platzierbar und hat unterschiedliche Möglichkeiten den Wert darzustellen. Es ist möglich den Wert in Prozent und zwei Nachkommastellen anzuzeigen, doch diese Einstellungen sind optional.

Abbildung 42 - Beispiele für unterschiedliche DisplayValues

Das SlideControl ist ein recht einfaches, aber nützliches Dialogitem.
Eine gesamt Übersicht über die Funktionen und Attribute befindet sich im Anhang (11.1.11)

Dies waren alle wichtigen Klassen des Dialog-System, nun folgend das Event-System, was die Funktionalität des Dialog-Systems garantiert sowie die Wiederverwendbarkeit der GUI in anderen Anwendungen.

6.5.4. Event-System

Das Event-System verwaltet die Eingabe des Benutzers, ermöglicht ihm die Interaktion mit der GUI und gewährleistet somit auch ihre volle Funktionalität. Das Event-System war anfangs direkt in die GameStates eingebettet, was unter anderem daran lag, dass es zu dem Zeitpunkt seiner Entstehung noch keine Schnittstellenklasse zwischen dem Dialogsystem und den GameStates gab. In Folge von Überarbeitungen wurde es in den Dialogmanager integriert. Seine Funktionsweise ähnelt einem scriptbasieretem System und könnte auch noch mit LUA so umgesetzt werden.

Der Ablauf des Event-Systems sieht folgendermaßen aus:

In der „Main-Schleife“ (ety::CEternity-Engine::start_Engine) wird bei jedem Durchgang die „handle_Events“-Funktion der CEternity-Engine aufgerufen. In dieser Funktion wird eine „While-Schleife“ durch gegangen, die die „poll_Event“-Funktion der CApplication aufruft. Sie fragt ab, ob ein Event eingegangen ist und schreibt dieses in die übergebene Variable, die einen Bool-Wert zurückgibt. Die Events von SFML funktionieren wie ein

Stapel und die „poll_Event“-Funktion des sf::RenderWindow gibt immer das älteste Event des Stapels zurück.

Das neu eingegangene Event wird dann an den GameStatemanager weiter geleitet, welcher es dann an den aktuellen GameState weitergibt. Vom aktuellen GameState wird nun die „handle_GameStateEvents“-Funktion aufgerufen. In dieser Funktion werden alle eingehenden Events bearbeitet und die gewünschten Funktionen und Aktionen ausgeführt.

Bei DialogGameStates ist das Grundgerüst immer gleich. Zuerst werden alle Aktionen ausgeführt, die mit „KeyEvents“ (Drücken oder Loslassen von Tasten) zusammenhängen und mit keinem Dialogitem verbunden sind.

```
if(sfEventLastEvent.type == sf::Event::KeyReleased)
{
    if(sfEventLastEvent.key.code == sf::Keyboard::Escape)
    {
        if(mp_c_etyDialogManager->get_DialogbyCustomID("MainMenuExitMenu")->get_Visibility() == true)
        {
            //Schließen des Exit-Menüs
            mp_c_etyDialogManager->get_DialogbyCustomID("MainMenuExitMenu")->set_Visibility(false);
            mp_c_etyDialogManager->get_DialogbyCustomID("MainMenuExitMenu")->set_Modal(false,sf::Vecto
        }
```

Abbildung 43 - Beispiel für ein KeyReleasedEvent

Danach werden in einer „else if“-Verzweigung alle Aktionen bei einem „MouseMoved“-Event ausgeführt, wenn kein Dialogitem auf dieses Event reagieren kann. Danach wird die „handle_Events“-Funktion des Dialogmanagers aufgerufen und die passende Event-Funktion des Dialogitems, sofern vorhanden, aufgerufen. Diesen Schritt werde ich später nochmals näher erläutern.
Der Dialogmanager besitzt vier Variablen, Zeiger auf CDialogitem, die für das Event-System sehr wichtig sind. Dies sind die Variablen „mp_c_etyDialogitemFocusedDialogitem“ (Zeiger auf das momentan fokussierte, zu Letzt angeklickte, Dialogitem) , „mp_c_etyDialogitemMouseEvent“ (Zeiger auf das Dialogitem, welches auf Events regieren kann und im Fokus der Maus liegt), „mp_c_etyDialogitemDoubleClickEvent“ und „mp_c_etyDialogitemNoFocusRelease“ (Beides Zeiger auf das zuletzt angeklickte Dialogitem, nähere Informationen und Erklärungen später im Text).

Die wichtigste Variable für das Event-System ist die Variable „mp_c_ etyDialogitemMouseEvent“, denn in dieser steht das einzige Dialogitem was auf Events reagieren kann und soll, da es im Fokus der Maus liegt.
Diese Variable wird immer in der Update-Funktion des Dialogmanagers gesetzt. Bei jedem Update-Durchlauf der Dialoge geben diese einen Zeiger auf ein mögliches Dialogitem zurück, welches auf Events reagieren kann und im Fokus der Maus liegt. Da die Dialoge und die Dialogitems geordnet in den Listen stehen wird auch nur am Ende in dem Zeiger das Dialogitem stehen, welches am weitesten im Vordergrund liegt, selbst wenn mehrere Dialoge einen Zeiger zurückgeben und nicht „NULL“.

Wieder zurück zu der „handle_Events“-Funktion des Dialogmanagers. In dieser Funktion werden zuerst bei Events, die mit der unmittelbaren Funktionalität des Dialogsystems zu tun haben behandelt, z.B. das Scrollen mit dem Mausrad. Danach wird die „handle_internEvents“-Funktion des EventDialogitems („mp_c_etyDialogitemMouseEvent“) aufgerufen. Durch diese Funktion, die von Dialogitem zu Dialogitem unterschiedlich ist, wird die Funktionalität des Dialogitems gewährt, z.B. wird bei einem „MouseButtonReleased“-Event einer ComboBox diese „aufgeklappt“ und der passende Sound abgespielt.
Die ganze „handle_Events“-Funktion des Dialogmanagers sichert nur die Funktionalität der GUI und ist nicht für explizite Events zuständig!
SFML bietet zwei wichtige Events nicht: „MouseButtonDoubleClick“ (Doppelter Klick auf ein Dialogitem) und „MouseButtonNoFocusRelease“ (Loslassen der Maustaste, ohne das Dialogitem im Fokus zuhaben, auf dem vorher die Maus gedrückt wurde).

Beide Events basieren auf dem „MouseButtonReleased“-Event und dadurch muss ein solches Event ggf. in eines der beiden umgewandelt werden. Diese Umwandlung findet auch in der „handle_Events“-Funktion des Dialogmanagers statt.

Nachdem der Dialogmanager das eingehende Event verarbeitet und ggf. umgewandelt hat, wird wieder im GameState die passende Event-Funktion des Dialogitems, sofern vorhanden, zum passendem Event aufgerufen.
In jedem DialogGameState gibt es eine „EventMap“. Diese std::map besteht auf der linken Seite aus der CustomID des Dialogitems als std::string und auf der rechten Seite aus einer weiteren std::map. Diese setzt sich links aus einem Enum ety::EventType und rechts aus einem Funktions-Zeiger zusammen.
Somit steht in dieser Map zu jedem Dialogitem eine weitere Map mit einem Event und einem dazu passendem Funktions-Zeiger, über diesen wird dann letztlich auch die passende Event-Funktion aufgerufen.
Dadurch, dass sf::Event in der Map nicht funktionierte, mussten wir ein eigenes Enum benutzen. Das sf::Event wird in der „get_EventFunction“-Funktion, welches den Funktions-Zeiger zurückgibt, in das Enum umgewandelt.

Wenn es nun zu dem Event und dem Dialogitem einen FunktionZeiger gibt, wird dieser aufgerufen.

```
sf::Event c_sfEvent = sfEventLastEvent;
mp_c_etyDialogManager->handle_Events(c_sfEvent);

if(p_c_etyDialogitemEvent != NULL)
{
    p_Function p_FunctionEvent = get_EventFunction(p_c_etyDialogitemEvent->get_CustomID(), c_sfEvent);
    std::string strCustomID = p_c_etyDialogitemEvent->get_CustomID();

    if(p_FunctionEvent != NULL)
    {
        return p_FunctionEvent(1, mp_c_etyDialogManager);
    }
}
```

Abbildung 44 - Beispiel eines allgemeinen Eventaufrufs

Das Besondere an den Event-Funktionen ist, dass sich beliebig viele Parameter übergeben lassen. Damit lässt sich alles in den EventFunktionen realisieren.

Standardmäßig wird aber nur ein Zeiger auf den Dialogmanager übergeben, da über ihn auf alle relevanten Dinge, Dialoge/Dialogitems und GameStates zugegriffen werden kann. Denn die Event-Funktionen müssen alle static-Funktionen sein, dadurch kann in ihnen auch nur auf static-Variablen zugegriffen werden, was die Übergabe des Dialogmanagers löst.

Die Realisierung, dass beliebig viele Parameter übergeben werden können funktioniert folgendermaßen:
Die Parameterliste der Funktions-Zeiger beinhaltet zuerst einen Integer, welcher die Anzahl der Parameter angibt. Danach folgen drei Punkte, welche die beliebigen Übergabewerte signalisieren.

```
typedef ety::GameStateRunning::en_etyGameStateRunning (*p_Function)(int, ...);
```

In den Event-Funktionen werden die Parameter über die „va-Makros“ ausgelesen. Zuerst wird die Argumentenliste (va_list) mit der Anzahl der Argumente bzw. Parameter initialisiert und dann können mit der „va_arg“-Funktion die einzelnen Parameter ausgelesen werden und per cast-Operator wieder in den gewünschten Datentyp konvertiert werden. Anschließend muss noch die Argumentenliste wieder „geschlossen“ werden.

Diese Makros sind in C geschrieben und die Argumentlisten basieren auf einem char-Feld, wir haben aber keine Möglichkeit in C++ gefunden.
Zur Erklärung des Bildes: Wir haben die Funktionen per Typedef bzw. Define umbenannt, um sie aussagekräftiger zu machen.

```
typedef va_list                 va_ArgumentList;
#define va_OpenArgumentList     va_start
#define va_getArgument          va_arg
#define va_CloseArgumentList    va_end
```

Abbildung 45 - Typedef und Defines der va-Operatoren

```
ety::GameStateRunning::en_etyGameStateRunning   ety::CMenuState::event_startGame    ( int iAmount, ... )
{
    va_ArgumentList va_ALParameter;
    va_OpenArgumentList(va_ALParameter, iAmount);

    CDialogManager* p_c_etyDialogManager =
        static_cast<CDialogManager*>(va_getArgument(va_ALParameter, void*));

    p_c_etyDialogManager->get_ParentGameState()->
        get_GameStatemanager()->pushback_GameState( "SpaceStationState" );

    va_CloseArgumentList(va_ALParameter);

    return ety::GameStateRunning::enFALSE;
}
```

Abbildung 46 - Beispiel für eine Event-Funktion

Jetzt fehlt letztlich nur noch die Verbindung der EventFunktion mit dem Dialogitem.
Diese Verbindung wird in der „Init“-Funktion erzeugt:

```
enum EventType::en_etyEventType en_etyEventType = EventType::enNONE;
std::map<enum EventType::en_etyEventType, p_Function> mapEventTypeFunction;
```

```
//////////////////////////EVENTS////////////////////////////////
en_etyEventType = EventType::enMOUSELEFTRELEASED;
mapEventTypeFunction[en_etyEventType] = &event_startGame;
en_etyEventType = EventType::enMOUSEHOVER;
mapEventTypeFunction[en_etyEventType] = &event_PadStartButtonHover;

m_EventMap["PadStartButton"] = mapEventTypeFunction;
mapEventTypeFunction.clear();
//////////////////////////EVENTS////////////////////////////////
```

Abbildung 47 - Beispiel für eine Eventverlinkung

Es wird der EventType des Events festgelegt und in einer std::map mit dem Funktions-Zeiger verbunden und zum Schluss über die CustomID des Dialogitems in die EventMap gespeichert.

Das ist die komplette Funktionalität des Event-System und ich denke es ist ziemlich gut gelungen, besonders im Vergleich zu vorher.
Der große Vorteil von unserem Event-System ist, dass wir jedes Event ordentlich in einer Funktion haben ohne „Spaghetti-Code“ zu erzeugen. Des Weiteren ist die Leistung dieses System besser.

6.5.5. Wiederverwendbarkeit

Unser Spiel/Projekt besteht aus zwei Teilen. Zum einen die Engine mit ihrer Ressourcenverwaltung und der GUI. Zum anderen die Klassen und Komponenten, die zum Spiel direkt gehören, wie z.B. das Component-System oder die Trigger. Dadurch besteht die Möglichkeit, beide Teile zu trennen und die Engine in anderen Anwendungen wieder zu verwenden.

Mit der Engine hat man die Möglichkeit, alle möglichen Anwendungsprogramme zu schreiben. Die Engine besitzt leider keinen Editor zum Formatieren der GUI, wie es bei MFC der Fall ist. Wäre noch genug Zeit gewesen, dann hätte ich noch einen solchen Editor bzw. die Möglichkeit zur einfachen Formatierung geschrieben, da es aber nicht zum Projekt direkt gehört, wurde erst einmal darauf verzichtet. Da diese Möglichkeit nicht besteht, muss die Formatierung selbst programmiert werden.

Die Frage ist nun, was würde es für Vorteile bringen unsere Engine zu benutzen, um Anwendungen zu programmieren. Es gibt einige Vorteile die unsere Engine gegenüber MFC bietet.
Grundsätzlich lässt sich sagen, dass die Programmierung der Funktionalität einfacher und übersichtlicher ist als in MFC. Selbst das Erstellen und Designen von GUI-Elementen ist einfacher. Das liegt daran, dass man den Code zu Erstellung der GUI und verschiedenen Fenstern, sowie die passenden Events alle in einer Quellcodedatei gebündelt hat. Um die Übersicht noch zu verbessern, kann man verschiedene Menüs und Fenster noch in andere States auslagern.

Ein Nachteil unserer Engine ist die Vielfalt der Dialogitems, MFC bietet ein sehr breit gefächertes Angebot an Steuerelementen, unsere Engine nur die Wichtigsten. Doch wer eine einfache Anwendung schnell und unkompliziert programmieren will, hat dennoch Vorteile gegenüber MFC, da eine viel bessere Übersicht über den eigenen Quellcode gegeben ist und die Verwendung der Funktionalitäten der GUI einfacher ist.

Um die Engine zu benutzen, benötigt man zum einen SFML und zum anderen nur gewisse Klassen der Eternity-Engine. Diese wären: Teile der Assistent-Klassen, alle Klassen des Dialog-Systems, die Klassen des Gamestate-Systems, das Ressourcen-System und das Settings-System. Auf der CD liegt ein fertiges Projekt, welches als Vorlage zum Programmieren einer Anwendung genutzt werden kann.

Dies waren nun alle Aspekte des Dialogsystems, welches den Hauptteil meiner Arbeit bildet. Im Folgenden geht es mit Klassen und Aspekten weiter, die nicht mehr direkt mit der Engine zusammenhängen, sondern mit dem Spiel an sich. Angefangen mit dem Entity-System, welches Julien geschrieben hat.

6.6. Entity-System (J. S.)

Das Entity[53]-System ist die Verwaltung der Spielobjekte. Mit diesem System lassen sich Spielobjekte erstellen und löschen, ebenso neue Spielobjekte definieren. Normalerweise würde man zu einer normalen Vererbungs-Hierarchie zurückgreifen um Spielobjekte zu definieren. In der nächsten Abbildung sehen wir ein kleines Beispiel einer solchen Hierarchie:

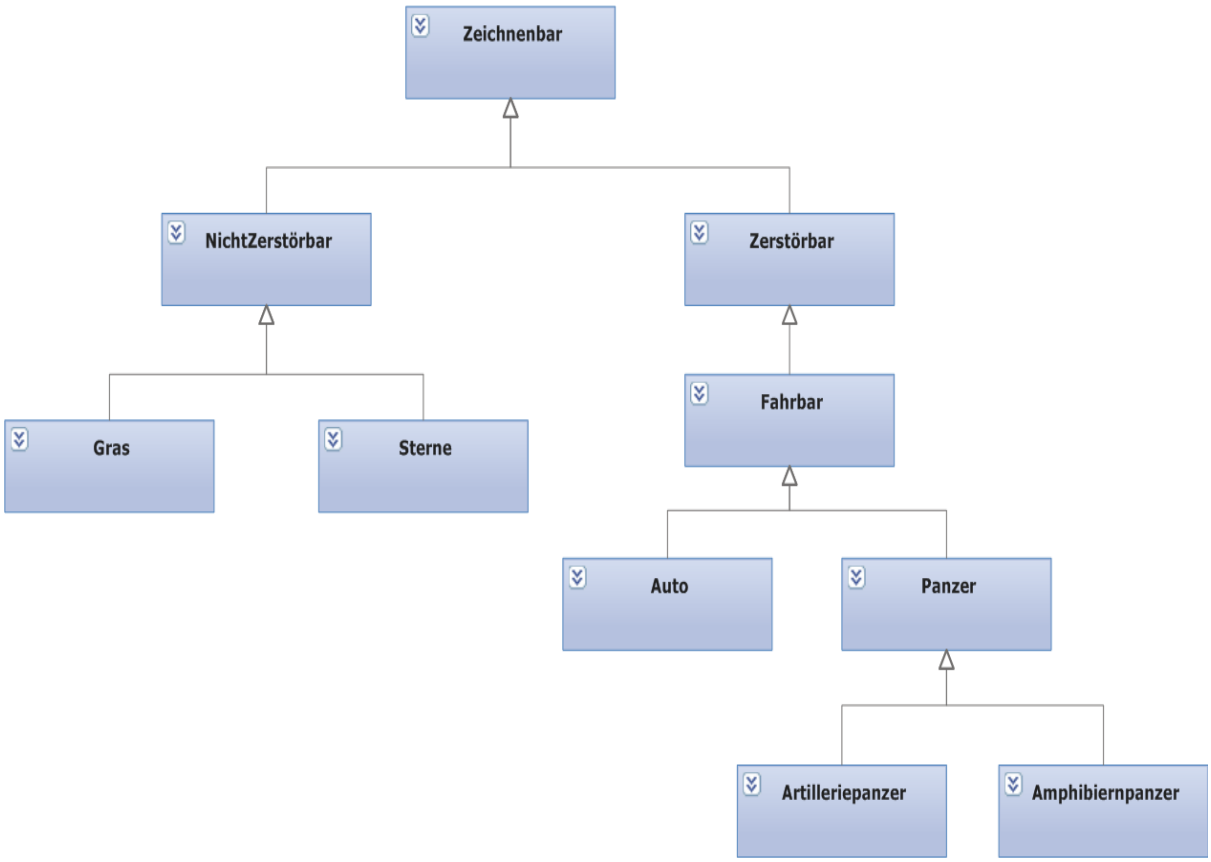


Abbildung 48 - UML-Diagramm: Beispiel einer ganz normale Vererbungs-Hierarchie

Eine solche Hierarchie ist für den Anfang leicht zu implementieren und für kleinere Projekte sehr nützlich. Sie wird jedoch zu einem Problem, wenn sie größer wird. Die Überschaubarkeit dieser Hierarchie sinkt stets und wenn solch eine Hierarchie erst einmal steht, ist es sehr schwierig mitten in der Hierarchie etwas einzufügen. Das System der Eternity-Engine hat diese Probleme nicht, ist jedoch für den Anfang etwas schwieriger zu implementieren und ebenso schwerer sich in das System rein zu denken.

Das Entity-System ist eine Art Baukasten für Spielobjekte, aber um einen genaueren Eindruck über das System zu erhalten, wird in der nächsten Abbildung das Entity-System als UML-Diagramm gezeigt:

[53] Aus dem Englischen und bedeutet „Wesen“

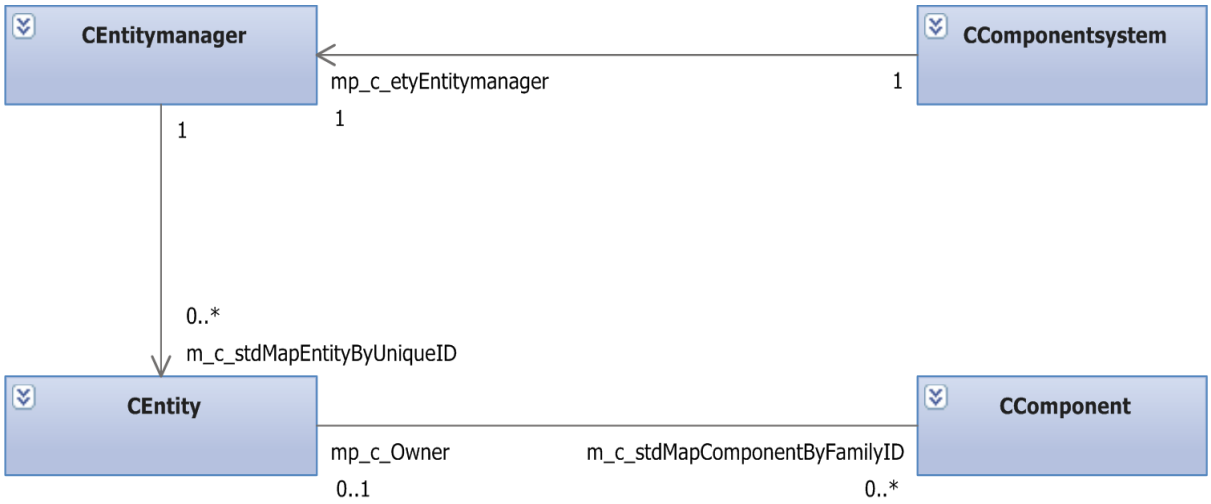


Abbildung 49 - UML-Diagramm: Entity-System

In diesem UML-Diagramm erkennt man eine relativ einfache Beziehung zwischen den Klassen. Der „Entity“-Manager beherbergt alle existierenden „Entities“, er erstellt sie und löscht sie auch wieder. Eine Instanz der Klasse „CEntity“ kann bis zu unendlich viele „CComponent“-Instanzen beherbergen, jede Instanz der Klasse „CComponent“ weiß auch über seinen „Besitzer“ bescheid. Die Klasse „CComponentystem“ besitzt eine Referenz zur Klasse „CEntitymanager“ um die Komponenten aller Instanzen der Klasse „CEntity“ verwalten zu können.

Es werden in den nächsten Abschnitten die einzelnen Klassen im Allgemeinen näher erläutert. Als erstes wird die Klasse „CEntity erklärt.

6.6.1. CEntity

Die Klasse „CEntity“ stellt unser Spielobjekt dar. Sie ist jedoch eine sehr kleine und überschaubare Klasse. Eine Instanz der „CEntity“-Klasse stellt, ohne Beherbergung einer Instanz von der Klasse „CComponent“, kein vollwertiges Spielobjekt dar. Sie ist dann nur ein einfacher einzigartiger „String“, über den sie angesprochen wird. Das ist der Grund, warum die Klasse „CEntity“ eine „Map“ von Komponenten beherbergt, denn erst durch diese erhält sie ihre Eigenschaften und Fähigkeiten, durch welche sie erst zum richtigen Spielobjekt wird.

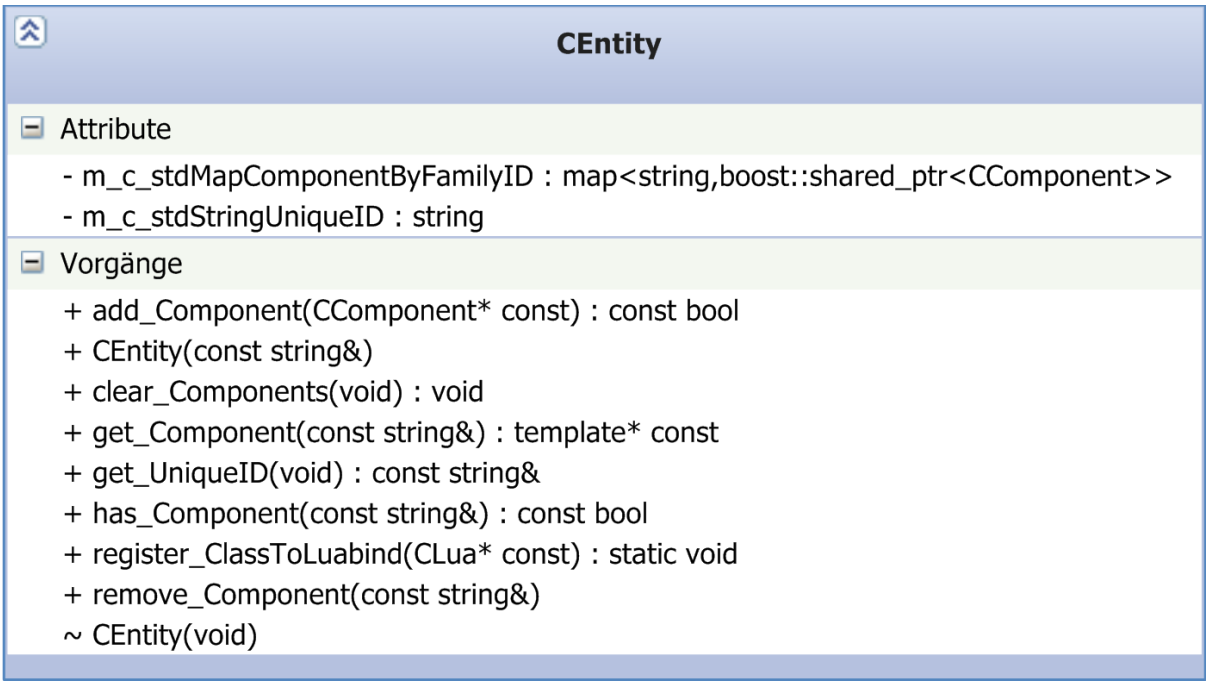


Abbildung 50 - Klassendiagramm: CEntity

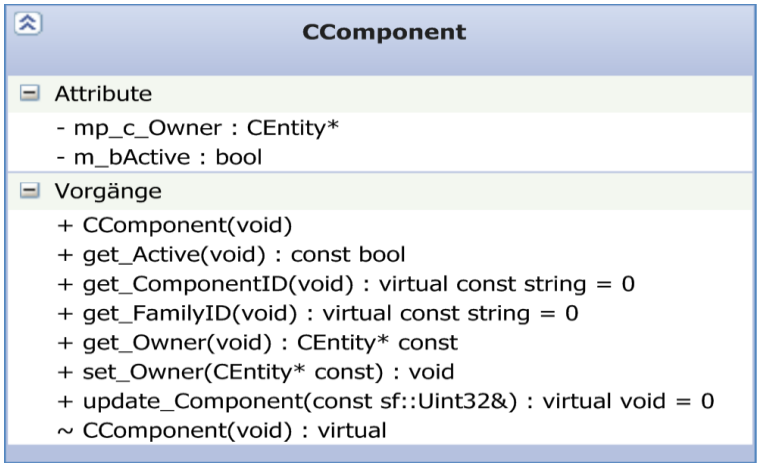
In der oberen Abbildung erkennen wir das Klassendiagramm der Klasse „CEntity“, wie oben beschrieben, sie besteht nur aus einem einzigartigen „String“ und eine „Map“ von Komponenten. Sie beherbergt außerdem Methoden zum Hinzufügen und Löschen einer Komponente. Die Klasse „CEntity“ kann ebenfalls mit Luabind in einem Script angesprochen werden. Wird jedoch zur jetzigen Zeit nur im „Trigger“- und „Quest“-System benutzt.

Im nächsten Abschnitt des Entity-Systems wird die Klasse „CComponent“ weiter erläutert, das Spielobjekt „CEntity“ besteht aus Instanzen dieser Klasse und erweckt das Spielobjekt erst zum Leben.

6.6.2. CComponent

Die Klasse „CComponent“ kann alle Arten von Eigenschaften, Fähigkeiten und Besonderheiten eines Spielobjektes darstellen, jedoch nur wenn der Programmierer von dieser Klasse ableitet und seine gewünschten Eigenschaften ausformuliert. Diese Klasse gilt für alle Komponenten eines Spielobjektes als Basisklasse. Die Klasse „CComponent“ im UML-Klassendiagramm:

Abbildung 51 - Klassendiagramm: CComponent



Wie es im Klassendiagramm zu sehen ist, beherbergt auch diese Klasse einige rein-virtuelle Methoden, welche ausformuliert werden müssen. Es gibt bei den Komponenten jedoch eine Besonderheit, neben der „update_Component“-Methode, die ausformuliert werden muss für die spezifische Aktualisierung der Komponente. Eine Komponente besitzt zwei IDs, die Komponenten-ID(„get_ComponentID“) und die Familien-ID(„get_FamilyID“). Diese werden nicht in den Attributen aufgelistet, sie werden in den zwei Methoden einfach zurückgegeben, müssen deshalb bei jeder neuen Komponente ebenfalls neu ausformuliert werden.

Die Familien-ID ist eine ID, welche die Art der Komponente beschreibt. Als Beispiel kann das unterschiedliche Bewegen der Spielobjekte ran gezogen werden. Ein Panzer hat eventuell eine andere Bewegungs-Komponente als ein Flugzeug, demnach haben die zwei verschiedenen Komponenten unterschiedliche Komponenten-IDs, jedoch dieselbe Familien-ID. Die Bewegungs-Komponente des Flugzeugs hat die Komponenten-ID „FlugzeugBewegung“ und die andere Komponente des Panzers „PanzerBewegung“, beide haben jedoch die Familien-ID „Bewegung“ gemeinsam. Daraus ergibt sich die Möglichkeit alle Spielobjekte mit bestimmten Komponenten raus zu suchen. Oft werden alle Komponente der Familien-ID „Bewegung“ gebraucht oder alle Komponente mit der Komponenten-ID „PanzerBewegung“, um sie spezifisch oder allgemein zu aktualisieren oder um die jeweiligen Typen der Spielobjekte raus zu finden, weil ein Spielobjekt ohne Beherbergung einer Komponente nichts weiter als ein einzigartiger „String“ ist.

Im nächsten Abschnitt geht es um die Klasse „CComponentsystem“, diese Klasse ist wichtig für jede Komponente.

6.6.3. CComponentsystem

Die Klasse „CComponentsystem“ ist essentiell für alle Komponenten. Diese Klasse verwaltet die Komponenten, sie führt spezifische Aktualisierungen für die jeweiligen Komponenten durch und zeichnet sie ebenfalls, je nach Komponente in unterschiedlichen Wegen. Diese Klasse ist ebenfalls eine Basisklasse und muss abgeleitet werden. Der Grund für diese Struktur ist die Vielfältigkeit einer Komponente. Eine Komponente kann jede mögliche Arbeit für ein Spielobjekt vollrichten, demnach müssen sie auch dementsprechend spezifisch aktualisiert oder/und gezeichnet werden. In unserem Spiel „Outerspace“ haben wir uns an folgende Regel gehalten. Für jede neu existierende Familien-ID für Komponenten, sollte ein neues Komponenten-System erstellt werden. Es ist meist der Fall, dass bei der Entstehung einer neuen Familien-ID für Komponenten, eine neue Art der Aktualisierung und des Zeichnens für diese Art von Komponenten vollführt werden muss. Es dient auch der Übersichtlichkeit, denn alle Komponenten-Systeme verwalten ihre jeweilige Art von Komponenten. In der nächsten Abbildung sehen wir die kleine Klasse „CComponentsystem“:

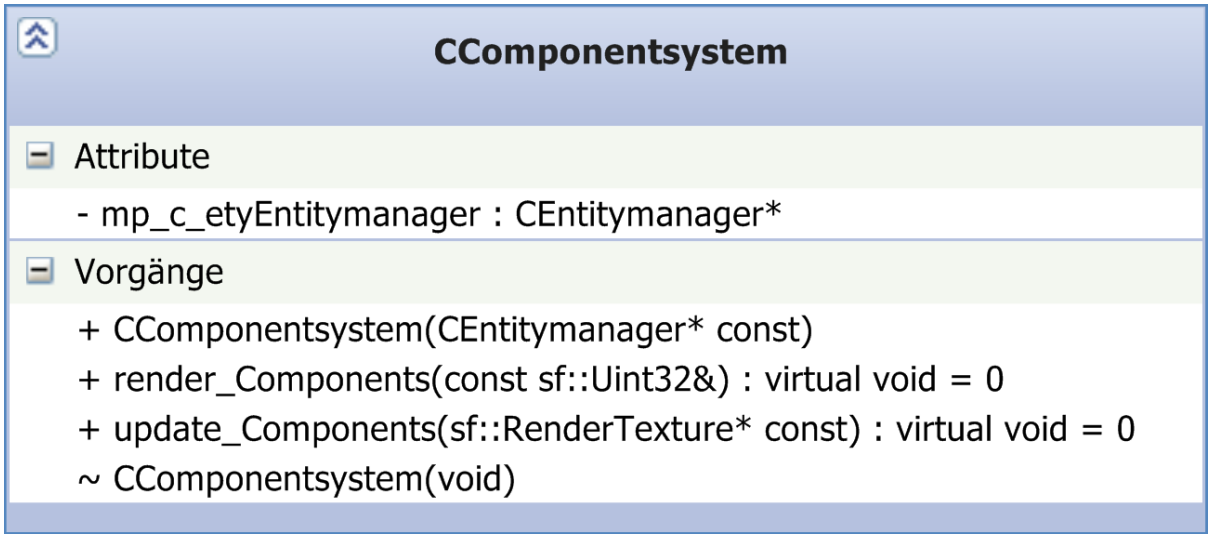


Abbildung 52 - Klassendiagramm: CComponentsystem

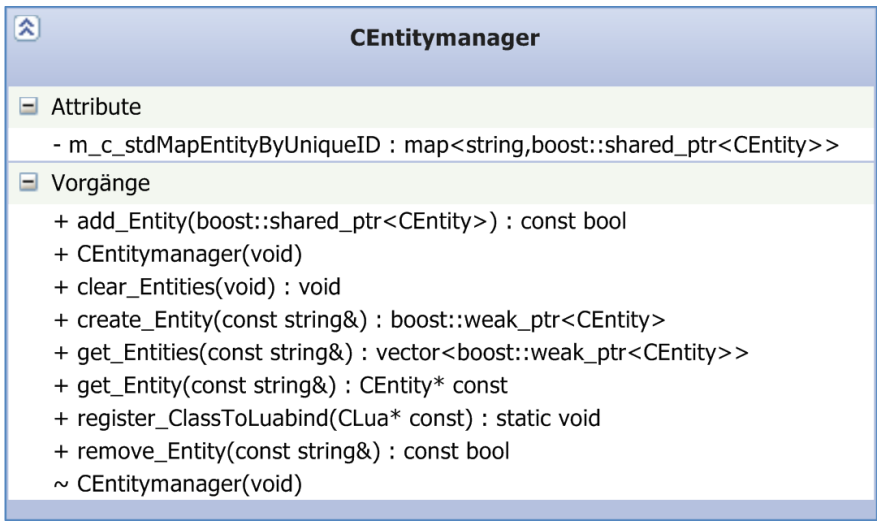
Dieses Klassendiagramm ist sehr klein und übersichtlich. Außer dem Attribut der Klasse „CEntitymanager“, sind nur noch der Konstruktor und zwei rein virtuelle Methoden zu sehen. Wie oben erwähnt muss meist jede Art von Komponente anders aktualisiert und gezeichnet werden, das ist der Grund warum diese Klasse die zwei rein virtuellen Methoden besitzt. Diese müssen bei jedem neuen Komponenten-System ausformuliert werden und an die zu verwaltende Art der Komponente angepasst werden.

Nun fehlt nur noch die Verwaltung der Instanzen der Klasse „CEntity“. Im nächsten Abschnitt geht es um die Klasse „CEntitymanager“.

6.6.4. CEntitymanager

Der „Entity“-Manager verwaltet alle Spielobjekte. Nur mit ihm ist es möglich neue Spielobjekte zu erstellen und zu löschen. Diese Klasse ist leichter zu erläutern mit der Hilfe der nächsten Abbildung:

Abbildung 53 - Klassendiagramm: CEntitymanager



Die Klasse „CEntitymanager“ besitzt eine „Map“ von Instanzen der Klasse „CEntity“. Mit Hilfe dieser „Map“ können alle Spielobjekte mit der Hilfe ihres einzigartigen „String“ angesprochen werden. Neben diesem privaten Attribut, existieren noch wichtige Methoden.

- “boost::weak_ptr<CEntity> create_Entity(const string&)”
  - Erstellt ein neues Spielobjekt mit dem übergebenen einzigartigen „String“
  - Es ist eine „Factory[54]“-Methode, deshalb gibt sie sofort das erstellte Spielobjekt zurück zum weiterverarbeiten

- „vector<boost::weak_ptr<CEntity>> get_Entities(const string&)“
  - Gibt alle Spielobjekte mit der gewünschten Komponenten-Art zurück

Das sind die zwei wichtigsten Methoden. Die anderen Methoden sind zum Löschen und zum Hinzufügen eines oder mehrerer Spielobjekte. Diese Klasse arbeitet eng zusammen mit den Instanzen der Klasse „CComponentsystem“, welche sich mit Hilfe der „Get“-Methoden der Klasse „CEntitymanager“ ihre jeweilige zu verwaltende Art von Komponenten holt.

[54] Methode zum Erstellen eines Objektes mit sofortiger Rückgabe des erstellten Objektes

7. Spiel

Neben den Elementen der Eternity-Engine existieren noch die Spiel-Elemente. Die Spiel-Elemente sind Systeme oder einzelne Klasse, welche ausdrücklich für das Spiel „Outerspace“ entwickelt wurden. Der Schwerpunkt bei diesem Projekt lag mehr auf die Entwicklung der Engine bis hin zum Spiel, als auf das Spiel an sich. Aus diesem Grund ist das Spiel noch nicht ausgereift beziehungsweise kann nicht ganz als Spiel bezeichnet werden, weil einige Systeme Bedarf nach Erneuerung oder Erweiterung haben. Bisher ist folgendes möglich:

- Karten erzeugen
- Spielobjekte auf Karten erzeugen
- Erstellung von neuen Spielobjekten
- Spielobjekte ausrüsten
- Neue Ausrüstung erstellen
- Missionen erstellen
- Auslöser erstellen

Die oben aufgelisteten Möglichkeiten, sind Möglichkeiten zum erweitern des Spiels „Outerspace“ an sich. Es gibt jedoch noch diejenigen Elemente, welche die Regeln des Spiels festlegen oder es zum Spiel werden lassen.

- Kollisionserkennung
- Bewegungen der Spielobjekte
- Künstliche Intelligenz
- Eigenschaften von Spielobjekten
- Eigenschaften von Karten

Diese Elemente müssen jedoch noch bearbeitet und erweitert werden. Es werden in Zukunft noch weitere Elemente hinzugefügt, aber nicht bis zur Präsentation dieses Projekt. Es wird eher an den bestehenden Elementen gearbeitet und verbessert, dass sie Präsentationsreif werden. Im nächsten Abschnitt geht es um die Erstellung von Karten.

7.1. CWorld

Die Klasse „CWorld“ stellt im Spiel „Outerspace“ die Karte dar, auf welcher man als Spieler herum fliegt. Sie ist eine recht große Klasse und beherbergt viele Fabrik-Methoden. Diese werden benötigt um die Karten individuell nach seinen Wünschen mit Hilfe von Script-Dateien zu gestalten. Demnach ist diese Klasse zum Erstellen aller Spielobjekte, Missionen und Auslöser zuständig. Eine Karte hat auch noch Eigenschaften, welche die Karte beschreiben. In Moment sind es folgende Eigenschaften:

- Das statische Hintergrundbild
- Start-Punkt des Spielers
- Die Größe der Karte
- Minimale und Maximale Ebene
- Die Skalierung der Spielobjekte pro Ebene

Diese Eigenschaften werden aktuell genutzt. Es sind jedoch noch weitere Eigenschaften zum Setzen verfügbar, diese werden jedoch in Moment nicht benutzt und sind für zukünftige Arbeiten bereits „verfügbar“. In den oben genannten Eigenschaften gibt es drei, welche näher erläutert werden müssen.

Das statische Hintergrundbild dient zur Atmosphäre des Spiels. Dieses Hintergrundbild ist die ganze Zeit im Spiel zu sehen und bleibt statisch auf seiner Position, während sich andere Spielobjekte von dem Spieler weg oder hin bewegen, wenn er sich entschließt sich zu bewegen.

Eine Karte ist in Ebenen aufgeteilt. Auf diesen Ebenen ist es möglich Spielobjekte zu platzieren. Je nach Ebene wird die Erscheinung des Spielobjektes in verschiedenen Arten beeinflusst. Die Ebenen liegen aufeinander und sind deshalb auch durchnummeriert. Ist die Nummer einer Ebene gering, befindet sie sich auch weiter unten. Dieses Prinzip wird in der folgenden Abbildung noch einmal verdeutlicht.

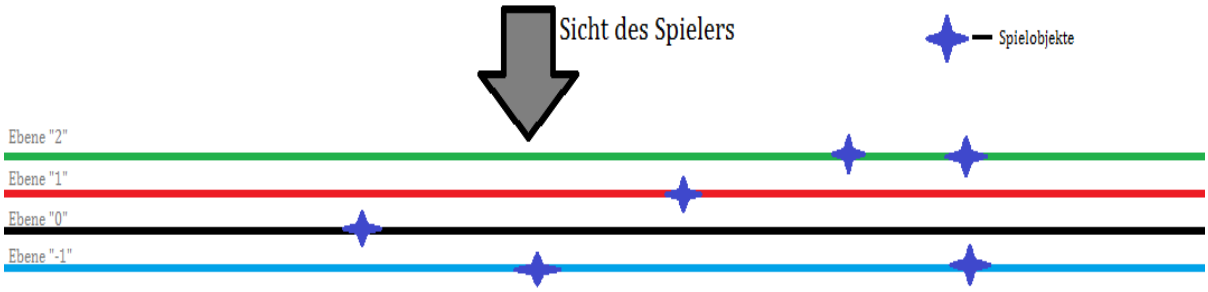


Abbildung 54 - Die Ebenen einer Karte

In dem Spiel „Outerspace“ ist die Ebene „0“ die Hauptebene, auf der sich alles abspielt und sich der Spieler bewegt. Das Vorhaben dieses Systems ist es, dem Spieler das Gefühl zu geben, dass etwas unter oder über ihm ist. Die Ebenen beeinflussen demnach folgendes:

- Die Zeichen-Reihenfolge der Spielobjekte
- Die Größe der Spielobjekte
- Die Stärke der Farben der Spielobjekte

Auf der Ebene „0“ bleibt das Spielobjekt von der Größe und von der Stärke seiner Farben unverändert. Ist ein Spielobjekt auf einer geringeren Ebene, wird die Größe des Spielobjektes herunter skaliert und die Farben werden schwächer. Das Gegenteil besteht auf höheren Ebenen als „0“, jedoch bleibt die Farbe unverändert. Die Zeichen-Reihenfolge ist hierbei ganz einfach. Die Spielobjekte der höchsten Ebene werden als erstes gezeichnet und dies zieht sich von Ebene nach Ebene bis zur Letzten runter. So ist es möglich, dass sich Spielobjekte unter oder über anderen befinden und die Veränderung der Größe und der Farbe geben das Gefühl von Weite. Die Veränderung der Größe bzw. die Skalierung pro Ebene ist für jede Karte anders einstellbar, um die Karten dynamischer zu gestalten.

An dieser Stelle verweise ich an den Anhang 12.1.4., in dem die Klasse „CWorld“ zu sehen ist. Diese Klasse besitzt folgende Methoden zum Erstellen von Spielobjekten:

- create_EnvironmentalEntity
- create_NPCEntity
- create_PlayerEntity

Erstere erstellt ein Spielobjekt, welches nur für die Umgebung nützlich ist und für Atmosphäre sorgt. Diese Spielobjekte sind auch zerstörbar, aber es soll später noch die Möglichkeit geben solche Objekte auch unzerstörbar zu machen. Denn es gibt auch Spielobjekte wie „Wurmlöcher“, welche den Spieler von einem Ort zum anderen „transportieren“. Diese sollen nicht zerstört werden können.

Die zweite Methode erstellt ein Spielobjekt, welches nicht nur zur Atmosphäre dient, sondern auch eine mögliche Bedrohung oder Erleichterung für den Spieler darstellt. Es sind die so genannten Nicht-Spieler-Charaktere. Diese NPCs besitzen eine künstliche Intelligenz und reagieren auf den Spieler oder auf andere NPCs.

Die wichtigste der drei Methoden zum Erstellen von Spielobjekten. Sie erstellt ein Spielobjekt für den Spieler selber, welches er im Spiel steuern wird.

Alle anderen „create“-Methoden der Klasse „CWorld“ sind zum Erstellen von Missionen oder Auslösern und natürlich gibt es die Methode zum Laden einer Karte.

7.1.1. Erstellung einer neuen Karte

Um eine neue Karte zu erstellen ist es vonnöten eine neue Initialisierungsdatei mit dem Namen der neuen Karte zu erstellen. Diese Initialisierungsdatei muss folgende Struktur und Werte-Paare enthalten:

Abbildung 55 - Die INI-Datei für eine Karte

```
[Alpha-Centauri-System-Map]
strStaticBackground=Alpha-Centauri-System.jpg
iStartPointX=0
iStartPointY=0
iWidth=4500
iHeight=3000
fMinLayer=-10
fMaxLayer=5
fLayerScalingAboveZero=0.20
fLayerScalingUnderZero=0.85
[Alpha-Centauri-System-CelestialMap]
strButtonTexture=Alpha-Centauri-System-Button.jpg
iWorldButtonX=100
iWorldButtonY=-100
strInfoKey=NULL
[Alpha-Centauri-System-Shop]
uiMinItemClassInShop=1
uiMaxItemClassInShop=10
```

Es ist dabei darauf zu achten, dass die Sektionen mit dem Namen der Karte anfangen und mit ihrem eigentlichen Namen fortgeführt werden. Ein Beispiel hierbei wäre die Sektion „Alpha-Centauri-System-Map“. Bei dieser Sektion ist „Alpha-Centauri-System“ der Name der Karte und „-Map“ der eigentliche Name der Sektion. Das System bedarf aber noch einiger Änderung und Erweiterungen.

7.1.2. Die erstellte Karte mit Leben füllen

Die neu erstellte Karte kann nun im Spiel verwendet werden. Sie ist aber vollständig leer, man sieht nur das gesetzte Hintergrund-Bild der Karte. Das Füllen der Karte findet über ein Script statt. Mit Hilfe dieses Scripts und der Klasse „CWorld“ ist es möglich die neue Karte zu füllen, in dem man die Fabrik-Methoden aufruft. In der nächsten Abbildung wird ein solches Script zum Füllen der Karte gezeigt:

```
if c_etyWorld ~= nil then
    c_etyWorld:create_NPCEntity("testEntity1", "AlienCarrier", sfVector2f(0, 100), 0, "Flyer");
    c_etyWorld:create_NPCEntity("testEntity2", "AlienCruiser", sfVector2f(100, 250), 0, "Flyer");

    c_etyWorld:create_EnvironmentalEntity( "AsteroidTest", "MeteoriteBig", sfVector2f( 0, 200 ), sfVector2f( -10, 10 ), -10, 25, 20 );
    c_etyWorld:create_EnvironmentalEntity( "AsteroidTest1", "MeteoriteMedium", sfVector2f( 0, 100 ), sfVector2f( 10, -5 ), -4, 45, 5 );

    c_etyWorld:create_PlayerEntity( "HumanFighter", 180 );

    c_etyWorld:create_Trigger( "initMissions", true );
    c_etyWorld:create_Trigger( "testOptionalMission", true );
end
```

Abbildung 56 - Script zum Füllen einer Karte

Es werden von oben nach unten zwei NPCs, zwei Umgebungs-Objekte, ein Spieler-Objekt und zwei Auslöser erstellt, welche alle ihren Platz in der Karte finden. Die NPCs sind Spielobjekte welche nicht vom Spieler gesteuert werden, sondern eine Künstliche Intelligenz besitzen, diese reagieren automatische auf das Verhalten des Spielers. Spielobjekte der Umgebung dienen nur der Atmosphäre und füllen die Karte mit einem individuellen Aussehen. Das Spieler-Objekt ist ein Objekt, dass der Spieler selber steuert. In Zukunft wird dieses Objekt nicht mehr durch ein Script erstellt, sondern wird automatisch erstellt. Der Grund dafür, dass es in Moment über ein Script fest erstellt ist, dass wir in Moment keine Struktur für das Speichern relevanter Spieler-Daten haben. Das Spieler-Objekt wird je nach Fortschritt des Spielers anders aussehen und anders ausgerüstet sein. Die Auslöser werden im nächsten Abschnitt genauer erklärt.

7.1.3. Trigger-System (J. W.)

Über dieses System werden Missionen, die Story- und Cinematic-Sequenzen und der Content des Spiels verwaltet und erstellt. Das System basiert komplett auf Skripte, was dafür sorgt, dass all die eben genannten Dinge einfach und schnell dem Spiel hinzugefügt werden. Selbst wenn man den Sourcecode nicht besitzt. Was dafür sorgt, dass das Spiel allein vom Content her komplett modifizierbar ist, jeder der wollte könnte eigene Missionen, eigene Sequenzen, im Endeffekt sein eigenes Spiel auf der Grundlage der Engine erstellen.

Das Trigger-System wurde, so wie viele meiner Teile, durch Warcraft3 und Starcraft2 inspiriert, die ein ähnlich funktionierendes System benutzen und in ihren Editoren anbieten, um das Spiel zu ändern und zu erweitern.

Das System funktioniert folgendermaßen: Ein Trigger ist ein Script, eine Lua-Datei, welche alle im Ordner „Scripts/Trigger“ liegen. Ein Trigger ist aber auch gleichzeitig eine Klasse im Code. Das Script besteht aus drei Parts: den Events, den Conditions und den Actions. Die Events sorgen dafür, dass der Trigger aufgerufen wird. Es gibt bisher zwei verschiedene Arten von Events, Unit- und TimeEvents, welche beide von der Basisklasse CTriggerEvent abgeleitet sind. Ersteres bezieht sich immer auf eine Entity und letzteres auf die Zeit. Dies heißt konkreter ein UnitEvent könnte sein, dass die Entity XYZ getötet wurde oder jemanden angreift. Ein TimeEvent wäre z.B. nach 20 Sekunden oder alle fünf Sekunden soll etwas geschehen. Die Conditions werden abgefragt, wenn eines der Events zugetroffen hat und sorgen dafür, wenn auch die Bedingungen zu treffen, dass die Actions ausgeführt werden. Eine Condition zu einem Event könnte z.B. sein, dass eine gewisse Einheit am Leben ist. Die Actions sind das Herzstück des Triggers, an diese Stelle wird gesprungen, wenn das angegebene Event zugetroffen hat und die erforderlichen Conditions richtig waren. Es werden dann alle beliebigen und gewünschten Aktionen vollzogen, z.B. dass mehrere Entities erstellt werden oder ein GUI-Element angezeigt wird.

Ein leerer Trigger sieht folgendermaßen aus:

```
--[Events]
function Events()
end
--[Events]

--[Actions]
function Actions()
end
--[Actions]

--[Conditions]
function Conditions()
    if true then
        Actions()
        return true;
    else
        return false;
    end
end
--[Conditions]
```

Abbildung 57 - Beispiel für einen leeren Trigger

Das Script ist aber nur der „externe“ Teil des Triggers. Damit der Trigger auch richtig funktioniert und ausgeführt wird, gibt es dazu verschiedene Klassen. Die Hauptklasse stellt hierbei CTrigger dar. Ein Objekt dieser Klasse wird immer zu einem Script erstellt, erst Klasse und Script zusammen bilden den Trigger.

In der Klasse sind die ID des Triggers, welche auch dem Namen des LUA-Scripts entspricht, eine bool-Variable, ob der Trigger aktiv ist und ein Vector von TriggerEvents gespeichert. Die Events sind der einzige Teil des Triggers, welcher im Code ausgeführt wird, Conditions und Actions werden direkt über das Script ausgeführt. Jeder Trigger besitzt eine beliebige Anzahl von Events, dabei gibt es bisher die zwei zuvor erwähnten Typen UnitEvent und TimeEvent. Die Klasse CWorld, besitzt eine Map von Triggern und geht in ihrer Update-Funktion jedes Mal die Trigger durch und updatet diese. Der Trigger updatet in seiner Update-Funktion wiederrum seine Events, welche ihm dann signalisieren, ob sie ausgelöst wurden. Z.B. ein UnitEvent vom Typ enDIES überprüft in seiner Update-Funktion ob die StructureStability der angegebenen Entity kleiner gleich Null ist und gibt dann dies an den Trigger zurück. Sollte nun ein Event ausgelöst worden sein, dann ruft der Trigger sein Script auf, in diesem werden die Conditions überprüft und dann ggf. die Actions ausgeführt.

Damit die Funktionen und Klassen des Spiels in den Scripts benutzt werden können, werden diese über LUA-Bind LUA hinzugefügt (näheres 4.4.1 LUA-Bind). Die Trigger sind damit ein mächtiges und vielseitig einsetzbares System, was die Wiederverwendbarkeit und die Modifizierbarkeit des Projekts immens erhöht.

7.1.4. Mission-System

Das Mission-System ist die Verwaltung der Mission (Quests/Aufgaben) des Spielers. Dem Spieler werden im Verlauf des Spiels immer wieder neue Aufgaben gestellt die dieser lösen muss, um das Spiel erfolgreich abzuschließen.
Es wird generell zwischen zwei Grundarten von Missionen unterschieden. Zum einen MainMissions, dies sind Hauptmissionen die der Spieler auf alle Fälle abschließen muss, um im Spiel weiter zu kommen bzw. um das Spiel abzuschließen. Zum anderen gibt es die OptionalMissions, dies sind optionale Missionen die nicht zwangsläufig abgeschlossen werden müssen. Doch diese Missionen können extra Vorteile bringen oder gewisse Punkte der Story weiter oder detailreicher, abseits der Hauptstory, erzählen. Des Weiteren sind die Missionen entweder bekannt/entdeckt bzw. werden im Verlauf des Spiels bekannt, oder sie sind versteckt und müssen erst durch gewisse Dinge entdeckt werden.

Das Mission-System wurde von dem Quest-System von Warcraft3/Starcraft2 inspiriert, denn in diesen beiden Spielen funktioniert es intern so ähnlich, denn es läuft auch vollständig über Scripts ab.

Es gibt hierzu eine Hauptklasse CMission. Diese Klasse stellt die gesamte Mission dar. Diese Klasse kann sowohl eine MainMission, als auch eine OptionalMission sein.

Eine Mission besitzt eine gewisse Anzahl von verschiedenen Aufgaben, die der Spieler lösen muss, sie werden Task genannt.
Es gibt verschiedene Arten von Tasks, welche alle eine eigene Klasse sind, die von CTask abgeleitet sind: KillTasks sind Aufgaben für die man verschiedene Ziele/Gegner eliminieren muss. ExploreTasks geben dem Spieler die Aufgabe gewisse Ziele/Regionen/Orte zu entdecken bzw. zu erreichen/besuchen. AttributeTaks sind Aufgaben, die sich auf den Zustand des Schiffs des Spielers beziehen, z.B. dass der Spieler im Verlauf der Mission nur einen gewissen Betrag von Schaden abbekommen darf usw. .

Zur Erfüllung einer Mission müssen ggf. mehrere Task gleichzeitig/nebeneinander und/oder unterschiedliche Tasks nacheinander erfüllt werden.
Das Erfüllen/Aktualisieren der Missionen und Tasks läuft komplett über das Trigger-System und nicht direkt über die Klassen des Mission-Systems ab. Dazu gibt es ein Script, welches die Missionen mit Task erstellt/initialisiert. Des Weiteren besitzt jede Mission mindestens ein Script in dem die Verwaltung/Aktualisierung des Missionsfortschritts verwaltet und gespeichert wird.

```
--[Actions]
function Actions()

    testMainMission = c_etyWorld:create_Mission("testMainMission", true, true, true);
    testMainMission:add_Task(c_etyWorld:create_KillTask("testKillTask", true, 5));

    testOptionalMission = c_etyWorld:create_Mission("testOptionalMission", false, true, true);
    testOptionalMission:add_Task(c_etyWorld:create_AttributeTask("testAttributeTask", true));
    testOptionalMission:get_AttributeTaskByID("testAttributeTask"):get_Attribute():
        add_UnsignedIntAttribute("uiStructureStability", 0);

end
--[Actions]
```

Abbildung 58 - Beispiel für die Missionsinitialisierung

Im Action-Part des Triggers „initMissions.lua“ werden die Missionen und ihre Tasks erstellt.

```
--[Events]
function Events()
    c_etyTrigger:add_TimeEvent( 0, 5.0, true);
end
--[Events]

--[Actions]
function Actions()

    print("YEHA! Mission completed!");

end
--[Actions]

--[Conditions]
function Conditions()

    playerAttribute =   c_etyEntitymanager:get_Entity("Player"):
        get_AttributeComponent("AttributeComponent").m_c_etyAttributes:
        get_UnsignedIntAttribute("uiStructureStability");

    taskAttribute   =   c_etyWorld:get_MissionByID("testOptionalMission"):
        get_AttributeTaskByID("testAttributeTask"):get_Attribute():
        get_UnsignedIntAttribute("uiStructureStability");

    if playerAttribute >= taskAttribute then
        Actions()
        return true;
    else
        print("YOU FAILED!");
        return false;
    end

end
--[Conditions]
```

Abbildung 59 - Beispieltrigger für eine Mission

Ein Beispiel für einen möglichen Missions-Aufbau. In diesem Fall wird nach fünf Sekunden überprüft, ob die „StructureStability“ des Raumschiffs größer gleich der vorgegebenen ist. Wenn nicht wird der Text „YOU FAILED!“ ausgegeben. Sollte der Spieler aber diese Vorgabe erfüllen, wird in den Actions-Part gesprungen und auch ein Text ausgegeben.

Das Mission-System basiert genau genommen ausschließlich aus Skripten, wodurch leicht neue Missionen dem Spiel hinzugefügt werden können.

7.2. Künstliche Intelligenz

Künstliche Intelligenz (kurz KI oder AI) in Computerspielen interessierte mich schon immer. Es ist faszinierend wie gut und auch wie schlecht eine solche KI agieren kann und umgesetzt wurde. Die KI-Programmierung ist auch eines der schwersten Bereiche der Spieleprogrammierung und ich würde mir wünschen später einmal in diesem Bereich zu arbeiten, da er mich am meisten reizt.

Eine KI zu programmieren ist sehr schwer, besonders wenn sie glaubwürdig und menschlich sein soll. In diesem Projekt ist zwar die KI ein Bestandteil meiner Aufgaben, doch er ist sehr kurz und einfach ausgefallen. Dies liegt zum einen an dem wirklich knappen Zeitrahmen und der Komplexität der KI-Programmierung. Es war mir bzw. uns von vorne herein klar, dass wir keine ausgeklügelte und umfangreiche KI in diesem Projekt haben bzw. programmieren werden. Des Weiteren ging es bei der Zielsetzung der KI eher darum, sich mit ihr ein wenig zu beschäftigen und mit KI-Programmierung vertraut zu machen. Die KI in diesem Projekt beschränkt sich auf eine ganz einfache Umsetzung und Funktionalität. Dies beschränkt sich erst einmal auf ein simples Patrouillen-Verhalten. Der aktuelle Stand ist nicht sehr ausgereift, was aber bei der Präsentation im Mai anders sein wird.

Jede NPC-Entity kann ein gewisses Behaviour (Verhalten) besitzen. Verhalten was sich auf den Bereich der Bewegung fällt, wird direkt in den Funktionen der MoveableComponent ausformuliert. Dort wird der Entity gesagt, welche Bewegungsroutine sie ausführen soll, z.B. das Verfolgen eines Gegners oder das Patrouillieren zwischen zwei Punkten.

Die Bewegung ist noch sehr wenig ausgereift und wird erst später in einer präsentierbaren Form vorliegen. Bisher gibt es zwei Routinen, zum einen, dass eine Entity den Spieler verfolgt und dabei immer einen gewissen Abstand zu ihm einhält, sowie versucht sich um ihn herum zu drehen. Zum anderen, dass die Entity zwischen zwei Punkten hin und her fliegt. Das Umsetzen dieser beiden Verhalten erübrigte sich als nicht sehr einfach. Anfangs versuchte ich dieselben Bewegungsmöglichkeiten wie die des Spielers zu benutzen. Doch es stellte sich heraus, dass diese sehr unpräzise sind und für die KI weniger geeignet sind und mussten daher teilweise modifiziert werden.

7.3. Mathematisches und Physikalisches (J. S.)

7.3.1. Kollisionserkennung

Die Kollisionserkennung ist einer der Baustellen unseres Spieles. Bisher besteht nur eine Art von Kollisionserkennung, die Kreis-Kollisionserkennung. Sie ist, neben den Erkennungen mit Hilfe von Rechtecken oder Pixeln, die leichteste Kollisionserkennung und auch die ungenauste, wenn es sich nicht um kreisförmige Objekte handelt. Jedenfalls besteht zumindest eine Kollisionserkennung, aber es besteht noch keine Reaktion auf Kollision. Das bedeutet dass die Kollision zwar erkannt wird, aber die Spielobjekte „durch“ andere Spielobjekte fliegen können. Bei der Kreis-Kollisionserkennung wird um jedes Spielobjekt, wie in der folgenden Abbildung, ein Kreis gezogen. „Schneiden“ sich die Kreise zweier Spielobjekte so entsteht eine Kollision.

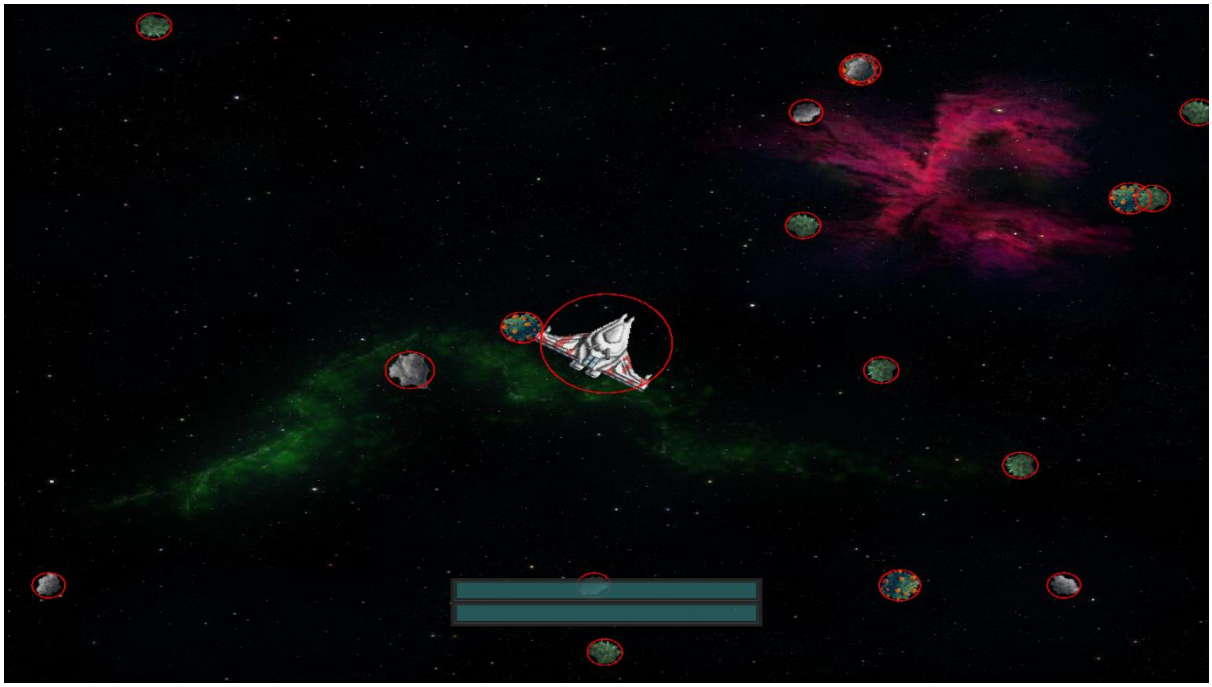


Abbildung 60 - Kreis-Kollision im Spiel

Für die Überprüfung zweier Spielobjekte auf eine Kreis-Kollision, wird die Position der beiden Spielobjekte benötigt und die Radien der ihn umgebenen Kreise. Der Programmiercode für diese Berechnung sieht dann wie folgt aus:

```
const sf::Vector2f c_sfVectorDifference = c_sfVectorCenterPosition1 - c_sfVectorCenterPosition2;

const float fDistance = ( c_sfVectorDifference.x * c_sfVectorDifference.x ) + ( c_sfVectorDifference.y * c_sfVectorDifference.y );

return fDistance <= ( fRadius1 + fRadius2 )*( fRadius1 + fRadius2 );
```

Es wird die Distanz der zwei Objekte berechnet. Ist die Distanz zwischen den zwei Objekten kleiner oder gleich der Summe beider Radien, so besteht eine Kollision zwischen diesen zwei Spielobjekten.

7.3.2. Bewegung

Die Bewegung ist mit Hilfe einer Formel aus der Physik leicht zu berechnen. Wir benötigen dafür die vergangene Zeit nach jeder Aktualisierung und die Geschwindigkeit und ebenso die Beschleunigung jedes Spielobjektes. Wird das Spielobjekt durch eine Interaktion beschleunigt wird bei jeder Aktualisierung erst die neue Geschwindigkeit berechnet durch die folgende Formel:

```
c_sfVectorVelocity += c_sfVectorDirection * fAcceleration * uiFrameTime / 1000
```

- uiFrametime ist die vergangene Zeit in Millisekunden seit der letzten Aktualisierung
- c_sfVectorDirection ist die Richtung in der das Spielobjekt zeigt
- fAcceleration ist die konstante Beschleunigung des Objektes in Abhängigkeit von der Masse des Spielobjekte

Die konstante Beschleunigung wird nach der folgenden Formel berechnet:

```
fAcceleration = fThrust / fMass
```

- fThrust ist der Schub in Newton
- fMass ist das Gewicht des Spielobjektes in kg

Nach diesen Berechnungen wird dann die Position aktualisiert:

```
c_sfVectorPosition += c_sfVectorVelocity * uiFrameTime / 1000
```

- c_sfVectorPosition ist die Position des Spielobjektes
- c_sfVectorVelocity ist die Geschwindigkeit des Spielobjektes

Nach diesen Berechnungen funktioniert unsere Bewegung einwandfrei. Die vergangene Zeit wird in den Berechnungen durch 1000 dividiert, weil die vergangene Zeit in Millisekunden angegeben ist und wir die Werte jedoch in Sekunden haben wollen.

8. Spielinhalte

8.1. Die Menüs (J. W.)

Es gibt im Spiel eine Vielzahl von unterschiedlichen Menüs. Zum einen das Hauptmenü, das der Benutzer betritt, wenn er das Spiel startet. Das Menü bietet Funktionalitäten und Möglichkeiten die nicht direkt mit dem Spiel zu tun haben. Es besteht aus mehreren Untermenüs, die sich um die Verwaltung der einzelnen Bereiche kümmern. Dann gibt es aber noch Menüs die direkt mit dem Spiel verbunden sind, wie das Lager bzw. der Shop des Spielers, das Missions-Menü usw. Alle Menüs sind über das Dialogsystem aufgebaut. Da diese Menüs sich teilweise stark unterscheiden stehen sie teilweise in eigenen States. Was die Übersichtlichkeit immens verbessert.

8.1.1. Das Hauptmenü

Wie bereits erwähnt ist das Hauptmenü das Menü, welches der Benutzer beim Starten des Spiels betritt. Es ist eine Sammlung von mehreren Menüs, dem Optionsmenü, dem Profilmenü, den Credits und dem ExitMenü. Das Hauptmenü an sich besteht aber nur aus drei Dialogen, dem Background, dem Pad, über das sich in die verschiedenen Menüs navigieren lässt, und das ExitMenü. Das Options- und Profilmenü sowie die Credits sind in eigene States ausgelagert, sind aber nur über das Hauptmenü erreichbar.

Abbildung 61 - Das Pad des Hauptmenüs

Das Kernstück des Menüs ist das Pad, es besteht aus fünf Buttons und ebenfalls fünf PictureControls. Über dieses Pad bzw. die Buttons kann man in die unterschiedlichen Menüs navigieren und das Spiel starten. Die PictureControls sind rein für das Optische, da je nachdem welcher Button angewählt ist, sie die Farbe der Lampe unten links ändert.

Über den Button „Start“ lässt sich das Spiel starten, sofern ein Profil ausgewählt wurde, wenn nicht bleibt der Button ausgegraut (siehe Abbildung). Über die Buttons „Options“ und „Profiles“ können die jeweiligen Untermenüs geöffnet werden. Der Button „Credits“ zeigt die Credits (Mitwirkenden) an, doch aus Zeit- und Prioritätsgründen wurden diese noch nicht programmiert, da sie erst einmal unwichtig sind. Über den „Exit“ Button kann man das Spiel beenden. Optional kann man auch die Escape-Taste drücken, dann öffnet sich das Exit-Menü.

Abbildung 62 - Das ExitMenü

Dieses Menü ist ein eigener Dialog und besteht aus einem Label, welches den Spieler darauf hinweist, dass er im Begriff ist, das Spiel zu beenden, und zwei Buttons zum „Beenden“ oder „Abbrechen“.

8.1.2. Das Profilmenü

Das Profilmenü erscheint oben links am Bildschirmrand, wenn der entsprechende Button auf dem Pad angeklickt wurde. Dieses Menü besteht aus zwei Labeln. Eines zeigt den Namen des Menüs an und ein weiteres das gerade ausgewählte Profil. Dieses ist auch im Hauptmenü unten links zu sehen. Das Herzstück des Menüs ist eine ListBox in der alle verfügbaren Profile gespeichert sind. Darüber hinaus befinden sich zwei TextButtons zum Löschen und Erstellen eines Profils in diesem Menü. Dieses und alle weiteren Untermenüs des Hauptmenüs besitzen zwei markante Buttons unten rechts. Zum einen ein Pfeil, um das Menü zu verlassen und eine Diskette um Änderungen zu speichern und zu übernehmen.

Abbildung 63 - Das Profilmenü

In diesem Menü kann nun ein neues Profil über den passenden Button erstellt werden (die Eingabe erfolgt über ein EditControl). Des Weiteren kann ein Profil ausgewählt werden, mit dem man spielen möchte. Das ausgewählte Profil wird in dem Menü und im Hauptmenü angezeigt. Letztlich besteht dieses Menü nur aus einem Dialog, ist aber dennoch in einem eigenen State ausgelagert.

8.1.3. Das Optionsmenü

Das Optionsmenü erscheint wie das Profilmenü oben links am Bildschirmrand und ist auch in einem eigenen State ausgelagert. In diesem Menü lässt sich über eine ComboBox die Sprache des Spiels einstellen (Deutsch, Englisch und Russisch). Darüber hinaus besitzt auch dieses Menü, genau wie die beiden folgenden, einen Titel in Form eines Labels.
Das Optionsmenü spaltet sich in zwei weitere Untermenüs auf, in die über Textbuttons navigiert werden kann. Das AudioOptionsMenü und das GraphicOptionsmenü, beide Menüs sind auch wieder in eigene States ausgelagert.

Abbildung 64 - Das Optionsmenü

8.1.4. Das AudioOptionsmenü

In diesem Menü kann man die Soundeinstellungen des Spiels vornehmen. Dazu besitzt dieses Menü drei SlideControls, um die Lautstärke von Musik und Sounds sowie die Masterlautstärke einzustellen, welche sich im Bereich von 0 bis 100 Prozent einstellen lassen. Des Weiteren besitzt dieses Menü wieder einen Titel und die beiden Buttons, welche sich in allen drei Menüs wieder finden.

Abbildung 65 - Das AudioOptionsmenü

8.1.5. Das GraphicOptionsmenü

In diesem Menü lassen sich verschiedene Einstellungen zur Grafik vornehmen. Über Checkboxen lassen sich einstellen, ob der Windows- oder der Spielcursor benutzt werden soll, ob Vertikal-Synchronisierung ein oder ausgeschaltet sein soll und ob sich das Spiel im Vollbild- oder Fenstermodus befinden soll. Die letzte Einstellung die vorgenommen werden kann, ist das Festlegen der gewünschten Auflösung. Alle verfügbaren Auflösungen werden in einer ComboBox aufgelistet und die gewünschte Auflösung kann mit der ComboBox ausgewählt werden.

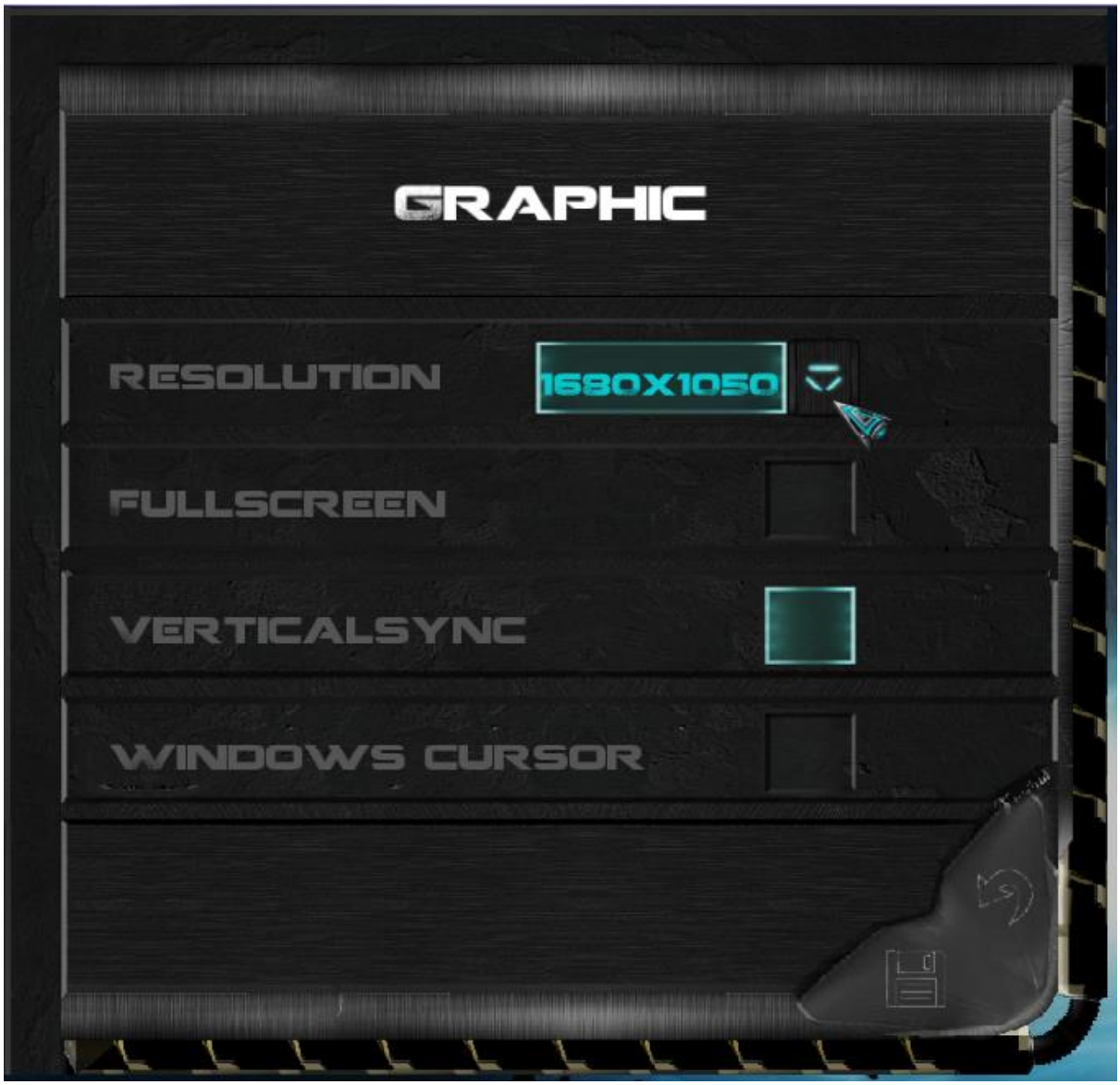


Abbildung 66 - GraphicOptionsmenü

8.1.6. Die SpaceStation

Die SpaceStation ist der Dreh- und Angelpunkt des Spiels. Sie ist ein eigener State und besteht aus vielen verschiedenen Menüs. In ihr befinden sich Möglichkeiten neue Missionen anzunehmen, sein Schiff im Hangar zu verändern, dass Weltall zu betreten usw. Sie gehört direkt zum Game-Content und ist daher noch weitgehend leer. Die Bilder der Hintergründe der SpaceStation und des Hangars wurden nicht von unserem Grafiker angefertigt. Skizzen, sowie Entwürfe finden sich im Anhang ab 11.2.

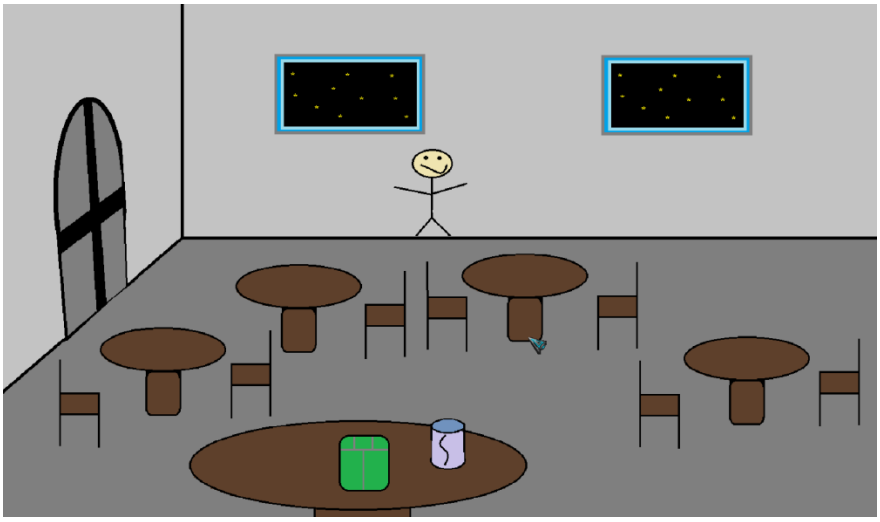


Abbildung 67 - Die SpaceStation

8.1.7. Der Hangar

Im Hangar kann man sein Lager bzw. den Shop betreten in dem man sein Schiff modifizieren kann, neue Ausrüstung kaufen kann usw. Des Weiteren lässt sich über den Hangar das Weltall betreten. Da auch dieser zum Game-Content gehört ist er weites gehend leer.

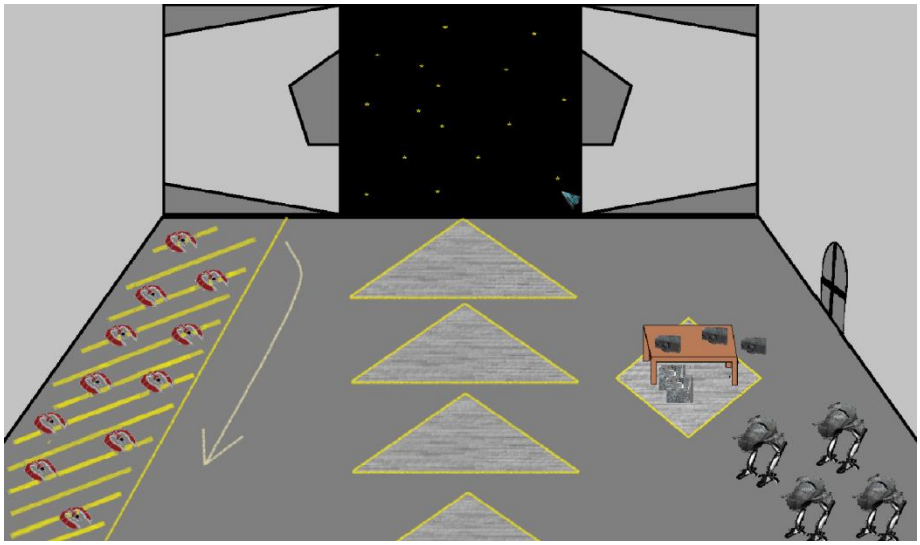


Abbildung 68 - Der Hangar

9. Projekt Fazit

9.1. Projekt Fazit von Johannes Winkler

Als wir das Projekt anfingen, waren mir die Größe und die Menge an Arbeit, die nun darin steckt, gar nicht bewusst gewesen. Was vor allem daran lag, dass ich noch nie ein Projekt in dieser Größenordnung bearbeitet habe. Wenn ich mir das Projekt nun ansehe komme ich zu dem Ergebnis, dass es sich definitiv gelohnt hat, dieses umzusetzen und die Fülle an Zeit zu investieren. Besonders deswegen, weil ich eine Menge gelernt und meine Fähigkeiten weiterentwickelt habe. Des Weiteren besitze ich damit eine Referenz in diesem Fach. Darüber hinaus wurden meine Erwartungen an dieses Projekt erfüllt.
Mein Ziel war es, eine wiederverwendbare Benutzeroberfläche, ein damit verbundenes Eingabe-System und ein Quest- bzw. Missions-System verknüpft mit einem Auslöser-System zu schreiben. Ausserdem wollte ich mich ein wenig mit KI beschäftigen und mich an dieser Aufgabe versuchen. Ich denke, dass mir die Umsetzung meiner Ziele gelungen ist.

Wenn ich den Hauptbestandteil meiner Arbeit betrachte, das Dialog-System, bin ich mit seiner Umsetzung und Funktionalität zufrieden. Es ist umfangreich, bietet viele Möglichkeiten zur Gestaltung und Umsetzung von Anwendungen und es ist wiederverwendbar. Natürlich gibt es noch einige Stellen an denen es sich erweitern und verbessern lässt, aber ich habe zuvor noch nie so etwas in diesem Umfang und in C++ geschrieben.
Die Umsetzung des Event-Systems gefällt mir besonders gut, da sie meiner Meinung nach sehr elegant und übersichtlich geworden ist.
Das Trigger-System stellt zusammen mit dem Mission-System ein mächtiges Werkzeug dar und erhöht die Möglichkeit zur Modifizierung immens. Vielleicht ist es so nicht die beste Lösung, da ich mich bei dieser Thematik nicht sehr tiefgreifend mit solchen Systemen in professionellen Spielen befasst habe, aber ich bin zufrieden mit meiner Umsetzung.
Ein wenig schade ist, dass die KI noch nicht wirklich ausgereift ist, bzw. noch völlig in den Kinderschuhen steckt. Ich werde sie bis zur Präsentation noch etwas verbessern und erweitern, aber ich kann mich ihr leider nicht in dem Umfang widmen, wie ich es gerne getan hätte. Natürlich war mir dies von Anfang an klar, da KI-Programmierung nicht der Hauptbestandteil dieses Projekts ist, sondern nur ein Teilbereich. Dennoch gehört KI-Programmierung zu den Dingen, die mich am meisten interessieren.

Ich denke, in diesem Projekt steckt noch viel mehr Potenzial, als durch diese Dokumentation ersichtlich wird, da es noch kein fertiges Spiel ist. Interessant wäre es später einmal, das komplett fertige Spiel mit dem Stand, der in dieser Dokumentation beschrieben wird, zu vergleichen.

Abschließend kann ich sagen, dass ich mit dem Gesamtergebnis zufrieden bin. Meiner Meinung nach haben wir mit diesem Projekt und dieser Dokumentation einen guten Einblick in die Entwicklung eines 2D-Spiels mit einer selbstentwickelten Engine geben.

9.2. Projekt Fazit von Julien Saevecke

Das Projekt war bei der Planung noch ein leichtes. Als wir dann versucht haben die benötigten Komponenten zusammen zu programmieren, ist mir klar geworden, dass es doch mehr Arbeit ist als wir zuerst dachten. Es hat mir aber sehr viel Spaß gemacht ein solches Projekt als Team zu starten und zu beenden und es hat meine Kommunikationsfähigkeiten ebenso wie meine Fähigkeiten in Programmierung erweitert.

Ich bin auf das Entity-System und das Gamestate-System am meisten stolz, denn es sind die zwei Systeme, an denen ich die meiste Zeit und Gedanken verloren habe. Sie sind zu dem auch in anderen Projekten wiederverwendbar.

Das Entity-System ist jedoch noch nicht ganz ausgereift und ich bin mit der Umsetzung noch nicht ganz zufrieden. Die Zusammenarbeit des Systems ist zwar solide, aber umso mehr das Entity-System ausgenutzt wird, fällt einen schon auf, dass das Zusammenspiel mancher Komponenten nicht ganz sauber programmiert worden ist.

Das Gamestate-System ist jedoch für mich sehr zufriedenstellend. Es kann ohne Probleme überall benutzt werden und erweiterbar ist es zu dem auch noch. In Zukunft will ich das System in der Richtung her ändern, dass man Spielzustände über Skripte erstellen und initialisieren kann.

Während diese Zeit habe ich einen großen Einblick in die Spieleprogrammierung gewonnen, auch wie anstrengend und zeitaufwendig gute Planung und Implementierung dessen sein kann. Aus diesem Projekt wurde mein Wunsch später in eine Spielebranche einzutreten nur noch verstärkt.

Um alles zusammenzufassen würde ich sagen, dass das Projekt zwar nicht so weit gekommen ist wie wir dachten, aber wir alle Punkte zufriedenstellend erfüllt haben, die wir mindestens erreicht haben wollten. Ich bin zufrieden.

10. Quellenverzeichnis und Literaturverzeichnis

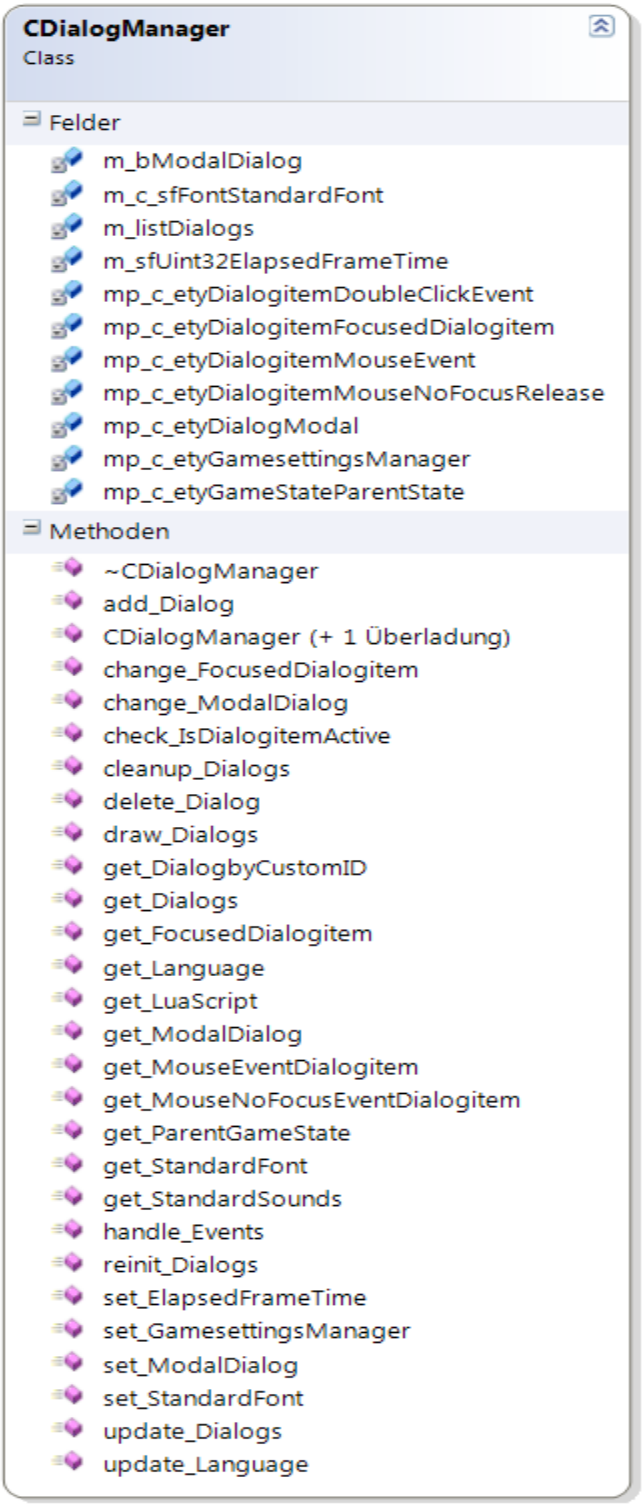
Seiten auf denen wir bei Problemstellungen nach Rat gesucht haben waren folgende:

- Fachinformatiker IT-Berufe Community, http://www.fachinformatiker.de
  - Letzter Zugriff: 13.03.2013
- cplusplus, http://www.cplusplus.com/
  - Letzter Zugriff: 22.02.2013
- Simple And Fast Multimedia Library, http://www.sfml-dev.org/
  - Letzter Zugriff: 05.03.2013
- Boost C++ Libraries, http://www.boost.org/
  - Letzter Zugriff: 15.01.2013
- Lua the programming language, http://www.lua.org/
  - Letzter Zugriff: 15.01.2013
- Rasterbar Software, http://www.rasterbar.com/products/luabind.html
  - Letzter Zugriff: 15.01.2013
- http://www.gamedev.net/topic/582048-dynamic-game-state-system/
  - Letzter Zugriff: 06.09.2012
- Artemis, http://gamadu.com/artemis/tutorial.html
  - Letzter Zugriff: 1.01.2013
- Heiko Kalista, 2009, C++ für Spieleprogrammierer, Carl Hanser Verlag GmbH & CO. KG; Auflage: 3

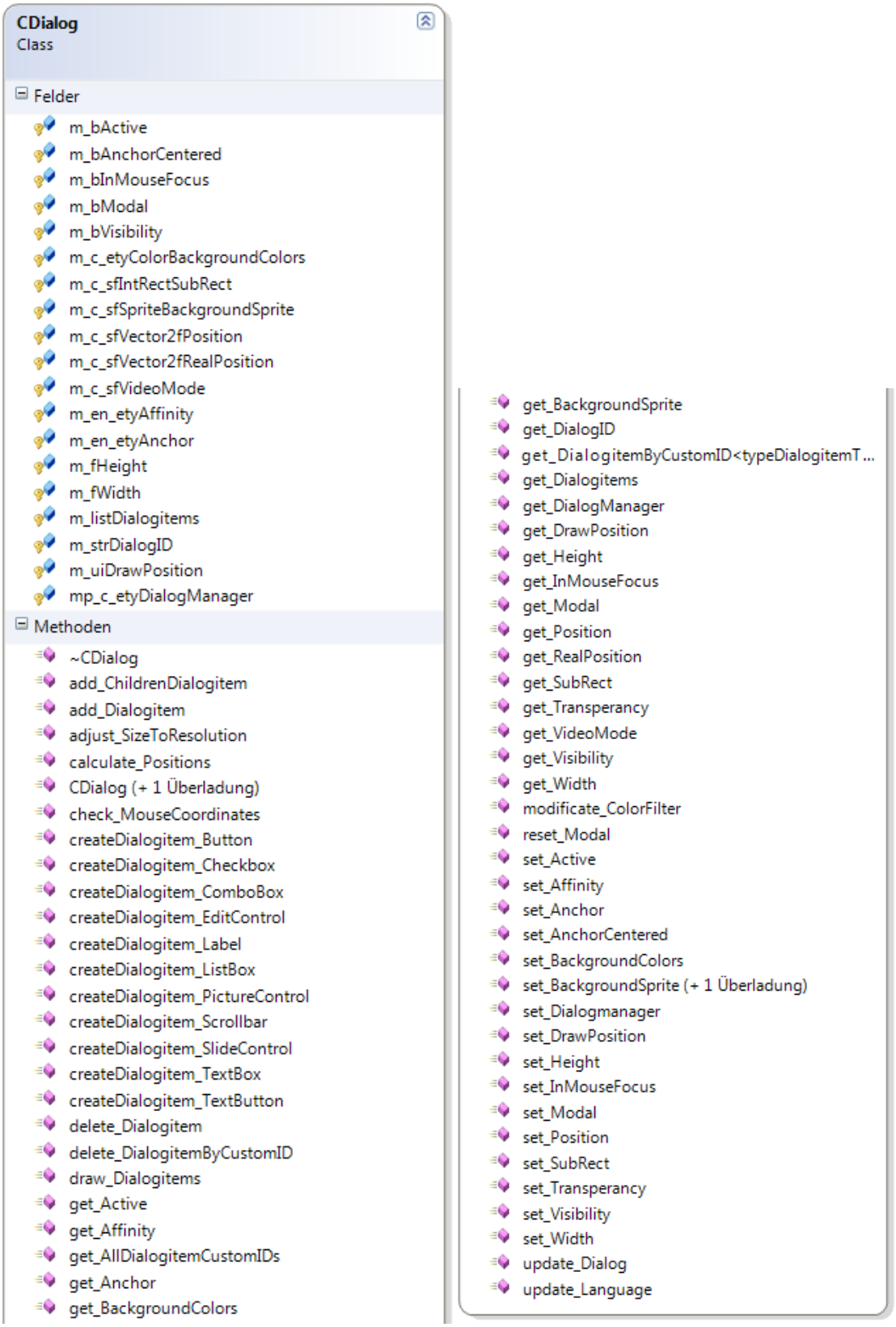
11. Anhang

11.1. Klassen-Diagramme

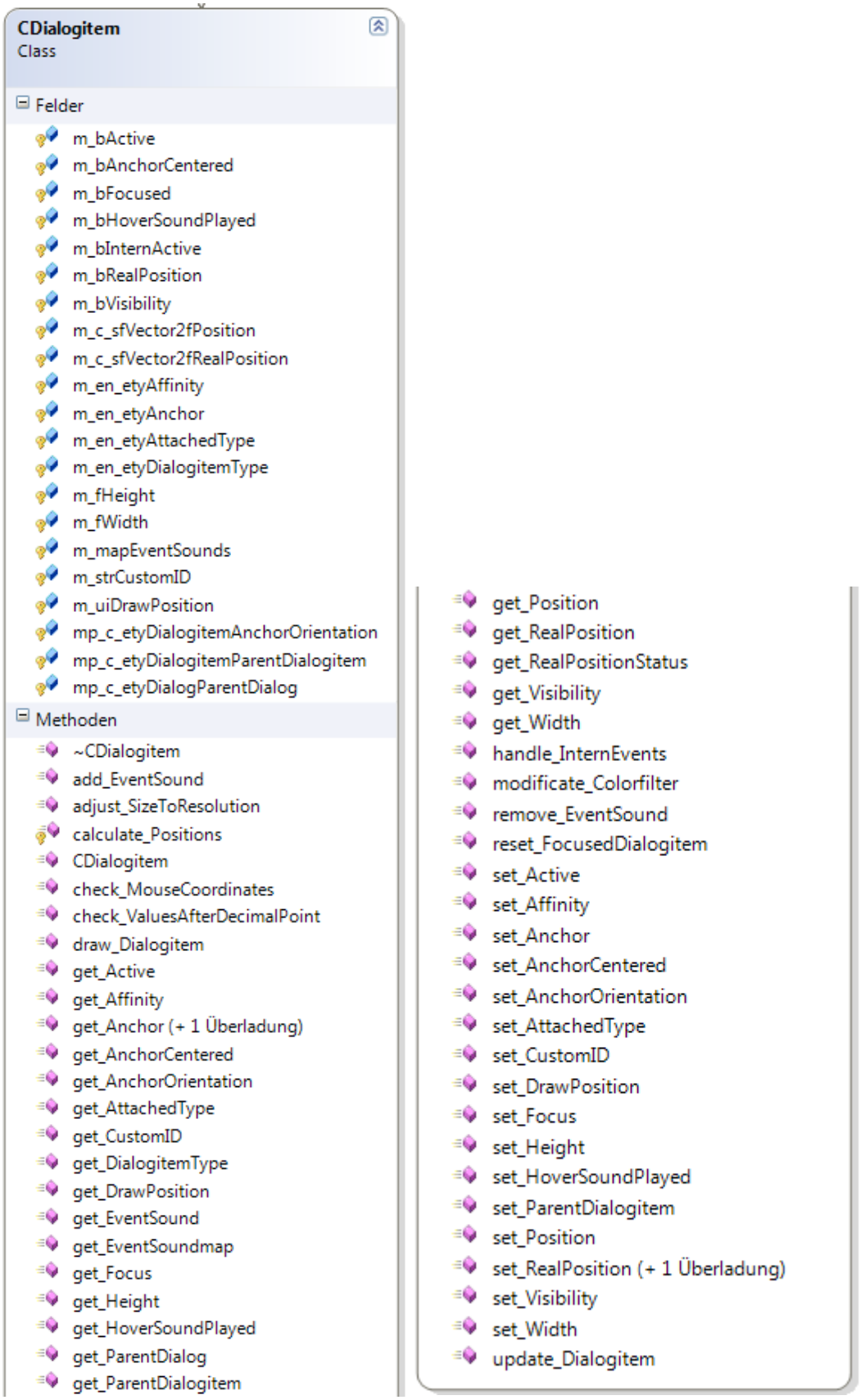
11.1.1. CDialogManager



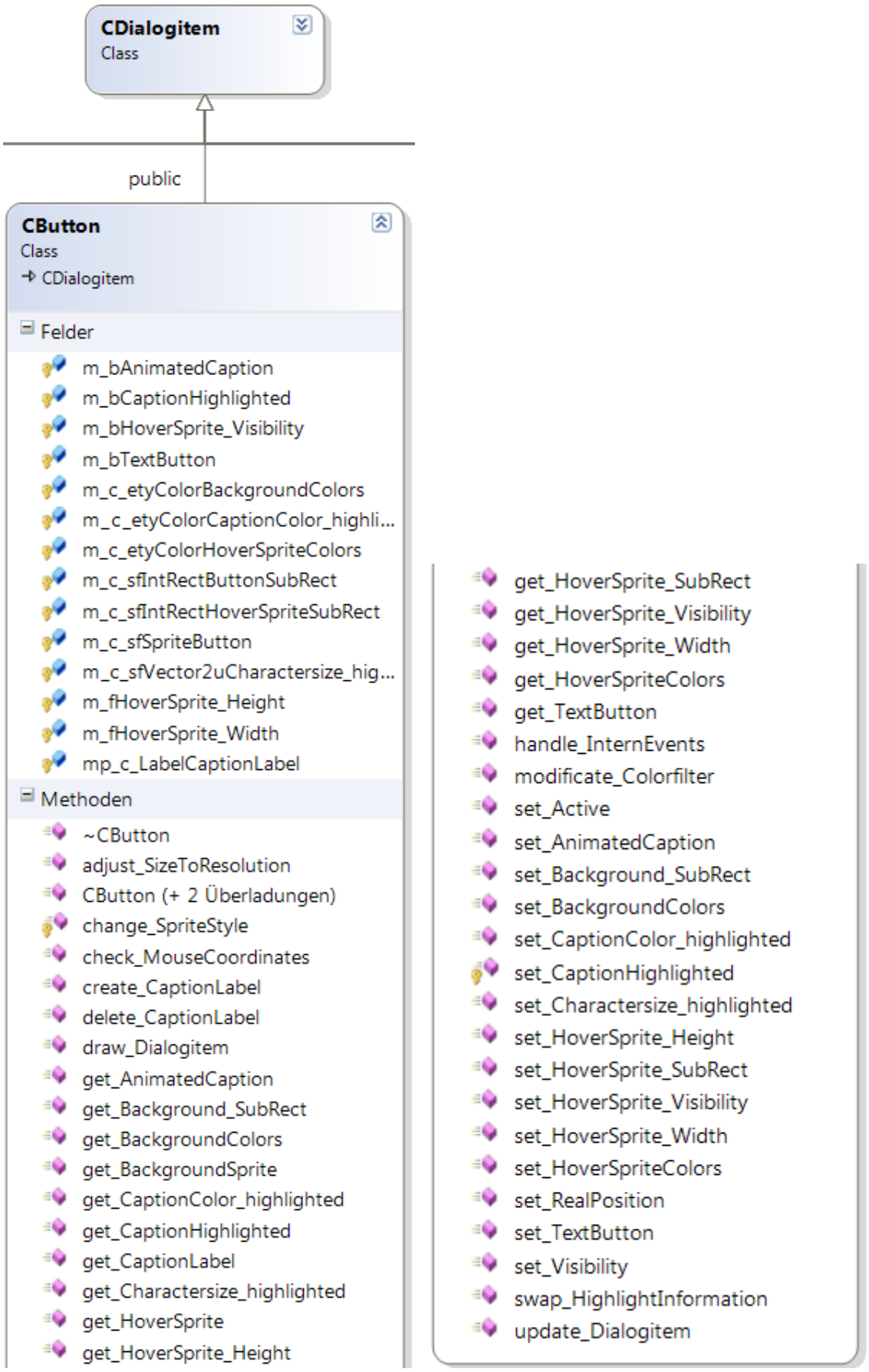
11.1.2. CDialog



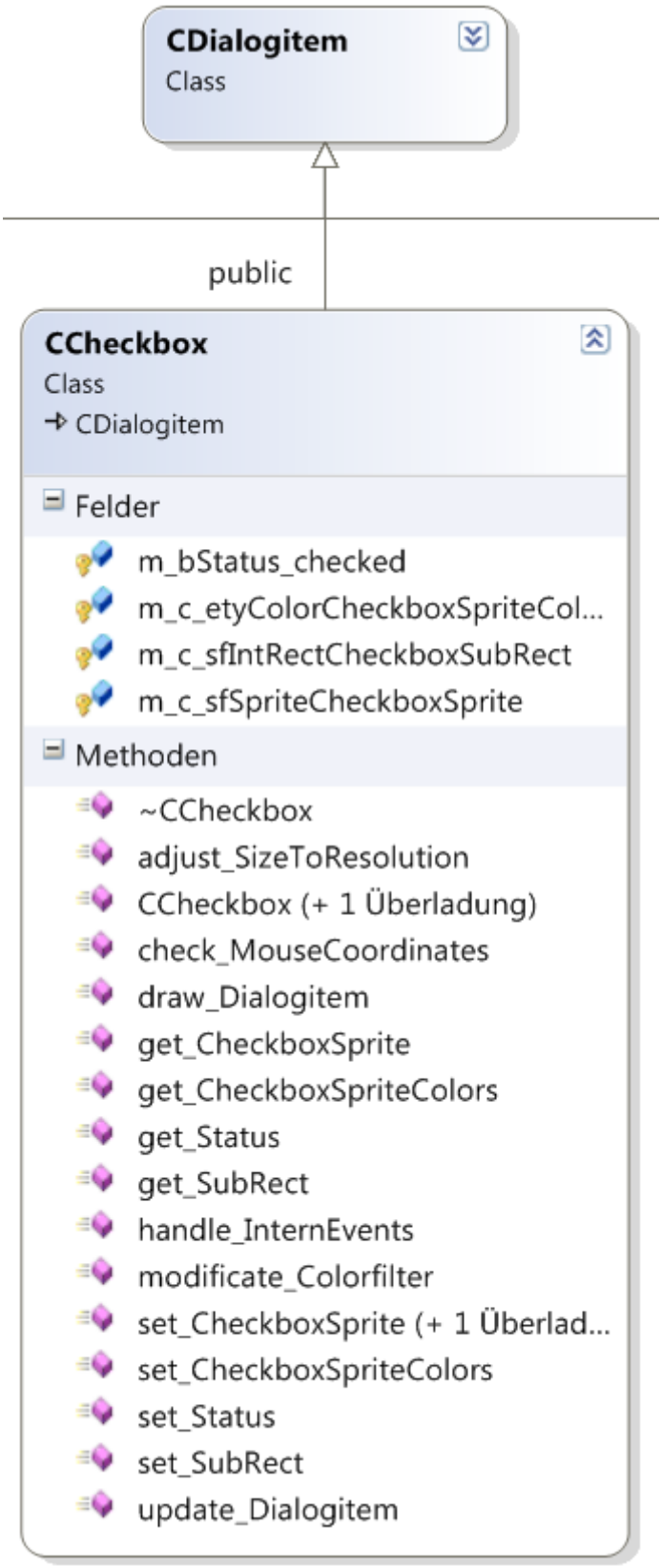
11.1.3. CDialogitem



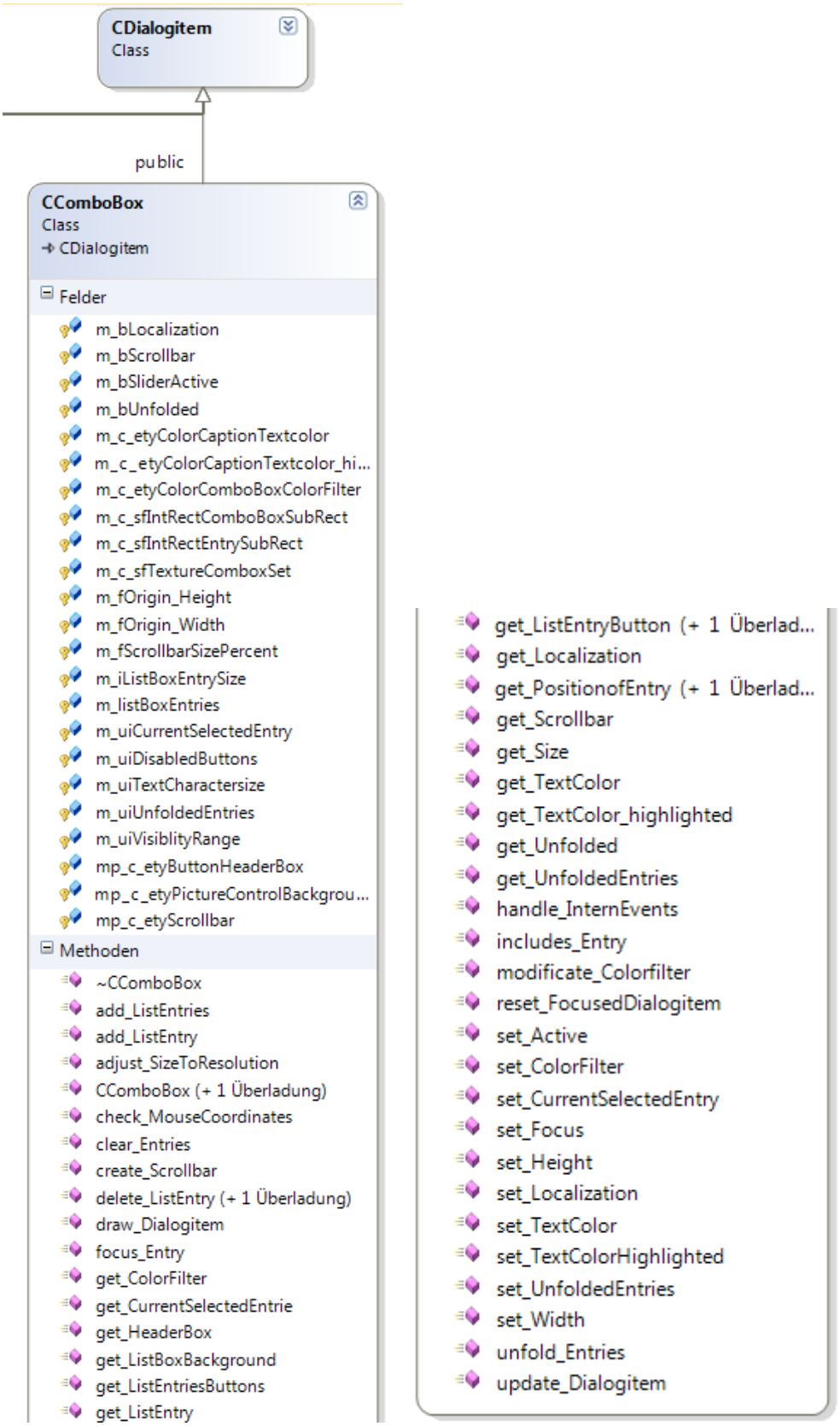
11.1.4. CButton



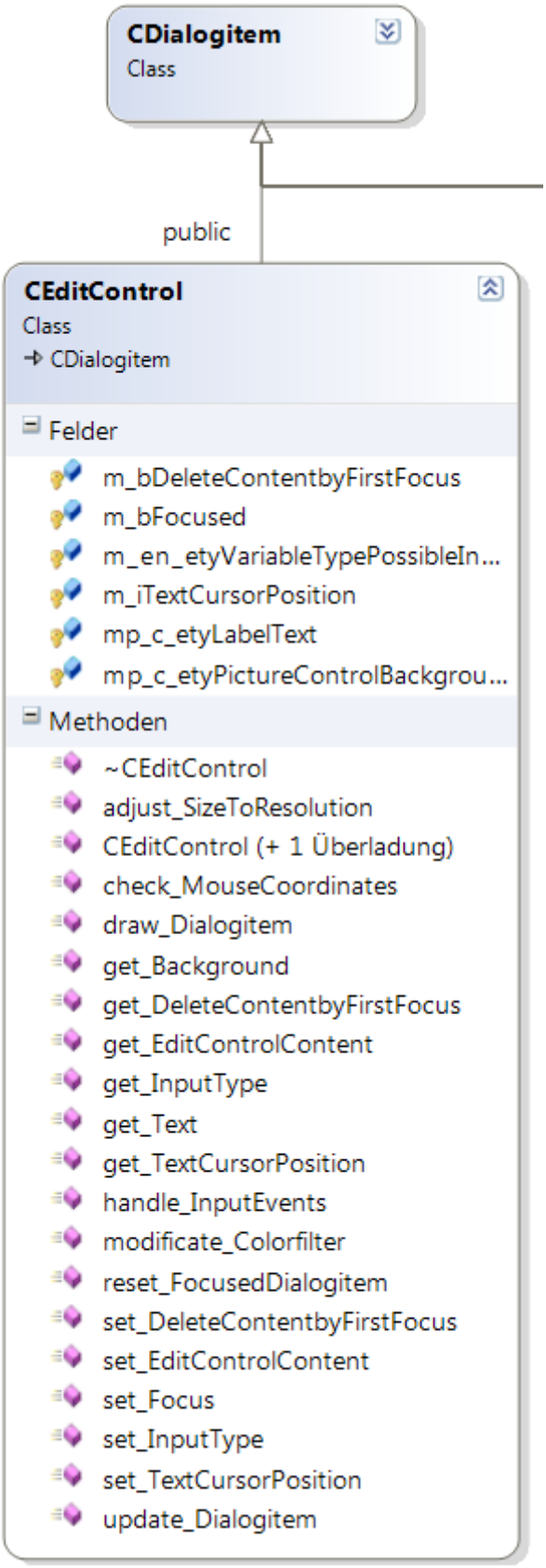
11.1.5. CCheckbox



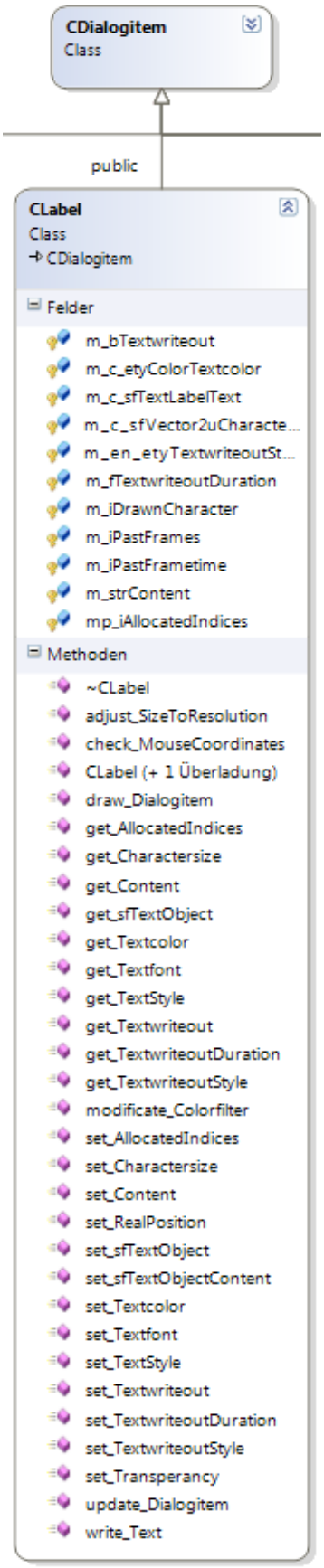
11.1.6. CComboBox



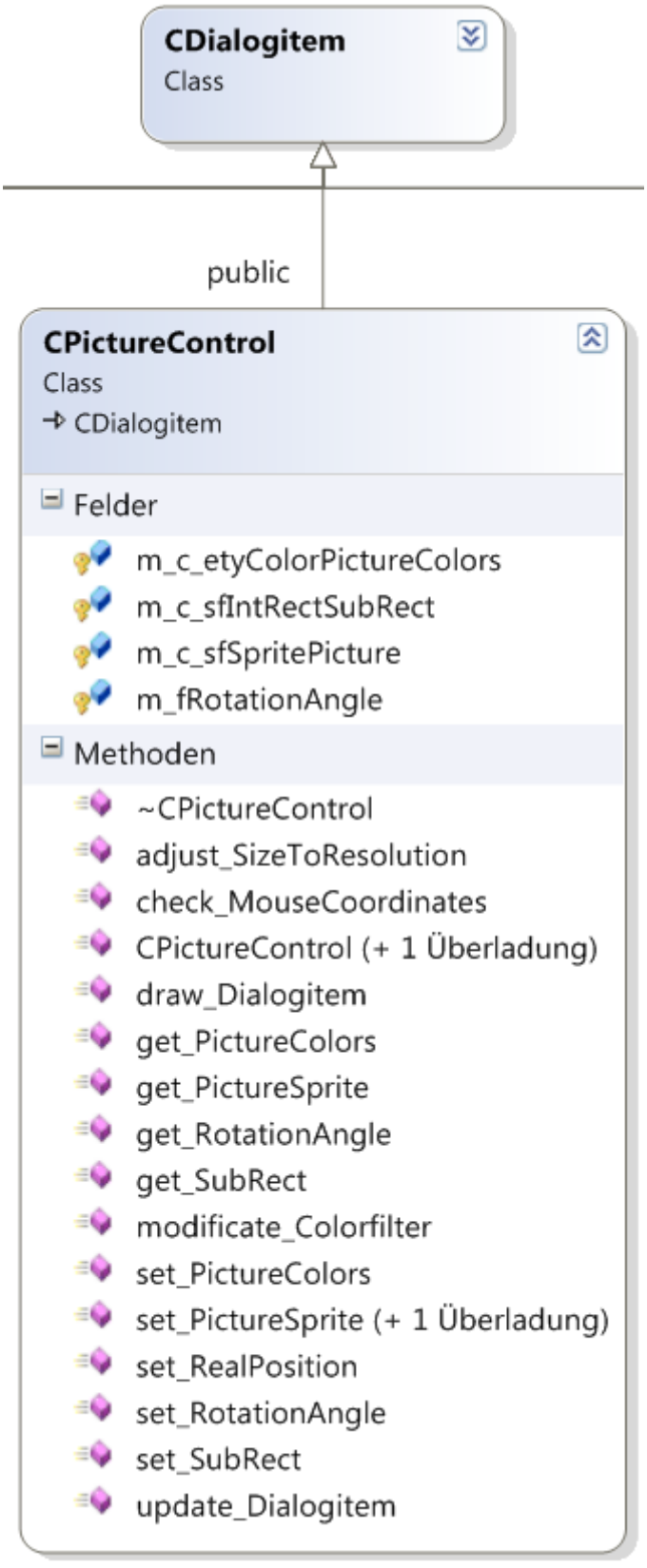
11.1.7. CEditControl



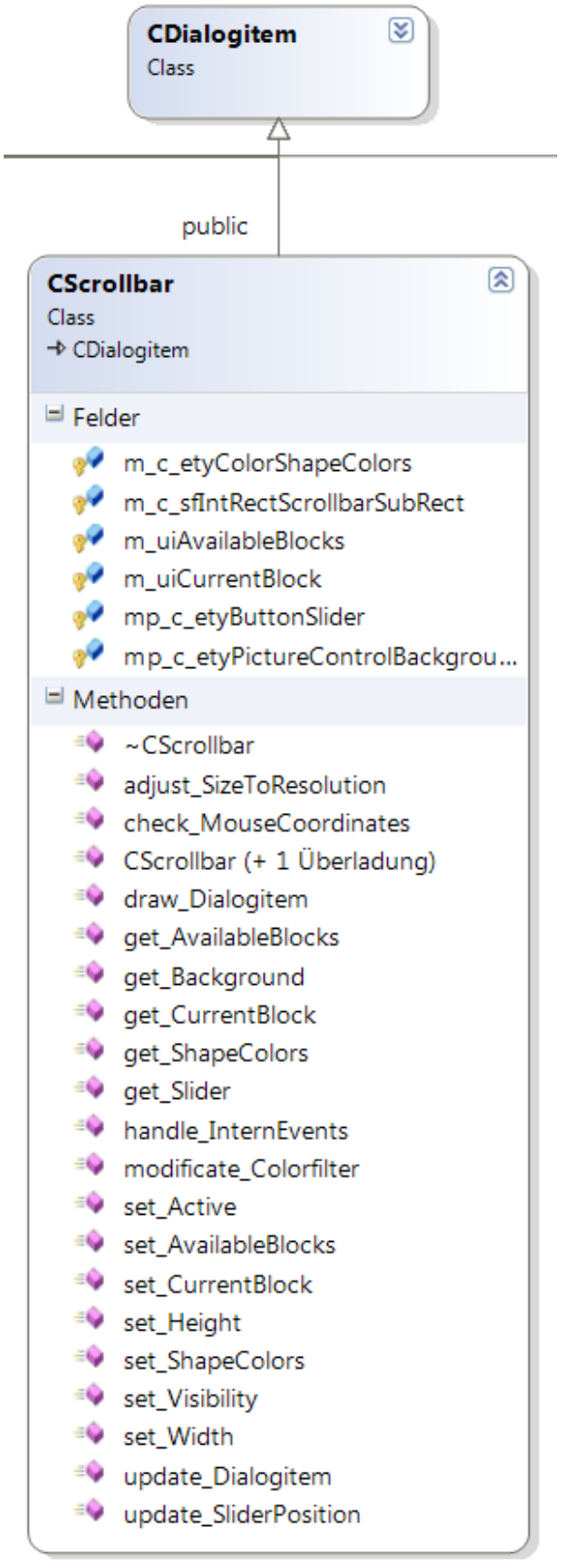
11.1.8. CLabel


CDialogitem
Class
public
CLabel
Class
CDialogitem
Felder
m_bTextwriteout
m_c_etyColorTextcolor
m_c_sfTextLabelText
m_c_sfVector2uCharacte...
m_en_etyTextwriteoutSt...
m_fTextwriteoutDuration
m_iDrawnCharacter
m_iPastFrames
m_iPastFrametime
m_strContent
mp_iAllocatedIndices
Methoden
~CLabel
adjust_SizeToResolution
check_MouseCoordinates
CLabel (+ 1 Überladung)
draw_Dialogitem
get_AllocatedIndices
get_Charactersize
get_Content
get_sfTextObject
get_Textcolor
get_Textfont
get_TextStyle
get_Textwriteout
get_TextwriteoutDuration
get_TextwriteoutStyle
modificate_Colorfilter
set_AllocatedIndices
set_Charactersize
set_Content
set_RealPosition
set_sfTextObject
set_sfTextObjectContent
set_Textcolor
set_Textfont
set_TextStyle
set_Textwriteout
set_TextwriteoutDuration
set_TextwriteoutStyle
set_Transperancy
update_Dialogitem
write_Text

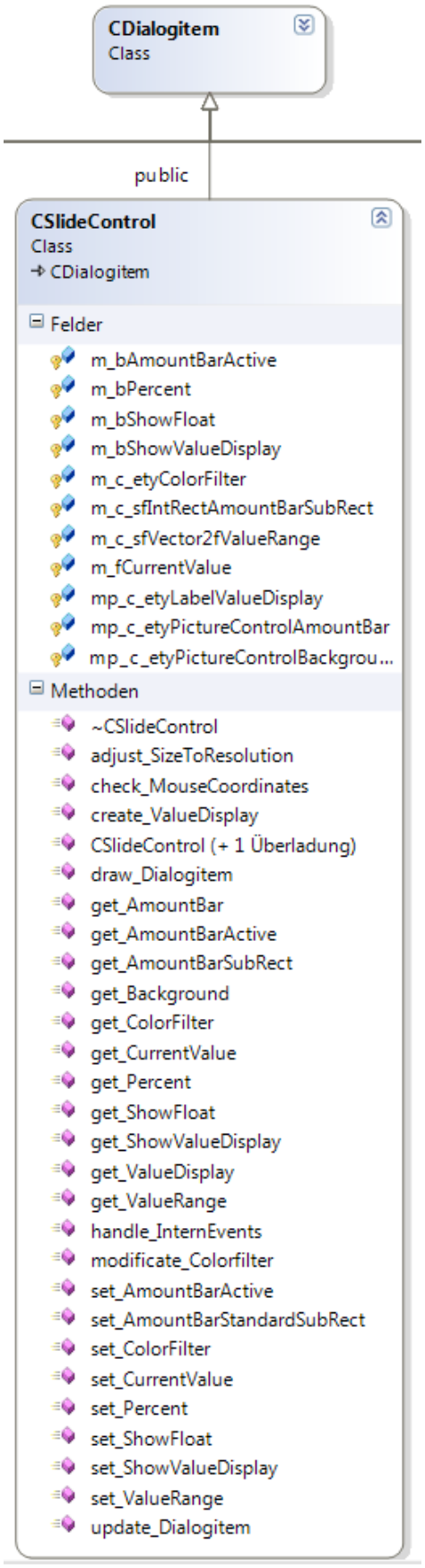
11.1.9. CPictureControl



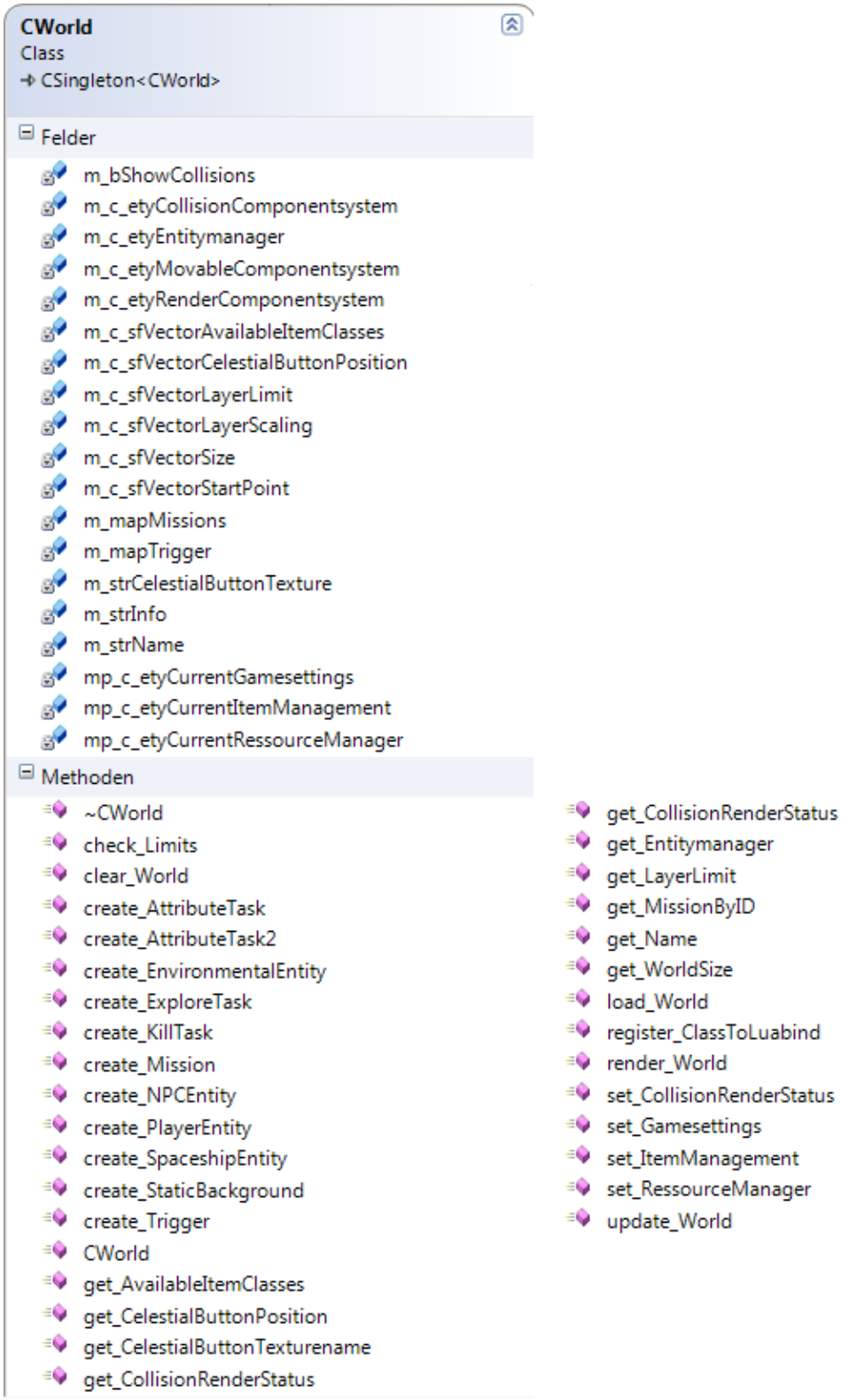
11.1.10. CScrollbar



11.1.11. CSlideControl

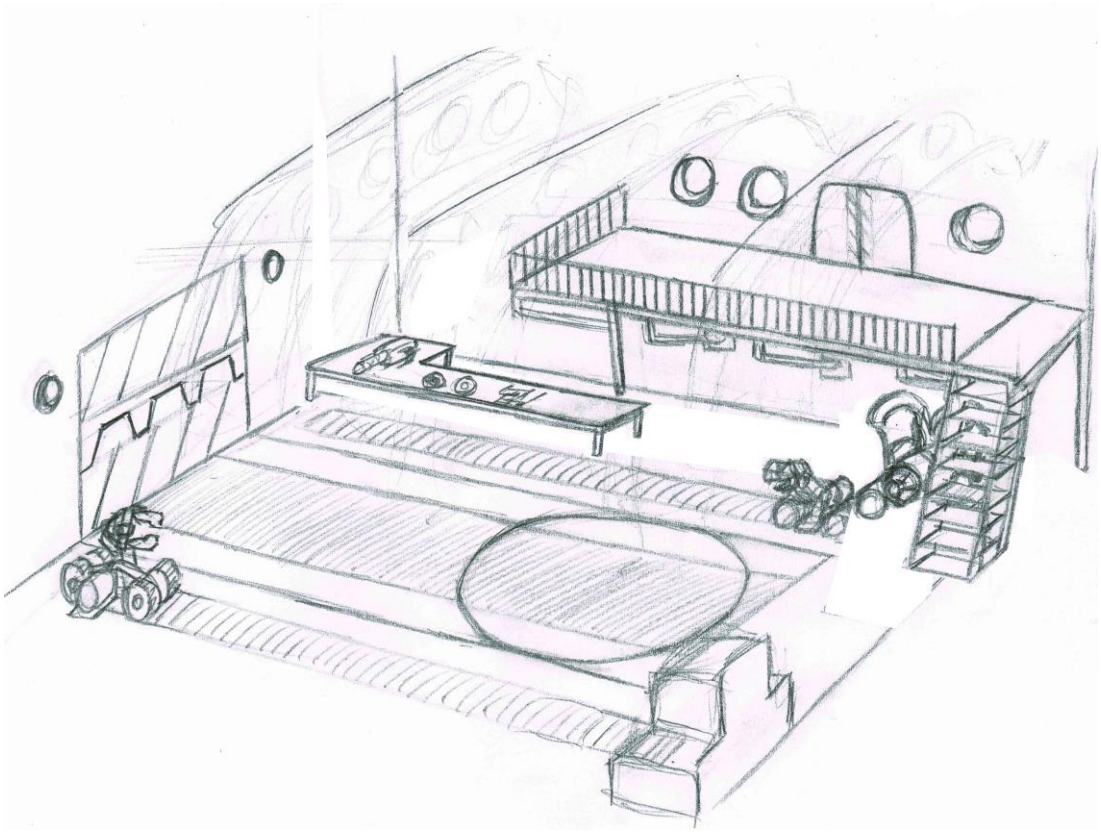


11.1.12. CWorld

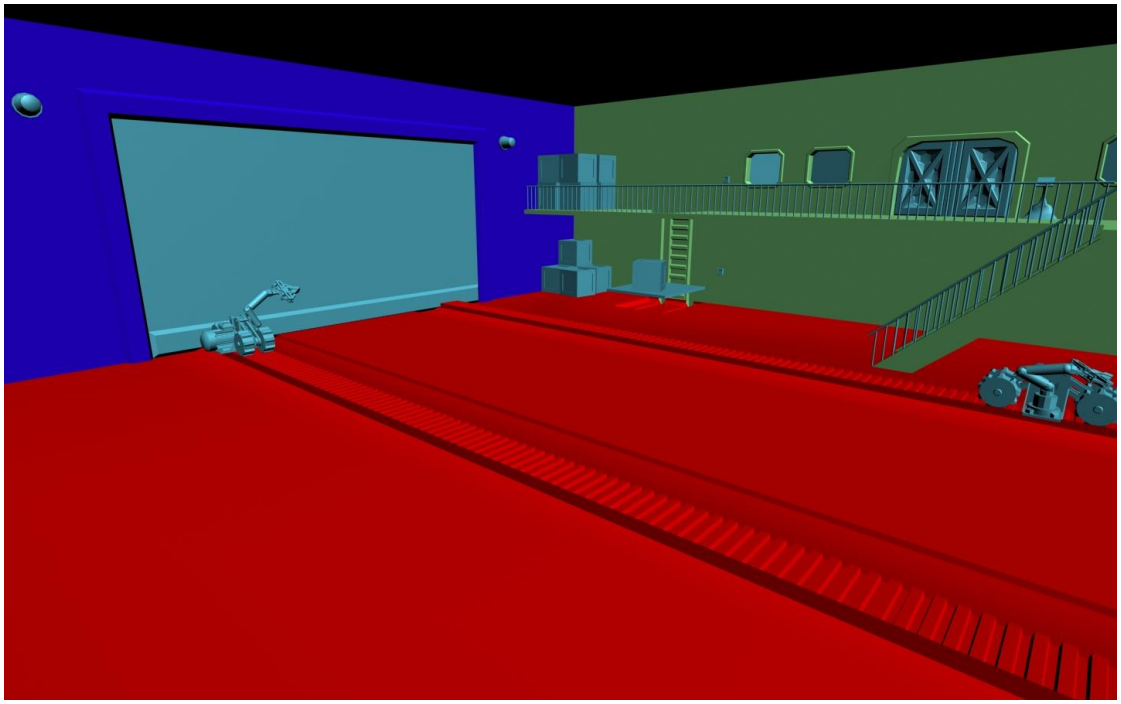


11.2. Skizzen und Entwürfe von Waldemar Hoppe

11.2.1. Hangar Skizze



11.2.2. Hangar Modellkonzept



11.2.3. Shop

11.3. Daten-CD