

|  |
| --- |
| Objektorientierte Programmierung- am Beispiel der Spieleprogrammierung in C++ |

eingereicht als Fachbereichsarbeit aus dem Fach

|  |
| --- |
| Informatik |

zum Maturahaupttermin 2011

von

|  |
| --- |
| Julian Watzinger |

8a Klasse

eingereicht bei Prof. Schinwald

|  |  |  |
| --- | --- | --- |
|  | Eingangsstempel Schule |  |

Inhaltsverzeichnis

[1. Einleitung: 2](#_Toc285650495)

[2. Grundlagen 3](#_Toc285650496)

[2.1 Grundlagen der Programmierung 3](#_Toc285650497)

[2.2. Spieleprogrammierung 5](#_Toc285650498)

[2.3. Programmierschnittstellen 8](#_Toc285650499)

[3. Grundkonzepte meines Spiels 10](#_Toc285650500)

[3.1. Genre 10](#_Toc285650501)

[3.2. Features 10](#_Toc285650502)

[3.3. Game-Engine 11](#_Toc285650503)

[4. Praxis & Umsetzung 15](#_Toc285650504)

[4.1. Die Programmiersprache C++ 15](#_Toc285650505)

[4.1.1 Operatoren 17](#_Toc285650506)

[4.1.2 Elemente 17](#_Toc285650507)

[4.1.3 Variablen: 18](#_Toc285650508)

[4.1.4 Funktionen/Methoden 19](#_Toc285650509)

[4.1.5. Schleifen 20](#_Toc285650510)

[4.1.6. Kontrollstrukturen 22](#_Toc285650511)

[4.2. Verwendete APIs 23](#_Toc285650512)

[4.2.1. Windows-API 23](#_Toc285650513)

[4.2.2. DirectX 23](#_Toc285650514)

[4.3. Verwendung von DirectX in C++ & Einführung in die OOP 24](#_Toc285650515)

[4.3.1 Objektorientierte Programmierung 24](#_Toc285650516)

[4.3.2. Direct3D: 32](#_Toc285650517)

[4.4. Verwendung von Klassen zur Abkapslung von Funktionalität 34](#_Toc285650518)

[4.4.1 Nutz-Klassen: 34](#_Toc285650519)

[4.4.2 System: 35](#_Toc285650520)

[4.4.3. World: 37](#_Toc285650521)

[4.5 STL-Container 39](#_Toc285650522)

[4.6 Problemstellungen 41](#_Toc285650523)

[4.6 Kollisionserkennung 41](#_Toc285650524)

[4.6.2 Gegenseitige Nutzung einzelner Klassen 45](#_Toc285650525)

[5. Schlusswort: 48](#_Toc285650526)

[Quellenverzeichnis: A](#_Toc285650527)

# 1. Einleitung:

Diese FBA behandelt „Objektorientierte Programmierung“, einen Teilbereich der Informatik, am Beispiel der Spiele-Entwicklung. Ich habe mich für dieses Thema aus mehreren Gründen entscheiden. Zum einen finde ich die Programmierung an sich schon lange Zeit faszinierend und habe schon vor einigen Jahren begonnen eigene, kleine Programme zu schreiben. Auch interessiere ich mich sehr für das Thema Computerspiele und deren Entwicklung und wollte mit dieser Arbeit mein Wissen auf diesem Gebieten erweitern und sehen, wie viel man ohne eine einschlägige Ausbildung erreichen kann. Ich sah in dieser Arbeit eine Chance, mich intensiv und mit einer großen Verpflichtung der Erweiterung und Vertiefung meiner Programmierkenntnisse zu widmen. In weniger als einen halben Jahr wollte ich ohne große Vorarbeit ein 2D-Spiel erstellen können. Als Projekt wählte ich dabei ein Jump’n’Run-Spiel. Das Erstellen der Grafiken und Soundeffekt war dabei nicht Teil meiner Arbeit, diese stammen aus dem Konsolen-Spiel „Super Mario World“ von Nintendo. Die Umsetzung dieses Projekts beanspruchte eine große Zeitspanne und erforderte die ständige Änderung bzw. Erweiterung der Konzepte. Aus diesem Grund habe ich diese Arbeit erst nach Vollendung des Projekts geschrieben. Trotzdem verlief die Umsetzung des Projekts ohne gröbere Probleme. Bedanken möchte ich mich an dieser Stelle bei der Community des Spiele-Entwicklungsforums „Gamedev.net“, die mich bei einigen Problemen beraten haben.

# 2. Grundlagen

## 2.1 Grundlagen der Programmierung

Programmieren bedeutet grundlegend das Erstellen von Computersoftware. Dabei geht die Definition vom konzeptuellen Entwurf bis hin zur konkreten Umsetzung im Quelltext, d.h. in für den Computer verwertbaren Programmcode.(vgl. Louis, 2001)

In der professionellen Softwareentwicklung erfolgen diese beiden Bereiche meist getrennt, d.h. ein Programmierer bekommt einen Entwurf vorgelegt und muss anhand dieses Entwurfes ein Programm durch Programmcode realisieren. Im privaten Rahmen, und vor allem wenn man alleine arbeitet, kann auf einen streng getrennten Ablauf des Entwurfs vor der eigentlichen Umsetzung abgesehen werden. So entstand z.B. jeglicher Code, den ich bis jetzt erstellt habe, nach groben Konzepten in meinem Kopf, die ich während der Entwicklung ausarbeitete und stetig verbesserte.

Dennoch gibt es auch als außerberuflicher Programmierer einige wesentliche Grundlagen zu beachten, wie bestimmte Qualitätskriterien der Programmierung, etwa Korrektheit, Robustheit, oder Verständlichkeit des Programmcodes (tse GmbH, 2011). Wenn man damit beginnt, das Programmieren zu erlernen, spielen diese Kriterien noch eine untergeordnete Rolle, schon allein deshalb, weil man ohne gefestigtes Wissen über die Programmierung noch mehr Aufwand darin setzen muss, überhaupt funktionsfähigen Code zu erzeugen, als dass man sich nebenbei noch mit der Ergonometrie dieses Codes beschäftigen kann. Je fortgeschrittener man wird, desto wichtiger wird es aber diese Kriterien zu beachten.

**Werkzeuge der Programmierung**

Während der Entwurf eines Computerprogrammes noch relativ frei durchgeführt werden kann (sofern eine für das Team verständliche Art und Weise wie z.B. Diagramme verwendet wird), benötigt man für die Umsetzung in konkretem Programmcode einige Hilfsmittel.

Zu allererst benötigt man eine bestimmte Programmiersprache. Eine Programmiersprache ist eine formale Sprache, d.h. eine Reihe aus Zeichenketten, die aus einem Zeichenvorrat zusammengesetzt wird, welche es ermöglicht die Computerhardware entweder direkt (in Maschinencode) oder indirekt (als besser lesbaren Quellcode) anzusprechen (vgl. Rousselle 2006). Es gibt eine Unzahl an unterschiedlichen Programmiersprachen, für mein Projekt verwende ich die Sprache *C++*, auf welche ich später noch genauer eingehen werde. Zu beachten ist nur dass alle Befehle Englisch als grundlegende Sprache verwenden. Deswegen werde ich im Weiteren auch viele englische Begriffe verwenden.

Eine sogenannte Hochsprache wie C++ hat den Vorteil leichter Les-, Erlern- und Wartbarkeit, kann im Gegenzug aber vom Computer nicht direkt verwendet werden und muss erst wieder in Form von Maschinencode gebracht werden. Da Maschinencode aber ein für Menschen nur schwer lesbarer Binärcode (es werden nur 0 und 1 als Zeichen verwendet) ist, geschieht dies durch einen sogenannten Compiler, der ‚Übersetzer‘, der diese Arbeit automatisiert durchführt.

Auch wenn man Quellcode mithilfe einer Hochsprache nun theoretisch wie einfachen Text in einem Editor schreiben und danach kompilieren könnte, finden meist sogenannte *Integrierte Entwicklungsumgebungen* (IDE) Verwendung. Eine IDE ist ein Computerprogramm, das die Funktionalität eines Editors und Compilers verbindet sowie eine Vielzahl an weiteren nützlichen Funktionen unterstützt. Dazu gehört der Debugger, ein Tool zur Fehlersuche bei der Programmausführung und Syntax-Fehlererkennung. Es ermöglicht die Diagnose von schwerwiegenden Programmfehlern vor dem Kompilieren. Genauso gibt es den Linker, welcher Programmcode aus verschiedenen Quellen verknüpft und Funktionen zur Formatierung des Quelltextes. Somit sind IDEs aus der heutigen Programmierung nicht mehr wegzudenken, da sie alle sonst einzeln verwendeten Werkzeuge vereinen. Für mein Projekt kommt das *Microsoft Visual C++ Studio 2008 Express Edition* zur Verwendung, eine kostenlos erhältliche C++ IDE.

**Programmierparadigmen**

Obwohl es bereits möglich ist, mit dem reinen Wissen um die Syntax, d.h. die Befehle und Struktur einer Sprache, ein funktionierendes Programm zu schreiben, wird man mit zunehmender Komplexität der Programme bald ein ziemliches Code-Chaos erzeugen. Weder man selber, geschweige denn andere werden damit in der Lage sein, den Programmcode zu lesen, und es passiert sehr leicht Schwierigkeiten bei der Weiterentwicklung des Codes zu bekommen. Deshalb gilt es bestimmte Programmierparadigmen zu beachten, d.h. fundamentale Programmierstile. Diese Paradigmen geben vor, wie die Ausführung eines Programmes beschrieben wird, also ob man das Programm und seine Elemente in Objekte einteilt oder in Folgen von Anweisungen, etc. Jede Hochsprache definiert durch ihre Eigenschaften schon das Programmierparadigma, das man verwenden sollte (vgl. Kastens, 2011). C++ verwendet ein objektorientiertes Paradigma. Was das bedeutet, werde ich an anderer Stelle erklären.

Damit sind die Grundlagen zur Programmierung geklärt.

## 2.2. Spieleprogrammierung

Wie bereits erwähnt ist mein Projekt ein Computerspiel. Spiele-Entwicklung gilt als einer der schwierigsten Teilbereiche der Softwareentwicklung (vgl Wikipedia, 2011a), was sich vor allem dadurch erklären lässt, dass die Spiele-Programmierung viele getrennte Bereiche einschließt, wie grafische Darstellung, Wiedergabe von Sound, künstliche Intelligenz, Netzwerktechnik, aber auch Datenverwaltung. Selbst wenn in einem Team immer mehrere Programmierer am Quellcode schreiben, ist es wichtig für den Spiele-Programmierer alle diese Teilbereiche zu beherrschen. Vor allem Personen, welche die Spiele-Programmierung als Hobby betreiben, sollten alle Aspekte beherrschen. Auch für mein Projekt war es nötig, mir das Wissen über alle diese Teilbereiche anzueignen, da ich in der gesamten Code-Umsetzung auf mich alleine angewiesen war.

Hinzu kommt noch folgendes:

 „Die meiste traditionelle Software reagiert auf eine Benutzereingabe und macht nichts ohne sie. Ein Texteditor z.B. formatiert Wörter und Text wenn sie der Benutzer eingibt. Wenn der Benutzer nichts eingibt, macht der Texteditor nichts. Manche Funktionen brauche eine lange Zeit zum Ausführen, aber alle sind eingeleitet vom Benutzer, der das Programm anweist etwas zu tun.

Spiele hingegen müssen unabhängig von der Benutzereingabe weiterarbeiten.“ (Wikipedia, 2011b)

Abhängig von der Anzahl der FPS, Bilder pro Sekunde (*Frames per Second*) muss das Programm also normalerweise bis zu 60-mal in einer Sekunde aktualisiert werden und z.B. Berechnungen durchführen.

**Spiele-Schleife**

Erreicht wird die ständige Aktualisierung durch eine Spiele-Schleife, die sogenannte „Game-Loop“. Diese Game-Loop ist lediglich eine Schleife, die nach der Initialisierung des Programms solange ununterbrochen durchgeführt wird, bis der Benutzer abbricht. Der folgende Pseudo-Code soll das verdeutlichen:

*während (Benutzer bricht nicht ab)*

 *Benutzereingabe überprüfen*

 *KI ausführen und Gegner bewegen*

 *Sounds abspielen*

 *Grafiken zeichnen*

*ansonsten*

 *Programm beenden*

Zur Verdeutlichung die folgende Abbildung:

Abbildung 1: Einfache Spieleschleife

Dies ist die extrem vereinfachte Art einer Game-Loop. In meinem eigenen Spiel wird jedoch eine Spiele-Schleife wie diese verwendet:

Abbildung 2: Komplexe Spielschleife meines Spiels

## 2.3. Programmierschnittstellen

Da nun die theoretischen Aspekte der Spieleprogrammierung geklärt sind, muss noch besprochen werden wie in der Praxis ein Spiel, unabhängig von der eigentlichen Umsetzung, auf den Bildschirm kommt. Dazu benutzt man Programmierschnittstellen, sogenannte APIs (application programming interface). Es gibt viele verschiedenen Typen von APIs, für mein Projekt sollen jedoch nur solche verwendet werden, welche den Zugriff auf die Hardware erlauben. Denn selbst wenn man mit einer Sprache wie C++ theoretisch direkten Zugriff auf den gesamten PC hat, wäre es übertriebene Arbeit, wenn jeder Programmierer seinen eigenen Hardware-Zugang schreiben müsste. Außerdem müsste dann z.B. für jede erhältliche Grafikkarte ein eigener Zugriffs-Code bestehen. Zu diesem Zweck wurden APIs entworfen. Sie bieten einen vordefinierten Satz an Befehlen, um Hardware (oder auch andere Software, wie das Betriebssystem) direkt anzusprechen. Die folgenden APIs finden in meinem Spiel Verwendung:

**Betriebssystem-API:**

Um überhaupt eine Darstellung eines Programms jeglicher Art zu ermöglichen, muss auf das Betriebssystem zugegriffen werden. Dafür bieten Betriebssystemhersteller für die unterschiedlichsten Programmiersprachen APIs und Laufzeitbibliotheken (Zusammenfassung von Befehlen, die das Betriebssystem ausführt) an. Damit lassen sich grundlegende Dinge wie ein Programm-Fenster anzeigen. Für mehr wird diese API in meinem Projekt auch nicht benötigt.

**Grafik-API**

Für ein Computerspiel ist das Betriebssystem selbst aber nicht flexibel bzw. schnell genug. Zu diesem Zweck gibt es eigene Grafik-APIs, welche die Grafikkarte direkt ansprechen können. Damit lassen sich selbst aufwändige 2D/3D-Szenarien darstellen, wobei für mein Spiel lediglich die 2D-Komponenten wichtig sind.

**Sound-API**

Auch die Sound-Ausgabe muss von einer API extra gehandhabt werden, um auf die Soundkarte zugreifen zu können, um Musik und Soundeffekte wiedergeben zu können. Solche APIs bieten meistens eine große Funktionalität wie 3D-Soundeffekte, jedoch werden für mein Spiel nur die Basis-Funktionen benötigt.

**Eingabe-API**

Während für normale Anwendungen die Betriebssystem-eigene Tastatur/Mauseingabe-Abfrage mehr als genug ist, ist sie für ein Action-Spiel nicht schnell genug. Für mein Spiel wird eine eigene API verwendet, die Tastatureingaben handhabt.

Später werde ich noch genau erklären, welche APIs ich verwende. Doch da nun die grundlegenden technischen Details geklärt wurden, erkläre ich hier das Konzept meines eigenen entwickelten Spiels.

#

# 3. Grundkonzepte meines Spiels

## 3.1. Genre

Ich habe mich für ein klassisches Jump’n’Run-Spiel entschieden, öfters auch als Platformer oder Side-Scroller bezeichnet.

Steuerung:

Der Spieler bewegt sich mittels linker und rechter Pfeiltaste in die jeweilige Richtung und springt per Druck auf die A-Taste.

## 3.2. Features

Der Spieler startet auf einer Art Weltkarte, auf der er sich auf Pfaden bewegen kann und Level auswählen kann. Die Weltkarte selbst ist in einer Art 2D-Vogelperspektive dargestellt. Wenn er ein Level gewählt hat, startet er am linken Ende des Levels und hat das Ziel, das Tor am rechten Ende des Levels zu erreichen. Das Level wird in einer 2D-Seit-Ansicht gezeigt. Abbildung 1 stellt einen Screenshot aus einem Testlevel dar.

Auf dem Weg findet der Spieler Items, die ihm Punkte bringen und solche, die ihn stärken. Er findet auch Gegner vor, die er ausschalten kann, indem er auf ihren Kopf springt, wodurch er auch Punkte bekommt. Wird er selbst getroffen, wird ein Leben abgezogen und er wird auf die Weltkarte zurückgesetzt, sofern er kein stärkendes Item bei sich trägt. Wenn er dies tut, wird sein Item entfernt, er selbst bleibt aber noch am Leben und wird für einige Sekunden unverwundbar. Eine weitere Möglichkeit zu sterben ist es, in ein Loch im Boden zu fallen, oder wenn ein Zeitlimit, welches am Anfang des Levels herunter zuzählen beginnt, auf Null ist. Gesamtziel des Spiels ist es, eine möglichst hohe Punktezahl zu erreichen. Durch erfolgreiches beenden von Levels werden neue Pfade und neue Levels auf der Weltkarte freigeschalten. Das Spiel endet, wenn der Spieler den letzten verfügbaren Level beendet hat, oder wenn seine Leben auf Null sinken. Danach findet eine Auswertung der Punkte statt, und ihm wird eine Highscore-Liste angezeigt.

****

Tile

Item

Gegner

Spieler

HUD

Abbildung 3: Ein Ausschnitt eines Levels des technisch fertigen Spiels.

Hinweis: Sämtliche in meinem Spiel verwendete Grafiken und Soundeffekte wurden nicht von mir selber erstellt, sondern stammen aus dem Spiel „Super Mario World“ von Nintendo.

## 3.3. Game-Engine

Das Spiel wird anhand einer Game-Engine (zu Deutsch: Spiele-Motor) umgesetzt. Dabei handelt es sich um ein Computerprogramm, das den Spielverlauf und die Darstellung der Spielwelt steuert. Dabei vereinen sich alle Teilaspekte der Spieleprogrammierung und alle nötigen APIs zu einem multifunktionalen Programm. Die Funktionen lassen sich grob in diese Bereiche einteilen (Anmerkung: die folgende Darstellung bezieht sich auf meine eigene Game-Engine. Andere Engines können mehr oder weniger Funktionalität haben):

**Grafik**

Ein großer Anspruch wird auf die grafische Darstellung gelegt. Die Engine soll die einzelnen Spielabschnitte wie Titelbildschirm, Level, usw. unter Verwendung möglichst weniger Befehle effizient und automatisiert darstellen. Dabei sollten alle Elemente (z.B. für das Level: Umgebung, Gegner, Items ..) voll automatisch beim Aufruf eines Zustandes (wie hier „Level“) automatisch geladen und auf den Bildschirm gezeichnet werden. Außerdem muss der Grafik-Baustein der Engine grafische Effekte wie ein Black-Fade-Effekt (der Bildschirm wird langsam von den normalen Farben komplett schwarz gerendert) oder Text-Ausgabe beherrschen.

**Physik & Kollisionsabfrage**

Die Physik ist ein weiterer wichtiger Punkt. Zwar benötigt meine Engine keine realistische physikalische Simulation anhand zahlreicher physikalischer Gesetzmäßigkeiten, eine grundlegende Funktionalität muss aber gegeben sein. So sollten z.B. Gegenstände die sich in der Luft befinden nach unten fallen, der Spieler sollte bei einer Bewegung langsam schneller werden und gegen Ende der Bewegung kurz noch weitergehen, bei einem Sprung soll eine realistische Sprungweite und Sprunghöhe erreicht werden. Die Kollisionsabfrage ist ein Teilbereich des Physik-Elements der Engine und ist dafür zuständig zu überprüfen, ob zwei Objekte sich berühren und gegebenenfalls Maßnahmen einleiten (z.B. damit Spieler oder Gegner nicht durch den Boden fallen werden diese vor der grafischen Darstellung bei tatsächlich stattfindender Überlappung „zurückgeworfen“).

**Steuerung**

Ein essenzieller, aber aus Engine-Sicht wenig komplexer Punkt ist die Steuerung. Mittels einfachem Befehl soll abgefragt werden können, ob eine gewisse Taste auf der Tastatur gedrückt wurde. Danach kann man dadurch an anderer Stelle das Verhalten des Spiels bzw. der Spielfigur beeinflussen.

**Sound**

Die Sound-Komponente dient lediglich dazu, kontinuierliche Hintergrundmusik wiederzugeben, aber auch auf Befehl hin einzelne, kurze Soundeffekte abzuspielen.

**Datenverwaltung**

Daten müssen geladen und gespeichert werden. Die Game-Engine liest Text-Dokumente in einem bestimmten Format aus (z.B. zum Laden von Level-Daten) und beschreibt diese bei Bedarf auch im selben Format mit Daten (in diesem Fall nur nötig zur High-Score-Speicherung). Bilder hingegen brauchen diesen Baustein nicht, da die verwendete Grafik-API bereits eigene vereinfachte Lade-Funktionen besitzt.

Weiters besitzt meine Engine noch ein Modul zur Speicherverwaltung. Bilder werden so nur einmal in den Speicher geladen, an ansprechender Stelle verwendet und wenn sie nicht mehr verwendet werden aus dem Speicher gelöscht. Ohne diese Funktionalität wäre eine effiziente Speicherverwendung nicht möglich.

Dies ist die grundlegende Funktionalität meiner Game-Engine. Es gibt jedoch noch andere Punkte, die meine Game-Engine nicht beherrscht, ich aber dennoch erwähnen möchte.

**Netzwerktechnik**

Viele moderne Engines besitzen die Möglichkeit, über ein Netzwerk oder das Internet mit anderen Leuten zu kommunizieren bzw. mit ihnen zu spielen. Da für mein Spiel jedoch kein Multiplayer-Modus vorgesehen ist, wird diese nicht behandelt.

**Scripting**

Ein weiteres, oft gesehenes Feature ist die Verwendung einer Skript-Sprache. Damit können auch Laien nach relativ kurzer Lernzeit Inhalte der Engine anpassen, ohne über den Quellcode oder Wissen über die Programmiersprache verfügen zu müssen. Da dieses Projekt aber lediglich von mir erstellt bzw. bearbeitet wird und auch keine Tools wie einen Level-Editor für Modder, das heißt Personen welche eigene Spielinhalte für ein Computerspiel erstellen, bereitgestellt werden, wird zugunsten von Geschwindigkeit und Einfachheit des Quelltextes darauf verzichtet, da eine Scriptsprache normalerweise erstens viel langsamer ist als die eigentliche Programmiersprache und außerdem im Quelltext selber für viel komplizierten Code sorgt.

Damit sind die Funktion meiner Engine sowie das Konzept meines Spiels geklärt. Die Umsetzung erfolgt anhand dieser Konzepte. Doch bevor ich genauer über die Umsetzung spreche, ist es notwendig die Grundlagen der verwendeten Programmiersprache C++ zu erklären:

#

# 4. Praxis & Umsetzung

## 4.1. Die Programmiersprache C++

Ein Einblick in die Hintergründe der Programmiersprache sowie deren Geschichte und Entstehung ist an dieser Weise unnötig. Stattdessen ist es notwendig, die Syntax und Elemente der Sprache zu erläutern, um später näher erklären zu können, wie ich bei der Entwicklung meines Projekts vorgegangen bin. Es werden dabei nur Elemente erklärt, die auch wirklich in meinem Projekt verwendet werden (Stein, 2011).

**Syntax:**

Die Syntax eines C++-Programmes sieht grundlegend so aus:

#include <iostream>

*// hello world*

*int main(int argc, char \*\*argv)*

*{*

*std::string s("Hello World");*

*std::cout << s << "\n";*

*return 0;*

*}*

Dieser Code kann, ohne auf die einzelnen Methoden und Funktionen sowie Namen einzugehen, wie folgt aufgeschlüsselt werden:

*#x y*

Die Raute veranlasst den Präprozessor (ein Hilfsprogramm das vor der Kompilierung ausgeführt wird, um den Quellcode zu modifizieren) den Code zu modifizieren. Das bedeutet, dass nach den Regeln von x im Quellcode y ersetzt wird, sei es durch externe Dateien, durch Zeichen, etc..(vgl. Rousselle, 2006) Der Vorteil liegt darin, dass der Benutzer davon nichts bemerkt, für ihn existiert nur diese eine Zeile.

<x>

Die beiden spitzen Klammern dienen nur dazu, einen Dateinamen x festzulegen und werden meistens nur für den #include-Befehl genutzt.

//x

Der doppelte Slash zeigt lediglich einen Kommentar des Programmierers an, der vom Compiler nicht berücksichtig wird. Dies dient dazu, anderen Personen den Code zu erklären.

(x)

Die geschlossene Klammer gibt entweder ein benötigtes Argument x in vordefinierter Form an (z.B. für eine Funktion), oder übergibt ein Argument x. Dies wird vor allem für Funktionen benötigt (diese werden später erklärt).

x, y, ..

Der Beistrich zählt wie es die Elemente x, y und folgende auf. Dies kann u.a. für Argumente verwendet werden, hat aber wesentlich mehr Verwendungszwecke.

\*x

Der Stern gibt einen Zeiger auf einen Speicherbereich, auch Pointer genannt, an. Hierzu erfolgt ebenso später die genaue Erklärung.

{…}

Die geschwungene Klammer definiert einen Inhalts-Bereich, sei es eine Funktion, aber auch einer Variable (auch hierzu später die Erklärung).

x::y

Der zweifach vorhandene Doppelpunkt gibt ein Element y aus dem Bereich x an. Dies kann entweder zur Definition oder aber auch zum Abrufen der Definition dienen.

“x“

Anführungszeichen geben lediglich Text wieder.

x << y

Die doppelten Pfeile übergeben das Element y an x. Diese Syntax wird in meinem Projekt sowie in der Windows-Programmierung allgemein nur selten verwendet.

x;

Das Semikolon muss nach den meisten Funktionen am Ende der Codezeile folgen. Es zeigt dem Compiler den Beginn einer neuen Zeile und auch den Abschluss vorhandener Operationen.

return x;

return beendet die Funktion in der es ausgeführt wird und gibt den Wert x zurück.

Somit ist der Code-Ausschnitt, der nicht mehr macht als „Hello-World“ in einer DOS-Konsole auszugeben, syntaktisch verständlich. Man beachte vor allem, dass die Syntax strikt eingehalten werden muss. Bei einem fehlenden ; bricht der Compiler mit einer Fehlermeldung zur betreffenden Zeile ab. Jedoch kann eine { auch etwas weiter eingerückt sein, ohne dass es Fehler gibt.

Nun können viele Angaben aber noch sehr verwirrend wirken. Deswegen ist es wichtig zu erklären, was mit Elementen, Funktionen, Methoden etc. gemeint ist.

### 4.1.1 Operatoren

Vorerst sind noch die Operatoren zu erwähnen. Operatoren führen festgelegte Operationen, grundsätzlich mathematischer Art durch. Es stehen somit alle Grundrechnungsarten +, -, \*, / sowie eine Vielzahl an weiteren Operator-Arten zur Verfügung. Ohne Operatoren wäre eine Daten-Verarbeitung unmöglich.

### 4.1.2 Elemente

Das Element ist in diesem Zusammenhang keineswegs eine gängige Bezeichnung, sondern lediglich von mir zur Veranschaulichung gewählt. Ein Element ist meiner Definition nach alles, von einer Variablen bis zu einer Funktion.

### 4.1.3 Variablen:

Variablen stellen die grundlegendsten Elemente eines C++-Codes dar. Sie dienen zur Speicherung von Daten, sowohl temporär an bestimmten Codestellen als auch dauerhaft während der gesamten Laufzeit des Programms und werden im Quellcode im folgenden Format definiert:

*x y = z;*

*typ name = wert;*

Der Typ einer Variablen gibt an, welche Werte sie später annehmen kann, der Name dient zur Identifikation der Variablen. Mit dem = Operator weist man einer Variable einen Wert zu, der eben mit diesem Typen übereinstimmen muss. Hier ein kurzer Quellcode zu den wichtigsten Variablentypen und deren Verwendung, sowie einigen fehlerhaften Verwendungen:

*//die Variablen werden im Vorhinein definiert:*

*int Ganzahl = 1; //int für Integer gibt eine Zahl ohne Kommastellen an*

*float Gleitkomma = 1.12; //float gibt eine Zahl mit Gleitkomma an*

*int Zahl; // der Variablen muss bei der Definition kein Wert zugewiesen werden*

*LPCWSTR Text = L“Ein Text“; //LPCWSTR speichert den Zeiger auf einen Text*

*bool Wahrheitswert;//bool gibt einen Wahrheitswert an, ist also entweder TRUE oder FALSE*

*//nun werden die Variablen verwendet*

*Zahl = 2; //Nachträgliche Zuweisung eines Wertes, Typ kann weggelassen werden*

*Ganzahl += Zahl; //Zahl wird zu Ganzzahl addiert. Ganzahl beträgt jetzt 3, Zahl bleibt //gleich*

*Gleitkomma -= Ganzzahl;//funktioniert, der Wert von Ganzzahl wird in 3.0 konvertiert //und subtrahiert, Ganzzahl selber bleibt unbetroffen*

*Zahl \*= Gleitkomma;//funktioniert, aber Gleitkomma wird in 1 konvertiert, es geht also //die Information von .12 verloren.*

*Text -= Ganzahl; //Fehler, da int nicht in LPCWSTR konvertiert werden kann*

*Ganzzahl s= NeueZahl; //Fehler, da NeueZahl nicht definiert wurde*

*int NeueZahl; //Variablen können jederzeit definiert werden*

*Ganzahl = NeueZahl; //immernoch Fehler, da NeueZahl keinen Wert zugewiesen hat;*

*NeueZahl = 15;*

*Ganzahl = NeueZahl; //funktioniert. Ganzzahl wird der Wert der Variablen NeueZahl //zugewiesen und beträgt jetzt 15. NeueZahl bleibt unverändert.*

Nun ist die Bedeutung und Verwendung einer Variablen geklärt. Es ist aber auch ersichtlich, dass die Verwendung von Variablen sehr strengen Regeln unterliegt. Es muss genau festgelegt werden, für welchen Zweck eine Variable steht.

Ein weiterer wichtiger Variablentyp ist void. Genau genommen handelt es sich dabei um keinen Typen, denn void definiert, dass die Variable keinen Typ hat, man kann ihr auch keinen Wert zuweisen. Deshalb wird dieser Typ nur selten für Variablen und mehr für andere Bereiche verwendet.

### 4.1.4 Funktionen/Methoden

Eine Funktion bietet die Möglichkeit, Code der öfters benutzt wird (meistens geschieht dies an mehreren Stellen im Code mit leicht veränderten Variablen) auszulagern und einfach aufzurufen. Der Aufbau einer Funktion sieht wie folgend aus:

*rückgabetyp funktionsname(argumenttyp argumentname)*

*{*

 *INHALT*

*}*

Der Funktionsname dient dem Aufruf der Funktion. Der Rückgabetyp verwendet diesselben Typen wie die Variablen (hier kommt auch void ins Spiel, will man keine Rückgabe haben, gibt man void als Typ an). Die angegebenen Argumente stehen der Funktion zur Verfügung, wobei beliebig viele angegeben werden können (man kann auch void angeben wenn man keine Argumente benötigt). Diese müssen jedoch schon bei der Definition festgelegt werden. Beim Funktionsaufruf müssen alle Argumente angegeben werden! Die Verwendung einer Funktion ist:

*float BerechneZinsen(int Kapital, float Zinssatz, int Dauer)*

*{*

*float Zinsen = 1.0 + Zinssatz; //in Funktionen definierte Variablen lassen sich nur dort nutzen*

 *for(int i = 0; i<Dauer;i++) //Schleife, Erklärung folgt. Wiederholt Berechnung*

 *{*

*Kapital \*= Zinsen; //Argumente können innerhalb der Funktion verändert*

 *//werden*

*}*

 *return Kapital; // Endkapital zurückgeben*

*}*

Die Funktion berechnet die Zinsen anhand eines Kapitals in Form eines ganzzahligen Eurobetrags, eines Zinssatzes mit Kommastellen (wichtig: Kommastellen werde nie mit , sondern immer mit . angegeben!) und einer Dauer in Jahren. Aufgerufen wird die Funktion wie folgend:

BerechneZinsen(1000, 8.8, 10); //reiner Befehlsaufruf mit wenig Sinn, nach Ablauf der //Funktion wird der Rückgabewert nicht genutzt

float EndKapital = BerechneZinsen(1000, 8.8, 10); //EndKapital enthält jetzt 1218,994..

Die Funktion kann jederzeit und überall im Bereich, wo sie definiert ist, verwendet werden. Man beachte, dass der Rückgabewert wie ein normaler Wert behandelt wird.

### 4.1.5. Schleifen

Was eine Programmiersprache erst ausmacht sind bestimmte Strukturen, die eine Wiederholung von Berechnungen ohne direkten Befehl des Benutzers möglich machen. Eine davon sind sogenannten Schleifen. Die erste C++-Schleife wurde bereits von mir erwähnt:

for(typ name = wert; abbruchbedingung; berechnung) {

INHALT}

Die for-Schleife wiederholt ihren Inhalt solange, bis die Abbruchbedingung erfüllt ist.

Dazu wird eine beliebige Variable, der sofort ein Wert zugewiesen werden muss definiert, wobei die Verwendung eines Zahlen-Types wie int meistens naheliegt. Diese Variable wird für die Abbruchbedingung verwendet. Außerdem wird nach jedem Schleifendurchlauf eine Berechnung durchgeführt, wobei naheliegt, die vorher definierte Variable zu verwenden. Die Variable ist nur innerhalb der Schleife gültig und verliert nach Ablauf dieser ihre Gültigkeit. Dieser Quellcode erläutert das:

*for(int i = 0; i<6;i++)*

*{*

 *std::cout << i; //gibt i in der Konsole wieder, also der Reihe nach 0-1-2-3-4-5*

*}*

*int Zahl = i; //Fehler, da i nur innerhalb for gültig und danach nach mehr definiert ist*

Diese Schleife definiert die Variable i mit dem Wert 0. Sie erhöht jeden Durchgang i um 1. Erreicht wird dies durch den ++-Operator. Sobald i gleich 6 ist, bricht die Schleife sofort ab und es wird kein Durchgang mehr durchgeführt.

Eine weitere wichtige Schleife ist while:

while(ausführungsbedingung){

INHALT }

Im Gegensatz zur for-schleife wird diese Schleife durchgeführt, solang eine bestimmte Bedingung gilt. Man muss sich um deren Erfüllung bzw. Nicht-Erfüllung komplett selber kümmern. Dabei gilt es vor allem Endlosschleifen auszuschließen. Diese entstehen, wenn die Bedingung nie erfüllt würde (bei der for-Schleife) bzw. immer erfüllt bleiben würde (bei der while-Schleife).

### 4.1.6. Kontrollstrukturen

Ein weiterer wichtiger Faktor sind sogenannte Kontrollstrukturen. Mit ihnen lässt sich überprüfen, ob ein Bedingung zutrifft oder nicht, und danach unterschiedliche Berechnungen durchführen.

Die wichtigste Kontrollstruktur ist if-else. Mit if wird abgefragt, ob eine Bedingung zutrifft, falls nicht, wird else ausgeführt. Der Quellcode dürfte selbstverständlich sein und bedarf keiner weiteren Erklärung:

*if(Wert != 0){*

*TUE\_DIES }*

*else {*

*TUE\_JENES}*

Dies sind die grundlegenden Komponenten von C++, die ich für mein Projekt benötige. Hiermit wäre es bereits möglich, ein Programm zu erstellen. Bevor ich aber daran gehen kann, die wichtigen objektorientierten Konzepte von C++ zu erläutern, welche erst einen gut strukturierten und über längere Zeit verwendbaren Quellcode ermöglichen, soll aber hier noch klar werden, was ich noch dazu benötige, um nicht nur ein Programm im DOS-Fenster, sondern eine Windows-Fenster-Anwendung sowie im Endeffekt ein gerendertes Spiel zu erstellen. Wie bereits vorweg erwähnt verwende ich dafür gewisse APIs.

##

## 4.2. Verwendete APIs

Die folgenden APIs finden Verwendung:

### 4.2.1. Windows-API

Da ich selbst ausschließlich Windows verwende, benutze ich auch dessen API, um auf das Betriebssystem zuzugreifen. Diese wird jedoch nur verwendet, um ein Fenster zu erzeugen, in dem dann die Grafiken meines Spiels gerendert (gezeichnet) werden.

### 4.2.2. DirectX

Eine wesentlich größere Bedeutung erhält hingegen das für Windows ausgelegte DirectX von Microsoft, welches eine Sammlung von APIs für Spiele ist. Es deckt fast den ganzen Multimediabereich ab, u.a. mit der Darstellung 2D/3D-Grafiken, Verarbeitung von Benutzereingabe und Sound.

Es hat den Vorteil, wie bereits erwähnt, dass es eine sogenannte Hardwareabstraktionsschicht für den Programmierer biete, und damit das Programm von der jeweilig verwendeten Hardware unabhängig macht.

Hierbei finden folgende Komponenten von DirectX in meinem Projekt Verwendung:

* DirectX Graphics: Über die API Direct3D werden alle 2D-Szenen in meinem Spiel gerendert
* XAudio2: Mit dieser API gibt mein Spiel Sound-Effekte und Musik wieder
* DirectInput: Diese API nimmt Befehle in Form von Tastatureingaben des Spielers an.

Folgend erkläre ich die Verwendung dieser APIs zusammen mit der objektorientierten Programmierung, da beides in der Spieleprogrammierung einher geht. Ohne Grundwissen über OOP wäre es nicht möglich, die Verwendung von APIs zu verstehen, jedoch wäre es auch falsch, die Objektorientierung komplett im Vorfeld zu klären, da diese doch eine wesentlich größere Bedeutung (vor allem in der 2D-Programmierung) hat als die reine API. Diese ist lediglich ein Werkzeug, um auf Hardware zuzugreifen, die genauen Konzepte zum Steuern der Hardware werden schließlich objektorientiert umgesetzt.

## 4.3. Verwendung von DirectX in C++ & Einführung in die OOP

Um DirectX zu verwenden, muss man zuerst ein DirectX-Objekt anlegen. Danach ist es erforderlich, mittels dieses Objekts ein Direct3D-Device, also ein Gerät, welches die Grafikkarte repräsentiert anzulegen. Der genaue Vorgang wird nur einmal im ganzen Code beschrieben, kann relativ leicht auswendig gelernt werden, weswegen ich ihn hier nicht beschreiben werde, jedoch eine kurze Übersicht gebe, wie das DirectX-Objekt und das Device definiert werden:

LPDIRECT3D9 lpD3D = NULL;

LPDIRECT3DDEVICE9 lpDevice = NULL;

Es gelten dieselben Regeln wie für die Variablen. Jedoch werden hier keine Werte abgespeichert, sondern jeweils ein Zeiger (LP steht hier für Long-Pointer) auf ein Objekt der Klasse IDirect3D9 bzw. IDirect3DDevice9. Diesem Pointer wird zuerst der Wert NULL zugewiesen. NULL ist eine definierte Konstante und ist gleich der Ziffer NULL. An dieser Stelle sollte jedoch geklärt werden, was eine Klasse und ein Zeiger ist.

### 4.3.1 Objektorientierte Programmierung

Wie bereits erwähnt kann man zwar mit der grundlegenden Syntax von C++ bereits Programme schreiben. Größere Projekte lassen sich aber nur schwer realisieren, es sei denn man verwendet OOP als Grundlage. OOP bedeutet einfach gesagt, dass alles ein Objekt ist. Für den Quellcode heißt dies, dass Daten und Funktionen logisch in Objekten abgekapselt werden, und somit Variablen, Funktionen, Methoden, etc.. nur da vorhanden und benutzbar sind, wo sie auch wirklich gebraucht werden. Hier ein Beispiel, um sich dies besser vorstellen zu können:

Angenommen, man will ein Programm schreiben, um eine Landschaft darzustellen. Verwendet man OOP, so ist dann z.B. ein Baum ein Objekt. Ein Baum hat verschiedene Eigenschaften (Variablen) wie z.B. Größe, aber auch die Anzahl der Blätter. Diese Eigenschaften müssen logischerweise gespeichert werden, jedoch wäre es unsinnig dies so zu tun, dass man im ganzen Quellcode darauf Zugriff hat, da ja die meisten anderen Objekte wie Wolken keine Verwendung für diese Daten haben. Würde man ohne OOP arbeiten, wären alle Daten aller einzelnen darzustellenden Gegenstände im Haupt-Code enthalten. Auch müssten Aktionen wie z.B. die Bewegung eines Gegenstandes für jeden Gegenstand im Code vorkommen und wären überall aufrufbar.

Es liegt auf der Hand, dass dies mit der Zeit unübersichtlich und schwer zu kontrollieren sein wird. Zu diesem Zweck kapselt man die Daten in Objekten ab.



Abbildung 4: Darstellung eines Baums als Objekt

Abbildung 4 macht dies noch einmal deutlich. Neben objektspezifischen Eigenschaften und Funktionen kann ein jedes Objekt logischerweise auch selbst Unterobjekte enthalten. Alle diese Dinge sind aber für jedes Objekt geregelt und nicht uneingeschränkt von außen abrufbar, weswegen man von Abkapslung spricht.

In C++ geschieht diese Abkapselung vor allem mit Klassen.

#### 4.3.1.1 Klassen

Klassen übernehmen diese Aufgabe auf eine einfache Art und Weise. Eine Klasse ist eine Sammlung von Daten und Funktionen zum Zweck der Abkapslung.

Eine Klasse besteht meistens aus zwei Dokumenten: Einem Klassenname.h und einem Klassenname.cpp – Dokument. Das ersteDokument ist ein Header-File und dient der Deklaration und Definition der Klasse und ihrer Bestandteile. Es muss auch überall eingebunden werden, wo die Klasse später benutz werden soll. Das zweite Dokument mit der Endung .cpp ist ein normales C++-Dokument und wird verwendet, um die vorher definierten Methoden auszuführen und ihre Funktion zu programmieren.

Eine Klasse ist wie folgt aufgebaut:

*//in SomeClass.h*

*class SomeClass*

*{*

*public:*

*SomeClass(void);*

 *~SomeClass(void);*

 *void SetValue(int Value);*

 *int GetValue(void);*

 *private:*

 *int m\_Value;*

 *void InternalMethod(void);*

 *protected:*

*}*

Die Definition der Klasse erfolgt durch das Vorwort class und den darauffolgenden Klassennamen. Er wird für die gesamte weitere Funktion der Klasse verwendet und sollte daher gut überlegt gewählt werden.

Danach folgt der Inhalt der Klasse, welchen man mit den 3 Begriffen public, private und protected festlegen kann.

Als erstes steht public hierbei für öffentlich verfügbare Daten. Alle Variablen und Methoden, die diesem Wort folgen, können von außerhalb der Klasse benützt werden.

Unter public findet man hier zuerst die Methode SomeClass(void). Dies ist die Konstruktor-Funktion, die jedesmal beim Erzeugen einer Instanz der Klasse aufgerufen wird. Denn jedesmal bei der Definition einer Klasse als Variable wird nicht die Klasse selbst übernommen, sondern eine Kopie von dieser erstellt. Diese Kopie verfügt über alle definierten Variablen und Methoden der Klasse, teilt sie jedoch nicht mit anderen Instanzen der Klasse. Der Konstruktor erstellt diese Instanzen und kann ganz normalen Quellcode ausführen.

~SomeClass(void) ist hingegen der Destruktor, er wird aufgerufen wenn man eine Instanz der Klasse löscht. Man kann hier z.B. besetzte Speicherbereiche freigeben (auch wenn die meisten Variablen automatisch freigegeben werden). Sowohl Konstruktor als auch Destruktor sind zwar als public deklariert, werden jedoch selber nicht aufgerufen. Die Arbeit übernimmt in diesem Fall der Kompiler.

Die nächste Funktion, void SetValue(int Value) ist eine normale Methode, die von außen aufgerufen werden soll und einen Wert der Klasse ändert. Deswegen bezeichnet man solche Methoden auch als „Setter“.

int GetValue(void) hingegen gibt denselben Wert, den der „Setter“ verändert zurück. Man bezeichnet solche Methoden als „Getter“. Zu beachten der Unterschied im Aufbau: Ein Setter erhält logischerweise einen Wert als Paramter, und gibt meistens nichts zurück (=void). Ein Getter jedoch braucht unter normalen Umständen keine Parameter (=void), muss jedoch einen Rückgabetyp besitzen, welcher mit dem zurückzugebenden Wert übereinstimmen muss.

Der zweite Wert, private definiert Methoden und Funktionen, die nur innerhalb der Klasse zur Verfügung stehen. Ein Aufruf von außen führt zur Fehlermeldung. Der Sinn liegt darin zu verhindern, dass Daten fälschlicherweise von außen manipuliert oder auch nur abgerufen werden.

Zuerst wird hier einmal die Variable m\_Value definiert. Sie soll zwar von außen abrufbar sein, jedoch geschieht dies durch die Setter/Getter-Methoden. Man könnte die Variable zwar auch public setzen, jedoch haben Getter und Setter den Vorteil, dass man z.B. fehlerhafte Variablen erkennen und Berechnungen durchführen oder im Fall des Setters auch komplexe Strukturen leicht bestimmen kann (z.B. wenn statt int eine andere Klasse verwendet wird).

Für protected gilt ähnliches wie für private. Jedoch sind unter protected deklarierte Variablen auch für abgeleitete Klassen verfügbar. Eine Klasse ist dann abgeleitet, wenn sie eine Basis-Klasse besitzt, von der sie alle Methoden und Variablen übernimmt, jedoch nur die Bereiche public und protected. Eine Klasse wird wie folgend abgeleitet:

*class NewClass : BaseClass*

*{*

 *…*

*}*

Vor dem Doppelpunkt steht also der Name der neuen Klasse und danach eine gültige bereits vorhandene Klasse, die als Basisklasse hergenommen wird. Ableitung wird vor allem für Objekte verwendet, die in einem Zusammenhang zueinander stehen und einige gleiche Variablen/Methoden haben. Wenn man z.B. eigene Klassen zur Speicherung Rechtecke, Dreiecke und Kreisen haben will, kann man eine Basis-Klasse GeometrischeObjekte mit Variablen für die Positionskoordinaten erstellen, dann die weiteren Klassen davon ableiten und um die weiteren benötigten Werte ergänzen (z.B. Radius für Kreis, Breite und Höhe für Rechteck). Damit spart man sich eine Menge an redundantem Code, vor allem wenn man viele Methoden in der Basisklasse verwendet, welche die abgeleiteten Klassen auch besitzen.

Die Methode einer Klasse wird so umgesetzt:

*int ClassName::GetValue (void)*

*{*

 *if(m\_Value < 0);*

*{*

 *return 0;*

*}*

*return m\_Value;*

*}*

Die Definition einer Member-Methode erfolgt mittels Klassenname und zwei darauffolgenden Doppelpunkten. Ansonsten entspricht alles den Regeln der normalen Methoden. Hier wird auch der Vorteil einer Getter-Methode deutlich. Man kann ohne Probleme z.B. überprüfen, ob m\_Value (m steht auch für Member und sollte zur guten Code-Lesbarkeit verwendet werden) in einem bestimmten Wertbereich ist. Hier wird überprüft, ob m\_Value kleiner als 0 ist. Wenn dies der Fall ist, wird 0 zurückgegeben. Beachtenswert ist auch das Fehlen der else-Bedingung, was sich daraus erklärt, dass return nicht nur einen Wert zurückgibt sondern die Methode abbricht. Ist return kleiner als 0 wird 0 zurückgegeben und die zweite return-Anweisung wird nicht mehr ausgeführt. Wenn m\_Value größer als 0 ist, wird der Inhalt des if-Statements übergangen und m\_Value zurückgegeben.

Mit diesem Wissen ausgestattet, kann man bereits eine einfache Klasse erstellen.

#### 4.3.1.2. Zeiger:

Nun sollte noch erklärt werden, was ein Pointer ist und welche Bedeutung er hat. Der Pointer zeigt, wie der Name bereits andeutet, auf den Speicherbereich einer Variablen.

Ein Beispiel:

*#include „SomeClass.h“*

*SomeClass Instance;*

*SomeClass\* Pointer = NULL;*

*Pointer = &Instance;*

An dieser Stelle eine weitere Bemerkung zu Klassen: Um eine Klasse zu verwenden, muss sie im entsprechenden Code eingebunden werden. Dies geschieht wie anfangs mit #include, jedoch dieses Mal mit „“ und dem Dateinamen des Header-Files der Klasse.

In diesem Beispiel wird also eine Klasse „SomeClass“ eingebunden. Danach wird eine Instanz der Klasse wie bei einer normalen Variablen erzeugt. Dies ist wie bereits erwähnt kein großer Aufwand: Man deklariert eine normale Variable mit dem Klassennamen als Dateityp, den Rest übernimmt der Konstruktor, man muss nicht einmal einen Wert zuweisen (wichtig ist hier noch, dass der einzig gültige Wert eine andere Instanz derselben oder eine abgeleiteten Klasse wäre).

Die Zeile SomeClass\* Pointer = NULL; erzeugt lediglich einen Zeiger. Wie bereits gesagt zeigt dieser auf eine Speicheradresse, in diesem Fall NULL (= 0). Der Pointer enthält also nichts anderes, als die Hexadezimal-Adresse eines Objekts im Speicher des Computers. Der Stern definiert, dass es sich um einen Pointer des besagten Dateityps handelt. Bis auf die Tatsache, dass auch für einen Zeiger der Dateityp übereinstimmen muss, gibt es hier nichts Wichtiges zu beachten.

Interessanter ist da schon die nächste Zeile Pointer = &Instance; Was passiert hier? Dem Pointer wird ein neuer Wert zugewiesen, nämlich die Speicheradresse des erzeugten Objekts SomeClass. Das geschieht mittels &, welches statt dem eigentlichen Speicherinhalt die Speicheradresse angibt. Nach der Operation enthält also Pointer einen Zeiger auf den Speicherbereich von SomeClass. Der Vorteil daraus ist, dass durch die Verwendung von einem Pointer keine weiteren Instanzen einer Klasse enstehen. An folgendem Beispiel sollte dies noch einmal deutlich werden:

SomeClass Instance;

… //irgendwo anders im Code

SomeClass InstanceCopy = Instance;

In diesem Beispiel würde also eine genaue Kopie der Instanz entstehen. Ein gängiger Anfängerfehler ist zu glauben, dass beide Instanzen genau denselben Speicherbereich verwenden und daher die Variablen gleich bleiben, wenn sie verändert werden. Dies ist jedoch falsch: Verändert man eine Variable von Instance bleibt InstanceCopy unverändert. Will man also dieselbe Instanz einer Klasse an mehreren unabhängigen Stellen im Code verwenden, so nutzt man Pointer, da man damit immer nur die eine Instanz verwendet.

Nun sagte ich ja am Anfang, die angelegten DirectX-Variablen wären Pointer.

*LPDIRECT3D9 lpD3D = NULL;*

*LPDIRECT3DDEVICE9 lpDevice = NULL;*

In diesem Code fehlt zwar der \*, jedoch ist LPDIRECT3D9 auch keine Klasse, sondern eine abstrakte Definition, die eben genau einen Pointer auf den Speicherbereich des Objekts zurückgibt. Man verwendet in DirectX immer diese Long-Pointer-Instruktionen, um nicht aus Versehen eine weitere Instanz desselben Objekts zu erzeugen.

Nun, da das Wichtigste über Pointer geklärt wäre, fehlt noch, wie man eine Methode oder Variable einer Klasse aufruft. Innerhalb der Klasse geschieht dies ganz normal. Aber wenn man eine Klassen-Instanz verwendet, muss man einen kleinen Umweg gehen:

Aufruf der Methode einer Klasse:

Eigentlich ruft man die Methode der Instanz einer Klasse auch außerhalb ganz normal auf, mit Methodenname, Parametern, etc. . Da die Methode aber nicht global im Code, der die Instanz beinhaltet verfügbar ist, funktioniert dies so:

*int Value = 0;*

*SomeClass.SetValue(1); //m\_Value von SomeClass*

*Value = SomeClass.GetValue(); //Value ist nun gleich 1*

Die Variable der Instanz kommt zuerst, danach folgt ein Punkt und hinter dem Punkt der Funktionsaufruf. Obwohl dies auf den ersten Blick etwas eigen aussieht, funktioniert es ganz normal, auch Rückgabewerte werden wie in Zeile drei zu sehen ist ganz gewöhnlich gehandhabt. Doch wie wendet man dies nun auf Pointer an?

Mittels eines einfachen Zeichens kann man ohne Probleme den Speicherbereich, auf den der Pointer zeigt, ansprechen:

*SomeClass\* Pointer;*

*Pointer->SetValue(2);*

Dieser Pfeil -> erlaubt Zugriff auf den Speicherbereich und ersetzt somit einfach den Punkt.

Damit ist die grundlegende Verwendung von Klassen abgedeckt. Noch wichtiger als die reine Nutzung der Klassen ist deren Umsetzung in einem Projekt, welche ich auch erklären werde. Jedoch zuerst noch ein kurzer Einblick in die verwendete Direct3D-Mechanik, um sich einen Überblick verschaffen zu können, wie denn die Spiele-Grafik letztendlich von meiner Engine verarbeitet und angezeigt wird.

### 4.3.2. Direct3D:

Ein jeder Computer-Monitor besitzt eine festgelegt Anzahl an Bildpunkten, sprich Pixel. Jeder Pixel kann zu jeder Zeit genau eine Farbe anzeigen. Auf diese Weise werden Computergrafiken angezeigt. Grafiken werden positioniert, indem man sie an ihren Pixel-Koordinaten zeichnet (z.B. ein Icon an Stelle 568/338). Auf diese Weise arbeitet auch DirectX, jedoch beginnt entgegen der allgemeinen Annahme dessen Koordinaten-System nicht links unten, sondern oben mit der entsprechenden Ecke des verwendeten Anwendungs-Fensters mit Punkt (0/0). Alles links davon oder darüber ist negativ, alles rechts davon geht bis ins Unendliche (Grafiken können damit auch abseits des Bildschirms gezeichnet vorhanden sein, werden dann jedoch nicht gerendert).

Doch wie entstehen Grafiken in Computerspielen am Bildschirm? Dafür stellt uns DirectX einige Werkzeuge zur Verfügung:

**LPDIRECT3DTEXTURE9 Texture:**

Das grundlegendste Direct3D-Interface ist eine Textur. Diese Textur enthält die Pixeldaten einer Grafik oder eines Bildes, d.h. die Farbwerte der einzelnen Bildpunkte der Reihe nach im Speicher der Grafikkarte. Die Texturen haben keinerlei Positionen und sind nicht geordnet, und enthalten quasi nur die Daten der Grafik, weswegen man allein damit nicht rendern kann. Es gibt viele Möglichkeiten eine Textur mit Pixeldaten zu füllen. Die von mir am häufigsten verwendete ist das Laden einer Grafik von der Festplatte.

**LPD3DXSPRITE9 Sprite:**

Das Direct3D-Sprite-Interface wiederum sorgt für komfortable Darstellung einer Textur. Unter Angabe einer Position und eines LPDIRECT3DTEXTURE9-Objekts kann die entsprechende Textur auf dem Bildschirm zeichnen. Im Gegensatz zu LPDIRECT3DTEXTURE9, wo jedes Objekt eine Grafik enthält, sollte man immer nur ein LPD3DXSPRITE9-Objekt verwenden, und alle Texturen mit diesem zeichnen (alles andere würde zu viel Ressourcen und Performance verbrauchen). Das LPD3DXSPRITE9-Objekt selber enthält nur wenige Daten und wird nur zum Rendern von Texturen verwendet.

**LPD3DIRECT3DSURFACE9 Surface:**

Ein Surface ist eine Oberfläche, auf die man Grafiken, aber auch einfach nur Farben oder ähnliches zur Laufzeit des Programms auftragen kann. Jedes Surface nimmt Speicherplatz im Grafikspeicher ein. Es gibt hier verschiedene Typen, z.B. Off-Screen-Surfaces, welche nicht sichtbar sind und nur verwendet werden, um Grafiken für Texturen zu erzeugen. Man kann entweder eine Textur direkt aus einem Surface laden (anstatt eines vorhandenen Bildes) oder auf umständliche Weise das Surface oder Teile des Surfaces auf die Textur auftragen. Wichtig ist noch, dass jede Textur ein Surface besitzt. Nur durch dieses Surface kann man die Textur verändern.

Es gibt noch weitere Direct3D-Interfaces, diese drei sind jedoch die wichtigsten und werden immer wieder benutzt. Heißt das nun, ich muss immer wenn ich eine Grafik rendern will im Code eine Textur erzeugen und dort einen Zeiger auf das Sprite-Interface ablegen, sowie Informationen wie Position speichern (man beachte dass mehrere hundert Objekte auf einem Bildschirm sein können, die sich alle verteilt im Code befinden)? Nein, denn hier kommen wieder die Klassen ins Spiel.

## 4.4. Verwendung von Klassen zur Abkapslung von Funktionalität

Dies ist ein gutes Beispiel zur Verdeutlichung der Wirksamkeit von Klassen. Anstatt an tausenden Codestellen immer dasselbe Muster zu verwenden um z.B. ein Objekt zu rendern, wird eine Klasse erstellt, welche die benötigten Funktionen abkapselt. Somit wird das ganze Programm in ein Netz aus Klassen verwandelt, wobei jede Klasse eine Funktionalität repräsentiert. So ist z.B. die Kamera eine eigene Klasse, der Level, aber auch die Gegner, etc… Hier eine Übersicht über die von mir für meine Engine verwendeten Klassen (unterteilt nach deren Funktionen). Die Klassen verwenden alle vorangestellt ein C, was für Custom steht. Dies soll jene Klassen klar als eigens erstellte Klassen identifizieren.

### 4.4.1 Nutz-Klassen:

Hierbei handelt es sich um Klassen, die bestimmte Funktionen für die Engine bereitstellen, wie die einfache Anzeige von Grafiken, die Abfrage der Tastatur, die Verwaltung von Ressourcen. Diese Klassen werden ausschließlich von außen verwaltet und werden fast überall im Quellcode genutzt.

**CDirect3D:** In dieser Klasse wird das Direct3D-Objekt sowie das Graphic-Device gespeichert und verwaltet, außerdem enthält es noch das Direct3D-Sprite-Interface. Man kann sich diese Objekte ausgeben lassen und an anderer Stelle im Code verwenden.

**CGraphics:** Eine Klasse, die grafische Effekte aller Art bietet. Besitzt Befehle zum Ausführen dieser Effekte.

**CInput:** Mit dieser Klasse fragt man die Benutzereingabe ab. Es gibt nach außen lediglich Methoden, um Tastenabfragen zu machen.

**CResourceManager:** Ein Ressourcen-Manager, in dem alle Texturen gespeichert werden (um Duplikate zu vermeiden). Besitzt Methoden zum Einfügen, Abfragen und Löschen von Texturen.

**CSprite:** Kapselt die vorher genannte Funktionen eines „virtuellen“ Sprites ab, indem Position, Animationsstatus, ein Zeiger auf die Textur usw gespeichert werden. Bietet eine Vielzahl an Methoden um alle Objekte im Spiel anzuzeigen und zu verwalten.

**CTextBox:** Dieser Klasse stellt eine Textbox zur Verfügung, mit der man mit Methoden einige Zeilen Text festlegen und anzeigen kann.

### 4.4.2 System:

Diese Klassen steuern und beeinflussen die Spielmechanik direkt. Im Gegensatz zu den Nutz-Klassen sind diese Klassen großteils autonom, d.h. sie verwalten sich selbst. Eine Kommunikation innerhalb der verschiedenen Klassen dieser Kategorie ist oft vonnöten, weswegen viele dieser Klassen Pointer zueinander haben. So kann die Klasse CSceneGame der Verwaltungs-Klasse CSystem mitteilen, dass der Spielablauf pausiert werden soll. Umgekehrt unterbindet CSystem in so einem Fall die Aktualisierung aller am Spielablauf beteiligter Klassen, also auch CSceneGame.

**CAnimations:** Zeigt kurze grafische Animationen für Spieler oder Gegner an (z.B. Explosion, Funken, etc.. normale Charakteranimation erfolgt in der CSprite-Klasse). Methoden um Animationen mit bestimmter Textur festzulegen.

**CCamera:** Eine Klasse, welche die Funktion einer Kamera übernimmt, und damit festlegt, welchen Ausschnitt der Spielwelt am Bildschirm angezeigt wird und wie sich dieser Ausschnitt bei Bewegung usw. verändert. Bietet Methoden zum Ändern der Kamera-Koordinaten.

**CCollision:** Eine kleine Klasse, welche Methoden für die Kollisionserkennung zwischen zwei Objekten wie Spieler, Gegner, Items und Umgebungsobjekten besitzt.

**CControl:** Im Gegensatz zur CInput-Klasse verwaltet diese Klasse die konkrete Steuerung des Spielers, verwaltet sich komplett selbstständig und aktualisiert andere Klassen je nach Steuerungs-Schema.

**CGameStates:** Eine Klasse zum Speichern von Status-Werten wie Punkte, Leben, geschaffte Levels, etc. . Methoden zum Ändern und Ausgeben dieser Daten.

**CHUD:** Diese Klasse stellt das Head-Up-Display bereit, d.h. das Interface des Spielers in dem Punkte, verbleibende Zeit, vorhandene Items, etc.. im Level angezeigt werden.

**CScene:** Wohl die wichtigste Klasse der ganzen Engine. Sie stellt den Bildschirminhalt dar, und je nachdem, wo man sich gerade im Spiel befindet, gibt es eigene abgeleitete Klassen:

**CSceneGame:** Stellt ein Level des Spiels mit allen dort befindlichen Objekten dar. Bietet kaum öffentliche Methoden und verwaltet sich fast total selbstständig. Verwaltet auch die im Level enthaltenen Objekte und aktualisiert sie.

**CSceneGameOver**: Rendert einen Game-Over-Bildschirm. Selbständige Verwaltung.

**CSceneOverworld:** Stellt die Weltkarte des Spiels dar und verwaltet sich soweit selber.

**CSceneTitle:** Eine Klasse zum Rendern des Titelbildschirms. Stellt gleichzeitig unter Verwendung einer Instanz von CSceneGame im Hintergrund eine Demo des Spiels durch Abspielen eines Levels dar.

**CSound:** Diese Klasse kapselt die XAudio2-Verwendung ab und bietet einfache Methoden zum Abspielen von Musikstücken als Hintergrundmusik oder Soundeffekt.

**CTimer:** Eine Instanz dieser Klasse erzeugt einen Timer, den man verwenden kann um Ereignisse zeitverzögert eintreten zu lassen. Man kann den Timer mittels Methoden auf ein Zeitlimit setzen und abfragen, ob es überschritten wurde.

**CSystem:** Verwaltet den aktuellen Status des Spiels, z.B. welche Szene gerade angezeigt werden muss, ob eine Szene gerade gewechselt wird oder ob andere wichtige Ereignisse eingetreten sind, die den Spielfluss beeinflussen.

### 4.4.3. World:

Alle Klassen dieser Art repräsentieren die Spielwelt, genau genommen Objekte daraus. Sie stellen nur eine logische Repräsentation dieser Objekte dar, d.h. sie beinhalten Werte und Status dieser Objekte. Die grafische Darstellung geschieht mittel Nutz-Klassen, allen voran CSprite. Genauso wie die Nutz-Klassen verwalten sie sich nicht selbst sondern werden in regelmäßigen Abständen von den System-Klassen aktualisiert. Dafür verwalten die Klassen der Kategorie World jeweils ihre verwendeten Nutz-Klassen selbst.

**CBackground:** Eine Klasse, welche das einfache Anzeigen von Hintergrundgrafiken, auch Panorama genannt, ermöglicht. Jede Instanz der Klasse repräsentiert dabei einen Hintergrund.

**CEnvironment:** Diese Klasse verwaltet Instanzen der Klasse CSprite, um Umgebungsobjekte ohne Interaktionsmöglichkeiten mit dem Spieler darzustellen. Dabei handelt es sich um Hinter- bzw. Vordergrundobjekte, sowie Objekte zur Dekoration.

**CLevel:** Diese Klasse dient dem Laden aller Objekte, die ein spezifischer Level beinhaltet, also Gegner, Items, Umgebungsobjekte sowie Hintergründe.

**CObjects:** Hierbei handelt es sich um eine sehr vielschichtig Klasse, welche sogenannte Game-Objects behandelt. Ein Game-Object ist ein Objekt in der Spielwelt, mit dem der Spieler interagieren kann. Die Klasse selber stellt nur rudimentäre Werkzeuge zur Erstellung einfacher Game-Objects zur Verfügung, jedoch leiten sich davon die Klassen CEnemy und CItem ab. Diese stellen jeweils Gegner bzw. Items dar.

**CPlayer:** Bei dieser Klasse handelt es sich um die Spielfigur. Sie ist eine Ausnahme der World-Klassen, da sie sich im Gegensatz zu den anderen Klassen dieser Kategorie großteils selbst verwaltet. Einzige Ausnahme ist die Ausführung einer Bewegung, die durch die Input-Klasse vorgegeben wird.

**CTileSet:** Die Tileset-Klasse beinhaltet alle Tiles des Levels, also alle 32\*32-großen Boden-, Wand- oder Decken-Felder, die den Level vorgeben und Spieler sowie GameObjects in ihren Aktionen beschränken.

Dies sind alle in meiner Engine verwendeten Klassen. Damit kann ich bereits mein Spiel darstellen und aktualisieren sowie die Spielmechanik beeinflussen. Für die meisten Klassen aus der Kategorie „System“ wird jeweils eine Instanz im Haupt-Code erstellt. World & Nutz-Klassen hingegen müssen öfters erstellt werden. So wird z.B. die Klasse CSprite für jedes grafische Element das angezeigt werden soll erstellt. Die folgende Abbildung zeigt die Verwendung der Klassen in der Game-Loop.

Abbildung 5: Komplexe Darstellung der Game-Loop sowie Funktion meiner Engine mit den verwendeten Klassen

Die Engine ist damit komplett. Sollte dennoch einmal noch zusätzliche Funktionen vonnöten sein, kann ich die Klassen einfach um weitere Methoden und Variablen erweitern. Wenn jedoch größere Erweiterung benötigt würde, wie z.B. Netzwerktechnik oder Scripting, müsste ich neue Klassen einführen. Diese könnten jedoch einfach in das bestehende System eingebaut werden.

## 4.5 STL-Container

Ein weiterer Punkt, der zum Erstellen von Software und insbesondere Spiele eine große Wichtigkeit hat sind sogenannte STL-Container. Dabei handelt es sich um eine Sammlung von Klassen, die dazu verwendet werden, andere Objekte in einem eigenen Objekt zu speichern. Ein STL-Containerobjekt kann dabei beliebig viele andere Objekte beinhalten, die alle nach einem dem Container entsprechenden Prinzip geordnet sind. Hier die Definition eines Objekts des Vector-Containers, welcher alle beinhalteten Objekte einer Zahlenreihe nach ordnet.

vector<CSprites> m\_Sprites;

In spitzer Klammer wird der Typ der enthaltenen Objekte definiert. Jeder Container kann nur einen Typen von Objekten enthalten, dieser kann auch bei Laufzeit nicht geändert werden. Verwendet wird der Container so:

*//Anzahl der maximal enthaltenen Objekte auf 5 festlegen*

*m\_Sprites.resize(5);*

*//ein Objekt an Platz 1 setzen*

*m\_Sprites[1] = Sprite;*

*CSprite Sprite2;*

*//das Objekt von Platz 1 abrufen*

*Sprite2 = m\_Sprites[1];*

Ohne diese Container wäre es unmöglich, komplexe Spiele zu entwerfen. Wollte ich ohne einen Container z.B. 10 Gegner in meinem Spiel haben, müsste ich jetzt für jeden Gegner ein eigenes Objekt im Quellcode erstellen:

*CEnemy Enemy1;*

*CEnemy Enemy2;*

*CEnemy Enemy3;*

*…*

Mit dem vector-Container kann ich nun dies alles in einem einzigen Objekt vereinen:

*vector<CEnemy> m\_Enemies;*

Um Zugriff auf alle Elemente eines Containers zu bekommen, verwendet man eine for-Schleife. Bei vector-Containern hat man noch relativ leicht mittels Zahlen auf die einzelnen Objekte Zugriff. Dagegen benötigt man bei der zweiten Art der von mir verwenden Container, der *map*, einen Iterator:

//Definition einer Map

map<int, CEnemy> m\_Enemies;

for(map<int, CEnemy>::iterator ii=m\_Enemys.begin();ii!=m\_Enemys.end();++ii)

{

//Gegner zeichnen

 ii->second.Draw

}

Die for-Schleife iteriert also durch den Container. ii enthält bei jedem Durchgang ein Paar aus dem ersten und zweiten Element der map. ii->first würde in diesem Beispiel einen int-Wert ausgeben, und ii->second eine Instanz der CEnemy-Klasse.

## 4.6 Problemstellungen

Im Folgenden möchte ich einige Problemstellungen erläutern, die sich während des Programmierens meines Spiels aufgetan haben. Alle diese Probleme haben mich dazu gezwungen, meine Konzepte zu überarbeiten und teilweise meine Engine grundlegend zu überarbeiten.

### 4.6 Kollisionserkennung

Die Kollisionserkennung ist eine der grundlegendsten und wichtigsten Techniken in der Spieleprogrammierung. Kollisionserkennung dient, wie der Name schon sagt, der Erkennung von Kollisionen zwischen Objekten, also z.B. Spieler und Umgebung, Spieler und Item, Item und Umgebung. Die Kollisionserkennung selbst ist an sich leicht umsetzbar.

*bool CCollision::Collision(CSprite \*Sprite1, CSprite \*Sprite2)*

*{*

 *int left1, left2;*

 *int right1, right2;*

 *int top1, top2;*

 *int bottom1, bottom2;*

 *left1 = Sprite1->GetPosition().x;*

 *left2 = Sprite2->GetPosition().x;*

 *right1 = left1 + Sprite1->GetWidth();*

 *right2 = left2 + Sprite2->GetWidth();*

 *top1 = Sprite1->GetPosition().y;*

 *top2 = Sprite2->GetPosition().y;*

 *bottom1 = top1+ Sprite1->GetHeight();*

 *bottom2 = top2 + Sprite2->GetHeight();*

 *if (bottom1 < top2) return false;*

 *if (top1 > bottom2) return false;*

 *if (right1 < left2) return false;*

 *if (left1 > right2) return false;*

 *return true;*

*}*

Dieser Methode werden zwei Zeiger auf Sprite-Objekte übergeben. Deren Positionen sowie Ausmaße werden dann miteinander verglichen um zu bestimmten, ob sie sich überschneiden. In diesem Fall gibt die Methode true zurück, andernfalls wird false zurückgegeben.

So einfach das Grundkonzept der Kollision aussieht, umso schwerer ist es, dies in der Realität umzusetzen. Denn es reicht nicht bestimmten zu können, ob sich zwei Sprites überschneiden, um eine Physik simulieren zu können. Hier taten sich schon erste Probleme bei der Kollision zwischen Spieler und Gelände auf. Mein erster Ansatz hierzu war wie folgt:

Für die gesamte Umgebung werden einzelne Sprites verwendet. Jedes Gelände-Objekt ist also durch ein eigenes Sprite mit eigener Textur repräsentiert. Nach der vorher gezeigten Kollisions-Methode wird nach jeder Bewegung des Spielers dessen Sprite mit jedem Gelände-Sprite verglichen. Bei einer Kollision wird der Spieler nach den Koordinaten des kollidierten Sprites ausgerichtet, sodass sich die Sprites nicht mehr überschneiden, sondern nur noch berühren. Die folgende schematische Abbildung zeigt dies.



Abbildung 6: Schritt 1-3 der Kollisionserkennung – rot: Gelände , gelb: Spieler

Man erkennt deutlich den Ablauf von Schritt 1 bis 3. In Schritt 1 führt der Spieler eine Bewegung durch. Schritt 2 zeigt eine Überschneidung des Spielers mit dem Gelände. In Schritt 3 wird der Spieler vertikal am Sprite des Geländes ausgerichtet. All dies geschieht schneller als der Spieler es wahrnehmen könnte.

Dennoch hatte diese Methode, wie ich später bemerkte, einige große Mängel. So musste für jedes einzelne Stück Gelände eine eigene Grafik vorliegen. Auch wenn sich Geländestücke sehr ähnelten, aber nur geringfügig anders geformt waren oder aussahen, musste eine eigene Grafik vorliegen. Dies ist sehr redundant und verbraucht unnötigen Speicherplatz auf der Festplatte. Außerdem musste jede diese Grafiken in den Grafikspeicher geladen werden, womit dort ebenfalls eine Unmenge an Speicher verbraucht wurde. Eine solche schlechte Ausnützung des Grafikspeichers kann sich negativ auf die Leistung auswirken und zu Fehlfunktionen im Programm wie Abstürzen führen. Außerdem ist diese Methode problematisch, falls der Spieler mit mehreren Gelände-Stücken gleichzeitig kollidiert, in diesem Fall kam es immer wieder zu fehlerhafter Ausrichtung des Spielers.

**Lösung:**

Um diese Probleme zu beseitigen, musste ich das gesamte Gelände-System auf die Nutzung sogenannter Tiles umstellen. Ein Tile beschreibt ein quadratisches, in meinem Fall 32\*32 Pixel großes Kästchen. Aus diesen Kästchen wird das ganze Level zusammengesetzt. Jedes Kästchen wird entweder leer gelassen oder mit einem Tile aus dem Tileset befüllt. Das Tileset ist eine Grafik die alle für ein Level möglichen Tiles enthält. Da DirectX für die Nutzung von quadratischen Grafiken optimiert ist, habe ich mich hier für eine Größe von 512\*512 Pixel, also 16\*16 Tiles, insgesamt 256 Tiles pro Tilset entschieden.



Abbildung 7: Ein unfertiges Tileset

Nun wird einfach für jede Position im Level der Index des jeweiligen Tiles gespeichert. Der Index ergibt sich aus xID+yID\*16. Für das 5.Tile von links und 2. von oben ergibt sich also der Index von 20 (es wird immer von 0 angefangen zu zählen). Beim Rendern wird dann der Index zurückgerechnet. Index%16 ergibt die xID, Index/16 ohne Kommastellen ergibt die yID. Mit diesen Daten wird dann das entsprechende Tile aus dem Tileset auf die Stelle im Level gerendert. Dieser Vorgang wird zwar jedes Frame durchgeführt, was aber erst bei Unmengen an Tiles zu einer Performance-Beeinträchtigung führt. Der Speichergewinn ist aber enorm, da jetzt nur noch die eine Tileset-Grafik im Grafikspeicher vorhanden sein muss. Auch die Kollisionsabfrage ist jetzt wesentlich einfacher: Da das Level nun in eine Grid, d.h. ein Raster eingeteilt ist, kann man jeden Eckpunkt des Kollisions-Rechtecks des Spielers auf seine Position in dem Raster überprüfen. Nun muss man nur noch überprüfen, ob ein Tile an dieser Position des Rasters liegt. In diesem Fall ist eine Kollision aufgetreten, andernfalls nicht. Der Vorteil liegt darin, dass nun nicht mehr mit allen Geländeobjekten auf Kollision überprüft werden muss, sondern nur noch einmal für das gesamte Gelände. Auch das Erstellen von Level ist auf diese Weise viel angenehmer. Somit stellte der Umstieg auf ein Tile-System für mich eine gewaltige Verbesserung dar.

### 4.6.2 Gegenseitige Nutzung einzelner Klassen

Wie ich bereits vorher verdeutlicht habe, hängen die Klassen meiner Game-Engine stark miteinander zusammen. Klassen wie CEnemy können für ihre grafische Darstellung eine eigene Instanz von CSprite erstellen, die nur sie selber nutzen. Jedoch kann es auch notwendig sein, dass die Instanz einer Klasse von mehreren anderen Instanzen anderer Klassen verwendet wird. Als Beispiel bietet sich hier die Klasse CResourceManager an, welche den für Grafiken verwendeten Textur-Speicher verwaltet. Diese Klasse muss allen anderen Klassen zur Verfügung stehen, welche Grafiken laden wollen. Dabei genügte es eben nicht, jedesmal eine eigene Instanz der Klasse CResoureManager zu erzeugen, da diese Klasse ja für die optimale Speichernutzung überprüft, ob eine Grafik bereits im Speicher vorhanden ist. Wenn jetzt mehrere Instanzen derselben Klasse bestehen kann es leicht vorkommen, dass ein und dieselbe Grafik mehrere Male geladen wird. Somit muss also exakt dieselbe Instanz überall verwendet werden.

Mein erster Ansatz war hier, zuerst eine Variable vom Typ CResoureManager in der jeweiligen Klasse zu deklarieren, und dann mit einer Methode die Instanz von CResoureManager zu übergeben.

*//in Tileset.h*

*class CTileset:*

*{*

 *public:*

 *void SetResourceManager(CResoureManager ResoureManager);*

 *private:*

 *CResoureManager m\_ResourceManager;*

*}*

*//in Tileset.cpp*

*void CTileset::SetResoureManager(CResoureManager ResoureManager)*

*{*

 *m\_ResourceManager = ResourceManager;*

*}*

In diesem Beispiel wird der Ressourcen-Manager an die Klasse für das Tileset übergeben. An sich funktionierte diese Lösung ziemlich gut. Jedoch merkte ich erst mit zunehmender Komplexität des Codes, das sich hier der Fehler auftat, den ich zu vermeiden versuchte: m\_ResourceManager teilte sich nicht die Variablen mit der übergebenen Instanz.

Lösung:

Nach einigen Überlegungen und Recherchen stellte sich heraus, was ich vorher nicht gewusst hatte: Beim Übergeben einer Variablen an eine Methode wird eine Kopie der Variablen erstellt. Diese Kopien sind zwar anfangs ident, verweisen aber nicht auf denselben Speicherbereich und sind daher wie eigenständige Instanzen. Das bedeutet, dass die einzige Möglichkeit ist, Pointer zu verwenden. Statt direkt die Instanz von CResourceManager zu übergeben, würde ich einen Zeiger auf den Speicherbereich dieser Instanz als Parameter angeben und diesen speichern.

*CResourceManager \*m\_ResourceManager;*

Damit tat sich aber noch ein Problem auf. Denn ein in einer Klasse gespeicherter Zeiger muss bereits bei der Erstellung einer Instanz der Klasse einen Speicherbereich zugewiesen bekommen. Dies geschieht dadurch, dass man dem Konstruktor einen Parameter vom Typ des Zeigers übergibt. Dieser Zeiger kann dann aber nicht normal zugewiesen werden, sondern muss über die *construktor initialisation list* gespeichert werden. Der folgende Code zeigt die Lösung:

*//in Tileset.h*

*class CTileset:*

*{*

 *public:*

 *CTileSet(CResourceManager &ResourceManager);*

 *private:*

 *CResoureManager \*m\_ResourceManager;*

*}*

*//in Tileset.cpp*

*void CTileset::CTileset(CResourceManager &ResourceManager)*

 *: m\_ResourceManager(&ResourceManager)*

*{*

*}*

Diesen Lösungsansatz muss ich nun jedesmal verwenden, wenn die Instanz einer Klasse geteilt verwendet werden muss. Bis auf den zusätzlichen Aufwand bei der Deklaration des Konstruktors tun sich dadurch aber keinerlei Nachteile auf.

# 5. Schlusswort:

Zum Abschluss möchte ich noch einmal die wichtigsten Erkenntnisse dieser Arbeit zusammenfassen:

Die Spieleprogrammierung ist eines der komplexesten und schwierigsten Themen der Computer-Programmierung. Die Erstellung eines Spiels benötigt das Wissen über viele Teilbereiche der Informatik und legt hohe Ansprüche an die Qualitäten des Programmierers. Bei der Umsetzung spielt die objektorientierte Programmierung eine große Rolle. Die Verwendung und strikte Umsetzung von objektorientierten Konzepten hilft, die Übersicht beim Programmieren zu behalten.

Durch die Verwendung von Klassen zur Abkapslung von Funktionalität kann man ein überschaubares und funktionsfähiges System erstellen, welches man auch leicht erweitern kann.

Weiters ist mir beim Erstellen meines Spiels aufgefallen, dass anfangs viele Konzepte, auch wenn sie noch so logisch erscheinen, nicht umgesetzt werden können. Manchmal konnte ich einen Programmcode nicht zum Laufen bekommen, obwohl mir selbst beim mehrmaligen Überdenken kein Fehler aufgefallen ist. Auch taten sich manchmal erst nach der Zeit Probleme auf, mit denen ich nicht gerechnet hatte. Doch je mehr Erfahrung man über das Programmieren sammelt und je mehr man sich an Programmparadigmen (wie die objektorientierte Programmierung) hält, desto mehr lernt man, auch auf den ersten Anhieb hin seine Konzepte richtig umzusetzen. Je mehr man programmiert, desto besser kann man auch seine Konzepte auf die Möglichkeiten im Code abzustimmen.

Somit habe ich eine Menge über die Spieleprogrammierung, sowohl theoretisch als auch praktisch gelernt. Es sind für mich keine Fragen offen geblieben.

# Quellenverzeichnis:

**Literaturverzeichnis:**

KASTENS, Uwe. Programmierparadigma. In:

<http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/technologien-methoden/Sprache/Programmiersprache/Paradigma>. 06.01.2011

LOUIS, Dirk. easy C++. München 2001

N.N. Kriterien für die Qualität von Software. In: <http://www.tse.de/papiere/ergonomie/Softwarekriterien.html>. 06.01.2011.

N.N. Spieleprogrammierung. In: <http://de.wikipedia.org/wiki/Spieleprogrammierung>.

21.12.2010a.

N.N. Game programming. In: <http://en.wikipedia.org/wiki/Game_programming>. 05.01.2011b.

ROUSSELLE, Christian. Spieleprogrammierung mit DirectX. München 2006.

STEIN, Daniel. Übersicht über die grundlegenden C++-Elemente. In: <http://www-i6.informatik.rwth-aachen.de/web/Teaching/LabCourses/SS08/Softwarepraktikum/aufgaben/folien_c_kurs.pdf>.14.01.2010.

**Abbildungsverzeichnis:**

[Abbildung 1: Einfache Spieleschleife 7](#_Toc285557462)

[Abbildung 2: Komplexe Spielschleife meines Spiels 7](#_Toc285557463)

[Abbildung 3: Ein Ausschnitt eines Levels des technisch fertigen Spiels. 11](#_Toc285557464)

[Abbildung 4: Darstellung eines Baums als Objekt 25](#_Toc285557465)

[Abbildung 5: Komplexe Darstellung der Game-Loop sowie Funktion meiner Engine mit den verwendeten Klassen 39](#_Toc285557466)

[Abbildung 6: Schritt 1-3 der Kollisionserkennung – rot: Gelände , gelb: Spieler 43](#_Toc285557467)

[Abbildung 7: Ein unfertiges Tileset 44](#_Toc285557468)

**Erklärung**

Ich, Julian Watzinger, erkläre hiermit, dass ich für meine Arbeit keine anderen als die von mir angeführten Quellen und Hilfsmittel benutzt habe.

Elsbethen, den 16. Februar 2011