

# Design und Umsetzung eines Spiels mit einer Multimedia-Bibliothek

---

**Autor:** Wilhelm Oks  
**Matrikelnummer:** 167447  
**Datum:** 22.06.2010

## Inhaltsverzeichnis

Abbildungsverzeichnis.....	3
Abkürzungsverzeichnis.....	3
Vorwort .....	4
1 Das Spielkonzept .....	5
1.1 Spielverlauf.....	5
1.2 Waffen.....	5
1.3 Steuerung .....	5
1.4 Konfiguration.....	6
2 Die Bibliotheken .....	7
2.1 Allegro .....	7
2.2 Fmod.....	7
3 Design und Umsetzung.....	8
3.1 Bibliothek-spezifische Implementierung.....	8
3.2 Grafik .....	8
3.3 Ton.....	9
3.4 Eingabe .....	9
3.5 Konfiguration.....	10
3.6 Tick und Draw.....	11
3.7 Zeit.....	11
3.8 Die Waffenklassen.....	12
3.9 Das Menüsystem .....	13
3.9.1 Allgemein.....	13
3.9.2 Aufbau und Verwendung .....	13
3.10 Utility Dateien .....	14
Quellenverzeichnis .....	16
Erklärung des Autors .....	16

## Abbildungsverzeichnis

Abbildung 1: Fahrzeugsteuerung .....	6
Abbildung 2: Implementierungsklassen .....	8
Abbildung 3: Aktualisierung mit Tick und Draw .....	11
Abbildung 4: Waffenklassen.....	13
Abbildung 5: Menü Klassen.....	14

## Abkürzungsverzeichnis

WAV	WAVE – Audio Dateiformat
VOC	Creative Voice – Audio Dateiformat
MIDI	Musical Instrument Digital Interface – Dateiformat und Protokoll für Musiknoten
GUI	Graphical User Interface – Grafische Benutzeroberfläche

## Vorwort

Dieses Dokument beschreibt das Design eines Computerspiels und wie es mit einer Multimedia-Bibliothek umgesetzt wurde.

Eines der wichtigsten Ziele war dabei, dass der Code möglichst unabhängig ist. Das bezieht sich in erster Linie auf die Austauschbarkeit der Multimedia-Bibliothek und in zweiter Linie auf die Wiederverwendbarkeit von Klassen und anderen Strukturen bzw. Routinen, sodass sie ohne Änderungen in andere Spiele übernommen werden können.

Ein weiteres Ziel war es, gut strukturierten und leicht erweiterbaren Code zu schreiben, damit neue Features hinzugefügt werden können.

Obwohl der Schwerpunkt auf das eigentliche Spiel-Design gelegt wurde, wird teilweise auch auf Details der Bibliotheken eingegangen um zu zeigen, wie etwas Umgesetzt wurde und warum.

Das erste Kapitel soll eine grobe Vorstellung von der Spielidee schaffen, indem es die grundlegenden Spielelemente, Eingabemöglichkeiten und das Ziel des Spiels beschreibt.

Im nächsten Kapitel werden die eingesetzten Bibliotheken aufgeführt und im dritten und letzten Kapitel geht es um das Design und die Umsetzung des Spiels.

# 1 Das Spielkonzept

Am ehesten lässt sich das Spiel mit „Snake“ oder „Tron“ vergleichen. Der Spieler steuert ein Fahrzeug (dargestellt durch eine Pfeilspitze) auf einer zweidimensionalen Spielfläche, das sich ständig in eine Richtung fortbewegt und dabei eine Spur hinterlässt. Die Richtung, in die es sich fortbewegt, kann geändert werden und es gilt zu verhindern, dass dieses Fahrzeug mit der Spur oder dem Rand der Spielfläche kollidiert.

## 1.1 Spielverlauf

Es spielen gleichzeitig 2 bis 4 Spieler auf demselben Spielfeld gegeneinander. Jeder Spieler versucht dabei sein Fahrzeug so lange wie möglich vor der Kollision zu bewahren.

Nach jeder Kollision werden dem jeweiligen Spieler Punkte gutgeschrieben und er scheidet aus dieser Runde aus. Wenn nur noch ein Spieler übrig ist, ist die Runde vorbei. Es wird ein Punktezweischenstand ausgegeben und eine neue Runde kann gestartet werden.

## 1.2 Waffen

Das Spiel bietet unterschiedliche Waffen, die aufgesammelt und gegen andere Spieler benutzt werden können:

- Wand: Senkrecht zur Fahrtrichtung eine Wand platzieren.
- Bombe: Kollisionspunkte in einem kreisförmigen Bereich entfernen
- Anhalten: Einen anderen Spieler für eine bestimmte Zeitspanne anhalten.
- Splitter: Aussenden von Splittern, die mit anderen Spielern kollidieren können.
- Verfolgung: Aussenden eines Objekts, das sich auf einen anderen Spieler zu bewegt und sich bei Kollision selbst zerstört.

Zum Beginn werden zufällig ausgewählte Waffen auf zufällige Positionen im Spielfeld platziert.

Die Waffen werden durch ihre Icons repräsentiert. Wenn ein Fahrzeug über das Icon einer Waffe fährt, wird diese Waffe in das Inventar des Fahrzeugs aufgenommen und verschwindet aus dem Spielfeld. Das Inventar bietet Platz für eine bestimmte Anzahl an Waffen. Die Waffe im ersten Inventarslot entspricht der aktuell selektierten Waffe.

## 1.3 Steuerung

Jeder Spieler hat die Möglichkeit, das Fahrzeug um einen bestimmten Winkel nach links oder nach rechts zu drehen, damit es seine Fahrtrichtung ändert.

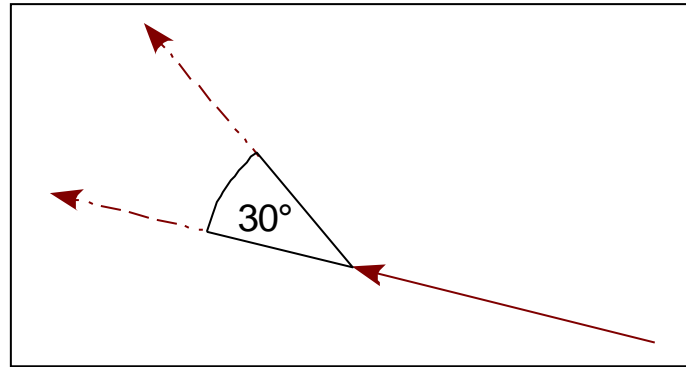


Abbildung 1: Fahrzeugsteuerung

Weiterhin kann jeder Spieler eine Waffe aus dem Inventar des Fahrzeugs aktivieren oder selektieren.

## 1.4 Konfiguration

Die Spielkonfiguration befindet sich der Datei „config.ini“ und kann zum Teil über das Optionsmenü im Spiel geändert werden. Die Konfigurationsmöglichkeiten sind:

- Vollbild/Fenstermodus
- Pixel Auflösung
- Anzahl Waffen auf Spielfeld
- Linienstärke (z.B. für Fahrzeugspuren)
- Und für je einen von 4 Spielerslots:
  - Tasten für Lenkung links/rechts und Waffe aktivieren/selektieren
  - Spielerfarbe
  - Spielertyp: menschlicher Spieler, Computerspieler Leicht/Normal/Schwer oder gar kein Spieler.

## 2 Die Bibliotheken

### 2.1 Allegro

Allegro ist eine Low Level Bibliothek für Spiele und Multimedia Anwendungen. Der Code ist in der Programmiersprache C geschrieben, plattformübergreifend und open source.

Der grobe Funktionsumfang besteht aus

- Grafikausgabe, eigene GUI
- Tonausgabe
- Benutzereingaben
- Timer
- Konfigurationsdateien

und deckt damit das Meiste ab, was für dieses Spiel erforderlich ist.

### 2.2 Fmod

Allegro kann zwar Audiodateien wiedergeben, aber nur solche, die unkomprimiert im Format WAV oder VOC vorliegen. Diese eignen sich für kurze Soundeffekte, die entweder in wiederholt (Bsp.: Automotor), oder einzeln (Bsp.: Gegenstand Fällt auf den Boden) abgespielt werden. Für Hintergrundmusik jedoch wären solche Formate ungeeignet, weil die Dateien zu viel Platz verbrauchen würden.

Allegro bietet noch speziell für Musik die Möglichkeit zum Abspielen von MIDI Dateien. Diese sind zwar sehr platzsparend, aber sie aufgrund von ihrem notenbasierten Format eignen sie sich nicht für jede Art von Klang.

Deswegen wird für dieses Projekt zusätzlich zu Allegro noch die Bibliothek Fmod benutzt.

Damit lassen sich komprimierte Audiodateien, insbesondere im Format MP3 abspielen. Die Verwendung von Fmod ist für nicht kommerzielle Projekte kostenlos.

### 3 Design und Umsetzung

Obwohl Allegro eine reine C Bibliothek ist, wurde das Spiel in der Sprache C++ geschrieben, da sie sich durch ihre objektorientierte Erweiterung gut dafür eignet, einzelne Spielelemente als Klassen zu modellieren.

#### 3.1 Bibliothek-spezifische Implementierung

Diejenigen Klassen, die eine Bibliothek benötigen, um beispielsweise die grafische Darstellung umzusetzen, sind abstrakt und werden von einer Bibliothek spezifischen Unterklasse implementiert (Abbildung 2). Jede dieser Implementierungsklassen befindet sich in derselben Datei wie die abstrakte Klasse.

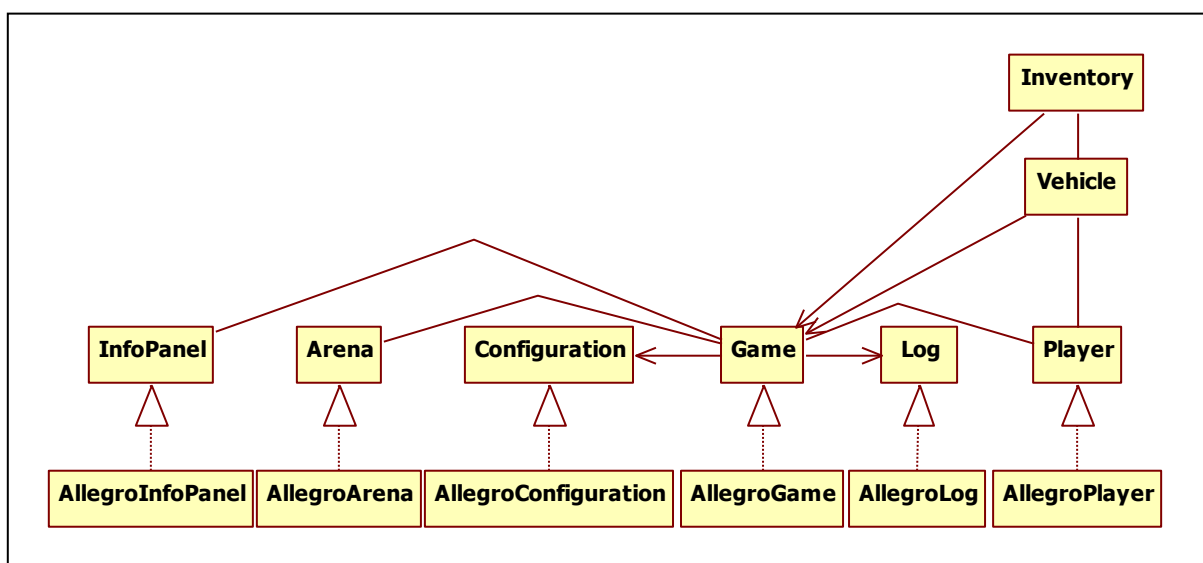


Abbildung 2: Implementierungsklassen

Diese Struktur soll es dem Entwickler erleichtern, das Spiel mit einer anderen Bibliothek umzusetzen. Die Kombination von Allegro und Fmod wird in diesem Projekt als eine Allegro Implementierung gesehen, weil Fmod die Funktionalität von Allegro lediglich erweitert und nicht ersetzt.

#### 3.2 Grafik

Für die grafische Ausgabe stellt Allegro die Struktur *BITMAP* bereit. Man kann sie als Datenstruktur für geladene Bilddateien verwenden und auf sie zeichnen. Die Bildfläche, die sich dem Benutzer in einem Fenster oder im Vollbildmodus zeigt, ist ebenfalls eine *BITMAP* und ist von Allegro global als *screen* definiert. Das Spiel definiert noch eine *screenBuffer BITMAP* und verwendet dann folgende Schritte für die Darstellung am Bildschirm:

- *screenBuffer* leeren (alle Pixel schwarz färben)
- Spielelemente auf *screenBuffer* zeichnen



- *screenBuffer* auf *screen* zeichnen

Das hat den Vorteil, dass das Bild nicht flackert, weil die Spielelemente nicht einzeln nacheinander, sondern als Ganzes auf dem Bildschirm auftauchen und weil das Leeren des *screenBuffers* als Schritt nicht sichtbar ist.

Das Spielfeld verwendet darüber hinaus in der Implementierungsklasse *AllegroArena* die *BITMAP* Struktur auch für den logischen Zustand. Das Fahrzeug hinterlässt eine Spur aus Pixeln, die für die Kollisionsprüfung herangezogen werden. Die abstrakte Oberklasse *Arena* ist jedoch so allgemein gehalten, dass eine andere Bibliothek stattdessen auch Vektordaten verwenden könnte. Die Pixeldarstellung bietet sich jedoch an, weil dadurch beliebige Pixelbilder als Kollisionsobjekte verwendet werden können.

### 3.3 Ton

Das Spiel unterscheidet bei der Tonausgabe zwischen kurzen Audioeffekten, die von *Allegro* abgespielt werden und längeren Musikstücken, die von *Fmod* abgespielt werden.

Die Auswahl der Musikstücke ist in diesem Spiel auf eins begrenzt, aber das Design sieht die Möglichkeit vor, dass es mehr sein könnten. Eine Erweiterung auf zwei Musikstücke hätte also minimalen Aufwand zur Folge:

```
enum MUSIC {  
    MUSIC_MAIN,  
    MUSIC_VICTORY, //neu  
  
    MUSIC_COUNT  
};  
  
//...  
  
_musicPath[MUSIC_MAIN] = "music/main.mp3";  
_musicPath[MUSIC_VICTORY] = "music/victory.mp3"; //neu  
  
//...  
  
PlayMusic(MUSIC_VICTORY);
```

### 3.4 Eingabe

Benutzereingaben werden in diesem Spiel über Tastatur und Maus getätigt. *Allegro* hat auch Joystick Unterstützung aber diese wird hier nicht in Anspruch genommen.

Es gibt zwei Möglichkeiten mit *Allegro* Tastatur- und Mauseingaben auszuwerten: Mit Callbacks oder durch Polling.

Die erstere Möglichkeit hat den Vorteil, dass immer nur Zustandsänderungen des Eingabegeräts mitgeteilt werden. Aber der Nachteil ist, dass dafür eine globale Callback Funktion angelegt werden muss, die aufgerufen wird, sobald sich ein Zustand geändert hat. Und diese globale Funktion passt nicht in das objektorientierte Design.

Deswegen wird in diesem Projekt die zweite Möglichkeit verwendet. Allegro hält die Tastenzustände für die Tastatur in dem globalen Array *key* und die Mauskoordinaten in den globalen Variablen *mouse\_x* und *mouse\_y*. Der Zustand von Tastatur und Maus wird von dem Spiel in jedem Tick neu abgefragt und ergibt entweder gedrückt oder nicht gedrückt bzw. die Werte für die Mausposition.

Das Spielkonzept erfordert jedoch nur eine Mitteilung über den Zustandswechsel von nicht gedrückt zu gedrückt. Das bedeutet, dass zusätzlich festgehalten werden muss was der Zustand einer Taste im letzten Tick gewesen ist.

Die Klasse *Player* setzt dies für die Steuerung des Fahrzeugs um und definiert eine Methode, die nur dann *true* zurückgibt, wenn ein Zustandswechsel stattgefunden hat.

```
bool AllegroPlayer::GetKeyImpulseForCommand(PYER_COMMAND command) {
    int keyForCommand = GetKeyForCommand(command);
    if (!_holdingKeyForCommand[command]) {
        if (key[keyForCommand]) {
            _holdingKeyForCommand[command] = true;
            return true;
        } else {
            _holdingKeyForCommand[command] = false;
            return false;
        }
    } else {
        if (key[keyForCommand]) {
            _holdingKeyForCommand[command] = true;
            return false;
        } else {
            _holdingKeyForCommand[command] = false;
            return false;
        }
    }
}
```

### 3.5 Konfiguration

Für die Konfiguration des Spiels existiert die Klasse *Configuration*, die die Konfigurationsdaten abkapselt. Die Unterklasse *AllegroConfiguration* (Abbildung 2) implementiert die abstrakten Methoden *Save* und *Load* und verwendet dazu die Allegro Routinen für Konfigurationsdateien.

Auf diese Weise ist es Möglich zu einer anderen Implementierung zu wechseln, ohne den Rest des Codes anzufassen. Zum Beispiel wenn man die Konfiguration lieber im XML Format haben möchte.

### 3.6 Tick und Draw

Da gleich mehrere Klassen den Spielfluss abbilden, müssen sie sich stets zum richtigen Zeitpunkt aktualisieren und grafisch darstellen. Aus diesem Grund besitzen diese Klassen je eine *Tick* und eine *Draw* Methode.

Tick aktualisiert den Zustand des Objekts während Draw für die grafische Darstellung sorgt.

Dabei gilt die Vereinbarung, dass ein Objekt von seinem Besitzer aktualisiert wird, indem der Besitzer während der eigenen Aktualisierung innerhalb seiner Tick bzw. Draw Methode auch die jeweilige Aktualisierungsmethode des besessenen Objekts aufruft.

Zum Beispiel besitzt ein Spieler ein Fahrzeug, das wiederum ein Inventar besitzt. Das Inventar wird also vom Fahrzeug aktualisiert und das Fahrzeug vom Spieler.

Abbildung 3 zeigt die Aktualisierungsketten der Spielfluss Klassen wobei die Pfeilspitzen auf die Klasse zeigen, dessen Aktualisierungsmethode aufgerufen wird. Die Game Klasse bildet dabei den Ursprung.

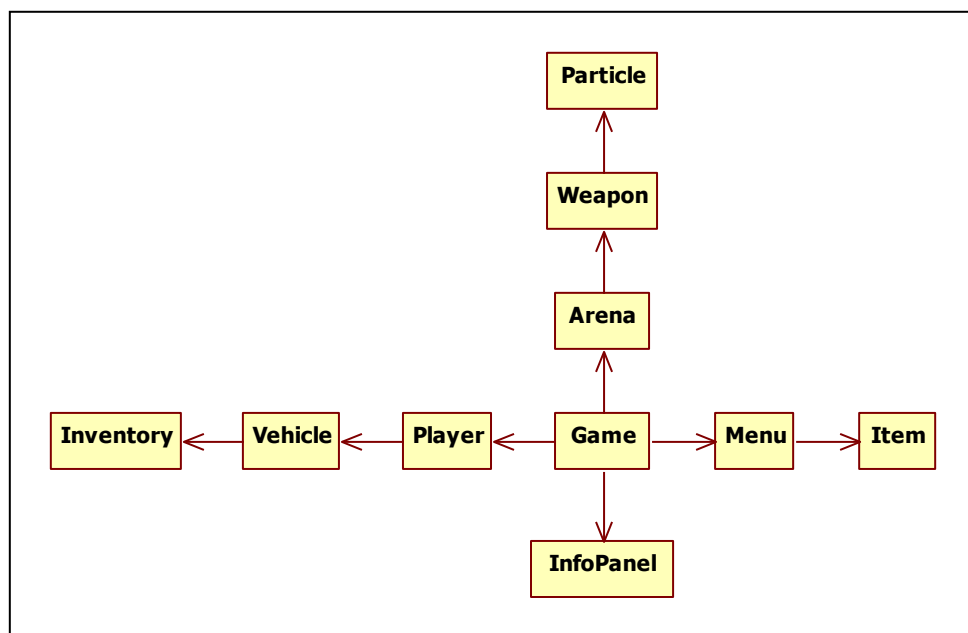


Abbildung 3: Aktualisierung mit Tick und Draw

### 3.7 Zeit

Mit der Game Klasse als Ursprung aller Aktualisierungsketten ist offensichtlich, dass alle zeitlichen Vorgänge und damit die Spielgeschwindigkeit, darunter auch die Bewegung der Fahrzeuge, davon abhängig ist, wie groß der Zeitintervall zwischen den Aufrufen der Aktualisierungsmethoden ist. Draw ist nur für die Darstellung zuständig und soll keinen Einfluss

auf die Spielgeschwindigkeit haben. Es gilt also dafür zu sorgen, dass die Tick Methode in regelmäßigen Zeitschritten aufgerufen wird. Mit Allegro lassen sich für diesen Zweck Timer einrichten.

```
volatile int ticks_to_do = 0;

void increment_ticks_to_do() {
    ticks_to_do++;
}
END_OF_FUNCTION(increment_ticks_to_do)

//...

LOCK_VARIABLE(ticks_to_do);
LOCK_FUNCTION(increment_ticks_to_do);
install_int_ex(increment_ticks_to_do, BPS_TO_TIMER(100));
```

Nach dieser Definition wird die *increment\_ticks\_to\_do* Funktion 100 Mal pro Sekunde aufgerufen.

Darin wird eine Variable hochgezählt, die für die Anzahl der notwendigen Aufrufe der Tick Methode steht. Danach lässt sich der gesamte Spielfluss mit der folgenden Endlosschleife realisieren:

```
while (true) {
    while (ticks_to_do > 0) {
        Tick();
        ticks_to_do--;
    }
    Draw();
}
```

Im Normalfall wird die meiste Zeit nur die Grafik aktualisiert. Wenn die Zeit für einen Tick gekommen ist, wird die Variable *ticks\_to\_do* größer als 0 und die Ticks werden durchgeführt. Falls die Grafik mehr Zeit braucht, als die Zeit, die zwischen den Ticks liegt, werden beim nächsten Beginn der Endlosschleife alle Ticks nachgeholt und erst dann wieder die Grafik berechnet. Dadurch bleibt die Spielgeschwindigkeit auch bei leistungsschwächeren Rechnern relativ konstant.

### 3.8 Die Waffenklassen

Damit neue Waffenklassen schnell und ohne viel Aufwand hinzugefügt werden können, teilen sich die Klassen für die 5 konkreten Waffen (im Ordner „Weapons/Impl“) die abstrakte Basisklasse *Weapon*. Jede konkrete Waffe besteht daher aus einer relativ kleinen Klasse, die

gleich vollständig in der Headerdatei definiert wird. Dadurch, dass die anderen Klassen nur die Waffen nur über *Weapon* ansprechen, wird jede Waffe gleich behandelt (Abbildung 4).

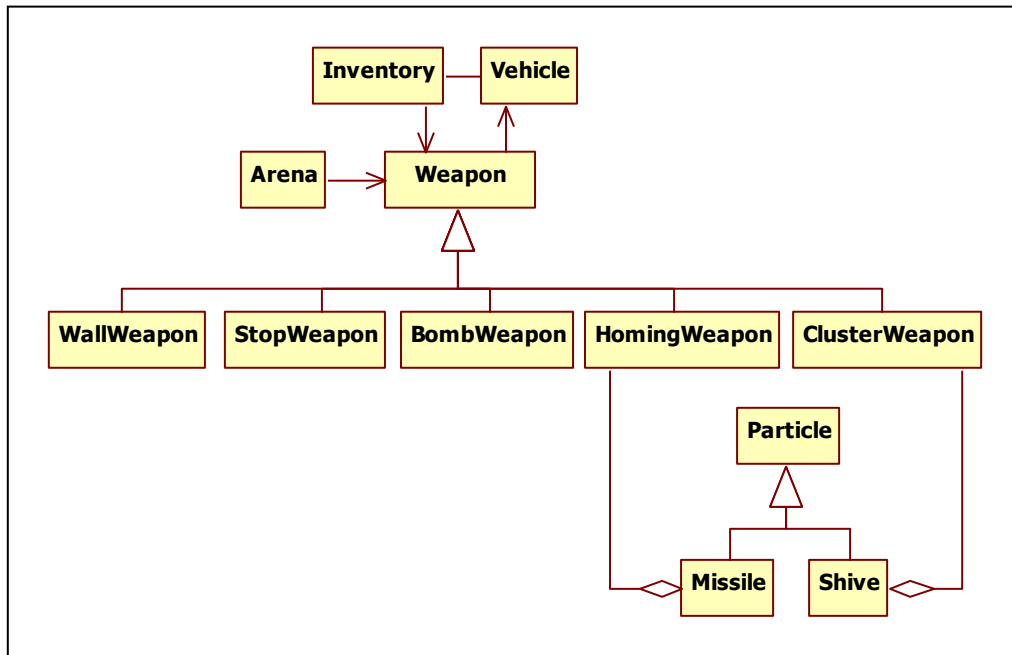


Abbildung 4: Waffenklassen

Um die Waffenobjekte zur Laufzeit dennoch unterscheiden zu können, um etwa die richtigen Icons darzustellen, wird in „Weapons/WeaponTypes.h“ der Aufzählungstyp *WEAPON\_TYPE* definiert und in *Weapon* als Membervariable verwendet.

Die Waffenklassen erfordern keine Implementierung durch eine Bibliothek.

## 3.9 Das Menüsystem

### 3.9.1 Allgemein

Das Menüsystem, das sich im Ordner „Menus/Imenu“ befindet, ist mausgesteuert und ist so gestaltet, dass es von diesem Spiel unabhängig ist, damit es auch in weiteren Spielen benutzt werden kann. Allerdings basiert es auf Allegro und existiert ausschließlich in den speziellen Allegro Implementierungsklassen. Eine andere Bibliothek müsste also ein eigenes Menüsystem umsetzen.

### 3.9.2 Aufbau und Verwendung

Ein konkretes Menü erbt von der Basisklasse *Imenu::Menu* und enthält Items, die wiederum Events an das Menü schicken. Ein Event enthält die Information, von welchem Item es geschickt wurde und um welche Art von Ereignis es sich handelt (z.B. Item wurde angeklickt).

Das Menüsystem aktualisiert sich ebenfalls durch Aufrufe von Tick und Draw.

Dieses Spiel implementiert ein *BaseMenu* und ein *BaseItem* als spielspezifische Basisklassen. Diese Klassen legen die Eigenschaften (Farbe, Schriftart) und das Verhalten (Abspielen eines Tons beim Klicken) für alle anderen konkreten Menüs innerhalb des Spiels wie beispielsweise das Optionsmenü fest.

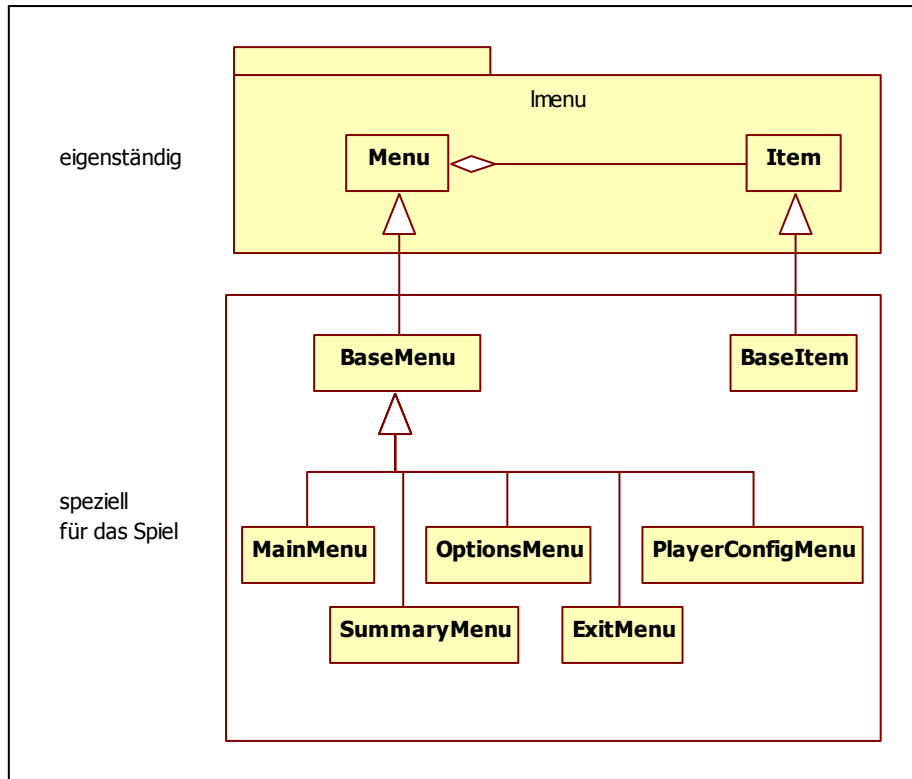


Abbildung 5: Menü Klassen

### 3.10 Utility Dateien

Im Ordner *Util* befinden sich Dateien mit Klassen und Funktionen, die generell überall gebraucht werden. Sie sind wie das Menüsystem auch eigenständig und können ohne Änderungen in anderen Projekten benutzt werden.

- Klasse *Log* und die Unterklasse *AllegroLog*:  
Dient der Ausgabe von unterschiedlich kategorisierten Textnachrichten auf Konsole und in eine Datei. Die Allegro Spezialisierung gibt die Nachrichten auch in einem kleinen Dialogfenster aus.
- Klasse *Vector2D*:  
Enthält viele praktische mathematische Operationen für zweidimensionale vektorielle Größen. Ist außerdem spezialisiert auf Daten, deren logische Größe mit der dargestellten Pixelgröße übereinstimmt. Zum Beispiel wird für Punktgleichheit überprüft, ob die Koordinaten auf das selbe Pixel fallen würden.

Mit Ausnahme der Menüs, wird diese Klasse für jede Größe verwendet, die eine Längen- und eine Breitenkomponente hat. Beispielsweise für Positionen und rechteckige Größen.

- Datei „Colors.h“:

Verwaltet vordefinierte Farben für die Spieler und die Menüs. Stellt außerdem eine Funktion zur Verdunkelung einer Farbe bereit.

- Datei „blitfill.h“:

Stellt die Funktion *blitfill* bereit, die an Allegros Funktion *blit* angelehnt ist.

Damit wird ein Bild sowohl in die Länge als auch in die Höhe so oft gestapelt gezeichnet, bis eine spezifizierte Fläche damit ausgefüllt ist.

## Quellenverzeichnis

*Allegro Manual*. (2010). Von allegro.cc: <http://www.allegro.cc/manual/> abgerufen

## Erklärung des Autors

Ich versichere, dass ich diese Arbeit selbständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst habe.

Kirchheim am Neckar, den 22.06.2010

---

Ort, Datum

Wilhelm Oks

---

Unterschrift