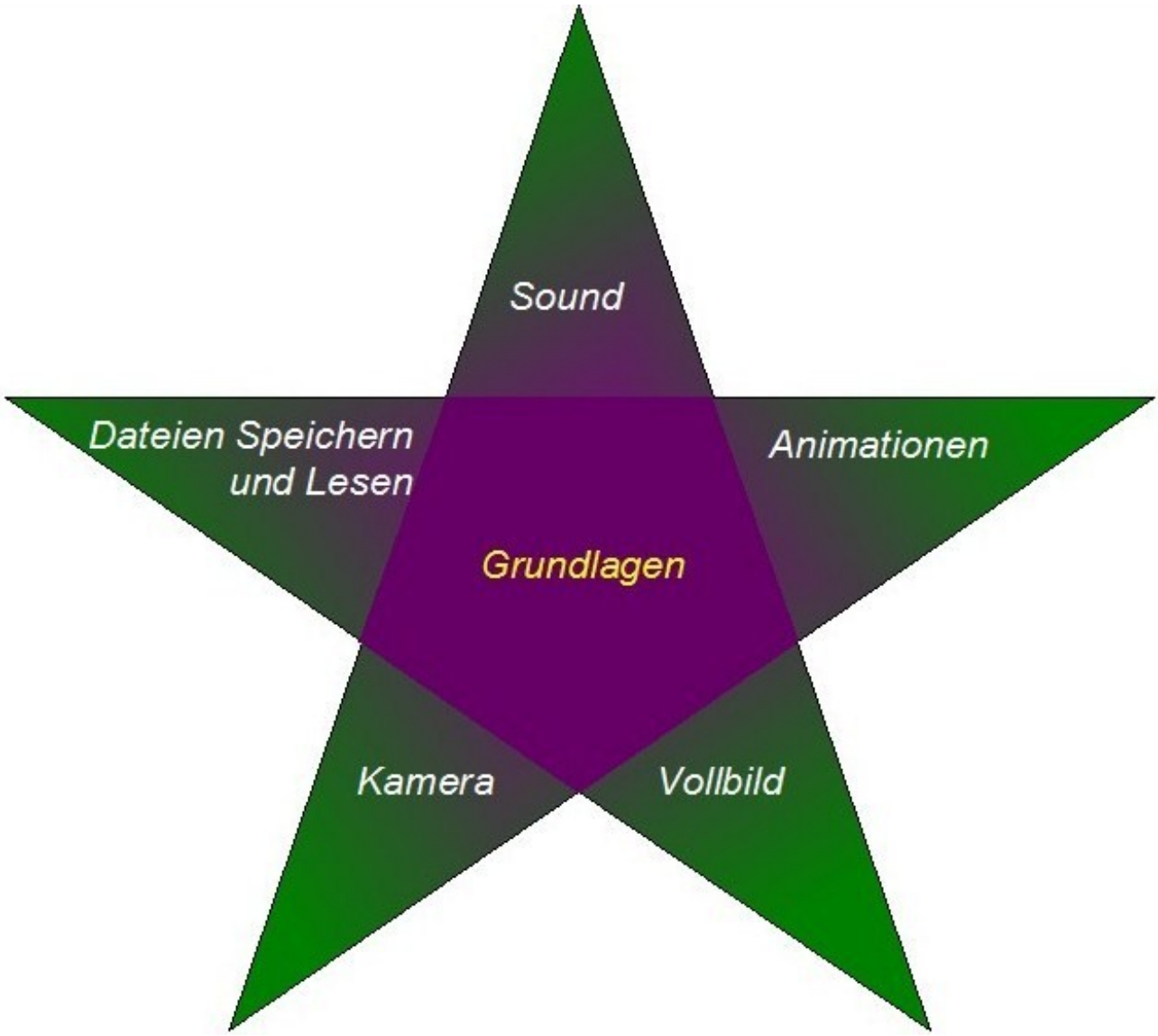
# ENGINE ALPHA

*-Handbuch-*

Version 2.0:
3. April 2011




*Michael Andonie*

# Inhaltsverzeichnis


I Vorwort....4
II Voraussetzungen für die Arbeit mit der Engine....5
III Einbringen der Engine....7
IV Programmieren mit der Engine Alpha – Die Grundlagen....9
A Die spielsteuernde Klasse....9
B Das Grafiksystem....12
1 Die Zeichenebene....12
2 Die Klassen der Zeichenebene....13
i Die Klasse Punkt....13
ii Die Klasse Vektor....14
iii Die Klasse BoundigRechteck....15
C Die Klasse Raum....17
1 Bilder....19
2 Texte....20
3 Geometrische Figuren....22
D Die Klasse Knoten....23
1 Definition der Klasse....23
2 Objekte werden sichtbar....25
3 Eine kleine Praxisübung....28
4 Ein weiterer Nutzen von Knoten: Vorder- und Hintergrund....32
V Optionale Funktionen....33
A Ein kurzes Vorwort....33
B Das Ticker-System (Multitasking)....34
1 Das Ticker-Interface....34
2 Den Ticker „zum Laufen bringen“....35
C Figuren....37
1 Erstellen einer Figur....38
2 Laden einer Figur und deren Einbringung ins Spiel....39
3 Figuren modifizieren....41
i Die Größe von Figuren ändern....41
ii Die Figur spiegeln....41
iii Farben der Figur ändern....42
iv Das geht alles: Zusammenfassung....43
D Sound....44
1Unterstützte Sound-Formate....44
2 Einbringen einer Sound-Datei....45
3 Sound-Dateien in Ordnern....47
4 Sounddateien im exportierten Projekt....47
E Die Kamera....48
1Was ist die Kamera?....48
2 Verschieben der Kamera....49
3 Fokus....51
4 Grenzen für die Kamera....54
5 Es muss sich ja nicht alles verschieben: Statische grafische Objekte....55
F Dateien Schreiben und Lesen....56
1 Dateien Schreiben....56
2 Dateien wieder Einlesen....57

3 Arbeiten mit Verzeichnissen....58
4 Laden und Speichern bei einem exportierten Projekt (.jar-Datei)....59
G Animationen....60
1 Animationsarten....60
2 Kreisanimationen....62
3 Streckenanimationen....63
4 Geradenanimationen....65
H Reagieren auf Aktionen des Spielers....67
1 Tastatur-Interfaces....69
i Das Interface TastenReagierbar....69
ii Das Interface TastenLosgelassenReagierbar....70
iii Das Interface TastenGedrueckReagierbar....71
iv Ein Beispiel für Alle Tastatur-Interfaces:....72
2 Die Maus....73
i Das ist für eine Maus alles nötig....73
ii Der Konstruktor der Klasse Maus....75
iii Einbringen einer Maus: Ein Beispiel....76
iv Reagieren auf Mausklicks....76
Das Interface KlickReagierbar....77
Anwendungsbeispiel für KlickReagierbar....78
Das Interface RechtsKlickReagierbar....79
Das Interface MausReagierbar....80
Anwendungsbeispiel für MausReagierbar....82
I Farben....84
1 Farben als Texte....84
2 Farben über R/G/B-Werte....85
3 Alpha-Werte....86
J Die Engine-Alpha-„Physik“....87
1 Die Idee....87
2 Die grundlegende Verwendung....88
3 Die Funktionen der „Physik“....90
i Bewegungen....90
Die Methode bewegen(...)....90
Erfolgreiche und nicht erfolgreiche Bewegungen....93
ii Springen....94
Die Methode sprung(...)....94
Erfolgreiche und nicht erfolgreiche Sprünge....95
iii Wie tief kann man fallen?....96
iv Das geht alles: Zusammenfassung....98
K Die Klasse Game unter der Lupe....100
1 Attribute....100
2 Konstruktoren....101
3 Methoden....101
L Vollbildmodus....104
M Das Projekt exportieren....106
1 Die Exportschritte bei BlueJ....106
2 Zusätzliche Dateien im Spiel....108
VI Schlusswort....110
VII Anhang....111

# I Vorwort

Du *willst* ein eigenes Spiel programmieren, bist aber nicht in die Feinheiten der Grafikprogrammierung eingewiesen, sondern möchtest nur Deine Planung des Spiels umsetzen?
Du hast Durchhaltevermögen, und bist bereit Zeit in Deine Wünsche zu investieren?

Dann ist die Engine Alpha richtig für Dich. Mit ihr lassen sich professionell wirkende 2D-Computerspiele entwickeln, unter der Voraussetzung, dass man das Informatikwissen der 10. Klasse des bayerischen naturwissenschaftlich-technologischen Gymnasiums oder das eines Hobbyprogrammierers in der Sprache **Java** hat.

Sie ist einfach zu benutzen, und mit einem Minimum an Funktionswissen dieser Engine lassen sich bereits vollständig interaktive Zusammenhänge programmieren, um möglichst schnell ein Spiel aufstellen zu können. Gleichzeitig bietet sich jedoch die Möglichkeit, tiefer in die Engine einzusteigen, dort wo Du es brauchst (Animationen, Soundwiedergabe, Dateien Speichern und Lesen etc.).
Daher wird zunächst erklärt, was Du wissen *musst*, um die Engine bedienen zu können, und im Folgenden stehen einzelne Kapitel zur Verfügung, mit denen Du für Dein Spiel eigene Akzente setzen kannst. Jedes Kapitel behandelt ein eigenes Highlight.

Um eines vorher noch klarzustellen: *Die Engine Alpha denkt nicht für Dich*. Du musst selber arbeiten, damit Dein Spiel funktioniert, aber die Engine bietet Dir eine leicht beherrschbare Grundlage hierfür.

Nun genug der Vorworte, wenn Du Lust auf Deine eigenen Computerspiele bekommen hast, fangen wir am besten gleich an.

# II Voraussetzungen für die Arbeit mit der Engine

Zunächst liste ich auf, was Du unbedingt an Grundwissen kennen solltest, um sowohl die Engine bedienen als auch um eine gute Struktur in Dein eigenes Spiel bringen zu können. Solltest Du etwas nicht beherrschen: keine Angst! Am Ende des Kapitels habe ich einen *Tipp* für Dich.

- **Wertzuweisungen**
  Die einfachste Form einer Programmierzeile. (`int punkte = 10;`)

- **Kontrollstrukturen**
  Hierunter fallen nur 3 verschiedene Anweisungen, die elementar wichtig sind.
  - Die `if-(else-)`-Anweisung
  - Die `switch`-Anweisung
  - Die `for`-Schleife

- **Grundlagen der objektorientierten Programmierung**
  Das beinhaltet vor allem:
  - Klassen schreiben
  - Objekte erstellen (*Instanziieren*)
  - Referenzattribute, Referenzen setzen und löschen
  - Methoden von Objekten Aufrufen (Punktschreibweise)
  - Kapselung/Zugriffsrechte (`public`, `private`, ...)

- **Felder (oder Arrays)**
  Die praktische Umsetzung von *1-zu-n-Kardinalitäten*.
  Das heißt vor allem:
  - Funktion von Feldern
  - Erstellen von Feldern
  - „Füllen“ von Feldern
  - „Füllen“ von Feldern in einer `for`-Schleife

- **Vererbung**
  Das größte „Knochenbrecher-Thema“, mit dem Du Dich auseinander setzen müsstest. Auch mit *abstraktem Methoden* solltest Du Dich auskennen.

- **Interfaces**
  Die Funktion eines Interfaces solltest Du kennen. Du wirst die Möglichkeit haben, über Interfaces in der Engine Alpha auf bestimmte Vorgänge (zum Beispiel Mausklicks) reagieren zu können.

- **Umgang mit Dokumentationen**
  Das eigenständige Dokumentieren ist sehr hilfreich, aber nicht unbedingt notwendig.
  Das Umgehen mit einer Dokumentation ist nötig; die beiliegende Dokumentation der Engine Alpha sollte verständlich sein.

Solltest Du mit einigen dieser Anforderungen noch einige Schwierigkeiten haben, so empfehle ich Dir einen Lehrgang, der Dir den elementaren Stoff der 10. Klasse Informatik nahe bringt, während Du ein einfaches Pac-Man-Spiel programmierst: „Krümel und Monster“.

```
http://www.kruemelundmonster.de/
```

Es gibt auch einen weiteren Lehrgang, der jedoch etwas härter ist. Hier lernst du das Programmieren weit selbstständiger und direkt bereits mit der Engine Alpha:

```
http://engine-alpha.org/html/unterricht.html
```

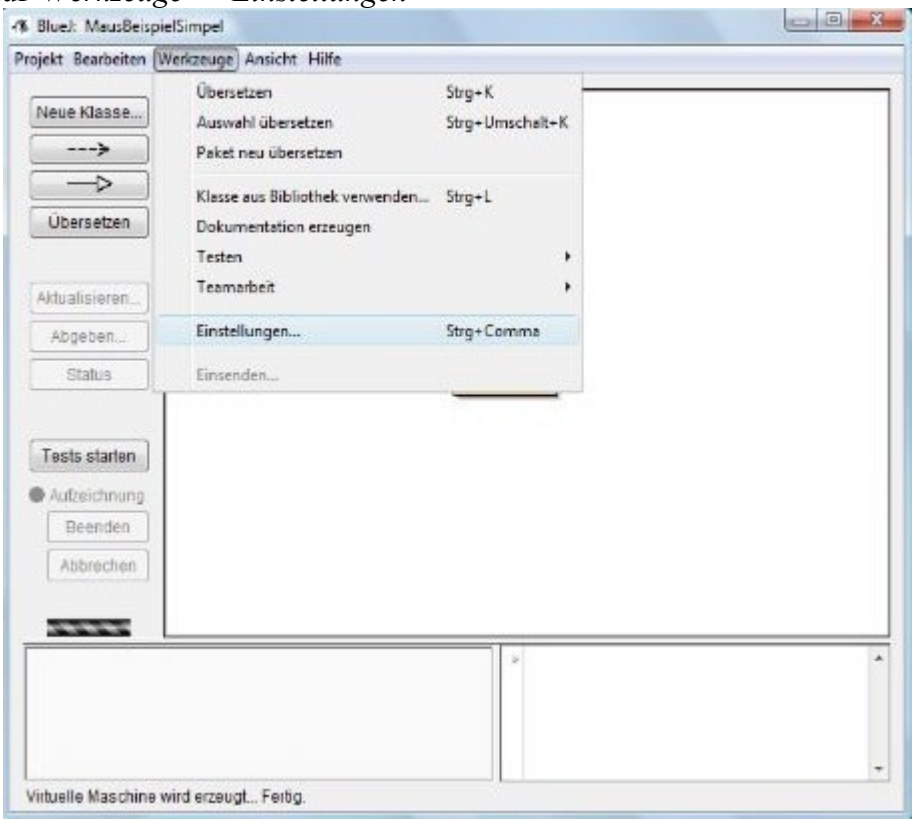
# III Einbringen der Engine

Bevor Du die Engine nutzen kannst, muss Du sie in Deine Entwicklungsumgebung einbinden.
Ich werde dies nun beispielhaft an *BlueJ* demonstrieren.

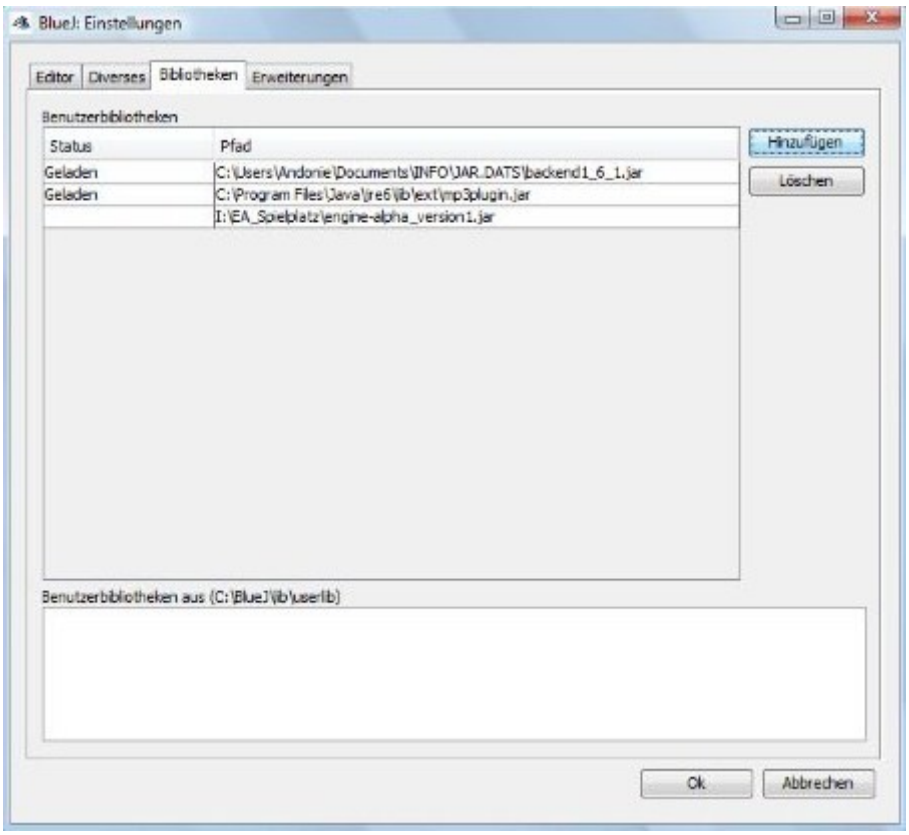
1. Die Engine Alpha ist als .jar-Datei gespeichert.
   Deponiere die Datei „engine-alpha_version[X].jar“ auf deinem Computer. Du kannst die Datei natürlich auch umbenennen.

2. Als nächstes muss Deine Entwicklungsumgebung darauf zugreifen können, dies machst Du bei BlueJ folgendermaßen:

   Gehe auf *Werkzeuge -> Einstellungen*




   Dann gehe auf *Bibliotheken>Hinzufügen* und wähle die am Computer deponierte .jar-Datei der Engine Alpha („engine-alpha_version[X].jar“).




Und das war es auch schon. Ab jetzt ist die Engine Alpha in Deinem Spiel verwendbar.

3. Um die Engine Alpha in Deine Klassen einzubinden, musst Du nun noch zu Beginn deines Quelltextes (noch vor der Klassendeklaration!) die folgende Zeile schreiben:

```
import ea.*;
```

Diese Zeile „importiert“ alle Klassen der Engine Alpha in den Quelltext, sodass Du sie verwenden kannst. Dies ist das „ea“-Paket. Das *Paketsystem* ist bei Programmsprachen sehr stark etabliert. Auch in Java kann die Engine Alpha es so für sich nutzen. Wenn Du verstehen willst, wie das Paketsystem funktioniert, schau Dir auch die Standard-Java-Pakete (das *JDK*) einmal an, als Stichwort das Paket `java.util`.

Kommt bei der `import`-Anweisung die folgende Fehlermeldung

```
package ea does not exist
```

so konnte die Engine Alpha nicht geladen werden. Dann wurde irgendetwas beim Einbinden der Engine falsch gemacht. Dann muss die Engine richtig in die Entwicklungsumgebung eingebunden werden.

# IV Programmieren mit der Engine Alpha – Die Grundlagen

Nun erkläre ich Dir die wichtigsten Konzepte der Engine Alpha, also das was Du **kennen musst**, damit Du Dein eigenes Spiel erstellen kannst.

Alle Klassen, die hier erläutert werden, sind in der Dokumentation vorhanden. Du solltest diese durchlesen, willst Du von einer Klasse den **vollen** Funktionsumfang kennen. Die Dokumentation ist immer umfangreicher und genauer als dieses Handbuch.

## *A Die spielsteuernde Klasse*

Es ist sinnvoll, dass bei einem Spiel immer eine Klasse „die Fäden in der Hand hat“. Dies ist die spielsteuernde Klasse.
**Mit dieser Klasse fängt jedes Spiel an, dies ist also der wichtigste Schritt!**

Diese Klasse muss sich aus der Klasse „`Game`“ in der Engine Alpha ableiten (Stichwort Vererbung).
Hierdurch wird die Superklasse `Game` bereits in ihrem Konstruktor alles tun, was notwendig ist, um die Spielgrundlagen fertig einzurichten. Hierin wird das Spielfenster erstellt, und die wichtigen internen Vorgänge der Engine gestartet.

Weiterhin hat die Klasse „`Game`“ eine abstrakte Methode:

```
/**
 * Diese Methode wird immer dann aufgerufen, wenn eine taste gedrueckt wurde.
 * @param tastenCode Der Zahlencode der Taste, die gedrueckt wurde, dadurch ist
 * jeder Tastendruck eindeutig zuzuordnen
 */
public abstract void tasteReagieren(int tastenCode)
```

Der Rumpf dieser Methode wird nun in Deiner eigenen Spielklasse geschrieben.
Diese Methode wird immer dann automatisch aufgerufen, wenn eine Taste auf der Tastatur gedrückt wird, die auch in der Engine Alpha verarbeitet wird.
Jede Taste hat einen Zahlencode, der die herunter gedrückte Taste eindeutig identifiziert und dadurch kann in dieser Methode jeder Tastendruck gesondert behandelt werden, obwohl nur eine Methode benötigt wird.
Welcher Code für welche Taste steht, ist in einer Tabelle vermerkt, die sich im Anhang dieses Handbuches befindet.

| Alphabet | | | | | Verschiedenes | |
|---|---|---|---|---|---|---|
| **A** | 0 | | **N** | 13 | **Pfeiltaste oben** | 26 |
| **B** | 1 | | **O** | 14 | **Rechts** | 27 |
| **C** | 2 | | **P** | 15 | **Unten** | 28 |
| **D** | 3 | | **Q** | 16 | **Links** | 29 |
| **E** | 4 | | **R** | 17 | **Leertaste** | 30 |
| **F** | 5 | | **S** | 18 | **Enter-Taste** | 31 |
| **G** | 6 | | **T** | 19 | **Escape-Taste** | 32 |
| **H** | 7 | | **U** | 20 | **0-Taste** | 33 |
| **I** | 8 | | **V** | 21 | **1** | 34 |
| **J** | 9 | | **W** | 22 | **2** | 35 |
| **K** | 10 | | **X** | 23 | **3** | 36 |
| **L** | 11 | | **Y** | 24 | **4** | 37 |
| **M** | 12 | | **Z** | 25 | **5** | 38 |

*Auszug aus der Tastenliste*

Die ganze Spielklasse sieht vom Quelltext her ungefähr so aus:

```
import ea.*; //Importieren der gesamten Engine Alpha

/**
 * Die Spielsteuerungsklasse fuer mein Spiel
 */
public class MeinSpiel

extends Game
{

    /**
     * Konstruktor, erstellt das Spielfenster und alle Hintergrundressourcen
     * in der Klasse <code>Game</code>
     */
    public MeinSpiel() {
        super(400, 300); //Aufruf des Konstruktors der Klasse Game;
        //erstellt ein Fenster der Breite 400 und Hoehe 300
    }

    /**
     * Diese Methode wird immer dann aufgerufen, wenn eine der Tasten der
     * Tastatur gedrueckt wurde.
     * @param   tastenCode  Der Code der gedrueckten Taste als Zahl. Dadurch
     *  kann man die Tasten unterscheiden un entsprechend auf darauf reagieren.
     *  Zum Beispiel mit einer <code>switch</code>-Anweisung
     */
    public void tasteReagieren(int tastenCode) {
        //Verarbeitung der Tastenbefehle, z.B. mit der switch-Anweisung
    }
}
```

Die Klasse, die sich aus der Klasse `Game` ableitet, wird im folgenden sehr oft als die „spielsteuernde Klasse“ bezeichnet.

Damit ist die eigene Spielklasse bereits grundlegend fertig.
Alles weitere wird in den folgenden Kapiteln behandelt.

## PROBLEME?

Ein Beispiel-Projekt mit einer einfachen, spielsteuernden Klasse kannst du von der EA-Website herunterladen: Hier.

# *B Das Grafiksystem*

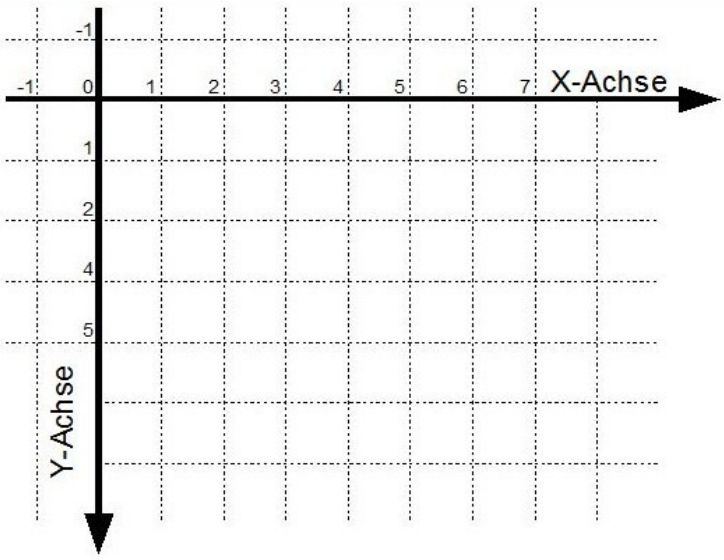
Im folgenden erkläre ich das Grafik-System der Engine. Denn die Grafik ist letzten Endes einer der wichtigsten Bereiche bei der Spieleprogrammierung.

## 1 Die Zeichenebene

Hierfür erkläre ich Dir kurz das grundlegende Konzept hinter der Grafik.

Alle Grafikelemente liegen auf der ***Zeichenebene***.
Die kannst Du Dir vorstellen wie ein Blatt Papier, auf das die verschiedenen Grafiken gemalt werden. Nur ist dieses Papier *in alle Richtungen unendlich groß*, so schnell sollte also kein Platzmangel herrschen.

Die Position eines Punktes auf der Zeichenebene lässt sich, wie auch in der Mathematik, durch ein Koordinatensystem definieren, wobei hier allerdings **die Y-Achse nach unten geht und der Ursprung links oben ist**.




Weiterhin sollte Dir klar sein, dass die Einheiten in diesem System **minimal winzig** sind! **Sämtliche Längeneinheiten werden in Pixel gerechnet**, also einer winzig kleinen Einheit.
*Angegeben werden Punkte und Maße natürlich immer wie im Mathematikunterricht: Als erstes die X-, dann die Y-Koordinate.*

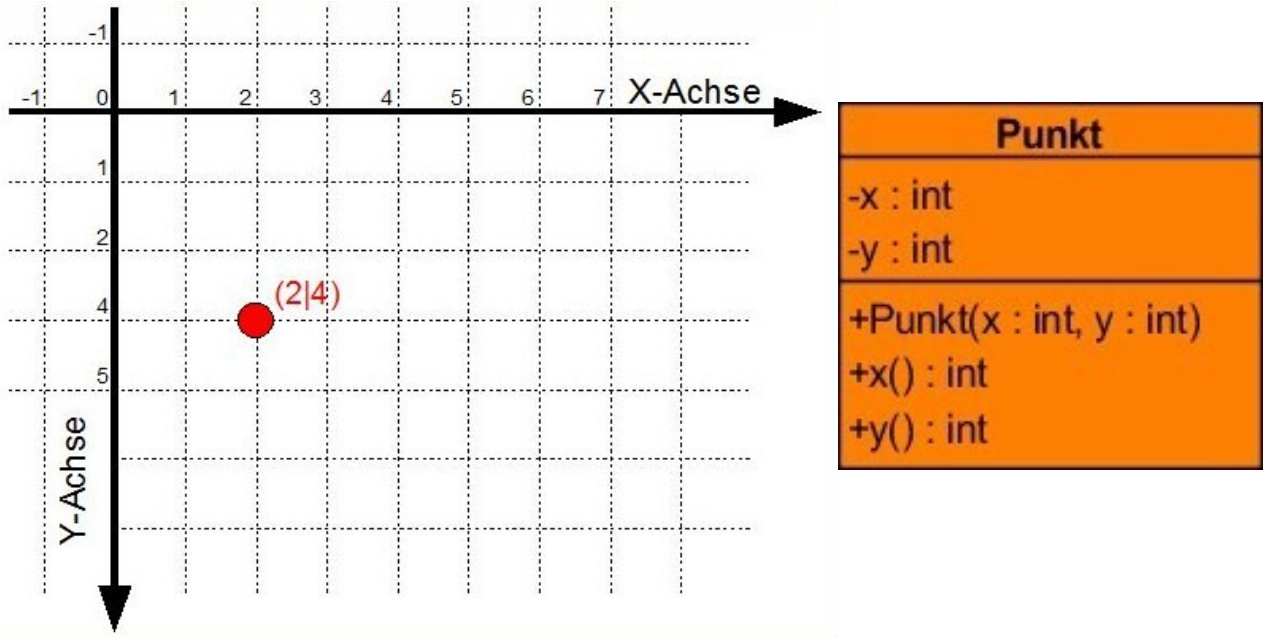
# 2 Die Klassen der Zeichenebene

Für diese Zeichenebene gibt es besondere, nicht grafische Klassen, mit denen Sachverhalte auf der Zeichenebene einfach beschrieben werden können.

## *i Die Klasse `Punkt`*

Die einfachste solche Klasse ist die Klasse `Punkt`. Diese beschreibt einen Punkt auf der Zeichenebene und wird vor Allem bei Positionsangaben verwendet.

Sie hat 2 Attribute, je ein `int`-Attribut für die X- und Y-Koordinate.



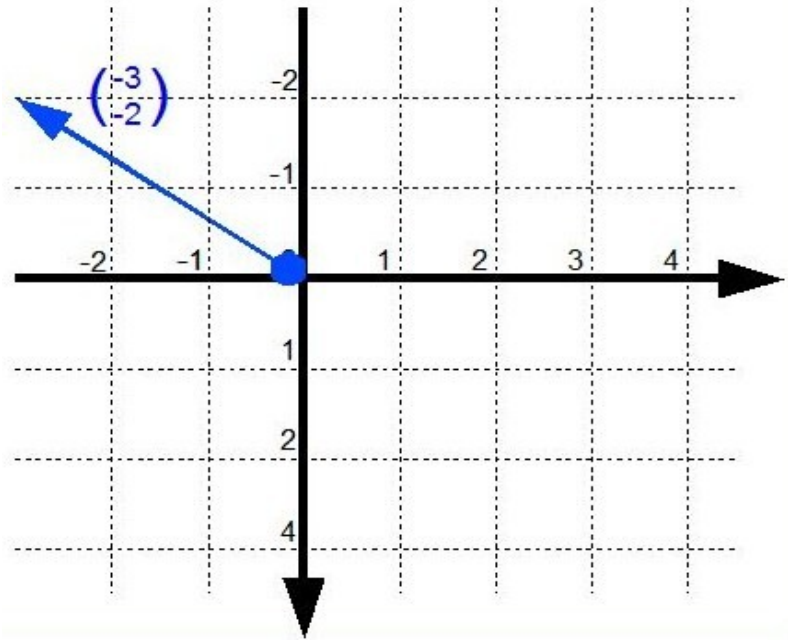

*Ein Punkt in der Zeichenebene mit den Koordinaten (2|4) und das Klassendiagramm von Punkt*

Erstellt wird dieser Punkt so:

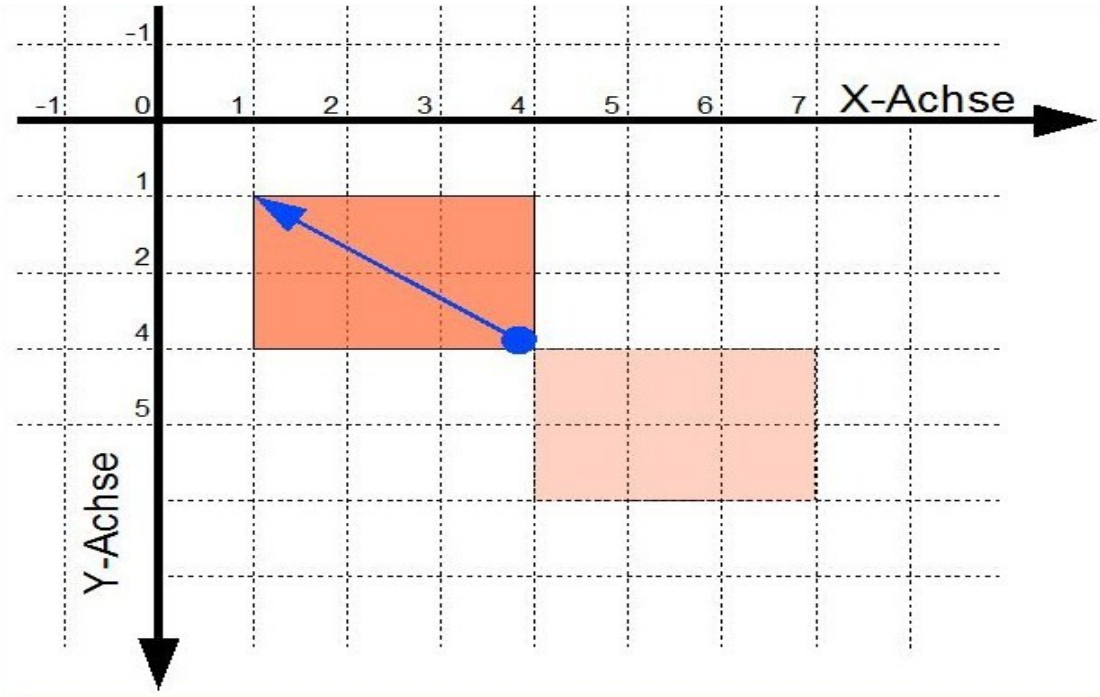
```
Punkt punktAn2_4 = new Punkt(2, 4);
```

### *ii Die Klasse `Vektor`*

Neben einem bestimmten Punkt auf der Zeichenebene kann man auch eine Bewegung auf der Zeichenebene um ein $\Delta x$ und ein b $\Delta y$ beschreiben. Dies macht die Klasse `Vektor`.
Diese hat, genau wie die Klasse `Punkt`, zwei `int`-Attribute für ihr $\Delta x$ und $\Delta y$ .




*Ein Vektor mit der Verschiebung (-3|-2)*




*Die Verschiebung als praktische Angabe*


*Das Klassendiagramm der Klasse Vektor*

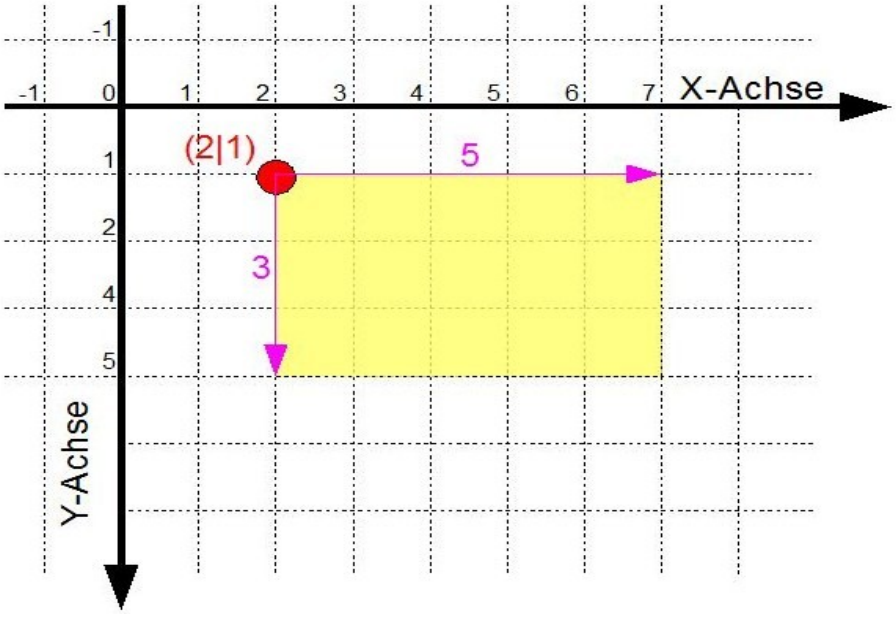
Erstellt wird dieser Vektor so:

```
Vektor verschiebung = new Vektor(-3, -2);
```

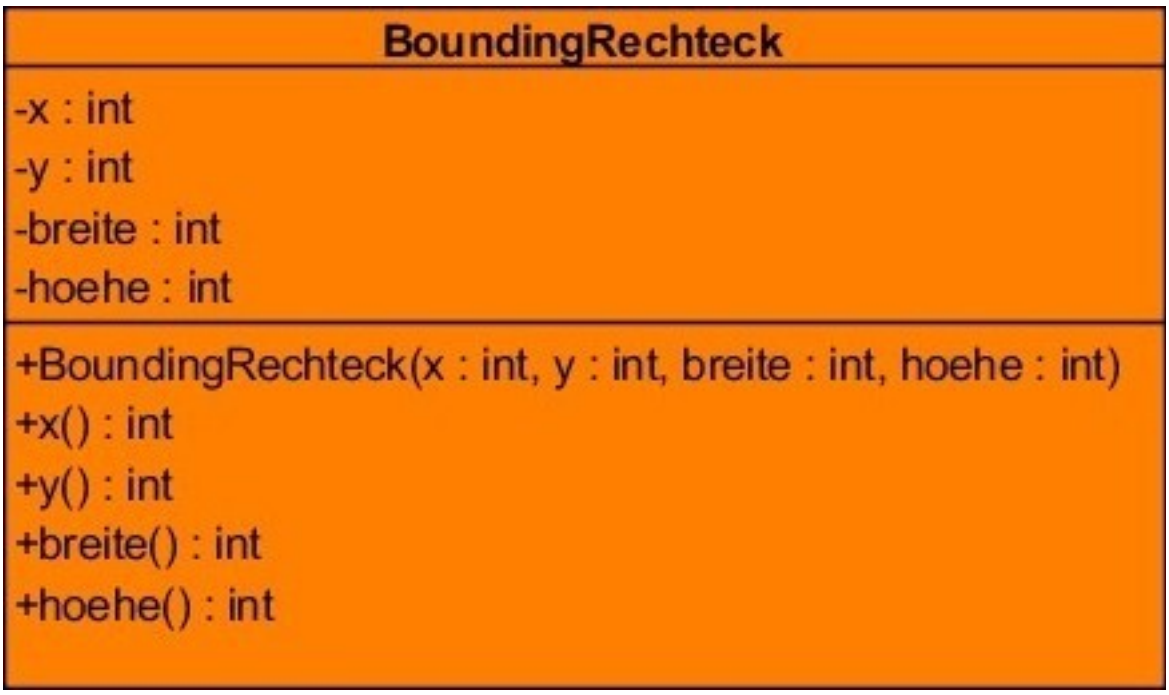
### *iii Die Klasse `BoundigRechteck`*

Neben Punkten und einfachen Bewegungen kann auf der Zeichenebene auch eine rechteckige Fläche beschrieben werden, deren Seiten immer parallel zu den Achsen sind.
So eine Fläche wird durch ein Objekt der Klasse `BoundingRechteck` beschrieben und ist definiert durch die X- und Y-Koordinate der linken oberen Ecke sowie seiner Breite und Höhe.

Diese Klasse wird jedoch vor allem intern benutzt und ist daher nicht unbedingt nötig für Dich. **Du kannst die Erklärung dieser Klasse ruhig überspringen**.




*Darstellung eines Bounding-Rechteks an der Position (2|1) mit der Breite 5 und der Länge 3*




*Klassenkarte der Klasse* `BoundingRechteck`

Erstellt wird dieses `BoundingRechteck` so:

```
BoundingRechteck br = new BoundingRechteck(2, 1, 3, 5);
```

## C Die Klasse `Raum`

Nun ist alles abgehandelt, was man zur Zeichenebene, dem „Blatt Papier“ wissen muss, und jetzt soll endlich auch etwas auf das Papier kommen!
Nun brauchen wir also Klassen, die gezeichnet werden können.
Gezeichnet werden können vor Allem:

- Texte
- Geometrische Figuren
- Bilder

Das sind die wichtigsten grafischen Elemente der meisten Computerspiele.
Und sie haben einiges gemeinsam.
Und spätestens hier klingelt das Glöckchen im Informatikerhinterkopf: „Hier kann man Vererbung nutzen!“

Genau das geschieht hier: Alle grafischen Klassen leiten sich aus einer Superklasse ab, die bereits alle wichtigen Eigenschaften für die grafischen Spielelemente definiert.

**Diese Klasse ist die Klasse `Raum`.**
Sie definiert neben den Zeichenoperationen einige sehr praktische Methoden, *die an jedem Grafik-Objekt ausgeführt werden können*:

| Methodenname | Parameter | Funktion |
|---|---|---|
| `verschieben` | `verschiebung : Vektor` | Verschiebt das Raum-Objekt um eine Verschiebung, diese wird als Vektor mitgegeben |
| `verschieben` | `x : int`<br>`y : int` | Dasselbe wie `verschieben(Vektor)`. Jedoch werden hier direkt die Verschiebungswerte verlangt, und nicht ein komplexer Datentyp.<br>Diese Alternativmethode gibt es auch für die folgenden Positionsmethoden. |
| `positionSetzen` | `position : Punkt` | Setzt die Position des Raum-Objektes neu. Der anzugebende Punkt ist die neue Position der obersten linken Ecke des Raum-Objektes. |
| `mittelpunktSetzen` | `mittelpunkt : Punkt` | Setzt den Mittelpunkt des Raum-Objektes. Es wird so verschoben, dass der Mittelpunkt auf dem angebenen Punkt liegt. |
| | | |
| `schneidet` | `anderesObjekt : Raum` | Berechnet automatisch und bequem, ob dieses Raum-Objekt ein anderes schneidet, und gibt in diesem Fall `true` zurück, ansonsten wird `false` zurückgegeben. |
| `dimension` | `[keine]` | Gibt ein `BoundingRechteck` zurück, dass dieses Raumobjekt exakt gerade so abdeckt |


*Klassenkarte von* `Raum` *mit den genannten Methoden*

Diese und einige andere Methoden (*in den umfangreichen Dokumentationen nachlesbar*) lassen sich beliebig auf jede grafische Klasse anwenden, sei es ein Bild, ein Text oder anderes. Denn alle Klassen hierfür leiten sich ja aus `Raum` ab. Dieser enorme Vorteil bietet auch gleichzeitig z.B. Kollisionstests zweier beliebiger grafischer Objekte.

Diese Klassen werden jetzt kurz mit ihren eigenen, nicht geerbten Eigenschaften behandelt:

**Achtung, vorab stelle Ich klar: wenn Du ein Objekt einer solchen Klasse zum Ausprobieren erstellst, wird es nicht im Fenster sichtbar sein!**
**Sei bitte daher nicht enttäuscht oder entmutigt, im nächsten Kapitel lernst Du, Deine Raum-Objekte sichtbar werden zu lassen.**
**Möchtest Du dennoch Deine Raum-Objekte vorzeitig sichtbar machen, binde diese Methode in Deine spielsteuernde Klasse ein und rufe sie nach Bedarf auf:**

```
/**
 * Macht ein beliebiges Raum-Objekt sichtbar.<br />
 * Dank der Vererbungshierarchie koennen ueber
 * diese Methode Texte wie Bilder und andere
 * grafische Elemente mit einer Methode behandelt werden.<br />
 * Diese Methode muss noch nicht verstanden werden. Sie wird
 * im Kapitel 'Knoten' behandelt und erklaert.
 * @param   m   Das sichtbar zu machende Raum-Objekt
 */
public void sichtbarMachen(Raum m) {
    wurzel.add(m);
}
```

Diese Methode wird im folgenden Kapitel, nach der Einführung der wichtigsten grafischen Klassen, erläutert.

Ich stelle bei den folgenden Klassen nur kurz die wichtigsten Methoden und längsten Konstruktoren vor, doch *besonders bei den Konstruktoren muss es nicht immer so kompliziert sein*. Nimm die Dokumentation der entsprechenden Klasse zur Hand und lies mit. Du wirst feststellen, dass das Arbeiten mit diesen Klassen so viel einfacher sein kann, wenn man nur die einfachereren Methoden braucht.

# 1 Bilder

Bilder sind weit einfacher einzubinden, nur eine einzige wichtige Eigenschaft muss angegeben werden, nämlich der Dateiname der dazugehörigen Bilddatei.
Diese Bilddatei legst Du einfach in Deinen Projektordner, und dann ist sie für die Engine Alpha erreichbar.

Hier der grundlegende Konstruktor:

```
public Bild(   int x,
               int y,
               String verzeichnis)
```

Eine kurze Erläuterung der Parameter:

| | |
|---|---|
| `x` | Die X-Koordinate der linken oberen Ecke des Bildes auf der Zeichenebene. |
| `y` | Die Y-Koordinate der linken oberen Ecke des Bildes auf der Zeichenebene. |
| `verzeichnis` | Das Verzeichnis des zu ladenden Bildes.<br>Lege das Bild in Deinen Projektordner; Du musst nur den Dateinamen angeben, zum Beispiel "`meinBild.jpg`". |

Das reicht schon, um Deine Bilder einzubinden.
Es gibt noch andere interessante Funktionen, wie Bilder vor dem Darstellen auf eine gewünschte Größe skalieren, oder ein Bild in einer Größe auf einer definierten Fläche immer wieder wiederholen.

Doch diese Spielereien lassen sich nach Bedarf in den Dokumentationen bestens einsehen.

## PROBLEME?

Ein Beispiel-Projekt, das nur ein Bild lädt, kannst du von der EA-Website herunterladen: Hier.

# 2 Texte

Hierfür gibt es die Klasse `Text`.
Kurz und knapp: Du kannst ein `String`-Objekt, also eine Zeichenkette hierdurch grafisch in Dein Spiel bringen.
Ein Text hat folgende Eigenschaften:

- Einen Inhalt
- Eine bestimmte Schriftgröße
- Einen bestimmten Font, in dem er dargestellt wird
- Eine Schriftart (Normal, Fett, Kursiv, oder beides)
- Eine Farbe

Diese Eigenschaften werden im Konstruktor mitgegeben:

```
public Text(String inhalt,
            int x,
            int y,
            String fontName,
            int schriftGroesse,
            int schriftart,
            String farbe)
```

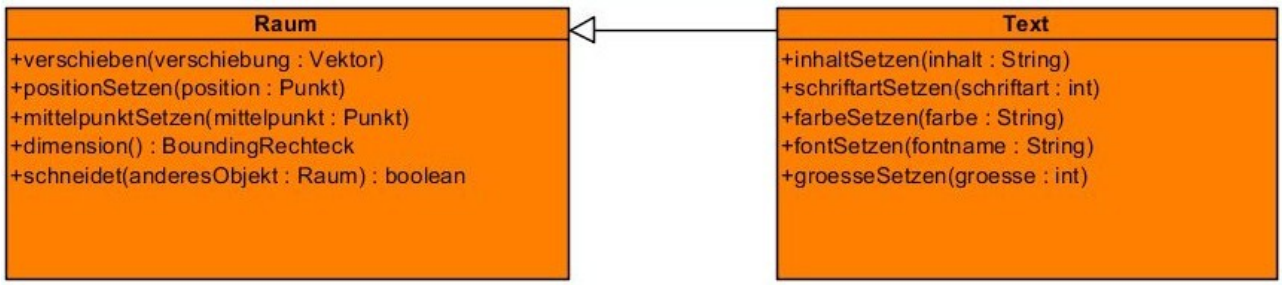
Du musst nicht alle Parameter benutzen, *es gibt alternative Konstruktoren*, bei denen Du von unten her Parameter weglassen kannst, bis Du nur noch den Inhalt, sowie die X- und Y-Koordinate angeben musst.
*Möchtest du es ganz genau wissen, kannst du in die Dokumentation der Klasse `Text` schauen.*

Für maßgeschneiderte Texte in Deinem Spiel, erkläre Ich hier alle Parameter des größten Konstruktors:

| | |
|---|---|
| `inhalt` | Der Inhalt des Textes als Zeichenkette. |
| `x` | Die X-Koordinate der linken oberen Ecke des Textes auf der Zeichenebene. |
| `y` | Die Y-Koordinate der linken oberen Ecke des Textes auf der Zeichenebene. |
| `fontName` | Der Name des Fonts, in dem der Text dargestellt werden soll (zum Beispiel „Times New Roman“ oder „Arial“).<br>**Du kannst auch eigene Fontdateien (.ttf-Dateien) in Dein Spiel einbinden:**<br>Du musst die Datei nur in den Projektordner legen, die Engine wird sie automatisch laden und Du kannst sie dann verwenden, als ob sie auf dem Computer wäre.<br>Hast Du alle gewünschten Schriftarten in deinem Projektordner, solltest Du folgende Quelltextzeile einmal aufrufen:<br>`Text.geladeneSchriftartenAusgeben();`<br>Dann werden Dir an der Konsole alle aus Deinem Projektordner geladenen Schriftarten genannt, inklusive der Namen, unter denen Du sie verwenden kannst. |

| | |
|---|---|
| `schriftGroesse` | Die Schriftgröße des Textes. Wie bei einem Textverarbeitungsprogramm. |
| `schriftart` | Die Schriftart des Textes, nur Werte zwischen einschließlich 0 und 3 sind möglich:<br>• 0: Normaler Text<br>• 1: Fetter Text<br>• 2: Kursiver Text<br>• 3: Fetter & Kursiver Text<br>Andere Werte hierfür sind **nicht** möglich! |
| `farbe` | Die Farbe, in der der Text dargestellt werden soll, als Zeichenkette.<br>Die Engine Alpha kann eine Reihe von Zeichenketten als Farben interpretieren (z. B. „Gruen“). Welche das sind, wird im Kapitel *Farben* aufgelistet. |

Alle diese Eigenschaften können mit entsprechenden „setzen“-Methoden geändert werden:




*Klassenkarte von `Text` ohne Konstruktor (mit angezeigter Vererbung aus `Raum`)*

## PROBLEME?

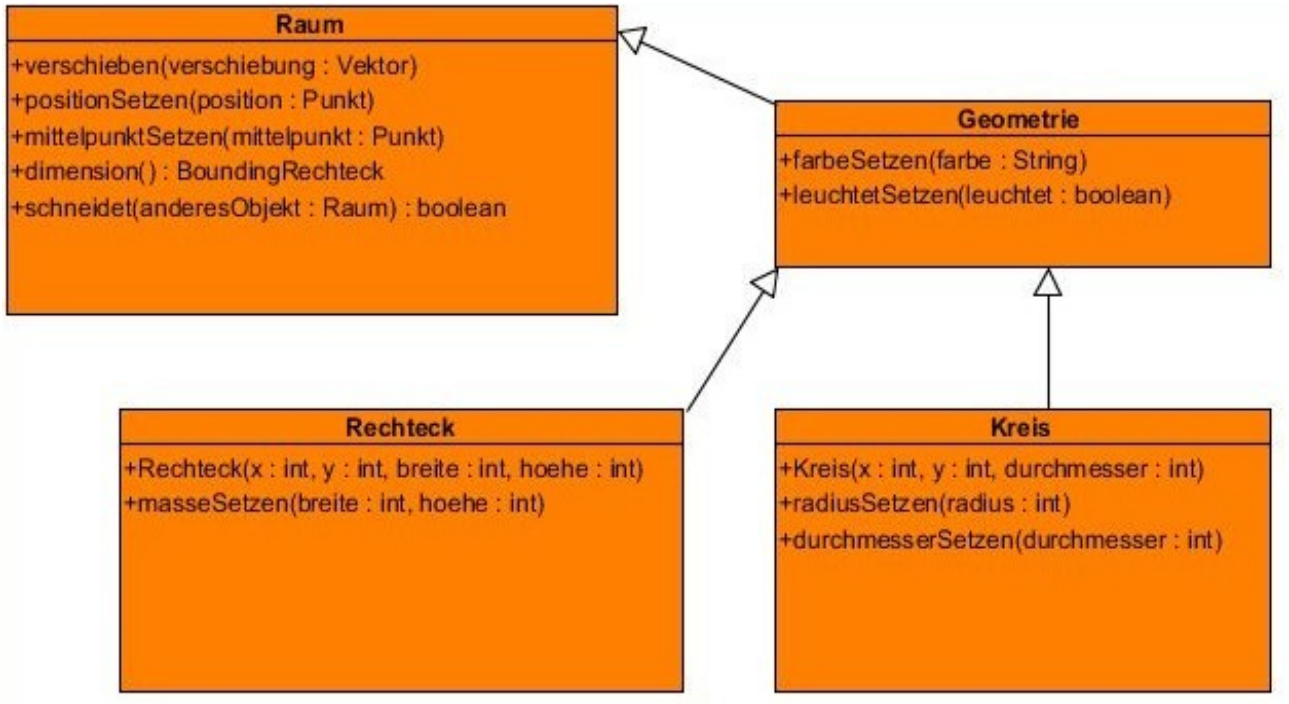
Ein Beispiel-Projekt, das nur einen Text erstellt und ihn etwas modifizieren kann, kannst du von der EA-Website herunterladen: Hier.

# 3 Geometrische Figuren

Für geometrische Figuren gibt es in der Engine Alpha jeweils eigene, einfache Klassen.
Jede dieser Klassen leitet sich aus der Klasse `Geometrie` ab, die sich selber aus Raum ableitet.

Diese Klassen sind sehr einfach zu benutzen, die zwei wichtigsten sind wohl die Klassen `Kreis` und `Rechteck`.

| **Klasse** | `Rechteck` | `Kreis` |
|---|---|---|
| **Konstruktor** | `Rechteck( int x, int y, int breite, int hoehe)` | `Kreis( int x, int y, int durchmesser)` |
| **Methode zum Ändern der Maße** | `masseSetzen( int hoehe, int breite)` | `durchmesserSetzen( int durchmesser)` |




*Die Hierarchie der wichtigsten Geometrieklassen*

Bei sämtlichen Geometrie-Figuren kannst du folgende Methode aufrufen:

```
public void farbeSetzen(String farbe)
```

Hiermit kannst du die Füllfarbe einer Geometrie-Figur ändern. Du kannst auch kompliziertere Farben mit dieser Methode setzen, indem du anstatt einem `String`-Objekt ein `Farbe`-Objekt eingibst. Näheres hierzu findest du im Kapitel Farben.

# *D Die Klasse `Knoten`*

Nun kennst Du die wichtigsten grafischen Klassen. Doch wenn Du versucht hast, sie in Dein Spielfenster zu bringen, konntest Du das nur durch die Spezialmethode, die Ich Dir im letzten Kapitel gezeigt habe.

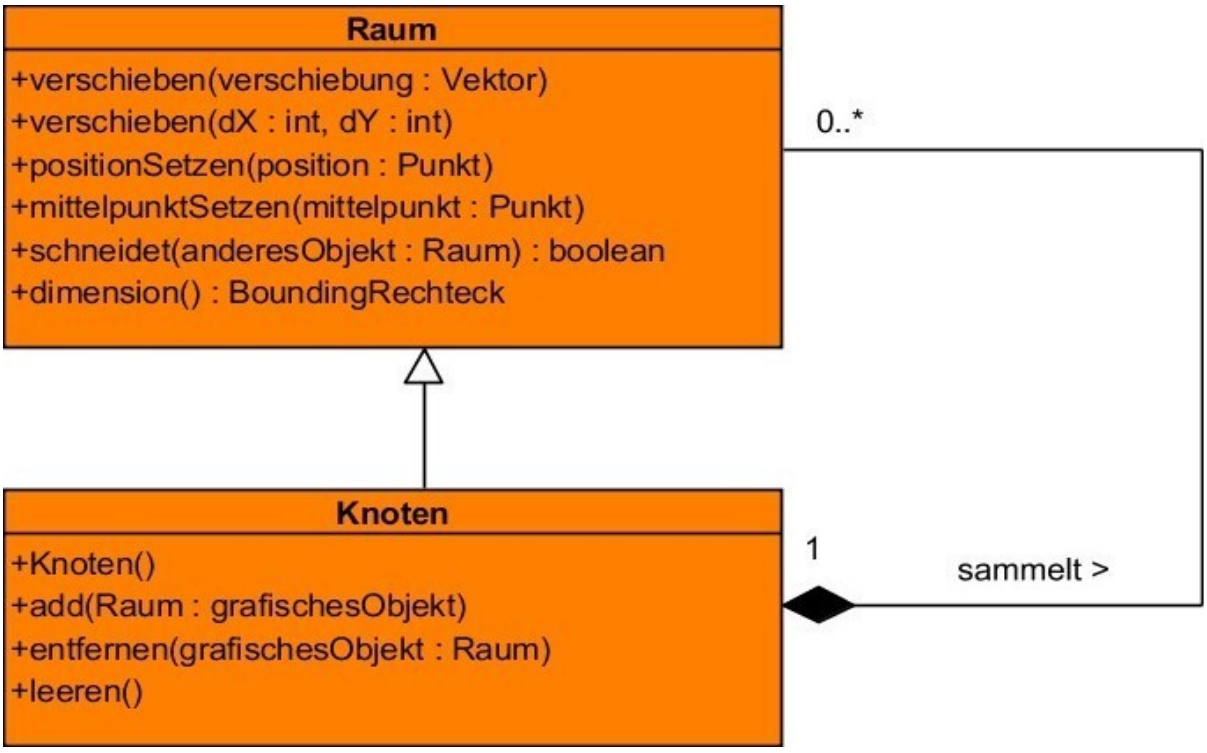
## 1 Definition der Klasse

Dahinter steckt ein spezielles System, auf dem das ganze Grafiksystem basiert, und das, wenn man es einmal durchblickt hat, ein unglaublich zeitsparendes und vorteilhaftes ist.

Dieses System ist das *Knoten-System*.

***Durchhalten, das ist das letzte Kapitel, dann kannst Du bereits Dein eigenes Spiel programmieren!***

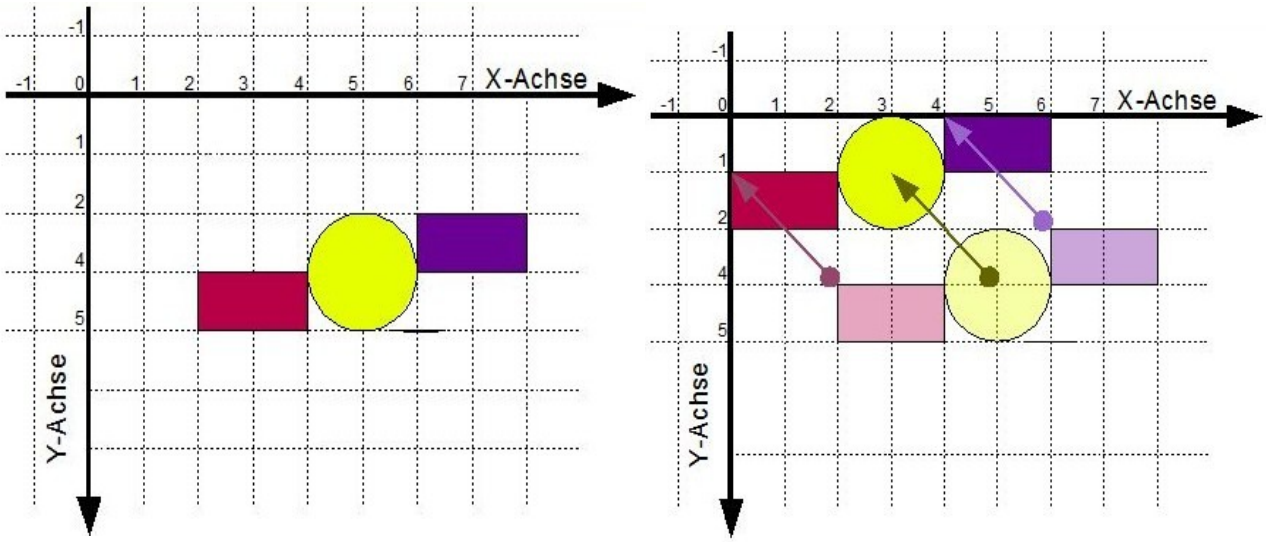
`Knoten` ist eine Klasse, die sich aus `Raum` ableitet.
Das besondere an ihr ist jedoch, dass sie kein richtiges sichtbares Grafikelement ist. Sie sammelt andere `Raum`-Objekte, und überträgt Befehle, die sie ausführen soll, auf alle gesammelten Elemente.



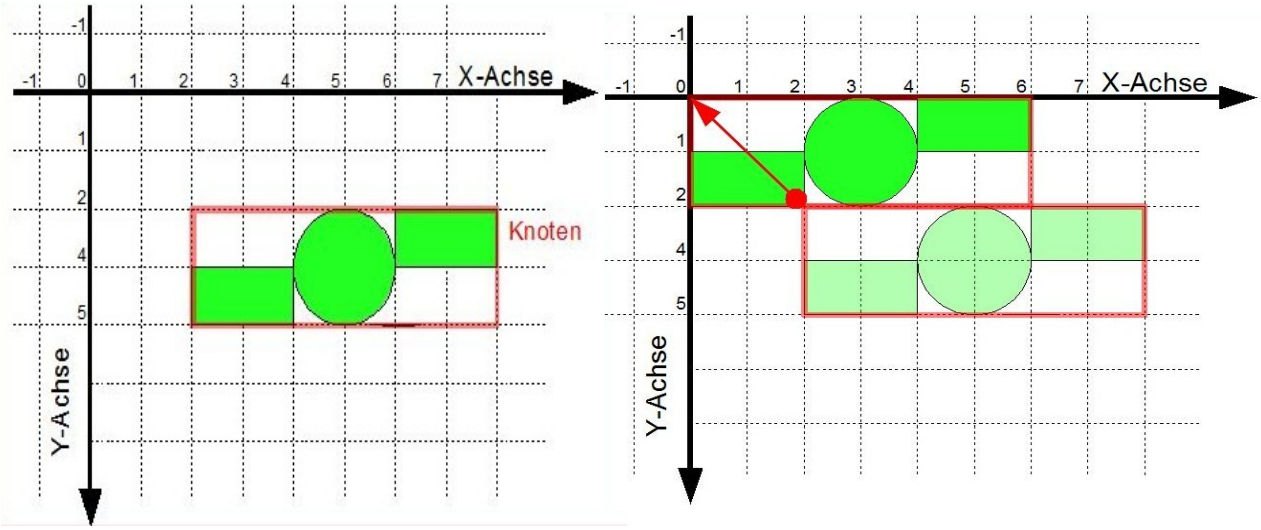

*Die Klasse `Knoten` und die Klasse `Raum` (mit Vererbung)*

Haben wir also eine Spielfigur, die aus einem Kreis und zwei Rechtecken besteht, können wir diese an einem `Knoten` sammeln und so zum Beispiel mit einem Verschiebebefehl in einer Quelltextzeile alles erreichen, wofür sonst 3 Programmzeilen nötig werden. Das spart zum einen Programmieraufwand und verhindert zum anderen Flüchtigkeitsfehler, und macht vor allem die gesamte Arbeit wesentlich strukturierter.

Das Prinzip das dahintersteht, heißt auch *Composite*.




*Drei einzelne Spielemente; sie müssen alle einzeln verschoben werden, damit die Figur bestehen bleibt.*




*An einem Knoten hängen alle Spielemente; nur der Knoten muss verschoben werden, und die Bewegung wird auf alle Objekte übertragen.*

Die Klasse Knoten verfügt - neben den aus Raum geerbten Methoden und **einem parameterlosen Konstruktor** – ausschließlich über Methoden zum Organisieren von anderen Raum-Objekten:

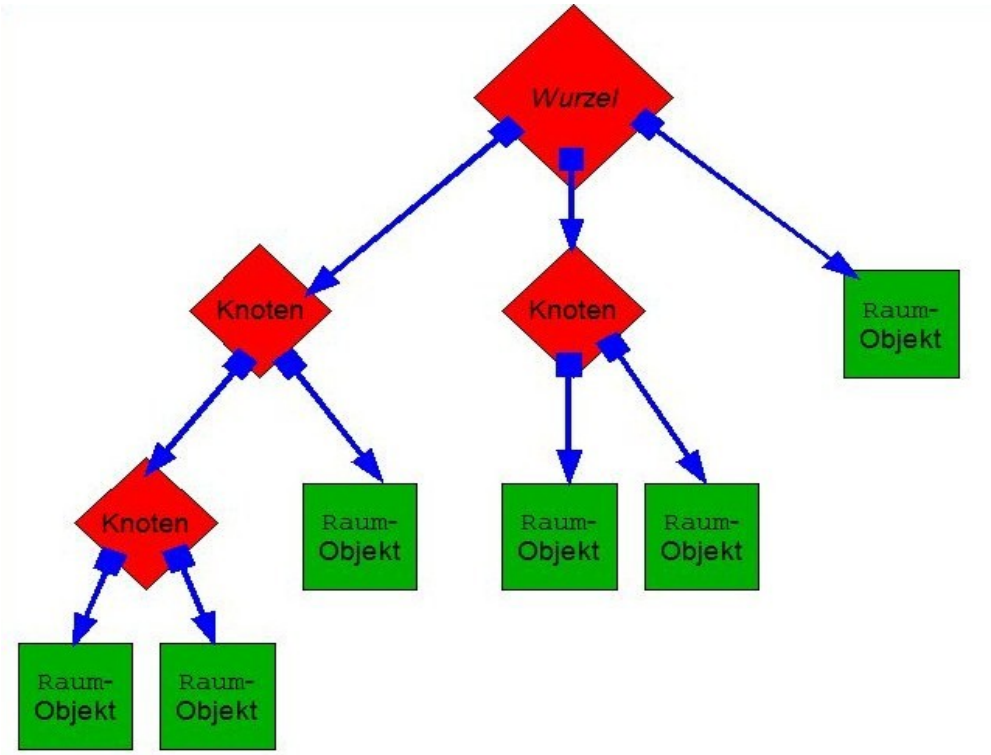
| Name | Parameter | Funktion |
|---|---|---|
| `add` | `raum : Raum` | Fügt diesem Knoten ein `Raum`-Objekt hinzu.<br>Das hinzugefügte `Raum`-Objekt wird ab sofort mit verschoben, wenn der Knoten verschoben wird. |
| `entfernen` | `raum : Raum` | Entfernt ein `Raum`-Objekt von diesem Knoten.<br>Das entfernte `Raum`-Objekt wird ab sofort nicht mehr bei Methoden mit aufgerufen. |
| `leeren` | `[keine Parameter]` | Entfernt alle `Raum`-Objekte von diesem Knoten. Dann ist der Knoten wieder leer. |

Nun weißt Du, wie ein Knoten funktioniert. Dieses System nennt der Informatiker auch *hierarchisches Prinzip der Baumstruktur*. Es entspricht dem System, nach dem Dateiordner funktionieren. Ein Dateiordner kann jede Art von Datei enthalten, auch einen weiteren Ordner.

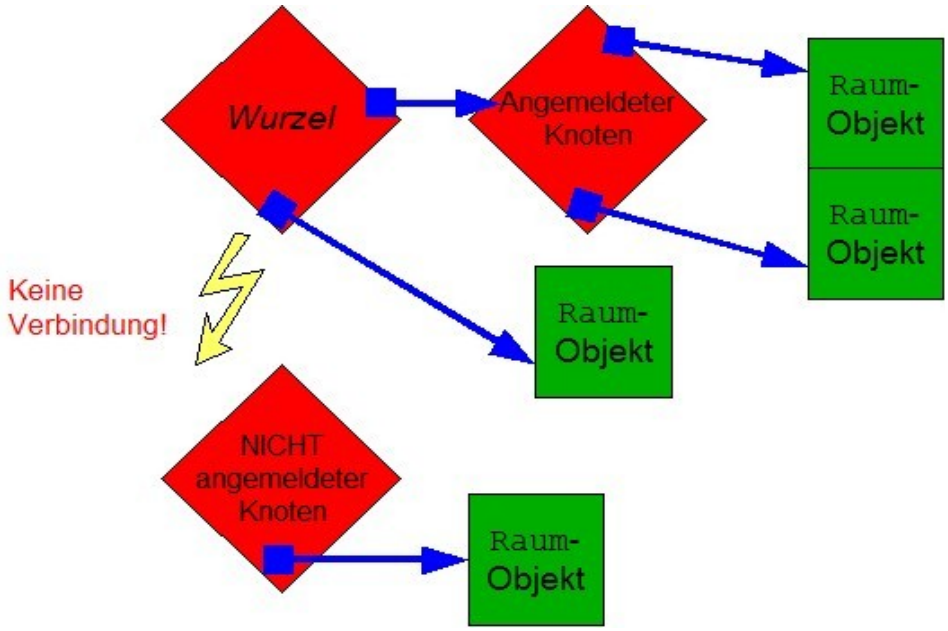
## 2 Objekte werden sichtbar

Wie Du nun sicher bereits gemerkt hast, musst Du immer noch diese kryptische Befehlszeile benutzen, die ich Dir gegeben habe, wenn Du ein Objekt sichtbar in Dein Fenster bringen willst. Nun will ich Dir erklären, welches System dahintersteckt.

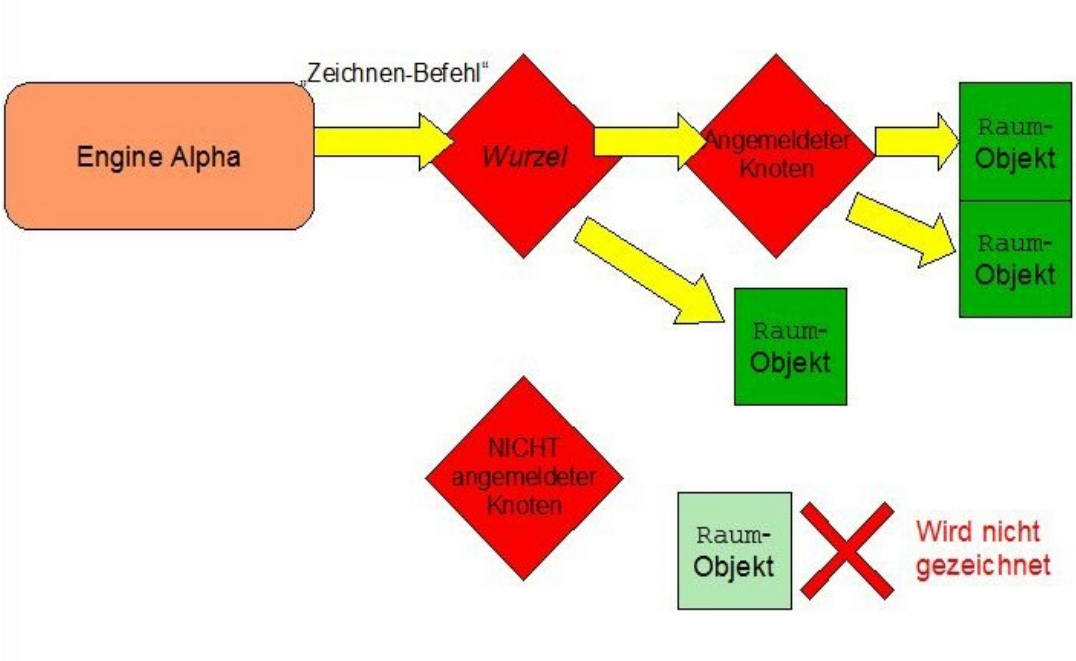
Jedes `Raum`-Objekt hat einen „Zeichnen-Befehl“, eine abstrakte Methode, mit der es sich zeichnen kann. Ein Knoten gibt natürlich diesen „Zeichnen-Befehl“ an alle gesammelten Objekte weiter.
In der Engine Alpha gibt es einen Superknoten, den obersten aller Knoten, die gezeichnet werden. Dieser Knoten ist die *Wurzel*. Die Klasse `Game` hat eine Referenz auf ihn, unter dem Namen `wurzel`.
Jedes Objekt, das sichtbar sein soll, muss an der Wurzel, oder natürlich an einem anderen Knoten, der wiederum an der Wurzel oder einem Unterknoten der Wurzel angemeldet ist, angemeldet werden. Das können beliebig viele sein, jeder Knoten kann ja unendlich viele Objekte erfassen.




*Beispiel für die Struktur von Objekten an der Wurzel. Alle Raum-Objekte werden gezeichnet, da sie alle – wenn auch teils über mehrere Ecken – mit der Wurzel verbunden sind.*




*Der untere Knoten ist nicht an der Wurzel angemeldet...*




*...daher wird das Raum-Objekt, das an ihm liegt, nicht gezeichnet.*

Jetzt ist auch die Programmzeile verständlich, die ein `Raum`-Objekt sichtbar werden lässt. Das entsprechende Objekt wird einfach an der Wurzel angemeldet, und damit kommt die Engine-Alpha auch an das Objekt heran und kann den Zeichnen-Befehl ausführen.

# 3 Eine kleine Praxisübung

Praktisch angewandt kann man das Knoten-System dann nutzen, wenn mehrere Abläufe (welcher Art auch immer: Bewegungen, Kollisionstests etc.) auf bestimmte Teile eines Grafikansammlung angewandt werden müssen.

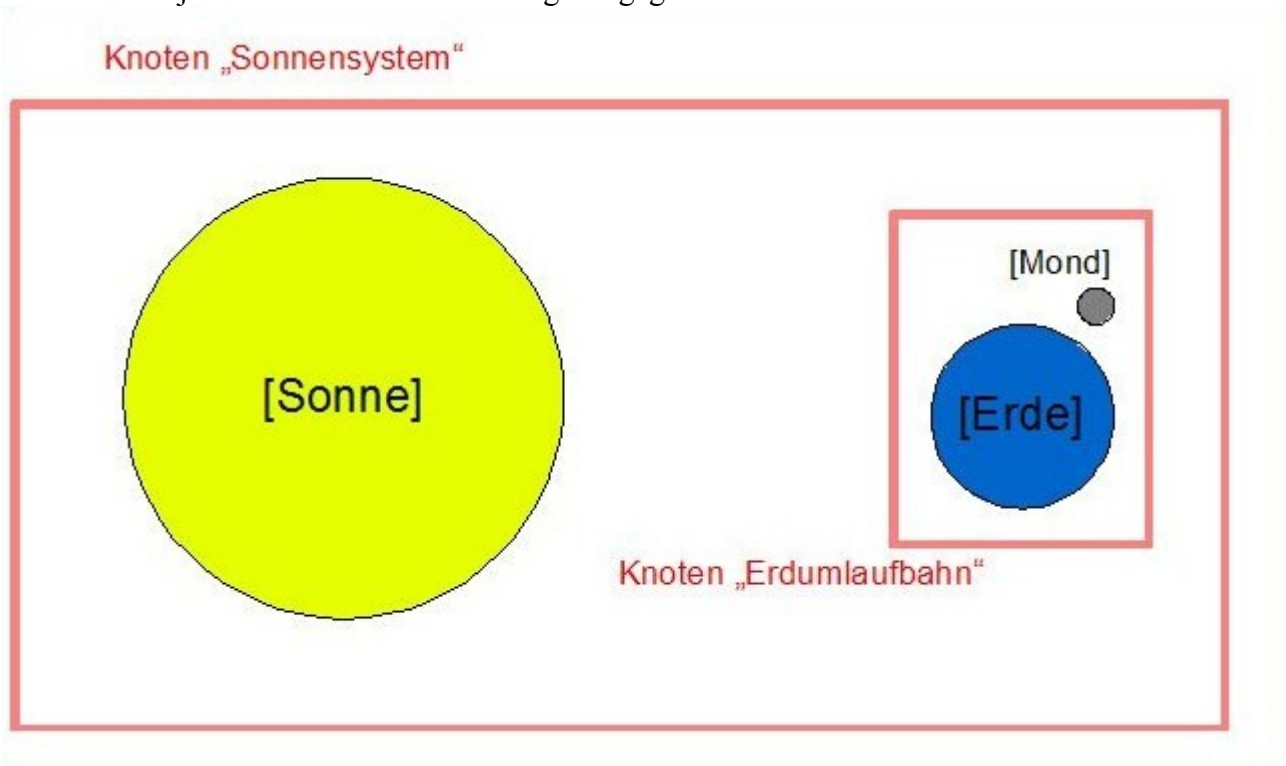
Ein einfaches Beispiel ist die Nachbildung unseres Sonnensystems (vereinfacht):

Es gibt drei Objekte:

- Die Sonne
- Die Erde
- Der Mond

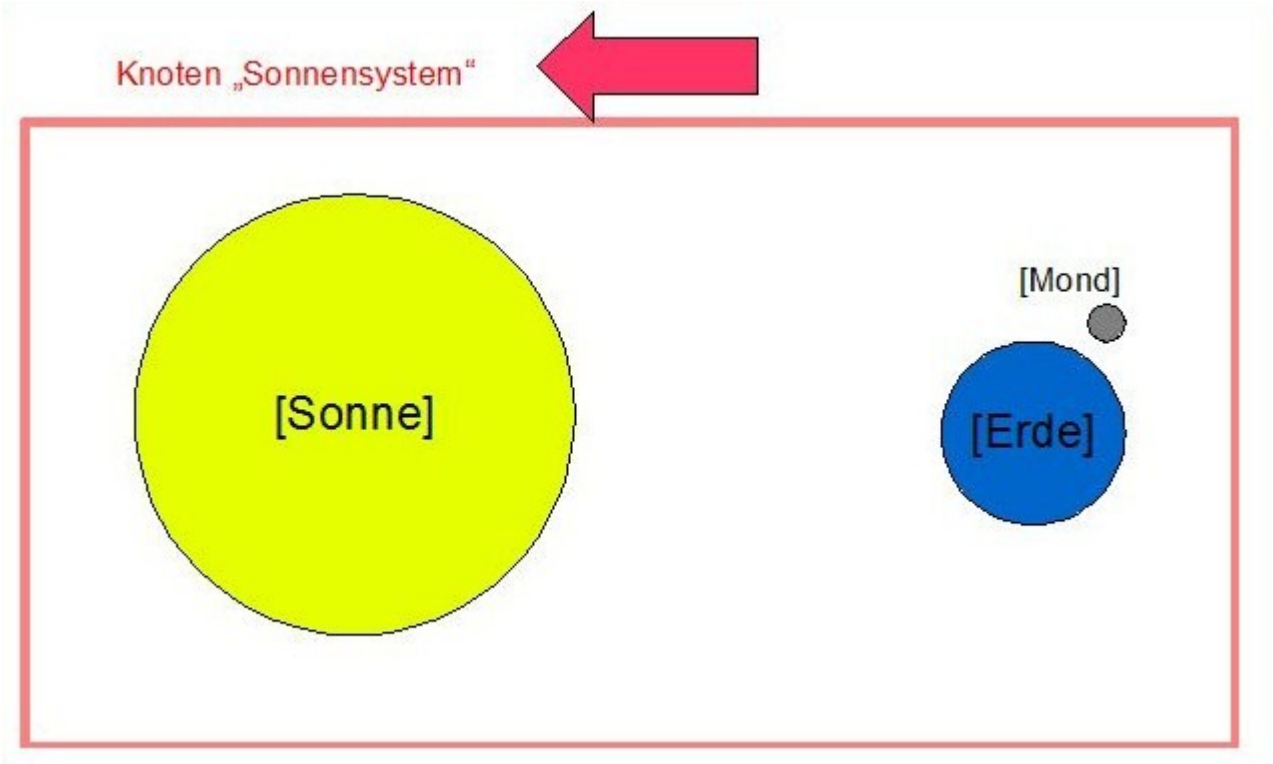
Alle lassen sich durch Kreise darstellen (siehe *Kapitel IV.C.3 – Geometrische Figuren*).

Diese drei Objekte werden über Knoten logisch gegliedert:

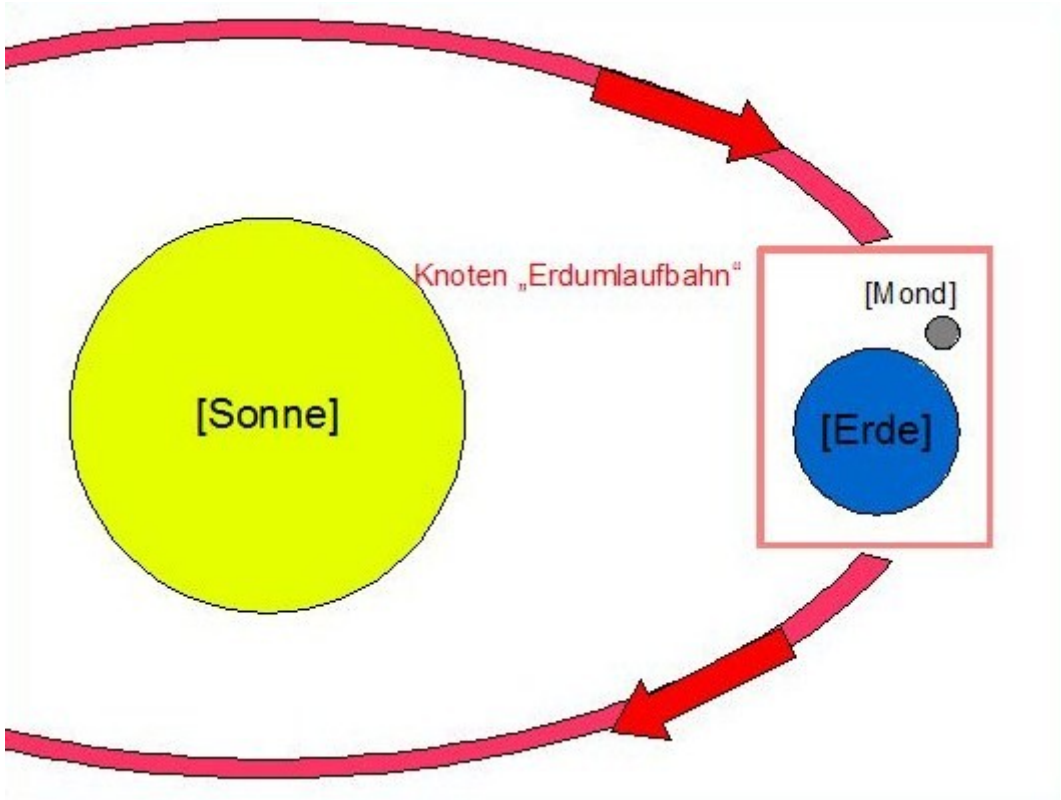



*Die drei Himmelskörper über 2 Knoten gegliedert*

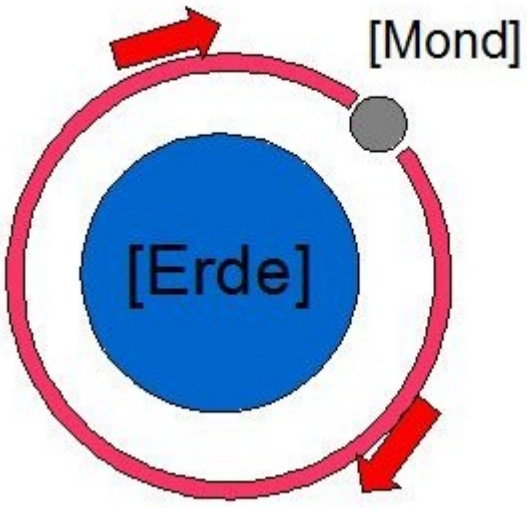
Nun können wir verschiedene Bewegungsschemata für die einzelnen Himmelskörper und Himmelskörpergruppen (sprich: die Knoten) aufstellen:




*Das ganze Sonnensystem ist in Bewegung*




*Die Erdumlaufbahn (Erde & Mond) dreht sich um die Sonne*




*Der Mond dreht sich um die Erde*

Die Erstellung der Himmelskörper als Kreise und das Einordnen in Knoten sollte kein Problem sein. Für die Drehungen schenke ich Dir zwei nette kleine Methoden:

```
/**
 * Dreht ein Objekt immer wieder um ein bestimmtes anderes.
 * @param   zentrum Das Raum-Objekt, um das sich das andere Objekt drehen soll.
 * @param   drehobjekt  Das Objekt, das sich drehen soll
 * @param   umlaufzeit  Zeit <b>in Millisekunden</b>, die eine Umdrehung dauern soll.
 */
public void dreheUm(Raum zentrum, Raum drehobjekt, int umlaufzeit) {
    animationsManager.kreisAnimation(drehobjekt, zentrum.zentrum(), umlaufzeit);
}
```

*Methode zum Drehen eines Raum-Objektes um ein anderes (in die spielsteuernde Klasse kopieren)*

```
/**
 * Dreht ein Objekt immer wieder um einen bestimmten Punkt.
 * @param   zentrum Das zentrum der Drehung
 * @param   drehobjekt  Das Objekt, das sich drehen soll
 * @param   umlaufzeit  Zeit <b>in Millisekunden</b>, die eine Umdrehung dauern soll.
 */
public void dreheUm(Punkt zentrum, Raum drehobjekt, int umlaufzeit) {
    animationsManager.kreisAnimation(drehobjekt, zentrum, umlaufzeit);
}
```

*Methode zum Drehen eines Raum-Objektes um einen Fixpunkt (in die spielsteuernde Klasse kopieren)*

Mit diesen zwei Methoden kannst Du ganz einfach ein `Raum`-Objekt um ein anderes (oder einen Fixpunkt) drehen. Denk daran, **auch ein Knoten ist ein `Raum`-Objekt!** Du kannst also ohne Probleme die *Erdumlaufbahn* mit dieser Methode um die *Sonne* drehen.

Ich wünsche Dir viel Spaß und Erfolg bei deinem Sonnensystem, gib nicht auf, wenn etwas nicht sofort klappt!

## PROBLEME?

Ein Beispiel-Projekt, das dieses Sonnensystem umgesetzt hat, kannst du von der EA-Website herunterladen: Hier.

Bei den Methoden zum Drehen von Objekten, die ich Dir abgedruckt habe, werden Animationen verwendet. Dies und viele weitere interessante Funktionen der Engine Alpha kannst Du nun nach belieben erkunden.
Zu allen wichtigen Funktionen findet sich jeweils ein weiteres Kapitel in diesem Handbuch.

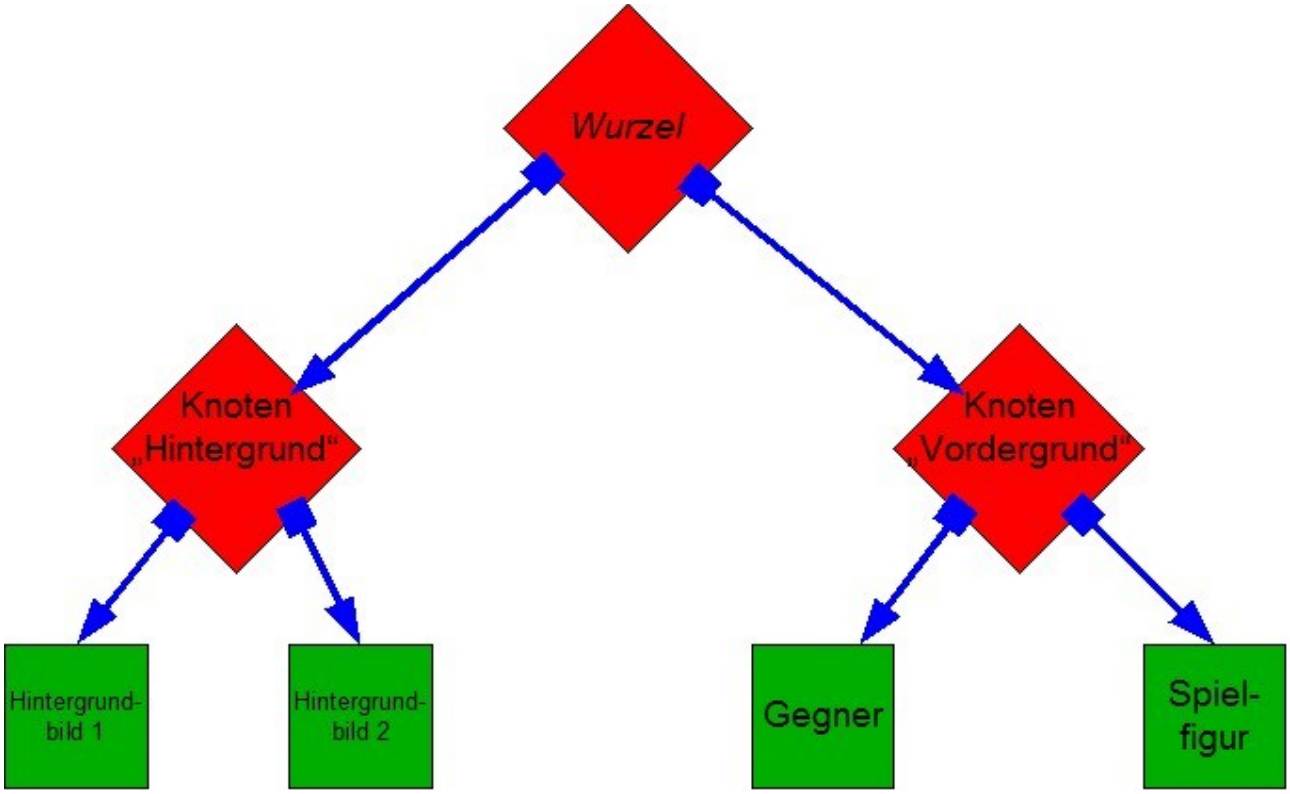
**Doch nun weißt Du alles für Dein erstes Spiel!**
**Also viel Spaß dabei!**

# 4 Ein weiterer Nutzen von Knoten: Vorder- und Hintergrund

Bei vielen Spielen liegen mehrere Spielelemente übereinander: Ganz hinten gibt es einen Hintergrund, davor läuft das Spielgeschehen ab, und manchmal gibt es noch ein paar Dinge, die davor liegen, zur Verzierung oder zum bewussten Blockieren der Sicht auf das Spiel.

Das lässt sich auch mit Knoten realisieren:

- Zuerst wird ein Knoten für den Hintergrund an der Wurzel angemeldet. An diesem Knoten werden alle Hintergrundelemente angelegt.
- Danach wird ein weiterer Knoten für den Vordergrund angemeldet. An diesem werden alle Vordergrundelemente angelegt.




*2 Bildebenen durch 2 Knoten. Mit mehr Knoten lassen sich natürlich beliebige weitere Ebenen schaffen.*

Dieses Prinzip lässt sich weiterführen für so viele Ebenen, wie man benötigt. Jede Ebene hat einen Knoten, der alle Elemente derselben sammelt. Jetzt müssen die Knoten aller Ebenen nur noch an der Wurzel angemeldet werden, und zwar in fester Reihenfolge: *Die unterste Ebene zuerst, die oberste Ebene zuletzt*.

# V Optionale Funktionen

## A Ein kurzes Vorwort

Hier wirst Du nun viele einzelne, Kapitel finden, die je ein abgeschlossenes Thema der Engine Alpha behandeln.

Du wirst hier viele interessante Highlights finden, mit denen Du Deinen Spielen eine professionellere Note verleihen kannst, unter anderem sind dies:

- Wiedergabe von Sound-Dateien
- Animationen
- Einbringen einer Maus
- Schreiben und Lesen von Dateien

Im Inhaltsverzeichnis findest Du einen Überblick und kannst nach Belieben stöbern und für deine Spiele deine eigenen Akzente setzen. Parallel zu den Kapitel solltest du auch die *Dokumentation* genau betrachten! In der Dokumentation wird alles am exaktesten beschrieben. Außerdem ist sie weit umfangreicher.

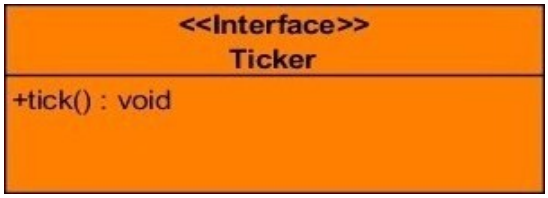
# B Das Ticker-System (Multitasking)

Ein wichtiges Konzept der Engine Alpha, neben dem Grafiksystem, ist das *Ticker-Konzept*. Es soll Multitasking ermöglichen. Es ist zwar nicht unbedingt nötig, um ein Spiel zu programmieren, wird jedoch *bei den meisten Spielen* gebraucht.

## 1 Das Ticker-Interface

Ein Ticker ist etwas, das immer wieder periodisch einen *Tick* macht.
Hierbei ist ein solcher *Tick* eine Methode, die immer wieder aufgerufen wird.

Ein Ticker ist als ein Interface mit einer einzigen abstrakten Methode definiert:




*Das Ticker-Interface*

Damit kann jede Klasse dieses Interface implementieren und damit einen *Tick* haben. In dem Tick kann ein Spielschritt gemacht, ein Punktezähler erhöht oder ein Spielzustand geändert werden; das Ticker-Prinzip ist sehr vielseitig.

Ein Ticker sorgt im Spiel meist für Bewegung.

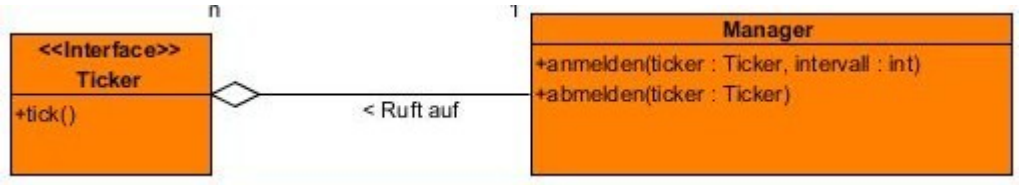
Nun schreiben wir eine einfache Klasse, die dieses Interface implementiert:

```
import ea.*;

/**
 * Dieser Ticker zaehlt aufwaerts und schickt jede Runde eine Nachricht an
 * die Konsole
 */
public class MeinTicker
implements Ticker //Implementieren des Ticker-Interfaces!
{
    /**Der Rundenzaehler*/
    private int zaehler;

    /**Der Konstruktor*/
    public MeinTicker() {
        zaehler = 0;
    }

    /**
     * Die tick()-Methode. Sie ueberschreibt die abstrakte tick()-Methode des
     * <code>Ticker</code>-Interfaces.<br />
     * Diese Methode soll in Regelmaessigen Abstaenden aufgerufen werden.
     */
    public void tick() {
        zaehler = zaehler + 1; //Rundenzaehler um 1 erhoehen
        System.out.println("Dies ist Runde " + zaehler + ".");
    }
}
```

Dieser Ticker sendet bei jedem *Tick* eine neue Konsolennachricht mit Angabe der aktuellen Runde.

## 2 Den Ticker „zum Laufen bringen“

Leider funktioniert dieser Ticker noch nicht, denn niemand ruft ihn regelmäßig auf. Dieser Ticker muss der Engine bekannt gemacht werden. Hierfür gibt es eine Klasse, die die Ticker aufruft. Diese Klasse ist die Klasse `Manager`.




Über die Methoden `anmelden` und `abmelden` kann ein Ticker an einem Manager gestartet und angehalten werden.

Bei `anmelden` wird zusätzlich ein Parameter vom Typ `int` verlangt. Dieser gibt an, in wie großen Abständen **in Millisekunden** die tick()-Methode des Tickers aufgerufen werden soll.

Praktischerweise hat die Klasse `Game` bereits ein Manager-Objekt mit dem Namen „`manager`“.

-

Dieses kann auch in der im Kapitel Die spielsteuernde Klasse erstellten Klasse `MeinSpiel` verwendet werden. Dies wäre der veränderte Konstruktor:

```
/**
 * Konstruktor, erstellt das Spielfenster und alle Hintergrundressourcen in der
 * Klasse <code>Game</code>
 */
public MeinSpiel() {
    super(400, 300); //Aufruf des Konstruktors der Klasse Game; erstellt ein
    //Fenster der Breite 400 und Hoehe 300

    MeinTicker ticker = new MeinTicker(); //Erstellen des Tickers
    manager.anmelden(ticker, 200); //Anmelden und Starten des Tickers beim
                                   //Manager
    //Die Referenz auf diesen manager liegt in der Klasse Game
}
```

Nun wird beim Erstellen des Spiels ein Ticker erstellt und angemeldet und seine Ausgabe wird an der Konsole gezeigt.

Anhalten kann man den Ticker einfach über den Aufruf der folgenden Methode:

```
manager.anhalten(ticker);
```

Nun weißt Du alles, um das Ticker-System und eigene Ticker in Deinem Spiel zu verwenden. Natürlich führen die Ticker in einem richtigen Spiel wichtige Aufträge aus, und zählen nicht einfach einen Wert hinauf. Typische Aufgaben für einen Ticker sind:

- Ein Spielobjekt bewegen
- Einen Kollisionstest machen
- Spielpunkte rauf-/runterzählen
- usw.

## PROBLEME?

Ein Beispiel-Projekt, das Ticker verwendet, kannst du von der EA-Website herunterladen: Hier.

# C Figuren

Ein Spiel, das nur mit unbewegten Bildern oder gar nur mit geometrischen Objekten funktioniert, ist für jemanden, der es nicht programmieren, sondern nur spielen will, meist nicht sonderlich reizvoll.

Deshalb gibt es in der Engine Alpha die Möglichkeit, ein ganz besonderes grafisches Element einzubinden, nämlich *Figuren*.

Eine Figur ist ein animiertes Bild.
Ähnlich wie bei einem Daumenkino besteht eine Figur aus mehreren Bildern, die immer wieder hintereinander durchgewechselt werden, wobei immer eines sichtbar ist.

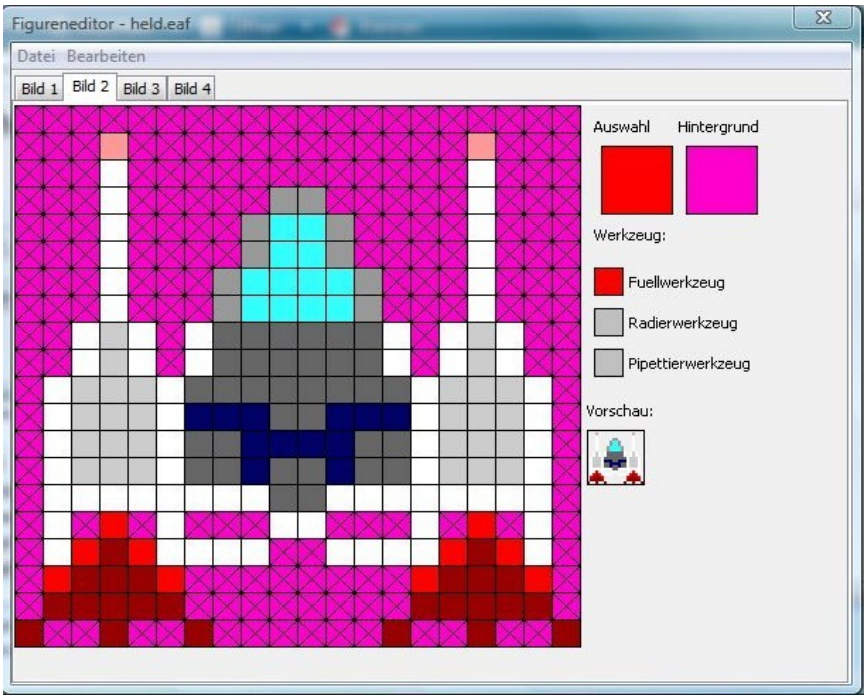
Jedes dieser Bilder besteht aus einer Fläche von Unterquadraten, ähnlich wie ein kariertes Blatt.
Jedes dieser Unterquadrate kann leer sein (dann ist es unsichtbar) oder eine Farbe haben (dann wird dieses Unterquadrat mit dieser Farbe gefüllt.
Die Länge aller Unterquadrate der Figur lässt sich immer variabel ändern, auch während das Spiel läuft, in Vielfachen der Pixellänge.

Um eine Figur in Dein Spiel zu bringen, musst Du zwei Schritte ablaufen:

1. *Die Figur Erstellen und Speichern*
2. *Die Figur Laden und ins Spiel Einbringen*

# 1 Erstellen einer Figur

Hierzu gibt es den *Figureneditor*.
Dies ist ein zusätzliches Programm, mit dem man Figuren erstellen kann. Dieser ist im Programmierkit enthalten, solltest Du ihn nicht bereits entdeckt haben. Die Datei kann auf Windows direkt mit Doppelklick geöffnet werden.
Auf Linux muss die Figureneditor-Datei zunächst ausführbar gemacht werden. Dann kann über Rechtsklick „Ausführen mit..“ aufgerufen werden und das Hauptfenster des Editors startet. Von dort aus kannst Du eigene Figuren neu erstellen oder öffnen.




*Der Figureneditor Version 0.2 – Nicht gefüllte Felder werden schwarz gekreuzt. Derzeitige Editor-Version ist 1.0*

Hier kann man die einzelnen Bilder befüllen und die Figur erstellen. Ist sie fertig, muss sie als „.eaf“-Datei (Engine Alpha Figur) gespeichert werden und kann dann ins Spiel geladen werden.

Achtung!
Die Figur sollte möglichst flächendeckend das Unterquadratfeld einnehmen, denn für Kollisionstests wird immer angenommen, dass die Figur das ganze Feld ausfüllt.

## 2 Laden einer Figur und deren Einbringung ins Spiel

Geladen wird die Figur über den folgenden Konstruktor:

```
public Figur(int x, int y, String pfad)
```

Wobei die Parameter `x` und `y` - wie immer – die Koordinate der linken oberen Ecke der Figur angeben und der Parameter `pfad` das Verzeichnis, unter dem die entsprechende Figurdatei zu finden ist.

Ins Spiel gebracht werden kann diese Figur dann wie ein Bild oder ein Rechteck, denn auch diese Klasse leitet sich aus Raum ab und lässt sich somit behandeln wie alle anderen grafischen Elemente auch.

Das volle Laden und Einbringen soll dieser Quellcode verdeutlichen:

```
/*Irgendwo in der spielsteuernden Klasse*/
Figur figur = new Figur(30, 30, "rakete.eaf");
wurzel.add(figur);
```

Die Datei „rakete.eaf“ befindet sich im Projektordner dieses Projekts.
Das Beispiel ist als funktionierendes Programm einsehbar.

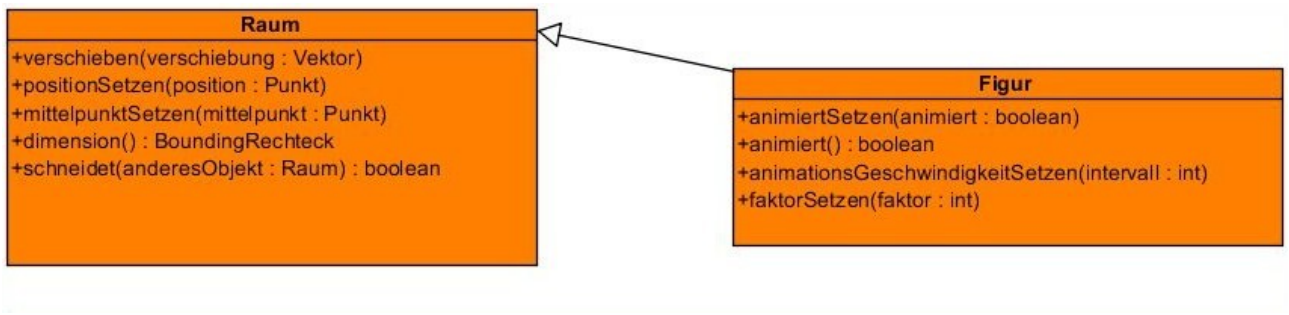
Die Animation eines Bildes lässt sich problemlos starten und anhalten über die folgende Methode:

```
public void animiertSetzen(boolean animiert)
```

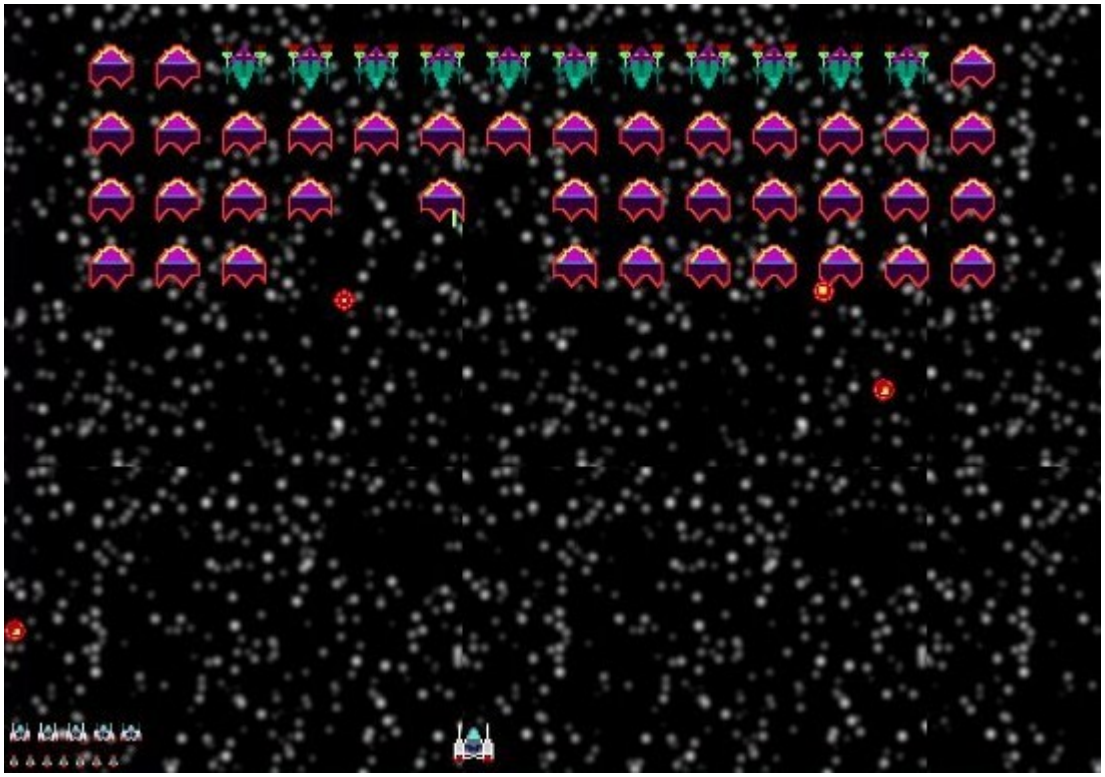
Über diese Methode lässt sich die Geschwindigkeit, in der die Animationsbilder wechseln, ändern:

```
public void animationsGeschwindigkeitSetzen(int intervall)
```

Diese Methode ändert die Animationsgeschwindigkeit dieser Figur.
Sobald die eingegebene Zeit in Millisekunden (das `intervall`) abgelaufen ist, wird das sichtbare Bild der Figur gewechselt.




*Klassenkarten von `Raum` und `Figur` mit den relevanten Methoden*



*Dieses Spiel besteht (außer dem Hintergrund) nur aus Figuren*

# 3 Figuren modifizieren

Man kann Figuren sehr leicht verändern – **zur Laufzeit** und beliebig oft!

Du kannst mit einer Figur folgende Veränderungen machen:

- Die Größe verändern
- Sie spiegeln
- Ihre Farben ändern

## *i Die Größe von Figuren ändern*

Jede Figur hat einen *Größenfaktor*.
Der Größenfaktor gibt an, wie groß ein Unterquadrat in Pixeln ist. Du kannst diesen Größenfaktor über den Aufruf folgender Methode ändern:

```
public void faktorSetzen(int faktor)
```

Der Wert `faktor` kann beliebige Werte über 0 annehmen.
Dies ist die einfachste Form der Veränderung einer Figur.

## *ii Die Figur spiegeln*

Eine Figur kann auf 2 Arten gespiegelt werden: waagrecht oder senkrecht.

Hierfür gibt es je eine Methode:

Diese Methode spiegelt an einer imaginären X-Achse, also *waagrecht*:

```
public void spiegelXSetzen(boolean gespiegelt)
```

Diese Methode spiegelt an einer imaginären Y-Achse, also *senkrecht*:

```
public void spiegelYSetzen(boolean gespiegelt)
```

### *iii Farben der Figur ändern*

Die Farben einer Figur lassen sich auf verschiedene Arten ändern.

#### Die Figur aufhellen und abdunkeln

Eine Figur kann „heller“ werden. Dies funktioniert über folgende Methode:

```
public void heller()
```

Es gibt auch eine Methode, um eine Figur abzudunkeln:

```
public void abdunkeln()
```

***Achtung!***
`heller()` und `dunkler()` sind nicht symmetrisch! Das bedeutet: Wenn du `heller()` aufrufst und später `dunkler()`, muss dies nicht bedeuten, dass deine Figur genauso aussieht wie vorher. Minimale Veränderungen sind möglich, weil hier intern gerundet wird.

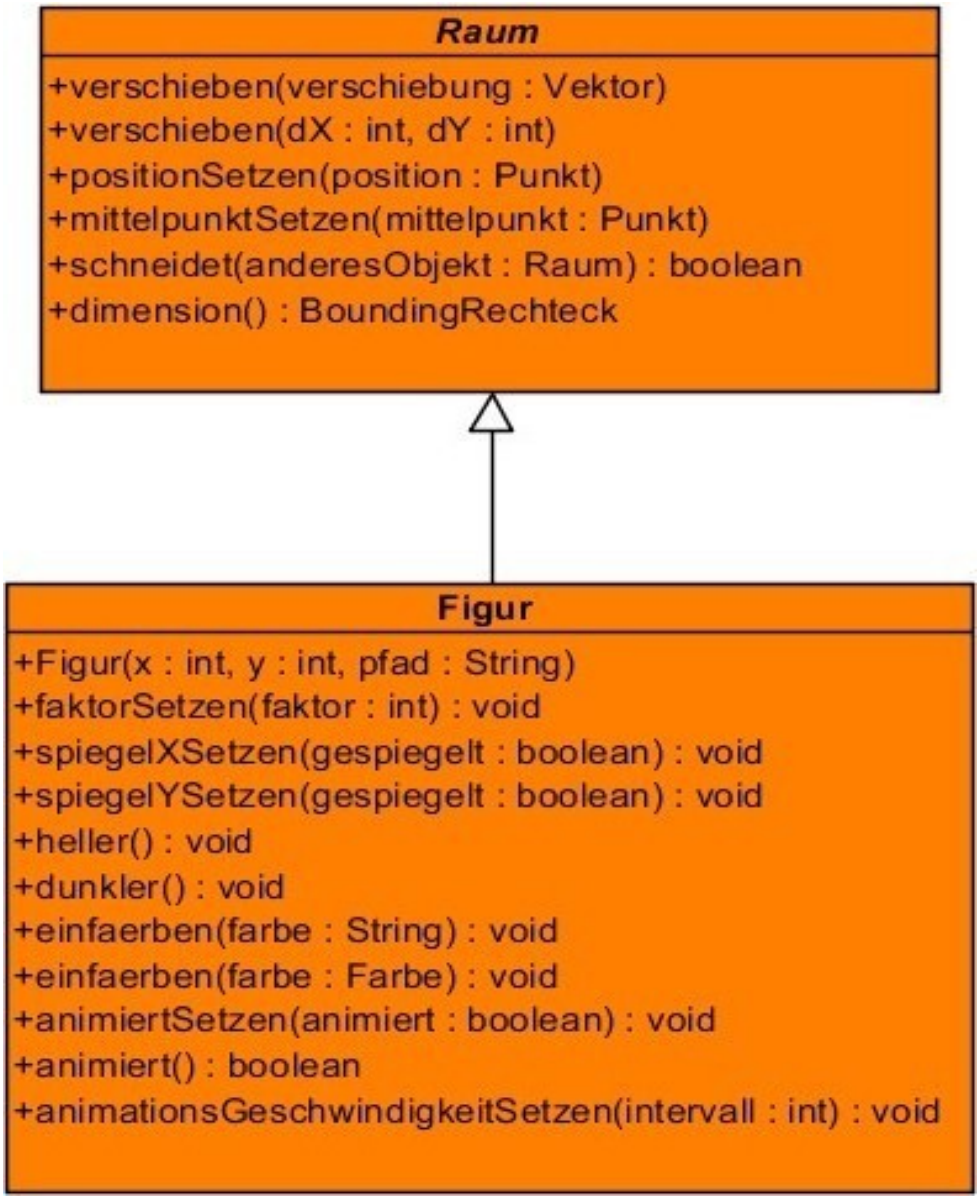
#### Die Figur einfärben

Man kann eine Figur auch einfach vollkommen einfärben. Das bedeutet, dass jedes gefüllte Unterquadrat dieselbe Farbe bekommt.
Diese Methode lautet folgendermaßen:

```
public void einfaerben(String farbe)
```

Hiermit lässt sich die Figur einfärben.
Willst du „speziellere“ Farben als die Standard-Farben kannst du anstatt einem `String`- auch ein `Farbe`-Objekt eingeben. Was das ist und wie es funktioniert, erläutert das Kapitel *Farben*.

### iv Das geht alles: Zusammenfassung




*Klassenkarte der Klasse `Figur` – inklusive der Klasse `Raum`, aus der sich diese ableitet*

Hier siehst du einmal eine recht umfangreiche Klassenkarte zur Klasse `Figur` – inklusive Vererbung. Alle hier festgehaltenen Methoden wurden in diesem Kapitel behandelt.

## PROBLEME?

Ein Beispiel-Projekt, das das Laden und modifizieren einer einfachen Figur demonstriert, kannst du von der EA-Website herunterladen: Hier.

# *D Sound*

Sounds lassen sich in der Engine Alpha leicht darstellen; vergleichbar mit einem `Bild`.

## 1 Unterstützte Sound-Formate

Zunächst benötigst Du die abzuspielende Sound-Datei. Folgende Formate werden voll unterstützt:

- **.wav-Dateien**
  Gesampelter Sound. Jede beliebige andere gesampelte Sounddatei (Darunter fällt unter anderem auch .mp3 und .ogg) lässt sich schnell dahin umkonvertieren.
  Kennst Du hierfür kein passendes Programm, so empfehle ich Dir **Audacity**.
  Das Programm ist als Open-Source-Programm kostenlos im Internet vollständig - ohne Werbung oder Demolaufzeit – herunterladbar.

- **.mid-Dateien (MIDI-Sequenzen)**
  MIDI-Sequenzen sind anders gesagt „Notenpapier“ für den Computer, der das dann abspielt. Wahrscheinlich wirst Du eher weniger Kontakt mit MIDI haben, aber es ist unter Musikern, die an einem PC arbeiten ein Standard. Es klingt jedoch nicht so „echt“ wie gesampelter Sound (z.B. .wav-Dateien). Vergleichbar ist ein MIDI-Sound eher mit dem eines alten Gameboy-Spiels. Doch auch das hat manchmal seine Reize.
  Hast Du kein Programm dafür, möchtest es aber nutzen, kann ich Dir den **Powertab Editor** (Notensetzung nach Gitarrentabulatur) empfehlen.

Leider werden .mp3-Dateien noch nicht unterstützt, da Java hierfür standardmäßig nicht gerüstet ist. Jedoch lassen sich .mp3-Dateien sehr leicht in .wav-Dateien umwandeln. Hierfür kann ich einmal mehr die kostenlose und GNU-GPL-lizenzierte Software **Audacity** sehr empfehlen.

Hast Du nun eine passende Sound-Datei, musst Du diese nur in Deinen Projektordner einlegen.

## 2 Einbringen einer Sound-Datei

`Sound` ist auch eine Klasse in der Engine Alpha.
Damit bringst Du die Sound-Datei in Dein Programm ein.

Die Klasse `Sound` hat nur einen Konstruktor:

```
public Sound(String dateiname)
```

Der Dateiname ist der Name Deiner Sound-Datei im Projektordner.
Somit lädst Du Deine eigene Sounddatei zum Beispiel so:

```
Sound meinSound = new Sound("meineSoundDateiImProjektordner.wav");
```


*Schön einfach: Die Klassenkarte der Klasse `Sound`*

Dadurch wird der Sound aber nicht abgespielt.
Der Sound wurde geladen und ist abspielbereit.
Dieser Konstruktor funktioniert sowohl für MIDI- als auch für gesampelte Sounds.

Abgespielt werden sie durch den `AudioManager`.

`AudioManager` ist die Klasse, die beliebig viele Sounds gleichzeitig abspielen kann.

Ein Objekt hiervon ist bereits in der Klasse `Game` und von Deiner spielsteuernden Klasse erreichbar; unter dem Namen „`audioManager`“.

Zum **Abspielen** hat die Klasse `AudioManager` die folgende Methode:

```
public void abspielen(Sound sound, boolean wiederholen)
```

| Parameter | Erläuterung |
|---|---|
| `sound : Sound` | Das `Sound`-Objekt, das abgespielt werden soll. |
| `wiederholen : boolean` | Ob die Wiedergabe wiederholt werden soll.<br>Ist dieser Wert `true`, wird der Sound in einer Dauerschleife abgespielt.<br>Ist dieser Wert `false`, wird der Sound angehalten, nachdem er einmal vollständig durchgelaufen ist. |

Bei dieser Methode kann man auf den zweiten Parameter verzichten, in diesem Fall wird der abzuspielende Sound *nicht wiederholt*.

Zum **Anhalten** eines Sounds gibt es auch eine Methode:

```
public void anhalten(Sound sound)
```

| Parameter | Erläuterung |
|---|---|
| `sound : Sound` | Dieser Sound soll angehalten werden.<br>Wird dieser Sound zur Zeit nicht abgespielt, so passiert gar nichts. |

Diese Methode spult den Sound auch zurück, sodass er beim nächsten Abspielen wieder von vorne abgespielt wird.

Soll er nur **pausiert** werden, damit er beim nächsten Abspielen da abgespielt wird, wo er angehalten wurde:

```
public void pausieren(Sound sound)
```

Zum besseren Verständnis kannst Du das dazugehörige *BlueJ-Projekt-Beispiel* einsehen.


*Klassenkarte der Klasse AudioManager*

## 3 Sound-Dateien in Ordnern

Benutzt du viele Sound-Dateien, ist es vielleicht unübersichtlich, alle „einfach“ in deinem Projektordner zu lagern. Du kannst natürlich auch einen neuen „Material“-Ordner anlegen und dann im `Sound`-Konstruktor den *Pfad* angeben, über deinen Ordner zu der gewünschten Sound-Datei.

Möchtest du dies nutzen, kannst du dich darüber im Kapitel Dateien Schreiben und Lesen – Arbeiten mit Verzeichnissen informieren.

PROBLEME?

Ein Beispiel-Projekt, das einen Sound aus einem Verzeichnis lädt und einbringt, kannst du von der EA-Website herunterladen: Hier.

## 4 Sounddateien im exportierten Projekt

Beim Exportieren des Projektes in eine .jar-Datei gehst du folgendermaßen vor:

1. Vor dem Exportieren, entferne alle Sound-Dateien aus dem Projektordner
2. Nach dem Exportieren, bringe alle Sound-Dateien in das selbe Verzeichnis wie die ausführbar .jar-Datei. Hast du Ordner genutzt, so bringe die Ordner in dieses Verzeichnis. Mehr hierzu im Kapitel *Das Projekt exportieren*.

# E Die Kamera

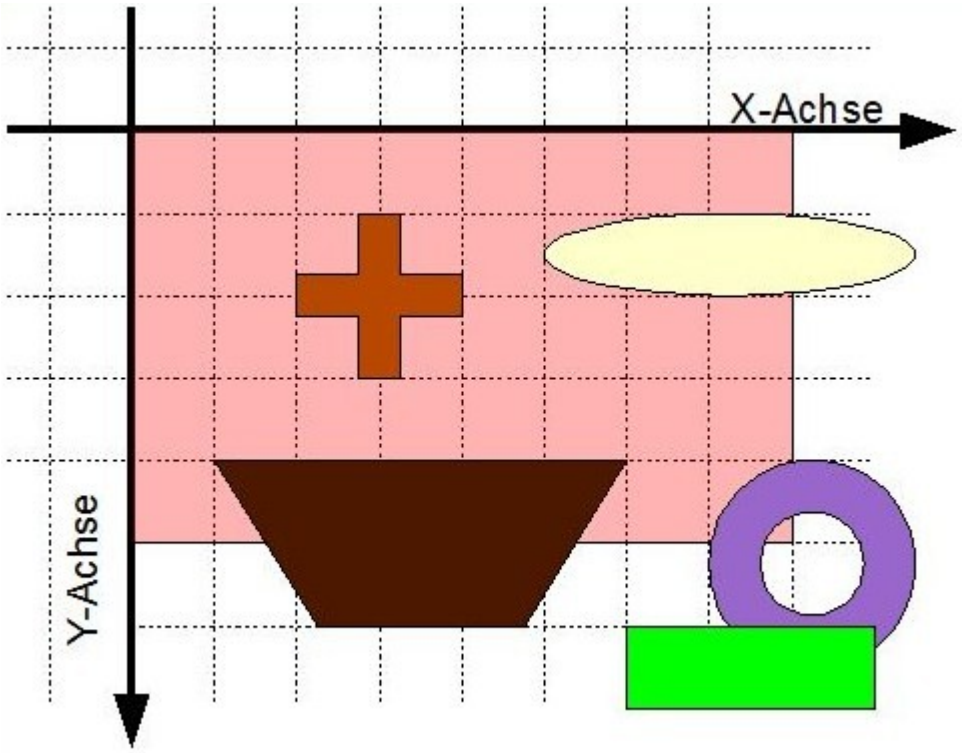
## 1 Was ist die Kamera?

Spiele wie Pac-Man oder Pong haben einen statischen Bildschirm.
Aber bei weit mehr Spielen bewegt sich das Bild, geht zum Beispiel mit der Hauptfigur und bleibt nicht statisch.

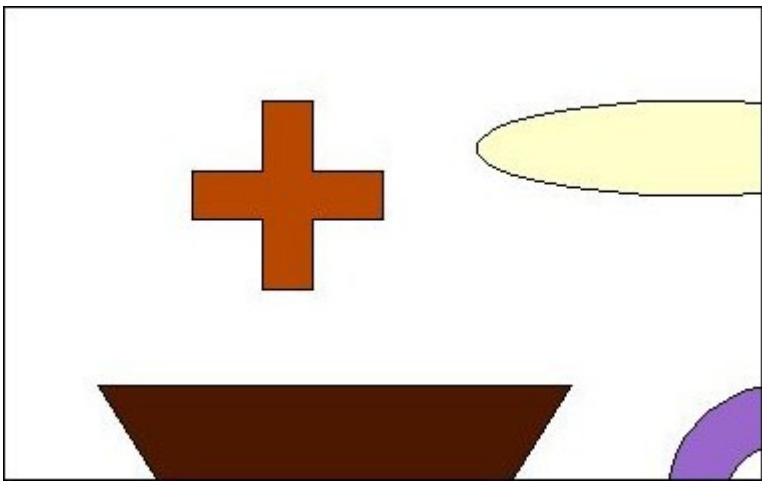
Ich hole mal etwas weiter aus:
Die ganze *Zeichenebene* ist unendlich weit in alle Richtungen, aber das Spielfenster nicht. Es kann also immer nur ein begrenzter rechteckiger Bereich der Zeichenebene im Fenster dargestellt werden.

Dieser Bereich wird bestimmt von der *Kamera*.
Die Kamera blickt auf die Zeichenebene. Sie „stanzt“ ein Rechteck mit den Maßen des Fensters aus und projiziert dies auf den Bildschirm. Der Rest der Zeichenebene ist also nicht sichtbar.

Dies macht ein Objekt der Klasse `Kamera`.
In der Klasse `Game` gibt es eine Referenz auf die aktive Kamera des Spiels und diese ist so auch in der eigenen spielsteuernden Klasse über den Namen „`cam`“ erreichbar.




*Die Kamera auf der Zeichenebene...*



*... und das Ergebnis auf dem Spielbildschirm*

## 2 Verschieben der Kamera

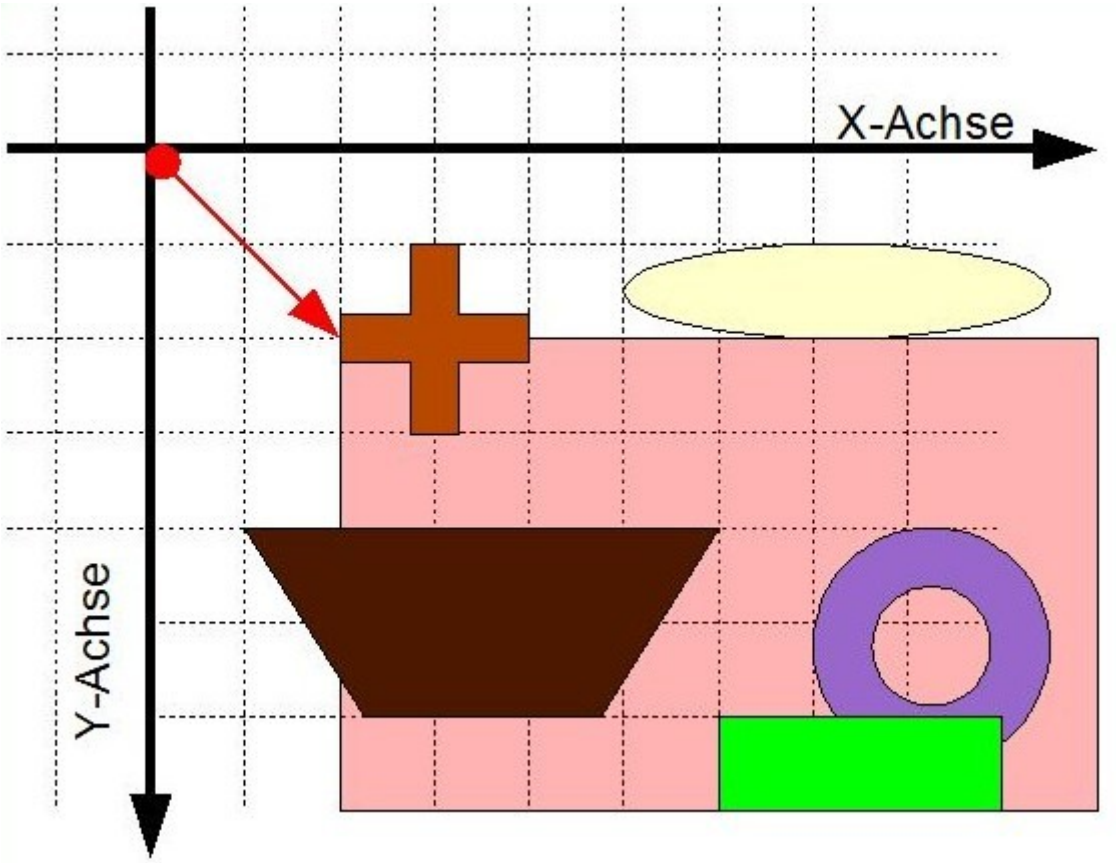
Die Kamera lässt sich einfach verschieben.
Hierfür gibt es die selbe Methode, wie auch für ein `Raum`-Objekt:

```
public void verschieben(Vektor verschiebung)

public void verschieben(int x, int y)
```

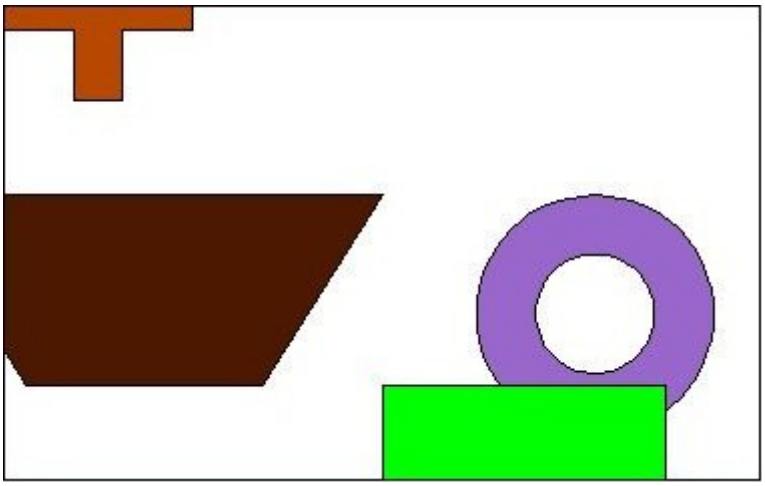
Hiermit verschiebt man die Kamera.
Im folgenden wird die Kamera um (2|2) verschoben. Dies ist allerdings eine kaum merkbare Verschiebung. Man kann natürlich auch größere Verschiebungswerte wählen.

Das würde im Spiel so realisiert werden:

```
/*
 * Irgendwo in der spielsteuernden Klasse...
 */

cam.verschieben(new Vektor(2, 2));

//Oder alternativ:
cam.verschieben(2, 2);
```




*Die Kamera wurde um (2|2) verschoben...*



*... und dadurch ändert sich das Bild im Fenster*

Die Kamera lässt sich also mit diesen Methode einfach und beliebig verschieben.

# 3 Fokus

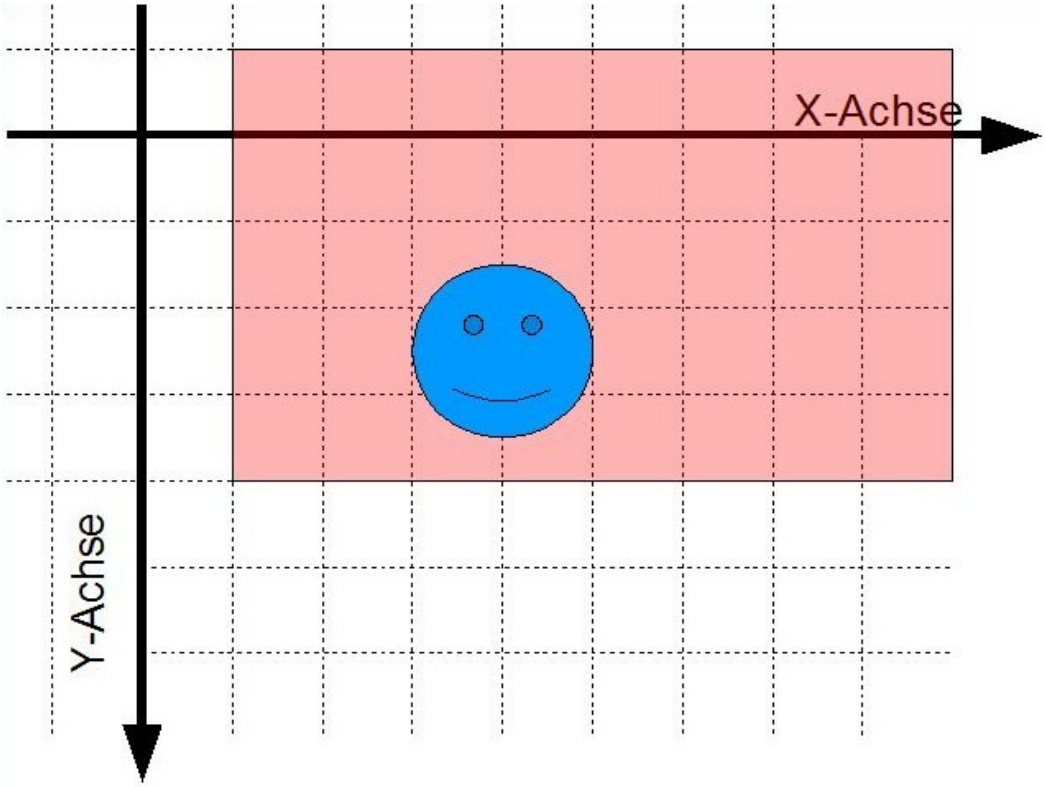
Kaum ein Spiel verlangt eine unabhängige Bewegung der Kamera.
Fast immer ist ein bestimmtes Objekt, das gesteuert wird, dauerhaft im Zentrum.

Diese *Fokus-Funktion* steht auch in der Engine Alpha zur Verfügung.

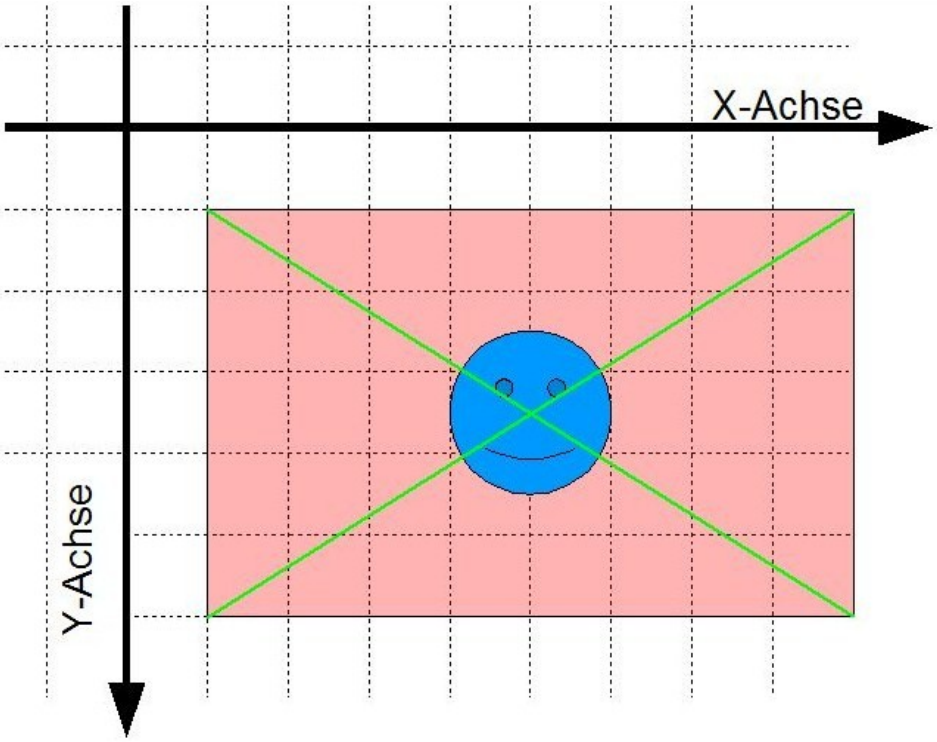
Hierfür gibt es die folgende Methode:

```
public void fokusSetzen(Raum fokusObjekt)
```

Damit übergibt man der Kamera ein neues Fokus-Objekt. Dieses Objekt wird fortan immer in der Mitte des Bildes sein.




*Die Kamera und ein Objekt ohne Fokuseinstellung*




*Das Objekt wurde in den Fokus genommen*

Die Fokuseinstellung lässt sich auch problemlos wieder beenden:

```
public void fokusLoeschen()
```

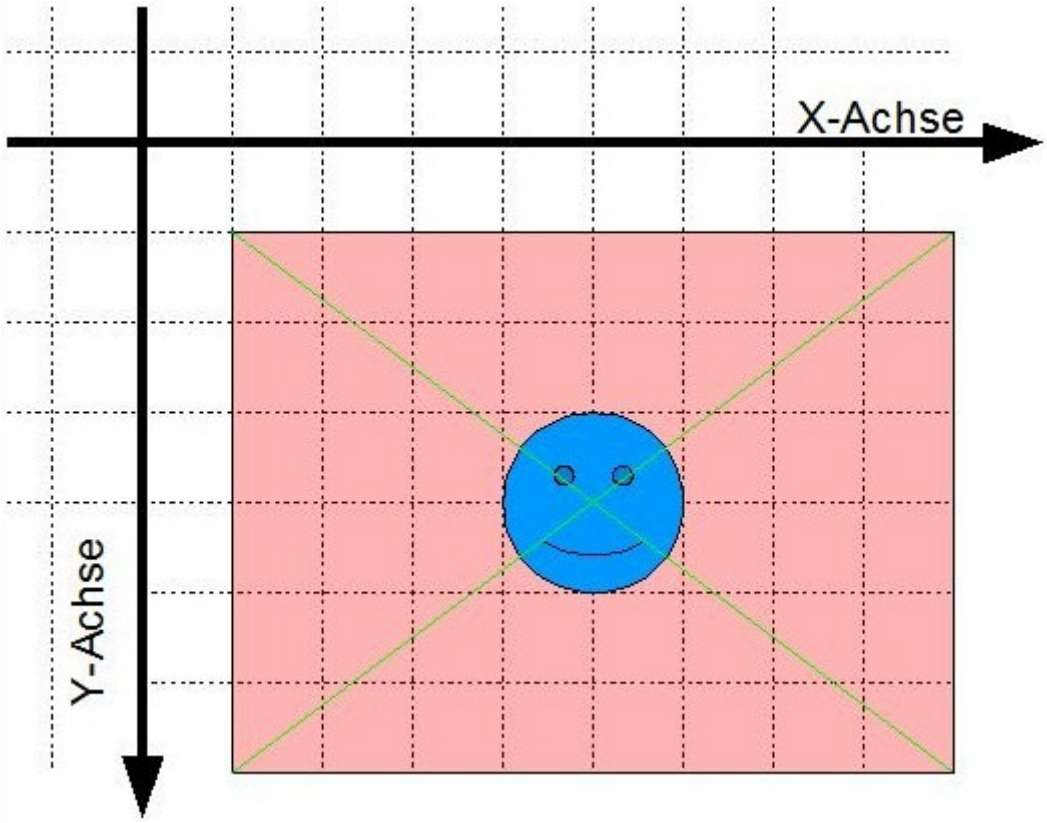
Ab dem Aufruf dieser Methode wird das Fokusobjekt nicht mehr in seiner Bewegung verfolgt und die Kamera bleibt stehen.

Zusätzlich kann man das Fokusverhalten spezifizieren:

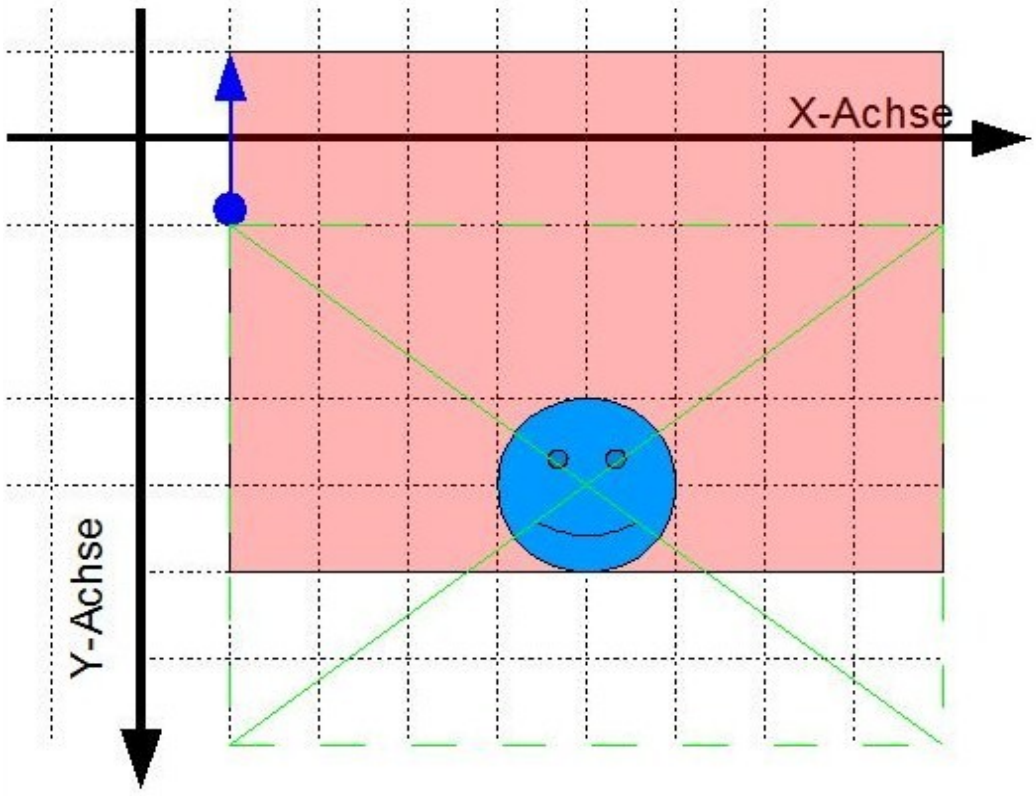
Zum einen muss der Fokus nicht immer exakt im Zentrum des Bildes sein, über den Aufruf der folgenden Methode kann man einen Verzug setzen:

```
public void fokusVerzugSetzen(Vektor verzug)
```

Hierbei wird die Kamera immer um den Verzug vom Zentrum des Fokus-Objektes wegbewegt.




*Kamera ohne Fokus-Verzug. Das Fokus-Objekt ist genau in der Mitte*




*Kamera mit Fokusverzug von (0|-2) – Siehe Vektor links oben. Soviel wird das Bild vom eigentlichen Zentrum (gestrichelt) wegbewegt. Diese Einstellung bietet sich z.B. für Jump n' Run-Spiele an*

# 4 Grenzen für die Kamera

Auch muss die Kamera dem Fokusobjekt nicht bis ins Nirvana folgen – oder sich manuell bis zur Unendlichkeit verschieben lassen; man kann der Kamera eine „Grenze“ vorschreiben, über die sie niemals gehen wird. Diese Grenze wird als `BoundingRechteck` beschrieben.

```
public void boundsSetzen(BoundingRechteck grenzen)
```

Dieses `BoundingRechteck` beschreibt die Fläche auf der *Zeichenebene*, die diese Kamera niemals übertreten wird. Aber: Die Grenzen der Kamera müssen natürlich ausreichend groß sein. Ein Spielraum mit der Höhe 10 Pixel und der Breite 2000 Pixel funktioniert nicht, denn die Kamera selbst ist ja (im Regelfall) mit einer sehr viel größeren Breite (nämlich die des Spielfensters) ausgestattet, daher gilt:

*Sinnvolle Werte für die Kamera-Bounds sind solche, bei denen die Höhe und Breite größer als die des Spielfensters ist.*


*Jetzt, wo du alle Methoden kennst: Die Klassenkarte der Kamera*

# 5 Es muss sich ja nicht alles verschieben: Statische grafische Objekte

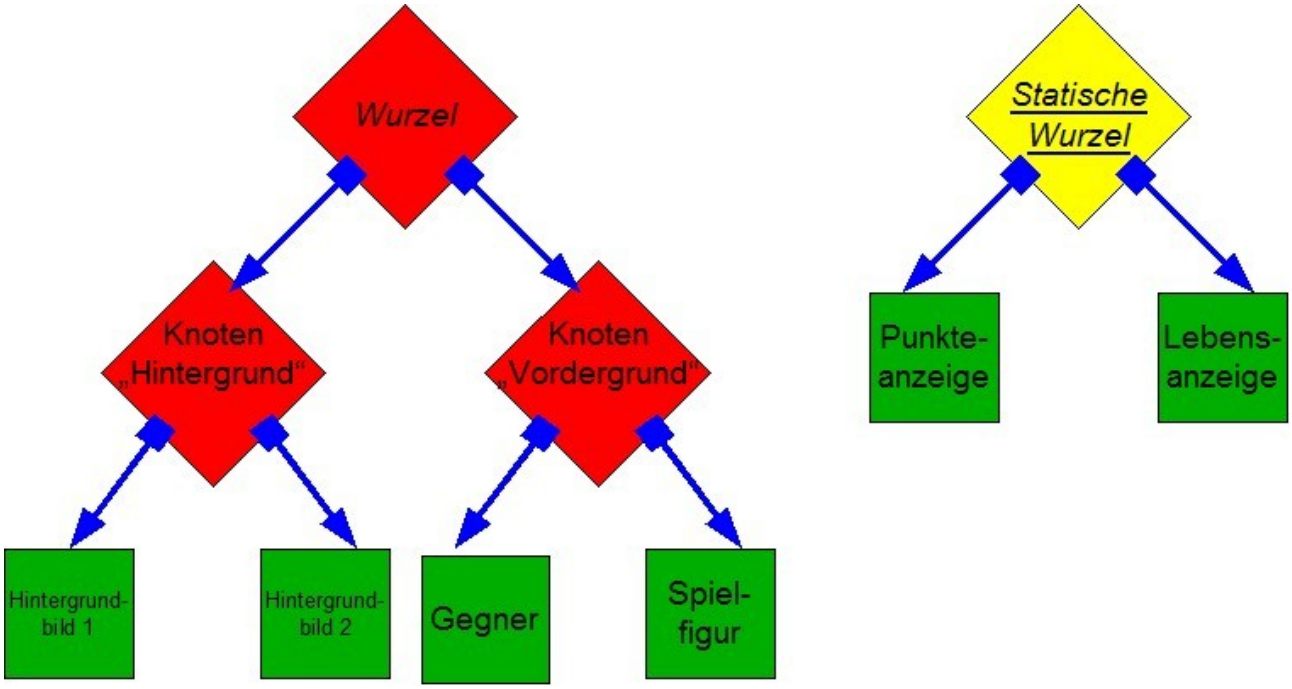
Bei einigen grafischen Objekten macht es keinen Sinn, dass sie sich mit der Kamera verschieben. Dies sind zum Beispiel: Level- und Punkteanzeigen, Lebensenergiebalken, Textnachrichten oder -beschreibungen etc.
***Damit sich diese grafischen Elemente nicht verschieben***, wenn sich die Kamera bewegt, gibt es hierfür eine besondere Möglichkeit:

| *Die statische Wurzel* |
|---|

Du kennst bereits die normale Wurzel. An dieser werden *alle normalen Spielelemente* angelegt. Doch für alle besonderen Spielelemente, die sich nicht mit der Kamera verschieben sollen, gibt es die statische Wurzel.
Auch auf die statische Wurzel kann man in der Klasse `Game` zugreifen. In deiner spielsteuernden Klasse kannst du auf die statische Wurzel direkt zugreifen. Die Referenz dafür heißt.

| `statischeWurzel` |
|---|

Füge hieran einfach alle deine Punkteanzeigen, Spielinformationen etc., von denen du nicht willst, dass sie sich mit der Kamera verschieben.




*Wurzel und statische Wurzel in der Praxis: **Beide** zeichnen alle anliegenden Objekte, **aber** die Elemente an der statische Wurzel liegen immer ganz oben und werden nicht durch die Kamera verschoben.*

## PROBLEME?

Ein Beispiel-Projekt, das einige Möglichkeiten der Kamera demonstriert, kannst du von der EA-Website herunterladen: Hier.

# F Dateien Schreiben und Lesen

Die meisten Informationen bei einem Spiel werden nicht wieder gebraucht, nachdem das Spiel beendet wird. Bei einem einfachen Spiel ist dies vor allem der ganze Spielaufbau. Dieser wird bei jedem Start von neuem erledigt. Das wird also nicht extra gespeichert.

Ausnahmen sind zum Beispiel Spielstände, Highscores oder Verlaufslisten.
Dies sind Dinge, die beim nächsten Spielstart nicht wieder so sein sollen wie beim letzten. Deshalb werden diese Informationen für den nächsten Spielstart *gespeichert*. Es wäre z.B. schade, wenn der Highscore mit der virtuellen Maschine von Java beim schließen des Spiels untergeht).

Deshalb kann man in der Engine Alpha *Daten speichern und laden*.

Die Daten können nur als Feld (oder auch genannt: Array) gespeichert werden, und zwar folgende Array-Typen:

- `int`-Arrays
- `String`-Arrays

Diese Daten werden *in einem eigenen Dateiformat* der Engine Alpha gespeichert, nämlich dem **.eaa-Format**.

Die hierfür verantwortliche Klassse, ist die Klasse `DateiManager`.

Die Klasse `DateiManager` besitzt für alle nötigen Operationen *statische Methoden*.
Das heißt **diese Methoden führt man nicht über ein Objekt der Klasse** `DateiManager` **aus, sondern über die Klasse selbst**.

Du kennst vielleicht schon die statische Methode:

```
System.out.println("Hi, das schicke ich direkt an die Konsole!");
```

Genau so funktioniert das bei der Klasse `DateiManager`.
Aber auch hierzu wird es viele Beispielcodes geben.

## 1 Dateien Schreiben

Für das Schreiben eines `int`-Arrays (anders gesagt: eines `int[]`-Objektes) gibt es die Methode:

```
public static void integerArraySchreiben(int[] array, String dateiname)
```

Der Dateityp ist immer **.eaa**! Das ist ein spezielles Dateiformat für Arrays.

Hierfür ein kleines Beispiel:

```
/**
 * Diese Methode schreibt ein Integer-Array und fuellt es mit den
 * Quadratzahlen.
 * Anschließend wird das Array als Datei gespeichert.
 */
public void quadratzahlenSchreiben() {
    int[] quadr = new int[10];
    for(int i = 0; i < quadr.length; i++) {
        quadr[i] = i*i;
    }

    //Durch den Aufruf dieser Methode wird das Array als Datei gespeichert
    DateiManager.integerArraySchreiben(quadr, "Quadratzahlendatei.eaa");
}
```

Die Datei "Quadratzahlendatei.eaa" wird im Projektordner angelegt und gespeichert.

## 2 Dateien wieder Einlesen

Anschließend lässt sich die Datei natürlich auch wieder einlesen, und zwar folgendermaßen:

```
public static int[] integerArrayEinlesen(String dateiname)
```

Somit lassen sich die Quadratzahlen folgendermaßen einlesen:

```
/**
 * Diese Methode ließt die geschriebene Datei wieder ein.
 * Anschließend wird sie die einzelnen Zahlen an der Konsole ausgeben.
 */
public void quadratzahlenEinlesen() {
    //Die folgende Methode liesst das Array wieder ein und gibt es zurueck
    int[] quadr = DateiManager.integerArrayEinlesen("Quadratzahlendatei.eaa");
    for(int i = 0; i < quadr.length; i++) {
        System.out.println(quadr[i]);
    }
}
```

Genau das selbe funktioniert bei `String`-Arrays.
Hierfür gibt ebenfalls je eine Methode zum Ein- und Auslesen:

```
public static void stringArraySchreiben(String[] array, String dateiname)

public static String[] stringArrayEinlesen(String dateiname)
```

# 3 Arbeiten mit Verzeichnissen

Bei einem größerem Spiel, für das viele Dateien geladen werden müssen (zum Beispiel Figuren, hierzu siehe Kapitel „Figuren“), ist es vielleicht sinnvoll, in mehreren Ordnern alle Dateien systematisch zu ordnen, um Übersicht zu schaffen.

Jedoch gibt es hier ein Problem:

**Die Pfadtrenner-Zeichen sind auf jedem Betriebssystem anders.**

Dann ließe sich auf Windows ein Verzeichnis so als String darstellen.

```
String verzeichnis = "meinUnterordner\\nocheinordner\\Quadratzahlendatei.eaa";
```

Das Problem ist, dass, sobald das Spiel auf einem Linux-Rechner laufen soll, das selbe Verzeichnis so heißen müsste:

```
String verzeichnis = "meinUnterordner/nocheinordner/Quadratzahlendatei.eaa";
```

Das heißt nicht, dass Du für Windows und Linux jeweils das Spiel umschreiben musst, es gibt eine sehr leichte Möglichkeit, **immer den richtigen Pfadtrenner zu erhalten**:

Die Klasse `Game` hat ein öffentliches `String`-Attribut mit dem Namen „`pfadtrenner`“.
Somit lässt sich das Verzeichnis in *der spielsteuernden Klasse* allgemein gültig, unabhängig vom laufenden Betriebssystem angeben:

```
/*-------------Irgendwo in der spielsteuernden Klasse-------------*/

String verzeichnis =      "meinUnterordner" + pfadtrenner +
                          "nocheinordner"   + pfadtrenner +
                          "Quadratzahlendatei.eaa";

int[] quadratzahlen = DateiManager.integerArrayEinlesen(verzeichnis);
```

# 4 Laden und Speichern bei einem exportierten Projekt (.jar-Datei)

Ist das Spiel fertig und wird als .jar-Datei exportiert, muss folgendes beachtet werden:

Alle zu ladenden Dateien müssen im selben Ordner, wie die ausführbare .jar-Datei abgelegt werden, damit sie standardmäßig geladen werden können. Während des Exportes sollten die Dateien dann nicht im Porjektordner enthalten sein. Das spart Speicherplatz.
Das hieße für das obige Beispiel, dass der Ordner „meinUnterordner“ im selben Verzeichnis liegen müsste, in dem die ausführbare .jar-Datei liegt.

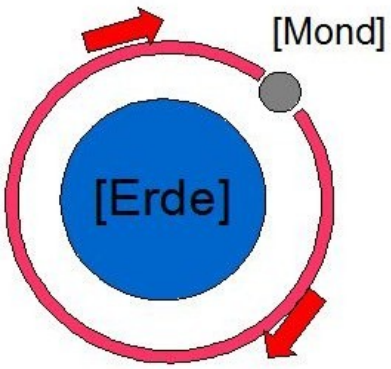
Genaueres hierzu und eine Anleitung für den Export findet sich
im Kapitel „Das Projekt exportieren“.

# *G* Animationen

## 1 Animationsarten

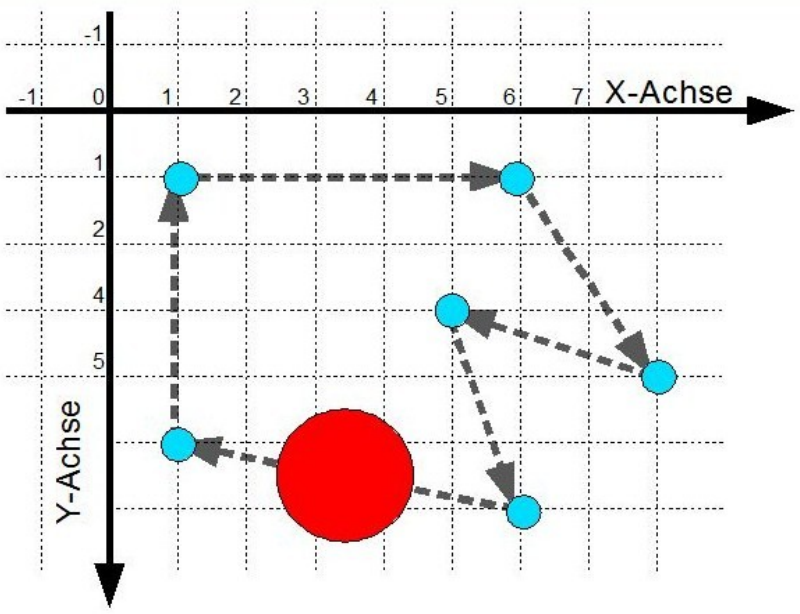
Animationen bringen mehr Bewegung ins Spiel.
In der Engine Alpha lassen sich leicht drei verschiedene Arten von Animationen regeln:

- **Kreisanimationen**
  Hierbei wird ein Objekt kreisförmig um einen bestimmten Punkt animiert.



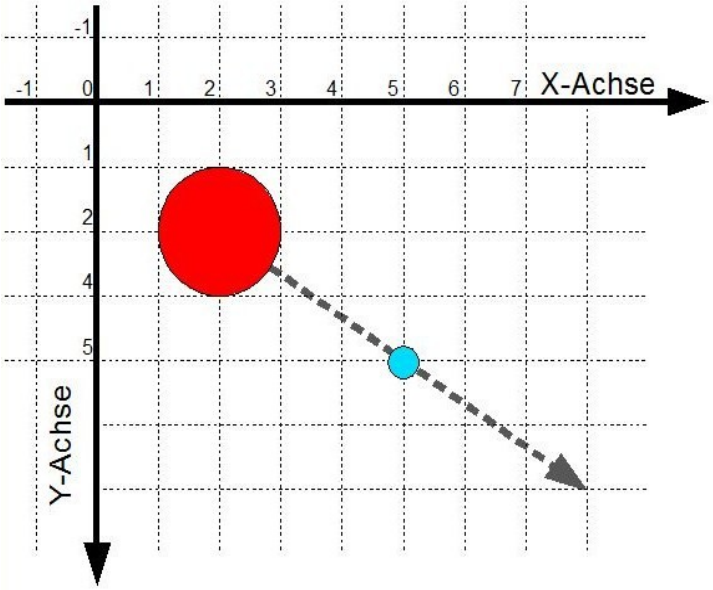

*Die Drehung des Mondes um die Erde aus dem Kapitel „Knoten“ ist eine Kreisanimation*

- **Streckenanimationen**
  Hierbei wird ein Objekt auf einem festgelegten Streckenzug zwischen einer beliebigen Anzahl an Punkten animiert.




*Der rote Kreis wird immer wieder auf einem Streckenzug zwischen 6 Punkten animiert.*

- **Geradenanimationen**
  Hierbei wird ein Objekt auf einer Gerade zwischen seinem Mittelpunkt und einem festgelegten Punkt bewegt.




*Der rote Kreis wird auf einer unendlich langen Gerade, bestimmt durch seinen Anfangs- sowie einen einzigen Orientierungspunkt animiert*

Animationen werden über die Klasse `AnimationsManager` eingeleitet.
Ein Objekt hiervon ist in der Klasse `Game` bereits bereitgestellt und sehr leicht in der spielsteuernden Klasse über den Namen „`animationsManager`“ erreichbar.

Für jede Animationsart gibt es überladene Methoden. Das bedeutet, dass es eine sehr umfangreiche Parameterzahl für eine Methode gibt, diese Methode aber auch mit weniger Parametern auskommt. *Wie die vereinfachten Methoden lauten, ist in der Dokumentation nachzulesen.*

## 2 Kreisanimationen

Der Methodenname lautet:

```
kreisAnimation
```

Die detaillierteste Methodenaufruf erwartet 4 Parameter:

| Parameter | Erläuterung |
|---|---|
| `ziel : Raum` | Dieses Objekt soll animiert werden. |
| `zentrum : Punkt` | Das Zentrum der Drehung; um dies dreht sich das zu animierende Objekt |
| `loop : boolean` | Wenn dieser Wert `true` ist, hört diese Drehung niemals auf. Ist er `false`, wird die Drehung beendet, wenn das zu animierende Objekt wieder an seiner Ausgangsposition ist. |
| `umlaufzeit : int` | Die Zeit **in Millisekunden**, die eine ganze Umdrehung dauert. |

# 3 Streckenanimationen

Der Methodenname lautet:

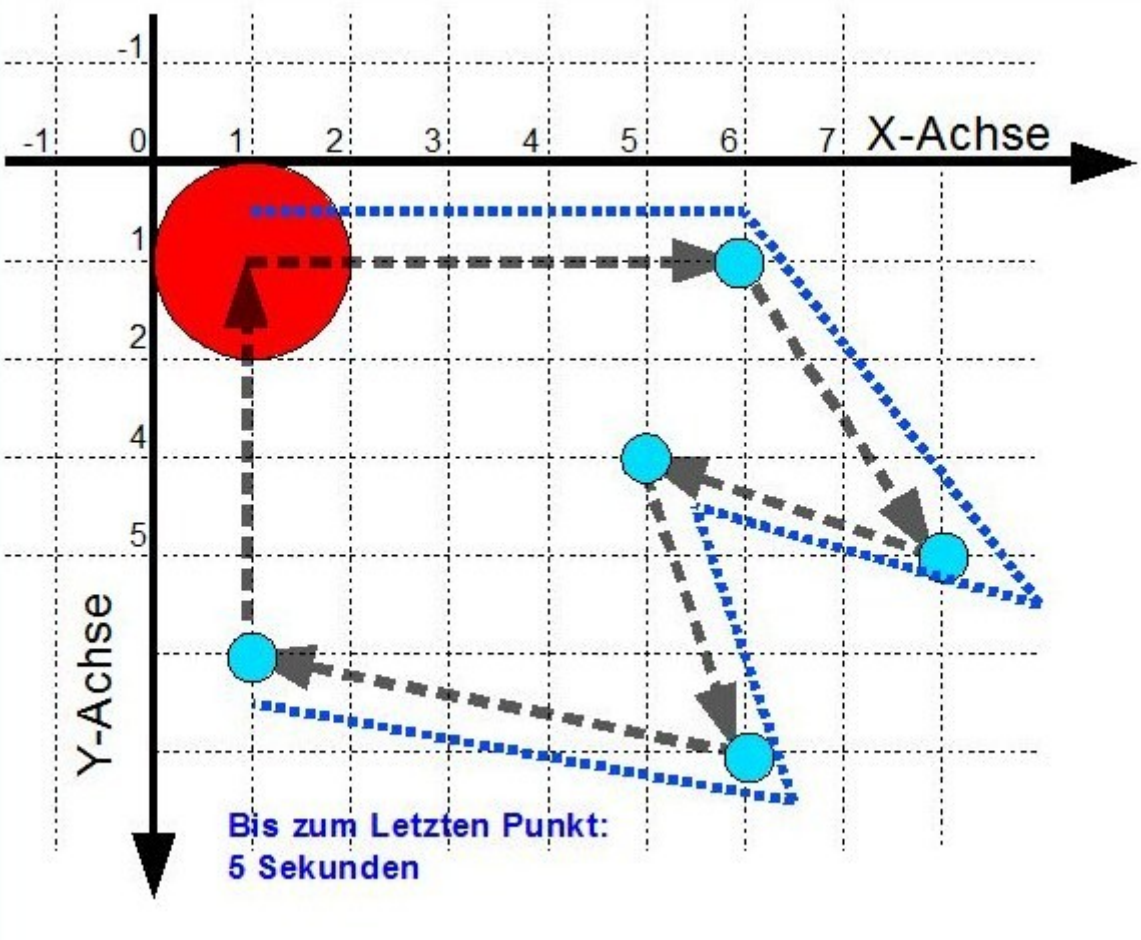
```
streckenAnimation
```

Die detaillierteste Methodenaufruf erwartet folgende Parameter:

| Parameter | Erläuterung |
|---|---|
| `ziel : Raum` | Dieses Objekt soll animiert werden. |
| `laufDauer : int` | Die Zeit - in Millisekunden -, die es dauert, bis einmal sämtliche Streckenteile abgelaufen worden sind |
| `wiederholen : boolean` | Wenn dieser Wert `false` ist, ist die Animation mit dem Erreichen des letzten Streckenpunktes vorbei.<br>Ist er `true`, so wiederholt sich die Animation abhängig von dem folgenden Parameter. |
| `geschlossen : boolean` | Ist dieser Wert true, so wird von dem letzten Streckenpunkt wieder zum ersten animiert und dann immer wieder im Kreis.<br>Ist er false, so wird vom letzten zum vorletzen (usw.) wieder zurück bis zum ersten animiert und dann wird wieder von vorne animiert.<br>**Dieser Parameter ist vollkommen wirkungslos, wenn die Animation nicht wiederholt werden soll.** |
| `streckenteil : Punkt` | Der letzte Parameter kann auf 2 Weisen eingegeben werden:<br>• Eine beliebige Anzahl an `Punkt`-Objekten, hintereinander als einzelne Argumente, die die einzelnen Streckenpunkte sind.<br>• Ein Feld (Array) gefüllt mit `Punkt`-Objekten, die die einzelnen Streckenpunkte sind. |

Dieser Quelltext erstellt eine solche Animation:

```
/* In der spielsteuernden Klasse */

//Erstelle einen Kreis
Kreis k = new Kreis(0, 0, 20);
k.farbeSetzen("Rot");
wurzel.add(k);

//Animiere den Kreis
//In 5000 Millisekunden (= 5 Sekunden) ist der Kreis bei dem letzten Etappen-Punkt
//angekommen
animationsManager.streckenAnimation(k, //Der Kreis
5000, //Nach 5 Sekunden ist die Animation beim letzten Punkt
true, //Ja, die Animation soll unendlich viele Umlaeufe machen
true, //Ja, die Animation soll immer in die selbe Richtung verlaufen
new Punkt(60, 10), new Punkt(80, 50), new Punkt(50, 40), new Punkt(60, 70),
new Punkt(10, 60) //Alle Etappen-Punkte der Animation in der richtigen Reihenfolge. Es
    //können beliebig viele angegeben werden.
);
```




*Die Animation des Quelltextes (eine Einheit auf dem Koordinatensystem sind 10 Pixel).*

Eine Streckenanimation mit weiter aneinander liegenden Punkten ist vorteilhafter. Denn hierdurch sind die Animationen exakter, da weniger Rundungsungenauigkeit entsteht.
Es gibt auch hierzu viele vereinfachte Methoden, nachzulesen in der Dokumentation.

# 4 Geradenanimationen

Der Methodenname lautet:

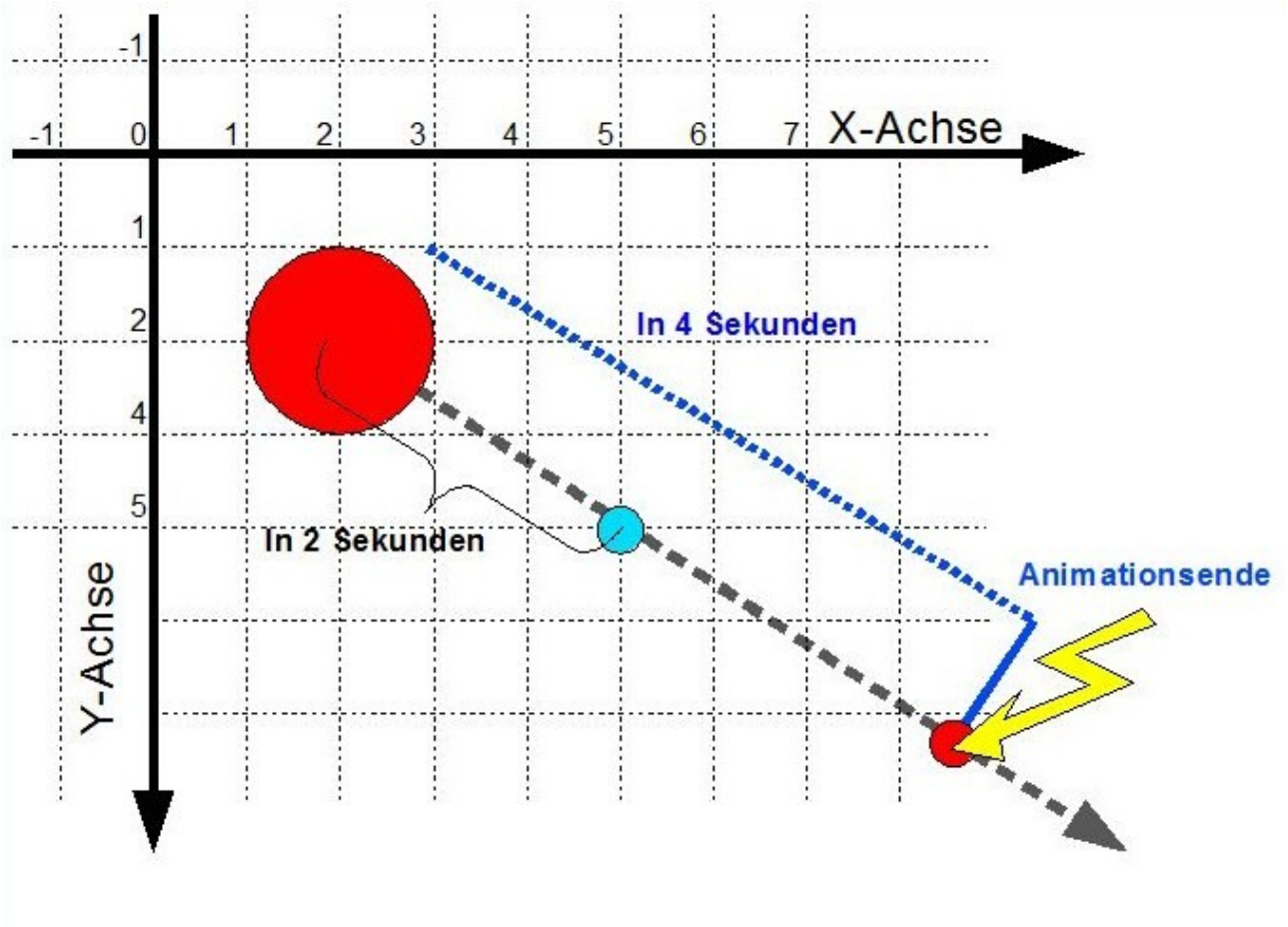
```
geradenAnimation
```

Die detaillierteste Methodenaufruf erwartet 4 Parameter:

| Parameter | Erläuterung |
|---|---|
| `ziel : Raum` | Dieses Objekt soll animiert werden. |
| `orientierung : Punkt` | Die Gerade, auf der das zu animierende Objekt bewegt wird, verläuft durch den Mittelpunkt des zu animierenden Objektes sowie diesen Punkt. |
| `zielGeschwindigkeit:int` | Die Zeit **in Millisekunden**, die es dauert, bis das zu animierende Objekt den Orientierungspunkt erreicht. |
| `dauerInMS : int` | So lange dauert es **in Millisekunden**, bis die Animation beendet wird. |

Dieser Quelltext erstellt eine solche Animation:

```
/* In der spielsteuernden Klasse */

//Erstelle einen Kreis
Kreis k = new Kreis(10, 10, 20);
k.farbeSetzen("Rot");
wurzel.add(k);

//Animiere den Kreis
//(2000 Millisekunden bis zum Orientierungspunkt, 4000 Millisekunden bis zum
//Animationsende
animationsManager.geradenAnimation(k, new Punkt(50, 50), 2000, 4000);
```




*Die Animation des Kreises aus dem Quelltext (eine Einheit auf dem Koordinatensystem sind 10 Pixel)*

Natürlich gibt es auch hierzu vereinfachte Methoden, die sich in der Dokumentation nachlesen lassen.

## H Reagieren auf Aktionen des Spielers

Bei jedem Spiel besteht über die Methode `tasteReagieren(int code)` bereits die Möglichkeit, auf das einfache Runterdrücken einer Taste zu reagieren.
Doch es gibt weit mehr Möglichkeiten, als Spieler interaktiv das Spiel zu lenken.
Die wichtigsten Arten sind in der Engine Alpha vertreten:

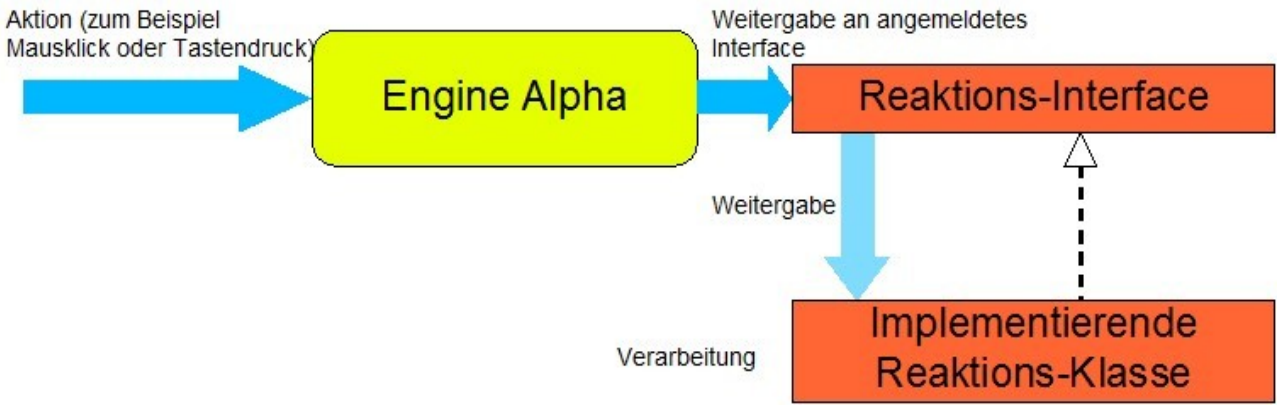
- Einfacher Tastendruck (bereits in der Klasse `Game` vorhanden)
- Gedrückthalten einer Taste
- Loslassen einer Taste
- Mausklicks

Das System, nach dem diese Aktionen verarbeitet werden können, funktioniert mit Interfaces.

Für jede Art der Interaktionen gibt es ein spezielles Interface. Dieses Interface besitzt nur eine abstrakte Methode, mit der es auf die entsprechende Interaktion (Ein Tastendruck, ein Mausklick...) reagieren kann.
Implementiert eine Klasse dieses Interface und überschreibt dessen abstrakte Methode, so kann sie sich über die Klasse `Game` anmelden.

Sobald sie angemeldet ist, wird sie die Engine Alpha automatisch bei jeder Interaktion, die sie betrifft, „benachrichtigen“, indem sie ihre Methode aufruft.

| Dieses System heißt *Listener*-System und ist weit über die Engine Alpha hinaus gebräuchlich. |
|---|




*Das Reaktions-Interface-System (Listener-System) der Engine Alpha als Grafik*

Es müssen bei diesem Listener-System immer bestimmte, gleiche Schritte unternommen werden. Diese Zeige ich Dir nun auf und unterstütze diese mit Beispielen. Diese Beispiele funktionieren jedoch nicht, denn sie sollen dir nur die Arbeitsschritte verdeutlichen!

1. Die Klasse, die die Interaktion bearbeiten soll, muss das Interface implementieren:

```
public class MeineReaktionsklasse
implements ReaktionsInterface { //<-- Implementieren!!
    //Code
}
```

2. Diese Klasse muss dessen Reaktionsmethode überschreiben:

```
public class MeineReaktionsklasse
implements ReaktionsInterface {

    // | Ueberschreiben der Interface-Reaktionsmethode!
    // V
    @Override //<-Zeigt an, dass eine Methode ueberschrieben
    //wird. Diese ueberschriebene Methode ist die im Interface
    public void aktionVerarbeiten(int parameter) {
        //Code
    }
}
```

3. Diese Klasse muss über eine Anmelde-Methode der Klasse `Game` bei der Engine Alpha angemeldet werden. **Vorher funktioniert das System nicht!**

```
public class MeineReaktionsklasse
implements ReaktionsInterface {

    //Das Anmelden passiert meistens im Konstruktor!
    public MeineReaktionsklasse() {
        reaktionsInterfaceAnmelden(this); //<-HIER wird
                                          //angemeldet!
    }

    @Override
    public void aktionVerarbeiten(int parameter) {
        //Code
    }
}
```

Sind alle diese drei Schritte getan, wird die überschriebene Reaktionsmethode fortan immer aufgerufen, wenn eine entsprechende Interaktion hierfür stattgefunden hat.
Keine Angst, die Verwendung dieser Interfaces wird im Folgenden auch an einem einfachen Beispiel erklärt.

Nachfolgend werden die einzelnen Interfaces vorgestellt, sowie ihre eigenen abstrakten Reaktionsmethoden und ihre Anmeldemethoden.

# 1 Tastatur-Interfaces

Für die Tastatur unterscheidet man drei Interaktionsarten, zu denen folgende Interfaces gehören:

1. Das einfache Herunterdrücken einer Taste : `TastenReagierbar`
2. Das Loslassen einer Taste : `TastenLosgelassenReagierbar`
3. Das Gedrückthalten einer Taste : `TastenGedruecktReagierbar`

Diese Interfaces werden auf den folgenden Seiten erläutert.

## *i Das Interface `TastenReagierbar`*


*Klassenkarte des Interfaces `TastenReagierbar`*

| | |
|---|---|
| Aufgerufen | Sobald eine Taste einmal heruntergedrückt wird. Diese Methode für die selbe Taste erst dann wieder aufgerufen, wenn diese Taste auch wieder losgelassen wurde. |
| Reaktionsmethode | `reagieren(int code)` |
| Erläuterung | Der Parameter `code` ist eine Zahl, die anzeigt, welche Taste gedrückt wurde. Welche Taste zu welcher Zahl gehört, lässt sich aus einer Tabelle im Anhang des Buches ablesen. Für alle Tastaturmethoden ist die Code-Tabelle dieselbe wie die bereits in *Kapitel IV.A – Die spielsteuernde Klasse* erwähnte Tabelle. |
| Anmeldemethode | `tastenReagierbarAnmelden(TastenReagierbar t)`<br>`//Anmeldemethode in der Klasse Game!!` |

**Achtung:**
Die Klasse `Game` implementiert bereits das Interface `TastenReagierbar`, das heißt, dass die spielsteuernde Klasse dieses Interface nicht erneut implementieren kann!!
Dieses Interface ist also nur erneut in anderen Klassen zu verwenden.

Im Grunde ist das Implementieren hiervon gar nicht nötig, denn das Reagieren auf einfachen Tastendruck ist ja bereits in der Klasse `Game` realisiert.

### ii Das Interface `TastenLosgelassenReagierbar`


*Klassenkarte des Interfaces* `TastenLosgelassenReagierbar`

| | |
|---|---|
| Aufgerufen | Immer dann, wenn eine Taste losgelassen wird. Sozusagen als Gegenstück zum `TastenReagierbar`-Interface. |
| Reaktionsmethode | `tasteLosgelassen(int code)` |
| Erläuterung | Der Parameter `code` ist eine Zahl, die anzeigt, welche Taste gedrückt wurde. Welche Taste zu welcher Zahl gehört, lässt sich aus einer Tabelle im Anhang des Buches ablesen.Für alle Tastaturmethoden ist die Code-Tabelle dieselbe wie die bereits in *Kapitel IV.A – Die spielsteuernde Klasse* erwähnte Tabelle. |
| Anmeldemethode | `tastenLosgelassenReagierbarAnmelden(TastenLosgelassenReagierbar t)`<br>`//Anmeldemethode in der Klasse Game!!` |

### iii Das Interface `TastenGedrueckReagierbar`


*Klassenkarte des Interfaces* `TastenGedruecktReagierbar`

| | |
|---|---|
| Aufgerufen | Solange eine bestimmte Taste runtergedrückt ist, wird diese Methode **in Abständen von 50 Millisekunden** immer wieder mit dem entsprechenden Tasten-Code aufgerufen.<br>Wenn mehrere Tasten runtergedrückt sind, wird diese Methode auch für alle runtergedrückten Tasten aufgerufen. |
| Reaktionsmethode | `tasteGedrueckt(int code)` |
| Erläuterung | Wird – wie bereits erwähnt – in Abständen von 50 Millisekunden für jede gedrückte Taste aufgerufen.<br>Der Parameter `code` ist eine Zahl, die anzeigt, welche Taste gedrückt wurde. Welche Taste zu welcher Zahl gehört, lässt sich aus einer Tabelle im Anhang des Buches ablesen.Für alle Tastaturmethoden ist die Code-Tabelle dieselbe wie die bereits in *Kapitel IV.A – Die spielsteuernde Klasse* erwähnte Tabelle. |
| Anmeldemethode | `tastenGedruecktReagierbarAnmelden(TastenGedrueckReagierbar t)`<br>`//Anmeldemethode in der Klasse Game!!` |

### *iv Ein Beispiel für Alle Tastatur-Interfaces:*

Das nachfolgende Beispiel zeigt anschaulich, wie die Listener funktionieren.

Eine spielsteuernde Klasse implementiert die Interfaces

- `TastenGedruecktReagierbar` und
- `TastenLosgelassenReagierbar`.

Das Interface.

- `TastenReagierbar`

kann nicht und muss nicht implementiert werden. Denn hierfür gibt es bereits die abstrakte Methode `tasteReagieren(...)` in der Klasse `Game`. Dies funktioniert intern nämlich bereits über das Interface `TastenReagierbar`.

```
import ea.*;

/**
 * Diese Klasse reagiert auf alle Arten von Tastendruck.
 */
public class Spiel
extends Game
implements TastenGedruecktReagierbar, TastenLosgelassenReagierbar{

    public Spiel() {
        super(130, 100);

        //Anmelden als 'Listener' fuer Gedrueckthalten und Loslasse
        super.tastenGedruecktReagierbarAnmelden(this);
        super.tastenLosgelassenReagierbarAnmelden(this);
    }

    public void tasteReagieren(int code) {
        System.out.println("Die Taste mit dem Code " + code + " wurde heruntergedrueckt.");
    }

    public void tasteGedrueckt(int code) {
        System.out.println("Die Taste mit dem Code " + code + " wird zur Zeit heruntergedrueckt");
    }

    public void tasteLosgelassen(int code) {
        System.out.println("Die Taste mit dem Code " + code + " wurde losgelassen.");
    }
}
```

# 2 Die Maus

Wie Du zweifelsfrei schon festgestellt hast, ist der Maus-Cursor nicht sichtbar, solange er auf dem Spielfenster liegt.
Dies liegt daran, dass noch keine Maus angemeldet ist.

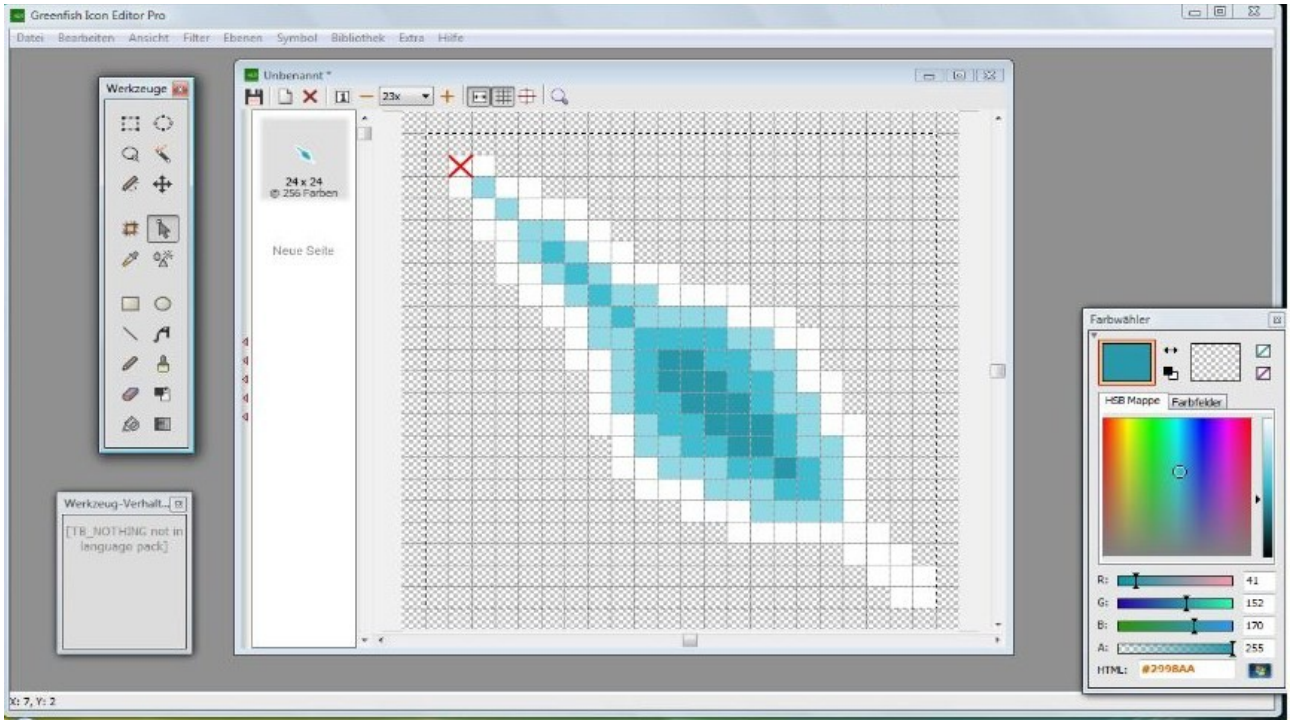
## *i Das ist für eine Maus alles nötig*

Eine Maus ist in der Engine Alpha vertreten durch die Klasse `Maus`.

Um eine Maus zu erstellen, müssen 2 Dinge angegeben werden:

- Ein Bild, dass der neue Maus-Cursor sein soll
- Ein *Hotspot*

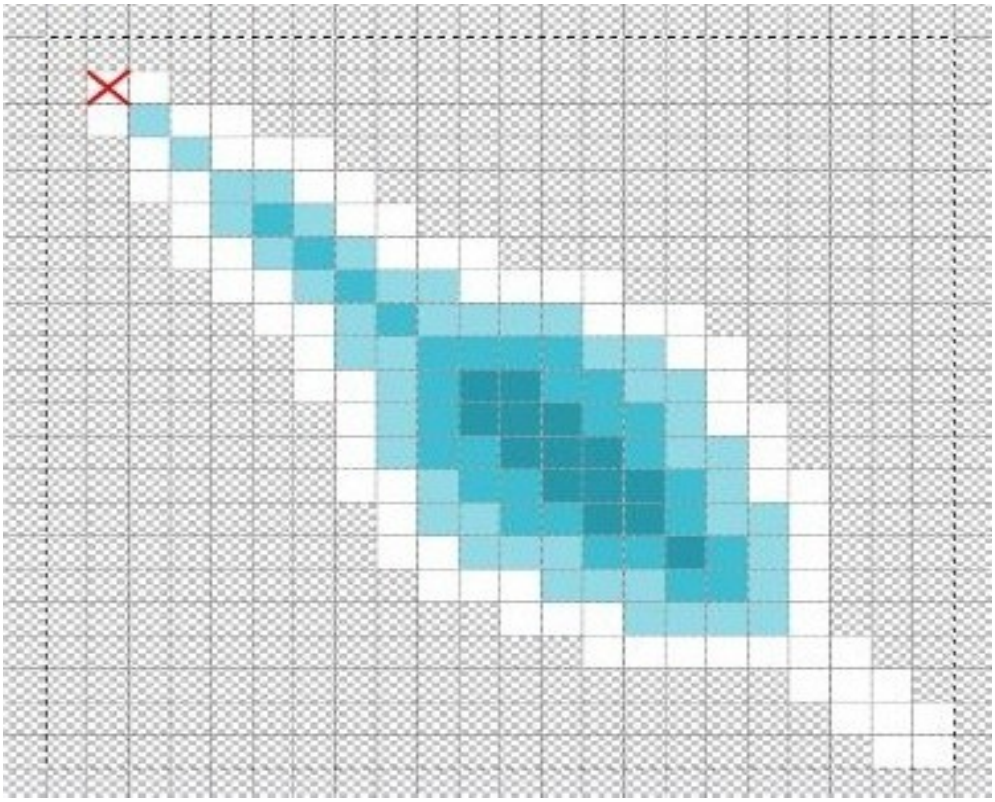
Das Mausbild sollte man selbst erstellen. Hierfür gibt es Programme, mit denen man solche Bilder erstellen kann.
Hierfür ist zum Beispiel das kostenlose Programm *Greenfish Icon Editor* sehr zu empfehlen, das sich problemlos aus dem Internet laden lässt.




*Ein Mausbild wird mit dem Greenfish Icon Editor erstellt*

Da eine Maus so gut wie nie rechteckig ist, sollte das Bild als .gif-Image exportiert werden, da hierbei auch besondere Formen zeichenbar sind, nicht nur Rechtecke.

Als zweites muss der *Hotspot* gesetzt werden.
Ein Mausbild kann eine größere Fläche einnehmen, aber ein Mausklick „zielt“ auf genau einen Punkt. Deshalb muss ein Punkt auf dem Bild ausgewählt werden, der die „Spitze“ der Maus und ausschlaggebend für die Klicks ist.



*Das Bild der Maus im Editor. Das rote Kreuz ist der Hotspot. Er hat die Koordinaten (2|2)*

Dieser Hotspot wird ebenfalls beim Erstellen einer Maus eingegeben.

Weiterhin gibt es zwei verschiedene Arten von Mäusen: *Absolute* und *relative* Mäuse.

- Eine absolute Maus
  bleibt immer im exakten Zentrum des Fensters. Und bei Bewegung wird dies direkt auf eine Kameraverschiebung übertragen. Diese Art der Maus wird vor allem für Shooter-Games verwendet.

- Eine relative Maus
  ist die Art von Maus, die bei einem Computer verwendet wird. Sie bewegt sich so, wie man es von einer Maus bei jedem Computer kennt.

Die letzte wichtige Eigenschaft, die eine Maus ausmacht, ist, ob sie das Bild verschieben kann.
Ist dies der Fall, so wird die *relative* Maus, wenn sie den Bildrand erreicht, das Bild verschieben. Eine *absolute* Maus wird dann bei jeder Bewegung die Kamera verschieben. Dies ist für eine absolute Maus nötig, denn sonst wäre jede Mausverschiebung absolut wirkungslos.

### ii Der Konstruktor der Klasse Maus

Alle eben genannten Eigenschaften werden im folgenden Konstruktor übergeben:

```
public Maus(Bild mausbild,
            Punkt hotspot,
            boolean absolut,
            boolean bewegend)
```

Dafür stehen die Parameter:

| Parameter | Erläuterung |
|---|---|
| `mausbild` | Ein `Bild`-Objekt. Diese Klasse wurde bereits bei den Grundlagen behandelt. Die Position des Bildes ist irrelevant, deshalb kann man es einfach an den Koordinaten (0\|0) oder beliebigen anderen erstellen. |
| `hotspot` | Der Hotspot. Dieses `Punkt`-Objekt gibt den Hotspot als relativen Punkt auf dem Bild an. |
| `absolut` | Ob diese Maus absolut ist, oder nicht. Ist dieser Wert `true`, so ist sie absolut, ist er `false`, so ist sie relativ. |
| `bewegend` | Ob diese Maus bewegend ist oder nicht. Ist dieser Wert `true`, so bewegt diese Maus das Bild mit, ist er `false`, bleibt sie immer in den Grenzen des Bildschirms. Um die Bewegung der Maus trotz Bewegung einzugrenzen, kann man der *Kamera* Grenzen setzen. Mehr hierzu im Kapitel *„Die Kamera – Grenzen für die Kamera“*. |


*Klassenkarte der Klasse `Maus` – mit Konstruktor und Anmelde-Methoden (werden im nächsten Teil benötigt)*

### *iii Einbringen einer Maus: Ein Beispiel*

Die folgende spielsteuernde Klasse zeigt, wie man die Maus praktisch in das Spiel einbringt.

```
import ea.*;

/**
 * Demonstration der Einbringung der Maus.
 */
public class Spiel
extends Game {
    private Maus maus;

    public Spiel() {
        super(300, 400);

        //Das Bild (Koordinaten des Bildes sind irrelevant s. o.)
        Bild mausbild = new Bild(0, 0, "fadenkreuz.gif");
        //Der Hotspot
        Punkt hotspot = new Punkt(12, 12);

        //Die Maus einbringen (NICHT absolut und NICHT bewegend)
        maus = new Maus(mausbild, hotspot, false, false);
        mausAnmelden(maus);
    }

    public void tasteReagieren(int code) {
        //Nicht in Gebrauch
    }
}
```

### *iv Reagieren auf Mausklicks*

Nun ist die Maus im Spiel, aber das Programm kann noch nicht auf Klicks reagieren.

So wie es Interfaces für die Tastatur gibt, gibt es auch Interfaces für die Maus. Auf folgende Arten von Klicks kann reagiert werden. So heißen die entsprechenden Interfaces:

- Ein genereller Linksklick : `KlickReagierbar`
- Ein genereller Rechtsklick : `RechtsKlickReagierbar`
- Ein Klick auf ein `Raum`-Objekt : `MausReagierbar`

Diese Interfaces werden wieder erläutert.

### Das Interface `KlickReagierbar`


*Klassenkarte des Interfaces `KlickReagierbar`*

| | |
|---|---|
| Anmeldemethode | `maus.klickReagierbarAnmelden(KlickReagierbar k)`<br>`//Anmeldung am Maus-Objekt!!` |
| Erläuterung der Anmeldemethode | `k : KlickReagierbar` — Dieses Interface wird ab sofort immer benachrichtigt, wenn ein Linksklick statt fand. |
| Reaktionsmethode | `klickReagieren(int x, int y)` |
| Erläuterung der Reaktionsmethode | Diese Methode wird bei jedem Linkslick aufgerufen und gibt die *exakten Koordinaten des Klicks auf der Zeichenebene* an, nicht auf dem Fenster.<br>Sprich, bei verschobener Kamera ist die Maus auch „verschoben“.<br>Zur Kamera siehe das Kapitel „Die Kamera“. |

*Die Maus-Interfaces müssen bei der Maus, und nicht bei der Klasse `Game` angemeldet werden.*

### Anwendungsbeispiel für `KlickReagierbar`

Hier ein einfaches Beispiel. Es soll folgendermaßen funktionieren: Bei jedem Linksklick wird genau auf dem Klick-Punkt ein Kreis erstellt.

```
import ea.*;

/**
 * Dieses Spiel kann auf Klick an Ort und Stelle Kreise erscheinen lassen.
 */
public class Spiel
extends Game
implements KlickReagierbar
{
    private Maus maus;

    /**
     * Der Konstruktor.<br />
     * Erstellt ein Spielfenster mit den Massen 500 x 500 und anschliessend die Maus.
     */
    public Spiel() {
        super(500, 500);
        //Maus erstellen und anmelden
        maus = new Maus(new Bild(0, 0, "fadenkreuz.gif"),
                        new Punkt(12, 12),
                        false,
                        false);
        mausAnmelden(maus);
        maus.klickReagierbarAnmelden(this);
    }

    /**
     * Die Klick-Reaktionsmethode. Wird bei jedem Klick aufgerufen.
     * @param   x   Die X-Koordinate des Klicks
     * @param   y   Die Y-Koordinate des Klicks
     */
    public void klickReagieren(int x, int y) {
        //Erstelle einen roten Kreis
        Kreis k = new Kreis(0, 0, 30);
        k.farbeSetzen("Rot");
        //Positioniere den Mittelpunkt des Kreises auf die Klick-Koordinaten
        k.mittelpunktSetzen(x, y);
        //Mache den Kreis sichtbar, indem er an der Wurzel angemeldet wird
        wurzel.add(k);
    }

    /**
     * Taste-Reagieren-Methode; wird nicht gebraucht
     */
    public void tasteReagieren(int code) {
        //
    }
}
```

### Das Interface `RechtsKlickReagierbar`

Dieses Interface funktioniert exakt parallel zu dem Interface `KlickReagierbar`. Der einzige Unterschied ist, dass dieses auf Rechts- das andere auf Linksklicks reagiert – und der Name.
Deshalb gibt es hierzu kein Beispiel.


*Klassenkarte des Interfaces* `RechtsKlickReagierbar`

| | |
|---|---|
| Aufgerufen | Bei jedem Rechtsklick wird dieses Interface – sofern angemeldet – aufgerufen. |
| Reaktionsmethode | `rechtsKlickReagieren( int x, int y)` |
| Erläuterung | Wird bei jedem Rechtsklick aufgerufen und gibt die *exakten Koordinaten des Klicks auf der Zeichenebene* an, nicht auf dem Fenster.<br>Sprich, bei verschobener Kamera ist die Maus auch „verschoben“.<br>Zur Kamera siehe das Kapitel „Die Kamera“. |
| Anmeldemethode | `maus.rechtsKlickReagierbarAnmelden(RechtsKlickReagierbar k)`<br>`//Anmeldung am Maus-Objekt!!` |

*Die Maus-Interfaces müssen bei der Maus, und nicht bei der Klasse* `Game` *angemeldet werden.*

### Das Interface `MausReagierbar`


*Klassenkarte des Interfaces* `MausReagierbar`

<table>
<tr><td>Anmeldemethode</td><td>

```
maus.mausReagierbarAnmelden(   MausReagierbar m,
                               Raum r,
                               int code)
//Anmeldung am Maus-Objekt!!
```

</td></tr>
<tr><td>Erläuterung der Anmeldemethode</td><td>
<table>
<tr><td><code>m : MausReagierbar</code></td><td>Das Interface, das ab sofort informiert wird.</td></tr>
<tr><td><code>r : Raum</code></td><td>Wird dieses grafische Objekt mit einem Linksklick angeklickt, so wird das Interface hierüber informiert.</td></tr>
<tr><td><code>code : int</code></td><td>Dieser Code wird dem Interface bei dem Aufruf der Reaktionsmethode als Parameter wieder mitgegeben.</td></tr>
</table>
</td></tr>
<tr><td>Reaktionsmethode</td><td><code>mausReagieren(int code)</code></td></tr>
<tr><td>Erläuterung der Reaktionsmethode</td><td>
Diese Methode wird immer dann aufgerufen, wenn das <code>Raum</code>-Objekt, das bei der Methode zum Anmelden übergeben wurde, mit Linksklick angeklickt wurde.
<table>
<tr><td><code>code : int</code></td><td>Der Code, der beim Anmelden übergeben wurde.</td></tr>
</table>
<br>
Der Vorteil des <code>code</code>-Parameters ist folgendes:<br>
So kann man an einem Reaktions-Interface die Klicks auf beliebig viele <code>Raum</code>-Objekte kontrollieren.<br><br>
Denn für jedes grafische Objekt kann man bei der Anmeldung einen anderen Code eingeben. Meldet man sein Interface zweimal an – einmal mit einem Kreis und dem Code 1 und einmal mit einem Rechteck und dem Code 2 – so wird beim Klick auf den Kreis wie auf das Rechteck dieselbe Methode aufgerufen, aber mit 2 verschiedenen Werten für den Parameter <code>code</code>. Ist
</td></tr>
</table>

| | |
|---|---|
| | dieser Wert 1, so wurde der Kreis angeklickt; ist der Wert 2, so wurde das Rechteck angeklickt.<br>Dieses Prinzip lässt sich mit den Tastatur-Codes bei den Tastatur-Interfaces vergleichen. Hier jedoch bestimmst Du selbst, wofür welcher Code-Wert steht. |

*Die Maus-Interfaces müssen bei der Maus, und nicht bei der Klasse `Game` angemeldet werden.*

### Anwendungsbeispiel für `MausReagierbar`

Nun soll das Anfangsbeispiel, indem eine Maus erstellt und angemeldet wurde, erweitert werden, und zwar so, dass die Maus auf 2 Kreise klicken kann. Der Klick auf einen Kreis soll den Hintergrund verändern, dafür wird zunächst ein einfarbiger Rechteckiger Hintergrund eingeführt. Zusätzlich wird das `MausReagierbar`-Interface implementiert:

```
public class Spiel
extends Game
implements MausReagierbar {

    private Rechteck hintergrund;

    [...] //Der Rest bleibt so wie am Anfang des Kapitels Die Maus
}
```

*Implementieren des Interfaces und Deklaration des Hintergrundes*

Nun wird im Konstruktor dieser Hintergrund instanziiert.
Zusätzlich werden im Konstruktor 2 unterschiedlich gefärbte Kreise erstellt und bei der Maus angemeldet für diese Klasse (die ja `MausReagierbar` implementiert) zur Beobachtung von Mausklicks, jeweils mit einem unterschiedlichen Code. Dadurch ist ein Klick, auch wenn immer die selbe Methode aufgerufen wird, eindeutig zuordbar, so wie bei der Tastatur und dem einheitlichen Tastencode-System.

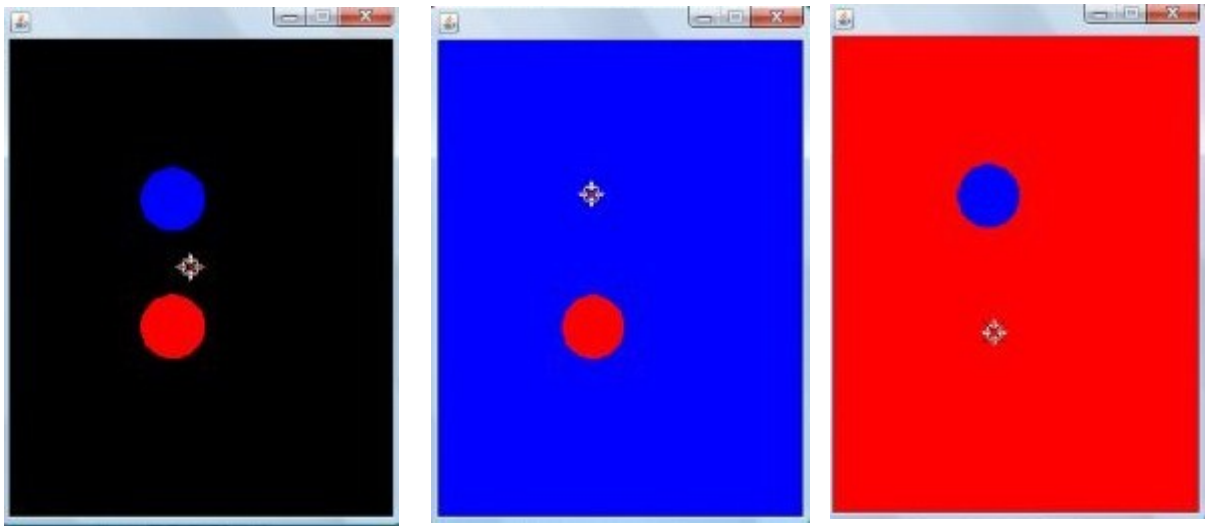
```
/*Erweiterung des Konstruktors*/

hintergrund = new Rechteck(0, 0, 300, 400);
hintergrund.farbeSetzen("Schwarz");
hintergrundSetzen(hintergrund);

Kreis blau = new Kreis(100, 100, 50);
blau.farbeSetzen("Blau");
wurzel.add(blau);
//Anmeldung an der Maus, nicht in der Klasse Game!!!!!
maus.anmelden(this, blau, 1); //Code 1

Kreis rot = new Kreis(100, 200, 50);
rot.farbeSetzen("Rot");
wurzel.add(rot);
//Anmeldung an der Maus, nicht in der Klasse Game!!!!!
maus.anmelden(this, rot, 2); //Code 2
```

*Der Konstruktor wird um diese Zeilen erweitert*

Versucht man nun, zu übersetzen, wird man spätestens dann feststellen, dass zunächst noch die Reaktionsmethode aus dem implementierten `MausReagierbar`-Interface überschrieben werden muss.
In dieser Methode soll der Hintergrund, je nachdem welcher Kreis angeklickt wurde, dessen Farbe annehmen. Dies wird folgendermaßen realisiert:

```
/**
 * Diese Methode wird immer dann aufgerufen, wenn
 * einer der beiden Kreise angeklickt wurde und je nach
 * dem, welcher, wird entweder code 1 oder 2 uebergeben.
 * @param   code    Dieser Code macht es eindeutig
 * zuordbar, welcher Kreis angeklickt wurde.
 */
public void mausReagieren(int code) {
    switch(code) {
        case 1: // Code 1: blau
            hintergrund.farbeSetzen("Blau");
            break;
        case 2: //Code 2: rot
            hintergrund.farbeSetzen("Rot");
            break;
    }
}
```

Nun funktioniert das Beispiel:



*Das Programm in seinen 3 Phasen: Ausgangslage, Blau und Rot*

# I Farben

Farben lassen sich in der Engine Alpha auf 2 Arten beschreiben.

1. Als Texte
2. Über R/G/B(/A)-Werte

## 1 Farben als Texte

Die einfachste Farbbeschreibung ist über ein Objekt der Klasse `String`. Diese Art der Farbgebung wurde in den Beispielen verwendet.

Folgende „Namen“ repräsentieren eine Farbe in der Engine (Groß/Kleinschreibung ist egal):

```
"Gelb", "Orange", "Weiss", "Grau", "Gruen", "Blau", "Rot", "Pink",
"Magenta"/"Lila", "Cyan"/"Tuerkis", "Dunkelgrau", "Hellgrau",
"Schwarz"
```

Jede andere Namenseingabe sorgt automatisch für die Farbe Schwarz.

## 2 Farben über R/G/B-Werte

Hierzu gibt es eine Alternative, mit der sich jede Farbe für jeden Zweck exakt maßschneidern lässt. Ein Farbton lässt sich über die **Anteile der drei Komplementärfarben Rot, Grün und Blau** beschreiben. Dieser kann zwischen 0 und 100% liegen.

Das kann man als digitales Farbenmischen betrachten, indem man verschiedene Anteile der Grundfarben zusammenmischt um den gewünschten Farbton zu erhalten.
Angegeben werden diese Anteile jedoch nicht in Prozent sondern **in Zahlen zwischen 0 (≙ 0% Anteil) und 255 (≙ 100% Anteil)**.
Nun lässt sich eine Farbe also durch 3 Zahlen beschreiben, den Rot/Grün/Blau-Anteilen. Dies wird mit R/G/B abgekürzt.

`Farbe` ist eine Klasse in der Engine Alpha.
Sie beschreibt eine Farbe und lässt sich sehr leicht instanziieren. Dieser Konstruktor sorgt für eine Farbe:

```
public Farbe(int r, int g, int b)
```

Die drei Zahlenwerte müssen Zahlen zwischen 0 und 255 sein und entsprechen den Anteilen der entsprechenden Grundfarbe, die zum „Mischen“ verwendet werden soll.

So erhält man zum Beispiel:

| | |
|---|---|
| Schwarz | `new Farbe(0, 0, 0);` |
| Weiß | `new Farbe(255, 255, 255);` |
| Orange | `new Farbe(255, 200, 0);` |
| Cyan/Türkis | `new Farbe(0, 255, 255);` |

Natürlich sind alle anderen Zahlenkombinationen mit Werten zwischen 0 und 255 möglich.

Farben setzen lassen sich bei allen färbbaren Klassen (`Text` und alle `Geometrie`-Klassen) mit diesen beiden Methoden.

```
public void farbeSetzen(String farbName)

public void farbeSetzen(Farbe farbe)
```

Die zweite Methode ist neu. Hierbei muss ein `Farbe`-Objekt als Parameter übergeben werden und ab diesem Moment füllt diese Farbe das Objekt aus.

# 3 Alpha-Werte

Ein weiterer Vorteil der Klasse `Farbe` ist die Möglichkeit, Farben zu einem gewissen Anteil durchsichtig machen zu können. Die „Nichtdurchsichtigkeit“ einer Farbe wird als *Alpha-Helligkeit* bezeichnet. Auch diese lässt sich mit einem Wert zwischen 0 und 255 beschreiben.
Ist sie 255, so ist die Farbe gänzlich sichtbar; ist sie 0, ist die Farbe absolut durchsichtig und damit unsichtbar. Alle Werte dazwischen sorgen für eine unterschiedlich stark durchsichtige, geister- oder glasartige Farbe.

Hierfür gibt es dann den vollen Konstruktor:

```
public Farbe(int r, int g, int b, int alpha)
```

Ein sehr einfaches Beispiel:
Erstellt werden soll ein halbdurchsichtiges Rechteck

```
/*In der spielsteuernden Klasse (wg. der "wurzel")*/

//Erstelle ein Quadrat mit Seitenlaenge 200
//in der linken oberen Bildecke
Rechteck recht = new Rechteck(0, 0 200, 200);

//Erstelle der Fuellfarbe
Farbe fuellFarbe = new Farbe(178, 255, 255, 200);

//Dem Quadratk die Fuellfarbe zuweisen
recht.farbeSetzen(fuellFarbe);

//Das Rechteck an der Wurzel anmelden. Dann wird es sichtbar
wurzel.add(recht);
```

# *J Die Engine-Alpha-„Physik“*

Eines meiner absoluten Lieblingsgenres ist *Jump n' Run:* Laufen, Springen und Fallen.
Deshalb gibt es in der Engine Alpha einen „Physik“-Modus, bei dem automatisch grafische Objekte Springen und Fallen können.

**Dieses Kapitel hat es ziemlich in sich!** Du solltest Dir Zeit nehmen und es möglichst exakt verstehen – wenn du die Physik benutzen willst.

## 1 Die Idee

Das Grundprinzip dahinter ist sehr einfach. Objekte in der Engine-Alpha-Physik lassen sich in 2 Gruppen aufteilen:

- **Aktiv-Objekte**
  Die „Spielfiguren“. Aktiv-Objekte können Fallen und Springen.

- **Passiv-Objekte**
  Passiv-Objekte können Aktiv-Objekte behindern. Das sind also Mauern, Böden und Decken.

Zusätzlich hierzu gibt es noch:

- **Neutrale Objekte**
  Alle Objekte, die nicht an der Physik der Engine beteiligt sind, sind neutrale Objekte. *Das ist bei allen Raum-Objekten die Standard-Einstellung*.

| **Mehr ist nicht dahinter!** |
|---|

Schließlich soll die „Physik“ möglichst einfach sein.

## 2 Die grundlegende Verwendung

Ebenso wie das Prinzip ist auch die Verwendung möglichst einfach. Zunächst erstellen wir eine sehr einfache spielsteuernde Klasse noch vollkommen ohne Physik. Es gibt in diesem Spiel zwei Rechtecke: Eine Ebene und ein Quadrat als „Spielfigur“. Die Spielklasse sieht zu Beginn so aus:

```
import ea.*;

public class Spiel
extends Game {
    /**
     * Die "Ebene"
     */
    private Rechteck ebene;

    /**
     * Das "fallende Objekt"
     */
    private Rechteck aktiv;

    /**
     * Konstruktor - erstellt das "Spiel"
     */
    public Spiel() {
        super(400, 400); //<-Aufruf des Konstruktors aus der
                         //Klasse Game (mit Fensterhoehe/-breite)
        //Instanziieren und Einrichten von Ebene und Fallobjekt
        ebene = new Rechteck(0, 300, 200, 10);
        ebene.farbeSetzen("Weiss");
        aktiv = new Rechteck(30, 30, 20, 20);
        aktiv.farbeSetzen("Rot");
        //Anmelden der grafischen Objekte an der Wurzel, um sie
        //sichtbar zu machen!
        wurzel.add(aktiv);
        wurzel.add(ebene);
    }

    /**
     * Taste-Reagiermethode. Wird bei jedem Tastendruck aufgerufen
     * @param   code    Der Code der gedruecken Taste
     */
    @Override
    public void tasteReagieren(int code) {
        switch(code) {
            //N. n. in Gebrauch
        }
    }
}
```

Startet man jetzt jedoch das Spiel, so passiert noch gar nichts. Denn bis jetzt ist die Physik ja noch gar nicht benutzt. Aber das lässt sich ganz leicht korrigieren, mit sehr einfachen Methoden in der Klasse `Raum`.

```
public void aktivMachen()

public void passivMachen()
```

Du kannst auch ein `Raum`-Objekt wieder neutral machen. Hierfür gibt es in der Klasse `Raum` folgende Methode:

```
public void neutralMachen()
```

Konkret im Beispiel musst Du also nur im Konstruktor am Ende diese beiden Methodenaufrufe ergänzen:

```
//Einbringen der Physik
aktiv.aktivMachen();
ebene.passivMachen();
```

Schau, was passiert. Du kannst auch einmal die Methode zum Aktiv-Machen oder Passiv-Machen weglassen und schauen, was passiert.
Wenn du meinst, dass du die Physik soweit beherrscht, kannst du dich nun an die Funktionen wagen!

# 3 Die Funktionen der „Physik“

Nachdem du nun weißt, wie man ein beliebiges `Raum`-Objekt einer Physik-Objektart (aktiv oder passiv) zuweist, kannst du dir nun einmal anschauen, was du mit der Physik machen kannst.

## *i Bewegungen*

### Die Methode `bewegen(...)`

Um ein `Raum`-Objekt zu „bewegen“, kennst du bereits die Methode `verschieben(...)`. Hierbei wird das Objekt immer voll verschoben.

Aber zum Beispiel beim Laufen einer Spielfigur, soll diese nicht weiter verschoben werden, wenn sie vor einer Mauer steht. Deshalb gibt es hierfür eine alternative Methode in der Klasse `Raum`:

```
public boolean bewegen(Vektor verschiebung)

public boolean bewegen(int x, int y)
```

Diese Methode macht fast das selbe wie die Methode `verschieben(...)`. Aber hierbei werden die physikalischen Eigenschaften des Objektes mit einbezogen. Das heißt konkret:

| *Die Methode `bewegen(...)` bewirkt:* | | |
|---|---|---|
| **Bei einem Aktiv-Objekt** | **Bei einem Passiv-Objekt** | **Bei einem neutralen Objekt** |
| Das Objekt wird im Idealfall soweit wie gewünscht verschoben. Wird das Objekt jedoch von einem Passiv-Objekt geblockt, so kann es sich nicht an ihm vorbei bewegen und die Bewegung wird **nicht voll ausgeführt**. | Das Objekt wird **immer voll verschoben**. Zusätzlich werden Aktiv-Objekte „mitgeschoben“ und Aktiv-Objekte, die auf diesem Passiv-Objekt stehen, mitgenommen. | Das selbe wie die Methode `verschieben(...)`, da die Physik für neutrale Objekte nicht gilt. |

*Nutzt man die Physik der Engine Alpha, sollten Raum-Objekte, die nicht neutral sind, immer mit `bewegen(...)` verschoben werden, damit die Physik die Bewegungen miteinberechnet!*

Diese Bewegung lässt sich sehr leicht im Beispiel realisieren. Wir nutzen das Reagieren auf einfachen Tastendruck und nehmen jetzt die `switch`-Anweisung in der Methode `tasteReagieren(...)` in Gebrauch:

```
/*
 * In der switch-Anweisung der Methode tasteReagieren(int code)
 */

//Bewegung der Ebene
case 26: //Pfeil oben
     ebene.bewegen(0, -10);
     break;
case 27: //Pfeil rechts
     ebene.bewegen(10, 0);
     break;
case 28: //Pfeil unten
     ebene.bewegen(0, 10);
     break;
case 29: //Pfeil links
     ebene.bewegen(-10, 0);
     break;
```

In diesem Quellcode wird die Ebene durch die Pfeiltasten bewegt.
Man sieht, dass sie das rote Quadrat als Aktiv-Objekt immer mitbewegt, solange es auf ihr drauf „steht“.

Erweitern wir die `switch`-Anweisung um einen weiteren Teil, so können wir auch das rote Quadrat als Aktiv-Objekt bewegen – mit den Tasten 'W', 'D', 'S' und 'A'.

```
/*
 * In der switch-Anweisung der Methode tasteReagieren(int code)
 */

//Bewegung des roten Quadrates
case 22: //W-Taste
    aktiv.bewegen(0, -10);
    break;
case 3: //D-Taste
    aktiv.bewegen(10, 0);
    break;
case 18: //S-Taste
    aktiv.bewegen(0, 10);
    break;
case 0: //A-Taste
    aktiv.bewegen(-10, 0);
    break;
```

Es ist bestimmt klug, ein bisschen mit den Bewegungen zu experimentieren: Du kannst die Zahlenwerte verändern, und du kannst mit den Bewegungen der Ebene und des roten Quadrates spielen.

Du kannst auch noch zusätzliche Passiv-Objekte einbauen, um zu sehen, wie eine echte Spielsituation laufen könnte.
Zum Beispiel durch eine Mauer. Dieser Quelltext muss nur im Konstruktor ergänzt werden, und schon hast du eine Mauer im Spiel:

```
//Erstellen einer zusaetzlichen "Mauer"
Rechteck mauer = new Rechteck(200, 100, 10, 200);
mauer.farbeSetzen("Weiss");
wurzel.add(mauer);
mauer.passivMachen();
```

Lasse die Mauer dein Quadrat aufhalten und probiere ein bisschen herum: Erstelle neue Passiv-Objekte und denke dir neue Situationen aus!

### Erfolgreiche und nicht erfolgreiche Bewegungen

Wie du gemerkt haben wirst, werden nicht alle Bewegungen voll geklappt haben: Wenn dein Quadrat gegen die Mauer oder die Ebene bewegt wurde ist **gar nichts** passiert: Die Bewegung war nicht erfolgreich.

Eine Bewegung ist immer dann nicht erfolgreich, wenn das zu bewegende Objekt seine Bewegung nicht beenden konnte, weil es von einem Passiv-Objekt geblockt wurde.

| |
|---|
| Möchtest du wissen, ob deine Bewegung erfolgreich war, kannst du *den Rückgabewert* der Methode `bewegen(...)` nutzen. |

| *Der Rückgabewert der Methode `bewegen(...)` ist:* | | |
|---|---|---|
| **Bei einem Aktiv-Objekt** | **Bei einem Passiv-Objekt** | **Bei einem neutralen Objekt** |
| `true`, wenn die Bewegung erfolgreich war. Dies ist der Fall, wenn das Objekt nicht bei der Bewegung von einem Passiv-Objekt geblockt wurde. Wurde es geblockt, ist die Rückgabe `false`. | Immer `true`, da bisher Passiv-Objekte ihre Bewegung immer voll ausfüllen können. | Immer `true`, da ein neutrales Objekt in seiner Bewegung nicht aufgehalten werden kann. |

Das Beispiel verwendet keine Rückgabewerte, da nicht wichtig ist, ob die Bewegung erfolgreich war oder nicht. Du musst den Rückgabewert also nur dann nutzen, wenn du ihn wirklich brauchst!

### *ii Springen*

Damit das Jump in „Jump n' Run“ kommt, können Aktiv-Objekte in der Physik springen.

#### Die Methode `sprung(...)`

Hierfür gibt es eine Methode in der Klasse `Raum`:

```
public boolean sprung(int kraft)
```

| *Die Methode `sprung(int kraft)` bewirkt:* | | |
|---|---|---|
| **Bei einem Aktiv-Objekt** | **Bei einem Passiv-Objekt** | **Bei einem neutralen Objekt** |
| **Nichts**, wenn das Objekt nicht auf einem Passiv-Objekt steht.<br>Steht das Aktiv-Objekt, springt es. Je höher der Eingabeparameter, desto kräftiger ist der Sprung | Nichts, da passive Objekte nicht springen können. | Nichts, da neutrale Objekte nicht springen können. |

In unserem Beispiel können wir also auch unser Aktiv-Objekt, das rote Quadrat, springen lassen. Das soll bei Druck auf die Leertaste passieren. Also erweitern wir wieder die `switch`-Anweisung in der Methode `tasteReagieren(...)`.

```
/*
 * In der switch-Anweisung der Methode tasteReagieren(int code)
 */

//Sprung des roten Quadrats
case 30: //Leertaste
    aktiv.sprung(5);
    break;
```

Experimentiere mit verschiedenen Werten für die Sprungkraft (im Beispiel ist der Wert `5`).

### Erfolgreiche und nicht erfolgreiche Sprünge

Genau wie die Bewegung in der Physik nicht immer erfolgreich sein kann, kann auch ein Sprung in der Physik nicht immer erfolgreich sein.

| |
|---|
| Möchtest du wissen, ob ein Sprung erfolgreich war, kannst du *den Rückgabewert* der Methode `sprung(...)` nutzen. |

| *Der Rückgabewert der Methode `sprung(...)` ist:* | | |
|---|---|---|
| **Bei einem Aktiv-Objekt** | **Bei einem Passiv-Objekt** | **Bei einem neutralen Objekt** |
| `true`, wenn das Aktiv-Objekt springen konnte. Das ist immer der Fall, wenn es auf einem Passiv-Objekt steht.<br>Ist dies nicht der Fall, kann das Aktiv-Objekt auch nicht springen und die Rückgabe ist `false`. | Immer `false`, da Passiv-Objekte nicht springen können. | Immer `false`, da neutrale Objekte nicht springen können. |

### *iii Wie tief kann man fallen?*

Beim Beispiel ist Dir bestimmt schon einmal passiert, dass dein rotes Quadrat aus dem Bild gefallen ist. ***Auf zu tiefes Fallen kann man reagieren!***

Dadurch kannst du, wenn deine Spielfigur zu tief aus dem Bild gefallen ist, Maßnahmen ergreifen.

| Auf Fallen unter eine bestimmte Höhe kannst du mit dem Interface `FallReagierbar` reagieren. |
|---|


*Die Klassenkarte des Interfaces `FallReagierbar`*

Zunächst musst du in irgendeiner Klasse das Interface implementieren. In unserem Beispiel mache ich das in der Klasse `Spiel`:

```
import ea.*;

public class Spiel
extends Game
implements FallReagierbar { //<-Implementieren des Interfaces!!
    [...]
}
```

Anschließend musst du die Methode `fallReagieren()` aus dem Interface `FallReagierbar` überschreiben. Überlege dir: „Was soll passieren, wenn mein Quadrat zu tief fällt?“

```
/**
 * Diese Methode wird aufgerufen, wenn das
 * rote Quadrat zu tief gefallen ist. Was "zu tief" ist, kann man
 * selbst bestimmen.
 */
@Override
public void fallReagieren() {
    //Das Quadrat wieder nach oben bringen
    aktiv.positionSetzen(30, 30);
}
```

Nun ist das Interface erfolgreich implementiert. Aber du bist noch nicht fertig!
Denn noch kennen sich dein Quadrat und dein Interface nicht. Du musst noch das Interface bei deinem Quadrat anmelden. Hierfür gibt es in der Klasse `Raum` eine Methode:

```
public void fallReagierbarAnmelden(     FallReagierbar fr,
                                        int kritischeTiefe)
```

Die Parameter bedeuten:

| | |
|---|---|
| `fr : FallReagierbar` | Die `fallReagieren()`-Methode **dieses** `FallReagierbar`-Interfaces wird immer wieder aufgerufen, solange es unterhalb der kritischen Tiefe liegt. |
| `kritischeTiefe : int` | Die Frage ist:<br>Wann ist ein Aktiv-Objekt „zu tief“ gefallen?<br>Das kannst Du selbst bestimmen!<br>Sobald das `Raum`-Objekt unterhalb dieser Tiefe ist, wird das angemeldete Interface **immer wieder** informiert.<br>Deshalb ist es wichtig, dass in der `fallReagieren()`-Methode das Objekt auch wieder **über die kritische Tiefe** gebracht wird. |

Diese Methode rufen wir jetzt bei dem Aktiv-Objekt unseres Beispiels – dem roten Quadrat – auf. Der Konstruktor der Klasse `Spiel` wird um diese Zeilen erweitert:

```
//Das FallReagierbar-Interface (=diese Klasse)
//am roten Quadrat anmelden (kritische Tiefe = 500)
aktiv.fallReagierbarAnmelden(this, 500);
```

Ab **jetzt** wird deine `fallReagieren()`-Methode aufgerufen!

Übrigens: Das ist ebenfalls das *Listener-Prinzip*, das hierbei genutzt wird. Wenn du mehr wissen willst, sieh dir das Kapitel „*Reagieren auf Aktionen des Spielers*“ an.

### *iv Das geht alles: Zusammenfassung*


*Die Klassenkarte von `Raum` mit den wichtigsten Methoden für die „Physik“*

Das waren nochmal alle Methoden, die hier behandelt wurden. Hier sind sie nochmal kurz zusammengefasst:

| Methodenname | Parameter | Erklärung |
|---|---|---|
| `aktivMachen` | - Keine - | Macht das `Raum`-Objekt zu einem Aktiv-Objekt. |
| `passivMachen` | - Keine - | Macht das `Raum`-Objekt zu einem Passiv-Objekt. |
| `neutralMachen` | - Keine - | Macht das `Raum`-Objekt zu einem neutralen Objekt. |
| `bewegen` | `idealeBewegung : Vektor` | Bewegt das `Raum`-Objekt in der Physik. Die Bewegung kann bei einem Aktiv-Objekt durch ein Passiv-Objekt blockiert werden. Der Rückgabewert gibt an, ob die Bewegung erfolgreich war. |
| `bewegen` | `x : int`<br>`y : int` | Macht das selbe wie `bewegen(Vektor)`, nur mit einfachen `int`-Parametern für die X- und Y-Bewegung. |
| `sprung` | `kraft : int` | Lässt das `Raum`-Objekt springen. Dies funktioniert nur bei einem Aktiv-Objekt. Der Rückgabewert gibt an, ob der Sprung erfolgreich war. |

| | | |
|---|---|---|
| `fallReagierbarAnmelden` | `fr : kallReagierbar`<br>`kritischeTiefe : int` | Meldet ein `FallReagierbar`-Interface am `Raum`-Objekt an.<br>Dies funktioniert nur bei einem Aktiv-Objekt.<br>Das Interface wird von nun an immer informiert, wenn das Raum-Objekt unterhalb einer kritischen Tiefe (als Parameter mitgegeben) gefallen ist. |

Details lassen sich auch in der Dokumentation der Klasse `Raum` nachlesen.

## PROBLEME?

Ein Beispiel-Projekt, das nach den Erklärungen gemacht wurde, kannst du von der EA-Website herunterladen: Hier.

# K Die Klasse Game unter der Lupe

Aus der Klasse `Game` leitet sich immer die spielsteuernde Klasse ab.
Hier wird sie nun einmal voll beleuchtet.

## 1 Attribute

Zwei Attribute der Klasse `Game` kennst Du schon, nämlich `wurzel` und `manager`.

Hier werden *alle* Attribute, auf die man zugreifen kann, aufgelistet und erläutert. Details zu den Klassen lassen sich in der Dokumentation nachlesen.

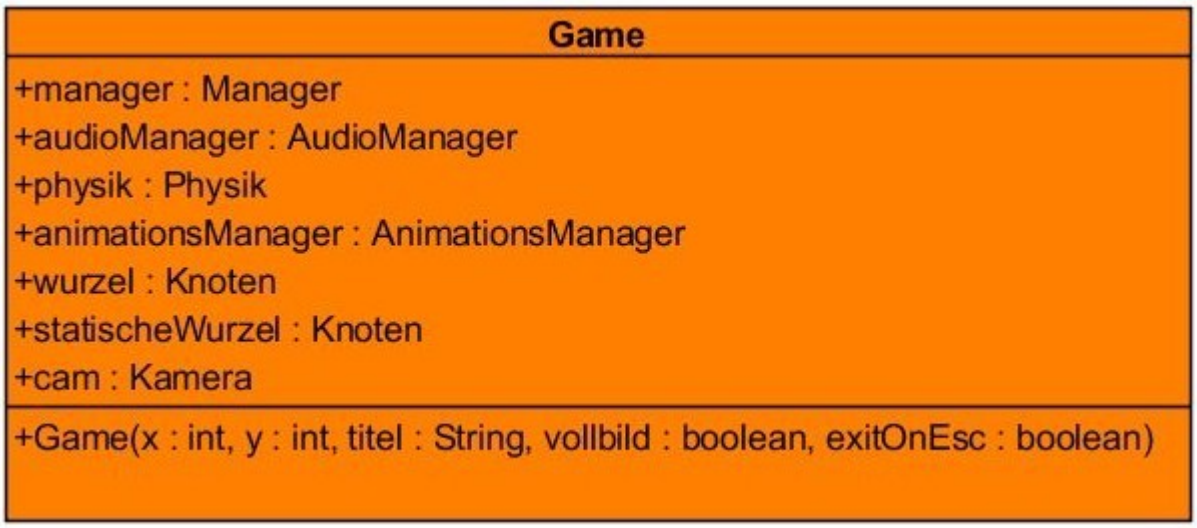
| Name | Klasse | Erläuterung |
|---|---|---|
| `manager` | `Manager` | Das „`Manager`“-Objekt.<br>Hieran können „Ticker“ angemeldet werden (bis zu 10 Stück gleichzeitig). |
| `audioManager` | `AudioManager` | Ein „`AudioManager`“-Objekt.<br>Hierüber können sehr leicht Sounds wiedergegeben werden. Mehr hierzu im Kapitel „*Sound*“. |
| `animationsManager` | `AnimationsManager` | Ein „Animations-Manager“.<br>Über dieses Objekt können `Raum`-Objekte animiert werden, zum Beispiel auf Kreisbahnen oder beliebig festgelegten Strecken. Mehr hierzu im Kapitel „*Animationen*“. |
| `wurzel` | `Knoten` | Die Wurzel.<br>Alle zu zeichnenden Objekte der Zeichenebene müssen hier angemeldet werden. |
| `statischeWurzel` | `Knoten` | Die statische Wurzel.<br>Bei Verschiebung des Bildes werden Objekte, die hier angemeldet sind, immer noch an der selben Position im Fenster angezeigt; das ist zum Beispiel für eine Punkte- oder Lebensanzeige sinnvoll. Mehr hierzu in Kapitel „*Die Kamera – statische grafische Objekte*“. |
| `cam` | `Kamera` | Die Kamera.<br>Über die Kamera kann man die Verschiebung des Bildes organisieren und ein Fokusobjekt bestimmen. Mehr hierzu im Kapitel „*Die Kamera*“. |

## 2 Konstruktoren

Die Klasse `Game` hat viele Konstruktoren. Der volle Konstruktor hat folgende Parameter:

| Parameter | Erläuterung |
|---|---|
| `x : int` | Die Breite des Fensters |
| `y : int` | Die Höhe des Fensters |
| `titel : String` | Der Titel, unter dem das Fenster läuft |
| `vollbild : boolean` | Wenn dieser Wert `true` ist, wird – sofern möglich – das Fenster als Vollbild fungieren. In diesem Fall werden die `x`- und `y`-Parameter ignoriert, da die Größe des Fensters nun die Maße bestimmt.<br>Mehr hierzu im Kapitel „*Vollbildmodus*“. |
| `exitOnEsc : boolean` | Ist dieser Wert `true`, wird beim Drücken des „Esc“-Knopfes das Fenster geschlossen und die virtuelle Maschine von Java beendet. Dies ist standardmäßig der Fall. |

Es gibt weitere verschiedene Konstruktoren, die gewisse Parameter nicht benötigen. Diese lassen sich alle in der Dokumentation der Klasse `Game` nachlesen.




*Klassenkarte der Klasse Game mit Attributen und größtem Konstruktor*

## 3 Methoden

Es gibt auch einige sehr praktische Methoden, mit denen man sein Spiel aufpeppen und ein breiteres Möglichkeitenfeld nutzen kann.

| Name | Parameter | Erläuterung |
|---|---|---|
| `zufallsZahl` | `maximum : int` | Gibt eine zufällige ganze Zahl zwischen 0 und dem definierten Maximum zurück, wobei sowohl 0 als auch das Maximum mögliche Ergebnisse sind. |
| `zufallsBoolean` | `[keine]` | Gibt zufällig einen Wahrheitswert zurück; das Ergebnis ist mit gleicher Wahrscheinlichkeit `true` oder `false`. |
| | | |
| `eingabeFordern` | `nachricht : String` | Öffnet ein neues kleines Fenster und fordert eine Texteingabe. Dazu wird eine Nachricht angezeigt, die erklären sollte, wozu diese Eingabe dient.<br>Das Ergebnis der Forderung wird dann als `String` zurückgegeben. |
| `frage` | `frage : String` | Öffnet ein neues kleines Fenster, in dem die einzugebende Frage angezeigt wird.<br>Darunter ist je ein Knopf mit „Ja“ und „Nein“.<br>Diese Methode gibt ein `boolean`-Argument zurück; `true`, wenn „Ja“ gedrückt wurde, `false`, wenn „Nein“ gedrückt wurde. |
| `sicherheitsFrage` | `frage : String` | Prinzipiell genau dasselbe wie die `frage`-Methode. Jedoch wird hier anstatt „Ja“ ein „OK“ und anstatt „Nein“ ein „Abbrechen“ abgebildet. |
| `nachrichtSchicken` | `nachricht : String` | Öffnet ein neues kleines Fenster, das nur eine kleine Textnachricht beinhaltet und einen „OK“-Knopf.<br>Wie alle anderen Fenster öffnenden Methoden ist diese erst dann beendet, wenn das Fenster geschlossen wurde. |
| `highscoreAnzeigen` | `namen : String[]`<br>`punkte : int[]` | Öffnet ein neues Fenster, das eine beliebig große Liste von Highscores anzeigen kann.<br>Dafür werden 2 gleich große, gefüllte Arrays erwartet, eines mit den Namen (Datentyp `String`) und eines mit den Punkten (Datentyp `int`) `namen[0]` und `punkte[0]` gehören zusammen usw. |
| `fensterFontSetzen` | `fontname : String` | Setzt den Font, in dem alle Texte in den zusätzlichen Fenstern (siehe obere Methoden) dargestellt werden.<br>Für den Fontnamen gelten die selben Regeln, wie sie auch für Fontnamen in der Klasse `Text` gelten.<br>Optional kann ein `int`-Argument hinzugefügt werden, um die Schriftgröße zusätzlich anzupassen. |
| | | |
| `beenden` | `[keine]` | Beendet absolut alles.<br>Schließt zuerst das Fenster und beendet dann die virtuelle Maschine von Java. |

| | | |
|---|---|---|
| | | Daher sollte diese Methode nur dann ausgeführt werden, wenn das Spiel wirklich beendet werden soll. |


*Erweiterte Klassenkarte der Klasse* `Game` *(ohne Konstruktoren)*

# L Vollbildmodus

In den beiden größten Konstruktoren der Klasse `Game` besteht die Option auf einen *Vollbildmodus*.

Im einfacheren Konstruktor:

```
public Game(int breite, int hoehe, String titel, boolean vollbild)
```

Ist der vierte Parameter `true` (standartmäßig ist er `false`), so wird der gesamte Bildschirm (inklusive der Taskleiste etc.) vom Spielfenster ausgefüllt, was die exakten Maße des Fensters des Computers annimmt, auf dem das Spiel gestartet wird.
Das macht alle anderen drei Parameter bedeutungslos.

Manche Computer unterstützen keinen echten Vollbildmodus. In diesem Fall wird ein möglichst großes, ausfüllendes Fenster erstellt, um möglichst nahe an Vollbild-Bedingungen zu kommen.

Im Grunde wäre damit die ganze Erläuterung beendet, aber eine dringende Warnung muss ausgesprochen werden:

Beim Vollbildmodus weiß der Entwickler nie, wie groß der Bildschirm des Benutzers ist, und damit sind auch die Maße des Fensters nicht bekannt!

Das kann heißen, dass beim Spielen auf einem anderen PC nicht die gesamte Spielfläche sichtbar ist, oder neben der Spielfläche klaffender, schwarzer Freiraum ist, weil der PC-Bildschirm andere Maße hat, als der, an dem das Spiel entwickelt und getestet wurde.
Daher muss ein solches Spiel wesentlich besser geplant werden als ein normales.

Die Größe des Fensters lässt sich jederzeit über die folgende Methode der Klasse `Game` erfragen:

```
public BoundingRechteck fensterGroesse()
```

Zurückgegeben wird ein `BoundingRechteck` an der Position (0|0), dessen Maße exakt die des Fensters sind.

Folgendermaßen kann man sie sich holen:

```
/*Irgendwo in der spielsteuernden Klasse*/

//Deklaration der Zahlen fuer Die Fensterbreite und -hoehe
int fensterbreite;
int fensterhoehe;

//Dieses BoundingRechteck repraesentiert die Fenstermasse
BoundingRechteck recht = fensterGroesse();

//Die Informationen fuer die Zahlen werden aus dem Rechteck geholt
fensterbreite = recht.breite();
fensterhoehe = recht.hoehe();
```

**Achtung!**
Der Vollbildmodus der Engine ist noch nicht ausgereift, aus einem einfachen Grund:
Die Größe des „Fensters“ hängt immer vom Bildschirm ab, nie aber von den eigenen Einstellungen.
Deshalb ist dies zu ungenau. Hierdran weiterzuarbeiten, ist geplant.

# M Das Projekt exportieren

Wenn Dein Spiel fertig ist, soll es natürlich nicht in der Entwicklungsumgebung bleiben. Damit es auch die Nutzen können, die auf ihrem PC keine Entwicklungsumgebung haben, kannst Du Dein Spiel in eine ausführbare .jar-Datei exportieren. Dann kann jeder Dein Spiel öffnen, der *JAVA* auf dem PC hat.

## 1 Die Exportschritte bei BlueJ

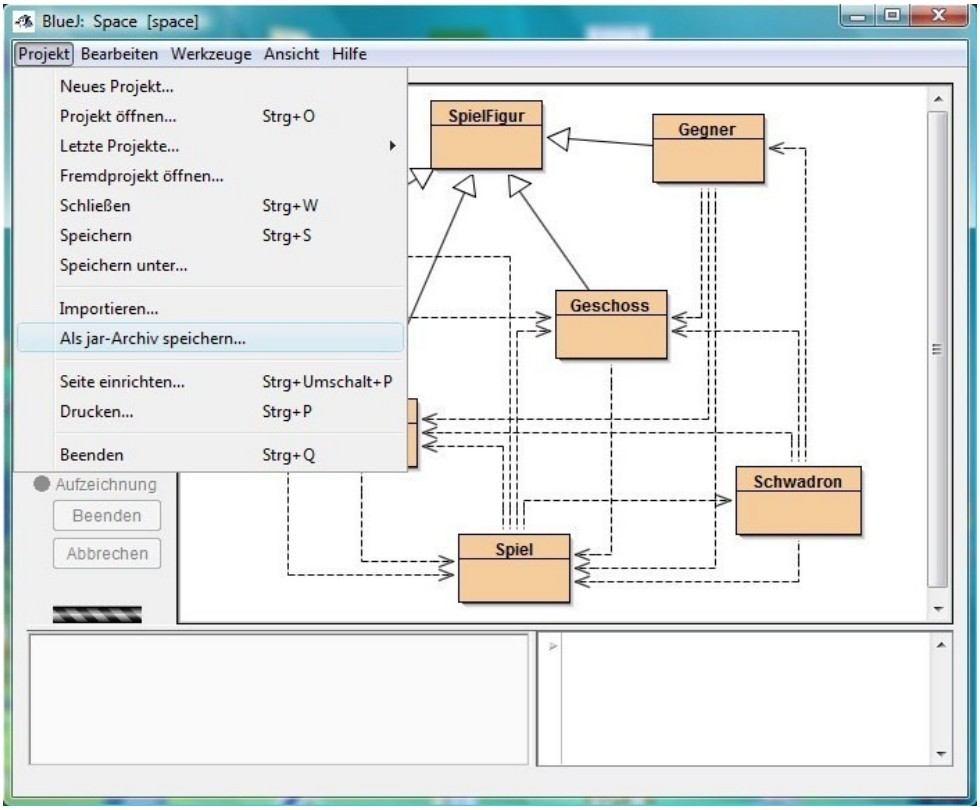
1. **Du brauchst eine `main`-Methode**
   Meist beginnt ein Spiel, indem Du ein Objekt Deiner spielsteuernden Klasse erstellst. So funktionieren unter Anderem die Beispiele des Handbuches. Damit ein solches Objekt auch im Exportierten Projekt wieder erstellt wird, muss dies in der `main`-Methode stattfinden.
   Im folgenden Beispiel soll ein Objekt der Klasse `Spiel` in der `main`-Methode erstellt werden:

```
/*
 * In der spielsteuernden Klasse 'Spiel'
 */

/**
 * Main-Methode. Wird automatisch beim Oeffnen des
 * exportierten Projektes aufgerufen.
 * @param args Der Parameter hat keinen Einfluss auf die
 * Methode
 */
public static void main(String[] args) {
    new Spiel();
}
```

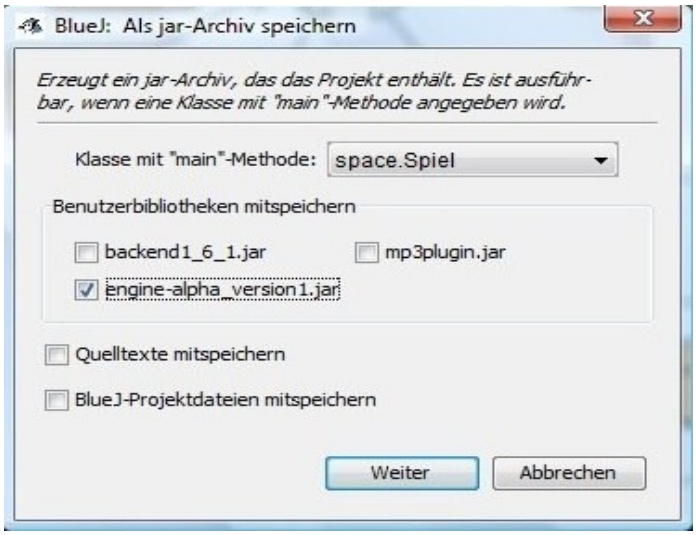
   Wenn Du Dir nicht sicher über die Verwendung der Methode bist, kopiere einfach diesen Quelltext in Deine `Spiel`-Klasse.

**2. Aus der Entwicklungsumgebung heraus**

Die `Spiel`-Klasse ist jetzt dank der `main`-Methode *ausführbar*. Als nächstes kann das Projekt exportiert werden. Gehe in *BlueJ* hierzu auf *Datei->als jar-Archiv speichern*.




Im anschließenden Fenster wähle Deine `Spiel`-Klasse als *„Klasse mit 'main'-Methode“* und hake bei *Benutzerbibliotheken mitspeichern* die Engine-Alpha-Bibliothek an. Gehe dann auf *Weiter*. Dann kannst Du das Projekt speichern.




Es entsteht ein Ordner mit 2 .jar-Dateien. Öffne die .jar-Datei, mit dem Namen, den Du beim Exportieren für das Projekt gewählt hast, um die `main`-Methode auszuführen.

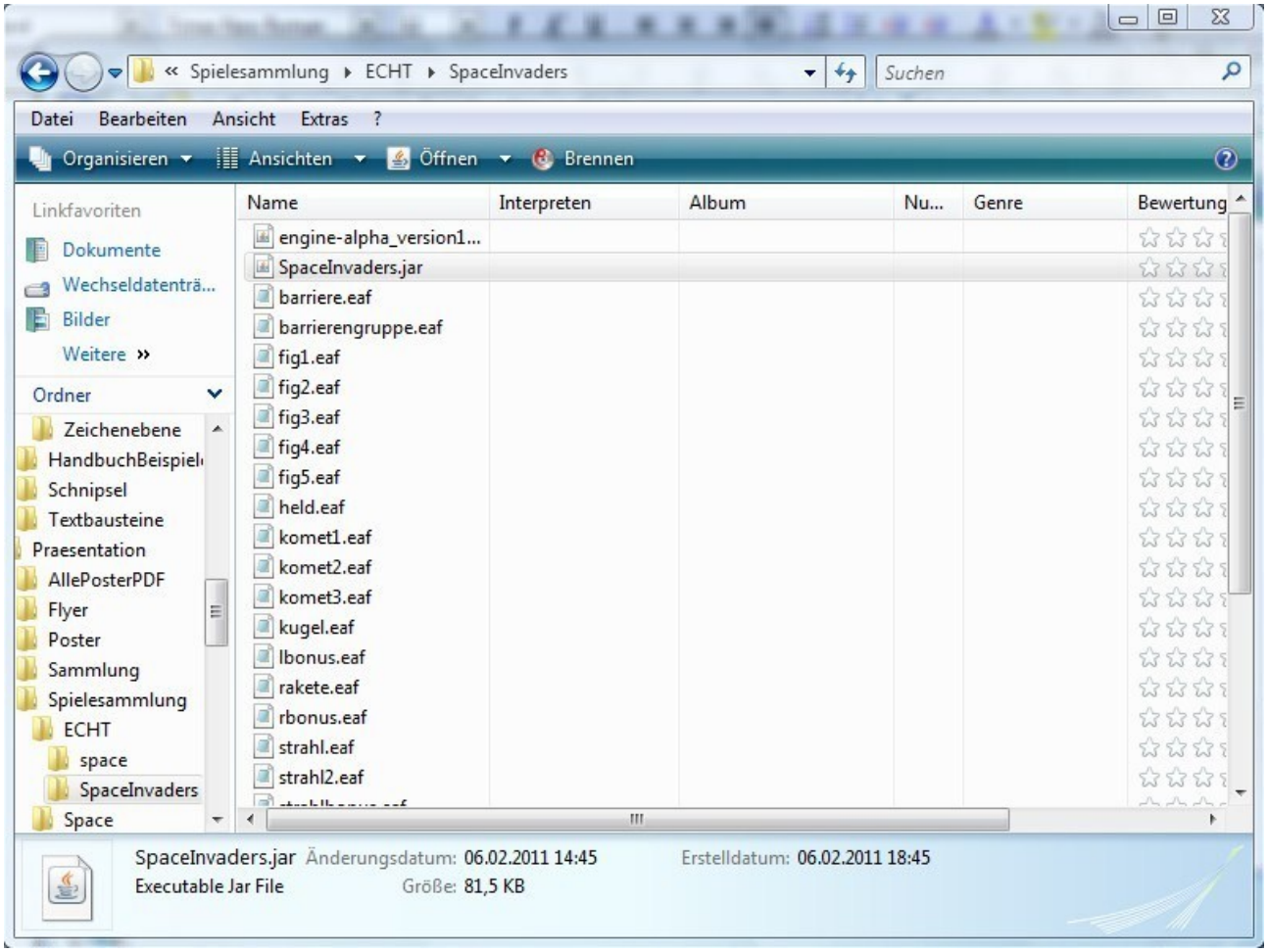
## 2 Zusätzliche Dateien im Spiel

Wenn Du in deinem Spiel zusätzliche Dateien hast, zum Beispiel

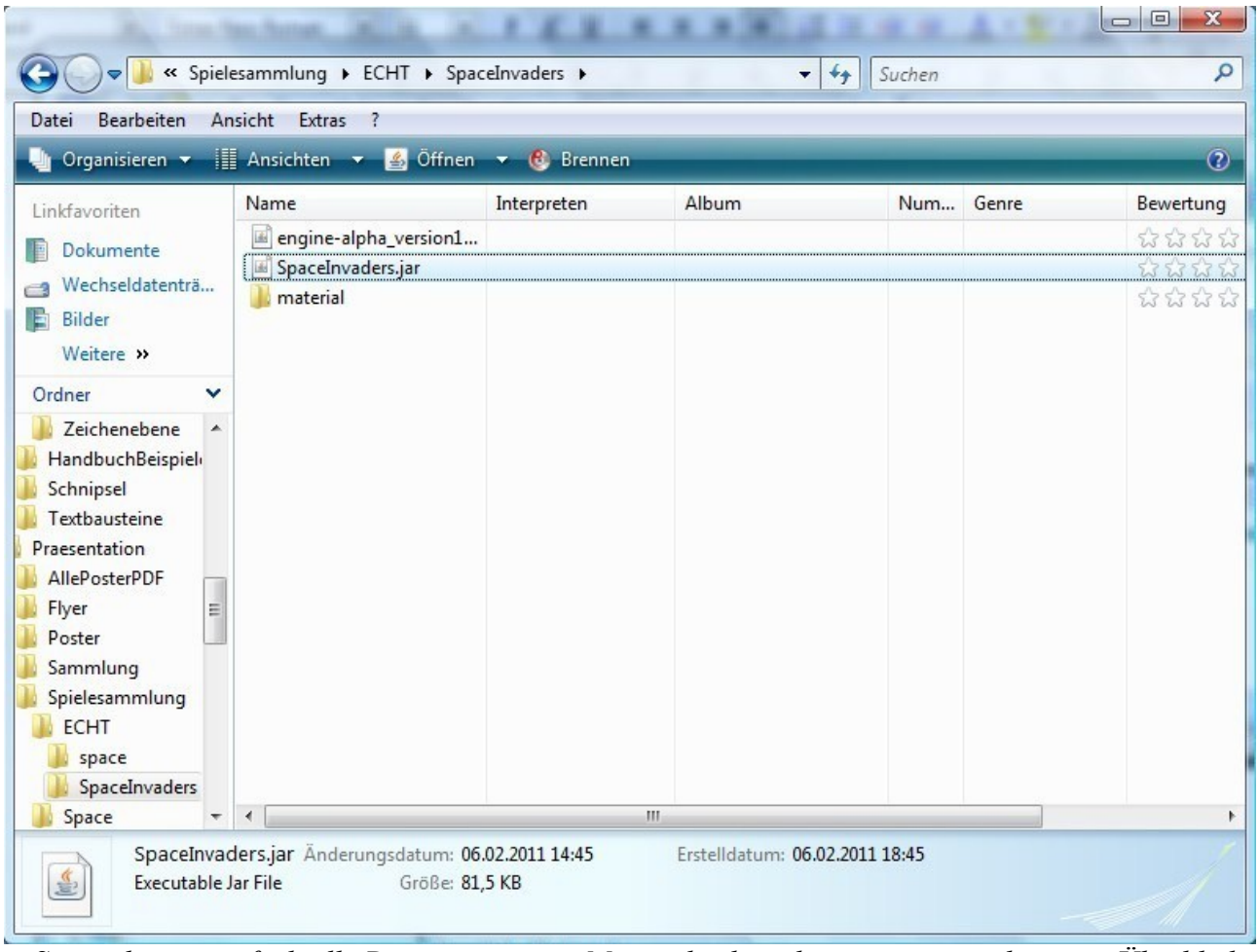
- Bilder
- Sounds
- Animierte Figuren (.eaa-Dateien)

müssen diese nach dem Export ebenfalls in den Ordner mit der ausführbaren .jar-Datei kopiert werden.

Das heißt:
Alle Dateien, die Du im Projekt nutzt (.jpg, .wav, .eaf etc.), werden in den Ordner mit dem exportierten Projekt kopiert und dann während des Exports aus dem Projektordner entfernt (spart Speicherplatz!). Damit hierbei nicht die Übersicht verloren geht, ist es sinnvoll, alle Dateien schon beim Programmieren in einem Ordner für Material zu sammeln.




*Ein exportiertes Projekt, bei dem die zusätzlichen Dateien ungebündelt liegen. Das ist sehr unübersichtlich, aber es gibt eine Alternative...*




*...Sammelt man einfach alle Dateien in einem Materialordner, hat man einen sehr guten Überblick.*

Nutzt man einen Materialordner, so muss dieser natürlich auch bei Verzeichnisangaben berücksichtigt werden. Wie dies funktioniert, erläutert das Kapitel „*Dateien Speichern und Lesen – Arbeiten mit Verzeichnissen*“.

# VI Schlusswort

„Spieleentwickler sind die Könige unter den Programmierern.“
Viele Programmierer – auch etablierte – sehen die Spieleprogrammierung als ein Ressort, das ihnen über den Kopf wächst. Im besten Fall konnte dir die Engine Alpha die Spieleentwicklung trotzdem ermöglichen und dich so zu einem kleinen König krönen.

Ich hoffe, dass Dich dieses ausführliche Handbuch nicht verschreckt, entmutigt oder eingeschüchtert hat. Mein Wunsch ist es, dass die Engine Alpha allen Interessierten ermöglicht, ein eigenes Spiel zu entwickeln, die umfangreichen Ansprüche und Möglichkeiten der Software auszureizen und Spaß an der Informatik zu finden.

Wenn Dir die Erklärungen des Handbuches und Referenzen auf Informatik-Modelle, die Arbeit mit der Engine und auch die Arbeit der Engine selbst gefallen und Dich beschäftigt haben, dann war die Engine Alpha für Dich und auch für mich ein voller Erfolg.

Ich wünsche Dir viel Spaß mit Deinen eigenen Computerspielen! :-)

Sonntag, 3. April 2011
Michael Andonie

# VII Anhang

**Die Tastenliste:**

| **Alphabet** | | | | | | **Verschiedenes** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 0 | | **N** | 13 | | **Pfeiltaste oben** | 26 | | **6** | 39 | |
| **B** | 1 | | **O** | 14 | | **Rechts** | 27 | | **7** | 40 | |
| **C** | 2 | | **P** | 15 | | **Unten** | 28 | | **8** | 41 | |
| **D** | 3 | | **Q** | 16 | | **Links** | 29 | | **9** | 42 | |
| **E** | 4 | | **R** | 17 | | **Leertaste** | 30 | | | | |
| **F** | 5 | | **S** | 18 | | **Enter-Taste** | 31 | | | | |
| **G** | 6 | | **T** | 19 | | **Escape-Taste** | 32 | | | | |
| **H** | 7 | | **U** | 20 | | **0-Taste** | 33 | | | | |
| **I** | 8 | | **V** | 21 | | **1** | 34 | | | | |
| **J** | 9 | | **W** | 22 | | **2** | 35 | | | | |
| **K** | 10 | | **X** | 23 | | **3** | 36 | | | | |
| **L** | 11 | | **Y** | 24 | | **4** | 37 | | | | |
| **M** | 12 | | **Z** | 25 | | **5** | 38 | | | | |