1 Allgemeiner Ablauf

Die Spiel-Enginge ist frame-basiert, im Gegensatz zu zeit-basierten Engines. Dies bedeutet, dass die kleinste Zeiteinheit innerhalb des Spiels ein „Frame“ ist, festgelegt auf mindestens $\frac{1}{50}$ einer Sekunde. Kann der Rechner, auf dem das Spiel läuft, diese Frame-Rate aufgrund mangelnder Leistung nicht erreichen, läuft das Spiel entsprechend langsamer ab.

Innerhalb eines Frames werden alle notwendigen Aktionen mit allen Objekten des momentanen Levels ausgeführt. Dies beeinhaltet:

- Abfrage von Tastatur und Joystick und ggf. Änderung der Bewegungsrichtungen der Spielfigur.
- Bewegen der Spielfigur:
  - Bewegung der Spielfigur unter Beachtung von Levelblöcken
  - Markieren aller berührten Bonuskisten
  - Markieren der Spielfigur als „tot“, falls sie den Bildschirm nach unten verlässt.
- Neuberechnung des Kamera-Ausschnitts.
- Test, ob das Ende des Levels erreicht wurde.
- Den Neu-Aufbau des Bildschirminhalts: Hintergrund, Level, Spieler, Gegner, sonstige Objekte, HUD
- Berechnung der Zustände für das nächste Frame. Dies beeinhaltet wiederum:
  - Prüfung der Kollision des Spielers mit Gegnern bzw. Objekten und Reaktion darauf (Gegner töten oder Energie abziehen oder Spieler töten)
  - Entfernen aller als „tot“ markierten Gegner
  - Behandlung aller als berührt markierter Bonuskisten
  - Prüfen, ob der Spieler ein Leben verloren hat und ggf. den Level oder das Spiel neu starten
- Falls der gesamte Vorgang keine $20ms$ gedauert hat, den Spiel-Thread für die restliche Zeit schlafen legen, um anderen Programmen die CPU zu überlassen.

2 Sprites

„Sprite“ ist der Oberbegriff für alle Typen von Grafikinformationen, die auf dem Bildschirm angezeigt werden können. Sprite ist auch der Name der Klasse, die all diese Informationen speichert und Methoden zum Zeichnen zur Verfügung stellt.

Das Spiel unterstützt sowohl die hardwarebeschleunigte Grafikausgabe per OpenGL als auch Software-Rendering über die SDL. Wird es mit OpenGL-Untersützung kompiliert, speichert Sprite eine OpenGL-Textur und die reinen Bilddaten als SDL-Surface. Wird die SDL-Version verwendet, speichert Sprite die Bilddaten einmal als SDL-Surface und noch einmal gespiegelt als SDL-Surface.

Der Grund für das Speichern der SDL-Surface auch bei Verwendung von OpenGL ist, dass das Laden von Texturen aus dem Speicher der Grafikkarte verhältnismäßig lange dauert, was die Performance bei der pixelgenauen Kollisionsprüfung stark verschlechtern werde.

Die gespiegelte Surface im SDL-Fall ist notwendig, da SDL selbst (im Gegensatz zu OpenGL) keine direkte Möglichkeit anbietet, Surfaces gespiegelt zu zeichnen.

3 Figur

Eine Figur ist ein beliebiges Objekt im Spiel, mit dem auf irgendeine Weise interagiert werden kann. Dies kann ein Gegner sein, ein Ring, ein Stachel oder ähnliches. Die vom Spieler gesteuerte Figur ist ebenfalls eine Instanz der Klasse Figur. Prinzipiell werden all diese Figuren erst einmal gleich behandelt: Sie führen eine Animation durch, unterliegen der Graviation (die aber auch 0 sein darf) und führen ihre Steuerfunktionen (sie weiter unten) aus. Unterschiedliches Verhalten wird durch Änderung der Attribute, Überschreiben der Methoden in Unterklassen und vor allem durch Hinzufügen von Steuerfunktion erreicht.

Andererseits haben einige spezielle Figurtypen natürlich auch spezielle Methoden, z.B. besitzen nur Bonuskisten die Methode `ausfuehren`, die ausgelöst wird, wenn eine solche Kiste von der Spielfigur von unten angesprungen wird. Außerdem werden die Figuren unterschiedliche weit bekannt gemacht. Gegner sind nur der Haupt-Steuerroutine bekannt, Bonuskisten dagegen auch dem Level, da sie undurchdringliche Objekte darstellen, was das Scrolling und den freien Fall von anderen Figuren beeinflusst.

3.1 Steuerfunktionen

Steuerfunktionen sind für die Bewegung aller Objekte (inklusive der Spielfigur selbst) zuständig. Dies beeinhaltet nicht die (objektspezifische) Gravitation, diese wirkt automatisch auf jedes Objekt, dass einer von Figur abgeleiteten Klasse angehört.

Bei Steuerfunktionen wird einmal pro Frame die Methode `ausfuehren` aufgerufen, innerhalb derer sie alles mit der zugehörigen Figur machen können. Außerdem werden, sollte während der Ausführung der Methode `naechsterZustand` einer Figur eine Kollision auftreten, alle angehängten Steuerfunktionen über die Methode `kollision` darüber informiert.

Steuerfunktionen besitzen außerdem die Methode `istBeendet`. Diese wird einmal pro Frame aufgerufen. Liefert die Methode den Werte `true`, so wird die Steuerfunktion von der zugehörigen Figur entfernt.

Theoretisch ist es möglich, beliebig viele Steuerfunktionen an eine Figur anzuhängen. In der Praxis ist es aber empfehlenswert, maximal eine Funktion, welche die X-Richtung beeinflusst und eine Funktion, welche die Y-Richtung beeinflusst, zu verwenden.

Momentan gibt es folgende Steuerfunktionen:

- Spielerbewegung: beeinflusst die X-Richtung der Bewegung einer Figur. tut normalerweise gar nichts und reagiert nur auf explizite Methodenaufrufe. Wird benutzt, um die Spielfigur über Eingabegeräte zu steuern.

- SpielerbewegungY: wie Spielerbewegung, führt nach einem Aufruf von `springen` oder `gegner_bounce` einige Frames lang selbsttätig die entsprechenden Sprungbewegungen durch.

- Gegnersprung: lässt eine Figur immer wieder springen, sobald sie den Boden berührt.

- Linksrechtsbewegung: bewegt eine Figur immer weiter in eine Richtung, bis sie auf ein Hindernis stößt, dann welchselt die Figur die Richtung.

- Bonussprung: Lässt eine Figur ein kleines Stück nach oben springen und wieder fallen. Sobald die Figur auf dem Boden aufkommt, wird sie gelöscht. Wird für die Animation von Ringen und Herzen aus Bonuskisten benutzt.

3.2 Animatoren und Animationen

Damit eine Spielfigur nicht nur aus einem Bild besteht, werden die sogenannten Animatoren verwendet. Diese stellen fünf Animationen (Stehen, Gehen, Rennen, Springen, Fallen) und zwei Richtungen (links, rechts) zur Verfügung. Diese wiederum bestehen aus einzelnen Bildern, die für eine gewisse Anzahl Frames angezeigt werden, bevor zum nächsten Bild gewechselt wird. Die Animationen werden aus `.anim`-Dateien geladen, welche einfach eine Liste von Grafikdateien und ihrer Anzeigedauer (in Frames) enthalten.

3.3 Level

Der Level besteht aus 32×32 Pixel großen Blöcken. Diese werden als Tiles (Sprites mit einer zusätzlichen Positionsangabe) in einem „Vector aus Vector“ gespeichert. Außerdem werden Bonuskisten als Teil des Levels gewertet, da diese ebenfalls undurchdringliche und begehbare Kollisionsobjekte darstellen.

4 Laden und Speichern der Leveldaten und Levelobjekte.

4.1 Objekte

Alle Levelobjekte sind Instanzen der Klasse Figur oder einer ihrer Unterklassen. Die Klasse Figur bietet mit der Methode `alsString` eine Möglichkeit an, Eine Figur mit all ihren momentanen Eigenschaften (Typ, Animation, Position etc.), Steuerfunktionen und Zusatzparametern in Textform umzuwandeln. Das Gegenstück dazu ist die statische Methode `ausString`, welche aus einem so erzeugten String eine Instanz von Figur (oder einer passenden Unterklasse) generiert.

Die beiden statischen Hilfunktionen `speichereListeAlsDatei` und `ladeListAusDatei` führen diesen Vorgang jeweils für eine komplette `std::list<Figur*>` durch. Sie werden vom Leveleditor respektive beim Laden eines neuen Levels benutzt.

4.2 Level

Jedem Tile-Typ wird ein Integer-Wert zugewiesen. Diese Werte werden zusammen mit der Höhe und Breite des Levels aus einer Datei gelesen bzw. in diese geschrieben.

4.3 Parameter

Parameter stellen eine Methode dar, neuen Spielobjekten auch neue Attribute anzubieten, ohne das Levelformat ändern zu müssen. Diese Parameter werden vom Leveleditor oder von Hand hinzugefügt und intern einfach als String gespeichert. Die Methode `Figur->setParameter` wertet diese Parameter aus. Unbekannte Parametern werden ignoriert aber trotzdem an eine interne Liste angehängt, so dass sie einen Laden-Speichern-Zyklus (z.B. in einer alten Version des Leveleditors) immer schadlos überstehen.

5 Manager

Manager erleichtern dem Entwickler die Arbeit. Sie sorgen dafür, dass Ressourcen nur einmal geladen werden, beliebig oft verwendet werden können und abschließend „in einem Rutsch“ wieder freigegeben werden können. Die Freigabe erfolgt, sobald der Destruktor des Managers ausgelöst wird. Dies kann also implizit am Ende eines Blocks, bei der Zerstörung eines ihn enthaltenden Objekts oder, bei Zeigern, ausdrücklich durch *delete* erfolgen. So ist es z.B. möglich, Manager zu erstellen, alle Ressourcen eines Levels zu laden und am Ende des Levels alle Manager zu zerstören und sich zwischendurch keine Gedanken über Speicherverwaltung machen zu müssen.

Es gibt momentan zwei Typen von Manager: SpriteManager und SoundManager.

5.1 SpriteManager

Der SpriteManager verwaltet (man höre und staune) eine Ansammlung von Sprites. Diese werden anhand des Dateinamens ihrer Grafikdatei verwaltet. Mit der Methode `createSprite` kann ein Sprite geladen werden. Der SpriteManager achtet darauf, dass jede Datei nur einmal geladen wird. Alle Aufrufe dieser Methode mit demselben Dateinamen liefern also einen Zeiger auf dieselbe Instanz von Sprite.

Eine bisher nur über den Bitmap-Font genutzte Methode ist `erzeugeFont`. Diese zerlegt eine Grafikdatei in 16×16 gleich große Einzelsprites und liefert diese zurück als `std::vector<Sprite*>`.

5.2 SoundManager

Der SoundManager verwaltet Spielgeräusche und Hintergrundmusik. Über die Methoden `spieleSound` und `spieleSoundtrack` können beliebige, von der SDL_mixer - Bibliothek unterstütze, Audiodateien abgespielt werden. Die Hintergrundmusik kann über `stoppeSoundtrack` wieder abgeschaltet werden.

Die beiden Methoden `ladeSound` und `ladeSoundtrack` dienen dazu, Audiodateien bereits vor dem Abspielen in den Speicher zu laden. Dies dient ausschließlich der Performanceoptimierung. Ist eine Audiodatei beim Aufruf von `spieleSound` oder `spieleSoundtrack` noch nicht im Speicher, wird sie automatisch nachgeladen.

Zur Vereinfachung enthält die Klasse SoundManager bereits einige statische Konstanten mit den Dateinamen von Standard-Sounds, ihre Verwendung ist aber optional.

6 Fonts

Die Textausgabe im Spiel erfolgt über Bitmapfonts mit fester Zeichenbreite. Diese werden aus einer Grafikdatei mit allen ASCII-Zeichen erzeugt, welche einfach in einen `std::vector` aus 256 Sprites zerlegt wird. Dadurch ist der ASCII-Code eines Zeichens gleichzeitig der Index der Grafik innerhalb dieses Vectors und die Textausgaberoutinen fallen entsprechend simpel aus.

7 Datenformate

Das Spiel verwendet neben den üblichen Grafik- und Audiodateiformaten auch einige eigene Formate zum Laden und Speichern von Daten.

7.1 Leveldaten

```
Level  ::= Hoehe newline
           Breite newline
           Tiles

Hoehe  ::= Integer
Breite ::= Integer
Tiles  ::= Tile | Tile " "  Tiles | newline
Tile   ::= Integer
```

7.2 Levelobjekte

```
Figur           ::= "FIG:" Typ newline
                    "X: " XPos newline
                    "Y: " YPOS newline
                    "ANI: " Animationsdatei newline
                    STFListe newline
                    PARListe newline

Typ             ::= "Standard" | "Ring" | "Stachel" | "Kiste"
                    | "Herzkiste"
```

```
XPos            ::= Integer
YPos            ::= Integer
Animationsdatei ::= Dateiname
STFListe        ::= Steuerfunktion newline STFListe | Steuerfunktion
                    | {Leer}
Steuerfunktion  ::= "STF: " STFname STFParliste
STFname         ::= "LinksrechtsBewegung" | "Gegnersprung"
STFParliste     ::= Integer " " STFParliste | Integer | {Leer}
PARListe        ::= Parameter newline PARListe | Parameter | {Leer}
Parameter       ::= "PAR: " PARname " " PARvalue
PARname         ::= "Schaden" | "Gravitation"
PARvalue        ::= Integer
```