JMPG
Java Multiplayer Game Framework

Projektarbeit
Praktikum Softwaretechnik 2
Hochschule der Medien Stuttgart
WS 05/06

von
Benjamin May [13458]
und
Thomas Reimann [13377]

Betreuender Dozent: Prof. Dr. Jens-Uwe Hahn

Inhalt

1 Einleitung

1.1 Projektziel

Grundsätzliches Ziel dieses Studienprojekts war es, Erfahrungen bei der Entwicklung eines komplexen Frameworks für Spiele, das flexibel eingesetzt werden kann, zu sammeln.

Insbesondere sollte eine einfache Mehrspieler-Schnittstelle geschaffen werden, die es einem Spieleentwickler ermöglicht, schnell und unkompliziert netzwerkfähige Spiele zu entwickeln, ohne sich direkt mit den Problemen der Netzwerkprogrammierung auseinander setzen zu müssen.

Aufgrund der zeitlichen Begrenzung für ein Studien-Projekt (3 1/2 Monate) haben wir uns entschieden, uns auf die Unterstützung von 2D-Spiele zu beschränken, die Multiplayer-Schnittstelle aber so aufzubauen, dass sie auch für 3D-Spiele angepasst werden kann.

Parallel zum Framework wurde ein darauf basierendes Multiplayer-Spiel entwickelt, um die Funktionalitäten zu testen und Details des Frameworks zu verfeinern.

1.2 Einführung Spieleentwicklung

Spiele unterscheiden sich in ihrem Design stark von typischen Fenster-Anwendungen.

AWT- oder Swing-Anwendungen verwenden Standard-UI-Elemente, die meist an das *Look & Feel* des Betriebssystems angepasst sind und basieren auf einer Event orientierten Architektur. Fensterinhalte werden gezeichnet, wenn sich Daten geändert haben – z.B. durch Benutzereingaben oder nach einer Berechnung. Zeitliche Verzögerungen bei der Darstellung sind vertretbar und werden vom Anwender akzeptiert.

Spiele dagegen besitzen meist aufwendige grafische Oberflächen, die individuell gestaltet sind und keine Standardkomponenten verwenden. Sie enthalten oft viele animierte Elemente. Besonders bei Echtzeitspielen sind schnelle Reaktionszeiten auf Benutzereingaben (z.B. Steuerung einer Spielfigur) von hoher Bedeutung. Der Spieler (Anwender) erwartet sofortiges Feedback auf seine Befehle. Wenn er z.B. seine Spielfigur nach links steuert um einem Hindernis auszuweichen, muss sie sofort reagieren – dabei können Bruchteile von Sekunden entscheidend sein. Ein weiterer wichtiger Punkt ist die Audioausgabe – sowohl Hintergrundmusik als auch qualitativ hochwertige Echtzeit-Soundeffekte sind wichtig um das Spielerlebnis abzurunden.

Hieraus lassen sich die Basisanforderungen von Spielen ableiten:

- Komplexe Grafik mit Animationen
- Geschwindigkeit
- Schnelle Reaktion auf Benutzereingaben
- Echtzeit Sound

1.3 Planung

Für die Planung des Projektes war es zunächst wichtig, allgemeine, für die Spieleentwicklung wichtige Funktionalitäten zu finden und anhand dieser, eine Grobstruktur für den Aufbau des Frameworks abzuleiten.

Die Umsetzung des Projektes sollte in zwei großen Phasen stattfinden:

Die in Kapitel 1.2 aufgeführten Basisanforderungen bildeten die Eckpfeiler der zu implementierenden Funktionalitäten in der ersten Projektphase. Parallel zur Entwicklung sollte zum Test die Singleplayer-Version von Gravity realisiert werden. Das Testspiel sollte das Auffinden von Funktionalitäten erleichtern, die während der Konzeption noch nicht berücksichtigt wurden, aber sinnvoll für das Framework erschienen.

In der zweiten Phase sollte die Netzwerkschnittstelle des Frameworks realisiert werden. Parallel zu dieser Erweiterung sollte Gravity zur Multiplayer-Version umgebaut werden und die bis dahin implementierten Funktionalitäten des Frameworks angepasst werden.

1.4 Entwicklungsumgebung

Die Entwicklung des Frameworks und des Demo-Spiels basiert komplett und alleine auf dem Java Standard SDK 1.5. und wurde mit Eclipse 3.1 als Entwicklungsumgebung auf separaten Rechnern durchgeführt. Für die Synchronisation der Projektdaten kam ein CVS-Repository auf einem zentralen Server zu Einsatz. Die Erstellung der Grafiken erfolgte in Photoshop wobei einige Spielelemente zuvor in LightWave aus als 3D-Modell erstellt und gerendert wurden.

2 Java Multiplayer Game Framework

2.1 JMPG Anforderungen

Wie oben beschrieben, liegen die Hauptanforderungen von Spielen in der Grafik.

Ziel des Projekts war es, ein Framework zu erstellen, das die bei der Spieleentwicklung immer wieder auftauchenden Probleme vereinfacht - also Funktionalitäten und Komponenten zur Verfügung stellt, die dem Spieleprogrammierer diese Aufgaben abnehmen. Dazu gehören:

- Schnelle Grafik mit *double buffering*
- Unterstützung von Sprites, Animationen und Ebenen, um Sprites zu gruppieren
- Echtzeit Sound, viele Sounds gleichzeitig
- Laden von Bildern, Animationen und Sound
- Kollisionserkennung
- Verarbeitung von Benutzereingaben auf spieleverträgliche Weise
- Einfache Netzwerkschnittstelle

2.2 Grundlagen

2.2.1 Game loop

Wegen den oben erwähnten Anforderungen von Spielen, eignet sich eine rein Event basierte Architektur schlecht als Basis für Spiele. Besonders wegen den aufwendigen grafischen Anforderungen (Animation, viele bewegte Objekte)

Basis eines jeden Spiels ist die so genannte *game loop*. In ihr wird in schneller Abfolge mit mindestens 12, besser aber mit mehr als 20 Bildern pro Sekunde (um die flüssige Wiedergabe von Animationen zu ermöglichen), der Spielzustand (*game state*) aktualisiert und angezeigt. Außerdem werden zu Beginn eines jeden Durchlaufs die aktuellen Benutzereingaben verarbeitet, die in die Berechnung des *game states* einfließen.

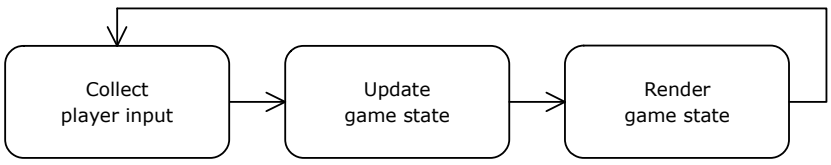


Abb. 2.1 Die game loop

Grundsätzlich gibt es drei Ansätze für das Timing der *game loop*:

Fixed frame rate

Das Spiel wird zeitlich so gesteuert, dass eine feste Bildrate erreicht wird. Alle Animationen und Berechnungen von Bewegungen erfolgen anhand dieser festen Bildrate, d.h. sie kennen

als zeitliche Komponente nur die Einzelbilder (so genannte *ticks*): so schreitet z.B. eine Animation bei jedem *tick* um ein Bild fort. Berechnungen sind einfach zu implementieren, da alle Bewegungen in einem festen Zeitintervall ablaufen und somit Zustandsänderungen im Spiel immer in einem festen Raster erfolgen.

Es treten jedoch Probleme auf, wenn die verfügbare Prozessorleistung nicht ausreicht, um die Änderungen am Spielzustand innerhalb der vorgegebenen Zeitspanne zu berechen, bzw. die Spielgrafik zu *rendern*: Dann wird das gesamte Spiel langsamer.

Variable frame rate

Das Spiel läuft mit der höchsten Geschwindigkeit, die mit der verfügbaren Prozessorleistung möglich ist. Alle Animationen und Berechnungen erfolgen abhängig von der tatsächlich vergangenen Zeit. Da die Dauer eines *Frames* im Normalfall nicht vor seiner Berechnung vorhersagbar ist, wird als zugrunde liegendes Zeitintervall die Dauer des letzten *Frames* verwendet. Nachteil dieses Vorgehens ist, dass das Spiel unnötig viel Prozessorzeit beansprucht, wenn es mit einer höheren Bildrate läuft, als sie vom Spieler wahrgenommen werden kann.

Außerdem kann diese Methode dazu führen, dass die Darstellung nicht flüssig erscheint, wenn die Zeiten für die Berechnung von *Frames* stark variieren.

Limited variable frame rate

Das Spiel basiert auf Echtzeitberechnungen, wird aber auf eine feste Bildrate gedrosselt. D.h. die Aktualisierungen erfolgen, analog der „variable-frame-rate“-Methode, auf der wirklich verstrichenen Zeit des letzten Frames, am Ende jedes Frames wird der Prozess aber so lange pausiert, dass eine feste Bildrate erreicht wird. Sollte es vorkommen, dass diese Zeit nicht für die Aktualisierung des Spielzustands ausreicht, wird ohne Verzögerung sofort das nächste Frame begonnen.

Dieses Vorgehen hat den Vorteil, dass das Spiel nicht (unnötiger Weise) die gesamte Prozessorleistung in Anspruch nimmt.

Alternativ können die Aktualisierung und das Zeichnen des Spielzustands asynchron in getrennten Threads vorgenommen werden, sodass die Berechnungen mit einer höheren zeitlichen Auflösung (und dadurch genauer) und die Darstellung mit einer konstanten Bildrate, die für flüssige Animationen ausreicht, erfolgen können. Diese Methode wird bei Spielen selten verwendet, da dort die Genauigkeitsanforderungen der Berechnungen meist keinen hohen Ansprüchen genügen müssen, ein hoher Aufwand für die Synchronisation von Darstellung und Berechnungen entsteht dennoch.

In der Server-Anwendung von Gravity verwenden wir eine vereinfachte Form dieser Methode um eine höhere Genauigkeit bei der Physikberechnung und Kollisionserkennung zu erreichen. Näheres hierzu in Kap. 3.

2.2.2 Double Buffering

Um eine einwandfreie Wiedergabe von Animationen und bewegten Objekten zu erreichen, genügt das einfache Zeichnen des Spielscreens nicht – es müssen aus hardware-technischen Gründen weitergehende Methoden angewendet werden.

Die Anzeige eines Bildes auf dem Monitor erfolgt periodisch, abhängig von der Bildwiederholrate des Monitors (z.B. 60 Hz). Ändert sich der Inhalt des Bildes (z.B. durch bewegte Objekte oder Animationen) während es gerade dargestellt wird, flackert das Bild (*tearing*).

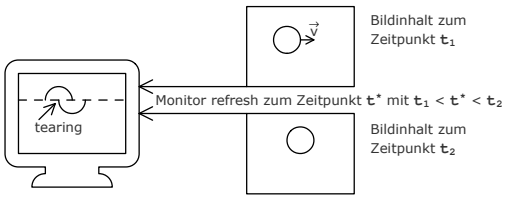


Abb. 2.2 Tearing

Um das Flackern zu verhindern, werden zwei getrennte Puffer verwendet: Ein Vordergrundpuffer (*display buffer*), der den aktuellen Bildschirminhalt enthält und ein Hintergrundpuffer (*back buffer*), in den der Inhalt des nächsten Bildes gezeichnet wird.

Ist das Zeichnen in den Hintergrundpuffer abgeschlossen, wird beim nächsten *Monitorrefresh* dieser als *display buffer* gesetzt – der *display buffer* wird nun als *back buffer* verwendet, so dass in ihn wieder gezeichnet werden kann. Diesen Vorgang nennt man *page flipping*.

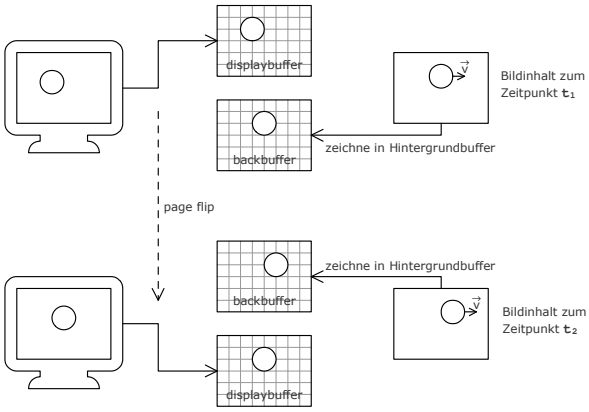


Abb. 2.3 *Page flipping* beim *double buffering*

Das *Java AWT*-Framework stellt für solche Grafikanforderungen die Klasse `BufferStrategy` bereit. Sie unterstützt verschiedene Arten von *buffering*: Das in **JMPG** verwendete *double buffering*, *tripple buffering* (wie *double buffering*, aber mit einem zusätzlichen Zwischenpuffer), sowie *stereo buffering* (für Ausgabe von stereoskopischen Darstellungen).

2.2.1 Optimierungen

Die Performance spielt bei der Spieleentwicklung eine große Rolle. Bei der Entwicklung haben wir versucht, dort wo es im Design des Frameworks möglich war, einige Optimierungsansätze mit einfließen zu lassen.

Typisch für Spiele ist die Verwendung von *lookup tables*: Oft benötigte Berechnungen werden im Voraus durchgeführt und deren Ergebnisse in Listen gespeichert – auf die vorausberechneten Werte kann dann sehr schnell zugegriffen werden. Da die Zugriffe auf Arrays über Integerwerte erfolgen müssen, bringt diese Methode natürlich einen Genauigkeitsverlust mit sich, der aber bei den geringen Genauigkeitsanforderungen von Spielen nicht weiter tragisch ist. Es muss zwischen gewünschter Genauigkeit und dem durch mehr Werte steigenden Speicherplatz der *lookup tables* abgewogen werden. Die Klasse `GameMath` stellt Sinus- und Kosinusfunktionen als *lookup tables* mit einer Länge von 4096 Werten bereit. Als Winkeleinheit werden in **JMPG** deswegen Integerwerte von 0 (0°) bis 4096 (360°) verwendet (das entspricht einer Auslösung von 0,0879°).

Große Performanceverluste treten bei der Erzeugung von Objekten auf. Da die Hauptschleife des Spiel in sehr engen Zeitabständen durchlaufen wird, sollte dort auf die Erzeugung von Objekten weitgehend verzichtet werden - insbesondere wenn es sich um temporäre Objekte handelt, die nur innerhalb eines Durchlaufs verwendet werden. Da sie nicht sofort aus dem Speicher entfernt werden, wird, wenn entsprechend viel Speicher belegt ist, die Java *Garbage Collection* ausgelöst und unter Umständen die Ausführung des Spiels blockiert, was zu sehr störenden Unterbrechungen im Spielfluss führen kann.

Dies versuchten wir bei der Implementierung zu berücksichtigen. Zum Beispiel verwenden wir für Positionsdaten keine Vektor-Klasse, die die beiden Komponenten kapselt, sondern jede Klasse besitzt die X- und Y-Koordinaten als zwei `float`-Werte. Würden Vektor-Objekte verwendet werden, müsste als Rückgabewert eine Kopie dieses Vektors (also ein neues Objekt) erzeugt werden. Alternativ könnte mit der Referenz des Vektors gearbeitet werden, was aber nicht zu empfehlen ist, da dann keine Entkopplung vom Ursprungsobjekt stattfindet und sich somit leicht unkontrollierbare Fehler einschleichen können.

2.3 Komponenten der 2D-API von JMPG

2.3.1 Übersicht JMPG-Packages

`org.jmpg.core`	Basiskomponenten
`org.jmpg.net`	Netzwerkkomponenten für Spiele mit Multiplayer-Unterstützung
`org.jmpg.resource`	Komponenten für das Ressource Management (Laden von Bildern, Animationen und Sound)
`org.jmpg.sound`	Komponenten für die Soundwiedergabe
`org.jmpg.input`	Komponenten für die Benutzereingabe
`org.jmpg.ui`	Alle Komponenten für die Grafik eines Spiels (Animationen, Sprites, Ebenen)
`org.jmpg.util`	Hilfsklassen
`org.jmpg.xml`	Einfacher XML-Parser

2.3.2 Basiskomponenten

2.3.2.1 GameFrame

Die Klasse `GameFrame` übernimmt die Grafikausgabe des Spiels. Es werden Fenster- und Vollbildmodus unterstützt, wobei eine flackerfreie Darstellung durch *double buffering* gewährleistet wird. Für das *double buffering* wird die AWT-Klasse `BufferStrategy` verwendet.

Im Vollbildmodus erzeugt `GameFrame` ein `JFrame`, dass als `FullScreenWindow` der aktiven `GraphicsDevice` gesetzt wird (`device.setFullScreenWindow()`). Die `BufferStrategy` wird im `FullScreenWindow` erzeugt.

Im Fenstermodus wird ein `JFrame` (`javax.swing.JFrame`) mit eingebetteter `Canvas` (`java.awt.Canvas`) verwendet. Die `BufferStrategy` wird im `Canvas` erzeugt.

Beim `GameFrame` können ein `GameFrameListener` und ein `InputListener` registriert werden. An den `GameFrameListener` werden `WindowEvents` weiter geleitet, sodass Vorgänge wie das Schließen oder das Minimieren des Fensters behandelt werden können.

Der `InputListener` dient dem Abfangen von Eingabe-Ereignissen. Dieses Interface wird durch den `InputManager` implementiert, der eine spieletypische Behandlung von Benutzereingaben ermöglicht (siehe *Kap. 2.3.4*).

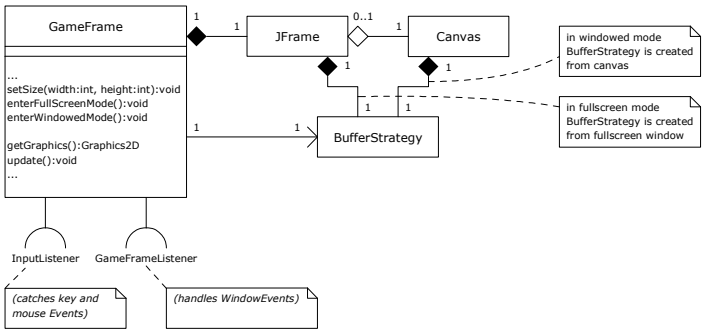


Abb. 2.4 GameFrame

Das `GameFrame` wird mit einer bestimmten Auflösung initialisiert und kann dann entweder im Fenster- oder Vollbildmodus angezeigt werden. Ein Umschalten des Modus während des Betriebs ist möglich, hier muss aber beachtet werden, dass das Umschalten in den Vollbildmodus, also das Wechseln der Bildschirmauflösung, recht lange Dauern kann (teilweise über 5 Sekunden) und diese Zeit in Abhängigkeit von der verwendeten Hardware (Grafikkarte und Monitor) stark schwankt.

Einige Methoden von `GameFrame` sind im aktuellen Zustand des **JMPG** Frameworks noch nicht, oder nur teilweise implementiert. Momentan müssen die Auflösung und Bildwiederholrate für den Vollbildmodus korrekt angegeben werden, sodass diese von der aktuellen Hardware unterstützt werden. Eine automatische Erkennung von möglichen Auflösungen und Wiederholraten ist vorbereitet, aber noch nicht voll funktionsfähig. Ebenfalls vorgesehen ist die Möglichkeit, die Vollbildauflösung unabhängig von der Spielauflösung zu setzen. Dies ist sinnvoll, wenn die Spielauflösung nicht zur Displayauflösung kompatibel ist. Insbesondere bei TFT-Displays (die eine feste Pixel-Auflösung besitzen) ist ein Wechsel in eine nicht kompatible Auflösung nicht ohne Qualitätsverluste möglich. Hier ist angedacht, dass die optimale Auflösung verwendet werden kann, wenn das Spiel eine geringere Auflösung verwendet – die Spielfläche würde dann zentriert, nicht bildschirmfüllend angezeigt werden

2.3.2.2 GameTimer

Die Auflösung des Standard-Systemzeitgebers der *Java VM* (`System.currentTimeMillis()`) ist unter Windows relativ gering: unter Windows 95/98/Me sind dies ca. 55 ms, unter Windows NT/2000/XP ca. 10 – 15 ms. Mit einer so geringen Genauigkeit kann keine exakte, zeitabhängige Berechnung durchgeführt werden. Ab *Java 1.5* stellt die Klasse `System` eine genauere Methode für die Zeitmessung zur Verfügung: `Systme.nanoTime()`. Sie liefert eine fast Nanosekunden genaue Zeit zurück. Der Startpunkt der Zeitmessung ist, anders als bei `System.currentTimeMillis()`, die die vergangenen Millisekunden seit Mitternacht des 01.01.1970 zurückliefert, rein virtuell ist. Die Methode kann also nur zu Messung von relativ vergangener Zeit verwendet werden (Differenz von zwei Zeitwerten).

Die Klasse `GameTimer` kapselt diese Genauigkeit und stellt sie umgerechnet in (die gewohnten und an Genauigkeit ausreichenden) Millisekunden zur Verfügung. Speziell zur Messung von Intervallen innerhalb der *game loop* steht die Methode `timeElapsed()` zur Verfügung, die immer die, zwischen zwei Aufrufen vergangene Zeit zurückliefert. Außerdem übernimmt `GameTimer` die Berechnung der Wartezeit am Ende der *game loop*. Diese ist

nötig, um feste Bildraten zur erreichen. Hierzu kann über `setMinFrameTime()` die Mindestdauer eines *Frames* gesetzt werden – über `getFrameWaitTime()` kann dann die Zeit abgefragt werden, die am Ende eines *Frames* pausiert werden muss, um die gewünschte Gesamtdauer des aktuellen *Frames* zu erreichen.

2.3.2.3 GameScreen

Um den Aufbau und die Funktionsweise der Spielbasisklasse `Game` verstehen zu können, sprechen wir an dieser Stelle vorab die UI-Klasse `GameScreen` kurz an:

Die Klasse `GameScreen` ist die Basis des **JMPG**-UIs. Alle grafischen Elemente (Ebenen, Bilder & Sprites) des Spielinterfaces werden zu einem Screen zusammengefasst, so dass einfach zwischen verschiedenen Screens umgeschaltet werden kann (z.B. zwischen Menü-Seiten oder verschiedenen Ansichten im Spiel). `GameScreen` delegiert die `update()`- und `render()`-Methoden an die enthaltenen `Layer`.

Das UI von **JMPG** und die darin enthaltenen Klassen werden in Kapitel 2.3.3 (Seite 9) genauer erläutert.

2.3.2.4 Game

Die Basis für alle **JMPG**-Spiele bildet die Klasse `Game`, die die Hauptschleife des Spiels (*game loop*) enthält und über ein `GameFrame` die Grafikausgabe vornimmt. Sie stellt einen globalen `ResourceManager`, einen Standard-`InputManager` und einen `GameTimer` bereit.

Prinzipiell ist `Game` ein einfacher Thread, der eine Schleife enthält, in der der *game state* aktualisiert und gerendert wird, sowie am Ende des Durchlaufs nach Bedarf die Soll-Bildrate eingehalten wird.

Die Darstellung von Bildschirminhalten erfolgt über die oben kurz erwähnte Klasse `GameScreen`. `Game` besitzt immer einen aktuellen Screen, der momentan gerendert wird. Dieser kann über `showScreen()` während der laufenden *game loop* gewechselt werden, wobei das Umschalten auf den neuen Screen immer zu Beginn des nächsten Durchlaufs der *game loop* erfolgt. Wird der `GameScreen` gewechselt, werden beim alten Screen die Methode `exit()` und beim neu angezeigten Screen die Methode `enter()` aufgerufen. Die `GameScreens` werden also über den Wechsel benachrichtigt, sodass dort entsprechende Initialisierungen vorgenommen werden können.

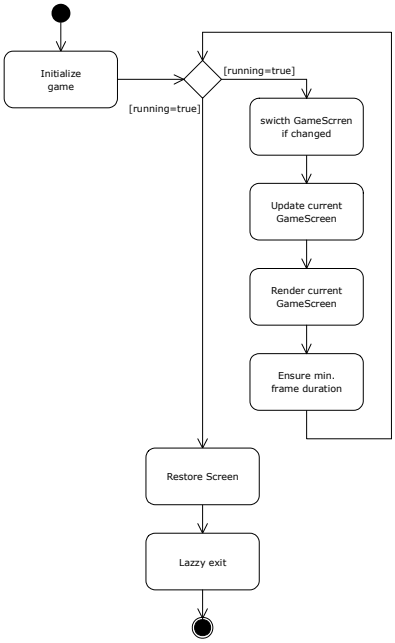


Abb. 2.5 Basisklasse Game

2.3.3 User Interface (UI)

Ein Spiel lebt von seiner Grafik – es besitzt viele animierte, meist bewegte Objekte, sowie statische Elemente (wie Hintergrundbilder, Rahmen etc.).

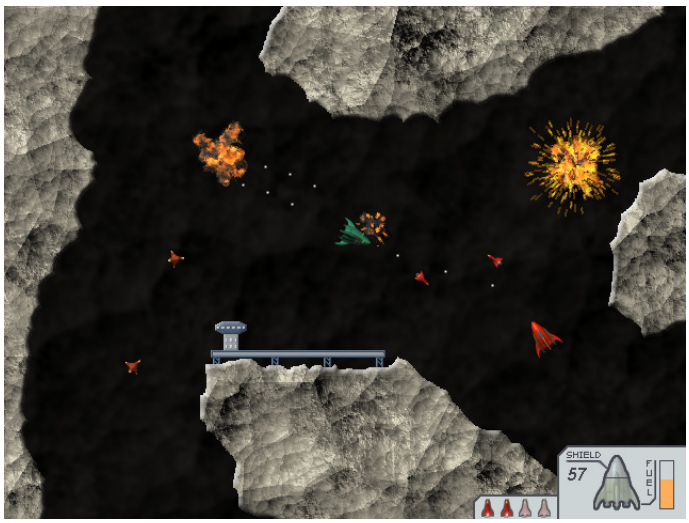


Abb. 2.6 Screenshot aus dem Spiel Gravity

Abb. 2.6 zeigt den Ausschnitt eines Screenshots des auf **JMPG** basierenden Spiel Gravity. Es sind viele animierte, bewegte Objekte (Schiffe, Raketen, Explosionen, …), sowie nicht animierte Bilder (Hintergrund, Level, HUD) enthalten.

JMPG stellt mehrere Klassen zur Verfügung, mit deren Hilfe Animationen, bewegte Objekte (*Sprites*) und Bilder erstellt und verwaltet werden können. Die Elemente können in Ebenen und Ebenengruppen verwaltet werden. Unterste Ebene der hierarchischen Struktur der Grafikelemente ist immer der `GameScreen`, der einem `Game`-Objekt zur Anzeige übergeben werden kann und `update()`- und `render()`-Methoden an seine Kinder delegiert.

Spielobjekte und andere veränderliche grafische Elemente werden durch Sprites repräsentiert. Sie können unabhängig positioniert und bewegt werden, sowie animiert sein.

Um Sprites zu gruppieren und die Reihenfolge des Zeichnens auf den Screen beeinflussen zu können werden sie in Ebenen (`SpriteLayer`) gruppiert. Jedes Sprite besitzt eine Animation.

Für größere statische Bilder (z.B. Hintergrund, transparenter Vordergrund), die nicht animiert sind, steht der `ImageLayer` zur Verfügung.

Mehrere Ebenen können in Gruppen zusammengefasst werden (`LayerGroup`).

Die hierarchische Organisation der UI-Elemente dient mehreren Zwecken:

- die in einer Ebene enthaltenen Elemente (Sprites & Ebene) werden relativ zu deren Koordinatensystem positioniert, d.h. sie können als Gruppe auf dem *Screen* platziert und bewegt werden
- alle, in einer Ebene enthalten Elemente können als Gruppe sichtbar/unsichtbar und aktiv/inaktiv gesetzt werden
- die Reihenfolge der Darstellung von Elementgruppen kann beeinflusst werden (später hinzugefügte Ebenen/Sprites werden über andere gezeichnet)

Nachstehend wird die Organisation der Ebenen in Gravity kurz erläutert:

Die unterste Ebene enthält den Hintergrund des Levels – er wird zuerst gerendert d.h. alle anderen Elemente überdecken diese Ebene

Abb. 2.7 Ebene 1 – Level-Hintergund

Die nächste Ebenen enthält Sprites, die Geschosse (Raketen, Minen usw.) und Levelelemente (z.B. Stationen) darstellen.

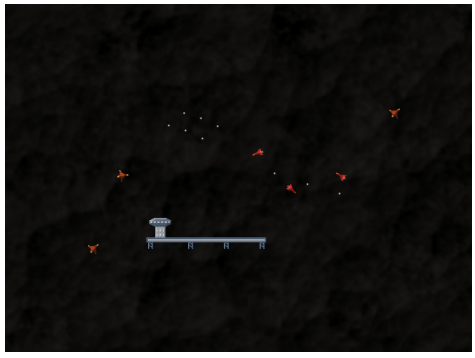


Abb. 2.8 Ebene 2 - Hintergrundsprites

Die Levelstruktur wird als dritte Ebene hinzugefügt. Sie wird über die Hintergrundsprites aus der zweiten Ebene gezeichnet.

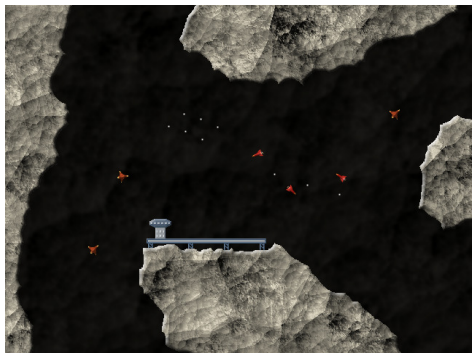


Abb. 2.9 Ebene 3 - Levelebene

Für die Spielerschiffe existiert eine eigene Sprite-Ebene.

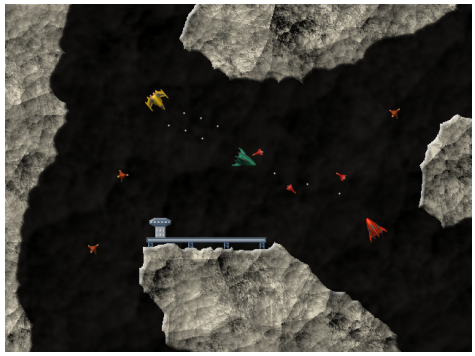


Abb. 2.10 UI-Beispiel 4 - Spielerebene

Explosionen sollen über die Schiffe und die Levelstruktur gerendert werden und werden deswegen in der letzten Ebene mit In-Game-Grafiken gruppiert.

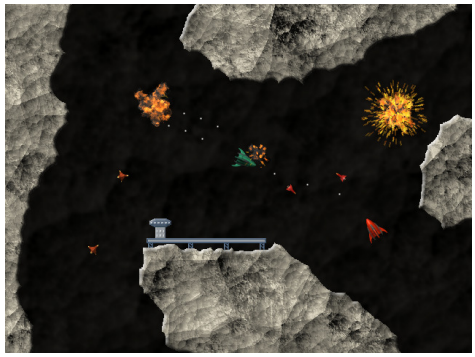


Abb. 2.11 Ebene 5 - Vordergrundsprites

Zuletzt werden die statischen Elemente des HUDs (*Head-Up-Display*) eingefügt.

Abb. 2.12 Ebene 6 - HUD-Ebene

2.3.3.1 Koordinatensystem

Bevor wir auf die einzelnen UI-Komponenten eingehen, wollen wir kurz die Art und Weise, wie in **JMPG** auf den Screen gezeichnet und welches Koordinatensystem verwendet wird erläutern.

Grafische Elemente werden meist durch (in der Regel transparente) Bilder repräsentiert, oder es werden direkt die Grundfunktionen der Graphics2D-Schittstelle von *AWT* (Linien, Rechtecke, etc.) verwendet.

Der Ursprung des Koordinatensystems liegt in der linken oberen Ecke des Bildschirms. Die positive `X`-Komponente zeigt nach links, die positive `Y`-Komponente nach unten.

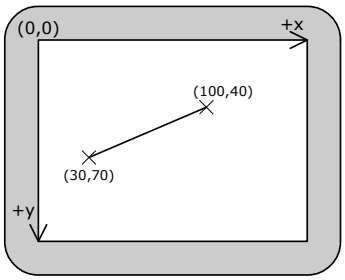


Abb. 2.13 Bildschirmkoordinatensystem

Gezeichnet wird auf das vom oben beschriebenen `GameFrame` zur Verfügung gestellte `Graphics2D`-Interface.

Ein Winkel wird im **JMPG-Framework** positiv im Uhrzeigersinn gemessen, wobei `0°` vertikal nach oben ausgerichtet ist. Aufgrund der aufwändigen Berechnung von trigonometrischen Funktionen werden für die Winkelmessung keine Fließkommawerte verwendet, sondern auf vorberechnete Sinus- und Kosinuswerte im Bereich von 0 bis 4095 zurückgegriffen (ein sog. *lookup table*). Diese Genauigkeit reicht für Spiele vollkommen aus und beschleunigt die Bestimmung von Winkeln immens, da sie nicht berechnet, sondern nur in einer Liste nachgeschlagen werden müssen.

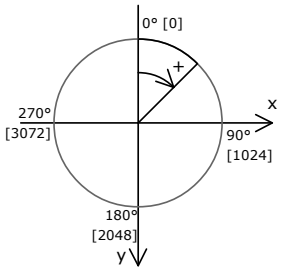


Abb. 2.14 Winkelmessung in

Die Klasse `GameMath` enthält mehrere Methoden, die die Bestimmung und Umrechnung von Winkeln im **JMPG**-Koordinatensystem ermöglicht. Zum Beispiel liefert `getAngle(x1, y1, x2, y1)` den Winkel zwischen zwei Punkten.

2.3.3.2 Animationen

Prinzipiell können alle veränderlichen Bildfolgen als Animationen bezeichnet werden – z.B. ein sich bewegender Ball.

Im **JMPG**-Framework wird der Begriff Animation ausschließlich für Einzelbildfolgen, die mit einer bestimmten Geschwindigkeit wiedergegeben werden verwendet, nicht aber für die Bewegung von Spielobjekten (Sprites). Jedes Sprite besitzt eine Animation, deren aktuelles Bild auf dem Bildschirm dargestellt wird.

Eine Animation besteht aus einer Reihe von Einzelbildern (`AnimationFrames`), die für eine bestimmte Dauer angezeigt werden. Diese Dauer muss nicht für jedes Frame gleich sein, sondern kann variieren.

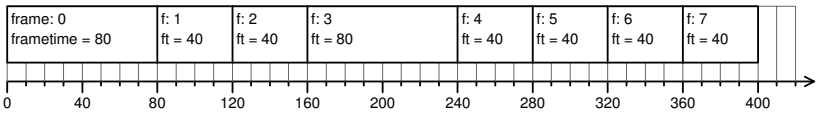
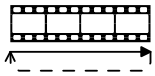
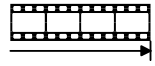
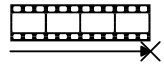
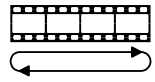
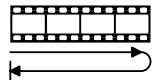


Abb. 2.15 Aufbau einer Animation mit Einzelbildern

Animationen können vorwärts und rückwärts abgespielt, sowie pausiert werden. Wie sich eine Animation verhält, wenn sie am Ende ihrer Bildsequenz angelangt ist wird über das `AnimEndBehavior` festgelegt. Standardmäßig wird die Animation nach einem Durchlauf von vorne fortgesetzt (`repeat`) und läuft in einer Endlosschleife. Mögliche Werte für das Ende-Verhalten sind in *Tabelle 2.1* aufgelistet.

Tabelle 2.1 Bedeutung der Werte von `AnimEndBehavior`

	`AnimEndBehavior`	**Bedeutung**
	`repeat`	Animation wird endlos wiederholt
	`stop`	Animation wird einmal abgespielt, dann gestoppt
	`hide`	Animation wird einmal abgespielt, dann nicht mehr angezeigt
	`pingPong`	Animation wird endlos vorwärts, rückwärts, vorwärts, rückwärts … abgespielt
	`reverseOnce`	Animation wird einmal vorwärts, dann zurück abgespielt

Basis für alle Animationen ist die abstrakte Klasse `Animation`, die eine Reihe von *Properties* für Animationen zur Verfügung stellt. Die Methode `getCurrentFrame()` muss so implementiert sein, dass sie das Bild des aktuellen `AnimationFrames` liefert.

Es stehen zwei Implementierungen von Einzelbildanimationen in **JMPG** zur Verfügung: `FrameBasedAnimation` und `TimeBasedAnimation`.

Die `FrameBasedAnimation` ist eine Animation, die bei jedem Aufruf der `update(timeElapsed)`-Methode genau ein *Frame* fortschreitet. Sie ist also unabhängig

von der verstrichenen Zeit. Sie eignet sich hauptsächlich für Animationen in Spielen mit fester Bildrate oder um Bildsequenzen für Sprites zu speichern – z.B. um rotierende Sprites darzustellen.

Die `TimeBasedAnimation` ist eine Zeit basierte Animation. Beim Aufruf von `update(timeElapsed)` schreitet die aktuelle Zeit um den verstrichenen Betrag fort – das aktuelle *Frame* ist abhängig von der Animationszeit und der Dauer der einzelnen Bilder. So kann eine feste Bildrate für die Animation erreicht werden, auch wenn das Spiel mit einer höheren und evtl. variablen Bildrate läuft. Sollte das Spiel mit einer geringeren Bildrate als die der Animation laufen, werden einzelne Bilder übersprungen.

Die einzelnen Bilder der Animation werden durch Objekte der Klasse `AnimationFrame` repräsentiert. Sie enthält das eigentliche Bild (`Image`) und dessen Anzeigedauer (in Millisekunden). Die Reihe der Einzelbilder wird nicht in der Klasse Animation, sondern in einer Collection `AnimFrameSequence` gespeichert, die Bestandteil jeder Animation ist. Diese Bildsequenz besitzt zusätzliche Informationen über die Gesamtdauer und die *frame rate* der Animation.

Diese zusätzliche Kapselung ist eingeführt worden, um eine flexible Mehrfachverwendung einer Animation zu ermöglichen:

Die Klasse `Animation` selber kennt nur die aktuelle Zeit und das aktuelle *Frame* der Animation. Soll im Spiel die exakt gleiche Animation mehrmals verwendet werden, gibt es mehrere Möglichkeiten:

- Das Zuweisen des gleichen `Animation`-Objekts zu mehreren Sprites auf dem Screen bewirkt, dass diese Animationen exakt synchron ablaufen, da alle Sprites die gleiche Instanz benutzen.
- Wird das `Animation`-Objekt geklont und die `AnimationSequence` beibehalten, können die Animationen asynchron, d.h. jede mit anderem Zeitindex abspielen. Das Ändern der Bildrate wirkt sich auf alle Animationen aus, da sie dieselbe Instanz der `AnimFrameSequence` benutzen.
- Wenn zusätzlich die `AnimationSequence` und die `AnimationFrames` geklont werden, sind sie vollständig unabhängig von einander – nur die in den `AnimationFrames` enthaltenen Bilder sind noch identische Instanzen.

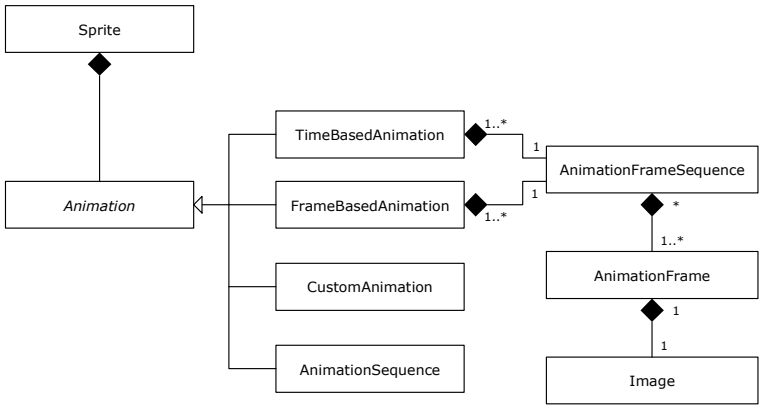


Abb. 2.16 Übersicht über die Animationsklassen

2.3.3.3 Sprites

Sprites repräsentieren Objekte im Spiel. Jedes Sprite besitzt eine Animation – bei nicht animierten Sprites ist dies eine Animation mit nur einem *Frame*.

Ein Sprite kann frei auf dem Bildschirm, bzw. auf der virtuellen Fläche der Ebene in der er sich befindet, positioniert werden. Die Position des Sprites bezeichnet nicht die linken oberen Ecke des Bildes der Animation, sondern dessen Mittel- oder Drehpunkte (*pivot point*). Er wird relativ zur linken oberen Ecke der Außenkante der Einzelbilder der Animation definiert. Die Koordinaten $\vec{v}'$, an denen ein Bild gezeichnet wird (linke obere Ecke), errechnet sich also durch Addition der Position des Sprites $\vec{v}$ und seines Pivot-Punkts $\vec{p}$ (siehe *Abb. 2.17*).

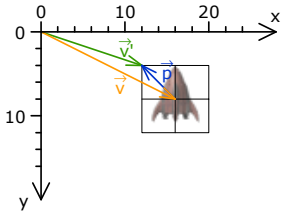


Abb. 2.17 Pivot-Punkt

In jedem Durchlauf der *game loop* werden bei jedem im aktuellen `GameScreen` enthaltenen Sprites zuerst die `update()`- und dann die `render()`-Methode aufgerufen. In `update()` wird die Animation aktualisiert, d.h. sie schreitet um den übergebenen Zeitbetrag fort. Diese Methode kann bei abgeleiteten Sprites überschrieben werden, um ein spezielles Verhalten des Sprites zu implementieren. In der `render()`-Methode werden das `Graphics2D`-Interface, sowie die relative Position bezüglich des Bildschirmkoordinatensystem, die durch Positionierung der übergeordneten Ebenen zustande kommt, übergeben.

Eine erweiterte Version von `Sprite` für bewegte Objekte ist die Klasse `MovingSprite`. Sie hält als zusätzliche *Properties* einen Geschwindigkeitsvektor (X- und Y-Komponente in Pixel/Sek.), sowie zusätzliche Hilfsmethoden um die Bewegungsrichtung (der normalisierte Geschwindigkeitsvektor) und die absolute Geschwindigkeit abzufragen und zu ändern. Die Position des Sprites wird innerhalb der `update()`-Methode abhängig vom Geschwindigkeitsvektor und der vergangenen Zeit aktualisiert.

Weitere spezialisierte Ableitungen von `Sprite` sind:

`TextSprite`

Dieser Sprite dient der Darstellung von Text (`String`). Ihm kann ein `Font` (Schriftart, -größe und -stil), sowie eine Farbe (`Color`) zugewiesen werden. Der Text kann wie jeder andere Sprite beliebig positioniert werden. Die Ausrichtung des Textes ist horizontal entweder links- oder rechtsbündig, oder zentriert (`H_ALIGN_LEFT`, `H_ALIGN_RIGHT`, `H_ALIGN_CENTER`).

`ButtonSprite`

Auch Menüelemente wie Buttons werden durch Sprites repräsentiert. In **JMPG** kann der Standardbutton drei Zustände annehmen: `enabled` (aktiv, auswählbar), `disabled` (inaktiv, nicht auswählbar) oder `selected` (ausgwählt). Diese Zustände werden durch getrennte Animationen repräsentiert. Die Klasse `ButtonSprite` besitzt deshalb für jeden Zustand eine eigene Animation. Die Sprite-Methode `getAnimation()` liefert die, dem Zustand entsprechende Animation zurück. Der Zustand des Buttons kann über `setEnable(boolean enabled)` und `setSelected(boolean selected)` geändert werden.

`MouseSprite`

Um einen eigenen (evtl. animierten) Mauszeiger im Spiel zu verwenden stellt **JMPG** die Klasse `MouseSprite` zur Verfügung. Dem `MouseSprite` wird ein `InputManager` zugewiesen – die Position des Sprites wird entsprechend der aktuellen Mauskoordinaten aktualisiert.

TextFieldSprite

Da in **JMPG** keine Standard-UI-Komponenten aus *AWT* oder *Swing* verwendet werden können, steht eine einfache Umsetzung eines Texteingabefeldes zur Verfügung. Die Klasse `TextFieldSprite` basiert auf dem in *Kap. 2.3.4* näher erläuterten `LineInputReader` und ermöglicht das Eingeben und Editieren einer Textzeile (mit einer festlegbaren Maximalzahl von Zeichen). Wenn das Textfeld aktiv ist (→ `setFocus(true)`), wird an die aktuelle Cursorposition eine Markierung gezeichnet und der Text kann editiert werden. Den aktuell enthaltenen Text liefert `getValue()` als `String` zurück.

2.3.3.4 Ebenen

Basisklasse für alle Ebenen ist die Klasse `Layer`. Ein `Layer` delegiert die `update()`- und `render()`-Aufrufe an alle, in ihm enthaltenen Elemente. Jede Ebene kann frei positioniert werden. Beim *Rendern* wird ihre aktuelle absolute Position (also die Position des Ursprungs des Ebenen-Koordinatensystems auf dem Bildschirm) an die enthaltenen Ebenen oder Sprites übergeben. Diese werden relativ zu diesen Koordinaten positioniert. Durch die Zusammenfassung von Sprite in eine Ebene können also mehrer Sprites als Gruppe bewegt werden.

Von der abstrakten Klasse `Layer` sind mehrer spezielle Ebenentypen abgeleitet, die unterschiedlichen Zwecken dienen:

LayerGroup

Die Klasse `LayerGroup` gruppiert mehrere Ebenen. Ebenen müssen in der Reihenfolge, in der sie gerendert werden sollen, hinzugefügt werden.

ImageLayer

Der `ImageLayer` repräsentiert ein einfaches, nicht animiertes Bild. Dieser Ebenentyp wird normalerweise für Hintergründe, Levelstrukturen oder statische UI-Komponenten verwendet.

SpriteLayer

Der `SpriteLayer` ist der Container für Sprites. Die Sprites müssen in der Reihenfolge, in der sie gerendert werden sollen, hinzugefügt werden.

BufferedSpriteLayer

Der `BufferedSpriteLayer` ist eine spezielle Version der Klasse `SpriteLayer`. Der große Unterschied besteht darin, dass die darin enthaltenen Sprites nicht direkt auf die Grafikausgabe (`Graphics2D`), sondern zunächst in einen Bildpuffer (ein `BufferedImage`), gezeichnet werden. Dieser wird dann separat gerendert. Dabei kann festgelegt werden, ob vor dem Zeichnen der Sprites der aktuelle Inhalt gelöscht wird, oder nicht. Diese Option kann für mehrere Zwecke eingesetzt werden: z.B. kann ein Sprite die Spur seiner Bewegung hinterlassen oder ein Sprite kann mehrmals positioniert und gerendert werden und sozusagen als Stempel verwendet werden, um Muster zu erzeugen.

MenuLayer

Der am höchsten spezialisierte `Layer` ist der `MenuLayer`. Er wird im `GameMenuScreen` eingesetzt und für einfache Menüs im Spiel verwendet.

2.3.3.5 GameScreen

Der `GameScreen` ist, wie in *Kap. 2.3.2.3* schon kurz erläutert, die Basisklasse des **JMPG**-UIs. Die Anzeige des aktuellen Screens wird von der Klasse `Game` verwaltet. Ein `GameScreen` verhält sich im Prinzip wie eine `LayerGroup`, mit dem Unterschied, das ein GameScreen

nicht frei positioniert werden kann – sein Ursprung liegt immer im Ursprung des Bildschirms. Es können mehrere `Layer` hinzugefügt werden. Die `update()`- und `render()`-Methodenaufrufe werden an die enthaltenen `Layer` delegiert.

Es existieren zwei speziellere Implementierungen von `GameScreen`:

`SimpleGameScreen`

Für einfache Spiele genügen meist eine Hintergrundebene und eine Ebene für die Sprites. Hier kann die Klasse `SimpleGameScreen` verwendet werden. Sie besitzt genau einen `ImageLayer` für den Hintergrund und einen `SpriteLayer` für die Spielobjekte. Zusätzlich steht noch eine `LayerGroup` für zusätzliche Vordergrundebenen zur Verfügung.

`GameMenuScreen`

Speziell für Gravity haben wir einen zusätzlichen `GameScreen`-Typ implementiert, der die schnelle Erstellung einfacher Menüs für das Spiel ermöglicht. Eine allgemeine **JMPG**-Version ist aus Zeitgründen nicht mehr implementiert worden. Die Funktionalität des `GameMenuScreen` wird in *Kap. 3.3.5* kurz erläutert.

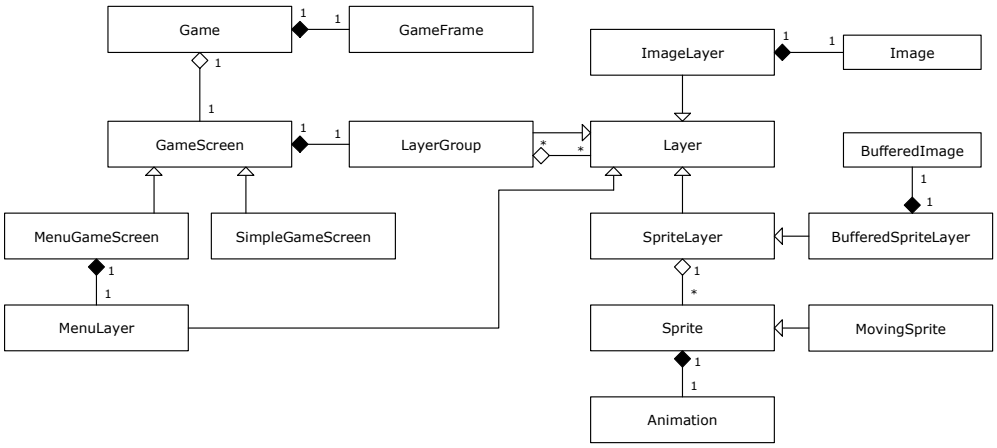


Abb. 2.18 Übersicht über die UI-Klassen von JMPG

2.3.4 Benutzereingabe

Zu Beginn eines jeden Durchlaufs in der *game loop* wird der aktuelle Zustand der Benutzereingabe abgefragt und der Spielzustand entsprechend aktualisiert. Aufgrund des Aufbaus eines Spiels (periodisches Aktualisieren des Spielzustands in der *game loop*), ist eine Ereignis basierte Architektur für die Verarbeitung von Eingaben wie sie AWT oder Swing zur Verfügung stellen ungeeignet. Der Zustand der gedrückten Tasten (und derer, die gedrückt wurden) soll genau einmal beim Eintreten in die *game loop* verarbeitet werden. Dabei ist es sinnvoll, Aktionen (oder Kommandos) auf unterschiedliche Tasten legen (mit der Möglichkeit, dass eine Aktion mehrere Tasten belegt), diese Tastenbelegung einfach ändern, sowie die Konfiguration laden und speichern zu können. Ebenso nützlich kann es sein, die Zahl der Tastendrücke während des letzten *game loop* -Durchlaufs abfragen zu können.

Diese Aufgaben übernimmt der `InputManager`. Ihm können `InputCommands` zugewiesen und bestimmten Tasten, Mausbuttons oder auch Mausachsen (unterschieden zwischen positiver und negativer Richtung) zugeordnet werden.

2.3.4.1 InputManager

Ein `InputCommand` steht für ein Kommando des Spielers wie z.B. Schuss oder Links- / Rechtsdrehung. Ist ein `InputCommand` über den `InputManager` einer (oder mehreren) Taste(n) zugewiesen, kann der Status über `isPressed()` (`true`, wenn die zugeordnete(n) Taste(n) gedrückt wurden, oder noch gedrückt sind) oder `getAmount()` (Anzahl der Tastendrücke seit der letzten Abfrage) abgefragt werden. In manchen Fällen ist es sinnvoll (z.B. bei Menüelementen), nur den einmaligen `Keydown`-Event des `InputCommands` zu behandeln. Für solche Fälle kann das Verhalten des Kommandos auf `InitialKeyPressOnly` gesetzt werden – dann liefert `isPressed()` nur bei der ersten Abfrage `true` zurück, danach erst wieder, wenn die Taste losgelassen und erneut gedrückt wurde.

Neben Tasten-Kommandos stellt der `InputManager` auch eine Schnittstelle zur Maus zur Verfügung: Die Methoden `getMouseX()` und `getMouseY()` liefern die absolute Position der Maus im `GameFrame` zurück. Die Maus kann aber auch im relativen Modus verwendet werden. Der `InputManager` sorgt für das Zentrieren der Maus im `GameFrame` und liefert über `getRelativeMouseX()` / `-Y()` die relative Bewegung der Maus seit dem letzten Spiel-Zyklus zurück.

Außerdem bietet der `InputManager` eine XML-Im- und Export-Funktion an. Die aktuellen Tasten-Zuweisungen können über `getXmlConfig()` geholt und über `setMapping()` gesetzt werden.

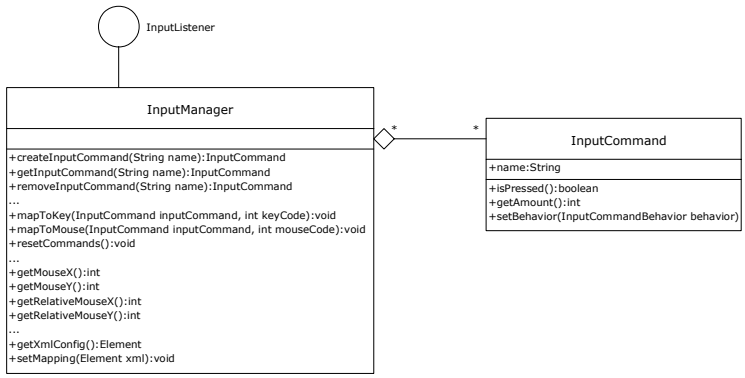


Abb. 2.19 InputManager und InputCommand

Zusätzlich zum normalen `InputManager` stehen ein `MenuInputManager` und ein `ExtendableInputManager` zur Verfügung, die den `InputManager` erweitern um:

- `MenuInputManager`: Vordefinierte `InputCommands` für Aktionen die häufig in Menüs benutzt werden (`ENTER`, `UP_ARROW`, `DOWN_ARROW`, `ESC`, etc.)
- `ExtendableInputManager`: Erweiterbar durch `KeyListener` (`KeyEvents` werden weitergeleitet)

Ein großer Nachteil, den das eigene *input handling* mit sich bringt, ist, dass keine Standard UI-Elemente verwendet werden können, um z.B. Texteingabe zu ermöglichen.

Um die Eingabe von Text zu ermöglichen wurde der `LineInputReader` implementiert, der, an einen `ExtendableInputManager` angehängt, eine einfache Texteingabe ermöglicht. Die aktuelle Implementierung unterstützt die Eingabe und das Löschen von Buchstaben und die Navigation des Cursors innerhalb der Eingabezeile. Die UI-Schnittstelle für den `LineInputReader` ist das `TextFieldSprite`, das, wie jedes andere Sprite, auf dem `GameScreen` platziert werden kann, und sich, wenn es den Focus hat, wie ein einfaches Textfeld verhält.

2.3.5 Sound

Soundeffekte und Hintergrundmusik sind in Spielen essentiell. Grundsätzlich unterscheidet man zwischen *sampled sound* und *MIDI clips*.

Sampled sounds sind digitalisierte analoge Audiodaten. Die Daten werden mit Abtastraten von z.B. 44,1 Hz, 22,05 Hz oder niedriger sowie in 8- oder 16-Bit Auflösung verarbeitet.

MIDI ist ein Protokoll das Befehlsfolgen für die Ansteuerung digitaler Instrumente enthält. Diese können extern über einen MIDI-Port angeschlossen sein oder werden, wie heutzutage üblich, direkt von der Soundkarte wieder gegeben.

Zur Wiedergabe von Sound stellt Java in der Package `javax.sound` einige Klassen und Schnittstellen zur Verfügung. Die einfachste Methode Sample-Audiodaten wiederzugeben ist die Verwendung der `AudioClip`-Klasse. Sie übernimmt das Laden der Sounds in den Speicher, die Weitergabe an das Audiosystem und ermöglicht das Ändern der Lautstärke, sowie das Stummschalten. Nachteil ist aber, dass je nach Hardware und Betriebssystem nur eine begrenzte Anzahl von Clips gleichzeitig abgespielt werden können und dass für jeden parallel abgespielten Sound ein eigener Clip benötigt wird, auch wenn der gleiche Sound mehrfach abgespielt wird (z.B. bei kurz hintereinander auftretenden Explosionen). Gegen ein Austauschen der Clips bei der Wiedergabe spricht, dass das Vorbereiten der Clips für das Abspielen verhältnismäßig lange dauert, da sie vollständig gepuffert werden. Außerdem können nur relativ kurze Audiosequenzen in einen Clip geladen werden.

Dies alles spricht gegen den Einsatz von `AudioClips`, da es in Spielen möglich sein muss, unter Umständen viele Soundeffekte (auch oft denselben) gleichzeitig und in Echtzeit wiederzugeben.

Als Alternative stellt Java die `SourceInputLine` zur Verfügung, die im Prinzip als direkte Quelle des Hardware-Audio-Mixers beschrieben werden kann. Der `SoundManager` verwendet diese zur Wiedergabe von *sampled sounds*. Zusätzlich besitzt er einen `MidiPlayer`, der MIDI-Sequenzen laden und wiedergeben kann.

2.3.5.1 SoundManager

Hauptaufgabe des `SoundManager` ist die Wiedergabe von *sampled sounds* (Abspielen und Laden). Als Dateiformat werden mit der momentanen Implementierung WAV-Dateien im Format *16 bit unsigned, mono* mit Sampleraten von *44.1*, *22* oder *11* kHz unterstützt.

Die Daten der Sounddateien werden in Objekten der Klasse `Sound` in einem Byte-Array gehalten und dem `SoundManager` zum Abspielen übergeben. Sie müssen direkt in eine `SourceDataLine` des `Mixer` vom Java `AudioSystem` geschrieben werden.

Da es möglich sein soll, viele Sounds gleichzeitig wiederzugeben, muss das Schreiben der Audiodaten in die `SourceDataLine` einzelnen Threads überlassen werden – d.h. für jeden (gleichzeitig) abgespielten Sound wird ein Thread benötigt. Um unnötige Performanceverluste durch ständiges Erzeugen neuer Threads zu vermeiden, ist der `SoundManager` von der Klasse `ThreadPool` abgeleitet, die das *thread pool* Design Pattern implementiert. Die *worker threads* des *thread pools* sind Objekte der Klasse `SoundPlayer`, die `Runnable` implementiert und in ihrer `run()`-Methode die, in einen `InputStream` gekapselten, Audiodaten an die, ihm zugeordnete `SourceDataLine`, schreibt.

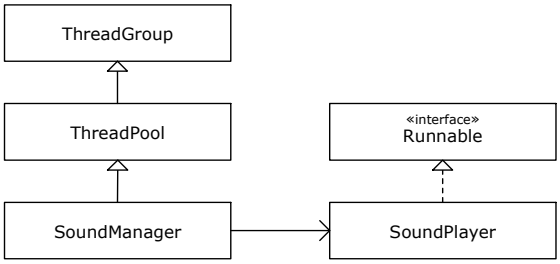


Abb. 2.20 Klassendiagramm SoundManager (vereinfacht)

Dieser Aufbau ermöglicht die einfache Manipulation der Audiodaten in Echtzeit, während sie in die Line geschrieben werden. Diese Funktionalität ist in der Klasse `SoundFilter` implementiert. Wenn zum Abspielen eines Sounds ein Filter verwendet werden soll, wird dem `SoundPlayer` statt eines `InputStreams` ein `FilteredSoundStream` (abgeleitet von `FilteredStream`), dem der `SoundFilter` angehängt ist, übergeben. Mit Hilfe eines Filters wird auch das Problem gelöst, dass die Lautstärke bei der Wiedergabe über eine `SourceDataLine` nicht geändert werden kann. Der `VolumeFilter` skaliert die im Sound enthaltenen Samplewerte um einen Faktor zwischen 0.0 und 1.0 – so kann im Spiel die Lautstärke einzelner Soundeffekte geregelt werden, auch während sie abgespielt werden.

2.3.5.2 MIDI-Player

MIDI Sequenzen können einfach über den `Sequencer` des *Java AudioSystems* wiedergegeben werden (sofern die Soundkarte dies unterstützt). Die **JMPG**-Klasse `MidiPlayer` kapselt die Funktionalitäten des Ladens und der Wiedergabe von MIDI-Dateien und ist im `SoundManager` enthalten.

2.3.6 Kollisionserkennung

Bei Spielen ist eine Kollisionserkennung unumgänglich. **JMPG** stellt einfache Hilfsmittel für eine 2D-Kollisionserkennung. Aus Zeitgründen haben wir uns auf eine rein statische Erkennung von Überlappungen einfacher geometrischer Formen und Bitmaps beschränkt. Bei einfachen Spielen mit sich relativ langsam bewegenden Objekten genügt diese Methode, sie kann aber bei schnellen Bewegungen zu Ungenauigkeiten führen, wenn sich ein Objekt innerhalb eines *Frames* durch ein anderes hindurchbewegt (siehe *Abb. 2.21*). Vermeiden lassen sich solche Ungenauigkeiten entweder durch die Verkleinerung des Zeitintervalls, in dem die Kollisionserkennung durchgeführt wird, oder durch Berechnung der Außenkanten der Bahn, auf der sich die Objekte bewegen (orangene Linien in *Abb. 2.21*).

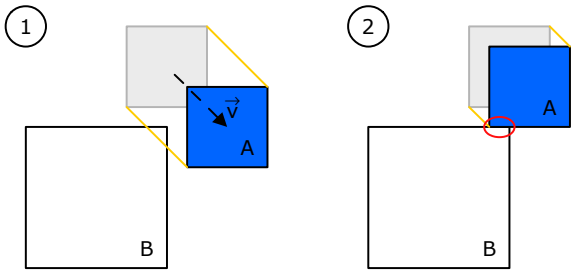


Abb. 2.21 Ungenauigkeiten bei der Kollisionserkennung

2.3.6.1 Collision shapes

Die Kollisionserkennung von **JMPG** basiert auf einigen Grundformen (`CollisionShape`), zwischen denen die Kollision berechnet werden kann. Den Sprites im Spiel werden diese Formen zugewiesen. Die Wahl der Form richtet sich nach der benötigten Genauigkeit der Kollisionsberechnung – bei sehr kleinen, wenige Pixel großen Objekten genügt es meist sogar, nur einen Punkt zu verwenden.

Folgende `CollisionShapes` stellt **JMPG** zur Verfügung:

`Rect`

Das Rechteck (`Rect`) ist die Basisform für Kollisionsobjekte. Sie ist in jedem anderen `CollisonShape` als Begrenzungsrechteck (`bounding rect`) enthalten. Es wird immer erst getestet, ob sich die Begrenzungsrechtecke zweier Objekte schneiden – ist dies nicht der Fall, braucht keine genauere Kollisionsabfrage durchgeführt werden.

`Circle`

Der Kreis (`Circle`) ist ähnlich einfach wie das Rechteck und eignet sich gut für kleine Objekte. Auch um zu testen, ob sich ein Objekt innerhalb eines bestimmten Abstands zu einem andern befindet, kann der Kollisionskreis eingesetzt werden.

`BitmapMask`

Um pixelgenaue Kollisionserkennung durchzuführen, kann eine Bitmap-Maske als Kollisionsform verwendet werden. Die Maske wird als Schwarz-Weiß-Bild geladen, wobei weiße Pixel die Form des Objekts beschreiben; schwarze Pixel sind leere Flächen.

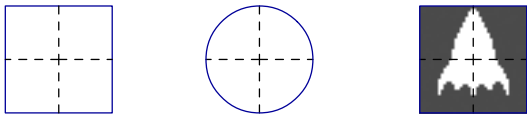


Abb. 2.22 Grundformen für die Kollisionserkennung (Rect, Circle, Bitmapmask)

`TiledBitmapMask`

Für die pixelgenaue Kollisionserkennung bei großen Bitmapmasken (z.B. für die Levelstruktur), steht die eine optimierte Version der `BitmapMask`, die `TiledBitmapMask` zur Verfügung. Die Maske wird in mehrere kleine Kacheln zerlegt die einen der drei Zustände haben:

`empty` → keine Pixel in dieser Kachel vorhanden

`full` → alle Pixel dieser Kachel sind gesetzt

`partly` → die Pixel dieser Kachel sind nur zum Teil belegt, d.h. es handelt sich um eine Kante

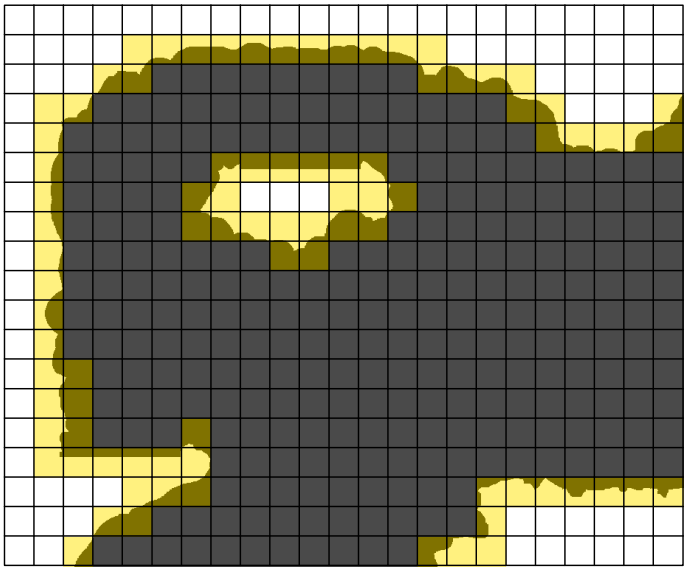


Abb. 2.23 Beispiel TiledBitmapMask

Durch diese Kachelung der Maske wird ein geringerer Speicherverbrauch, sowie eine schnellere Kollisionserkennung erreicht.

2.3.6.2 CollisionEngine

Die Klasse `CollisionEngine` stellt zwei grundlegende Methoden zur Verfügung:

Die Methode `collides(CollisionShape s1, collisionShape s2)` prüft, ob `s1` und `s2` kollidieren.

Die Methode `hittest(CollisionShape s, float posX, float posY)` prüft, ob sich der Punkt (`posX`, `posY`) innerhalb von `s` befindet.

Die Methoden sind überladen und für jede `CollisionShape`-Paarung getrennt implementiert. Allen gemein ist, dass wegen der Einfachheit des Tests, zunächst die Überlappung der Begrenzungsrechtecke getestet wird, wodurch die Geschwindigkeit der Kollisionserkennung besonders bei pixelgenauen Tests deutlich erhöht wird. Im Folgenden werden nur die Funktionen der Kollision von Bitmap-Masken kurz erläutert.

`BitmapMask`

Die Pixel der Bitmap-Maske werden durch ein zweidimensionales Byte-Array repräsentiert. Beim Kollisionstest mit den Formen Rechteck oder Kreis, wird geprüft, ob sich innerhalb der Schnittfläche des Maskenrands und der Form Pixel befinden; falls ja, findet eine Kollision statt. Beim Maske-Maske-Kollisionstest wird über die einzelnen Pixel im Bereich der Schnittflächen der Masken iteriert. Wenn in beiden Masken der getestete Pixel gesetzt ist (weiß), findet eine Kollision statt.

`TiledBitmapMask`

Beim Kollisionstest mit gekachelten Masken wird zunächst geprüft, welchen Zustand die Kacheln haben, die sich mit dem Begrenzungsrechteck der zu testenden Form überschneiden. Sind alle Kacheln leer, kann keine Kollision stattfinden (orangene Bereiche in Abb. 2.24 ①) - sind alle Kacheln voll, findet die Kollision mit Sicherheit statt. In beiden Fällen muss nicht weiter geprüft werden. Wenn eine oder mehrere der Kacheln teilweise gefüllt sind, d.h. einen Pixelrand enthalten, muss in den Schnittbereichen zwischen Form und diesen Kacheln wie bei der Bitmap-Maske pixelgenau getestet werden (rote Bereiche in Abb. 2.24 ②).

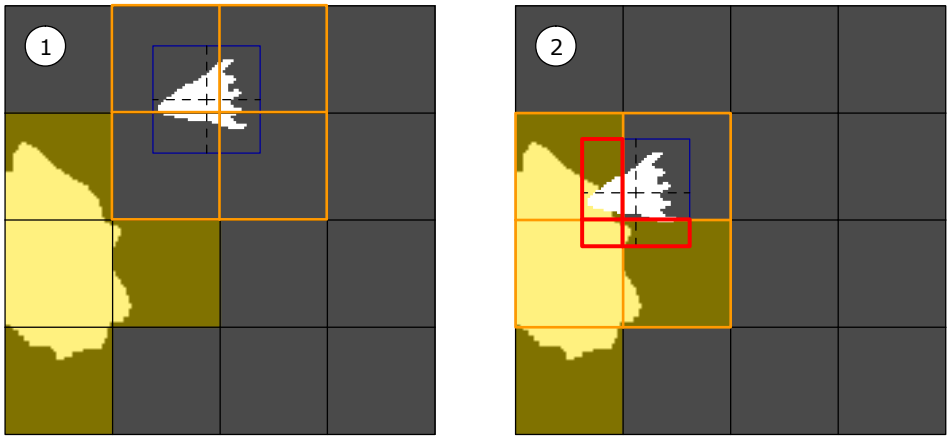


Abb. 2.24 Kollisionstest innerhalb einer TiledBitmapMask

2.3.7 Ressource Management

Alle im Spiel verwendete Ressourcen (Bilder, Animationen und Sound) können über die Ressource-Manager unkompliziert geladen und verwaltet werden. Sie werden in einer oder mehreren XML-Dateien definiert – diese können dem vom `GameResourceManager` hinzugefügt werden, der dann die Dateien entweder dann, wenn das erst mal auf sie Zugegriffen wird, oder alle Dateien im Voraus lädt. Letzteres kann synchron, also als blockierender Methodenaufruf, oder asynchron in einem eigenen Thread, geschehen.

Jede **JMPG**-Ressource wird durch die abstrakte, parametrisierte Klasse `GameResource` repräsentiert, die für die jeweiligen Ressource-Typen abgeleitet wird. Sie besitzt einen Namen, der die Ressource eindeutig identifiziert (er muss innerhalb eine Ressource-Typs eindeutig sein), die Daten der Ressource (z.B. ein `Image` bei Bildern) und eine Eigenschaft `loaded`, die angibt, ob die Ressource geladen ist. Jede von GameResource abgeleitet Klasse muss die `load()`- und `unload()`-Methoden für den entsprechenden Datentyp implementieren.

Für jeden Ressource-Typ (Bilder, Animationen, Sound) existiert ein eigener Ressource-Manager, die ähnlich aufgebaut sind: Sie halten die `GameResource`-Objekte in einer `HashMap`, die den Namen der Ressource als *Key* verwendet.

2.3.7.1 ResourceManager

Die Klasse `GameResourceManager` stellt die globale Schnittstelle zu den speziellen Ressource-Managern `ImageResourceManager`, `AnimationResourceManager` und `SoundResourceManager` dar und ermöglicht das Laden von Ressource-Dateien, die beliebige Ressourcetypen enthalten können. Nachdem eine oder mehrer Ressource-Dateien hinzugefügt wurden (→ `addResourceFile()`), kann auf die dort definierten Ressourcen direkt zugegriffen werden. Noch nicht geladene Daten werden bei Bedarf geladen. Über die Methode `loadAll(boolean forceReload, ResourceListener listener)` können alle Ressourcen vorgeladen werden. Der Parameter `forceReload` gibt dabei an, ob schon vorhandene Daten neu geladen werden sollen. Wird ein `ResourceListener` übergeben, erfolgt das Laden asynchron in einem eigenen Thread, der den `listener` informiert, wenn alle Daten geladen wurden.

Die zu ladenden Bilder können entweder in dem JAR-File enthalten sein, oder als externe Dateien im Dateisystem vorliegen. Es wird immer zunächst versucht die Ressource

außerhalb des JAR-Files zu finden. Existiert die Datei dort nicht, wird der relative Pfad zur Ressource innerhalb des JARs benutzt. Falls die Datei auch dort nicht zu finden ist, wird ein Fehler ausgegeben.

Die Ressourcen werden in den spezialisierten Managern (abgeleitet von der Klasse `AbstractResourceManager`) verwaltet.

2.3.7.2 ImageResourceManager

Der `ImageResourceManager` übernimmt das Laden und die Verwaltung der im Spiel verwendeten Bilder. Ein Bild wird im `ImageResource`-Objekt als `Image` gespeichert. Das Laden eines Bildes erfolgt über die Klasse `ImageIcon`, die beim Laden den `MediaTracker` verwendet.

Bilder werden in der Ressource-XML-Datei unter dem `images`-Element zusammengefasst und in einzelnen `image`-Elementen definiert. Jedes `image`-Element hat zwei Attribute:

- `name` - der eindeutige Name des Bildes
- `fileName` - der Dateiname des Bildes

```
<gameResources>
   <images>
      <image name="all_sprites" fileName="res/img/all_sprites.png"/>
      <image name="background" fileName="res/img/background.png"/>
   </images>
</gameResources>
```

2.3.7.3 AnimationManager

Der `AnimationManager` verwaltet Animationen (→ `AnimationResource`) und deren Einzelbilder (→ `AnimationFrameResource`). Der verwendet den `ImageReourceManager` um auf die Bilder, die die Einzelbilder der Animationen enthalten.

Die Einzelbilder einer Animation können entweder in einzelnen Bilddateien vorliegen, oder Teil einer größeren Bilddatei sein, wobei dort auch *Frames* von mehreren Animationen enthalten sein können.

In der Ressource-Datei werden zunächst die Einzelbilder, die auf ein vorher definiertes Bild referenzieren, angegeben und dann in einer Animationsdefinition zusammengefasst.

Einzelbilder können flexibel definiert werden: Entweder werden alle *Frames* mit ihrer absoluten Position und Größe innerhalb eines Bildes als `frame`-Element angegeben, oder mehrere *Frames* werden in einer `frameGroup` zusammengefasst. Solche `frameGroup`-Elemente definieren Bereiche eines Bildes, in denen die Einzelbilder zeilenweise enthalten sind.

Wenn sich mehrere `frames` oder `frameGroups` auf das gleiche Bild beziehen, können sie in einem `sourceImage`-Tag eingeschlossen werden. Man spart sich dann die Referenzierung des Bildes innerhalb jedes einzelnen *Frame*-Elements.

Der `AnimationManager` kopiert die Einzelbilder aus dem Quellbild heraus und verwaltet sie in `AnimationFrameResource`-Objekten, auf die einzeln zugegriffen werden kann. Er kann aus ihnen aber auch direkt Animationen erzeugen: Animationen werden in der Ressource-Datei durch ein `animation`-Element definiert, in dem mehrere *Frames*, oder *frameGroups* referenziert und weitere Eigenschaften der Animation festlegt werden.

Nachfolgend wird die Definition von Animation in der Ressource-Datei an einem Beispiel erläutert.

Abb. 2.25 zeigt ein Bild, das drei verschiedene Explosions-Animationen enthält: Die Animationen bestehen jeweils aus 40 Einzelbildern, die in drei Gruppen zeilenweise in einem Bild angelegt sind.

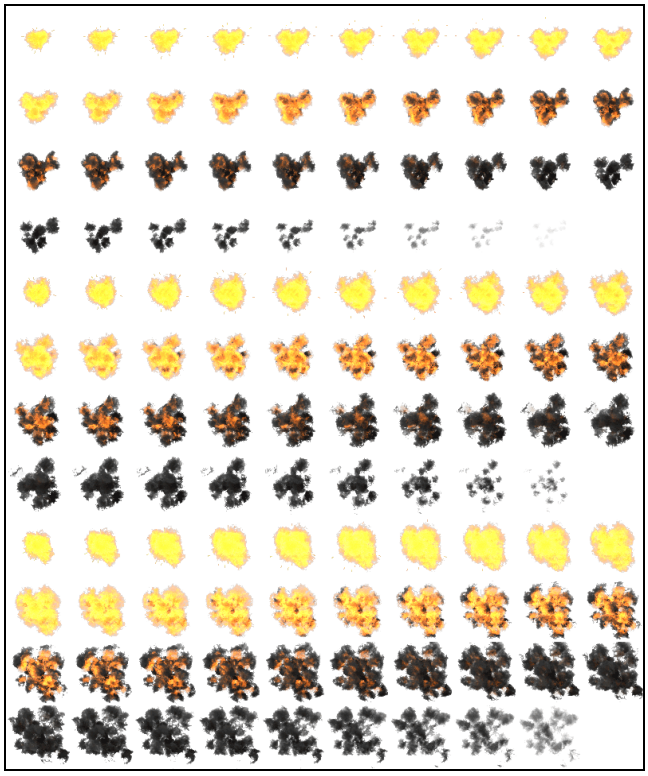


Abb. 2.25 Einzelbilder von drei Animationen in einer Bilddatei

Zunächst muss das Bild, das die Einzelbilder enthält, als `ImageResource` definiert werden:

```
<gameResources>
   <images>
      <image name="explosions" fileName="res/img/explosion_sprites.png"/>
   </images>
```

Da die Einzelbilder der drei Animation zusammenhängend im Bild vorliegen, können sie als `frameGroup`-Elemente definiert werden. Alle beziehen sich auf das Bild „explosions“ – also werden sie in einem `sourceImage`-Tag eingeschlossen.

```
   <animationFrames>
      <sourceImage name="explosions">
         <frameGroup name="expA" frameCount="40">
            <position startX="0" startY="0"/>
            <frameSize width="80" height="80"/>
            <frameCount x="10" y="4"/>
         </frameGroup>
         <frameGroup name="expB" frameCount="40">
            <position startX="0" startY="320"/>
            <frameSize width="80" height="80"/>
            <frameCount x="10" y="4"/>
         </frameGroup>
         <frameGroup name="expC" frameCount="40">
            <position startX="0" startY="320"/>
            <frameSize width="80" height="80"/>
            <frameCount x="10" y="4"/>
         </frameGroup>
      </sourceImage>
   </animationFrames>
```

Auf die Einzelbilder können jetzt über ihren Namen und ihrem Index referenziert werden. Die Notation hält sich dabei an den Array-Zugriff von Java (z.B. `expA[0]`). Es können aber mit einem `frameSequence`-Element auch Gruppen referenziert werden.

```
   <animations>
      <animation name="explosionA" type="timebased">
         <animFrames>
            <frameSequence groupName="expA" startFrame="0" frameCount="40"/>
         </animFrames>
         <pivot x="40" y="40"/>
         <constantDuration value="40"/>
      </animation>
   <animations>
   <animations>
      <animation name="explosionB" type="timebased">
         <animFrames>
            <frameSequence groupName="expB" startFrame="0" frameCount="40"/>
         </animFrames>
         <pivot x="40" y="40"/>
         <constantDuration value="40"/>
      </animation>
   <animations>
   <animations>
      <animation name="explosionC" type="timebased">
         <animFrames>
            <frameSequence groupName="expC" startFrame="0" frameCount="40"/>
         </animFrames>
         <pivot x="40" y="40"/>
         <constantDuration value="40"/>
      </animation>
   <animations>
</gameResources>
```

Alle Animationen werden als Zeit basierte Animationen (→ `TimeBasedAnimation`) mit einer konstanten Bildrate von 25 fps (40 ms pro *Frame*) angelegt. Zusätzlich wird der Pivot-Punkt in die Mitte (40,40) der 80x80 Pixel großen Animationen gesetzt.

2.3.7.4 SoundResourceManager

Der `SoundResourceManager` übernimmt das Laden und die Verwaltung von `GameSounds`. Ein `GameSound` kann entweder Sample-Sound- oder MIDI-Daten enthalten. Für Laden der Daten wird auf die Methoden des `SoundManagers` zurückgegriffen.

Die im Spiel verwendeten Sounds werden in der Ressource-Datei unter dem Element `sounds` zusammengefasst. Dort werden die einzelnen Sounds in sound-Elementen mit Name und Dateiname definiert – der Typ des Sounds wird anhand der Dateiendung festgelegt (`.wav` für Sampledsound und `.mid` für MIDI-Sound).

```
<gameResources>
   <sounds>
      <sound name="launch-missile" fileName="res/snd/launch-missile.wav"/>
      <sound name="explosion-missile" fileName="res/snd/explosion-missile.wav"/>

      <sound name="music" fileName="res/snd/geodome.mid"/>
   </sounds>
</gameResources>
```

Die GameSound-Objekte werden im `SoundResourceManager` über seinen Namen referenziert und können direkt an den `SoundManager` (→ *Kap. 2.3.5.1*) zur Wiedergabe übergeben werden.

2.3.8 JMPG 2D-API Demo

Um den Einsatz der **JMPG** 2D-API zu veranschaulichen, stellen wir kurz ein einfaches Beispiel vor, das die Verwendung der Basisklassen demonstriert.

Im Beispiel sollen sich mehrere animierte Objekte (die Mine aus Gravity) zufällig umherbewegen und dabei am Bildschirm- bzw. Fensterrand abprallen.

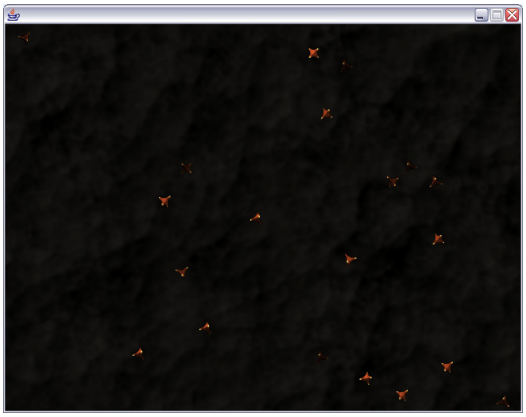


Abb. 2.26 Demo-Anwendung

Es werden zwei Bildressourcen verwendet:

Ein Hintergrundbild `"background.png"` und ein Bild `"mine.png"`, das die Einzelbilder der Animation für die Mine enthält:

Abb. 2.27 Hintergrundbild (`background.png`)

Abb. 2.28 Animationssequenz der Mine (`mine.png`)

In der Ressource-Datei werden die beiden Bilder und die Animation definiert:

```
<?xml version="1.0"?>
<gameResources>
   <images>
      <image name="background" fileName="res/img/background.png"/>
      <image name="mine" fileName="res/img/mine.png"/>
   </images>
   <animationFrames>
      <frameGroup name="mineFrames" frameCount="20" sourceImage="mine">
         <position startX="0" startY="0"/>
         <frameSize width="20" height="20"/>
         <frameCount x="10" y="2"/>
      </frameGroup>
   </animationFrames>
```

```
   <animations>
      <animation name="mine" type="timeBased">
         <animFrames>
            <frameSequence groupName="mineFrames" startFrame="0" frameCount="20"/>
         </animFrames>
         <pivot x="10" y="10"/>
         <constantDuration value="80"/>
      </animation>
   </animations>
</gameResources>
```

Listing 2.1 `demoRes.xml`

Um die Anwendung mit der **JMPG** 2D-API umzusetzen genügt eine Java-Datei (*Listing 2.2*):

Den Hauptteil stellt die Klasse `SpriteAnimationDemo` dar, die von `Game` abgeleitet ist und auch eine `main()`-Methode implementiert, in der die Demo gestartet wird. Von `Game` wird die `init()`-Methode überschrieben.

Der Einfachheit halber beschränken wir uns bei der Umsetzung des Abprallen der Sprites vom Rand des Fensters auf das Umkehren der Geschwindigkeit der Sprites, wenn sie sich über den Rand hinaus bewegt haben. Dazu wird ein von `MovingSprite` abgeleitetes `DemoSprite` implementiert, in dessen `update()`-Methode die Position geprüft wird und, falls nötig, die entsprechende Geschwindigkeitskomponente invertiert wird.

In der `init()`-Methode von `SpriteAnimationDemo` wird der Reihe nach:

- das GameFrame initialisiert (640 x 480 Pixel) und als Fenster angezeigt,
- die Bildrate auf 25 Bilder (40 ms pro Bild) gesetzt,
- die RessourceDatei `demoRes.xml` dem Standard-`ResourceManager` hinzugefügt,
- ein `SimpleGameScreen`-Objekt `screen` erzeugt,
- das Bild `"background"` des `ImageResourceManagers` in die Hintergrund-Ebene von `screen` gesetzt,
- die (virtuelle) Größe der Sprite-Ebene von `screen` auf die Auflösung des Spiels gesetzt,
- in einer Schleife 20 `DemoSprites` erzeugt, denen die Animation `"mine"` mit zufälligem Zeitindex, sowie eine zufällige Position und Geschwindigkeit zugewiesen wird, und die der Sprite-Ebene von `screen` hinzugefügt werden, und
- der `screen` angezeigt.

```
import org.jmpg.core.Game;
import org.jmpg.core.GameTimer;
import org.jmpg.ui.SimpleGameScreen;
import org.jmpg.ui.sprite.MovingSprite;

public class SpriteAnimationDemo extends Game
{
    @Override
    public void init() {
        // Auflösung des Spiels festlegen
        getGameFrame().setSize(640,480);
        // In den Fenstermodus schalten
        getGameFrame().enterWindowedMode();
        // Bildrate auf 25 Bilder pro Sekunge beschänken
        GameTimer.getDefaultTimer().setMinFrameTime(40);

        // Ressource-Datei laden
        this.getResourceManager().addResourceFile("res/demoRes.xml");
```

```
        // Als GameScreen wird ein SimpleGameScreen verwendet
        SimpleGameScreen screen = new SimpleGameScreen();

        // Setzen des Hintergrundbildes im backGroundLayer
        screen.getBackgroundLayer().setImage(this.getResourceManager().getImgResManager().get("background").getData());

        // Größe des SpriteLayer an die Spielauflösung festlegen, um das Apprallen der Sprites am Rand zu ermöglichen
        screen.getSpriteLayer().setWidth(getGameFrame().getWidth());
        screen.getSpriteLayer().setHeight(getGameFrame().getHeight());

        // Erzeugen von 20 Sprites
        DemoSprite demoSprite;
        for(int i = 0; i < 20; i++)
        {
            // neues DemoSprite
            demoSprite = new DemoSprite();
            // setzen der Animation
            demoSprite.setAnimation(this.getResourceManager().getAnimationManager().getAnimation("mine").getCopy());
            // Sprite zufällig auf dem Screen positionieren
            demoSprite.setPosition((float)Math.random() * 620 + 10, (float)Math.random() * 460 + 10);
            // Geschwindigkeit festlegen
            demoSprite.setVelocity((float)Math.random() * 200 - 100, (float)Math.random() * 200 - 100);

            // da eine Animationskopie verwednet wird, kann die Animationszeit der einzelnen Sprites
            // unabhängig variiert werden
            demoSprite.getAnimation().setCurrentTime((int)(Math.random() * demoSprite.getAnimation().getTotalDuration()));

            // Hinzufügen des Sprites zum spriteLayer des SimpleGameScreens
            screen.getSpriteLayer().addSprite(demoSprite);
        }

        // GameScreen anzeigen
        showScreen(screen);
    }

    public static void main(String[] args) {
        SpriteAnimationDemo demo = new SpriteAnimationDemo();
        demo.start();
    }

    private class DemoSprite extends MovingSprite
    {
        // Die update Methode wird überschrieben, um das Abprallen am
        // Fensterrand zu implementieren
        @Override
        public void update(long timeElapsed) {
            // super.update aktualisiert die Animation und
            // bewegt den Sprite
            super.update(timeElapsed);

            // Falls sich der Sprite außerhalb des screens befindet
            // wird die entsprechende GEschwindigkeits-Komponente invertiert.
            if(getPosX() < 10 || getPosX() > getParent().getWidth() - 10)
                setVelocityX(-getVelocityX());
            if(getPosY() < 10 || getPosY() > getParent().getHeight() - 10)
                setVelocityY(-getVelocityY());
        }
    }
}
```

Listing 2.1 `SpriteAnimationDemo.java`

2.4 JMPG Multiplayer-API

Ziel bei der Entwicklung der Multiplayer-Komponente von **JMPG** war es, eine einfache Schnittstelle zu schaffen, auf deren Basis Multiplayer-Spiele entwickelt werden können.

Wir entschieden uns für eine Event basierte Client-/Server-Architektur – d.h. Server und Client tauschen über das Netzwerk Events aus, die die Änderungen am Spielzustand beschreiben.

2.4.1 Grundlagen

Bei Multiplayer-Spielen muss der *game state* an alle teilnehmenden Spieler über ein Netzwerk kommuniziert werden. Bei Runden basierten Spielen spielt die zugrunde liegende Netzwerktechnik keine große Rolle, da keine Daten in Echtzeit übertragen werden müssen. Bei Echtzeitspielen ist es essenziell, dass die Daten im Netz schnell übertragen werden.

Grundsätzlich gibt es zwei Netzwerk-Modelle für Multiplayer-Spiele:

Peer-to-peer

Beim *peer-to-peer*-Model gibt es keine zentrale Spielinstanz - das Spiel läuft vollständig auf allen Clients, d.h. jeder Client hat Kenntnis über den kompletten *game state* und kann diesen verändern. Jeder Client kommuniziert Änderungen am *game state* an alle anderen Clients. Bewegt der Spieler z.B. seine Spielfigur, wird diese Bewegung zunächst lokal verarbeitet und dann an alle Clients weitergegeben, sodass diese die Zustandsänderung in ihren *game state* aufnehmen können.

Dieses Modell bringt einige Nachteile mit sich: Der Spielzustand ist nie eindeutig, denn durch die Verzögerung, die das Senden der Informationen über das Netzwerk hervorruft, sind die Zustände des Spiels der Clients untereinander immer asynchron. Dies kann Probleme bereiten, wenn auf zwei Clients parallel Situationen eintreten, die sich gegenseitig ausschließen, da beide Clients noch einen veralteten Zustand der anderen Seite kennen. Die Entscheidung, wie die Situation behandelt wird, muss dann entweder einem Client überlassen, oder zwischen den Clients ausgehandelt werden. Sind viele Clients am Spiel beteilig, müssen sehr viele Daten über das Netz übermittelt werden.

Aus diesen Gründen wird das *peer-to-peer*-Modell – wenn überhaupt – nur in Netzen mit geringen Latenzzeiten und hoher Bandbreite (LAN) eingesetzt – oder in Spielen mit nur zwei Spielern.

Client/Server

Beim *Client/Server*-Modell wird der *game state* zentral auf dem Server gehalten und an die teilnehmenden Spieler kommuniziert. Die Clients besitzen nicht den kompletten Zustand des Spiels, sondern zeigen nur die Informationen an, die sie vom Server erhalten. Aktionen (z.B. Bewegungen) teilen sie dem Server mit, der den Spielzustand entsprechend aktualisiert. Vorteil dieser Methode ist, dass nur genau eine Instanz des Spiels auf dem Server läuft und er damit die komplette Entscheidungsgewalt besitzt – Konflikte zwischen den Clients können nicht auftreten. Außerdem kann er die Netzlast verringern, indem er an jeden Client nur die Teile des *game state* sendet, die direkten Einfluss auf dessen Aktionen haben können und in seinem Sichtfeld liegen – z.B. benötigt ein Client für weit entfernte, nicht sichtbare Objekte keine Positionsdaten. Das Problem, dass Verzögerung im Netzwerk auftreten und die Bandbreite beschränkt ist, bleibt weiterhin bestehen. Es gibt verschiedene Ansätze dem entgegen zu wirken. Einerseits ist die Wahl des Netzwerkprotokolls entscheidend (TCP oder UDP). TCP bietet zwar im Gegensatz zu UDP eine zuverlässige Verbindung, d.h. es stellt sicher, dass gesendete Pakete bei der Gegenstelle ankommen (es sei denn, die Verbindung wird unterbrochen) und dass sie in der richtigen Reihenfolge ankommen, hat aber im Vergleich zu UDP einen größeren Overhead (20 Byte, UDP: 8 Byte) und ist langsamer als UDP (auch wegen der Fehlerkorrektur). UDP stellt nichts sicher – weder, dass gesendete Pakete ankommen, noch, dass sie in der Reihenfolge ankommen, in der sie gesendet wurden. Diese Nachteile überwiegen aber nicht den Geschwindigkeitsvorteil denn, falls einzelne Pakete, die z.B. Spielerposition enthalten, nicht verloren gehen, äußert sich das beim Client nur durch leichtes Springen – diesen Effekt kann man durch Inter- bzw. Extrapolation vermindern. Erst durch den Verlust sehr vieler Pakete wird das Spielerlebnis beeinträchtigt. Das Problem der Latenz im Netzwerk besteht aber immer: die Eingaben des Spielers werden so viel später vom Server registriert, wie die Pakete zu ihm unterwegs sind. Dazu kommt der Weg vom Server zu den anderen Clients – d.h. es kommt eine weiter

Verzögerung hinzu. Moderne Spiele verwenden *client prediction* um die Latenz zu verbergen: der Client nimmt an, dass der Server die Bewegungsinformation wirklich erhalten hat und diese verarbeiten wird und aktualisiert seinen Zustand dem entsprechend. Der Server lässt die Information rückwirkend in den *game state* mit einfließen und kommuniziert die Änderung an die Clients.

Wie man sieht gibt es viele Ansätze, Netzwerk-Spiele umzusetzen. Auch sind die Anforderungen, die unterschiedliche Spieltypen an das Netzwerk stellen sehr unterschiedlich. Eine allgemeine Lösung zu finden war in der kurzen Zeit, die wir für die Entwicklung der **JMPG**-Multiplayer-Schnittstelle zur Verfügung hatten nicht möglich.

Deswegen haben wir uns für eine einfache Implementierung eines Client-Server-Modells auf Basis von UPD beschränkt. Diese ermöglicht einem Spieleentwickler auf einfache Art und Weise eine Kommunikation zwischen Server und Clients aufzubauen, ohne sich direkt mit der Programmierung der unteren Netzwerkschichten auseinandersetzen zu müssen.

In Gravity wurde die **JMPG**-Multiplayer-API erfolgreich eingesetzt: Mehrere Spieler (getestet mit 8 Spielern) können im LAN oder über das Internet spielen.

2.4.2 JMPG-Multiplayer-Spiele

Die **JMPG**-Multiplayer-Schnittstelle stellt eine Client-Server-Verbindung über das UDP Protokoll zu Verfügung. Die Clients tauschen mit dem Server `GameEvents` aus, wobei keine Flusskontrolle implementiert ist.

Ein Spiel-Server verwendet die Klasse `GameServer`, die das Senden der Events über das Netz übernimmt und Events aus der Netzwerkschnittstelle liest. Nach dem eine Instanz von `GameServer` gestartet ist, können `GameClients` eine Verbindung zum Server aufbauen. Dieser hält eine Liste aller verbundener Clients in `RemotePlayer`-Objekten, die über ihre `playerId` eindeutig identifizierbar sind (Hashwert aus Client-IP & -Port).

Die Client-Version des Spiels benötigen nur eine Instanz der Klasse `GameClient`, und kann über diesen die Verbindung zu einem Server (mit bekannter Adresse) aufbauen und sofort Events senden.

Für die Netzwerkfunktionalität werden die seit *Java 1.5* vorhandene NIO-API (*new I/O*) verwendet. Die Schnittstelle zu UDP stellt der `DatagramChannel` bereit. Das Senden der Daten erfolgt gepuffert über einen `ByteBuffer`, dessen Inhalt direkt in den `DatagramChannel` geschrieben wird.

2.4.3 Client-Server-Verbindung

Um an einem Spiel teilzunehmen, muss ein Client eine Verbindung zum Server aufbauen. Da UDP verbindungslos arbeitet, wurde dies in die **JMPG**-Multiplayer-API integriert.

Mit dem Aufruf von `connect()` beim `GameClient` wird die Verbindung zum Server hergestellt. Dazu werden zunächst Informationen über das laufende Spiel angefordert (`REQUEST_GAMEINFO`) – diese liefert der Server als `GameInfo`-Objekt in einem `GameInfoEvent` zurück. Die `GameInfo` enthält Informationen über den aktuellen Level, über die momentane und maximal Anzahl von Spielern und über die Spielart.

Nachdem der Client das `GameInfo`-Objekt erhalten hat, kann er die Verbindung aufbauen (`CONNECTIONREQUEST`). Dabei vergleicht der Server die Spielversion des Client mit der eigenen. Sind beide kompatibel, antwortet der Server mit einem `CONNECTIONREPLY`-Event, der die `playerId` des Spielers enthält, und nimmt den Client in die Liste angemeldeter Spieler auf (vorher wird der Client nur temporär gehalten) – von nun an versucht der Server die Verbindung zum Client aufrecht zu erhalten in dem er, falls er lange keine Pakete von

ihm erhalten hat, KEEPALIVE-Events schickt, die der Client wiederum mit KEEPALIVE beantwortet.

Ist der Client verbunden, kann er dem aktuellen Spiel beitreten. Dazu sendet er einen JOINGAMEREQUEST – der Server antwortet (falls er die Anfrage akzeptiert) mit einem JOINGAMEREPLY und informiert die andere Spieler mit einem PLAYERJOINED-Event über den Beitritt des neuen Spielers. Die Behandlung des Spielbeitritts muss selber implementiert werden (z.B. im GameController), da die Entscheidung, wann ein Spieler einem Spiel beitreten darf nicht vom Framework übernommen werden kann.

Jetzt können die Spiel-Events über die Verbindung ausgetauscht werden – GameClient und GameServer spielen hierbei keine aktive Rolle mehr, sondern leiten die Events nur über das Netzwerk weiter.

Verlässt ein Spieler das Spiel, sendet er einen LEAVEGAME-Event an den Server, der mit einem LEAVEGAMEREPLY antwortet und die übrigen Spieler mit einem PLAYERLEFT-Event informiert. Der PLAYERLEFT-Event wird auch dann gesendet, wenn bei einem Client eine Zeitüberschritung auftritt, also wenn er zu lange nicht auf die KEEPALIVE-Pakete des Servers reagiert hat.

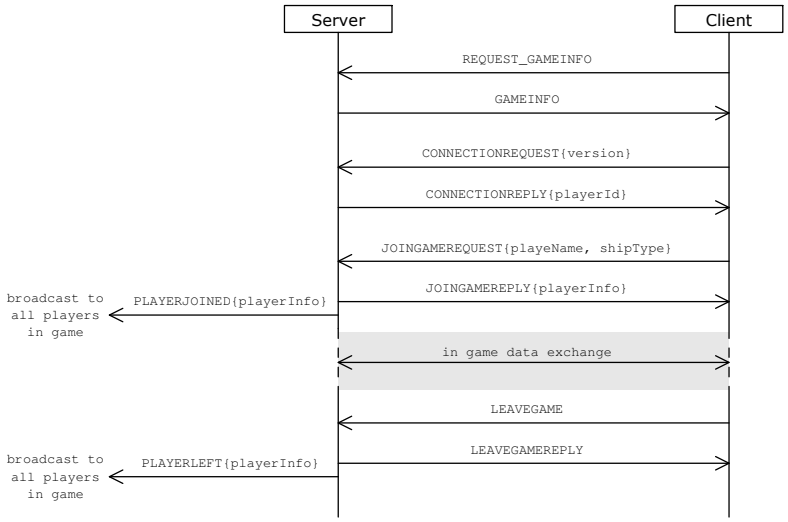


Abb. 2.29 Netzwerk-Verbindung zwischen Client und Server

2.4.4 Komponenten der Multiplayer-API

2.4.4.1 GameEvent

Die abstrakte Klasse GameEvent ist die Basis für den Austausch von Events zwischen Clients und Server. Sie beinhaltet immer eine Spieler-ID (playerId), den Type des Events (type), die Quelladresse, von der der Event empfangen wurde (source), sowie einen Wert der angibt, ob der Event schon verarbeitet wurde (handled). Optional kann in recipients eine Liste von Empfängern (genauer: deren playerIds) angegeben werden, falls der Event nur an ausgewählte Clients gesendet werden soll.

Zunächst muss ein Format definiert werden in dem die Events über das Netz gesendet werden. Damit der Empfänger einen Event parsen kann, benötigt er die Information, um was für einen Eventtyp es sich handelt und wie groß der Datenteil ist. Zusätzlich ist es für unsere Architektur sinnvoll die Spieler-ID immer mit zu senden. Diese Daten werden im *Header* zusammengefasst, er ist immer gleich aufgebaut und kann also leicht ausgewertet werden. *Abb. 2.30* zeigt den Aufbau des GameEvent wie er über das Netzwerk gesendet wird.

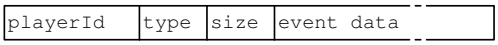


Abb. 2.30 GameEvent als Netzwerkpaket

Der *Header* (`playerId`, `type`, `size`) wird von der Methode `eventToBuffer()` der Klasse `NetUtil` in den übergebenen `ByteBuffer` geschrieben. Die Daten muss jeder `GameEvent` selber serialisieren. Hierzu müssen die Methoden `readData()` und `writeDate()` in jeder von `GameEvent` abgeleiteten Klasse implementiert werden. Wie dort die Daten des Events in den Puffer geschrieben werden, spielt prinzipiell keine Rolle – die Hauptsache ist, dass die `readData()` den Event aus den von `writeData()` geschriebenen Bytes wiederherstellen kann. Für elementare Datentypen stellt `ByteBuffer` jeweils Methoden zum Lesen und Schreiben bereit. Die Behandlung komplexer Datentypen und Objekte muss eigens implementiert werden.

Der Typ eines `GameEvent` ist unabhängig von der ihn implementierenden Klasse. Zum Beispiel kann mit Hilfe eines `MessageEvent` ein String transportiert werden - er kann aber für verschiedene Event-Typen verwendet werden.

JMPG stellt drei Implementierungen von `GameEvent` bereit:

`GameEventDefault`

Dieser `GameEvent` enthält keine Daten - nur der Event-Typ trägt seine Information. Er wird deswegen nur für einfache Benachrichtigungen verwendet (z.B. als Empfangsbestätigung, oder Spiel-Verlassen-Nachricht, etc.).

`MessageEvent`

Dieser Event enthält eine Textnachricht (`String`) als Nutzlast. Strings können mit den Hilfsmethoden `putString()` und `getString()` der Klasse `NetUtil` in einen `ByteBuffer` geschrieben bzw. aus ihm gelesen werden. *Abb. 2.31* und *Abb. 2.32* zeigen, wie eine `MessageEvents` geschrieben und gelesen wird.

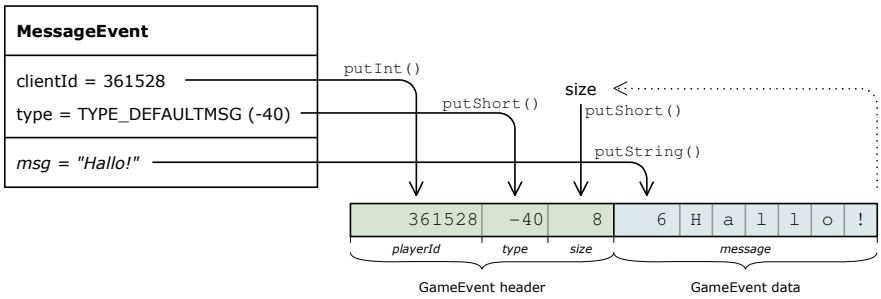


Abb. 2.31 Schreiben eines MessageEvents in einen ByteBuffer

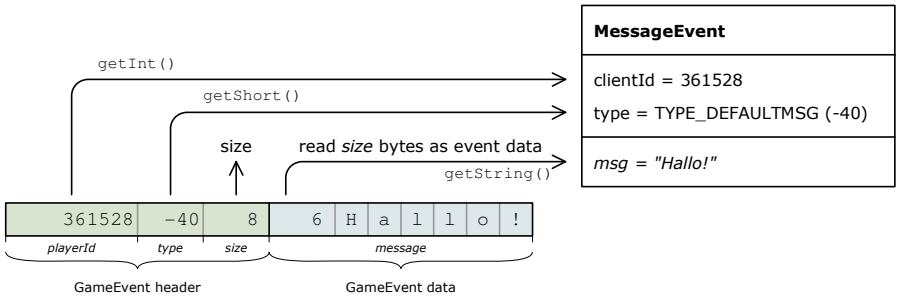


Abb. 2.32 Lesen eines MessagesEvents aus einem ByteBuffer

GameInfoEvent

Der GameInfoEvent enthält die GameInfo des auf dem Server laufenden Spiels. Die *Properties* der GameInfo werden der Reihe nach in den ByteBuffer serialisiert.

Diese drei Eventtypen reichen für den Verbindungsaufbau von GameClient zu GameServer aus. Alle weiteren im Spiel benötigten Events müssen speziell implementiert werden.

Das bedeutet, die Netzwerk-API muss eine erweiterbare Schnittstelle für das Parsen von Events zur Verfügung stellen, da die Event-Klasse nur anhand des, aus dem *Header* gelesenen, Eventtyps bestimmt werden kann.

Die Methode parseEvent() der Klasse NetworkEventFactory liefert den, zum im ByteBuffer codierten Eventtype passenden, GameEvent zurück. Dazu wertet sie zunächst den *Header* des Events aus. Ist die Größe des Events Null, wird ein GameEventDefault zurück geliefert. Andernfalls wird geprüft, ob die Methode getCustomEvent(short type) ein Objekt zurückliefert – ansonsten wird ein passender Standardevent gesucht. Um eigene Events mit eigenen Eventtypen zu verwenden, muss also die Klasse NetworkGameFacotry abgeleitet und die getCustomEvent()-Method überschrieben werden. Dabei können auch für Standardeventtypen Objekte eigener GameEvent-Implementierungen zurückgegeben werden. Die für den Verbindungsaufbau nötigen Events dürfen jedoch nicht geändert werden – es sei denn, der Verbindungsaufbau wird in GameServer und GameClient überschrieben.

JMPG stellt eine Reihe von Standardevents zur Verfügung, die unter anderem für den Verbindungsaufbau verwendet werden. Sie sind als statische short-Werte der GameEvent Klasse definiert und alle negativ – Eventtypen eigener Implementierungen müssen positive Werte haben, um sie von den Standardevents unterscheiden zu können. *Tabelle 2.2* zeigt eine Übersicht über die Eventtypen. Kursiv geschriebene Typen werden zur Verbindungskontrolle verwendet – die zugehörigen GameEvents dürfen nicht ausgetauscht werden (bei Bedarf nur erweitert).

Tabelle 2.2 Standardeventtypen

Eventtyp	*Wert*	*Beschreibung*
TYPE_UNKNOWN	0	Unbekannter Event
TYPE_ACK	-1	Bestätigung für den Empfang eines Events
TYPE_GAMEINFO	-2	GameInfoEvent mit Informationen über das Spiel
TYPE_REQUEST_GAMEINFO	-3	Anfrage für das Senden eines GameInfo-Events
TYPE_CONNECTIONREQUEST	-4	Verbindungsaufbauwunsch
TYPE_CONNECTIONREPLY	-5	Antwort bei erfolgreichen Verbindungsaufbau
TYPE_JOINGAMEREQUEST	-6	Anfrage, einem Spiel beizutreten
TYPE_JOINGAMEREPLY	-7	Antwort bei erfolgtem Spielbeitritt
TYPE_JOINGAMEFAILED	-8	Antwort bei nicht erfolgten Spielbeitritt
TYPE_PLAYER_JOINED	-9	Benachrichtigung, dass ein neuer Spieler dem Spiel beigetreten ist
TYPE_LEAVEGAME	-10	Spieler verlässt Spiel
TYPE_LEAVEGAMEREPLY	-11	Antwort auf LEAVEGAME
TYPE_PLAYER_LEFTGAME	-12	Benachrichtigung, dass ein Spieler das Spiel verlassen hat
TYPE_KEEPALIVE	-16	Event zum Aufrechterhalten der Verbindung

TYPE_TIMEOUT	-17	Timeout ist aufgetreten
TYPE_CONNECTION_FAILED	-22	Antwort bei fehlgeschlagenem Verbindungsaufbau
TYPE_SERVERSHUTDOWN	-30	Nachricht, dass der Server beendet wird
TYPE_WRONGVERSION	-33	Antwort, wenn die Versionsüberprüfung beim Verbindungsaufbau fehlschlägt
TYPE_DEFAULTMSG	-40	Standard Nachricht in Textform

2.4.4.2 GameClient

Der `GameClient` stellt eine Verbindung zu einem `GameServer` her und ermöglicht das Senden und Empfangen von `GameEvents`.

Es werden für das Schreiben zu und Lesen von der Netzwerkschnittstelle zwei getrennte Threads verwendet. Die Events werden dabei zunächst in *Queues* zwischen gespeichert, die periodisch abgearbeitet werden.

Soll ein Event gesendet werden (`sendEvent()`), wird er in den *Queue* für ausgehende Events gelegt (`outgoing`). Der Hauptthread des `GameClient` arbeitet die anstehenden Events im Takt von 10 ms ab und schreibt sie in den Ausgangs-`ByteBuffer`, der dann dem `DatagramChannel` zum Senden übergeben wird. Im gleichen Thread werden die eingegangenen Events, die der `Reader`-Thread aus dem `DatagramChannel` liest und im `incomming`-Queue ablegt, verarbeitet und an den dem `GameClient` angehängten `EventHandler` weitergeleitet, falls es sich nicht um Verbindungsaufbau- oder Kontroll-Events handelt.

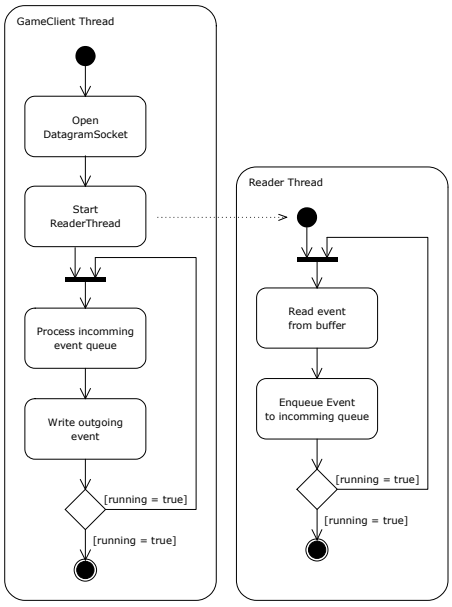


Abb. 2.33 Sende- und Lese-Thread von GameClient

Beim Senden der Events werden die anstehenden momentan im `outgoing`-Queue anstehenden Events nicht als einzelne UDP-Pakete gesendet, sondern es werden so viele Events zu einem Paket zusammengefasst (→ *Abb. 2.34*), bis der Sende-Puffer voll ist (Standardgröße: 512 Byte). Damit kann die Anzahl der zu sendenen Pakete deutlich verringert werden (→ *Abb. 2.35*).

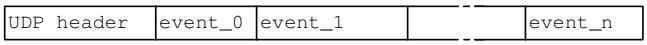


Abb. 2.34 UPD-Paket mit mehren GameEents

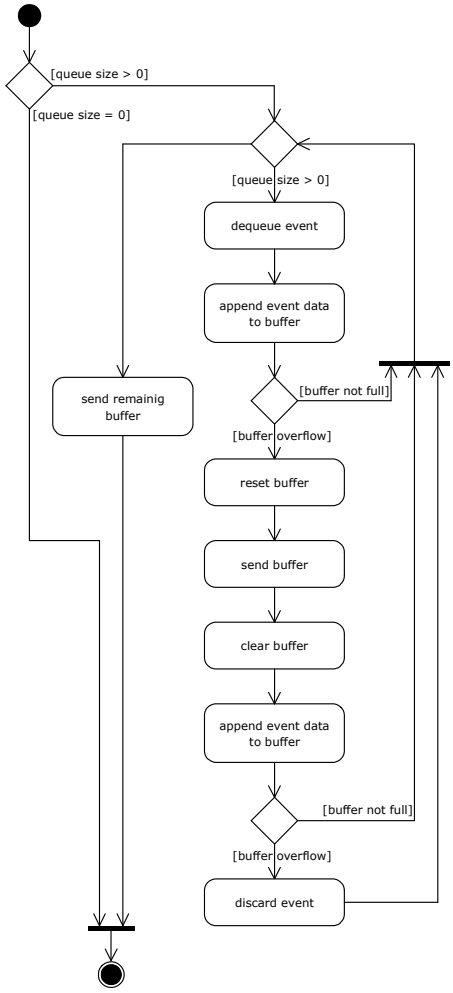


Abb. 2.35 Events in UDP-Pakete schreiben

Die Kontrolle des Verbindungsaufbaus übernimmt im `GameClient` ein zusätzlicher Thread: der `ClientConnectionController`. Er wiederholt das Senden der Verbindungs-Events wenn nötig mehrfach und wartet auf die Antworten des Servers. Wenn die Verbindung besteht, leitet er den `CONNECTIONREPLY`-Event an den `GameClient` und dieser an den `EventHandler` (typischer weise der *ClientController*) weiter.

Ein Klassendiagram des GameClient findet sich in *Abb. 2.37*.

2.4.4.3 GameServer

Der `GameServer` ist prinzipiell genauso aufgebaut, wie der `GameClient` – auch er besitzt zwei Threads zum Lesen und Schreiben von Netzwerkpaketen. Der Hauptunterschied ist, dass er keinen zentralen Ausgangs-Queue besitzt, sondern für jeden Client einen eigenen bereithält. Eine Client wird durch ein `RemotePlayer`-Objekt repräsentiert. Darin sind die Spieler-ID (`playerId`) der Spielername (`playerName`), die Adresse des Clients (`remoteAdress`), sein aktueller Zustand (`connected`, `joined`, `lastEventReceived`, ...), sowie der *Queue* für an diesen Spieler zu sendende Events und noch einige Steuerinformationen enthalten.

Erhält der Server einen Event eines unbekannten Clients, wird nur auf einen `GAMEINFOREQUEST`- oder `CONNECTIONREQUEST`-Event geantwortet. Der `GAMEINFOREQUEST`-Event wird mit einem `GAMEINFOREPLY`-Event beantwortet – dazu merkt sich der Server den Client nur temporär, bis zum nächsten Sendezyklus. Ein `CONNECTIONREQUEST`-Event bewirkt, dass der Client fest in die Spielerliste aufgenommen wird und ein entsprechendes `RemotePlayer`-Objekt erzeugt wird.

Im Hauptthread von `GameServer` wird – zusätzlich zu den schon im `GameClient` vorhandnen Aufgaben – die Verbindung der angemeldeten Spieler überwacht. Dazu wird immer, wenn ein Event empfangen wird, die Empfangszeit im zugehörigen `RemotePlayer` gespeichert. In der Methode `checkPlayerConnection()` wird geprüft, ob die Zeit seit dem zuletzt empfangenen Event größer als der `KEEPALIVE_TIMEOUT` (default: 200 ms) ist – sollte dies der Fall sein, wird ein `KEEPALIVE`-Event gesendet. Antwortet der Client nicht auf die `KEEPALIVE`-Events, wird nach der Zeit `TIMEOUT` (default: 3000 ms) der Client abgemeldet.

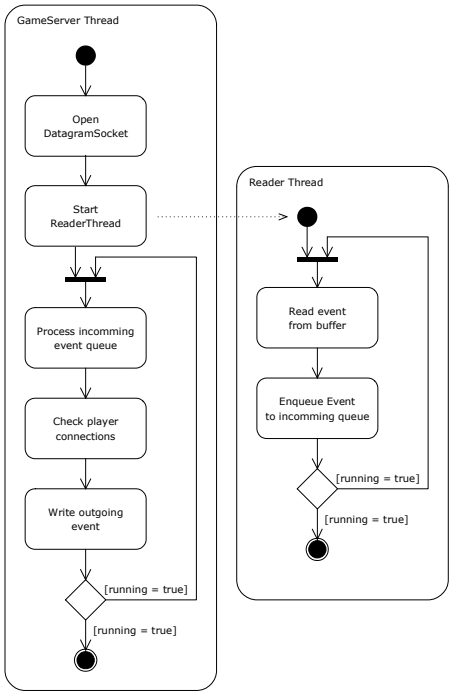


Abb. 2.36 Sende- und Lese-Thread von GameServer

In *Abb. 2.37* sind alle Klassen der **JMPG**-Multiplayer-API in einem Klassendiagramm zusammengefasst.

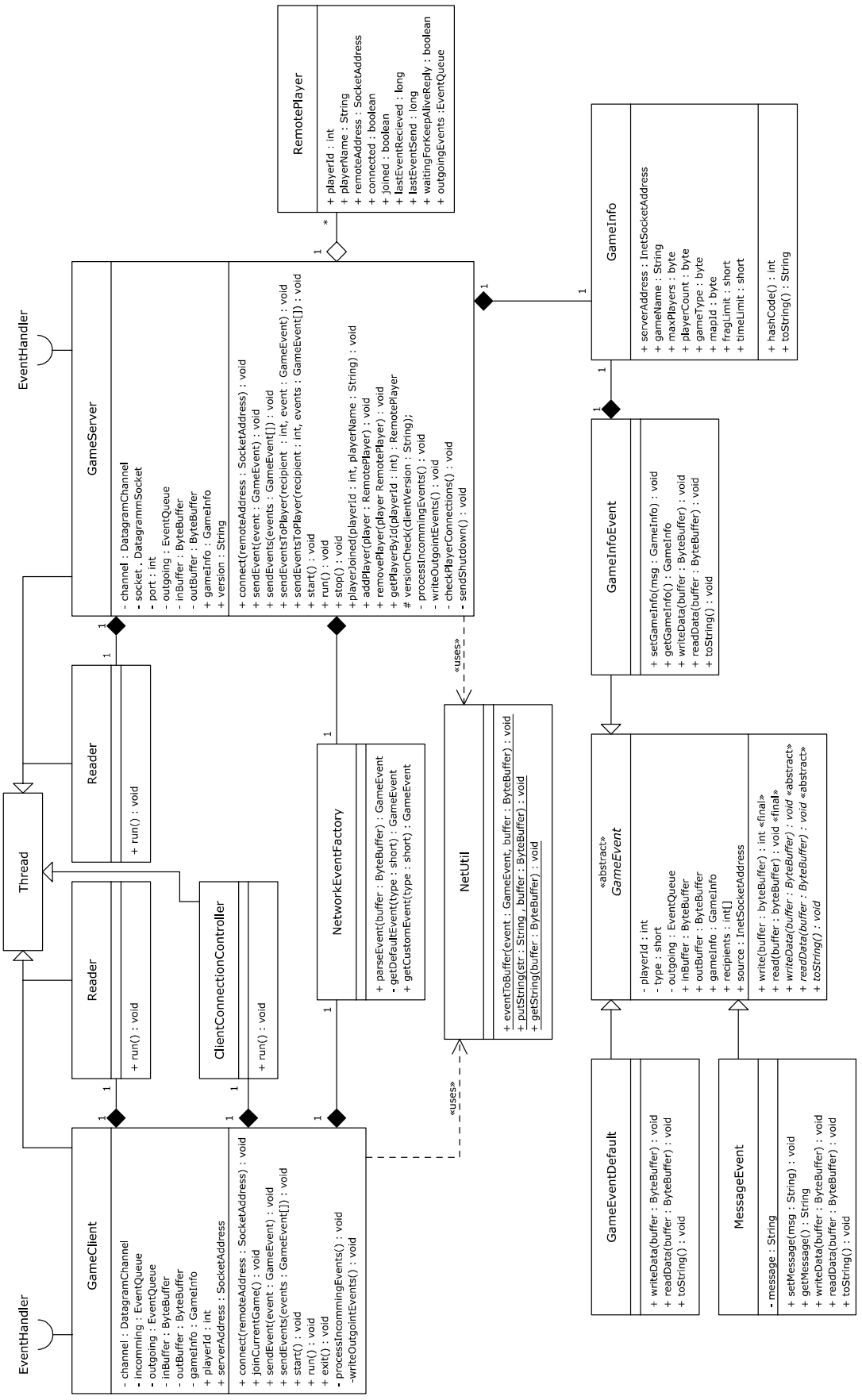


Abb. 2.37 Klassen der JMPG-Multiplayer-API

3 Gravity

Gravity ist ein auf **JMPG** basierendes Spiel mit Netzwerk-Unterstützung. Wir haben es parallel zum **JMPG**-Framework entwickelt, um die dort implementierten Komponenten zu testen und zu verbessern.

Wir begannen mit der Implementierung einer einfachen Einspieler-Version, sobald die ersten Teile des Frameworks einsatzfähig waren.

Während der Entwicklung der Multiplayer-API und dem Umbau von Gravity für Mehrspieler-Unterstützung, kam es teilweise zu größeren Änderungen an der 2D-API. Diese Änderungen haben wir aus Zeitgründen nicht mehr in der Einspieler-Version nachgezogen, weswegen zum Abschluss des Projekts nur die Multiplayer-Version lauffähig ist.

3.1 Spielprinzip

Die Spielidee basiert ursprünglich auf dem Amiga-Spiel *Gravity Force* (*Kingsoft, 1989)*, wurde aber insbesondere für den Multiplayer-Modus stark erweitert.

Das grundlegende Spielprinzip findet sich in vielen Spielen wieder:

Man fliegt mit einem Raumschiff umher und versucht seine Gegner abzuschießen. Dabei kann man das Schiff nur drehen und Gas geben, wobei das Schiff immer in die Richtung beschleunigt, in die das Schiff ausgerichtet ist. Auf das Schiff wirken die Schwerkraft und eine geringe Luftreibung, d.h. es fällt, wenn man nicht dagegen steuert, nach unten und wird in seiner Bewegung nur schwach abgebremst. Die Schwierigkeit besteht darin, dass man einerseits die Schwerkraft ausgleichen und zu schnelle Bewegungen durch Gegensteuern abfangen muss.

Die genauen Regeln von Gravity werden in der Anleitung Kap. 3.4 erläutert.

3.2 Grafik

Die Spielgrafiken wurden mit Lightwave3D und Photoshop umgesetzt.

3.2.1 Spielobjekte

Die Schiffe, Minen, Raketen und Granaten sind einfache 3D Modelle (→ *Abb. 3.1* bis *Abb. 3.5*), die entsprechend animiert und in Auflösungen von 40 x 40 (Schiffe) bzw. 20 x 20 Pixel gerendert wurden. Die Schiffe liegen jeweils in den acht im Spiel verwendeten Spielerfarben vor.

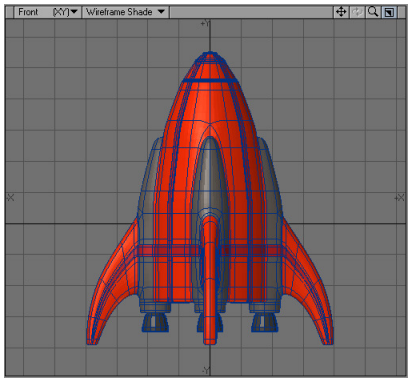


Abb. 3.1 3D-Modell Schiff 1

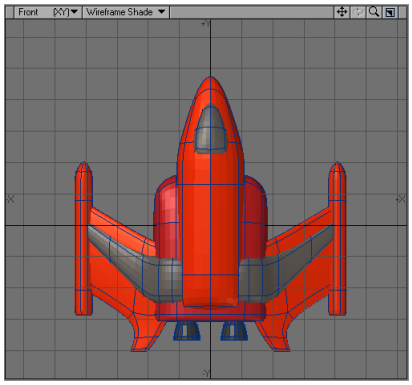


Abb. 3.2 3D-Modell Schiff 2

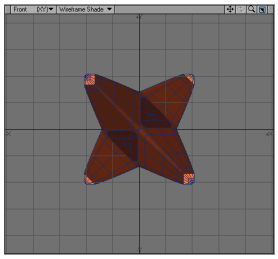


Abb. 3.3 3D-Modell Mine

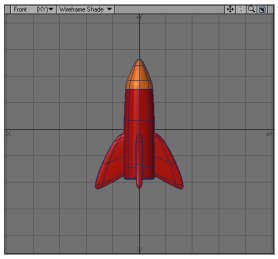


Abb. 3.4 3D-Modell Rakete

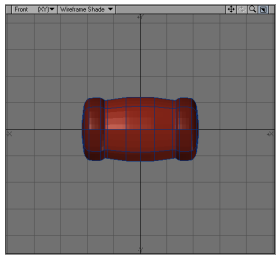


Abb. 3.5 3D-Modell Granate

Objekte, die im Spiel gedreht werden können (Schiffe und Rakete), wurden mit einer 360° Drehung über 64 Einzelbilder bei den Schiffen, bzw. 32 Einzelbildern bei der Rakete (da sie sehr klein ist, genügt diese Genauigkeit) animiert.

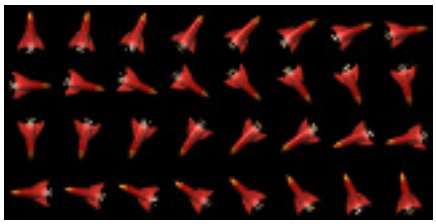


Abb. 3.6 Rotations-Einzelbilder am Beispiel der Rakete (11,25° Schritte)

Die Animationen von Mine und Granate bestehen aus mehreren Teilen. Während der gesamten Animation werden sie so rotiert, dass ihre torkelnde Rotationsbewegung in einer Endlosschleife wiedergegeben werden kann. Beide werden im ersten Teil der Animation zu ihrer Endgröße skaliert (für das Ablegen der Mine bzw. den Abschuss der Granate). Die Mine besitzt zusätzlich noch eine zweite Version der Endlosrotation in der ihre Spitzen blinken (als Warnsignal, wenn sich ein Spieler der Mine nähert).

Für die Explosionsanimationen wurde die Lightwave-Komponente Hypervoxels verwendet. Hypervoxels ist ein Volumen orientiertes Renderingsystem, das unter anderem erlaubt, feuer- und rauchartige Effekte zu erzielen. Indem Punkte aus einem Partikelsystem als Basis für Hypervoxel-Volumen verwendet werden, können sie animiert werden.

Die Basis für die Explosionen bilden zwei identische Partikelemitter, die (etwas zeitversetzt) einige Partikel explosionsartig ausstoßen. Die Partikel des ersten Emitters werden für den Feuerball, die des zweiten für den Raucheffekt verwendet. Die Hypervoxels-Objekte beider Emitter werden so ein- und ausgeblendet, dass der Feuerball am Ende der Explosion in Rauch übergeht.

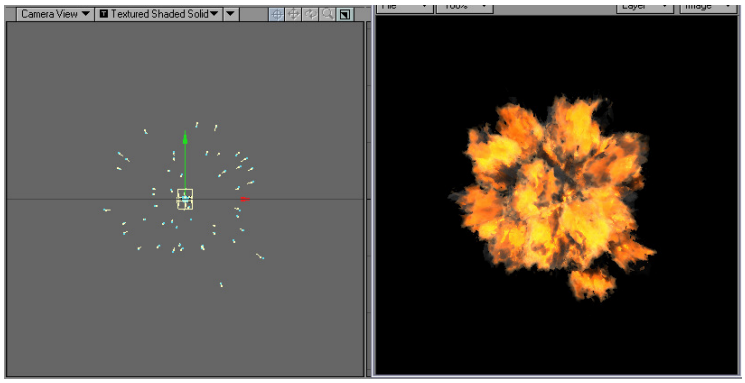


Abb. 3.7 3D-Explosion: Hypervoxel-Partikelemmiter und Ergebnis

Alle 3D-Animationen wurden als einzelne 32-Bit PNG Dateien mit Alphakanal gespeichert, so dass die Anti-Aliasing-Information durch die Transparenz im Spiel erhalten bleibt.

Für die Verwendung im Spiel sollten diese Einzelbilder aber in einer Datei zusammengefasst werden. Für diesen Zweck haben wir eine kleine Java Anwendung geschrieben, die als Kommandozeilen-Anwendung läuft. Sie lädt die angegebenen Einzelbilder und speichert sie in der gewünschten Form als eine PNG-Bilddatei ab.

Option	***Bedeutung***
`-i` *`inputFilenamePattern`*	Name für Quelldateien mit Platzhalter für die Bildnummer in einer der beiden Formen: • `####` - wird mit der Bildnummer ersetzt, die am Anfang mit Nullen aufgefüllt wird. z.B.: `Bild####.png` → `Bild0012.png` • % - wird einfach mit der Bildnummer ersetzt. z.B.: `Bild%.png` → `Bild12.png`
`-o` *`outputFilename`*	Name der Zieldatei
`-start` *`N`* / `-s` *`N`*	Nummer des ersten Bildes
`-end` *`N`* / `-e` *`N`*	Nummer des letzten Bildes
`-frameCount` *`N`* / `-fc` *`N`*	Anzahl der Bilder (falls keine `end`-Option verwendet wird)
`-x` *`N`*	Anzahl der Einzelbilder in einer Zeile im Ausgabebild
`-y` *`N`*	Anzahl der Zeilen im Ausgabebild

Alle anderen Spielgrafiken (Spielobjekte und HUD) wurden in Photoshop erstellt.

Ein Level besteht aus zwei Bitmaps: Eine 256 Farben-Bitmap (mit 1Bit Transparenz), in der die Freiräume aus der Felsstruktur ausradiert sind und eine Schwarz-/Weiß-Bitmap, die als Maske für die Kollisionserkennung dient. Je nach Levelgröße liegt die Größe der Level-Bitmap zwischen 1024X1024 und 4096 x 4096 Pixel.

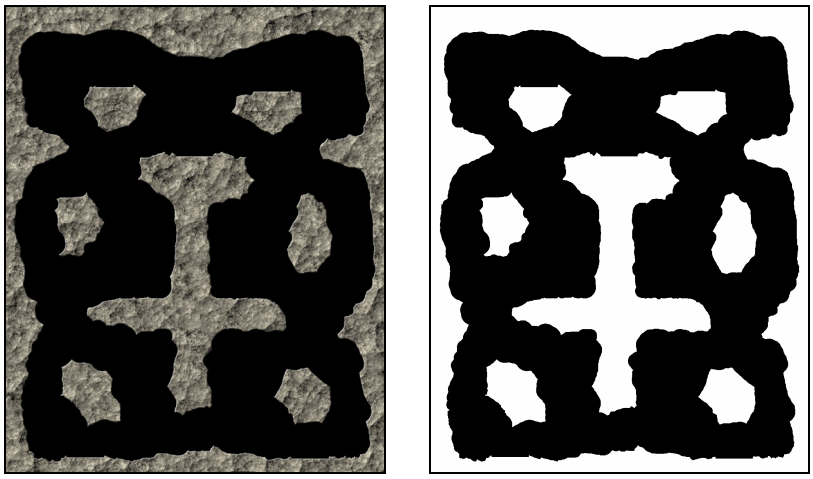


Abb. 3.8 Levelstruktur und 1Bit-Kollisionsmaske

3.2.2 Spielmenü

Alle Menüelemente sind ebenfalls mit Photoshop erstellt. Dabei wurden, um den groben Pixeleffekt zu erzielen, die einzelnen Screens in einem Viertel der Endgröße erstellt, dann ohne Interpolation vergrößert und ein Linienraster übergeblendet, um die 3D-Anmutung der Pixel zu erreichen. Für fast alle Screens wird die gleiche Hintergrund-Bitmap verwendet, wobei jeweils der Titel und die Buttons ausgetauscht werden. Nur bei komplexeren Seiten (z.B. Menü About und Options) wird ein anderer Hintergrund verwendet, der spezielle Elemente enthält. Die Buttons sind pro Menüseite in einer Bilddatei abgelegt. Über den ResourceManager werden sie als Animationen geladen, wobei die Einzelbilder der Animation die beiden Zustände `active` und `selected` der jeweiligen `ButtonSprites` darstellen.

Abb. 3.9 Screenshot des Hauptmenüs von Gravity

3.3 Umsetzung mit JMPG

Das Spiel wurde auf Basis des **JMPG**-Frameworks umgesetzt.

Zum Projektabschluss wurde nur die Multiplayer-Version fertig gestellt, bei der der Server getrennt vom Client gestartet werden muss (über Kommandozeilenoptionen). Server und Client können aber auf dem gleichen Rechner laufen, sodass kein dedizierter Server-PC nötig ist.

Im Folgenden gehen wir nur kurz auf einige wesentliche Aspekte von Gravity ein. Die Diskussion aller Probleme und der Lösungsansätze würden den Rahmen dieses Projektberichts sprengen.

3.3.1 Allgemeiner Aufbau

Client und Server unterscheiden sich grundsätzlich darin, dass der Server den *game state* enthält und keine Grafikausgabe verwendet – der Client stellt im Wesentlichen nur den vom Server erhaltenen *game state* dar.

Die Spiellogik ist im Server-Controller, sowie in den Spielobjekten untergebracht. Er verwaltet den gesamten *game state* und die am Spiel teilnehmenden Spieler. Der Client-Controller verarbeitet die Events des Servers und aktualisiert die Positionen der Sprites in einem `GameScreen`. Um die Netzlast zu verringern werden nur Objekte, deren Bewegungspfad sich ändern (Spieler-Schiffe und Raketen) laufend über das Netzwerk aktualisiert. Für statische oder sich nach vorhersehbarem Muster bewegende Objekte (Geschosse, Minen und Granaten) werden bei ihrer Erzeugung nur die Anfangswerte von Position und Geschwindigkeit übertragen und ihre Bewegung dann clientseitig aktualisiert. Um Fehler zu vermeiden wird in relativ langen Zeitabständen (600 ms) der aktuelle Zustand dieser Objekte gesendet, auch wenn keine Änderungen stattgefunden haben. Fehlen zwei dieser Update-Events (wegen der Unzuverlässigkeit der UDP-Verbindung), wird das Objekt beim Client entfernt.

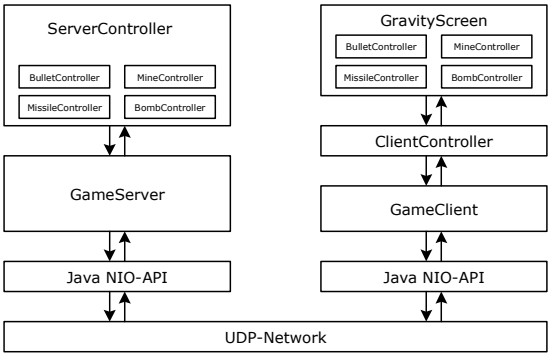


Abb. 3.10 Aufbau der Multiplayer-Version von Gravity

3.3.2 Server-Controller

Die Hauptklasse des Gravity-Servers ist der `ServerController`. Er verwaltet den gesamten Spielzustand und kommuniziert direkt mit einer Instanz von `GameServer`. Die am Spiel teilnehmenden Spieler werden direkt im Controller verwaltet, wobei die Spielerinformationen, die Schiffe und der Status der Schiffe getrennt behandelt werden. Sie werden über die `playerId` der `RemotePlayer`-Objekte des `GameServers` identifiziert. Für die übrigen Spielobjekte existiert jeweils ein Controller, der das Aktualisieren, Erzeugen und Entfernen dieser Objekte übernimmt – die Spielobjekte werden dabei durch Sprites repräsentiert, obwohl sie auf dem Server nicht dargestellt werden. So können die gleichen Controller auf Clientseite eingesetzt werden, um dort die darzustellenden Sprites zu verwalten. Um Server- und Client Version der Sprites zu unterscheiden besitzen sie ein `ghost`-Flag: Auf dem Server werden nur Ghost-Sprites erzeugt.

3.3.2.1 Game loop

Der Server-Controller läuft, nachdem der Level initialisiert wurde, durchgehend in der *game loop*, in der der Spielzustand aktualisiert wird. Diese läuft mit einem festen, beim Start des Servers konfigurierbaren Zeitintervall (Standard: 20 ms). Das Senden des Spielzustand an die Clients kann in größeren Zeitabständen vorgenommen werden (Standard: 40 ms). Dabei ist das Sendeintervall direkt an das Aktualisierungsintervall gekoppelt: Das Senden erfolgt immer am Ende der *game loop*. Durch diese einfache Methode, kann eine genauere Berechnung des Spielzustands (Bewegung der Objekte, Kollisionserkennung) erreicht werden, ohne eine all zu hohe Netzwerklast zu erzeugen. Die Zeiten der Aktualisierungs- und Paket-Sende-Intervalle können beim Starten des Servers als Parameter übergeben werden. Bei Tests stellte sich heraus, dass die Standardwerte für ein flüssiges Spiel sowohl im LAN, als auch über das Internet ausreichend sind. Etwas niedrigere Werte verbessern aber die Reaktionszeit der Steuerung der Schiffe: 10 ms für das Aktualisierung- und 20 ms für das Sendeintervall stellten sich als optimal heraus – die CPU-Auslastung liegt dabei auf einem aktuellen PC mit 2,5 GHz bei ca. 2, maximal 5 % (4 Spieler).

Die `GameServer`-Instanz gibt die von den Clients empfangenen `GameEvents`, an den `GameController` weiter, der sie in einem Queue ablegt. Zu Beginn der *game loop* werden die `GameEvents` ausgewertet. Dies können Events sein, die auftreten, wenn ein Spieler dem Spiel beitritt, oder es verlässt. Hauptsächlich handelt es sich während dem laufenden Spiel aber um Steuerbefehle des Spielers, anhand derer der Zustand der Schiffe aktualisiert wird.

3.3.2.2 Spielerverwaltung

Ein Spieler wird durch ein `PlayerInfo`-Objekt repräsentiert, das die Spieler-ID (`playerId`), den Namen des Spielers (`playerName`), Schiffstyp und -farbe (`shipType`, `shipColorIndex`), sowie seine aktuellen Punktestand (`score`) enthält.

Tritt ein Spieler dem Spiel bei, werden aus dem übermittelten `JoinGameRequestEvent` der Name des Spielers und der gewählte Schiffstyp ausgewertet. Dem Spieler wird eine Farbe zugeordnet, ein entsprechendes `PlayerInfo`-Objekt erzeugt und an alle anderen Spieler in einem `PlayerInfoEvent` vom Typ `PLAYER_JOINED` gesendet. Außerdem werden ein `PlayerShip`-Objekt, das das Spieler-Schiff repräsentiert, und ein `ShipStatus`-Objekt (enthält Informationen über den Schildzustand, vorhandene PowerUps und die Treibstoffmenge) erzeugt. Dem neu beigetretenen Spieler werden der komplette Spielzustand (alle momentan im Spiel vorhandenen Objekte), sowie alle `PlayerInfos` der andern Spieler gesendet. Alle Änderungen an den `PlayerInfo`-Objekten der Spieler werden an alle Spieler kommuniziert (→ `PLAYERINFO_UPDATE`-Event). Der `ShipStatus` dient dem aktualisieren des *HUDs* des Spielers und wird bei Änderungen in einem `ShipStatusEvent` an den Client gesendet.

Die Spieler werden immer anhand der Spieler-ID identifiziert. In jedem vom Spieler erzeugtem Objekt (Geschosse) wird diese ID im `owner`-Property gespeichert, sodass festgestellt werden kann, von wem ein anderer Spieler getroffen wurde. Wird ein Spieler abgeschossen, kann über die `owener`-ID festgestellt werden, von wem er abgeschossen wurde. Dieser Spieler erhält dann Punkte – der Punktestand wird in seinem `PlayerInfo`-Objekt aktualisiert und an die anderen Spieler in einen `PlayerInfoEvent` gesendet.

3.3.2.3 Spiellogik

Der größte Teil, der im `ServerController` geleisteten Arbeit, ist das Aktualisieren des *game states* innerhalb der *game loop*.

Zu Beginn der *game loop* werden die `PlayerMoveEvents` ausgewertet, die Spieler-Schiffe aktualisiert und die Kollisionserkennung ausgeführt.

Das Verhalten der Spieler ist in der Klasse `PlayerShip` implementiert. Die Physiksimulation ist sehr einfach gehalten: Abhängig von den aktuellen Eingaben des Spielers zunächst die Rotationsgeschwindigkeit und der Rotationswinkel aktualisiert. Dann gehen die Auswirkungen der Schwerkraft und (falls der Spieler Gas gibt) der Beschleunigung in Abhängigkeit von der vergangenen Zeit in die Berechnung der der Geschwindigkeit ein. Zuletzt wirkt auf den neuen Geschwindigkeitsvektor die durch die vereinfachte Luftreibung (abhängig von der Geschwindigkeit) auftretende Kraft.

Die neue Geschwindigkeit berechnet sich also zu:

$$\vec{v}' = \vec{v} + \vec{a} * dt + \vec{g} * dt - \vec{v} * d * dt$$

mit: $\vec{v}$: Geschwindigkeit, $\vec{a}$: Beschleunigung,
$\vec{g}$: Schwerkraft, d : Reibungsfaktor,
dt : Zeitintervall

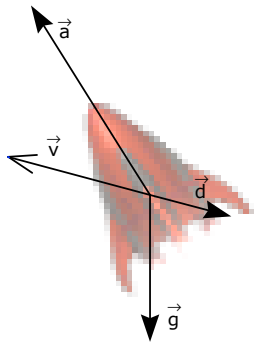


Abb. 3.11 Die auf das Spielerschiff wirkenden Kräfte

Das Aktualisieren und die Verwaltung der anderen Spielobjekte (Geschosse) erfolgt in den entsprechenden Controllern (`BulletController`, `MineController`, `MissileController` und `BombController`) . Die Controller stellen Methoden für das Erzeugen, Entfernen und Aktualisieren der Objekte zur Verfügung und liefern (falls nötig) die entsprechenden `ObjectUpdateEvent`-Objekte, die der `ServerController` an die Clients schickt.

Bei der Kollisionserkennung wird für alle Objekte geprüft, ob sie mit anderen Objekten oder dem Level kollidieren. Die Levelstruktur ist hierfür in einer `TiledBitmapMask` hinterlegt. Tritt eine Kollision auf, wird entsprechend reagiert. Zum Beispiel nimmt ein Spieler-Schiff Schaden, wenn es von einer Rakete getroffen wird. Die Rakete wird dann in den Zustand `exploding` versetzt, dem Schiff wird ein entsprechender Betrag vom Schild abgezogen.

Am Ende der *game loop* werden (falls die Paket-Sende-Zeit überschritten ist) für alle Objekte, deren Zustand sich geändert hat, die entsprechenden `GameEvents` gesammelt und an die Clients gesendet. Dabei liefern die einzelnen Objekt-Controller, die für die von ihnen verwalteten Objekte – die Änderungen der Spieler-Schiffe werden direkt vom `ServerController` ausgewertet.

3.3.3 ClientController

Der Gravity-Client verwendet für die Darstellung des Spiels einen `GameScreen` (`GravityScreen`), der von der `Game`-Klasse aktualisiert und gerendert wird. Die GameEvents werden vom `ClientController` verwaltet. Er kümmert sich um den Beitritt des Spielers zu einen Spiel, das Laden der Level-Grafiken und aktualisiert die Sprite-Positionen im `GravityScreen`.

Prinzipiell werden clientseitig nur die Positionsdaten, die der Server liefert, verarbeitet und Spielobjekte entsprechend der erhaltenen Events erzeugt oder entfernt. Spielerschiffe werden direkt im ClientController verwaltet. Für die übrigen Objekte (Geschosse) werden die gleichen Controller, die auch im Server eingesetzt werden, verwendet. Dabei werden, wie oben angesprochen, voraussehbare Bewegungen der Objekte (Bullets, Minen und Bomben)

berechnet – alle anderen Objekte werden anhand der erhaltenen `ObjectUpdateEvents` aktualisiert. Zum Beispiel haben Minen, wenn sie gelegt werden, eine Anfangsgeschwindigkeit, die von der aktuellen Geschwindigkeit des Spieler-Schiffs abhängig ist. Danach wird die Bewegungsrichtung nicht mehr geändert. Die Mine wird einfach nur nach einer festen Formel, die unabhängig von äußeren Einflüssen ist, abgebremst („Luftreibungswiederstand“) – d.h. dieses Abbremsen kann auf dem Client berechnet werden. Nur wenn die Mine explodiert oder während ihrer Anfangsbewegung eine Wand berührt ändert sich ihr Zustand, der dann vom Server gesendet wird. Um zu vermeiden, dass beim Verlorengehen von Events das Sprite einer bereits explodierten Mine nicht entfernt wird, sendet der Server in Abständen von 600 ms den aktuellen Zustand der Mine. Werden für über 1200 ms keine solchen Events empfangen, wird die Mine einfach entfernt.

3.3.4 GameEvents

In Gravity wird die Palette der von **JMPG** zu Verfügung gestellten `GameEvents` erweitert. Für das Erzeugen der passenden Event-Objekte beim Lesen aus der Netzwerkschicht, ist die `GravityEventFactory` verantwortlich.

Tabelle 3.1 zeigt eine Übersicht, über die in Gravity verwendeten Eventtypen. Sie sind im Interface `GravityEvents` deklariert.

Tabelle 3.1 Gravity GameEvnts

Eventtyp	*Wert*	*Beschreibung*	*Event-Klasse*
`TYPE_JOINGAMEREQUEST`	-6	Anfrage, einem Spiel beizutreten	`JoinGameRequestEvent`
`TYPE_JOINGAMEREPLY`	-7	Antwort bei erfolgtem Spielbeitritt	`PlayerInfoEvent`
`TYPE_PLAYER_JOINED`	-9	Benachrichtigung, dass ein neuer Spieler dem Spiel beigetreten ist	`PlayerInfoEvent`
`TYPE_PLAYER_LEFTGAME`	-12	Benachrichtigung, dass ein Spieler das Spiel verlassen hat	`PlayerInfoEvent`
`TYPE_PLAYERINFO_UPDATE`	9	Aktualisierung der Spielerinformation	`PlayerInfoEvent`
`TYPE_PLAYER_MOVE`	11	Bewegungsinformation eine Spielers	`PlayerUpdateEvent`
`TYPE_PLAYER_UPDATE`	11	Spielerschiff aktualisieren	`PlayerUpdateEvent`
`TYPE_OBJECT_UPDATE`	12	Spielobjekt erzeugen	`ObjectUpdateEvent`
`TYPE_OBJECT_CREATE`	13	Spielobjekt aktualisieren	`CreateObjectEvent`
`TYPE_OBJECT_REMOVE`	14	Spielobjekt entfernen	`RemoveObjectEvent`
`TYPE_REQUESTGAMESTATE`	17	Anfordern des aktuellen Spielzustands	`GameEventDefault`
`TYPE_GAMESTATE_DONE`	18	Spielzustand vollständig übertragen	`GameEventDefault`
`TYPE_GAMESTATE_ACK`	19	Spielzustand wurde empfangen	`MessageEvent`
`TYPE_READY`	20	Initialisierung des Spiels abgeschlossen (vom Client)	`GameEventDefault`
`TYPE_STARTGAME`	21	Spiel starten	`GameEventDefault`
`TYPE_LEVELLOADED`	22	Client hat Leveldaten geladen	`GameEventDefault`
`TYPE_RESETLEVEL`	23	Zurücksetzen des Levels	`GameEventDefault`
`TYPE_RESETLEVEL_ACK`	24	Bestätigung auf `RESETLEVEL`	`GameEventDefault`
`TYPE_SHIPSTATUS`	30	Aktualisierung des Schiffstatus	`ShipStatusEvent`
`TYPE_CHATMESSAGE`	55	Chatnachricht	`ChatMessageEvent`
`TYPE_SERVERMESSAGE`	56	Servernachricht	`MessageEvent`

Um die Netzwerkauslastung zu verringern werden bei größeren, oft verwendeten Events nicht alle Daten, sondern nur die, die sich seit der letzten Aktualisierung geändert haben, übermittelt. Dafür besitzen sie als zusätzliches Feld ein Byte, in dem die Information, welche Daten im Event-Paket vorhanden sind kodiert ist (`dataFlags`). Jedes Bit steht für ein Informationsteil des Events und wird, wenn die Daten des Events über ihre *Setter*-Methoden geändert werden, auf 1 gesetzt. Nur wenn das entsprechende Bit gesetzt ist, werden die Felder geschrieben (→ *Abb. 3.12*). Beim Lesen des Events wird zunächst das `dataFlags`-Byte ausgewertet und dann die vorhandenen Felder verarbeitet. In *Abb. 3.12* ist als Beispiel das Serialisieren eines `ObjectUpdateEvents` schematisch dargestellt. Nur die Positions- und Rotationseigenschaften sind geändert, also sind das erste und dritte Bit der `dataFlags` gesetzt. Die Properties `objectType` und `objectId` werden immer geschrieben, danach nur das `dataFlags`-Byte und die darin markierten Informationen.

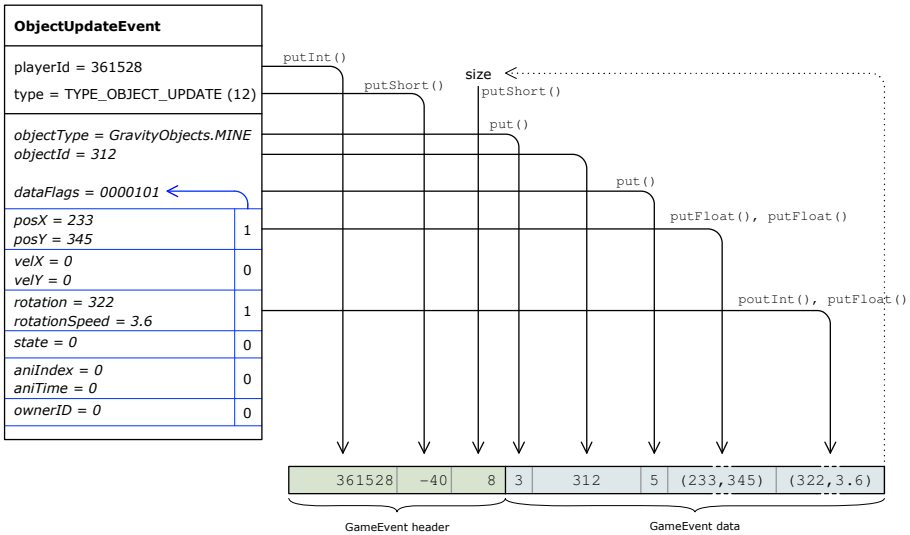


Abb. 3.12 Serialisierung eines `ObjectUpdateEvents`, der nur Positions- und Rotationsdaten enthält

3.3.5 Spielmenü

Die Menüfunktionalität stellt die Klasse `GameMenuScreen` zur Verfügung. Dieser `GameScreen` enthält einen `MenuLayer` und einen `MenuInputManager` (siehe *Kap. 2.3.4.1*).

Menüs sind immer gleich aufgebaut und können sowohl mit Maus als auch Tastatur bedient werden. Sie enthalten ein Hintergrundbild (`ImageLayer`), ein Sprite für den Titel des Menüs, sowie eine Liste von Buttons (`ButtonSprite`). *Abb. 3.9* zeigt ein Beispiel-Screenshot aus dem Menü von Gravity:

Der Hintergrund ist für jedes Menü identisch.

Der Titel (*MAINMENU*) wird als `Image` im oberen Bereich angezeigt.

Die sechs Buttons werden in festen Abständen als Liste unter den Titel gesetzt. Sie werden ausgewählt, wenn sich die Maus über ihnen befindet (hier der Button *Multiplayer*). Navigiert man mit den Pfeiltasten (hoch / runter) wird der Mauszeiger versteckt und der nächste (aktive) Button ausgewählt.

Die von den Buttons auszuführenden Aktionen (z.B. Wechsel zur nächsten Menüseite) beim Klicken oder Enter-Taste Drücken eines ausgewählten Buttons, werden vom `MenuController` verwaltet. Das Drücken der *ESC*-Taste führt auf die vorherige Menüseite zurück.

3.3.6 Test

Wir haben das Spiel während der Entwicklung mehrfach im Netzwerk gestestet und dabei Fehler beseitigt sowie das *game play* angepasst.

Der erste größere Dauertest von Gravity fand bei der Präsentation auf der *MediaNight* statt. Dort wurde intensiv an insgesamt 6 PC gespielt. Das Spiel lief über mehrere Stunden hinweg fehlerfrei mit guter Performance.

Spätere Tests mit bis zu 8 Spielern verliefen auch problemlos.

3.4 Gravity Anleitung

Gravity wird als ZIP-Datei ausgeliefert und kann in ein beliebiges Verzeichnis entpackt werden – die im ZIP enthaltene Verzeichnisstruktur muss dabei beibehalten werden.

3.4.1 Starten des Spiels

Sowohl der Server, als auch das Spiel selbst werden über die Batch-Datei Gravity.bat gestartet. Folgende Kommandozeilenoptionen stehen zur Verfügung:

Option	***Bedeutung***
`-fullscreen N (-fs N)`	Startet das Spiel im Vollbildmodus (1024x768). Mit dem Parameter N wird die Bildwiederholrate festgelegt (default: 60).
`-frameTime N (-ft N)`	Legt die Zeit für einen Durchlauf der *game loop* fest. Sie sollte beim Client der `packetSendTime` des Servers entsprechen.
`-server`	Startet Gravity im Server-Modus
`-packetSendTime` *N* / `-pst` *N*	Nur Server: Legt das Intervall für das Senden der Pakete an die Clients fest.
`-map N`	Nur Server: Zu ladende Levelnummer: 1 ≤ N ≤ 4

Beim Starten des Servers sollte der Wert von `packetSendTime` ein Vielfaches der angegebenen `frameTime` sein (ideal: `-ft 10 -pst 20`) – die Clients sollten die `packetSendTime` des Servers als `frameTime` verwenden.

Client und Server können auf demselben PC gestartet werden.

Der Server besitzt kein Userinterface und läuft nur in einem Konsolenfenster.

Nach dem Starten des Spiels gelangt man über die Menüpunkte `Multiplayer` → `Join Network Game` auf die Menüseite, um einem Netzwerkspiel beizutreten. Hier kann die Server Adresse (IP oder Host-Name) und der Spielername angegeben, sowie der Schiffstyp ausgewählt werden. Texteingaben müssen mit `[ENTER]` bestätigt werden – `[ESC]` verwirft die Änderungen. Durch Auswahl von `Join game` wird das Spiel gestartet.

3.4.2 Tastaturbelegung

Über [F1] kann zwischen Standard- und Alternativsteuerung umgeschaltet werden.

Standardsteuerung

[→], [←]	Schiff drehen
[Space]	Gas geben (Beschleunigen)
[↑]	Schießen
[↓]	Alternativ-Waffe abfeuern (Rakete, Mine, Granate)

Alternativsteuerung

[→], [←]	Schiff drehen
[↑]	Gas geben (Beschleunigen)
[Space]	Schießen
[V]	Alternativ-Waffe abfeuern (Rakete, Mine, Granate)

Weitere Befehle

[T]	Chat: Am unteren Bildschirmrand erscheint ein Texteingabefeld, in dem eine Nachricht eingegeben werden kann. Mit [RETURN] wird sie an alle andere Spieler gesendet - [ESC] bricht den Vorgang ab.
[N]	Die Namen der Spieler werden über den Spielerschiffen eingeblendet
[ESC]	Spiel verlassen

3.4.3 Spielen

Die Spieler steuern ihre Raumschiffe durch ein Höhlensystem und versuchen sich gegenseitig abzuschießen. Für jeden Abschuss erhält man Punkte.

Steuerung

Das Schiff kann sich nur drehen und in die Richtung, in die es gedreht ist, beschleunigen. Dabei wirkt die Schwerkraft auf das Schiff – d.h. es fällt, wenn nicht dagegen gesteuert wird, nach unten.

Im linken unteren Bereich des Bildschirms wird der Status des Schiffs angezeigt: der aktuelle Zustand des Schutzschilds (*shield*) und die vorhandene Treibstoffmenge (*fuel*). Nimmt das Schiff Schaden, wird das Schutzschild reduziert – ist es auf null gesunken, wird das Schiff zerstört und entsteht, nach kurzer Verzögerung, an einer zufällig ausgewählten Station neu. Der Treibstoff wird durch das Gasgeben verbraucht. Ist der Tank leer, kann das Schiff nicht mehr gesteuert werden. Der Tank kann durch Landen auf einer Station wieder aufgeladen werden.

Steuert man das Schiff gegen den Fels oder andere Objekte im Level, nimmt es abhängig von seiner Geschwindigkeit Schaden. Um auf Stationen (mit Turm) und Plattformen (ebene, graue Flächen) zu landen, muss das Schiff einigermaßen gerade nach oben ausgerichtet sein und langsam nach unten fallen (d.h. die Schwerkraft muss durch Gegensteuern ausgeglichen werden). Das kleine Schiffsymbol in der Anzeige *landing speed* zeigt an, ob mit der aktuellen Ausrichtung und Geschwindigkeit eine Landung möglich ist.

Schiffstypen

Es stehen zwei Schiffstypen zur Auswahl, die sich hauptsächlich durch ihre Geschwindigkeit und Navigierbarkeit unterscheiden:

Typ 1 Dieses Schiff ist relativ langsam und träge und daher einfach zu steuern. Es besitzt eine Standardkanone, die mit einer konstanten Rate von 6 Schüssen pro Sekunde feuern kann.

Typ 2 Dieses Schiff fliegt wesentlich schneller und ist wendiger als der Schiffstyp 1 – es ist dadurch aber (besonders in engen Leveln) schwieriger zu kontrollieren. Es besitzt ein schwächeres Schutzschild (es nimmt bei Treffern 25% mehr Schaden) und hat einen wesentlichen höheren Treibstoffverbrauch. Dafür feuern die Standardkanonen zwei Geschosse parallel in schnell aufeinander folgenden Salven.

PowerUps

Um das Schutzschild zu erneuern kann das *Shild-PowerUp*  eingesammelt werden. Es fügt 50 Punkte zum Schutzschildzustand hinzu (Maximalwert: 100).

Das Schiff verfügt immer über den Standardschuss, der je nach Schiffstyp mit unterschiedlicher Frequenz abgefeuert werden kann. Um die Alternativwaffen zu erhalten, müssen die entsprechenden *PowerUps* eingesammelt werden. Nach dem Einsammeln eines *PowerUps* wird im linken unteren Bildschirmrand die Anzahl der zur Verfügung stehenden Geschosse angezeigt (maximal 4). Diese können in Abständen von 2 Sekunden benutzt werden.

 Mine — Minen können beliebig im Level gelegt werden, um z.B. enge Gänge zu blockieren. Sie werden hinter dem Schiff abgelegt.

Eine Mine wird ausgelöst, wenn sie von einem Schiff berührt oder abgeschossen wird. Die Explosion der Mine besitzt einen bestimmten Einflussradius, innerhalb dem sie Schaden verursacht und durch ihre Druckwelle wegschleudert. Minen haben eine beschränkte Lebensdauer (ca. 70 Sekunden) nach der sie automatisch explodieren.

 Granate — Granaten werden in Blickrichtung des Schiffs abgeschossen, wobei ihre Initialgeschwindigkeit von der aktuellen Geschwindigkeit des Schiffs abhängt. Sobald die Granate in ihrer Bewegung zum Stehen kommt, oder sich in der Nähe eines Schiff oder der Wand befindet, explodiert sie. Ihre Explosion hat, ähnlich wie die Mine, einen relativ großen Einflussradius, ihre Wirkung ist aber stärker als die der Mine.

 Rakete — Die Raketen in Gravity sind zielsuchend: sofort, nachdem sie abgeschossen wurden, suchen sie sich in ihrem Sichtbereich das nächste gegnerische Spielerschiff als Ziel und verfolgen dieses - konnte kein Ziel ausgemacht werden fliegt sie einfach geradeaus. Der Sichtbereich beträgt +/- 90° in Flugrichtung der Rakete mit einem Radius von 450 Pixel. Wenn das aktuelle Zielschiff den Sichtbereich verlassen hat, sucht sich die Rakete ein neues Ziel. Sie hat wie die Mine eine Lebensdauer ca. 70 Sekunden.

Punkte

Im Spiel werden für Abschüsse Punkte vergeben. Ein Spieler erhält

- 4 Punkte für den Abschuss eines anderen Spielers mit dem Standardschuss,
- 3 Punkte für den Abschuss eines anderen Spielers mit Alternativwaffen,
- 1 Punkt, wenn ein gegnerisches Schiff durch Rammen zerstört wird (0 wenn er dabei selber stirbt),
- -1 Punkt, wenn er sich durch seine eigenen Geschosse selber zerstört,
- -1 Punkt für das Zerschellen am Fels oder misslungene Landungen.

Der Punktestand der Spieler wird im rechten oberen Bereich des Bildschirms angezeigt.