Spieleprogrammierung in Java

Gliederung

1. Vorwort

“Warum Spieleprogrammierung?” Diese Frage ist einfach zu beantworten, wenn man selbst gerne spielt und vielleicht auch schon ab und zu davon geträumt hat ein “eigenes” Spiel zu entwickeln: es macht einfach großen Spaß! Aber wenn man sich etwas intensiver mit dem Thema auseinandersetzt, dann wird man recht bald feststellen, daß sich zum Spaß auch schnell die Arbeit gesellt, denn Spiele sind, auch wenn sie auf den ersten Blick oft einfach erscheinen, alles andere als einfach! Genaugenommen sind Spiele kleine Simulationen einer realen Welt und verhalten sich als solche nach einem klar definierten und abgegrenzten Logik- und Regelwerk.

Die Spieleprogrammierung ist daher fachlich betrachtet nicht uninteressant! Sie vereint in sich nicht nur verschiedene Disziplinen wie *Audio*, *Grafik*, *Animation* und *Logik* sondern stellt den Programmierer auch im Bereich *Performance* und *Timing* vor einige Herausforderungen. Soll ein Spiel dann auch noch netzwerkfähig und über das Internet in Echtzeit spielbar sein, muß man sich mit dem Thema *Synchronisation* außeinandersetzen und dabei natürlich auf die Unzuverlässigkeit des Internets Rücksicht nehmen. Dieses ist nämlich, aufgrund ständiger unvorhersehbarer Verzögerungen im Datentransfer, für eine flüssige *Synchronisation* nicht gerade optimal!

2. Grundlagen der Spieleprogrammierung in Java

“Kann man überhaupt Spiele in Java programmieren?” Das ist die wichtigste Frage, die man klären muß, bevor man mit der Entwicklung eines Spiels in Java beginnt! Die Antwort dazu ist “grundsätzlich ja, aber natürlich nicht alles”.

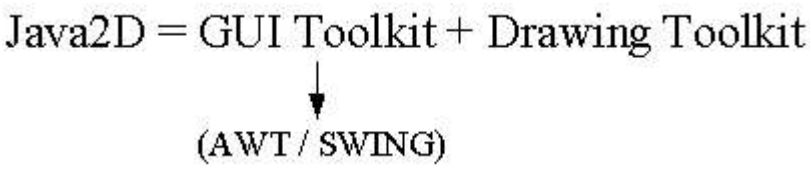
Java ist mittlerweile technisch weit fortgeschritten und obwohl es in manchen Bereichen noch etwas hinter der Performance von C/C++ oder anderen Sprachen zurückliegt, wird es doch langsam mehr und mehr eine ernste Alternative für die Spieleentwicklung. Durch zahlreiche interne und externe Bibliotheken und APIs kann man auch unter Java mittlerweile auf Hardwarebeschleunigung durch *OpenGL* oder *DirectX* zurückgreifen und so performance-lastige Funktionen auf die darunterliegende Hardware auslagern. Es gibt außerdem Bibliotheken, die ein direktes Ansteuern der Eingabegeräte über *DirectInput* unterstützen und so ein Event-System-unabhängiges Abfragen von Eingabegeräten (wie z.B. Joysticks oder Gamepads) ermöglichen.

Es gibt zahlreiche Studien, die bewiesen haben, daß Java in der Ausführung von Programmcode im Vergleich zu C/C++ langsamer ist. Bei manchen Funktion benötigt Java etwa 130-150% der Zeit um die gleichen Rechenoperationen durchzuführen, während es bei anderen sogar schneller ist und nur etwa 90% der Rechenzeit benötigt. Im Durchschnitt kann man dann mit etwa 20-30% mehr Rechenzeit im Vergleich zu C/C++ rechnen. Diese Zahlen sind jedoch stark abhängig von der verwendeten JVM Version und wurden mit jeder neuen Version von Java immer etwas besser. Wenn man diese Entwicklung sieht und auch noch im Hinterkopf behält, daß bisher nur etwa 50% der Optimierungsmöglichkeiten in den JVMs genutzt wurden, kann man davon ausgehen, daß Java auch in Zukunft noch stark an Performance gewinnen wird.

Aber auch wenn Java um vielleicht 30% langsamer als C++ ist, reicht die Leistung immer noch aus, um interessante Spiele zu erstellen, die auch aktuellen Erwartungen an die Qualität von Grafik und Sound gerecht werden können. Natürlich können keine Highend-Spiele erstellt werden, die durch hardwarenahe Programmierung und hardwarespezifische Optimierung jedes bischen Performance nutzen. Aber dafür ist Java auch nicht gedacht! Der Vorteil bei der Verwendung von Java ist ein hohes Abstraktionsniveau durch Objektorientierung und die Verwendung funktionsreicher und “intelligenter” APIs, um so eine erhöhte Produktivität und Wiederverwendbarkeit zu erziehlen.

2.1. Java 2D

Java bietet mit *Java2D* ein mächtiges und flexibles Framework für die Darstellung von geräte- und auflösungsunabhängiger 2D Grafik. Es ist eine Erweiterung des *Abstract Windowing Toolkits (AWT)* und stellt eine Sammlung von Klassen, für die Arbeit mit geometrischen Formen, Text und Rasterbildern, zur Verfügung. Als Bestandteil der *Java Foundation Classes (JFC)* ist *Java2D* automatisch in jeder aktuellen JVM vorhanden.



Zusammen mit *Java Swing* bietet *Java2D* alle Grundlagen für die Darstellung und Verarbeitung von Grafik. Eine Einführung zu *Java Swing* und *Java2D* befindet sich im Anhang.

Als generelle, vielseitige und mächtige API zur Darstellung von Grafik und GUI sind *Java Swing* und *Java2D* natürlich nicht auf Spiele optimiert und deshalb weniger performant als andere spezialisierte Bibliotheken. Seit der Java Version 1.4 sind jedoch zahlreiche Grafikfunktionen von *Java Swing* und *Java2D* hardwarebeschleunigt und somit gut für die Spieleprogrammierung einsetzbar. Insbesondere das Zeichnen von Lienen und das Füllen von Rechtecken, sowie das Zeichnen / Kopieren von Bildern sind optimiert. 1-Bit Transparenz von Bildern (also komplett durchsichtige Bereiche) ist ebenfalls hardwarebeschleunigt, während teilweise Transparenz rein softwarebasiert berechnet wird.

In Java Version 1.5 soll die Hardwarebeschleunigung noch weiter ausgebaut werden. Da es aber zu Beginn unserer Arbeit noch keine (stabile) Version von Java 1.5 gab, können wir darüber auch noch keine Angaben machen. Für unsere Arbeit haben wir Java 1.4.2_04 verwendet und alle Aussagen sind daher auf dieser Version von Java basiert.

a) Aktives Rendern

Will man eine Anwendung schreiben, die den Bildschirminhalt häufig verändert und neuzeichnet, dann wird man früher oder später zwangsläufig an die Grenzen des *Event-Handling-Systems* stoßen. In *AWT* / *Swing* wird nämlich ein Neuzeichnen von Fenstern und Komponenten über die Methode `repaint()` angefordert. Diese Methode gibt dann die Anforderung in Form eines *Repaint-Events* an den *Main-Thread* des *GUI-Toolkits* weiter. Dieser ist jedoch zuständig für das Zeichnen (Rendern) aller *GUI Komponenten* und somit ziemlich beschäftigt. Wird nun ein `repaint()` in häufigen Intervallen (z.B. 25 mal pro Sekunde) durchgeführt, dann führt das zu einer erhöhten Belastung des *Event-Systems* und des *Main-Threads*. Dies erzeugt nicht nur einen erhöhten Verwaltungsaufwand für Events und so eine verzögerte Verarbeitung von Benutzereingaben wie z.B. Tastatur- und Mouse-Events, sondern auch eine insgesamt langsamere Reaktion der gesamten GUI.

Um dies zu vermeiden, sollte das Neuzeichnen in einem eigenständigen Thread unter Umgehung des Swing Repaint-Modells erfolgen. Hierfür wird die entsprechende Methode `paintComponent (Graphics g)` überschrieben und eine zusätzliche Methode `renderScreen()` erstellt, die dann in regelmäßigen Abständen vom Render-Thread aufgerufen wird. Diese Methode ist nun für das Rendern der Komponente zuständig und verwendet hierfür das *Graphics*-Objekt der Komponente.

Der entsprechende Quellcode dazu könnte in ungefähr so aussehen:

```
import java.awt.*;
import javax.swing.JComponent;

public class MyComponent extends JComponent
{
  private Image myImage;

  // more code to write here ...

  public void paintComponent(Graphics g)
  {
    // do nothing here ...
  }

  public void renderScreen()
  {
    // get Graphics object from component
    Graphics g = getGraphics();

    // perform rendering
    g.drawImage(myImage,0,0,this);
    g.drawLine(0,0,10,20);
    // ...
  }
}


public class Renderer extends Thread
{
  private MyComponent myComponent = new MyComponent();

  // more code to write here ...

  public void run()
  {
    while(true)
    {
      // render component
      myComponent.renderScreen();

      // rest a bit and give time to other Threads
      try
      {
        Thread.sleep(20L);
      }
      catch (InterruptedException ex) {}
    }
  }
}
```

Double Buffering

Führt man diesen Code aus, dann wird man feststellen, daß das Rendern zwar funktioniert, aber nicht ohne ein gelegentliches Flackern des gerenderten Bildes. Das Flackern tritt nämlich dann auf, wenn ein Bereich des Bildschirms verändert wird, während der Monitor gerade versucht diesen Bereich auf die Bildröhre zu projezieren. Diesen Effekt kann man jedoch umgehen, indem man das zu zeichnende Bild zuerst in einem nicht-sichtbaren Speicherbereich erstellt und dann diesen Bereich anschließend in den sichtbaren Bereich kopiert. Das Zeichnen selbst nimmt nämlich deutlich mehr Zeit in Anspruch als das Kopieren das Speicherbereiches!

Diese Methode des Renderns nennt man *Double Buffering*. In Java verwenden wir für das *Double Buffering* ein eigens erstelltes *BufferedImage*, daß die Größe der zu rendernden Oberfläche hat und im Folgenden als *BackBuffer* bezeichnet wird. Der *BackBuffer* wird über die Methode *createImage (int width, int height)* der Komponente erstellt und verhält sich ansonsten wie ein normales *Image*-Objekt. Das Kopieren des *BackBuffers* in einen sichtbaren Bildschirmbereich erfolgt daher über die *drawImage(...)* Methode des zugehörigen Graphics-Objektes.

Rendert man nun die Komponente über *Double Buffering*, dann erfolgt das Rendern in 3 Schritten:
1. Erzeuge einen *BackBuffer*, falls dieser noch nicht vorhanden ist
2. Verwende das *Graphics*-Objekt des *BackBuffers* für alle Renderoperationen
3. Zeichne den *BackBuffer* auf den Bildschirm über das *Graphics*-Objekt der Komponente

Der entsprechende Quellcode dazu könnte in ungefähr so aussehen:

```
import java.awt.*;
import javax.swing.JComponent;

public class MyComponent extends JComponent
{
  private Image backBuffer;
  private Image myImage;

  // more code to write here ...

  private void createBackBuffer()
  {
    backBuffer = createImage(getWidth(),getHeight());
  }

  public void paintComponent(Graphics g)
  {
    updateScreen();
  }

  public void renderScreen()
  {
    // if backBuffer doesn't exist, create one
    if (backBuffer == null) createBackBuffer();

    // get Graphics object from backBuffer
    Graphics g = backBuffer.getGraphics();

    // render screen on backBuffer
    g.drawImage(myImage,0,0,this);
    g.drawLine(0,0,10,20);
    // ...
  }

  public void updateScreen()
  {
    Graphics g = getGraphics();
    if (g != null) // component already visible?
    {
      // is there a backBuffer to draw?
      if (backBuffer != null) g.drawImage(backBuffer, 0, 0, null);
      else
      {
        // if not, create one and render on it
        createBackBuffer();
        renderScreen();
      }
    }
  }
}
```

```
public class Renderer extends Thread
{
  private MyComponent myComponent = new MyComponent();

  // more code to write here ...

  public void run()
  {
    while(true)
    {
      myComponent.renderScreen(); // render component
      myComponent.updateScreen(); // draw backBuffer to screen

      // rest a bit and give time to other Threads
      try
      {
        Thread.sleep(20L);
      }
      catch (InterruptedException ex) {}
    }
  }
}
```

Führt man diesen Code jetzt aus, dann stellt man fest, daß das Flackern nun nicht mehr auftritt.

Interessant ist an dieser Stelle vielleicht noch, daß Java Swing Komponenten schon standardmäßig einen internes *Double Buffering* verwenden, um so ein Flackern zu vermeiden. Dieses sollte man, wenn man Komponenten nach dem oben beschriebenen Verfahren selbst rendert, über die Funktion `setDoubleBuffered(false)` deaktiveren, um so einen Performanceverlust durch ein zusätzliches (unnötiges) Puffern durch die Swing Komponente zu vermeiden!

b) Bilder und Sprites

Programmiert man in Java, dann macht man sich eigentlich zuerst einmal keine Gedanken darüber wo und wie ein Objekt gespeichert wird, denn diese Arbeit nimmt uns ja glücklicherweise die JVM ab! Und da *Image*-Objekte ebenfalls Objekte sind, trifft das auch für sie zu. Aber trotzdem ist *Image*-Objekt nicht gleich *Image*-Objekt und Java hat auch nicht zufälligerweise verschiedene Bildtypen.

BufferedImage

Der einfachste und damit auch komfortabelste Bildtyp ist *BufferedImage*. Ein *BufferedImage* wird nämlich komplett von Java verwaltet und ist somit perfekt für den Einstieg in die Programmierung mit Bildern geeignet.

Was macht aber Java im Hintergrund? Und wie sieht so eine “Verwaltung” eigentlich intern aus? Nun genaugenommen werden *Image*-Objekte in der Regel zuerst einmal im Hauptspeicher angelegt und dann dort durch verschiedene Grafik-Funktionen (wie z.B. Draw, Transform, Filter, etc.) bearbeitet. Zum Anzeigen des Bildes wird anschließend der Speicherbereich, der die eigentliche Bild-Information (also die einzelnen Bildpunkte / Pixel) enthält, aus dem Hauptspeicher in den Videospeicherbereich (VRAM) kopiert und von dort aus dann über die Grafikkarte auf den Bildschirm projeziert. Das Anlegen der Kopie im Hauptspeicher, sowie das Kopieren der Pixel (*Blitting*) aus dem Hauptspeicher in das VRAM führt Java bei einem *BufferedImage* automatisch durch. Um jedoch den Kopiervorgang zwischen Hauptspeicher und VRAM zu beschleunigen, legt Java bei einem *BufferedImage*, eine zusätzliche Kopie des Bildes in einem hardwarebeschleunigten

Bereich an und synchronisiert diese dann bei Bedarf automatisch mit der Kopie aus dem Hauptspeicher. Dies bringt vor allem dann einen Geschwindigkeitsvorteil, wenn es sich bei dem *BufferedImage* um ein Bild mit relativ statischem Inhalt handelt.

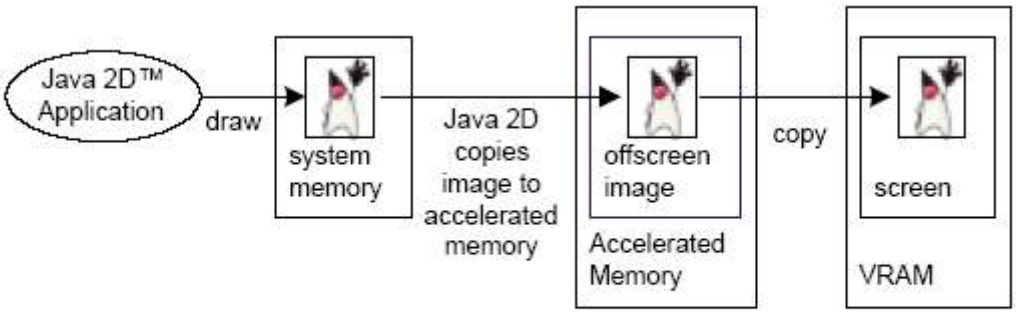


FIGURE 2 Rendering to a surface with no content loss

(Quelle: VolatileImage.pdf)

Sprites

Sprites sind Bilder mit statischem Inhalt, die Charaktere und Objekte eines Spiels darstellen. Da ihr Inhalt sich nach dem Laden nicht mehr verändert, sollte für sie ein *BufferedImage* verwendet werden. Das *BufferedImage* erstellt nämlich beim ersten Rendern automatisch eine Kopie im VRAM und verwendet diese anschließend für weitere Render-Durchläufe. Da der Inhalt der Bilder statisch ist, ist eine permanente Synchronisation zwischen Hauptspeicher und VRAM nicht notwendig, wodurch man hier eine performante und automatisch-verwaltete Version des Bildes bekommt.

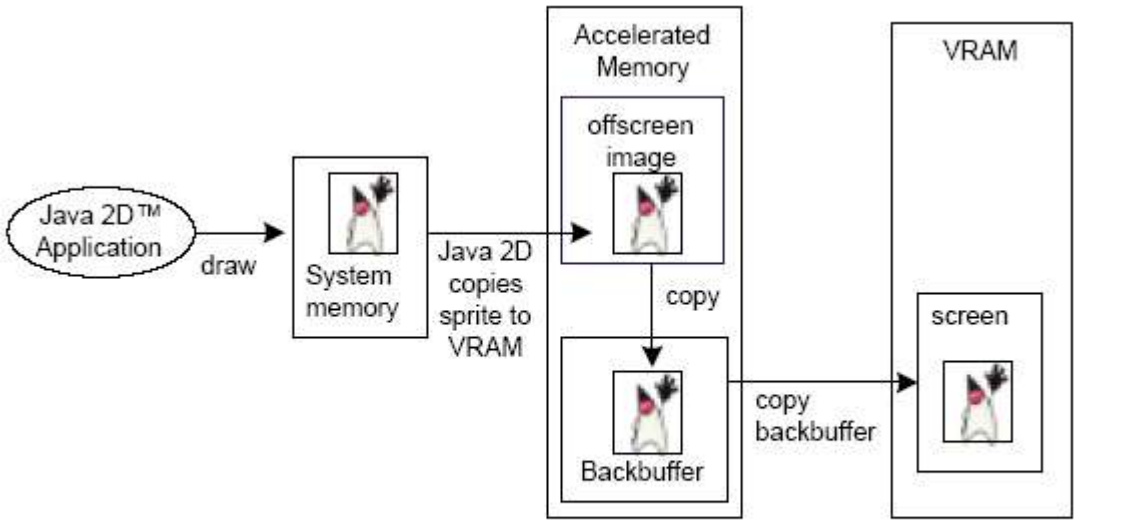


FIGURE 4 Rendering sprites

(Quelle: VolatileImage.pdf)

VolatileImage

Wird der Inhalt eines Bildes jedoch häufig verändert, dann muß eine evtl. vorhandene Kopie im hardwarebeschleunigten Speicherbereich ebenfalls häufig aktualisiert werden. Wie man sich sicher vorstellen kann, geht in dieser Situation sehr viel Zeit für das Kopieren zwischen Hauptsspeicher und hardwarebeschleunigtem Speicher verloren und Java kann hier intern auch nicht viel optimieren. Deshalb gibt es seit Java Version 1.4 einen neuen Bildtyp, der auf die automatisierte Verwaltung und Synchronisation verzichtet und das Bild direkt im hardwarebeschleunigten Bereich, also dem VRAM bei Windows Betriebssystemen, anlegt. Dadurch entfällt das Kopieren aus dem Hauptspeicher und das Verändern der Pixel wird durch hardwarebeschleunigte Grafik-Funktionen direkt im VRAM durchgeführt. So wird nicht nur die CPU entlastet, sondern Grafik-Operationen können auch parallel, durch die Verwendung der Grafik-Hardware, durchgeführt werden. Dieser neue, beschleunigte Bildtyp heißt *VolatileImage*.

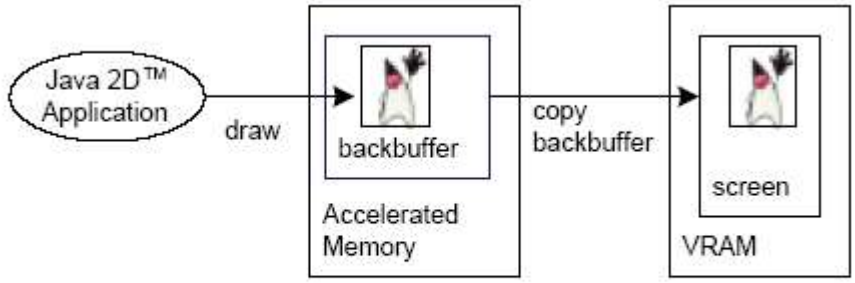


FIGURE 3 Double-Buffering

(Quelle: VolatileImage.pdf)

Wie das Wort *Volatile* (auf Deutsch "flüchtig") aber schon andeutet, ist das VRAM ein "flüchtiger" Speicherbereich, in dem Daten jederzeit überschrieben werden können. Deshalb müssen bei der Verwendung eines *VolatileImages* auch spezielle Maßnahmen getroffen werden, um den korrekten Inhalt des Bildes sicherzustellen bzw. bei Bedarf zu erneuern. Diesen Zusatzaufwand muß man hier leider betreiben, bekommt aber im Gegenzug einen extrem performanten Bild-Typ. Da ein Überschreiben der Bilddaten im VRAM seltener auftritt, als man es zunächst annehmen würde und zudem auch noch durch das Betriebssystem signalisiert wird, eignet sich das *VolatileImage* perfekt als *BackBuffer* für das *DoubleBuffering*.

Folgende Situationen führen zu einem Überschreiben von Daten im VRAM:

- Ausführen einer anderen Anwendung im Fullscreen-Modus
- Starten eines Bildschirmschoners
- Unterbrechen eines Tasks über den Taskmanager (unter Windows)
- Verändern der Bildschirmauflösung

VolatileImage als BackBuffer

Wird ein *VolatileImage* als *BackBuffer* verwendet, dann muß man vor jedem Zugriff auf den *BackBuffer* überprüfen, ob dieser noch gültig ist und bei Bedarf einen neuen *BackBuffer* erzeugen. Außerdem muß man nach dem Rendern überprüfen, ob der Inhalt des *VolatileImages* in der Zwischenzeit vielleicht überschrieben wurde und ggf. nochmal neu rendern. Ist dieser Fall jedoch nicht eingetreten, dann kann man den *BackBuffer* jetzt auf dem Bildschirm anzeigen.

Der veränderte Quellcode könnte in ungefähr so aussehen:

```
import java.awt.*;
import java.awt.image.*;
import javax.swing.JComponent;

public class MyComponent extends JComponent
{
  private VolatileImage backBuffer;
  private Image myImage;

  // more code to write here ...

  private void createBackBuffer()
  {
    // get the actual GraphicsConfiguration and create a compatible
    // VolatileImage as BackBuffer
    GraphicsConfiguration gc = getGraphicsConfiguration();
    backBuffer = gc.createCompatibleVolatileImage(getWidth(),getHeight());
  }

  public void renderScreen()
  {
    // if backBuffer doesn't exist, create one
    if (backBuffer == null) createBackBuffer();

    do
    {
      // validate the backBuffer
      int valCode = backBuffer.validate(getGraphicsConfiguration());
      if (valCode == VolatileImage.IMAGE_RESTORED)
      {
        System.out.println("backBuffer - IMAGE_RESTORED");
        // This case is just here for illustration
        // purposes.  Since we are
        // recreating the contents of the back buffer
        // every time through this loop, we actually
        // do not need to do anything here to recreate
        // the contents.  If our VImage was an image that
        // we were going to be copying _from_, then we
        // would need to restore the contents at this point
      }
      else if (valCode == VolatileImage.IMAGE_INCOMPATIBLE)
      {
        // backBuffer Image is incompatible with actual screen
        // settings, so we have to create a new compatible one
        System.out.println("backBuffer - IMAGE_INCOMPATIBLE");
        createBackBuffer();
      }

      // get Graphics object from backbuffer
      Graphics g = backBuffer.getGraphics();

      // render on backbuffer
      g.drawImage(myImage,0,0,this);
      g.drawLine(0,0,10,20);
      // ...

      // rendering is done; now check if contents got lost
      // and loop if necessary
    } while (backBuffer.contentsLost());
  }

  // ... more code to write here
}
```

Einsatz und Grenzen der VolatileImage API

Die *VolatileImage API* ist momentan noch stark in Entwicklung und wird mit kommenden Java Versionen noch weiter verbessert werden. Derzeit ist nur das Zeichnen von Lienen und das Kopieren und Füllen von rechteckigen Bereichen hardwarebeschleunigt, sowie einige komplexere Funktionen, die als Kombination dieser Basisfunktionen auftreten. Beim Rendern von Text können die einzelnen Buchstaben nach dem Rendern im VRAM zwischengespeichert und von dort bei weiteren Render-Durchläufen wieder kopiert werden, wodurch man sich ein aufwendiges Neu-Rendern spart. Komplexere Funktionen jedoch wie z.B. diagonale Linien, Curved Surfaces, Anti-Aliasing und Komposition sind nicht hardwarebeschleunigt und werden derzeit über reines Software-Rendering realisiert. Da aber Software-Rendering im Hauptspeicher stattfindet, führt das Verwenden dieser Funktionen auf einem *VolatileImage* zu starken Performance-Einbußen. Deshalb muß man je nach Anwendung und Einsatzbereich abwägen, ob sich die Verwendung eines *VolatileImages* lohnt oder nicht.

Hinweis: Da Linux und Solaris keinen direkten Zugriff auf das VRAM unterstützen, ist der Inhalt eines *VolatileImage* unter diesen Betriebssystemen auch nicht "flüchtig" und somit auch nicht hardwarebeschleunigt. Lediglich bei der Verwendung eines X-Terminal Clients kann man mit einem *VolatileImage* an Performance gewinnen, da der *BackBuffer* hier serverseitig in einer *pixmap* gespeichert wird.

c) Animation und Timing

Über *Aktives Rendern* und *Double Buffering* kann man nun Komponenten mit sich schnell und regelmäßig ändernder Grafik darstellen und hat so schon die Grundlage für *Animation* geschaffen. Diese benötigt nämlich ein kontinuierliches, flüssiges und performantes Neuzeichnen von Bildern.

Das menschliche Auge ist darauf spezialisiert, Bilder in schneller Folge zu verarbeiten und auf Gemeinsamkeiten bzw. Unterschiede im Bildaufbau hin zu analysieren. Dabei fallen ihm besonders die unterschiedlichen Merkmale auf, während ähnliche Merkmale als "normal" hingenommen und damit mehr oder weniger ignoriert werden. Dieses Wissen wird z.B. im militärischen Bereich bei der Tarnung angewandt, wo bewußt zum Umfeld passende Muster verwendet werden, um so die unregelmäßigen Merkmale des zu tarnenden Objektes zu verschleiern und dadurch das menschliche Auge zu täuschen.

High-Resolution Timer

Wendet man dieses Wissen nun auf die *Animation* an, dann stellt man fest, daß das menschliche Auge schon kleine Unregelmäßigkeiten im Bildaufbau besonders stark wahrnimmt und den Spieler recht schnell darauf aufmerksam macht. Deshalb sollte eine gute Animation so flüssig und regelmäßig wie nur möglich sein!

Java selbst bietet hierfür allerdings noch nicht die besten Voraussetzungen. Der bisher integrierte *Timer* `System.currentTimeMillis()` liefert nur Werte im Mikrosekundenbereich und ist zudem auch noch recht ungenau. Für eine flüssige Animation wird jedoch ein extrem genauer *Timer* benötigt, der im Nanosekundenbereich arbeitet. Abhilfe können hier externe Bibliotheken schaffen, die z.B. über JNI auf *Timer* des Betriebssystems zugreifen. Der Preis dafür ist jedoch die zusätzlich verwendete Bibliothek und evtl. damit verbunden Lizenzrestriktionen. Dieses Problem ist bekannt und soll nun in der neuen Java Version 1.5 gelöst werden. Diese soll nämlich den lange ersehnten *high-resolution Timer* beinhalten und damit eine einheitliche Lösung für alle Java Plattformen bieten.

Timing über Threads

Eine Alternative zu einem externen *high-resolution Timer* wäre noch ein Timing über das Thread-System von Java. Der Ausführungszeitpunkt eines Threads ist zwar ungenau, aber die Methoden werden in relativ regelmäßigen Abständen und immer(!) nacheinander ausgeführt. Nutzt man dieses Wissen und bewegt jetzt mit jedem Methoden-Aufruf die zu bewegenden Bilder um die gleiche Anzahl an Maßeinheiten weiter, dann läßt sich eine relativ flüssige *Animation* erzeugen, wenn die Auslastung der CPU und die Anzahl der aktiven Threads gering ist. Diese Art der *Animation* eignet sich daher bedingt für kleineren 2D Spiele wie z.B. Applets, ist jedoch von der Performance des verwendeten Rechners abhängig und wird auf schwächeren Systemen natürlich deutlich langsamer laufen. Die Synchronisation von Multiplayer-Spielen kann man deshalb nicht über ein reines Thread-Timing realisieren, sondern muß hier zusätzlich noch eine andere (evtl. auch ungenauere) und von der Performance des Systems unabhängige Form des Timings verwenden.

2.2. Java 3D und andere Bibliotheken

Will man 3D Grafik oder 3D Sound in Java-Programmen verwenden, dann hat man hier verschiedene Möglichkeiten: entweder man greift auf die *Java3D* API von Sun zurück, oder aber man verwendet externe Bibliotheken von Drittherstellern.

Die einfachste und auch komfortabelste Lösung ist die Verwendung der *Java3D* API von Sun. Diese bietet nämlich, neben 3D Grafik, ein Framework für 3-dimensionalen Sound, sowie Unterstützung für zahlreiche, teilweise auch exotische Eingabegeräte, wie z.B. “VR-Headsets” und “Data Gloves”. Ihr größter Vorteil jedoch ist die Verwendung eines Szene-Graphen, welcher nicht nur ein objektorientiertes Arbeiten mit 3D Grafik und 3D Sound ermöglicht, sondern auch noch zusätzliche Features wie z.B. “collision detection” und “collision handling” bereitstellt. Der Programmierer kann mit *Java3D* von einer sehr mächtigen und funktionsreichen API profitieren und so auf einem extrem hohen Abstraktionsniveau arbeiten. Dadurch spart man sich natürlich jede Menge Entwicklungszeit. Allerdings zahlt man auch einen Preis dafür, denn die *Java3D* API wurde als generelle API für 3D Anwendungen konzipiert und ist somit natürlich nicht primär auf die Anforderungen von Spielen optimiert. Ein Großteil ihrer Funktionalität wird in der Regel auch nicht in Spielen benötigt und ist damit erst einmal unnötiger Ballast. Trotzdem ist *Java3D* eine stabile API, deren Performanz für eine Vielzahl von 3D Anwendungen und Spielen locker ausreicht. *Java3D* ist nicht Bestandteil des Java “Core” und muß daher zusätzlich heruntergeladen und installiert werden. Allerdings kann *Java3D* danach problemlos in die “Java Foundation Classes” eingebunden werden und mit AWT-Komponenten (mit etwas Anpassung auch Swing-Komponenten) zusammenarbeiten. Eine Einführung in *Java3D* befindet sich im Anhang.

Eine Alternative zu *Java3D* wäre die Verwendung verschiedener externer Bibliotheken von Drittherstellern. Beispielsweise könnte man eine OpenGL-Bibliothek wie JOGL (https://jogl.dev.java.net) oder GL4JAVA (http://www.jausoft.com/gl4java.html) verwenden, um so die 3D Grafik über die OpenGL Schnittstelle zu realisieren. Dadurch hat man nicht nur einen direkten Zugriff auf die Rendering Pipeline und somit die Möglichkeit, das Rendering selbst zu beeinflussen, sondern kann zusätzlich sogar noch, durch die Verwendung hardwarespezifischer Funktionen, ein extremes “Performance-Tuning” vornehmen. Wie man sich sicher vorstellen kann, ist diese Variante natürlich performanter, erfordert dafür aber ein Arbeiten auf einem relativ niedrigen Abstraktionsniveau und bedeutet damit auch einen deutlichen Mehraufwand in der Entwicklungszeit. Funktionalitäten wie 3D Sound, “collision detection / handling” oder die Unterstützung von Eingabegeräten muß man hier entweder selbst implementieren, oder aber in zusätzlichen Bibliotheken “einkaufen”. Dadurch steigt natürlich zum Einen die Komplexität der Anwendung und zum Anderen die Abhängigkeit von verschiedenen externen Anbietern, sowie evtl. damit verbundene Lizenzrestriktionen (z.B. GPL). Zusätzlich muß man sich dann auch noch

Gedanken über die Kompatibilität der einzelnen Bibliotheken untereinander machen und natürlich auch den Installationsaufwand auf Seiten der Nutzer (Spieler) berücksichtigen.

Da die derzeit verfügbaren Java-Bibliotheken für Spiele fast alle unter GPL oder ähnlichen Lizenzmodellen genutzt werden können, muß man sich als Hobby-Spieleprogrammierer erst einmal keine großen Gedanken über Lizenzkosten machen. Allerdings sollte man berücksichtigen, daß die meisten Bibliotheken noch stark in der Entwicklung sind und deshalb von Version zu Version sehr unterschiedliche Funktionalität aufweisen können. Eine vielversprechende Bibliothek scheint derzeit die "**L**ight**w**eight **J**ava **G**ame **L**ibrary" (http://www.lwjgl.org) zu sein, die neben einem OpenGL-Binding sowohl 3D Sound als auch eine Unterstützung für die direkte Ansteuerung zahlreicher Eingabegeräte bietet. Sie wird speziell für die Spieleprogrammierung unter Java entwickelt und soll Performanz und Einfachheit verbinden. LWJGL agiert als gemeinsame Schnittstelle zwischen Java und nativen Funktionen des Betriebssystems und gewährleistet so, bei Verwendung der Bibliothek, eine Plattformunabhängigkeit der damit erstellten Spiele. Ein weiterer Bonus, den man durch die Verwendung von LWJGL schon erhält, ist ein *high-resolution Timer* (siehe *2.1 c High-Resolution Timer*), der sonst in den Java Versionen vor Version 1.5 noch fehlt.

2.3. Sound und Musik

Ein weiteres Sinnesorgan, mit dem wir unsere Umwelt aktiv wahrnehmen, ist das Ohr. Es verarbeitet Geräusche und filtert Frequenzen und hält uns so über das akustische Geschehen um uns herum auf dem Laufenden. Deshalb spielen auch Ton und Musik in Spielen eine wichtige Rolle. Sie tragen nämlich nicht nur verstärkt zum Realismus eines Spieles bei, sondern generieren eine passende (oder auch unpassende) Geräuschkulisse und somit ein stimmungsvolles Ambiente. Neben einer ansehnlichen Grafik, einer guten Spielidee und einem gut-durchdachten Benutzerinterface sind also vor allem auch Ton und Musik ein wichtiger Bestandteil eines erfolgreichen Spieles.

AudioClip-Objekte

Für die Wiedergabe von Sound bietet Java verschiedene Möglichkeiten. Die älteste und einfachste Form ist die Verwendung von *AudioClip*-Objekten. Diese wurden ursprünglich nur in Verbindung mit *Applets* benutzt, aber dann in späteren Versionen von Java für eine Verwendung in Applikationen erweitert. *AudioClips* ermöglichen ein einfaches und vor allem automatisiertes Laden und Abspielen einer Sounddatei und sind somit sehr komfortabel für die Erstellung von einfachen Spielen oder Anwendungen. Java unterstützt jedoch nur lineare PCM Sounddateien, also Sounddateien ohne Komprimierung.

Das Laden einer Sounddatei und das Erzeugen des zugehörigen *AudioClip*-Objektes erfolgt über die statische Methode `newAudioClip(URL clip)` der Klasse *Applet*. Dadurch ist es möglich *AudioClips* direkt in Applikationen zu laden und zu verwenden, ohne ein zusätzliches *Applet*-Objekt erzeugen zu müssen. Das Abspielen eines *AudioClips* erfolgt dann anschließend über die `play()`-Methode des *AudioClip*-Objektes. Wird der *AudioClip* jedoch beim Aufruf der `play()`-Methode schon wiedergegeben, dann wird er automatisch an den Anfang zurückgespult und danach wieder neu abgespielt. Deshalb ist es also nicht möglich, den gleichen *AudioClip* zeitgleich mehrfach abzuspielen. Will man das trotzdem tun, dann muß man für die entsprechende Sounddatei einfach mehrere *AudioClip*-Objekte erzeugen.

Verwenden von Resourcen in JAR-Dateien

Da die Methode `newAudioClip(URL clip)` mit *URL*-Objekten arbeitet, wollen wir uns an dieser Stelle auch gleich schon mit einer Besonderheit beim Zugriff auf Ressourcen beschäftigen. Damit Ressourcen später nämlich auch aus JAR-Dateien heraus korrekt geladen und verwendet werden

können, müssen sie über *URL*-Objekte referenziert werden. Hierfür verwendet man am Besten die Methode `getResource(String name)` des *ClassLoader*-Objektes der JVM. Dadurch wird nämlich gewährleistet, daß das Laden dann später auch unter *JavaWebstart* (siehe *2.7 Build & Deploy*) funktioniert, denn *JavaWebstart* läßt aufgrund von strengen Sicherheitsrestriktionen einen Zugriff auf Ressourcen nur dann zu, wenn diese mit dem gleichen *ClassLoader*-Objekt geladen wurden. Um nun sicherzustellen, daß auch wirklich das gleiche *ClassLoader*-Objekt verwendet wird, kann man sich über `Klassenname.class.getClassLoader()` eine Referenz auf das *ClassLoader*-Objekt zurückgeben lassen, mit dem die Klasse selbst geladen wurde. Dies ist nämlich bei allen Klassen direkt der *ClassLoader* der JVM.

Mit diesem Wissen wollen wir uns nun an das Erstellen einer einfachen Jukebox wagen, die Sounds in Form von *AudioClips* verwaltet und bei Bedarf wiedergibt. Der entsprechende Quellcode dazu könnte in ungefähr so aussehen:

```
import java.net.URL;
import java.applet.*;

public class Jukebox
{
  private AudioClip[] sounds;    // AudioClips


  public Jukebox()
  {
    initSounds();
  }

  private void initSounds()
  {
    try
    {
      // needed for correct loading of resources in JAR-Files
      ClassLoader loader = Jukebox.class.getClassLoader();

      // load AudioClips
      sounds = new AudioClip[2];
      sounds[0] = Applet.newAudioClip(loader.getResource("sound.wav"));
      sounds[1] = Applet.newAudioClip(loader.getResource("sound1.wav"));
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
    }
  }

  public void playSound(int index)
  {
    if (index>0 && index<sounds.length) sounds[index].play();
  }

  public static void main(String[] args)
  {
    Jukebox jukebox = new Jukebox(); // initialize jukebox
    jukebox.playSound(0); // play first sound
    try
    {
      Thread.sleep(100); // wait a bit
    }
    catch (InterruptedException ex) {}
    jukebox.playSound(1); // play second sound
  }
}
```

JavaSound API

Nachdem wir nun einfache Sounddateien abspielen können, wollen wir uns jetzt auch noch mit dem Abspielen von Musik befassen. Hier bietet sich aufgrund der geringen Größe und großen Vielseitigkeit das *MIDI*-Format an. Da *AudioClip*-Objekte *MIDI*-Dateien aber nur unzureichend unterstützen, wollen wir uns an dieser Stelle kurz mit der *JavaSound* API und deren Möglichkeiten ausseinandersetzen.

Die *JavaSound* API ist eine mächtige, funktionsreiche und komplexe Bibliothek, die primär für die professionelle, digitale Soundverarbeitung entwickelt wurde. Sie unterstützt das Aufnehmen, Bearbeiten und Wiedergeben von digitalem ("sampled") Sound sowie das Interpretieren und Wiedergeben von *MIDI*. Allerdings ist das Verwenden der *JavaSound* API nicht ganz so einfach und komfortabel wie das Verwenden von *AudioClip*-Objekten. Während *AudioClips* vollständig automatisiert und über ein eigenes Thread-System abgespielt werden, muß man sich bei der Wiedergabe von digitalem Sound über die *JavaSound* API um viele Dinge selbst kümmern. Dies reicht vom Anfordern einer digitalen *DataLine* über das Laden und Verarbeiten der Sounddaten bis hin zum eigentlich Abspielen (Rendern) des Sounds in einem eigenständigen Thread. Da die notwendigen Schritte und Anweisungen hierfür sehr umfangreich sind und wir mit auch *AudioClip*-Objekten bereits ausreichende Unterstützung für digitale Sounds haben, wollen wir diesen Teil dem geneigten Leser zum Selbstexperimentieren überlassen und uns stattdessen mit der Wiedergabe von *MIDI*-Dateien über die *JavaSound* API befassen.

Wiedergabe von MIDI-Dateien

Für die Wiedergabe einer *MIDI*-Datei benötigt man die beiden Klassen *Sequencer* und *Sequence*. *Sequence* repräsentiert eine Sequenz von *MIDI*-Befehlen, also den Inhalt der *MIDI*-Datei, während *Sequencer* die Schnittstelle zum *MIDI*-Subsystem des Betriebssystems darstellt. Dieses kann je nach Systemkonfiguration eine Hardwarekomponente oder aber auch ein emuliertes Softwaregerät sein. Die *JavaSound* API ist zwar ebenfalls in der Lage MIDI zu emulieren, benötigt aber für eine gute Qualität eine zusätzlich installierte Soundbank. Diese kann z.B. bei Sun unter http://java.sun.com/products/java-media/sound/soundbanks.html heruntergeladen werden und behebt unter anderem auch evtl. auftretende Probleme beim Timing von MIDI Befehlen, weshalb man eine Installation in Erwägung ziehen sollte.

Bevor man eine *MIDI*-Sequenz wiedergeben kann, muß man zuerst ein *Sequencer*-Objekt anfordern. Dieses bekommt man über die statische Methode `getSequencer()` der Klasse *MidiSystem*. Ist das *MIDI*-System verfügbar und hat man eine gültige Referenz erhalten, dann wird der *Sequencer* über die Methode `open()` initialisiert. Das Laden einer *MIDI*-Sequenz erfolgt ebenfalls über die Klasse *MidiSystem* aber dieses mal durch die Methode `getSequence(URL url)`. Das Abspielen einer *Sequence* wird schließlich über die beiden Methoden `setSequence(Sequence seq)` und `start()` des *Sequencer*-Objektes durchgeführt. Ist der *Sequencer* zu diesem Zeitpunkt mit der Wiedergabe einer *MIDI*-Sequenz beschäftigt, dann wird die Wiedergabe der alten *Sequence* unterbrochen und mit der Wiedergabe der neuen *Sequence* begonnen.

Hinweis: Alle Methoden können spezielle *MidiExceptions* erzeugen, die vom Programmierer abgefangen und entsprechend behandelt werden solllten.

Mit dem soeben neu-erworbenen Wissen können wir nun unsere Jukebox erweitern und dazu bringen, *MIDI*-Sequenzen abzuspielen. Der entsprechende Quellcode dazu könnte in ungefähr so aussehen:

```
import java.net.URL;
import javax.sound.midi.*;
import java.applet.*;

public class Jukebox
{
  // ... more code to write here


  public Sequence loadSequence(String filename)
  {
    Sequence result = null;
    try
    {
      ClassLoader loader = Jukebox.class.getClassLoader();
      URL url = loader.getResource(filename);
      result = MidiSystem.getSequence(url);
      return result;
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
    }
    return null;
  }

  public void playSequence(Sequence seq)
  {
    if (seq == null) return;
    try
    {
      sequencer.setSequence(seq);
      sequencer.start();
    }
    catch (InvalidMidiDataException ex)
    {
      ex.printStackTrace();
    }
  }

  public static void main(String[] args)
  {
    Jukebox jukebox = new Jukebox(); // initialize jukebox

    Sequence music = jukebox.loadSequence("test.mid"); // load Sequence
    jukebox.playSequence(music); // start playing Sequence

    jukebox.playSound(0); // play first sound
    try
    {
      Thread.sleep(100); // wait a bit
    }
    catch (InterruptedException ex) {}
    jukebox.playSound(1); // play second sound
  }
}
```

2.4. Benutzereingabe

Wir kennen nun schon einige Möglichkeiten, wie ein Spiel oder auch Programm dem Benutzer visuelles und akustisches Feedback geben kann. Allerdings sind Spiele, im Gegensatz zu Filmen, interaktiv und bieten dem Spieler so ein Zwei-Wege-Kommunikationssystem, bei dem er direkt oder indirekt Einfluß auf das Spielgeschehen nehmen kann. Diese Steuerung wird über Eingabegeräte realisiert.

Die wohl am häufigsten benutzten Eingabegeräte sind Maus und Tastatur. Danach kommen noch Gamepads, Joysticks, Lenkräder und viele andere, teilweise sehr exotische Dinge wie z.B. Lichtgriffel, VR-Headsets oder Data-Gloves.

Maus und Tastatur

Für die meisten Spiele wird in der Regel eine Steuerung über Maus und Tastatur ausreichen und Java bietet hier, durch das Event-Listener System von AWT / Swing, schon eine ausreichende Unterstützung. Über die *KeyListener*-Schnittstelle können Tastatur-Eingaben behandelt werden und über die *MouseListener*- bzw. *MouseMotionListener*-Schnittstelle Mouse-Eingaben. Die Funktionsweise dieser Schnittstelle ist, falls noch nicht bekannt, in der *Einführung in Swing* im Anhang beschrieben.

Wichtig ist an dieser Stelle allerdings, daß man bedenkt, das über das *Event*-System in der Regel sehr viele *Events* behandelt werden und daher die Bearbeitungszeit für einzelne *Events* so gering wie möglich gehalten werden sollte. Deshalb ist es auch wichtig, das *Event*-System z.B. nicht durch langwierige Funktionsaufrufe zu blockieren, sondern stattdessen lieber Status-Variablen zu verwenden, die den Zustand der Eingabe repräsentieren und später dann von einem eigenständigen Thread wie z.B. dem Game-Loop (siehe 2.6 Threads und Performance) bearbeitet werden.

Eine einfache Klasse zur Behandlung von Tastatur-Eingaben könnte nach diesem Modell ungefähr so aussehen:

```
import java.awt.event.*;

public class InputController implements KeyListener
{
  // player control variables
  public boolean moveRight = false;
  public boolean moveLeft = false;
  public boolean moveDown = false;
  public boolean moveUp = false;
  public boolean attack = false;

  /*
   * process KeyTyped events
   */
  public void keyTyped(KeyEvent e) {}

  /*
   * process KeyReleased events
   */
  public void keyReleased(KeyEvent e)
  {
    // handle player input and reset player control variables
    if (e.getKeyCode() == KeyEvent.VK_UP) moveUp = false;
    else if (e.getKeyCode() == KeyEvent.VK_DOWN) moveDown = false;
    else if (e.getKeyCode() == KeyEvent.VK_LEFT) moveLeft = false;
    else if (e.getKeyCode() == KeyEvent.VK_RIGHT)  moveRight = false;
  }
```

```
  /*
   * process KeyPressed events
   */
  public void keyPressed(KeyEvent e)
  {
    // exit program with ESCAPE
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE)  System.exit(0);

    // move player up with UP-ARROW key
    else if (e.getKeyCode() == KeyEvent.VK_UP)
    {
      moveUp = true;
      moveDown = false;
    }
    // move player down with DOWN-ARROW key
    else if (e.getKeyCode() == KeyEvent.VK_DOWN)
    {
      moveUp = false;
      moveDown = true;
    }
    // move player left with LEFT-ARROW key
    else if (e.getKeyCode() == KeyEvent.VK_LEFT)
    {
      moveLeft = true;
      moveRight = false;
    }
    // move player right with RIGHT-ARROW key
    else if (e.getKeyCode() == KeyEvent.VK_RIGHT)
    {
      moveRight = true;
      moveLeft = false;
    }
    // attack with CONTROL key
    else if (e.getKeyCode() == KeyEvent.VK_CONTROL)
    {
      attack = true;
    }
  }
}
```

Joysticks und Gamepads

Während einige Spiele wie z.B. Strategiespiele und Adventures noch mit realtiv wenig Benutzereingaben auskommen, fordern andere Spiele wie z.B. Jump&Run-Games oder Flug-Simulatoren deutlich mehr Interaktion vom Spieler. Hier kann eine reine Steuerung über Tastatur und Maus recht schnell zur Qual werden, weshalb man sich an dieser Stelle auch mit komfortableren und speziell für diesen Zweck entwickelten Eingabegeräten, wie Joysticks und Gamepads, befassen sollte.

Für die Ansteuerung von solchen Eingabegeräten bietet Java selbst erst einmal keine Unterstützung. Deshalb muß man hier auf andere Bibliotheken wie z.B. *Java3D* oder *LWJGL* (siehe *2.2. Java 3D und andere Bibliotheken*) zurückgreifen. Eine sehr komfortable und auch funktionsreiche Bibliothek, die sich ausschließlich mit der Unterstützung von Eingabegeräten befaßt, ist *JXInput* (http://www.hardcode.de/jxinput/). *JXInput* unterstützt eine sehr große Zahl von Eingabegeräten und arbeitet auch, durch die Verwendung von nativen *DirectInput*-Funktionen, sehr performant. Die Bibliothek versucht eine einheitliche Schnittstelle zwischen Java und nativen Funktionen des Betriebssystems zu bieten und ist mittlerweile nicht nur für Windows-Systeme, sondern auch für die Linux- und MacOS-Plattform erhältlich. Für die Qualität der Bibliothek spricht auch die Tatsache, daß selbst die NASA *JXInput* für die Steuerung eines Roboters auf der Internationalen Raumstation ISS einsetzt (http://www.hardcode.de/jxinput/jxinput_in_space/index.html)!

Die Initialisierung eines Eingabe-*Devices* erfolgt bei JXInput über die statische Methode `getJXInputDevice(int deviceID)` der Klasse *JXInputManager*. Über die statische Methode `getNumberOfDevices()` kann man die Anzahl der verfügbaren *Devices* erfahren. Hat man ein gültiges *Device* erhalten, dann kann man auf die Achsen des Gerätes über die Methode `getAxis(int axisID)` und auf die Tasten des Gerätes über die Methode `getButton(int buttonID)` zugreifen. Die Anzahl der verfügbaren Achsen erhält man über die Methode `getMaxNumberOfAxes()` und die Anzahl der verfügbaren Tasten über die Methode `getMaxNumberOfButtons()`. Den Auslenkungswert für eine Achse bekommt man jeweils über die Methode `getValue()` und den Zustand einer Taste über die Methode `getState()`. Da Joysticks und ähnliche Eingabegeräte, im Gegensatz zu Maus und Tastatur, nicht über ein Event-System zur Benachrichtigung bei Zustandsänderungen verfügen, muß man hier, vor jeder weiteren Verarbeitung der Eingabewerte, den Zustand des Gerätes zuerst abfragen. Diese Art der Abfrage nennt man *Pollen*. Das *Pollen* wird in *JXInput* über die statische Funktion `updateFeatures()` der Klasse *JXInputManager* realisiert und aktualisiert bei jedem Aufruf, den Zustand aller angeschlossenen und initialisierten *Devices*.

Um ein einfaches und anschauliches Beispiel für den Einsatz von *JXInput* zu geben, wollen wir, mit dem soeben erworbenen Wissen, eine *Controller*-Klasse entwickeln, die einen Joystick mit 3 Achsen und 2 Tasten unterstützt und dabei die Abfrage der Eingabewerte über *JXInput* realisiert. Durch die Kapselung der Funktionen können wir später so ohne großen Änderungsaufwand auch andere Bibliotheken verwenden. Der entsprechende Quellcode könnte in ungefähr so aussehen:

```
import de.hardcode.jxinput.*;

public class DeviceController
{
  private byte axes = 0;     // number of axes
  private Axis xAxis = null; // x-axis
  private Axis yAxis = null; // y-axis
  private Axis zAxis = null; // z-axis

  private JXInputDevice device = null;


  public DeviceController()
  {
    initDevice(0); // pick the first available device
  }

  public DeviceController(int deviceID)
  {
    initDevice(deviceID);
  }

  private void initDevice(int deviceID)
  {
    // invalid value for deviceID!
    if (deviceID < 0) return;

    // get number of available devices
    int cnt = JXInputManager.getNumberOfDevices();

    // is there a device with this id?
    if (cnt >= deviceID )
    {
      device = JXInputManager.getJXInputDevice(deviceID); // pick it

      // is device valid?
      if (device != null)
      {
        // get number of available axes and store axis-id's
        int axes = device.getMaxNumberOfAxes();
        if (axes >= 1) // more than 1 axis?
```

```
        {
          axes = 1;
          xAxis = device.getAxis(Axis.ID_X); // remember id for x-axis

          if (axes >= 2) // more than 2 axes?
          {
            axes = 2;
            yAxis = device.getAxis(Axis.ID_Y); // remember id for y-axis

            if (axes >= 3) // more than 3 axes?
            {
              axes = 3;
              zAxis = device.getAxis(Axis.ID_ROTZ); // remember id for z-axis
            }
          }
        }
        else // less than 1 axis
        {
          axes = 0;       // invalid number of axes
          device = null; // device useless!
        }
      } // end - is device valid?
    } // end - is there a device with this id?
  } // end initDevice()


  public boolean isReady()
  {
    if (device != null) return true;
    else return false;
  }

  public int getX()
  {
    if (xAxis != null)
    {
      // get value and make it a nice int value
      int result = (int)(xAxis.getValue() * 100);
      return result;
    }
    else return 0;
  }

  public int getY()
  {
    if (yAxis != null)
    {
      // get value and make it a nice int value
      int result = (int)(yAxis.getValue() * 100);
      return result;
    }
    else return 0;
  }

  public int getZ()
  {
    if (zAxis != null)
    {
      // get value and make it a nice int value
      int result = (int)(zAxis.getValue() * 100);
      return result;
    }
    else return 0;
  }

  public String getName()
  {
```

```
    // get the name of the device
    if (device != null) return device.getName();
    else return "null";
  }

  public boolean Button1Pressed()
  {
    if (device != null) return device.getButton(0).getState();
    else return false;
  }

  public boolean Button2Pressed()
  {
    if (device != null) return device.getButton(1).getState();
    else return false;
  }

  public void update()
  {
    JXInputManager.updateFeatures();
  }

  public static void main(String[] args)
  {
    DeviceController control = new DeviceController();

    // print the name of the device
    System.out.println(control.getName());

    // while device is ready for input
    while(control.isReady())
    {
      // poll new values from device
      control.update();

      // print values on screen
      System.out.println("x: "+control.getX());
      System.out.println("y: "+control.getY());
      System.out.println("z: "+control.getZ());
      if (control.Button1Pressed()) System.out.println("button 1 pressed!");
      if (control.Button2Pressed()) System.out.println("button 2 pressed!");

      // wait a bit before next polling
      try
      {
        Thread.sleep(300);
      }
      catch (InterruptedException e) {}
    }
  }
}
```

2.5. Netzwerk

Wozu braucht man bei einem Spiel Netzwerkunterstützung? Diese Frage sollte man sich so früh wie möglich stellen, am Besten sogar schon beim Entwickeln der ersten Spielidee! Viele Spiele werden gänzlich ohne Netzwerkunterstützung auskommen und ein gutes Spielprinzip muß auch nicht immer komplex sein und nicht unbedingt eine Mehrspieleroption haben. Allerdings wird man sich früher oder später oftmals fragen, ob man eigentlich nicht auch zusammen mit Freunden spielen, und den Spielspaß so vielleicht sogar noch vergrößern könnte. Es ist zwar in vielen Fällen noch möglich, Spielen nachträglich eine Mehrspieleroption zu verpassen, aber es ist auf keinen Fall ratsam. Neben den vielen Fallstricken nämlich, die bei der technischen Realisierung und Implementierung einer Multiplayeroption auftreteten können, kann eine solche Erweiterung auch

starke Änderungen am Spielkonzept erforderlich machen und je komplexer dieses ist, desto gravierender können dann die Auswirkungen sein. Viel besser ist es, sich schon zu Beginn der Entwicklung zu überlegen, ob und inwieweit die Spielidee überhaupt mehrspielertauglich ist und welche Auswirkungen eine Mehrspieleroption auf das Spielkonzept und Balancing haben wird. Stellt man diese Überlegungen frühzeitig an und hat auch Design und Spielkonzept entsprechend angepaßt, dann steht dem Spielspaß im Team oder auch gegeneinander nur noch die technische Implementierung im Wege. Diese aber sollte wirklich gut sein, denn Spiele erhalten gerade durch die Interaktion mit anderen, menschlichen Spielern meistens eine völlig neue Dimension, was oftmals über Erfolg oder Mißerfolg des Spieles entscheiden kann!

a) Java NIO

Java bietet mit der *Java NIO* API die Möglichkeit, eine gute und auch performante Implementierung einer Netzwerkunterstützung zu schaffen. Die Abkürzung *NIO* steht dabei für *New I/O* und deutet schon darauf hin, daß die neue API wohl Unterschiede zur alten *Java IO* API haben muß. Der wohl größte Unterschied ist, neben einer besseren Strukturierung und einer höheren Performanz, die Unterstützung von asychroner I/O-Verarbeitung, sowie die Möglichkeit des Multiplexens.

Während die *Java IO* API bisher nur die synchrone I/O-Verarbeitung unterstützte und man daher aufgrund der "blockierenden" Funktionen jeweils auf das Ende der I/O-Verarbeitung warten mußte, was eine Verwendung von zahlreichen zusätzlichen Threads notwendig machte, unterstützt *Java NIO* nun auch eine asynchrone I/O-Verarbeitung, bei der die verwendeten I/O-Funktionen nicht mehr die Ausführung des gesamten Threads, bis zum Ende der I/O-Verarbeitung, blockieren. Die Netzwerkimplementierung kann daher unter *Java NIO* mit weit weniger Threads auszukommen und führt so zu einer Senkung der Komplixität des Programmcodes bei einer gleichzeitigen Steigerung der Performanz. Während man mit *Java IO* und sychroner I/O-Verarbeitung für den Server in der Regel 2 Threads für Client-Verbindungsanfragen und Server-Logik, sowie je einen Thread pro Client-Verbindung benötigt, kann man durch die Verwendung von *Java NIO* und asynchroner I/O-Verarbeitung die Zahl der benötigten Threads auf insgesamt 1-2, für Verbindungshandling und Serverlogik, reduzieren. Betrachtet man diese Zahlen, dann kommt man mit *Java IO* auf insgesamt 2+n Threads und mit *Java NIO* dagegen auf insgesamt 2 Threads.

Einen weiteren Permormance-Gewinn erhält man mit *Java NIO* über das Multiplexen. *Java NIO* ermöglicht es nämlich, einzelne Verbindungen, ähnliche wie die Fadenstränge eines Seiles, über *Selector*-Objekte zu bündeln und so gemeinsam zu bearbeiten. Dabei werden nur jeweils die Verbindungen berücksichtigt, die aktuell eine Bearbeitung benötigen. Dies ermöglicht eine bedarfsgesteuerte Verarbeitung von I/O und erspart so ein zeitaufwendiges und weniger performantes Abfragen (*Pollen*) der einzelnen Verbindungen. Der Zeitverlust, der durch ein *Pollen* von inaktiven Verbindungen entsteht, ist bei wenigen Verbindungen noch gering, wird aber bei einer größeren Anzahl deutlich wahrnehmbar, da hier mehrere 100 Abfragen pro Sekunde vorgenommen werden, die bei der Verwendung eines bedarfsgesteuerten Multiplexers natürlich wegfallen.

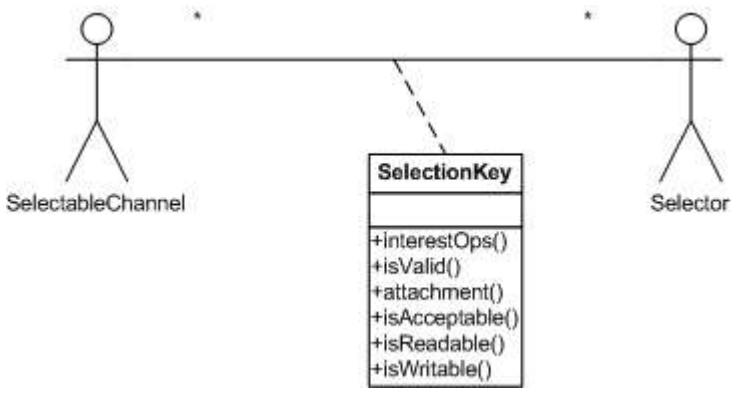
Aufbau der Java NIO

Die Java NIO besteht aus den Paketen

- java.nio.*
- java.nio.channels.*
- java.nio.charset.*

Die 3 wichtigsten Klassen sind *SelectableChannel*, *Selector* und *SelectionKey*.

Ein *SelectableChannel*-Objekt repräsentiert eine Verbindung mit einem *Socket* und kann beliebig vielen *Selector*-Objekten zugeordnet werden. Ein *Selector*-Objekt dagegen bündelt mehrere *SelectableChannel*-Objekte und ermöglicht ein gemeinsames Bearbeiten der gebündelten Verbindungen. Ein *SelectionKey*-Objekt repräsentiert den "Zustand" einer Zuordnung zwischen einem *SelectableChannel* und einem *Selector* und kapselt alle relevanten Daten, die für das Multiplexen des *SelectableChannels* über den *Selector* wichtig sind.



Betrachtet man noch einmal das Seil-Beispiel von vorhin, dann repräsentiert der *SelectableChannel* einen einzelnen Fadenstrang und der *Selector* das Seil selbst. Der *SelectionKey* wäre dann der Kleber, der die Fadenstränge und das Seil zusammenhält und dabei die "Beweglichkeit" festlegt.

Ein einfacher Server

Um die Zusammenhänge noch ein wenig deutlicher zu machen und auch gleichzeit schon ein praktisches Beispiel zu geben, wollen wir eine einfache Server-Klasse schreiben, die Client-Verbindungen annimmt und jeweils ein einfaches "Hallo" als Antwort zurückschickt. Hierfür müssen wir zuerst ein *Selector*-Objekt über die statische Methode `open()` der Klasse *Selector* anfordern. Danach wird für alle verfügbaren Netzwerk-Interfaces je ein *ServerSocketChannel* erstellt, der eingehende Verbindunsanfragen für beliebige Adressen auf einem festgelegten *Port* verarbeitet. Anschließend werden alle *ServerSocketChannel*-Objekte dem *Selector*-Objekt zugeordnet. Damit der *Selector* später allerdings weiß, wann ein *SocketChannel* für die Bearbeitung ausgewählt werden soll, gibt es in *Java NIO* genau 3 verschiedene *SocketChannel*-Operationstypen:

- `OP_ACCEPT` - der *SocketChannel* reagiert auf Verbindungsaufbau-Anfragen
- `OP_READ` - der *SocketChannel* kann Daten empfangen
- `OP_WRITE` - der *SocketChannel* kann Daten senden

Für die *ServerSocketChannels* wählen wir also den Operationstyp `OP_ACCEPT` und registrieren diese damit beim *Selector* über die Methode `register(Selector sel, int ops)`. Damit signalisieren wir dem *Selector* unser Interesse an eingehenden Verbindungen und werden später dann benachrichtigt, sobald Verbindungsanfragen vorliegen.

Wurden alle *ServerSocketChannels* erfolgreich registriert, dann ist die Initialisierung des Servers abgeschlossen und wir können mit der Vearbeitung der Verbindungen beginnen. Hierfür müssen wir zunächst alle *SocketChannels*, für die eine Bearbeitung notwendig ist, über die Methode `select()` des *Selectors* auswählen. Anschließend holen wir uns die Liste aller zu bearbeitenden *SelectionKeys* über die Methode `selectedKeys()` des *Selectors*. Wichtig ist an dieser Stelle, daß wir die einzelnen *SelectionKeys* nach der Bearbeitung aus dem erhaltenen *Iterator* manuell entfernen, da diese bei einem erneuten Aufruf der `select()` Methode automatisch wieder hinzugefügt werden, sofern die Notwendigkeit einer Bearbeitung besteht. Über die Methoden `isAcceptable()`, `isReadable()` und

`isWritable()` erfahren wir, ob für das *SelectionKey*-Objekt und den dazugehörigen *SelectableChannel* eine Verbindungsanfrage vorliegt bzw. ob Inputdaten verarbeitet werden müssen oder ob der *SelectableChannel* bereit für das Senden neuer Outputdaten ist.

Möchte man eine Verbindung nach der Bearbeitung wieder beenden, dann muß man den *SelectionKey* über die Methode `cancel()` stornieren, sowie den zugehörigen *SelectableChannel* über die Methode `close()` schließen. Durch das Stornieren des *SelectionKeys* wird dieser mit dem nächsten Aufruf der `select()` Methode aus dem *Selector* entfernt.

Der Quellcode für den Server könnte in ungefähr so aussehen:

```
import java.net.*;
import java.io.*;
import java.util.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

public class Server implements Runnable
{
   private Thread thread;  // the server thread

   private Selector selector;  // selector for multiplexing
   private int keysAdded = 0;  // number of keys (connections) added

   // buffer for reading and writing
   private ByteBuffer buffer = ByteBuffer.allocate(1024);

   // Charset and encoder for US-ASCII
   private static Charset charset = Charset.forName("US-ASCII");
   private static CharsetEncoder encoder = charset.newEncoder();


   public Server(int port)
   {
      init(port);
   }

   /*
    * init server for given port
    */
   public void init(int port)
   {
      // stores all listening server sockets on all network interfaces
      Vector allServerSocketChannels = new Vector();

      try
      {
         // create a selector for channel multiplexing
         selector = Selector.open();
      }
      catch (Exception ex)
      {
         System.err.println("could not open selector!");
         ex.printStackTrace();
      }

      // bind to all network interfaces
      try
      {
         // get available network interfaces
         Enumeration interfaces = NetworkInterface.getNetworkInterfaces();
         while (interfaces.hasMoreElements())
         {
            // process next network interface
            NetworkInterface ni = (NetworkInterface) interfaces.nextElement();

            // get all addresses of selected network interface
            Enumeration addresses = ni.getInetAddresses();
```

```
            while (addresses.hasMoreElements())
            {
               // process next address of selected network interface
               InetAddress address = (InetAddress) addresses.nextElement();

               System.out.println("binding to port " + port + " on InetAddress "
                     + address);

               // create a socket address for selected address and port
               InetSocketAddress isa = new InetSocketAddress(address,port);

               System.out.println("opening a non-blocking ServerSocketChannel on port "
                           + port + " on InetAddress "  + address);

               // open a (listening) ServerSocketChannel
               ServerSocketChannel ssc = ServerSocketChannel.open();
               ssc.configureBlocking(false); // set channel to non-blocking mode
               ssc.socket().bind(isa); // bind socket to socket address
               allServerSocketChannels.add(ssc); // store socket channel in channel list
            }
         }
      }
      catch (IOException ex)
      {
         System.err.println("could not start server!");
         ex.printStackTrace();
         System.exit(0);
      }

      // process all listening server sockets
      for (Iterator it = allServerSocketChannels.iterator(); it.hasNext();)
      {
         // process next ServerSocketChannel
         ServerSocketChannel ssc = (ServerSocketChannel) it.next();
         try
         {
            // register ServerSocketChannel with selector for multiplexing
            // announce interest for answering incoming connection-requests
            ssc.register(selector, SelectionKey.OP_ACCEPT);
         }
         catch (IOException ex)
         {
            System.err.println("could not register new SSC [\"" + ssc + "\"]!");
            ex.printStackTrace();
            System.exit(0);
         }
      }
      System.out.println("server started");
   }

   /*
    * start the server (if not started already)
    */
   public void start()
   {
      if (thread == null)
      {
         thread = new Thread(this);
         thread.start();
      }
   }

   /*
    * stop the server
    */
   public void stop()
   {
      thread = null;
   }

   /*
    * processing loop for server
    */
   public void run()
   {
```

```
   // while not stopped
   while (thread == Thread.currentThread())
   {
      try
      {
         // select ready channels and process them
         while ((keysAdded = selector.select()) > 0)
         {
            // handle client requests
            handleClientRequests();

            // rest a bit and give time to other threads
            threadSleep(50L);
         }
      }
      catch (Exception ex)
      {
         System.err.println("unexpected Exception during channel processing!");
         ex.printStackTrace();
         System.exit(0);
      }
   }
}

/*
 * this method is used to handle all kind of requests from clients.
 */
private void handleClientRequests() throws Exception
{
   // get ready keys (connections) from selector
   Set keys = selector.selectedKeys();
   Iterator i = keys.iterator();

   // process ready keys (connections)
   while (i.hasNext())
   {
      SelectionKey key = (SelectionKey) i.next();
      i.remove(); // remove from selected set
      // (key is added automatically with next select() when processing is needed)

      if (key.isAcceptable()) // new connection?
      {
         // get channel and accept connection
         ServerSocketChannel ssc = (ServerSocketChannel) key.channel();
         SocketChannel sc = ssc.accept();
         sc.configureBlocking(false); // set channel to non-blocking mode

         sc.register(selector, SelectionKey.OP_READ|SelectionKey.OP_WRITE);
         // register SocketChannel with selector for multiplexing
         // announce interest for reading and writing

         Socket s = sc.socket();
         System.out.println("new connection ("+s.getInetAddress()+":"+s.getPort()+")");

         // send a greeting to the client
         sc.write(encoder.encode(CharBuffer.wrap("Hallo!\r\n")));
      }
      else if (key.isReadable()) // existing connection, ready for reading input?
      {
         processReadableKey(key); // read input from channel
      }
      else if (key.isWritable()) // existing connection, ready for sending data?
      {
         processWritableKey(key); // write output to channel
      }
   }
}

/*
 * this method is used to handle input from clients.
 */
protected void processReadableKey(SelectionKey key) throws IOException
{
   // get the channel for this key
   ReadableByteChannel channel = (ReadableByteChannel) key.channel();
```

```
      buffer.clear(); // clear buffer before reading

      // read input data into buffer
      int numBytesRead = channel.read(buffer);
      if (numBytesRead < 0)
      {
         // this is an undocumented feature - it means the client has disconnected
         closeConnection(key);
         return;
      }
      else
      {
         buffer.flip(); // flip buffer for reading

         // process data in buffer here ...
      }
   }

   /*
    * this method is used for responding directly to client requests.
    * global game update messages are sent throw game.
    */
   protected void processWritableKey(SelectionKey key) throws IOException
   {
      buffer.clear(); // clear buffer before writing

      // write your data in the buffer here ...

      // get the channel for this key
      WritableByteChannel channel = (WritableByteChannel) key.channel();
      channel.write(buffer); // write data in buffer to channel
   }

   /*
    * close the connection and cancel the related key,
    * so it will be removed form selector later
    */
   private void closeConnection(SelectionKey key)
   {
      try
      {
         key.channel().close(); // close channel
      }
      catch (Exception ex)
      {
      }
      finally
      {
         key.cancel(); // cancel key
      }
   }

   /*
    * utility method to call Thread.sleep()
    */
   private void threadSleep(long time)
   {
      try  {  Thread.sleep(time);  }
      catch (InterruptedException e) {}
   }

   /*
    * start server on port 8000
    */
   public static void main(String[] args)
   {
      Server server = new Server(8000);
      server.start();
   }
}
```

Der Client

Um den Server testen zu können, benötigen wir jetzt noch einen passenden Client. Natürlich könnte man der Einfachheit halber auch einen Terminal-Client wie z.B. Telnet benutzen, aber nachdem wir ja lernen wollen, wie man einen Client mit der Java NIO API entwickelt, werden wir uns noch kurz mit den notwendigen Änderungen gegenüber dem Server befassen.

Da der Client nur über eine einzige Verbindung zum Server verfügt, macht hier ein Multiplexen keinen Sinn und somit entfällt auch die Notwendigkeit ein *Selector*-Objekt zu verwenden. Stattdessen erzeugen wir ein *InetSocketAddress*-Objekt mit passender Serveradresse und -port und Verbinden über dieses einen *SocketChannel* direkt mit dem Server. Der Verbindungsaufbau wird dabei über die Methode `connect(SocketAddress remote)` initiiert. Da der Verbindungsaufbau aber bei einem asynchron-betriebenen *SocketChannel* unter Umständen etwas verzögert durchgeführt wird und man beim Zugriff auf den *SocketChannel* während des Verbindungsaufbaus auch lediglich *Exceptions* bekommt, kann und sollte man an dieser Stelle über die Methode `finishConnect()` zuerst das Ende des Verbindungsaufbaus abwarten, bevor man mit dem eigentlichen Datenaustausch über den *SocketChannel* beginnt.

Der Quellcode für den Client könnte in ungefähr so aussehen:

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

public class Client implements Runnable
{
   private Thread thread;  // the client thread

   // server address and port
   private String host;
   private int port;

   // SocketChannel for client (connection to server)
   private SocketChannel channel = null;

   // buffer for reading and writing
   private ByteBuffer buffer = ByteBuffer.allocate(1000);

   // Charset and encoder for US-ASCII
   private static Charset charset = Charset.forName("US-ASCII");
   private static CharsetEncoder encoder = charset.newEncoder();


   public Client(String host, int port)
   {
      this.host = host;
      this.port = port;
      init();
   }

   /*
    * connect client to given host and port
    */
   private void init()
   {
      try
      {
         // create address object for given host and port
         InetSocketAddress isa =
            new InetSocketAddress(InetAddress.getByName(host), port);
         channel = SocketChannel.open();   // open a channel
         channel.configureBlocking(false); // set channel to non-blocking mode
         channel.connect(isa); // connect channel to given address
```

```
      // set a timeout of 5 seconds and wait for connect to finish
      long timeout = System.currentTimeMillis() + 5000;
      while(!channel.finishConnect())
      {
         threadSleep(250);
         if (System.currentTimeMillis() > timeout)
         {
            throw new Exception("connection timout!");
         }
      }
   }
   catch (Exception ex)
   {
      ex.printStackTrace();
      cleanup();
   }
}

public String getHost()
{
   return host;
}

public int getPort()
{
   return port;
}

/*
 * start the client (if not started already)
 */
public void start()
{
   if (thread == null)
   {
      thread = new Thread(this);
      thread.start();
   }
}

/*
 * stop the client
 */
public void stop()
{
   thread = null;
}

/*
 * processing loop for client
 */
public void run()
{
   try
   {
      // while not stopped
      while (thread == Thread.currentThread())
      {
         processIncomingMessages();

         // do game related things like rendering here ...
         // send output to server here ...

         // rest a bit and give time to other threads
         threadSleep(50L);
      }
   }
   catch (Exception ex)
   {
      System.out.println("error occured! connection terminated!");
      ex.printStackTrace();
   }
   finally
   {
```

```
         //cleanup();
      }
   }

   /*
    * close channel (if existing) and exit
    */
   private void cleanup()
   {
      if (channel != null)
      {
         try { channel.close(); }
         catch (Exception ex) {}
      }
      System.exit(0);
   }

   private void processIncomingMessages() throws IOException
   {
      ReadableByteChannel rbc = (ReadableByteChannel) channel;

      buffer.clear(); // clear buffer before reading

      // read input data into buffer
      int numBytesRead = rbc.read(buffer);
      if (numBytesRead < 0)
      {
         // this is an undocumented feature - it means the client has disconnected
         System.out.println("connection terminated");
         cleanup();
         return;
      }
      else if (numBytesRead > 0) // is there some input?
      {
         try
         {
            buffer.flip(); // flip buffer for reading
            CharBuffer cb = charset.decode(buffer); // read from buffer and decode
            System.out.println(cb.toString()); // display message on console
         }
         catch (Exception ex)
         {
            ex.printStackTrace();
         }
      }
   }

   /**
    * utility method to call Thread.sleep()
    */
   private void threadSleep(long time)
   {
      try  {  Thread.sleep(time); }
      catch (InterruptedException e) {}
   }

   /*
    * start client and connect to server on localhost port 8000
    */
   public static void main(String[] args)
   {
      Client client = new Client("localhost",8000);
      client.start();
   }
}
```

Arbeitsweise eines ByteBuffers

Da das Lesen und Schreiben bei *SocketChannels* über *ByteBuffer* funktionniert, wollen wir uns auch noch kurz mit den Besonderheiten eines *ByteBuffers* befassen. Ein *ByteBuffer* hat ein feste Größe und wird über die statische Methode `allocate(int capacity)` der Klasse *ByteBuffer* erzeugt. Er speichert beliebige Daten in Form von Bytes und bietet verschiedene Arten des Zugriffs auf seinen Inhalt:

- Lese / Bearbeite Wert an aktueller Position
- Lese / Bearbeite alle Werte nach der aktuellen Position
- Lese / Bearbeite alle Werte vor der aktuellen Position

Die folgenden Attribute sind dabei für den Zugriff auf den Inhalt wichtig:

- *capacity* - gibt die aktuelle Anzahl der Elemente an, die der *ByteBuffer* enthält
- *limit* - Index für das erste Element, daß nicht gelesen oder überschrieben werden sollte
- *position* - Index für das nächste zu verarbeitende Element

Bei jedem Lese- oder Schreibzugriff auf den *ByteBuffer* erhöht sich das Attribut *position* um den Wert 1. Wird dabei der *limit* Wert überschritten, dann erzeugt der *ByteBuffer* eine *BufferUnderflowException*. Wird dagegen der *capacity* Wert überschritten, dann erzeugt der *ByteBuffer* eine *BufferOverflowException*. Diese *Exceptions* sollten abgefangen werden, um ein fehlerhaftes Benutzen des *ByteBuffers* zu verhindern, da der *ByteBuffer* selbst keine internen Mechanismen hat, um ein Auftreten von solchen Ausnahmesituationen zu unterbinden.

Das Lesen eines Wertes erfolgt über eine entsprechende *get*-Funktion und das Schreiben eines Wertes über eine passende *put*-Funktion. Bevor die Werte jedoch, nach einem Schreiben in den *ByteBuffer*, wieder ausgelesen werden können, muß dieser zuerst über die Methode `flip()` "zurückgespult" werden. Dabei wird der *limit* Wert auf den Wert von *position* und das Attribut *position* auf den Wert 0 gesetzt. Die Anzahl der Bytes, die anschließend gelesen werden können, erhält man über die Methode `remaining()`, die die aktuelle Differenz zwischen den Attributen *limit* und *position* zurückgibt. Will man das Auslesen der Werte kurzzeitig unterbrechen, um neue Werte in den *ByteBuffer* zu schreiben, ohne dabei jedoch die noch nicht verarbeiteten Werte zu überschreiben, dann kann man dies über die Methode `compact()` tun. Diese kopiert nämlich alle noch nicht verarbeiteten Werte, also alle Werte zwischen *position* und *limit* an den Anfang des *ByteBuffers* und setzt anschließend *limit* auf den Wert von *capacity* und *position* auf die erste Position nach dem letzten nicht-verarbeiteten Wert.

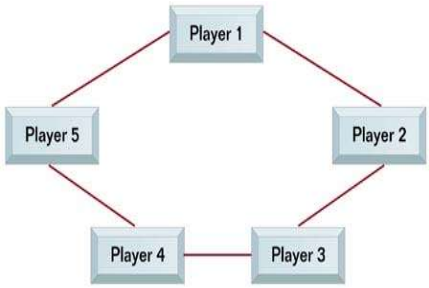
b) Client-Server Architektur

Bevor man nun mit dem Schreiben des Netzwerkcodes beginnt, muß man sich noch ein paar Gedanken über die Netzwerkarchitektur machen. Diese beeinflußt nämlich sehr stark Performanz und Komplexität der Anwendung. Spricht man bei Netzwerken von Performanz, dann meint man in der Regel damit 2 Faktoren: *Bandbreite* und *Latenz*. Die *Bandbreite* legt den Durchsatz einer Verbindung fest und kann recht gut mit der Dicke einer Rohrleitung verglichen werden. Die *Latenz* dagegen ist die Verzögerungszeit, die beim Verschicken von Netzwerkpaketen über das Internet auftritt. Diese müssen in der Regel nämlich zahlreiche Router und Switches durchlaufen und werden so oftmals durch die Unregelmäßigkeiten des Internets, wie z.B. Überlastungen, Ausfälle, Verbindungs- und Paketverluste, aufgehalten oder verzögert ausgeliefert. Dies hat natürlich erhebliche Auswirkungen auf eine kontinuierliche Kommunikation und muß daher auch bei der Wahl der Netzwerkarchitektur berücksichtigt werden.

Insgesamt gibt es 3 Alternativen, die unterschiedliche Vor- und Nachteile haben.

Ring Topologie

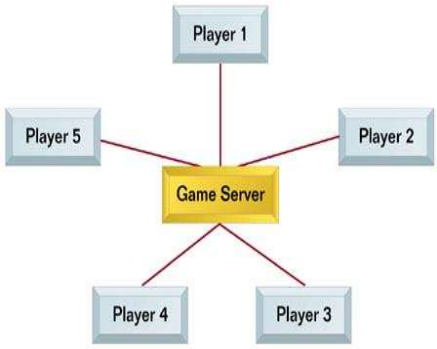
Bei einer ringförmigen Architektur sind die Clients jeweils mit ihren direkten Nachbarn verbunden und können nur über diese mit den übrigen Clients kommunizieren. Pakete mit Statusinformationen oder Positionsangaben werden hier einfach durchgereicht und jeder Client filtert die Informationen heraus, die für ihn relevant sind. Ein Vorteil dieser Variante ist es, daß man keinen dedizierten Server benötigt und trotzdem eine Mehrspielerfunktion realisieren kann, z.B. bei einfachen Spielen oder Situationen in denen der Einsatz eines Servers unmöglich ist. Ein großer Nachteil aber ist, daß bei einem Ringnetz die Performanz von der Verbindungsqualität eines jeden einzelnen Clients beeinflußt wird und daß ein Verbindungsverlust zwischen 2 Clients schon die Kommunikation zum Erliegen bringen kann.



Quelle: Gamasutra - "Designing Fast-Action Games For The Internet"

Stern Topologie

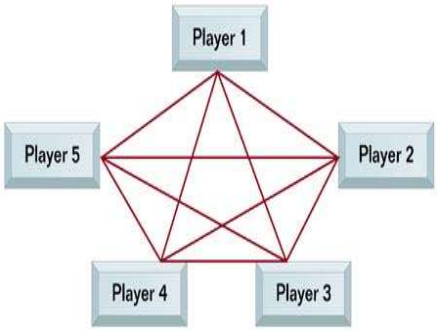
Bei einer sternförmigen Architektur sind alle Clients mit einem zentralen Server verbunden, welcher sich um Kommunikation und Spiellogik kümmert und gleichzeit die Möglichkeit einer zentralen Verwaltung und Kontrolle bietet. Durch die sternförmige Vernetzung ist man unabhängig von der Verbindungsqualität einzelner Clients und muß so nicht mehr Performanzeinbußen bei Kommunikation und Synchronisation hinnehmen, die z.B. bei der *Ring Topologie* durch schlechte Verbindungen einzelner Clients oder sogar komplette Verbindungsverluste auftreten können. Die *Stern Topologie* ist damit robuster und weniger fehleranfällig und ermöglicht es einzelne Clients individuell und abgestimmt auf deren Bandbreite und Latenzzeiten zu behandeln. Ein Nachteil ist jedoch der Zusatzaufwand, den hier man für die Entwicklung des Servers betreiben muß und natürlich die hohen Anforderungen, die an die Verbindungsqualität des Servers gestellt werden. Dieser dient nämlich als zentraler Verteiler und muß daher in der Lage sein, eine große Anzahl an Clients zu bedienen und benötigt deshalb auch eine Verbindung mit relativ hoher Bandbreite und möglichst kurzer Reaktionszeit. Die *Stern Topologie* wird in der Regel für Spiele mit einer hohen Spieleranzahl verwendet, oder wenn ein zentrales Management erforderlich ist.



Quelle: Gamasutra - "Designing Fast-Action Games For The Internet"

All-to-All Topologie

Die *All-to-All Topologie* ist ein Mischung aus *Ring* und *Stern Topologie* und versucht Vorteile aus beiden Welten in sich zu vereinen. Jeder Client ist hier mit allen anderen Clients direkt verbunden und kann so mit diesen direkt kommunizieren. Zum Einen wird dadurch die Abhängigkeit der Gesamt-Netzwerkperformanz von der Verbindungsqualität einzelner Clients vermindert, sowie die Robustheit des Netzes insgesamt verbessert. Zum Anderen wird der Einsatz eines dedizierten Servers überflüssig. Dies führt jedoch, aufgrund der verteilten Logik und dadurch weitaus komplexeren Synchronisationsanforderungen, zu einer erhöhten Komplexität des Netzwerkcodes, sowie zu einem ebenfalls erhöhten Bandbreitenbedarf auf Seiten der Clients. Diese müssen nämlich nun die (n-1)-fache Menge an Verbindungen aufrechterhalten, was zu einer deutlich höheren Belastung der Netzwerkverbindung führt. Aufgrund dieser Tatsache ist diese Art der Netzwerkarchitektur auch nur für Spiele mit einer kleinen und begrenzten Anzahl von Spielern geeignet.



Quelle: Gamasutra - "Designing Fast-Action Games For The Internet"

c) Synchronisation von verteilten Systemen

Alle verteilten Anwendungen müssen normalerweise in irgendeiner Form synchronisiert werden. Die Synchronisation erfolgt in der Regel über Nachrichten, die die einzelnen Systeme untereinander austauschen. Die Kommunikationsrichtung kann dabei sowohl einseitig (z.B. Benachrichtigung / Statusabfrage) also auch zweiseitig sein. Die Nachrichtenübermittlung kann über verschiedene Medien wie z.B. Dateien, Netzwerkströme oder Datenbanken stattfinden. Eine Nachricht ist eine Informationseinheit, die sowohl einzelne Statuswerte als auch ganze Binärdatenketten enthalten kann.

Will man ein Spiel mit Mehrspieleroption erstellen, dann muß man sich zwangsläufig mit diesem Thema auseinandersetzen. Da Spiele in der Regel auch noch Echtzeitsysteme sind, benötigen sie zudem eine extrem zeitkritische Synchronisation. Diese ist nämlich besonders wichtig für eine flüssige Animation, da der Spieler nicht erst auf eintreffende Pakete warten will, bevor endlich der nächste Frame gezeichnet wird, und auch sonst jede Verzögerung im Spielfluß als extrem störend empfindet. Um eine flüssige und schnelle Synchronisation zu gewährleisten, bedarf es jedoch einiger Tricks, besonders dann, wenn die Kommunikation über das Internet stattfinden soll. Dieses ist nämlich nicht für einen kontinuierlichen und zeitkritischen Datenaustausch ausgelegt und kann daher auch keine festen Übertragungszeiten für Pakete garantieren. Durch Überlastung und Paketverlust kommt es oftmals zu einer Verzögerungen bei der Auslieferung von Paketen, die dann durch die Anwendung abgefangen werden muß. Hierfür gibt es verschiedene Ansätze und Techniken, von denen wir nachfolgend zwei etwas genauer betrachten wollen.

Brute Force Methode

Die wohl einfachste und auch häufig angewandte Methode ist die *Brute Force* Methode. Wie der Name schon vermuten läßt, werden hier vom Server in regelmäßigen und relativ kurzen Intervallen eine große Anzahl von Positionsupdate- und Statuspaketen an alle Clients verschickt, wodurch natürlich die Netzwerkverbindung permanent stark belastet wird. Um eine relativ kurze Laufzeit der Pakete zu gewährleisten verwendet man hierfür in der Regel das *UDP* Netzwerkprotokoll. Dieses

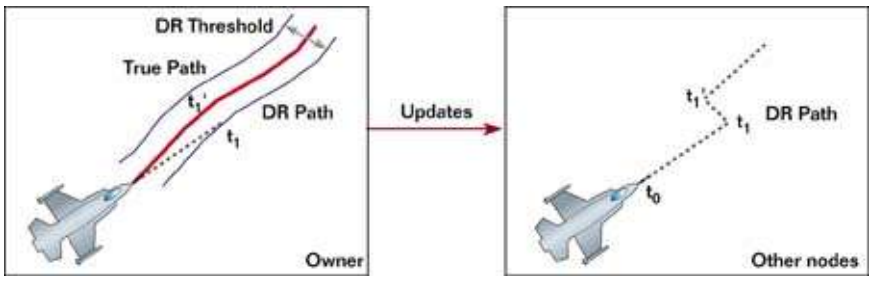
garantiert im Gegensatz zu *TCP* nicht die Auslieferung einzelner Pakete und auch nicht deren korrekte Reihenfolge. Dafür werden die Pakete aber schneller übertragen und eingetroffene Pakete können auch, anders als bei *TCP*, sofort verarbeitet werden. Bei *TCP* müssen nämlich erst alle Pakete in der korrekten Reihenfolge eingetroffen sein, bevor die empfangenen Pakete schließlich zur Verarbeitung an die Anwendung weitergegeben werden. Kann man ohne eine garantierte Auslieferung der Pakete leben und spielt die Reihenfolge ebenfalls keine Rolle, dann ist man mit *UDP* gut beraten. Das trifft z.B. für Positionsangaben zu. Diese werden nämlich in regelmäßigen Intervallen neu gesendet, weshalb beim Verlust eines solchen Paketes auch kein größerer Schaden entsteht. Der Client hat ja schließlich die letzte Positionsangabe und kann so bei Bedarf anhand von Bewegungsrichtung und -geschwindigkeit die nächste Position problemlos abschätzen. Da die Sendeintervalle recht kurz sind, müssen in der Regel auch nur sehr kleine Zeiten überbrückt werden, wodurch eine Abweichung zwischen geschätzter und tatsächlicher Position sehr klein sein sollte und vom Client normalerweise interpoliert werden kann. Treten irgendwann größere Abweichungen auf, z.B. durch Paketverluste bei einer zeitweise schlechten Verbindung, dann können diese durch ein "Teleportieren" an die aktuelle Position behoben werden. Dies ist jedoch eine absolute Notfallmaßnahme, um eine Synchronisation wiederherzustellen und sollte daher auch nur in Extremfällen verwendet werden, da der Spieler derartige Positionssprünge als extrem störend wahrnimmt.

Da eine Auslieferung von Paketen mittels *UDP* nicht gewährleistet werden kann, muß man für Nachrichten, die nicht verloren gehen dürfen, wie z.B. "Tot des Spielers" oder "Treffer" unbedingt *TCP* verwenden, oder aber ein eigenes Empfangsbestätigungssystem für *UDP* implementieren, daß die betroffenen Nachrichten dann bei Nicht-Empfang einer Bestätigung erneut versendet. Aufgrund der Komplexität eines solchen Systems und der Tatsache, daß *TCP* dafür eigentlich bestens geeignet ist, sollte man wenn möglich, die erste Variante bevorzugen.

Keyframing: Ein Möglichkeit, um die übertragene Datenmenge bei der *Brute Force* Methode ein bischen zu reduzieren, wäre anstatt der absoluten Positionsangaben relative Positionsangaben zu verwenden, die dann nur die Änderungen (Deltas) zur letzten absoluten Position enthalten. Da das Ausliefern der Pakete aber nicht gewährleistet werden kann, müßte man hier in regelmäßigen Intervallen wieder absolute Positionsangaben verschicken, mit denen sich dann der Client synchronisieren kann. Diese Methode nennt man in der Animation auch *keyframing*.

Dead Reckoning

Bei dieser Methode werden aktuelle Positionen aufgrund von alten Positionsangaben, vergangener Zeit, sowie Bewegungsrichtung und -geschwindigkeit geschätzt und vorausberechnet. Die Methode stammt aus dem militärischen Bereich und wird dort zur Simulation und Synchronisation von Objekten (typischerweise Fahrzeuge) in 3-dimensionalen Welten verwendet. Ihr Ziel ist es, die übertragene Datenmenge so gering wie möglich zu halten und gleichzeitig eine akkurate Synchronisation zu gewährleisten. Beim *Dead Reckoning* geht man davon aus, daß das zu schätzende Objekt einer Bewegungskurve folgt und daß geringe Abweichungen davon auf dem Client nicht besonders stark wahrgenommen werden. Da geringe Positionsunterschiede in der Regel auch nicht nicht problematisch für die Kollisionserkennung (z.B. Zusammenstoß mit anderen Fahrzeugen, Treffer durch Granate) sind, kann man hier mit kleinen Ungenauigkeiten und Interpolation gut leben, solange die Abweichung einen bestimmten Grenzwert nicht überschreitet. Wird dieser Wert jedoch überschritten, dann werden die Objekte einfach an die neue Position "teleportiert" und die Vorausberechnung an dieser Stelle fortgesetzt. Da der Client den Bewegungsverlauf der Objekte simuliert, können Positionsupdates vom Server in der Regel in größeren Intervallen verschickt werden. Da das Empfangen und Verarbeiten der Pakete dabei im Hintergrund und unabhängig von der Animation erfolgt, verwendet man hierfür normalerweise das TCP Protokoll. Durch die garantierte Auslieferung der Pakete können relative Positionsangaben verschickt werden, wodurch die übertragene Datenmenge weiter gesenkt werden kann.



Quelle: Gamasutra - Dead Reckoning Latency Hiding for Networked Games

Ein wichtiger Punkt den man jedoch beim Einsatz dieser Methode nicht vergessen darf, ist die Zeitverschiebung der Animation zwischen Server und Client. Da die Übertragung der Pakete nämlich einige Zeit in Anspruch nimmt und meistens auch *lag*-bedingt unterschiedlich lange dauert, vergehen normalerweise mehrere Frames zwischen dem Abschicken einer Nachricht und dem Feedback durch den Server. Diese Verzögerung in der Reaktionszeit muß man in der Anwendung berücksichtigen. Außerdem ist es natürlich notwendig, die unterschiedlichen Uhren und Zeitsysteme der beteiligten PCs in irgendeiner Form zu synchronisieren. Dies ist jedoch nicht immer trivial, da aufgrund der Latenzzeiten für eine Synchronisation über das Internet spezielle Techniken angewand werden müssen. Auch hier gibt es verschiedene Ansätze, die sich mit dem Thema beschäftigen und versuchen das Problem der Zeitsynchronisation zu lösen. Einer davon ist das z.B. das Verwenden von *Vektorzeit*. Da dieser Bereich allerdings sehr komplex ist, soll an dieser Stelle ein kurzer Hinweis reichen, um dem geneigneten Leser einen Einstiegspunkt für weitere Studien zu geben.

2.6. Threads und Performance

Ein sehr wichtiger Punkt bei der Entwicklung von Spielen, besonders unter Java, ist die Perfomance. Diese kann nämlich durch den Programmierer sowohl positiv, z.B. durch den gezielten Einsatz von optimierten und spezialisierten Klassen und Bibliotheken, also auch negativ, z.B. durch schlechten Programmierstil, schlechtes Design oder einfach nur Unwissenheit, beeinflusst werden. Diese Einflüsse sind aber weitestgehend unabhängig von der verwendeten Programmiersprache und müssen daher generell beachtet werden!

Multi-Threading

Das Parallelisieren von Abläufen durch verschiedene Threads bringt zwar neue Möglichkeiten, aber auch neue Probleme mit sich. Es ist oft sehr komfortabel, verschiedene Funktionen gleichzeitig ablaufen zu lassen, da man sich so keine Gedanken über eine gerechte Zeitverteilung machen muß . Allerdings haben Threads auch ihre Tücken und benötigen nicht nur zusätzlichen Verwaltungsaufwand durch die JVM, sondern erhöhen auch oftmals stark die Komplexität von Funktionen, besonders wenn diese *thread-safe*, also von mehreren Threads gemeinsam benutzbar, sein sollen. Spätestens dann nämlich muß man sich Gedanken über mögliche *Deadlocks* machen und die betroffenen Code-Passagen mit *synchronized*-Anweisungen sichern.

Da Spiele jedoch meistens linear ablaufen, kann und sollte man sich in der Regel diesen Aufwand sparen! Betrachtet man nämlich den generellen Ablauf eines Spiels, dann ähnelt dieser fast immer der folgenden Stuktur:

```
while(!finished)
{
   // check and handle user input
   ...

   // move actors and modify game state
   ...

   playField.renderScreen(); // render screen to BackBuffer
   playField.updateScreen(); // draw BackBuffer on screen

   // wait a bit before next loop
   try
   {
      Thread.sleep(150);
   }
   catch (InterruptedException e) {}
}
```

Game-Loop

Betrachtet man diese Struktur, dann stellt man fest, daß es keinen Sinn machen würde die Abläufe zu parallelisieren, da man erst dann den Bildschirm neuzeichnen muß, wenn sich dort auch wirklich etwas verändert hat. Und bevor man etwas verändern kann, sollte man zuerst überprüfen, ob vielleicht Steuerungsinformationen vom Spieler vorliegen. Dieser Ablauf wiederholt sich dann bei jedem Bild, daß auf den Bildschirm gezeichnet wird, und wird daher als Schleife realisiert. Diese Schleife ist quasi der "Herzschlag" eines Spieles und wird *Game-Loop* oder oft auch *Rendering-Loop* genannt.

Framerate

Die Anzahl der Bilder (*Frames*) pro Sekunde, die über den *Game-Loop* auf den Bildschirm gezeichnet werden, nennt man auch *Framerate*. Die *Framerate* sollte für eine flüssige Animation mindestens bei 25 *Frames* / Sekunde liegen und am besten sogar der aktuellen Bildschirmaktualisierungsrate (75, 85 oder 100 Hertz) gleichkommen.

Eine Maßnahme, um auch bei performance-schwächeren Systemen eine akzeptable *Framerate* und Spielgeschwindigkeit zu erreichen, ist das *Frame Skipping*, das Auslassen von einzelnen Frames bei Zeitknappheit. Die Steuerung wird dabei über den *Game-Loop* vorgenommen und benötigt ein möglichst genaues Timing über einen *high-resolution Timer*.

2.7. Logging & Fehlersuche

Da auch Programmierer nur Menschen sind, unterläuft selbst den Besten unter ihnen hin und wieder ein Fehler, der dann in der Regel zu einer Fehlfunktion des Programmes führt. Je komplexer dabei der Programmcode bzw. die Klassenstruktur ist, desto höher ist die Wahrscheinlichkeit kleine Dinge zu übersehen und somit Fehler zu erzeugen. Für das *Debugging*, also die Fehlersuche und -eingrenzung, bieten gängige Entwicklungsumgebungen meistens gute Werzeuge. Mit zunehmender Komplexität der Programme, kann ein manuelles *Debugging,* über die Tools der Entwicklungsumgebung, aber sehr aufwendig werden, weshalb es sich hier oft anbietet, alternative Wege zu gehen und ein intelligentes *Logging* zu betreiben. Unter *Logging* versteht man das Mitprotokollieren von Methodenaufrufen und Inhalten von Variablen, welches bei gezielter Einsetzung eine Eingrenzung des Fehlerbereiches sehr erleichtern kann. Dadurch gewinnt man nicht nur an Geschwindigkeit bei der Fehlersuche, sondern kann den Prozess sogar teilweise automatisieren bzw. durch vorangegangene *Debugging*-Durchläufe gewonnene Erfahrungen und Erkenntnisse festhalten und später wiederverwenden. Oftmals ist das *Logging* auch die einzige

Alternative zur Fehlersuche, wie z.B. bei verteilten Anwendungen, also Client-Server Programmen, wo ein manuelles *Debugging* meistens nur sehr schwer bis gar nicht zu realisieren ist.

Log4J

Ein einfaches *Logging* läßt sich in Java über die Funktion `System.out.println(String s);` realisieren. Diese reicht zwar für eine Vielzahl von Anwendungsfällen aus, ist aber nicht besonders komfortabel und auch leider nicht sehr flexibel. Daher empfiehlt sich der Einsatz einer speziellen *Logging*-Bibliothek wie z.B. *Log4J*. *Log4J* bietet eine Vielzahl von Funktionen und ermöglicht eine einfaches An -und Abschalten des *Loggings*, sowie ein Umleiten der Ausgabe in Dateien oder sogar Netzwerk-*Streams*. Desweiteren kennt *Log4J* verschiedene *Logging*-Level, über die die Anzahl der mitprotokollierten Aktionen, also die "Gesprächigkeit" der Anwendung konfiguriert werden kann.

Um ein einfaches *Logging* mit *Log4J* realisieren, reichen schon folgende Anweisungen:

`Logger.getLogger(MyLogger.class);`	erzeugt ein *Logger*-Objekt für die Klasse
`BasicConfigurator.configure();`	konfiguriert *Log4J* für das *Logging* auf der Konsole
`logger.setLevel(Level.DEBUG);`	setzt den *Logging*-Level für das *Logger*-Objekt. *Log4J* kennt die Level *OFF*, *INFO*, *DEBUG*, *WARN*, *ERROR*, *FATAL* und *ALL*.
`logger.info("this is readable in INFO level");`	gibt den übergebenen Text aus, wenn ein *Logging*-Level gewählt wurde, der größer oder gleich dem *INFO*-Level ist. Analog dazu gibt es auch entsprechende Funktionen für die anderen *Logging*-Level.
`Logger.getRootLogger().setLevel (Level.DEBUG);`	setzt den *Logging*-Level für das *RootLogger*-Objekt und damit auch für alle anderen *Logger*-Objekte

Für jede Klasse wird ein *Logger*-Objekt erzeugt, daß das *Logging* für diese Klasse realisiert. Die einzelnen *Logger*-Objekte sind Teil einer Hierarchie, die analog zur Klassenhierarchie aufgebaut ist. Das *RootLogger*-Objekt ist dabei immer das oberste Objekt der Hierarchie und beeinflußt dadurch auch das Verhalten aller untergeordneten *Logger*-Objekte. Über das *RootLogger*-Objekt läßt sich so z.B. recht einfach das *Logging*-Verhalten der gesamten Anwendung konfigurieren.

Die *Logging*-Level sind ebenfalls Teil einer Hierarchie und ermöglichen dadurch eine gezielte Zuordnung und Ausgabe von *Logging*-Statements, geordnet nach der Relevanz und Wichtigkeit ihrer Informationen. Dabei ist zu beachten, daß höhere Level niedrigere Level miteinschließen, also das z.B. im *DEBUG*-Level auch Informationen aus dem *INFO*-Level mitangezeigt werden, während im *INFO*-Level dagegen Statements aus dem *DEBUG*-Level ignoriert werden.

Ein einfaches Programmcodebeispiel:

```
import org.apache.log4j.*;

public class MyLogger
{
  // Logger object for this class
  private static Logger logger = Logger.getLogger(MyLogger.class);

  public static void main(String[] args)
  {
    // Set up a simple configuration that logs on the console.
    BasicConfigurator.configure();

    // Set logging level to DEBUG for all Logger objects
    Logger.getRootLogger().setLevel(Level.DEBUG);

    logger.info("this is readable in INFO and DEBUG level");
    logger.debug("this is readable in DEBUG level only");
  }
}
```

Profiling

Ein Thema, mit dem wir uns an dieser Stelle noch ganz kurz befassen wollen, ist das *Profiling* von Anwendungen. Das *Profiling* dient nämlich zum Einen der Performance-Analyse und dem Aufdecken und Beseitigen von Flaschenhälsen, sowie der Optimierung von Speichernutzung, Objekterzeugung und Einsatz des Garbage-Collectors. Zum Anderen erhält man aber auch interessante Informationen über Auslastung und Laufzeiten von Threads und kann so einen kleinen Eindruck über die Arbeit bekommen, die die JVM "hinter den Kulissen" durchführt. Durch das *Profiling* von Anwendungen kann man diese nicht nur optimieren und auf einen deutlich höheren Qualitätslevel befördern, sondern auch gleichzeitig noch jede Menge über die Arbeitsweise von Java lernen, um so schließlich verschiedene interne Zusammenhänge besser zu verstehen. Dies ist besonders für die Spieleprogrammierung unter Java notwendig, da hier Performance und Laufzeit im Vordergrund stehen und der Programmierer deshalb auch wissen muß, wie er diese beiden Faktoren beeinflussen kann!

Java bietet eine einfache *Profiling*-Funktion über die JVM durch die Verwendung der Option *-prof* beim Start des Java-Programmes an. Allerdings werden hierbei alle Informationen in Textdateien geschrieben, die anschließend vom Anwender in mühseliger Handarbeit analysiert werden müssen. Dies erfordert nicht nur gute Kenntnisse der Dateistruktur, sondern ist recht unübersichtlich und bietet auch nur ein statisches Gesamtbild des Programmablaufes nachdem das Programm beendet wurde. Will man jedoch Live-Informationen über aktuelle Speichernutzung oder Performance erhalten, dann muß man an dieser Stelle auf ein externes Tool zurückgreifen. Ein sehr gutes und komfortables Programm ist z.B. die *OptimizeIt*-Suite von Borland. Allerdings ist diese kommerziell und nur als zeitlich begrenzte Testversion erhältlich. Für die Eclipse-Entwicklungsumgebung gibt es mittlerweile schon einige freie *Profiling*-Plugins, z.B. *Eclipse Profiler Plugin* http://eclipsecolorer.sourceforge.net/index_profiler.html und *JMechanic* http://jmechanic.sourceforge.net/, die sich allerdings noch in der Entwicklung befinden und daher auch noch nicht ganz so komfortabel wie z.B. *OptimizeIt* sind. Ein interessanter externer Profiler ist außerdem EJP (*Extensible Java Profiler*), welcher als OpenSource-Programm unter http://ejp.sourceforge.net/ erhältlich ist.

2.8. Build & Deploy

Da auch Spiele in der Regel komplexe Systeme sind, können sie oft recht schnell über den Umfang von ein paar Klassen hinauswachsen. Spätestens dann aber macht es Sinn, die ganzen Dateien in Archiven zusammenzusammen, um so ein einfacheres Handling durch den Spieler zu ermöglichen. Java bietet mit seinen JAR-Dateien schon das perfekte Medium dafür. Die Ressourcen werden hier nämlich nicht nur komprimiert und zusammengefasst, sondern man kann auf sie auch problemlos und komfortabel aus Java-Programmen heraus zurückgreifen. Allerdings ist hierfür eine spezielle Vorgehensweise notwendig, die bereits in Kapitel *2.3 Sound* unter *Verwenden von Resourcen in JAR-Dateien* beschrieben wurde.

Build

Da das Erstellen von JAR-Dateien nicht immer ein einfacher Prozess und zudem auch oft noch zeitaufwendig ist, sollte man sich spätestens an dieser Stelle Gedanken über eine automatisierte Lösung machen. Viele Entwicklungsumgebungen bieten zwar mehr oder weniger komfortable Möglichkeiten JAR-Dateien zu erstellen, allerdings bleiben sie oft auch schon an dieser Stelle stehen und der Programmierer muß sich dann um alle weiteren Dinge selbst kümmern.
Abhilfe kann hier *Ant* (http://ant.apache.org/) schaffen, daß als freies Build-Tool für Java von der Apache / Jakarta Gruppe (http://jakarta.apache.org/) entwickelt wird. *Ant* ist nicht nur in der Lage Verzeichnisse zu erstellen, Dateien zu kopieren und Java-Klassen zu kompilieren, sondern kann auch JAR-Dateien erstellen, signieren und bietet sogar eine Unterstützung für den Zugriff auf CVS-Server an. Dazu kommen dann noch Features wie z.B. der Zugriff auf FTP Server, die ein automatisiertes Veröffentlichen von Dateien auf einem Webserver ermöglichen.

Ant wird durch eine XML-Datei mit dem Namen *build.xml* konfiguriert, in der verschiedene Pfade und Tasks definiert und dadurch die einzelnen Arbeitsschritte des Build-Prozesses festgelegt werden. Einzelne Tasks können dabei unterschiedliche Abhängigkeiten haben und sich auch gegenseitig aufrufen. Zum Ausführen des Build-Prozesses muß man dann nur noch *Ant* mit der passenden Konfigurationsdatei als Parameter starten, oder aber, bei einer Entwicklungsumgebung mit Ant-Unterstützung wie z.B. bei Eclispe, nur noch auf einen Build-Button drücken. Durch diese Vorgehensweise kann man den Build-Prozess weitgehend automatisieren und spart sich so bei größeren Projekten eine Menge Zeit und Frust.

Eine Beispielkonfigurationsdatei für einen einfachen Build-Prozess könnte folgendermaßen aussehen:

```
<project name="MyFirstAntBuild" default="build" basedir=".">

  <!-- set global properties for this build -->

  <property name="src" value="${basedir}/src"/>
  <property name="resources" value="${basedir}/resources"/>
  <property name="build" value="${basedir}/build"/>
  <property name="www" value="\\webserver\myProject"/>

  <!-- set global path elements -->

  <path id="class.path">
    <pathelement path="${classpath}"/>
  </path>

  <!-- define targets -->

  <!-- perform a cleanup-->
  <target name="cleanup">
    <delete dir="${build}"/>
```

```
  </target>

  <!-- perform a cleanup first and create needed directories after -->
  <target name="init" depends="cleanup">
    <tstamp/>
    <mkdir dir="${build}"/>
  </target>

  <!-- compile java files (after init) -->
  <target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${build}" classpathref="class.path"
     excludes="test/**"
     />
  </target>

  <!-- build and sign jar file (after compile) -->
  <target name="build" depends="compile">

    <copy todir="${build}">
      <fileset dir="${resources}">
       <exclude name="cvs/**"/>
       <exclude name="templates/**"/>
      </fileset>
    </copy>

    <jar file="${dist}/myProject.jar"
         manifest="${basedir}/conf/mainclass.mf">
      <fileset dir="${build}"/>
    </jar>

    <signjar jar="${dist}/myProject.jar" alias="YourNameHere"
              keypass="password" storepass="password" />

    <copy file="${dist}/myProject.jar" todir="${www}" overwrite="yes"/>
    <antcall target="cleanup"/>
  </target>
</project>
```

Ant generiert beim Erstellen der JAR-Datei automatisch ein Manifest, wenn keine externe Manifest-Datei angegeben wurde. Soll die JAR-Datei jedoch z.B. durch einen Doppelklick automatisch startbar sein, dann muß man in einer externen Manifest-Datei angeben, welche Klasse aus der JAR-Datei beim Laden gestartet werden soll.

Eine minimale Manifest-Datei wäre:

```
Manifest-Version: 1.0
Main-Class: MyApplication
```

Deploy

Will man sein Spiel schließlich veröffentlichen, dann reicht es zwar, die Archiv-Dateien auf einem Webserver zum Download zur Verfügung zu stellen, allerdings überläßt man so die Installation des Spieles komplett dem Anwender. Die Installation kann aber, besonders bei Verwendung von externen Bibliotheken, recht kompliziert und mühsam sein und wird daher wahrscheinlich viele Anwender erst einmal abschrecken. Um dies zu verhindern und gleichzeitig eine komfortable Möglichkeit der Distribution von Java-Anwendungen zu schaffen, hat Sun die *JavaWebstart*-Technologie entwickelt.

JavaWebstart ist ein Programm, daß etwa seit Java Version 1.4.1 zusammen mit der JVM installiert wird und sich als *MIME*-Type im Webbrowser für Dateien mit der Endung *JNLP* registriert. *JavaWebstart* ist in der Lage, Anwendungen über das Internet zu installieren und hält diese später, wenn gewünscht, durch eine automatische Update-Funktion auf dem aktuellsten Stand. Hierfür wird

eine spezielle *JNLP*-Konfigurationsdatei verwendet, die alle notwendigen Informationen für die Installation enthält. Dazu gehören, neben Name und Bezugsort der Programmarchieve und benötigten Bibliotheken, auch Informationen über die von der Anwendung geforderten Zugriffsrechte. Ähnlich wie bei Java *Applets* laufen nämlich Anwendungen unter *JavaWebstart* in einer Sandbox ab und können nicht so ohne Weiteres Zugriff auf Dateisystem oder Netzwerk erhalten. Ist dies jedoch notwendig, dann müssen alle JAR-Dateien der Anwendung signiert sein und der Anwender wird bei der Installation gefragt, ob er der Quelle vertrauen und die geforderten Zugriffsrechte gewähren will. Handelt es sich bei dem Zertifikat um ein selbst erstelltes Zertifikat, dann warnt *JavaWebstart* den Anwender sogar davor, die Anwendung zu installieren und startet diese nur dann, wenn der Anwender einer Installation trotzdem zustimmt. Wurde das Zertifikat nach einer Installation zwischenzeitlich geändert, dann wird dies dem Anwender beim nächsten Update mitgeteilt und er muß erneut die geforderten Zugriffsrechte bestätigen.

Obwohl viele Nutzer gegenüber *JavaWebstart* sehr skeptisch sind, bietet die Technologie wohl eine der sichersten und komfortabelsten Distributionsmethoden für Java-Programme. Diese werden nämlich lediglich in ein spezielles *Cache*-Verzeichnis auf der Festplatte kopiert und können von dort jederzeit, auch über JavaWebstart, wieder gelöscht werden. Registry-Einträge können und werden bei einer Installation nicht gemacht und auch der Zugriff auf sonstige Komponenten des Betriebssystems wird durch die Ausführung in der Sandbox unterbunden.

Die Verwendung von *JavaWebstart* ist sehr einfach: auf der Client-Seite muß lediglich *JavaWebstart* installiert sein, was bei neueren JVM Versionen meist automatisch der Fall ist. Auf der Server-Seite reicht es, wenn man einen Webserver hat, der den JNLP MIME-Type unterstützt. Danach muß man dann nur noch die Programmdateien zusammen mit der JNLP-Datei auf den Webserver stellen und z.B. aus einer Internetseite heraus auf die JNLP-Datei verlinken. Klickt der Anwender anschließend auf diesen Link, dann wird er von *JavaWebstart* automatisch durch den Installationsprozess geführt und muß dort nur noch evtl. geforderte Zugriffsrechte bestätigen.

Mit *JavaWebstart* ist es möglich, je nach Betriebssystem des Clients verschiedene Bibliotheken für Anwendungen zu installieren, sowie das Ausführen einer Anwendung ausschließlich “online” zu gestatten und so eine Installation gänzlich zu unterbinden. In diesem Falle werden die Anwendungsdateien nur temporär übertragen und müssen bei jedem Start der Anwendung neu geladen werden.

Wer *JavaWebstart* gerne live in Aktion sehen möchte, der kann dies über die *JavaWebstart*-Demos von Sun unter folgender Adresse tun: http://java.sun.com/products/javawebstart/demos.html

Eine einfache Beispielkonfigurationsdatei für *JavaWebstart* könnte folgendermaßen aussehen:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for MyApplication -->
<jnlp
  spec="1.0+"
  codebase="http://www.mydomain.com"
  href="myApplication.jnlp">
  <information>
    <title>My first JavaWebstart Application</title>
    <vendor>YourNameHere</vendor>
    <homepage href="index.html"/> <!-- myApplication Website -->
    <description>This is my first test with JavaWebstart</description>
    <description kind="short">my first JavaWebstart test</description>
    <icon href="myLogo.gif"/>     <!-- the logo image for MyApplication -->
    <offline-allowed/>            <!-- this program can be run offline -->
  </information>
  <resources>
    <j2se version="1.4"/>
```

```
    <jar href="myApplication.jar" main="true"/>
  </resources>
  <resources os="Windows"> <!-- the needed libraries for a windows system -->
    <nativelib href="myApplicationWindowsLibraries.jar"/>
  </resources>
  <application-desc main-class="myApplication"/> <!-- the class to start -->
  <security>
    <all-permissions/>
  </security>
</jnlp>
```

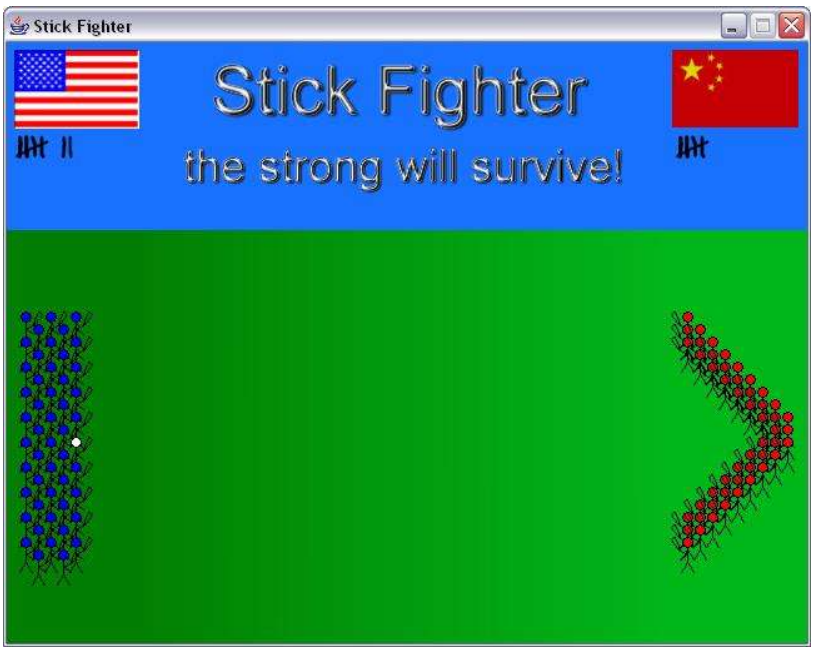
3. Stickfighter - ein Anwendungsbeispiel

3.1. Motivation

Da wir uns während des Studium auch schon privat einige Zeit mit der Spieleprogrammierung unter Java beschäftigt haben, wollten wir in unserer Projektarbeit nun ein Spiel realisieren, daß uns die Möglichkeit geben würde, die bereits erlernten Fähigkeiten unter Beweis zu stellen, sowie neue Erfahrung im Bereich Netzwerktechnik und Internet zu sammeln. Um das Projekt dabei im Rahmen des Machbaren zu halten und uns auch voll auf das eigentliche Ziel der Arbeit zu konzentrieren, nämlich die Entwicklung einer internettauglichen Netzwerkarchitektur für Mehrspieler Spiele, haben wir uns schließlich darauf geeinigt, ein einfaches Spielkonzept mit simpler Grafik zu realisieren.

3.2. Spielprinzip

Es gibt eine begrenzte 2D Spielfläche, die als Schlachtfeld für 2 gegnerische Teams (rot und blau) dient. Jedes Team besteht aus einer Anzahl von Kämpfern, die sich frei über die Spielfläche bewegen und dabei versuchen, mit einer Keule, möglichst viele Gegner kampfunfähig zu machen. Jeder Kämpfer hat eine bestimmte Anzahl an Lebenspunkten, die bei einem Treffer durch eine gegnerische Einheit jeweils um einen Punkt verringert werden. Fallen die Lebenspunkte dabei auf 0, dann wird der Kämpfer kampfunfähig und bleibt noch für einige Sekunden auf dem Spielfeld liegen. Hat ein Team schließlich alle Kämpfer verloren, dann wird das Spiel beendet und das gegnerische Team gewinnt. Jeder Kämpfer wird von einem menschlichen Spieler gesteuert, oder aber vom Computer, wenn nicht genug menschliche Spieler da sind.



3.3. Problembereiche und Lösungsansätze

Im Folgenden möchten wir noch kurz auf ein paar Problembereiche eingehen, die sich bei der Entwicklung von Stickfighter ergeben haben und unsere Lösungsansätze dazu vorstellen.

Übertragene Datenmenge

Die größte Herausforderung war für uns die zu übertragende Datenmenge zwischen dem Server und den Clients. Da es unser Ziel war, bis zu 100 gleichzeitige Client-Verbindungen zu bedienen und den Betrieb des Servers evtl. auch über eine DSL-Leitung zu ermöglichen, haben wir versucht, die übertragene Datenmenge und damit den Bandbreitenbedarf so gering wie möglich zu halten. Deshalb haben wir uns auch für einen *Dead Reckoning* Algorithmus in Verbindung mit *TCP* entschieden, anstatt der sonst üblichen *Brute Force* Methode über *UDP*. Durch die Verwendung von *TCP* wird die Auslieferung der Pakete garantiert, weshalb wir für die Positionsupdate-Nachrichten lediglich Deltas anstelle von absoluten Positionsangaben verschicken müssen und so natürlich mehrere Bytes pro Nachricht einsparen können. Dies macht sich bei 100 Clients und mehreren Updates pro Sekunde auch durchaus bemerkbar. Um die Datenmenge dann noch weiter zu drosseln, haben wir versucht, insgesamt so wenig Positionsupdates wie möglich zu verschicken, im Idealfall 1-2 pro Sekunde und auf dem Client dafür eine intelligente Vorausberechnung zu realisieren. Allerdings hat dieser Ansatz, trotz starker Bemühungen und relativ langer Entwicklungszeit, nicht ganz zu dem gewünschten Ergebnis geführt, da die einzelnen Kämpfer sich oftmals realtiv planlos bewegen und auch jederzeit abrupt die Richtung wechseln können. Dadurch kommt es häufig zu kleineren Positionsunterschieden zwischen Client und Server, die der Spieler dann auch recht schnell bemerkt. Die kleinste Bewegungseinheit auf dem Spielfeld ist ein nämlich 1 Pixel, wodurch selbst minimale Abweichungen schon einen großen visuelle Unterschied ausmachen und dies kann leider, nach unserer Erkenntnis, auch nicht durch einen intelligenten *Dead Reckoning* Algorithmus abgefangen werden. Für die computergesteuerten Kämpfer wäre es zwar vielleicht noch möglich eine AI-gestützte akkurate Vorausberechnung auf Client-Seite zu realisieren, aber für die von menschlichen Spielern gesteuerten Kämpfer ist dies ausgeschlossen. Deshalb würden wir an dieser Stelle auch empfehlen, eine Synchronisation über den *Brute Force* Ansatz mittles *UDP* zu versuchen und die Datenmenge dann vielleicht über Keyframing (vgl. *2.5.c Brute Force Method*) zumindest ein bischen zu drosseln. Da der Umstieg auf *UDP* aber den Rahmen dieser Arbeit sprengen würde, haben wir beschlossen, dies dem geneigten Leser selbst zu überlassen und für den Moment bei der Theorie zu verbleiben.

Performance

Ein wichtiger Punkt, mit dem wir uns auch auseinandersetzen mußten, war die Gesamtperformanz des Programmes. Um das *Rendern* zu beschleunigen, haben wir *VolatileImages* als *Backbuffer* eingesetzt, was besonders bei älteren Rechnern zu einem deutlichen Geschwindigkeitsunterschied geführt hat. Aber selbst auf einem Athlon 1800XP mit 1GB RAM und einer GeForce 4 Grafikkarte ist die Framerate bei gleichzeitigem Betrieb von Server und Client von ursprünglich 40 fps auf insgesamt 60-70 fps gestiegen. Um den unnötigen Einsatz des *Garbage Collectors* und damit verbundene kurze Unterbrechungen in der Animation zu vermeiden, haben wir natürlich sämtliche Objekte, soweit dies sinnvoll und möglich war, schon im Voraus instantiiert und anschließend wiederverwendet. Für die Netzwerk-Messages haben wir deshalb auch einen speziellen Messagepool erzeugt, der die Verwaltung und Wiederverwendung der Message-Objekte übernimmt. Da wir aber vermeiden wollten, im Messagepool verschiedene Unterklassen zu instantiieren und anschließend für jede Unterklasse eine eigene Queue zu führen, haben wir sämtliche Messages, mit Ausnahme von ein paar weniger häufig verwendeten Messages wie z.B. Text-Message oder Setup-Messages, als Objekte der Oberklasse instantiiert und nur die entsprechenden Sende- und Empfangsmethoden überschrieben. Wir mußten hier einen etwas unkonventionellen Weg über statische Methoden gehen, haben aber, bis auf die etwas erhöhte Komplexität des Modells, keine

negativen Auswirkungen festgestellt. Um die Netzwerkperformance insgesamt zu beschleunigen haben wir auch von Anfang an Java NIO eingesetzt, daß im Vergleich zur alten Java IO deutlich schneller und auch weit weniger komplex und fehleranfällig ist.

Timing

Hier haben wir zunächst den einfachen Weg gewählt und das Timing anfangs über die Thread-Engine von Java realisiert. D.h. wir haben dafür gesorgt, daß die einzelnen Threads nur n-mal pro Sekunde ausgeführt wurden und so schließlich auch eine extrem flüssige und konstante Animation erreicht. Da dieser Ansatz aber nur funktionniert, wenn alle beteiligten Rechner eine annähernd gute Performance aufweisen, sind wir später auf das Timing über den Java-internen Timer umgestiegen. Dies hat zu einer weniger ruckelfreien Animation geführt, aber dafür garantiert, daß das Spiel auch noch auf leistungsschwachen Rechner spielbar bleibt. Um später auf einen *high resolution* Timer umsteigen zu können, haben wir eine spezielle Timer-Klasse verwendet, die die Zeitfunktionen kapselt. Für die Zeitsynchronisation zwischen Client und Server verschicken wir eine spezielle Synchronisations-Nachricht zu Beginn eines Spieles, die den Startzeitpunkt der Animation auf dem Server an den Client übermittelt. Alle später vom Client empfangenen Positionsangaben enthalten dann einen Zeitstempel, der ein Delta zu diesem Startwert ist. Bei geringen Lag-Zeiten, ist der zeitliche Abgleich zwischen Client und Server durch diese Methode gut realisierbar.

3.4. Klassen, Funktionen und Architektur

An dieser Stelle wollen wir noch kurz auf die Programm-Architektur von Stickfighter eingehen.

Pakete

default	enthält die Start-Klassen zum Starten von *Demo*, *Client* und *Server*
stickfighter	enthält übergreifende Klassen, wie z.B. *PlayField*, *Jukebox* und *SpriteSet*.
stickfighter.client	enthält alle Client-spezifischen Klassen
stickfighter.client.gui	enthält alle GUI-Komponenten des Clients
stickfighter.controller	enthält die *Controller* Basisklasse sowie für das Timing relevante Klassen
stickfighter.fighter	enthält verschiedene *Fighter*-Klassen, die die Spielfiguren auf dem Spielfeld repräsentieren.
stickfighter.net	enthält Mehrspieler-spezifische, übergreifende Klassen
stickfighter.net.message	enthält für das Message-System relevante Klassen, wie z.B. die *Message* Basisklasse und den *MessagePool*.
stickfighter.server	enthält alle Server-spezifischen Klassen
stickfighter.test	enthält verschiedene Testklassen

Die wichtigsten Klassen

Jukebox	Wiedergabe und Verwaltung von Sound und Musik
PlayField	Grafische Darstellung des Spielfeldes
SpriteSet	Verwaltung aller Bilder und Grafiken
Gamecontroller	Verwaltet und kontrolliert den Zustand eines Spieles, regelt Animation und Timing und ist insgesamt das Herzstück eines Spieles (Game-Loop).
Game	Verwaltet ein Spiel und führt Listen über die beteiligten *Player* sowie eine *MessageQueue* für die an alle Spieler zu sendenden Nachrichten.
Player	Repräsentiert einen Spieler und kann einem *Fighter* zugeordnet werden.
Fighter	Abstrakte Basisklasse für Kämpfer. Enthält alle Attribute für den Zustand eines Kämpfers, sowie Bewegungscode und Zeichenroutinen und ist zuständig für die grafische Animation der Sprites.
AIFighter	Erweiterte Kämpfer-Klasse, die einen computergesteuerten Kämpfer repräsentiert und alle hierfür relevanten Bewegungs- und Handlungsmethoden enthält.
PlayerFighter	Erweiterte Kämpfer-Klasse, die einen Kämpfer repräsentiert, der von einem menschlichen Spieler gesteuert wird. Enthält alle hierfür relevanten Bewegungs- und Handlungsmethoden.
NetControlledFighter	Dummy-Klasse für den Client. Repräsentiert einen Kämpfer, der durch den Server über des Netz gesteuert wird.
Message	Nachrichten-Basisklasse. Ist zuständig für die korrekte Zuordnung der Sende- und Empfangsmethoden zu den einzelnen Unterklassen. Verwendet intern einen *MessagePool* um *Message*-Objekte zu verwalten und wiederzuverwenden.
MessageQueue	Eine verlinkte Liste für *Message*-Objekte
MessagePool	Instantiiert und verwaltet *Message*-Objekte und ermöglicht deren Wiederverwendung.
MessageDispatcher	Stellt eine *MessageQueue* für das gebündelte Versenden / Empfangen von Nachrichten zur Verfügung und bietet zahlreiche Hilfsmethoden, die das Erstellen und korrekte Befüllen der *Message*-Objekte kapseln.
Server	Bündelt alle für den Server relevanten Klassen und ist für das Annehmen von Client-Verbindungen, sowie für das Senden und Empfangen von Nachrichten zuständig. Beinhaltet den *Main*-Thread des Servers.
ServerGameController	Herzstück eines Spieles auf dem Server

ClientConnection	Stellt eine Verbindung zu einem Client dar und verwaltet den Zustand dieser Verbindung. Verarbeitet außerdem von diesem Client erhaltene Nachrichten und leitet sie an entsprechende Funktionen im *ServerGameController* weiter. Enthält zusätzlich eine *MessageQueue*, die alle Nachrichten speichert, die ausschließlich an diesen Client gerichtet sind.
GameManager	Verwaltet alle auf dem Server geöffneten Spiele
Client	Gegenstück zur *Server*-Klasse. Verwaltet alle für den Client relevanten Klassen und regelt das Senden und Empfangen von Nachrichten. Beinhaltet den *Main*-Thread des Clients.
ClientGameController	Herzstück eines Spieles auf dem Client
ServerConnection	Gegenstück zur *ClientConnection*-Klasse. Stellt eine Verbindung zu einem Server dar und verwaltet den Zustand dieser Verbindung. Verarbeitet alle empfangenen Nachrichten und leitet sie an entsprechende Funktionen im *ClientGameController* weiter. Enthält außerdem eine *MessageQueue*, die alle Nachrichten speichert, die an den Server gesendet werden sollen.
ClientWindow	Hauptfenster des Clients. Bündelt und verwaltet alle GUI-Komponenten. Zeigt und versteckt auf Anforderung durch den *ClientGameController* aktuell benötigte Fenster und Dialoge.
DemoGameController	Herzstück für den Demo-Modus das Spiels. Arbeitet ohne Netzwerkcode und verwendet Basiskomponenten wie *PlayField*, *SpriteSet*, *Jukebox* und *Fighter*.

3.5. Fazit

Die Komplexität eines netzwerkfähigen Spieles, sowie die Herausforderung der grundsätzlichen Spieleprogrammierung unter Java erschienen uns als geeignete Rahmenbedingungen für eine Projektarbeit. Neben Design und Implementierung mußten wir uns auch mit den Themen Debugging, Logging und Profiling auseinandersetzen, sowie geeignete Build & Deploy Mechanismen erarbeiten. Zusätzlich konnten wir auch noch in Randbereich der Informatik hineinschnuppern und uns mit einigen Grundzügen der Bild- und Soundbearbeitung auseinandersetzen.

Was haben wir durch die Projektarbeit gelernt? - Ein Spiel, auch wenn es klein und einfach ist, kann recht schnell zu einer komplexen Anwendung heranwachsen. Aktuell hat Stickfighter über 70 Klassen und mehr als 4000 Programmcodezeilen. Mit einem durchschnittlichen Wert von 2.15 für die zyklomatische Komplexität nach McCabe, liegen wir mit unserem Projekt weit unter der akzeptierbaren Komplexitätsgrenze von 10. Ab dieser Grenze nimmt nämlich, laut zahlreichen Studien, die Fehleranfälligkeit für Programme stark zu. Die Fehlersuche war trotzdem, aufgrund der verteilten Anwendung, teilweise sehr zeitaufwendig und konnte auch nur über eine Internetverbindung in Teamarbeit realisiert werden, da Tests im LAN nicht ausreichend und aussagekräftig genug waren. Wir möchten uns an dieser Stelle auch noch einmal bei den freiwilligen Helfern bedanken, die an unseren Belastungstests teilgenommen und uns mit zahlreichen Anregungen unterstützt haben!

Was haben wir erreicht? - Wir haben eine Version von Stickfighter geschaffen, die im lokalen Netzwerk spielbar ist und über das Internet, bei guter Verbindungsqualität, eine einigermaßen annehmbare Performance aufweist. Wir sind außerdem zu der Erkenntnis gekommen, daß es nur sehr schwer wenn nicht sogar unmöglich ist, eine flüssige Synchronisation und damit "ruckelfreie" Animation bei einem derartig schnellen und aktionsreiches Echtzeitspiel zu gewährleisten. Das Problem ist dabei nicht nur die Anzahl der Clients, sondern vielmehr auch die Tatsache, daß man jede noch so kleine Positionsabweichung der Spielfiguren bemerkt, da sich diese alle gleichzeitig auf der Spielfläche tummeln und auch fast ständig in Bewegung sind. Dabei folgen sie aber nicht, wie z.B. ein Fahrzeug oder Flugzeug, einer relativ gut vorhersehbaren Bewegungskurve, sondern können jederzeit abrupt die Bewegungsrichtung wechseln, was eine Vorausberechnung oder Schätzung von Positionen (mittels *Dead-Reckoning*) sehr schwer bis fast unmöglich macht. Wir haben versucht im Entwicklungsforum von *JavaGaming.org* weitere Ideen für mögliche Lösungsansätze zu finden, haben dort aber erfahren, daß selbst ein einfaches "Pong"-Spiel für 2 Spieler über Internet nur schwer realisierbar ist, da die Reaktionszeiten so kurz wie möglich sein müssen und eine akkurate Vorausberechnung hier nicht möglich ist.

Wie wird es mit Stickfighter weitergehen? - Da uns das Projekt sehr viel Spaß gemacht hat, werden wir es vielleicht auch privat noch ein bischen weiterentwickeln. Erweiterungen wie Hindernisse auf dem Spielfeld oder Fernkampf-Einheiten wären denkbar und der Netzwerkcode könnte ebenfalls noch weiter verbessert werden. Dabei wäre z.B. ein Umstieg auf das *UDP* Protokoll für Positionsupdates denkbar, um so evtl. die Spielbarkeit über Internet weiter zu verbessern. Ob, wie und wann sich das allerdings realisieren läßt muß erst noch abgewartet werden.

Wir möchten uns auch nochmal recht herzlich bei Prof. Grötsch bedanken, der unsere Idee für eine Studienarbeit nicht nur mit Wohlwollen aufgenommen hat, sondern uns auch die notwendige Gestaltungsfreiheit gab, dieses Projekt nach unseren Vorstellungen zu realisieren. Ohne seine Unterstützung und die Motivation durch die Studienarbeit wäre Stickfighter wahrscheinlich nie realisiert worden! Und das wäre sehr schade gewesen, denn wir haben nicht nur jede Menge gelernt, sondern auch viel Spaß gehabt und dabei auch noch einen kleinen Kindsheitstraum verwirklicht! :-)

4. Anhang

4.1. Stichwortverzeichnis

Double Buffering	Methode, um ein Flackern beim *Rendern* zu vermeiden. Alle Zeichenoperationen werden dabei zuerst in einem nicht-sichtbaren Speicherbereich durchgeführt. Anschließend wird der nicht-sichtbare Bereich auf den Bildschirm gezeichnet.
Frame	ein gerendertes Bild der Spielfläche
Framerate	Anzahl der gerenderten Bilder pro Sekunde
Game-Loop	Schleife, die alle spielrelevanten Aktionen durchführt. Dazu gehören: Verarbeiten von Benutzereingaben, Bewegen der Charaktere, Verändern der Spielwelt, *Rendern* der Spielfläche. Der *Game-Loop* wird als eigenständiger Thread realisiert und kann als "Herzschlag" des Spieles gesehen werden. Er ist ein erweiterter *Rendering-Loop*.
Lag	Verzögerungszeit, die beim Transport von Netzwerkpaketen durch Datenstaus oder Paketverluste auftritt. Übliche Latenzzeiten liegen zwischen 50-250 ms.
Rendern	Zeichnen / Neuzeichnen einer Komponente
Rendering-Loop	Schleife, die das *Rendern* der Komponente(n) durchführt. Wird in einem eigenständigen Thread ausgeführt und verwendet in der Regel *Double Buffering*.
Sprites	Grafiken, die aus Bildern mit statischem Inhalt bestehen. Animierte *Sprites* bestehen aus einer Folge von Bildern mit statischem Inhalt. Werden benutzt um die Charaktere und Objekte eines Spieles darzustellen.

4.2. Literaturverzeichnis

- Diskussions Foren von JavaGaming.org (sehr gute Informationsquelle!!!) - http://www.javagaming.org oder http://community.java.net/games/
- Sound Tutorial von Sun - http://java.sun.com/docs/books/tutorial/sound/
- JavaSound API Programmer's Guide - http://java.sun.com/j2se/1.4.2/docs/guide/sound/programmer_guide/
- Java NIO - http://javanio.info/
- Professional Java Game Development - http://java.sun.com/developer/technicalArticles/games/gdc2004.html
- Introduction to NIO Networking - http://www.grexengine.com/sections/people/adam/nio/Introduction_to_NIO_Networking.html
- Dead Reckoning: Latency Hiding for Networked Games - http://www.gamasutra.com/features/19970919/aronson_01.htm
- Designing Fast-Action Games For The Internet - http://www.gamasutra.com/features/19970905/ng_01.htm
- Distributed Gaming - http://www.gamedev.net/reference/articles/article1948.asp

4.3. Quellenverzeichnis

Das Literaturverzeichnis ist gleichzeitig auch Teil des Quellenverzeichnisses. Zusätzlich haben wir noch die im Anhang beigefügten Artikel und PDFs verwendet, sowie natürlich persönliche Erfahrungen, die wir im Laufe der letzten Jahre durch das Lesen von Diskussions Foren wie z.B. *JavaGaming.org* oder verschiedene Publikationen im Internet erworben haben.

4.4. Nützliche Links im Internet

Publisher / Diskussionsforen

- *JavaGaming.org* - http://www.javagaming.org
- *Gamasutra.com* - http://www.gamasutra.com

Bibliotheken

- *Lightweight Java Game Library* - http://www.lwjgl.org/
- *GL4Java* - *http://www.jausoft.com/gl4java.html*
- *JOGL* (Java OpenGL Binding) - https://jogl.dev.java.net/
- *JOAL* (Java OpenAL Binding) - https://joal.dev.java.net/
- *JXInput* - http://www.hardcode.de/jxinput/
- *Java3D* - http://java.sun.com/products/java-media/3D/ und http://www.java3d.org/
- *Log4J* - http://logging.apache.org/log4j/docs/

Profiler

- *Eclipse Profiler Plugin* - http://eclipsecolorer.sourceforge.net/index_profiler.html
- *JMechanic* - http://jmechanic.sourceforge.net/
- *Extensible Java Profiler* - http://ejp.sourceforge.net/

Sonstige Tools

- *Ant* - http://ant.apache.org/

4.5. Einführung in Java Swing

(Swing Einführung aus der Vorlesung Java für Fortgeschrittene bietet sich hier an)

4.6. Einführung in Java 2D

(Die Datei “grafik.und.animation.pdf” bietet eine gute Einführung in Java2D)

4.7. Einführung in Java 3D

(Java3D Einführung aus der Vorlesung Java für Fortgeschrittene bietet sich hier an)