ZFX-Community-Engine: Design und Implementierungen

Kim Kulling, Bjoern Meier, Enrico Zschemisch

E-mail: kim.kulling@web.de, salacryldeliancourt@web.de, enrico.zschemisch@gmx.de

Keywords

3D, engine, ZFXCE, scripting, OpenGL, D3D, OpenAL,

Abstract

Eine Zusammenfassung über Details der ZFX-Community-Engine, in der Design, Ideen und Aussichten zusammengetragen werden

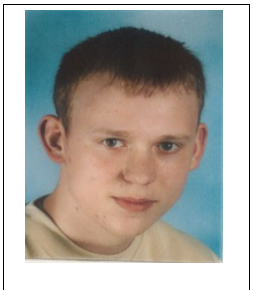
Author Biography

Nach dem Abschluss der höheren Handelsschule und der Ausbildung zum technischen Informatiker, arbeitete Björn Meier 3 Jahre für ein Softwarehaus, dass Software für PPS zu Verfügung stellte. Zur Zeit leitet er die EDV des Caritasverbandes in Hildesheim.

Das Hobby Programmierung bezog sich bisher immer auf Anwendungen. Das Buch “3D Spieleprogrammierung mit DirectX in C/C++” bot den Sprung zur Spieleprogrammierung und zur Mitgliedschaft im ZFXCE-Team.

Nach der Realschule und dem Abitur auf dem Fachgymnasium Eutin, Fachbereich Elektrotechnik, hat Kim Kulling an der Universität Rostock sein Diplom im Fachbereich Bauingenieurwesen, Gebiet Baumechanik abgeschlossen. Bedingt durch das Diplomthema begann das Interesse an Implementierungen physikalischer und grafischer Simulationen mittels verschiedenen Sprachen zu erwachen. Heute arbeitet Kim Kulling bei der Intes-GmbH als Softwareentwickler für FE-Anwendungen und betreibt die 3D-Grafikprogrammierung mit wachsender Leidenschaft als Hobby.

Enrico Zschemisch begann nach seinem Abitur & Zivildienst ein Studium an der Berufsakademie Stuttgart zum Diplom-Ing. für Netzwerk- und Softwaretechnik. Er arbeitet im Rechenzentrum der Universtität Stuttgart (www.rus.uni-stuttgart.de); dort war er zuerst in der Visualisierungsabteilung mit Programmierung des VR Cubes beschäftigt. Momentan entwickelt er Sicherheitslösungen für das lokale CERT (www.cert.uni-stuttgart.de). Zur CE kam er durch sein Interesse an Grafikprogrammierung.



Nach Abschluss der Realschule besucht Florian Bauer jetzt die staatl. Fachhochschule in Altötting und plant in zwei Jahren ein Informatikstudium an einer FH zu beginnen. Das Hobby Programmieren beschäftigt ihn nun seit ca. 5 Jahren. Mit der 3D- und Spieleprogrammieren begann er durch Stefan Zerbst erstes Buch.

Geschichte

Die 3D-Community ZFX (bekannt und entstanden aus den Büchern von Stefan Zerbst) ist Anlaufpunkt für viele 3D-Interessierte aus dem deutschsprachigen Raum. Hier wurde die Idee der ZFX-Community-Engine als Gemeinschaftsprojekt geboren. Ziel war es, die verschiedenen Stärken und Erfahrungen der Community-Mitglieder in einem Projekt zu bündeln und so eine Quersumme der Community in einer eigenen 3D-Engine zu vereinigen. Um allen diese Erfahrungen zukommen zu lassen, wurde sie unter die LGPL-Lizenz gestellt und wird bei Sourceforge gehostet [4].
Nach einigen Startschwierigkeiten wuchs das Team zusehends, momentan hat die ZFXCE folgende aktive Mitglieder:

– Kimmi (Kim Kulling)	Teamleiter, Core, Scenegraph
– Enrico (Enrico Zschemisch)	OpenGL-Rendersystem, Shadersupport
– Jack (Marcel Barkholz)	D3D9-Rendersystem, Shadersupport unter D3D
– Ankon (Andreas Kohn)	Portierung, Core, Automake-Framework
– Salacryl (Björn Meier)	Scripting, Core, GUI
– Lordhoto	Netzwerkcode
– jubu (Jürgen Buntrock)	Netzwerkcode
– Florianx (Florian Bauer)	Sound, Threading Support im Core, Input
– eXile (Paul Müller)	Input, Core

Mehr Informationen zur Geschichte findet man unter [1].

Grundlegendes Design

1) Die ZFXCE arbeitet durch den Einsatz von portablen und betriebssystem-spezifischen Libs sowie der Benutzung von Standard-C++ auf folgenden Plattformen (getestet):

   - WindowsXP, Window2000, Windows98
   - Linux (32 Bit)
   - BSD

2) Zur Zeit kommen die folgenden Fremd-Libs zum Einsatz:

• SDL Version 1.2	Portables Framework für Grafik, GUI, Multithread-Unterstützung und Plugins
• LUA Version 5.0.2	Scripting (Quality Assurance und User Interface)
• OpenAl	Sound
• OpenGL 1.1 mit Extensions	3D Unterstützung
• DirectX 9.0c	3D- und DirectSound-Unterstützung
• dumb	OS-Soundlib
• oggvorbis-lib	OS-Soundlib
• DevIL	Image-Library, wird im OGL-Renderer benutzt
• CPPUnit	Quality Assurance per Unittest
• zlib	Unterstützung von komprimierten I/O

Um eine möglichst modulare Aufteilung mit möglichst hohen Refaktorisierungsfaktoren anzubieten, wurde die ZFXCE in verschiedene problemspezifische Komponenten aufgeteilt. Komponenten beschreiben die grundlegenden Problemstellungen wie Rendering, Sound, Netzwerk, Core oder VFS. Die zugehörigen Implementierungen werden Module genannt, wobei die Komponente nur die Zugehörigkeit beschreibt, das Modul definiert durch seine Implementierung die Funktionalität. Dabei kann Komponente und Modul ein und dasselbe sein, wenn es nur eine geplante Implementierung gibt (z.B. SceneGraph).
Folgende Komponenten und Module sind bisher in der ZFXCE implementiert worden:

Module

1. **Core:**
    - VFS : Grundlegender I/O-Support, Archivmanagement, Serialisierung von Engine-Objekten
    - Logging : Reporting Funktionalitäten zum Protokollieren mit Domains für verschiedender Streams
    - Performance-Messungen über internen Profiler
    - Debug: Callstack, Exceptionhandling, Ausnahmetests über eigene Assert-Definition
    - Fundamentale Multi-Thread-Unterstützung
2. **Scripting:**
    - Core-Funktionen sowie Interface-Definitionen
    - Lua-Scripting-Support
3. **Renderer:**
    - Core-Funktionen sowie Interface-Definitionen
    - OpenGL-Renderer
    - Direct 3D9.0 Renderer
4. **SceneGraph:**
    - Tree-Abbildung der Scene, spezialisierte Implementierungen und Optimierungen per Visitor-Pattern
5. **Input:**
    - Core : Funktionalitäten und Interface Definition
    - SDL-Input : SDL- sowie DirectSoundimplementierung
6. **Sound:**
    - Core: Basis-Funktionalitäten sowie Interfacedefinition und -export
    - OpenAl: Implementierung der Interface-Definition mit OpenAL
7. **Math:**
    - Grundlegende lineare Algebra wie Vektoren und Matrizen
    - Bounding Volumes (AABB- OBB Boxen)
    - Geometrische Objekte wie Frustum, Ebenen, Strahlen und Polygone

Komponente Core

Der Core bietet die grundlegenden Funktionalitäten und Datenstrukturen der Engine an. Andere Komponenten können auf diese dann zugreifen.
Das Debug-Modul bietet einen Tracestack an, der dem Anwender im Falle eines kritischen Fehlers den momentanen Callstack der Applikation bis zum Fehler anbietet, ohne dass ein Debugger benutzt werden muss. Kritische Fehler können mittels einer eigenen Assert-Definition überprüft werden, diese gibt bei einem fehlgeschlagenen Test Callstack, Filename und Zeilennummer im aktuellen Source-File an. Diese ist durch das Werfen einer Exception implementiert worden.

Das Resourcesystem verwaltet Massendaten wie zum Beispiel ein geladenes Modell. Ein Resourcemanager verwaltet offene Ressourcen, diese werden bei Bedarf geladen und bei nicht ausreichenden Speicher wieder in ein File gemapped. Ressourcen werden mittels einer Typ-ID identifiziert. So ist das komplette Serialisieren verwalteter Szenen möglich.

Das VFS (Virtual File System) bietet Funktionalitäten für das I/O-Handling an. Um eine möglich gute Performance anbieten zu können, werden je nach Betriebssystem verschiedene Implementierungen in Form vom File-Managern angeboten (momentan gibt es nur einen Standardfilemanager, geplant sind allerdings POSIX-FileManager, Win32-FileManager sowie eventuell ein MAC-spezifischer FileManager). Daten können durch ein sogenanntes Scratchsystem zwischengespeichert beziehungsweise gecached werden, das Scratchsystem kann dazu Datenstrukturen geladener Ressourcen performant laden und entladen. Der Archivmanager im VFS unterstützt das Laden von komprimierten Dateien.
Ferner kann das VFS Daten in einem virtuellen Verzeichnisbaum ablegen und verwalten, damit zum Beispiel die geladenen Daten je nach Art schnell wiedergefunden werden können:

<DATA_Root>/data/mesh/model1
<DATA_Root>/data/mesh/model2
<DATA_Root>/data/texture/texture1
<DATA_Root>/data/texture/texture2
<DATA_Root>/data/texture/texture3
<DATA_Root>/data/texture/texture4

DATA_Root ist ein definiertes Arbeitsverzeichnis der Applikation.
Das Scratchfile setzt sich aus einzelnen Blocks zusammen, einzelne gemappte Files werden durch verkettete Blocks auf die Festplatte geschrieben. Ressourcen, die gerade nicht benötigt beziehungsweise zu viel Platz im Speicher verbrauchen, können so in ein eigenes Scratchfile endladen und wieder zurück geladen werden.
Der Timer misst mittels SDL-spezifischer Funktionen die jeweiligen Differenz seit der letzten Zeitmessung und kann somit als Taktgeber für die jeweilige Applikation benutzt werden.
Der Profiler kann Perfomance-Messungen in einzelnen Codebereichen durchführen, in dem er die verbrauchte Zeit in den Codebereichen misst. Dazu unterstützt die ZFXCE bereits grundlegende Verfahren zum Multithreading, im Moment wird an diesem Teil der Komponente jedoch noch geplant.
Das Protokollieren (Logging) wird durch ein Domainsystem abgesteuert, jeder Domain wird dabei ein Outputstream zugewiesen. Auf diese Weise können verschiedene Loglevel eingerichtet werden, um nach gesetzten Modus Informationen in gewünschter Granularität zu protokollieren zu können.

Komponente Math

Grundlegende mathematische Werkzeuge werden in allen möglichen Komponenten benötigt. Daher wurden die Mathetools komplett in eine eigene Komponente ausgelagert. Momentan unterstützt die Mathelib in ihrer Implementierung von 2D- und 3D-Vektoren, 4x4-Matrizen, Quaternions, Bounding Volumes (nach Achsen oder nach Objekt ausgerichtet), Planes , Rays, Polygons und ein Frustum mit spezialisierten Implementierungen für D3D und OpenGL.
Spezielle Optimierungen für 3DNow- und SSE-Prozessor-Architekturen sind sporadisch vorhanden, weitere Optimierungen sollen in der Zukunft folgen.

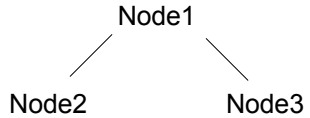
Komponente Testing

Um voll funktionierede Methoden auch bei grösseren Umbauten an bestehenden Interfaces garantieren zu können, benutzt die ZFXCE Unittests, um mittels Standardtests grobe Fehler schnell finden zu können. Gerade mit wachsenen Abhängigkeiten in der Engine können sich schnell Folgefehler einschleichen.
Diese Standardtests sind mittels der Unittest-Lib implementiert worden [3]. Bei jeder neu hinzugefügten Implementierung kann der Entwickler hier die Funktionen mittels Standardtests abfragen. Fälle, die Probleme beziehungsweise Fehler werfen müssen, kann man so ebenfalls verifizieren. Neue Implementierungen und Umbauten müssen nun diese Ergebnisse wiederum erzielen. Ist das nicht der Fall, hilft einem ein generiertes Protokoll bei der Fehlersuche weiter.

Komponente Scenegraph

Einführung

Der Scenegraph stellt die geometrischen Abhängiggkeiten der darzustellenden Scene dar. Dazu bildet der Scenegraph die physikalischen, geometrischen oder renderspezifischen Abhängigkeiten durch einen gerichteten Graphen ab (zur Zeit die Spezialform eines gerichteten Graphen):



Will man nun einen Node , werden auch alle Childnode-States auf den neuesten Stand gebracht. Auf diese Weise kann der Scenegraph Informationen der dargestellten Scene verwalten. Ob zum Beispiel ein Teil des Modelles im Viewport liegt, ob ein Strahl (beispielsweise ein Geschoss) irgendein verwaltetes Objekt im Scenegraph trifft, ist durch die abgebildete Geometrie einfach zu ermitteln. Dazu wird ein Boundingvolume-Tree verwaltet, Parentnodes beinhalten dabei die eigenen Abmasse sowie die der zugehörigen Childnodes. Wird nun beim Cullingtest ein Parent ausserhalb des Viewfrustum lokalisiert, kann die komplette Geo-Hierarchie durch den rekursiven Bounding-Tree geculled werden. Dazu kann man z.B. die Schnittpunkte eines Rays durch die Scene schnell auf die zugehörigen Scenenodes schliessen.

Einsatz des Visitor-Pattern:
Datenmodell und spezielle Implementierungen sollen voneinander durch das Design getrennt werden. Des weiteren sollen Erweiterungen ohne grossen Aufwand in das Scenegraph-Design eingebunden werden können. Daher werden spezialisierte Implementationen per Visitor Pattern angeboten. Je nach Bedarf können benötigte Implementierungen alloziert und in den Scenegraphen eingehängt werden. Momentan werden die folgenden Visitoren unterstützt:

- RenderVisitor :
  - Implementiert das Interface vom Scenegraph zum Renderer
  - Scene-Hierarchie ist von der Renderimplementierung getrennt
- CullVisitor:
  - Implementiert das Frustumculling, soll später durch alternative Cullingstrategien austauschbar sein
  - Benutzt AABB Boxen zum Cullen

Nodes im Scenegraphen
Szenen-Objekte werden durch Knoten (Nodes) verwaltet. Diese können einfach nur andere Nodes zusammenfassen, geladene Netze (Meshes genannt) repräsentieren oder eine Camera mit deren Position und Ausrichtung beschreiben. In der ZFXCE sind momentan folgende Knoten implementiert:

- Node-Interface:
  - Beinhaltet Position und Pointer auf Childnodes
  - Implementiert das Basisinterface für alle Scenegraph-spezifischen Nodes wie die Verwaltung der Nodehierarchie, rekursive Such-Algorithmen und weitere Traversierungsstrategien zum Durchlaufen der Hierarchien
- Node:
  - Aktualisiert die komplette Childnode-Hierarchie
  - Gruppiert Nodes zu Node-Sets, ohne eigene Geometrien zu verwalten, nur die Childnodes werden in einem Einhüllungskörper (Bounding-Volume) zusammenfasst
- SceneNode:
  - Transformiert die Geometrie und die der Childs
  -Bildet die Geometrie-Repräsentanz der statischen Szene ab
  - Verwaltet Position des Knotens
  - Beinhaltet Referenz auf Bounding-Volume Tree für das Culling
- Camera:
  - Verwaltet Position und Sichtvektor auf die dargestellte Szene
  - Verwaltet Sicht-Kegel (Frustum) zum Cullen der Szene

Aussichten
Die ZFXCE soll in Zukunft Blending im Scenegraph unterstützen, durch diesen soll dabei die Vorsortierung der zu rendernden Objekte implementiert werden (Renderdevices beherrschen diese bereits). Da die Renderer ihre jeweilige Renderqueue nicht optimieren können, soll der aktive Rendervisitor diese Aufgabe übernehmen (je nach Bedarf nach Textur-Status, benutzten Shader oder nach einem anderen Gesichtspunkt).
Ein Landschaftsknoten soll mittels verschiedener Methoden wie Quadtree etc. die Darstellung von komplexeren Landschaften ermöglichen, hierbei sollen LOD- und Softwareocclusionculling-Strategien erprobt werden.

Komponente Scripting

Scripting stellt ein Interface bereit, mit dem ein Code, der häufigen Änderungen ausgesetzt ist, nicht stets übersetzt werden muss. So werden z.B. die K.I. oder nichtinteraktive Sequenzen in Textdateien ausgelagert. Diese Textdateien werden dann von einem Interpreter ausgelesen und verarbeitet (z.B in einem Datei namens test.lua).

Das Scripting für Sequenzen arbeitet eng mit dem Scenegraph zusammen, der zu bestimmten Ereignissen (Events) die Schnittstelle über den Visitor aufruft und das passende Script interpretiert. So können z.b. Fallen, Treffpunkte, Tastaturbefehle und ähnliches mit dem Scripting abgesteuert werden, ohne das gesamte Programm zu rekompilieren.

Natürlich stellt das Scripting auch die Möglichkeit zur Verfügung, ganze Klassen (z.b. PlayerClass) eines Spiels zu modifizieren. So kann man direkt in das Spiel eingreifen, ohne es neu übersetzen zu müssen.
Zusammen mit ceConsole bietet ceScript in Zukunft Möglichkeiten Funktionen direkt (sozusagen “on-the-fly”) aufzurufen.
Klassen, die exportiert werden sollen, können per Proxy-Template exportiert werden, so dass man auf die Methoden in Lua zugreifen kann.

Ferner wird ein Tool geplant, das ZFXCE-Module laden kann und diese ohne einen erneuten Kompiliervorgang ausführen kann.

Komponente Rendering

Einführung

Der Renderer ist nur zum sturen Rendern von geometrischen Objekten da. Es gibt nur sehr wenige Optimierungen, um alles andere muss sich die übergeordnete Ebene/Layer (z.B. unser SceneGraph) kümmern.
Als erstes muss man seine Geometrie- und Indexdaten in einem ceRenderObject ablegen. Diese RenderObjects werden in einem ceRenderJob kombiniert. Ein solcher RenderJob speichert jedoch nicht nur den Vertex- und IndexBuffer. In einem RenderJob kann man alle möglichen Einstellungen der Render-Pipeline ändern, als da wären bsw. Blending, Viewport oder Texture Combining. Auf diesem Weg ist es möglich, Geometriedaten mehrfach zu verwenden und wertvollen Grafikkartenspeicher zu sparen. Sind alle zu rendernden RenderJobs bestimmt, gibt man diese an die ceRenderQueue. Die RenderQueue geht einfach die Liste der Jobs durch und rendert diese in ein Render-Target. Zur Auswahl stehen hier verschiedene Texturformate und natürlich normale Fenster. Auf diese Weise lassen sich viele verschiedene Effekte umsetzen, z.B. Shadow Mapping, Full Scene Glow oder High Dynamic Range Rendering. In der RenderQueue gibt es eine zweite Optimierung: einen StateCache. Dieser stellt sicher, dass nicht zuviele States gesetzt werden, ohne dass sie geändert wurden.
D3D9 bietet von Haus aus sehr viel mehr Optionen, was die Render-Pipeline angeht, als OpenGL. In OpenGL sind diese zwar auch implementiert, allerdings meist nur über Extensions. Da unsere beiden Render-Implementierungen momentan mittels OpenGL und D3D9 umgesetzt sind, benötigen beide natürlich das gleiche Feature-Set. Momentan werden dazu umfangreiche Tests gefahren. Sobald diese Tests abgeschlossen sind, werden weitere Features eines modernen Renderers implementiert. Einige Ideen sollen hier beispielhaft vorgestellt werden.

Shader-Unterstützung

Shader-Unterstützung ist heutzutage ein essentielles Feature. Um die Shader-Einheiten der Grafikkarte anzusteuern, gibt es viele verschiedene Programmiersprachen, z.B. mehrere Assembler-Dialekte, GLSL oder HLSL. Diese alle zu unterstützen, einzubinden und zu testen verursacht einen sehr großen Aufwand. Deswegen haben wir uns von der CE entschieden, Shader-Support nur mittels D3D Effect Files anbieten. Einige werden sich jetzt wundern, warum nur diese Effect Files mit HLSL? Andere Engines wie Ogre oder Irrlicht bieten ja auch alle Sprache für Shader an. Ein Argument ist der eben genannte, immense Aufwand. Weiterhin erlauben Effect Files einen sog. „Ressource Driven Renderer“, d.h. man braucht zum Ändern oder Neuerstellen von Shader-Effekten keinen Programmierer. Dieser kann sich auf andere Sachen konzentrieren, die Effekteinstellungen können vom Designer selbständig und einfach geändert werden. Ein dritter Punkt ist die große Verfügbarkeit von Entwicklungstools für D3D Effect Files; einige Beispiele sind der nVidia FX Composer oder ATIs ShaderMonkey.

Für D3D9 ist die Umsetzung denkbar einfach, da die Effect Files ja Teil der API sind. Im Gegensatz dafür ist die Implementierung in OpenGL um einiges komplexer. Wir benötigen auf jeden Fall einen Parser, der die Dateien ausliest und die Einstellungen in ein für OpenGL verständliches Format umwandelt. Nach dem Einlesen müssen noch die Shader-Funktionen in ein für die Grafikkarte brauchbares Format kompiliert werden. Hierfür benötigt man einen Compiler. Da Microsoft und nVidia HLSL zusammen entwickelt haben und nVidia daraus eine eigene Sprache für OpenGL abgeleitet hat, kann man den Cg Compiler dafür verwenden [2]. In den nVidia-Treibern ist dieser per Extension schon eingebaut. Und es gibt eine OpenSource-Variante, diese hat allerdings nur ein generisches Profil, erzeugt also keinen Assembler-Output, wie der richtige Cg-Compiler. Wir brauchen also ein eigenes Profil für den Compiler.

Als Alternative bietet sich folgender Ansatz an: Man benötigt wiederum den Parser. Dieser erzeugt dann die Funktionen aus dem Effect File. Diese Funktionen übergibt man an einen „Übersetzer“. Dieser wandelt den HLSL in nativen GLSL-Code um. Wichtiger Vorteil ist das Fehlen des fehleranfälligen und langwierigen Entwickeln eines eigenen Compilers. Die Zeitdauer, die die Umsetzung und dann anschließende Kompilierung dauert, dürfte sich im Bereich von wenigen ms bewegen. Ist also zu vernachlässigen. Einziger Nachteil ist, dass eventuell ältere Grafikkarten auf diesem Weg nicht unterstützt werden können. Inwieweit man hier Standardeffekte implementieren sollte, ist noch zu prüfen. Letztendlich bevorzugen wir natürlich die Variante mit dem „Übersetzer“, da diese schnell und einfach umzusetzen ist.

Hardware Occlusion Culling

Die Vertex- und Objektleistungen von Grafikkarten steigen immer weiter an. Allerdings steigt die Geschwindigkeit des Depth Buffers und Depth Tests nicht annähernd so schnell. Zusätzlich verursachen lange Shader-Programme einen hohen Rechenaufwand pro Fragment. Um die Last für die Grafikkarte zu senken, ist effizientes Occlusion Culling sehr wichtig.

Bis vor einiger Zeit wurde dies mit einem separaten Software Depth Buffer umgesetzt. D3D9 und OpenGL bieten jedoch seit geraumer Zeit an, den Depth Buffer der Grafikkarte für Occlusion Culling zu benutzen. Dabei rendert man zuerst ein paar große Objekte in den Depth Buffer. Danach deaktiviert man Depth-Buffer-Schreiben und rendert die Occluder. Die zugrundeliegende API gibt dann ein Sichtbarkeitsergebnis zurück.

Die Umsetzung in der CE wird einfach sein: ceRenderJob bekommt ein neues Flag, dass auf Occlusion mittels Depth Buffer testen soll, normales Rendern wird dabei deaktiviert. Die RenderQueue speichert dann in dem RenderJob, ob und wieviel des Objektes sichtbar war. Sollte Occlusion Culling nicht ünterstützt werden, wird sicherheitshalber immer „sichtbar“ zurückgegeben.

Komponente Input

Der Input besteht im wesentlichen aus einen Input-Handler, der eingehende Events verarbeitet. Dazu kann man auch eigene Events generieren und entsprechend versenden. Diese können bisher manuell in einem eigenen Messageloop verarbeitet werden oder direkt per Listener-Interface an eine eigene Implementierung gebunden werden. Für die Zukunft soll so die Möglichkeit geschaffen werden, Controller für eigene Game-Entities zu definieren, um so speziell generierte Events eigens zu empfangen und entsprechend verarbeiten zu können. Auf diese Wiese soll die interne Kommunikation spezialisiert auf das jeweilige Objekt implementiert werden können.

Um das ganze etwas performanter zu machen, werden verarbeitete Events wieder recycelt und dann später erneut verwendet.

Mittleweile unterstützt der Inputhandler Unicodes, die für eine Chatdemo speziell von J. Buntrock entwickelt wurde, da er den Unterstützung von chinesischen Schriftzeichen plant.

Aufgrund der gewünschten Portabilität existiert momentan eine auf die SDL spezialisierte Implementierung.

Komponente Netzwerk

Die ZFXCE unterstützt momentan eine auf TCP/IP basierende Socked-Implementierung.

Komponente Sound

Die ZFXCE beinhaltet zwei Sound-Implementierungen, eine mittels OpenAL und eine mit DirectSound werden von Soundquellen ausgesendet, sowohl 2D- als auch 3D-Soundquellen können angegeben werden. Ferner kann man bei 3D-Sound
noch Dopplereffekte per Geschwindigkeitszuweisungen (im ZFXCE: VelocitySetter) simulieren. Weitere Effekte sind geplant.
Sounds werden durch ein extra Modul ceSoundFile geladen, welches aber für den User unsichtbar ist. Unterstützt werden zurzeit folgende Formate: .ogg, .wav, .mod, .it, .s3m und .xm. Diese können entweder komplett in den RAM geladen oder beim abspielen nachgeladen werden. Zu diesem Zweck wurde im ZFXCE-Core die ersten Implementierungen für Multithreading implementiert, so dass das Nachladen in einem separaten Thread stattfinden kann.
In Zukunft ist das Laden und Streamen von Sounds mit dem VFS und dem Ressource Manager geplant, dafür müssen diese jedoch noch entsprechend erweitert werden.

Speziellen Dank an:

Exile (Korrekturheld), arBmind, windowsint, dv, das gesamte ZFX-Team und die Community, Seraph sowie allen Bugsubmittern und Testern und all die, die ich vergessen habe...

Referenzen:

[1][http://zfxce.zfx.info] Anlaufpunkt der ZFXCE im Web, Links zu allen Third-Party-Libs und Anleitungen zum Programmieren

[2][http://developer.nvidia.com/object/cg_toolkit.html] nVidia Toolkit

[3][http://sourceforge.net/projects/cppunit] Cpp-Implementierung von Unittests

[4][http://sourceforge.net/projects/zfxce/] Sourceforge-Projektseite