Das Framework

Das Framework

Was verstehen wir unter einem Framework?

- Orxonox ist in erster Linie ein Spiel, aber um dieses Spiel zu erstellen, benötigt man eine Umgebung. Die Umgebung stellt alle benötigten Features zur Verfügung. Diese Umgebung nennen wir Framework, weil sie den Rahmen von Orxonox bildet.

Das Framework

Die Grundlagen des Frameworks:

- Code und Content werden in zwei getrennten SVN Repositories verwaltet.
- Um Orxonox zu kompilieren, verwenden wir CMake als Buildsystem. CMake erstellt Makefiles für unser Projekt, die anschliessend vom bekannten make Tool verwendet werden.
- Unser Code besteht aus verschiedenen Libraries. Externe Libraries werden grösstenteils als Dependencies vorausgesetzt (d.h. sie müssen auf dem System installiert oder in einem bestimmten Ordner vorhanden sein). Einzelne kleinere und/oder wenig verbreitete Libraries werden mit Orxonox mitgeliefert und kompiliert.

Installation

Installation

Orxonox installieren:

- Wenn man Orxonox nur spielen will, reicht es, ein Paket herunterzuladen, das die Binaries enthält.
- Will man Orxonox entwickeln (Code), muss man Orxonox aber selber kompilieren können. Dazu müssen, abhängig vom System, einige Voraussetzungen erfüllt werden.
- Installationsanleitungen für die wichtigsten Systeme befinden sich auf der Wiki. Ergänzungen und Korrekturen sind willkommen, damit die Anleitungen aktuell bleiben.

Installation

Installationsanleitungen auf der Wiki:

Login | Settings | Help/Guide | About Trac

Home | About | Screenshots | Download | Development | Forum

Download Orxonox

Source access

The best and most convenient way to access our source code is to use SVN and check out the trunk as it contains the most updated and recent stable features:

https://svn.orxonox.net/orxonox/trunk

You also need the media files. The path can be configured with CMake at [Wiki:Buildsystem compile time] or later by editing orxonox.ini (look for "mediaPath"). The default search locations for CMake is one directory above slash media and in the checkout directory slash media.

https://svn.orxonox.net/data/media

Anonymous Access

If you only want to download the source, you can use the link above but replace https with http.

TracNav menu

- Development
  - Overview
  - SVN
  - Download
    - Overview
    - Gentoo
    - Debian
    - Ubuntu
    - ETH Tardis
    - Problems under Linux
    - Windows with MinGW
    - Windows with Visual Studio
    - Fedora
    - Mac OSX
  - Contribute
  - Programming:
    - Modules

Installation

Tardis:

- Auf Tardis sind bereits alle Libraries vorhanden, die Installation reduziert sich daher auf Checkout, CMake, Build.

```
> mkdir orxonox
> cd orxonox
> svn co https://svn.orxonox.net/orxonox/trunk
> svn co https://svn.orxonox.net/data/media
> cd trunk
> mkdir build
> cd build
> cmake-2.6.2 ..
> make
> ./run
```

Installation

HTTPS:

- Wichtig: Unbedingt https verwenden, wenn man später etwas commiten (d.h. verändern) will. Mit http hat man nur read Rechte. Verwendet man https, muss man beim ersten Checkout seinen Login angeben. Dabei handelt es sich um den selben Account wie für die Wiki. Name und Passwort sollten bekannt sein.

Installation

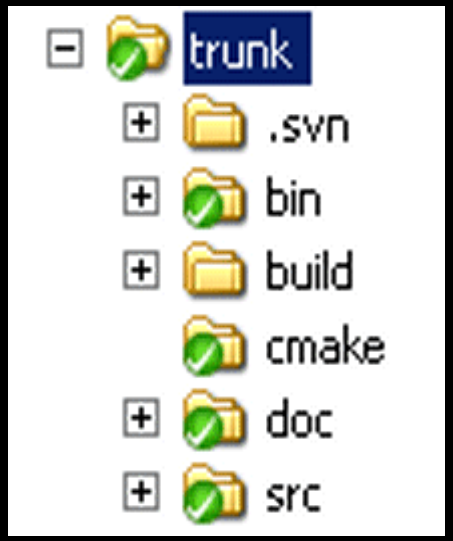
Installation auf anderen Systemen:

- Auf Linux Systemen sollte es kein grosses Problem sein, die benötigten Libraries zu installieren oder notfalls auch selber zu kompilieren. Einige Distributionen sind in der Wiki dokumentiert.
- Für Windows sind Pakete mit allen vorkompilierten Libraries vorhanden. Sie müssen in einem Ordner „dependencies“ abgelegt werden, der im selben Verzeichnis liegt, wie der trunk und das media Verzeichnis.
- Für Mac gibt es keinen offiziellen Support, weil bisher kein Core Entwickler auf diesem System gearbeitet hat. Dennoch sollte es im Prinzip möglich sein. Eventuell müssen einige Anpassungen im Code und im Bildsystem vorgenommen werden.

Die Verzeichnisstruktur

Die Verzeichnisstruktur

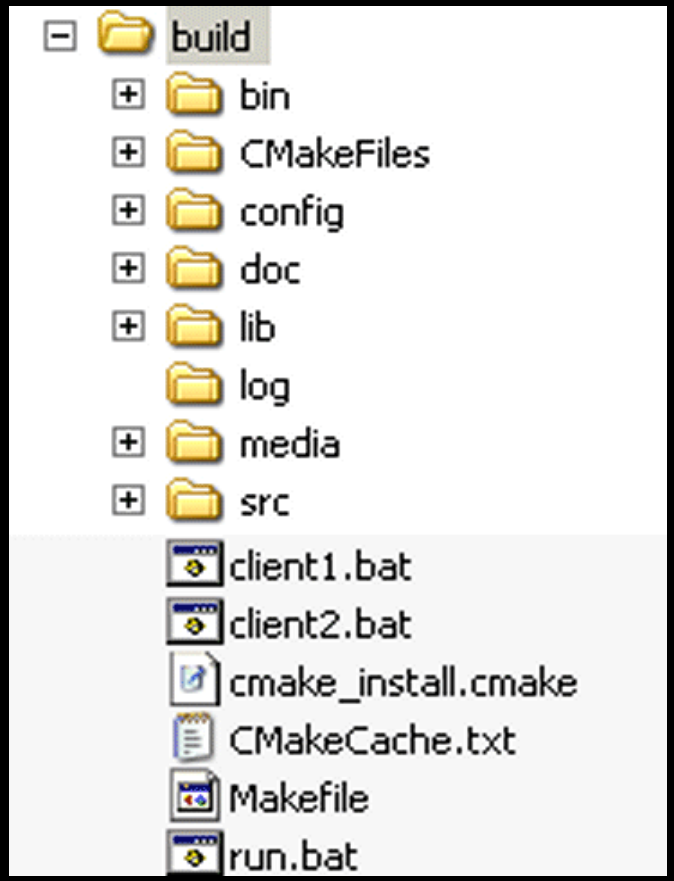
Das Hauptverzeichnis:



- `.svn:` Hilfsverzeichnis für SVN
- `build:` Wurde von euch angelegt, enthält alles wasvon CMake und make erstellt wird
- `cmake:` Enthält CMake Scripts
- `src:` Enthält den Code

Die Verzeichnisstruktur

Das Buildverzeichnis:



- `bin:` Executables und Libraries
- `config:` Config-Files
- `log:` Output
- `run:` Startet Orxonox (runscript)

Die Verzeichnisstruktur

Die Config-Files:

- `orxonox.ini`: Konfiguriert Orxonox
- `keybindings.ini`: Definiert Keybindings (Commands die bei einem Tastendruck ausgeführt werden)
- `masterKeybindings.ini`: Keybindings im Startmenü
- `ogre.cfg`: Konfiguriert Ogre (Auflösung, Vollbild oder Fenster, Antialiasing, ...)

Die Verzeichnisstruktur

Die Logfiles:

- `orxonox.log`: Enthält den gesamten Output von Orxonox. Der Umfang (bzw. der Detailgrad) des Outputs ist im Configfile (orxonox.ini) konfigurierbar.
- `ogre.log`: Output nur von Ogre (z.B. interne Probleme beim Laden einer Textur)

Libraries

Libraries

Die wichtigsten internen Libraries von Orxonox:

- Orxonox: Der grösste und wichtigste Teil. Enthält alle Klassen, die für das Gameplay relevant sind.
- Core: Der Kern von Orxonox. Enthält wichtige interne Klassen, die für das Management von Objekten und diverse weitere Features des Frameworks verantwortlich sind.
- Util: Diverse kleine Hilfsklassen und -Funktionen, die das Programmieren einfacher gestalten.
- Network: Wird verwendet, um Orxonox im Multiplayer zu spielen. Definiert unter anderem ein Interface und Hilfsfunktionen für synchronisierbare Klassen.

Libraries

Die wichtigsten externen Libraries:

- Ogre: Die Grafikengine
- CEGui: Menüs
- Bullet: Physikengine
- ENet: Netzwerkbasics
- Lua und Tcl: Skriptsprachen
- Boost: Eine Sammlung von nützlichen Klassen

Ogre

Ogre

Warum wir Ogre als Grafikengine verwenden:

- Ogre läuft auf allen wichtigen Plattformen (Linux, Windows, Mac und mehr)
- Ogre unterstützt alle gängigen Features von aktuellen Grafikengines (Shaders, dynamisches Licht und Schatten, Partikeleffekte und vieles mehr)
- Der Code ist open source (und LGPL lizenziert)
- Ogre wird von einer grossen und aktiven Community entwickelt und supportet
- Der Code ist gut strukturiert und dokumentiert

Ogre

Ogre

Ogre

Ogre

Ogre

Gehe zu

Core

Core

Wichtige Features des Core:

- Identifier: Identifiziert die Klasse eines Objekts, speichert Informationen zu Klassen und den zugehörigen Instanzen.
- Objectlists: Speichern alle Objekte einer bestimmten Klasse.
- Iterator: Iteriert durch eine Objectlist und gibt die gespeicherten Objekte zurück.
- Super: Ermöglicht den Aufruf einer Funktion in der Basisklasse.

Core

Identifier:

- Jedes Objekt hat eine Klasse und jede Klasse hat einen Identifier:

```
MyClass* a = new MyClass();
Identifier* id = a->getIdentifier();
cout << id->getName() << endl;

// returns: „MyClass“
```

- Der Identifier einer Klasse ist auch ohne Objekt verfügbar:

```
Identifier* id = Class(MyClass);

// Class ist ein Makro - nicht zu verwechseln
   mit class als Keyword.
```

Core

Identifier:

- Abfrage ob ein Objekt einer bestimmten Klasse angehört:

```
MyClass* a     = new MyClass();
OtherClass* b  = new OtherClass();
Identifier* id = Class(MyClass);

if (a->getIdentifier() == id)
   cout << „a ist MyClass“ << endl;
if (b->getIdentifier() == id)
   cout << „b ist MyClass“ << endl;

// returns: „a ist MyClass“
```

Core

Objectlist und Iterator:

- Jedes Objekt wird in der Liste seiner Klassen (also auch der Parent-Klassen) gespeichert::

```
MyClass* a  = new MyClass();
MyClass* b  = new MyClass();

ObjectList<MyClass>::iterator it;
int counter = 0;

for (it = ObjectList<Myclass>::begin();
     it != ObjectList<Myclass>::end(); ++it)
   counter++;

cout << „Anzahl Objekte: “ << counter << endl;

// returns: „Anzahl Objekte: 2“
```

Core

Super:

- Ruft die Funktion der Basisklasse auf:

```
void ParentClass::test()
{
    cout << „parent“ << endl;
}

void MyClass::test()
{
    SUPER(MyClass, test);

    cout << „myclass“ << endl;
}

// returns: „parent“
//          „myclass“
```

Core

Super:

- Anstelle von SUPER könnte man auch explizit die Basisklasse angeben. Allerdings führt dies zu Fehlern, wenn sich die Klassenhierarchie im Verlauf der Entwicklung ändert. Mit SUPER muss man jedoch die eigene Klasse angeben, wodurch der Aufruf unabhängig von der Hierarchie wird.

```
void MyClass::test()
{
    SUPER(MyClass, test); // Diese zwei Zeilen sind
    ParentClass::test();  // äquivalent.

    cout << „myclass“ << endl;
}
```

- Eine Funktion muss allerdings erst als SUPER-Funktion deklariert werden.

Util

Util

Die wichtigsten Features der Util Library:

- Debug Output, der im Logfile erscheint:

```
COUT(0) << „Hello World!“ << endl;
```

- Die Zahl in Klammern steht für den Level des Outputs:

  0: Erscheint immer

  1: Für Fehlermeldungen

  2: Für Warnungen

  3: Für Informationen über den Programmverlauf

  4-6: Für Debuginformationen

Util

Die wichtigsten Features der Util Library:

- Math.h: Ein Headerfile, das einige häufig verwendete Funktionen mit mathematischem Inhalt deklariert.
- String.h: Ein Headerfile, das diverse Funktionen zur Manipulation von Zeichenketten deklariert.
- Convert.h: Ein Headerfile, das mehrere Funktionen zum Konvertieren von Datentypen definiert (z.B. einen int in einen string konvertieren - oder aber auch genau andersrum).

Util

Die wichtigsten Features der Util Library:

- MultiType: Eine Klasse, die einen Wert von (fast) jeden beliebigen Typ annehmen kann. Gleichzeitig ermöglicht sie auch die Rückgabe desselben Wertes, konvertiert in einen anderen Datentyp.

Util

Die wichtigsten Features der Util Library:

```
MultiType a = 10;    // Ein int wird zugewiesen
float b = 11.111;
a = b;               // Ein float wird zugewiesen. Allerdings
                        hat der MultiType bereits den Typ int
                        angenommen. Es gilt nun also (a == 11)
                        und NICHT (a == 11.111)

a.convert<float>(); // Der MultiType wird in einen float
                        konvertiert.
a = b;               // Nun gilt (a == 11.111)

string c = a;        // Der MultiType wird einem string
                        zugewiesen. Es gilt nun (c = „11.111“)

testfunction(a);     // Eine beliebige Funktion wird mit dem
                        MultiType aufgerufen. Der Wert wird
                        automatisch richtig konvertiert.
```

Input und Output

Input und Output

Output von Text:

- Wie bereits erwähnt, kann Text mit dem COUT(level) Makro ausgegeben werden. Dieser Text erscheint an drei Orten: Im Logfile, in der Konsole und in der ingame Shell.
- Der User kann über das Config-File entscheiden, wie viel Output er in jedem der drei Outputkanäle sehen will (indem er den maximalen Output-Level definiert). Zur Erinnerung: Je höher der Level, desto unwichtiger der Output.
- Stürzt Orxonox ab, gibt manchmal ein Blick in das Logfile wichtige Hinweise. Unter Umständen muss aber erst der Outputlevel erhöht und der Absturz reproduziert werden, damit die notwendigen Informationen erscheinen.

Input und Output

Output von Text:

- Gut platzierter Debug-Output im eigenen Code kann bei der Entwicklung sehr nützlich sein. Unwichtiger Output sollte nach getaner Arbeit allerdings wieder entfernt werden, um die Performance zu verbessern.

Input und Output

Input von Commands:

- Orxonox besitzt eine eigene ingame Shell. Sie kann üblicherweise über die Taste unterhalb ESC geöffnet werden. Ist dies nicht möglich, muss die Tastenbelegung in `masterKeybindings.ini` geändert werden.
- In der ingame Shell erscheint einerseits der Output von Orxonox, andererseits können Commands über die Eingabezeile eingegeben werden.
- Commands werden mit Hilfe eines Makros im Code definiert. Ein Command ist im Prinzip lediglich eine Verknüpfung zwischen einem String (der Name des Commands) und einer Funktion. Die Shell führt zu jedem eingegebenen String die zugehörige Funktion auf, sofern diese als Command deklariert wurde.

Input und Output

Die Shell:

```
KeyBinder: Loading key bindings...
Loaded config file "keybindings.ini".
KeyBinder: Loading key bindings done.
client connected
Loading level...
Start loading C:\Orxonox\Programme\msys\home\xen\orxonox\media\\Lev
 els\presentation.oxw...
Mask: +BaseObject
player entered level (id: 0, name: Player)
Player entered the game
Finished loading C:\Orxonox\Programme\msys\home\xen\orxonox\media\\
 levels\presentation.oxw.
|
```

Input und Output

Die Shell:

- Ein einfacher Command ist „log text“. Er gibt „text“ in allen drei Outputkanälen aus. Dies ist äquivalent zu `COUT(0) << „text“ << endl;` im Code.

Input und Output

Die Shell:

```
Loading level...
Start loading C:\Orxonox\Programme\msys\home\xen\orxonox\media\\lev
 els\presentation.oxw...
Mask: +BaseObject
player entered level (id: 0, name: Player)
Player entered the game
Finished loading C:\Orxonox\Programme\msys\home\xen\orxonox\media\\
 levels\presentation.oxw.
log text
text
log Hello World!
Hello World!
|
```

Input und Output

Die Shell:

- Um komplexe Ausdrücke zu verarbeiten, verwendet die Shell eine Scriptsprache.
- Es handelt sich dabei um Tcl (sprich „Tickle“).
- Tcl ermöglicht unter anderem die Auswertung von verschachtelten Commands über [eckige Klammern].

Input und Output

Tcl:

```
Finished loading C:\Orxonox\Programme\msys\home\xen\orxonox\media\\
 levels\presentation.oxw.
log text
text
log Hello World!
Hello World!
log [expr 2+3]
5
log 2 plus 3 gibt [expr 2+3]
2 plus 3 gibt 5
log [expr 5-3] plus [expr 5-2] gibt [expr 2+3] :)
2 plus 3 gibt 5 :)
```

Input und Output

Tcl:

- Tcl bringt ausserdem eine grosse Anzahl eigener Befehle mit sich. So ist es zum Beispiel möglich, in einer Zeile einen Telnet Server zu erstellen. Mit ein paar weiteren Zeilen entsteht daraus ein Remote Controll System für Orxonox.

Input und Output

Die Syntax von Tcl:

- Die Syntax von Tcl ist auf den ersten Blick nicht wesentlich anders als in den meisten anderen Sprachen.
- Im Detail gibt es jedoch einen wichtigen Unterschied: Tcl ist darauf ausgelegt, in einer Shell verwendet zu werden. Deswegen ist alles auf einer Zeile möglich.
- Ein Zeilenumbruch bedeutet das Ende eines Commands.
- Ein Whitespace bedeutet das Ende eines Arguments.
- Zeilen und Argumente können mit Hilfe von {geschweiften Klammern} gebündelt werden. Die öffnende Klammer muss aber vor dem ersten Zeilenumbruch stehen.
- Argumente können auch mit „Anführungszeichen“ zusammengefasst werden.

Input und Output

Die Syntax von Tcl:

- Tcl folgt einem sehr einfachen aber ungewöhnlichen Prinzip: Es gibt keine Keywords, nur Funktionen.
- Es gibt also z.B. keinen if Befehl, sondern eine Funktion. Sie nimmt mindestens zwei Argumente (1. die Bedingung und 2. den Code). Optional kann das dritte Argument else oder auch elseif sein.
- if {a == b} {log Hello World}
- Wie man sieht, wurden {a == b} und {log Hello World} mit den Klammern zu jeweils einem Argument zusammengefasst.
- Es entsteht eine vertraute Syntax, die jedoch einen anderen Ursprung hat.

Input und Output

Die Syntax von Tcl:

```
log [expr 2+3]
5
log 2 plus 3 gibt [expr 2+3]
2 plus 3 gibt 5
log [expr 5-3] plus [expr 5-2] gibt [expr 2+3] :)
2 plus 3 gibt 5 :)
set a 2
set b 3
if {$a+$b == 5} {log $a+$b gibt 5} else {log $a+$b gibt nicht 5}
2+3 gibt 5
if {$a+$b == 7} {log $a+$b gibt 7} else {log $a+$b gibt nicht 7}
2+3 gibt nicht 7
|
```

Input und Output

Die Syntax von Tcl:

- Es können übrigens alle Funktionen umdefiniert werden. Das erlaubt uns, die volle Kontrolle über die ausgeführten Scripts zu behalten. Denkbar ist z.B. das unterbinden von Endlosschleifen, die Orxonox freezen würden.

Input und Output

Die Syntax von Tcl:

```
Start loading C:\Orxonox\Programme\msys\home\xen\orxonox\media\\lev
 els\presentation.oxw...
Mask: +BaseObject
player entered level (id: 0, name: Player)
Player entered the game
Finished loading C:\Orxonox\Programme\msys\home\xen\orxonox\media\\
 levels\presentation.oxw.
log text
text
proc log args {puts HAHA}
log text
HAHA
|
```

Input und Output

Das Config-File:

- Im Config-File werden Werte von Variabeln definiert, die allen Objekten der entsprechenden Klasse zugewiesen werden.
- Beispiel:

```
[MyClass]
myValue = 10

---------------------------

MyClass* a = new MyClass();
cout << a->myValue << endl;

// returns „10“
```

Input und Output

Das Config-File:

- Jede Variable im Config-File entspricht einer Variable im Code.
- Wird ein Objekt einer solchen Klasse erstellt, wird ihm der Wert aus dem Config-File zugewiesen.
- Ändert der Wert im Config-File, werden alle bestehenden Objekte aktualisiert.

Input und Output

Das Config-File:

orxonox.ini - Editor

Datei Bearbeiten Format Ansicht ?

```
[Core]
softDebugLevelConsole_=3
softDebugLevelLogfile_=4
softDebugLevelShell_=3
language_=default
bInitializeRandomNumberGenerator_=true
mediaPathString_=C:/Orxonox/Programme/msys/home/xen/orxonox/media/

[GSRoot]
statisticsRefreshCycle_=250000
statisticsAvgLength_=1000000

[GSGraphics]
resourceFile_=resources.cfg
ogreConfigFile_=ogre.cfg
ogrePluginsFolder_=.
ogrePlugins_=Plugin_ParticleFX, Plugin_CgProgramManager, RenderSystem_GL, RenderSystem_Direct3
ogreLogFile_=ogre.log
ogreLogLevelTrivial_=5
ogreLogLevelNormal_=4
ogreLogLevelCritical_=2

[Projectile]
damage_=15
lifetime_=4

[Bot]
names_[0]=Dr. Julius No
names_[1]=Rosa Klebb
names_[2]=Auric Goldfinger
names_[3]=Emilio Largo
```

Input und Output

Das Config-File:

- Das Ändern von Config-Values ist auch über die Konsole möglich. Der Command heisst `config` und verfügt über ein Autocompletion System, so dass man einfach die richtige Variable findet.
- Ein Config-Value, der mit `config` geändert wurde, wird mit dem neuen Wert im Config-File gespeichert.
- Will man einen Wert nur temporär ändern, ohne das Config-File zu modifizieren, kann der Command `tconfig` verwendet werden.

Input und Output

Das Config-File:

```
Player entered the game
Finished loading C:\Orxonox\Programme\msys\home\xen\orxonox\media\\
 levels\presentation.oxw.
Bot Camera ChatOverlay ControllableEntity Core Engine GSGraphics GS
 Level GSRoot Gametype GraphicsEngine HumanPlayer InGameConsole Inp
 utBuffer InputManager KeyBinder Projectile Shell SpaceShip Spectat
 or Stats Test
Gametype
bAutoStart_ bForceSpawn_ initialStartCountdown_ numberOfBots_ score
 boardTemplate_
numberOfBots_
Old value: 0
config Gametype numberOfBots_ 5
```

Input und Output

Keybindings:

- Um Orxonox zu steuern, existieren Keybindings.
- Ein Keybinding bindet einen Command auf einen Key.
- Bei diesen Commands handelt es sich um dieselben Commands, die auch in der Shell verwendet werden können.
- Es ist also möglich, jedes Tcl Script auf eine Taste zu binden. Gleichzeitig ist es auch theoretisch möglich, Orxonox per Script zu steuern.
- Um die Defaultwerte für Keybindings zu definieren, existiert eine Vorlage für `keybindings.ini` im Media Repository. Die Vorlage wird allerdings nur dann übernommen, wenn `keybindings.ini` noch nicht existiert, also direkt nach dem Checkout oder wenn man die Datei löscht.

Input und Output

Keybindings:

keybindings.ini - Editor

Datei Bearbeiten Format Ansicht ?

```
[Keys]
KeyA="scale -1 moveRightLeft"
KeyABNT_C1=
KeyABNT_C2=
KeyAT=
KeyAX=
KeyApostrophe=
KeyApps=
KeyB=
KeyBack=
KeyBackslash=
KeyC=switchCamera
KeyCalculator=
KeyCapsLock=
KeyColon=
KeyComma=
KeyConvert=
KeyD="scale 1 moveRightLeft"
KeyDelete=
KeyDivide=
KeyDown=
KeyE="scale -1 rotateRoll"
KeyEnd=
KeyEquals=
KeyEscape="exit"
KeyF="scale -1 moveUpDown"
KeyF1=
KeyF10=
KeyF11=
KeyF12=
KeyF13=
```

Input und Output

Keybindings:

- Globale Keybindings, die schon im Startmenü vorhanden sein müssen, können über `masterKeybindings.ini` definiert werden. Diese Keybindings werden nicht zum Steuern verwendet, sondern zum Beispiel um die Konsole zu öffnen.

Die Objekthierarchie

Die Objekthierarchie

Definition eines Objekts:

- Ein Objekt in Orxonox ist eine Klasse, die das Spielgeschehen beeinflusst oder von ihm beeinflusst wird, direkt oder indirekt.
- Nicht zu Verwechseln mit Objekt im Sinne einer Instanz einer Klasse.
- Beispiele für Objekte:

  Ein Spaceship: Ganz klar ein Objekt
  Eine Künstliche Intelligenz: Ist zwar nicht physikalisch, beeinflusst aber trotzdem das Spielgeschehen, ist also ein Objekt.
  Ein HUD: Wird vom Spielgeschehen beeinflusst, ist also ebenfalls ein Objekt.
  Ein Callbackmanager: Kein Objekt
  Ein Timer: Kein Objekt

Die Objekthierarchie

Eigenschaften eines Objekts:

- Alle Objekte erben von einer Basisklasse, genannt BaseObject.
- Alle Objekte, bis auf ein paar wenige Ausnahmen, befinden sich in der Orxonox Library (ein paar sind im Core).
- Alle Objekte können per XML geladen werden, sofern sie nicht abstrakt sind oder das Laden ausdrücklich verbieten.
- Mehr noch: Objekte sind die einzigen Klassen, die per XML geladen werden können.
- Alle Objekte, egal ob über ein XML File geladen oder später dynamisch erstellt, werden am Ende eines Levels zerstört.

Die Objekthierarchie

Eigenschaften eines Objekts:

- Die Eigenschaft, dass ein Objekt nur für die Dauer eines Levels existiert, erklärt auch, warum selbst unphysikalische Dinge wie eine künstliche Intelligenz oder ein HUD Objekte sind. Nämlich ganz einfach, weil sie ansonsten auch nach dem Beenden eines Levels noch existieren würden, was zu erheblichen Problemen führen würde.
- Bei allen Nicht-Objekten wird implizit davon ausgegangen, dass sie entweder während der ganzen Programmlaufzeit existieren, oder aber selbständig für ihre Zerstörung sorgen.

Die Objekthierarchie

Interfaces:

- Die Klassenhierarchie der Objekte muss einen Baum bilden, Mehrfachvererbung ist nicht erlaubt.
- Es ist allerdings erlaubt, Interfaces zu verwenden. Interfaces sind selbst keine Objekte, jene können aber von einem Interface erben.
- Im Gegensatz zu Java definieren wir ein Interface nicht als reine Deklaration, sondern wir verstehen darunter eher so was wie eine abstrakte Klasse, die selbst nicht erzeugt werden kann, jedoch durchaus eigene Variabeln und Funktionen implementiert.
- Der Sinn an Interfaces ist, Gemeinsamkeiten von nicht-verwandten Klassen zu vereinheitlichen. Über eben dieses Interface können jene Klassen gemeinsam angesprochen werden.

Die Objekthierarchie

Tickable:

- Ein Beispiel für ein Interface, ist Tickable. Es deklariert folgende Funktion:

```
void tick(float dt);
```

- Die Funktion `tick` wird in jedem Programmdurchlauf einmal aufgerufen. Das Argument `dt` beinhaltet die Zeit seit dem letzten Aufruf in Sekunden.
- Dies ermöglicht Klassen, die von Tickable erben, in jedem Tick etwas zu tun (zum Beispiel eine Bewegung durchzuführen oder ein Update zu vollziehen).

Die Objekthierarchie

Synchronisable:

- Ein anderes wichtiges Interface ist Synchronisable. Es ist die Grundlage für alle Klassen, die über das Netzwerk synchronisiert werden können.

Die Objekthierarchie

Das BaseObject:

- Das BaseObject ist die Basisklasse aller Objekte in Orxonox.
- Das BaseObject implementiert die gemeinsamen Grundlagen aller Objekte. Dies beinhaltet einen Namen, das Laden über XML (und später auch über Savefiles), das Erzeugen und Empfangen von Events, die Zugehörigkeit zu einem Level und vieles mehr.
- Das BaseObject selbst implementiert noch keine Interfaces.

Die Objekthierarchie

Das WorldEntity:

- Das WorldEntity ist die zentralste Klasse aller Objekte.
- Ein WorldEntity ist ein Objekt mit einer Position im Raum.
- Alle Objekte die sichtbar, beweglich, physikalisch oder auch einfach nur lokalisierbar sind, sind WorldEntities (bzw. abgeleitete Klassen).

Die Objekthierarchie

Das WorldEntity:

- Das WorldEntity selbst ist eine abstrakte Klasse, kann also nicht direkt erzeugt werden. Allerdings gibt es eine Reihe von Unterklassen, die Unterschiedliche Arten von WorldEntities implementieren.
- Diese Unterklassen ermöglichen die Optimierung des WorldEntities auf eine spezielle Aufgabe. Beispielsweise muss ein bewegliches Objekt in jedem Tick aktualisiert werden (Tickable), während dies bei einem statischen Objekt nicht nötig ist.

Die Objekthierarchie

Das WorldEntity:

- Zur Zeit existieren drei Unterklassen von WorldEntity: StaticEntity, MovableEntity und ControllableEntity.
- Ein StaticEntity ist alles, was sich nicht bewegt.
- Ein MovableEntity ist optimiert für gleichförmige Bewegungen, also z.B. ein geradeaus fliegendes Projektil oder ein drehender Asteroid.
- Ein ControllableEntity kann jede beliebige Bewegung durchführen. Dies trifft z.B. auf Spaceship, aber auch auf zielsuchende Raketen zu.

Die Objekthierarchie

Das WorldEntity:

- WorldEntities können eine grafische Representation besitzen, z.B. durch ein Modell, aber auch ein Billboard oder eine andere grafische Klasse ist möglich.
- Mehr noch, ein Objekt kann NUR DANN eine grafische Repräsentation besitzen, wenn es sich dabei um ein WorldEntity handelt.
- Dasselbe gilt für die physikalischen Eigenschaften. Nur WorldEntities sind in der Lage, zu kollidieren oder sonstigen physikalischen Gesetzen unterworfen zu werden.
- WorldEntities werden ausserdem automatisch über das Netzwerk synchronisiert, es sei denn sie unterbinden es explizit.

Die Objekthierarchie

Das WorldEntity:

- WorldEntities können zusammengehängt werden. Dies läuft über SceneNodes, eine Klasse die aus Ogre (der Grafikengine) stammt.
- Zusammengehängte WorldEntities müssen einen Baum bilden. Bewegt sich der Root, bewegen sich die angehängten Objekte automatisch mit. Bewegt sich ein Unterobjekt, so ist seine Bewegung relativ zur Bewegung und Ausrichtung des Parents.
- Dies ermöglicht das Anhängen von Lichtern und Triebwerkeffekten an ein Spaceship, aber auch das Spaceship selbst ist an die Root-Node der Szene gebunden.

Die Objekthierarchie

Die Szene:

- WorldEntities können nur innerhalb einer Szene existieren.
- Die Szene ist ebenfalls ein Konzept aus Ogre.
- Eine Szene kann eine Skybox, Ambient Light und vieles mehr definieren.
- Es können gleichzeitig mehrere Szenes parallel existieren, ohne dass sich die WorldEntities überschneiden.
- Ein einzelndes WorldEntity kann jedoch nur in einer Szene existieren.
- Noch nicht einmal der Wechsel der Szene ist möglich. Dies ist nur über eine Kopie des Objekts möglich.

Die Objekthierarchie

Die Szene:

- Ein Levelfile muss mindestens eine Szene enthalten.
- Bei mehreren Szenes ist z.B. der Wechsel von Aussen- in Innenleves denkbar, ohne dass der Innenlevel den aussenlevee irgendwie beeinflusst.
- Diese Konzept wird auch an die Phyiskengine weitergegeben, d.h. Objekte in unterschiedlichen Szenen können nicht kollidieren. Es sind im Prinzip zwei unterschiedliche Welten.

Die Objekthierarchie

Der Gametype:

- Damit ein Spiel stattfinden kann, muss in jeder Szene mindestens ein Gametype definiert sein, wobei aber durchaus mehrere Szenen zum selben Gametype gehören können.
- Ein Gametype definiert, wie das Spiel abläuft, d.h. was nach dem Laden des Levels geschieht, welches steuerbare Objekt einem Spieler zugewiesen wird, was geschieht, wenn jemand Stirbt und natürlich auch, wann das Spiel wieder beendet wird.

Die Objekthierarchie

Der Gametype:

- Durch die Verknüpfung von Szenes und Gametypes, aber auch Unter-Nodes in einer Szene, sind beliebige Kombinationen möglich.
- Es ist zum Beispiel möglich, ein Minigame in einer anderen Szene stattfinden zu lassen.
- Es ist aber auch möglich, dasselbe Minigame anhand einer bewegten Node durch den Raum fliegen zu lassen.
- Gleichzeitig können auch normale Gametypes wie etwa Capture the Flag als Minigame verwendet werden, oder aber natürlich auch ein Minigame als Hauptgame. Der Unterschied ergibt sich lediglich aus der Reihenfolge.

Die Objekthierarchie

Der Spawnpoint:

- Damit der Spieler überhaupt an einer bestimmten Stelle in einer bestimmten Szene starten kann, wird ein SpawnPoint verwendet. Ein SpawnPoint gehört zu einem Gametype. Existieren mehrere SpawnPoints, wird zufällig einer gewählt.
- Existieren Spawnpoints in mehreren Scenes, verteilen sich die Spieler zufällig auf die beiden Schauplätze.

Die Objekthierarchie

Das ControllableEntity:

- Startet der Spieler an einem SpawnPoint, wird ihm vom Gametype ein Objekt zugewiesen, das er steuern kann.
- Bei diesem Objekt handelt es sich um ein ControllableEntity.
- Ein ControllableEntity kann aber nicht nur vom Spieler gesteuert werden, sondern zum Beispiel auch von einer künstlichen Intelligenz.
- Sowohl der Spieler als auch die künstliche Intelligenz sind Controller.

Die Objekthierarchie

Der Controller:

- Controller und ControllableEntity stehen in einem Master-Slave Verhältnis.
- Der Controller kann das ihm zugewiesene Objekt über ein Steuerungs-Interface kontrollieren.
- Durch die Trennung von Controller und ControllableEntity bleibt das System flexibel. Der selbe Spieler kann auch das Spaceship wechseln, bzw. das selbe Spaceship kann abwechselnd von mehreren Spielern gesteuert werden.
- Der Controller stellt also quasi die logische Instanz eines Spielers dar. So lange der Spieler „lebt“, existiert der Controller. Was den Spieler sterben lässt, definiert der Gametype. Es ist durchaus denkbar, dass der Controller die Zerstörung seines Spaceships überlebt.

Die Objekthierarchie

Das PlayerInfo Objekt:

- Allerdings wäre es sinnlos, alle Informationen über einen Spieler zu verwerfen, sobald er stirbt. Zum Beispiel können Scores nur dann mitgezählt werden, wenn der selbe Spieler über den gesamten Spielverlauf durch das selbe Objekt repräsentiert wird.
- Diese Aufgabe übernimmt das PlayerInfo Objekt. Es enthält unter Anderem auch den Namen des Spielers und Informationen über seine Verbindung zum Server.

Die Objekthierarchie

Die Kontrollstruktur im Überblick:

- Ein Spieler, der auf einen Server joint und darauf spielt, wird durch ein **PlayerInfo** Objekt identifiziert.
- Dieser Spieler übernimmt eine virtuelle Spielfigur, die beliebig oft zerstört und neu zugewiesen werden kann. Es handelt sich dabei um den **Controller**.
- Der Controller kann, so lange er existiert, wiederum beliebig viele Objekte steuern, die selbst auch zerstört werden können, ohne dass dies aber zwangsläufig den Controller beeinflussen muss. Diese Objekte sind **ControllableEntities**.

Die Objekthierarchie

Der Pawn:

- Da es Unterschiedliche Arten an steuerbaren Objekten (ControllableEntities) gibt, also z.B. sowohl ein Spaceship, als auch ein Spectator, existiert eine weitere Basisklasse, der Pawn.
- Pawn bedeutet auf Englisch Bauer und steht für die Basisklasse aller Objekte, die Leben und Sterben können (und auch Töten).
- Er unterscheidet sich vom ControllableEntity dadurch, dass er Lebenspunkte und einen Zustand (lebendig oder tot) hat.
- Ein SpaceShip ist ein Pawn, der Spectator hingegen nicht.
- Trotzdem können natürlich beide Objekte über das gemeinsame Interface von ControllableEntity gesteuert werden.

Die Objekthierarchie

Die grafische Repräsentation:

- Damit man ein Objekt sehen kann, sollte es ein Modell oder zumindest ein Billboard besitzen.
- Ein Modell ist ein dreidimensionales Objekt, das aus einer geschlossenen Oberfläche, zusammengesetzt aus Polygonen (meist Dreiecke), besteht.
- Ein Billboard ist eine zweidimensionale Fläche, die sich immer gegen den Betrachter richtet und eine einzelne Textur enthält.
- Partikeleffekte bestehen aus vielen kleinen Billboards.

Die Objekthierarchie

Die grafische Repräsentation:

- Ein Objekt kann nur dann sichtbar sein, wenn es eine Position im Raum hat. Daraus folgt, dass nur WorldEntities eine grafische Repräsentation haben können.
- Es gibt eine Reihe von Klassen in der Objekthierarchie, die eine bestimmte Form der grafischen Repräsentation implementieren:

  Modell (hat ein 3D Modell, in Ogre auch Mesh genannt)
  Billboard (hat ein BillboardSet)
  Light (eine Lichtquelle, wahlweise punktuell, direktional oder gebündelt)
  ParticleEffect (ein ParticleEmitter der dauerhaft aktiv ist)
  ParticleSpawner (ein ParticleEmitter der zu einem bestimmten Zeitpunkt oder in einem bestimmten Intervall für eine bestimmte Zeit aktiv wird)

- Ausserdem gibt es noch ein paar weitere, speziellere Klassen.

Die Objekthierarchie

Die grafische Repräsentation:

- Eine wichtige Designentscheidung bei Orxonox war, dass die grafische Repräsentation nicht Teil der Klasse ist, die repräsentiert werden soll, sondern dass es sich dabei um zwei getrennte Objekte handelt, die über SceneNodes verknüpft werden.
- Dies ermöglicht es einem Spaceship, wahlweise null, eines, zwei oder mehrere Modelle zu haben, ohne dass sie in der Spaceship-Klasse hardcoded sein müssen.
- Auch das Projektil einer Waffe kann zum Beispiel aus einem Modell bestehen (Torpedo), kann aber auch nur ein Billboard sein (Energiekugel) oder sogar als Strahl (Laser) oder Partikel-Effekt (Rauchspur) dargestellt werden.
- Der Nachteil ist, dass man für jede grafische Komponente ein zusätzliches Objekt benötigt.

XML

XML

Der Aufbau eines Levelfiles:

- Levelfiles werden in Orxonox als XML-Files definiert.
- Eine Klasse in Orxonox kann in XML folgnedermassen erzeugt werden:

```
<Klassenname />
```

- Klassenvariabeln können als Argumente übergeben werden:

```
<Klassenname variable1="value" variable2="value" />
```

- Unterklassen (z.B. ein angehängtes WorldEntity) können durch Verschachtelung definiert werden:

```
<Klassenname variable1="value" variable2="value">
    <Unterklasse variable1="value" />
    <Unterklasse variable1="value" />
</Klassenname>
```

XML

Der Aufbau eines Levelfiles:

- In vielen Fällen ist allerdings nicht klar, was mit einer Unterklasse geschehen soll. Deshalb gibt es sogenannte **Subsections**, die die Unterklassen aufteilen und nach einer vordefinierten Regel verarbeiten:

```
<SpaceShip position="100,0,0">
    <attached>
        <Model mesh="spaceship1.mesh" />
        <Billboard position="0,5,0" colour="1,0,0" />
        <Billboard position="0,-5,0" colour="1,0,0" />
    </attached>
    <weapons>
        <Lasergun munition="100" />
        <Torpedo munition="5" />
    <weapons>
</SpaceShip>
```

XML

Der Aufbau eines Levelfiles:

- Um einen Level korrekt zu definieren, muss zuerst ein Level-Objekt erzeugt werden:

```
<Level
 name          = "Test"
 description   = "A simple testlevel"
>
</Level>
```

XML

Der Aufbau eines Levelfiles:

- Anschliessend muss eine Scene definiert werden:

```
<Level
 name          = "Test"
 description   = "A simple testlevel"
>
    <Scene
     ambientlight = "0.5, 0.5, 0.5"
     skybox       = "Orxonox/Skybox1"
    >
    </Scene>
</Level>
```

XML

Der Aufbau eines Levelfiles:

- Innerhalb dieser Scene können dann Objekt erstellt werden. Unter anderem muss mindestens ein SpawnPoint vorhanden sein, ansonsten kann der Level nicht geladen werden:

```
<Level
 name          = "Test"
 description   = "A simple testlevel"
>
    <Scene
     ambientlight = "0.5, 0.5, 0.5"
     skybox       = "Orxonox/Skybox1"
    >
        <SpawnPoint position="100,0,0" lookat="0,0,0"
spawnclass=SpaceShip pawndesign=spaceshipassff />
    </Scene>
</Level>
```

XML

Die Verarbeitung eines XML-Files:

- Um ein XML-File einlesen zu können, wird in den zu ladenden Objekten eine Funktion namens XMLPort(...) implementiert.
- Für jedes Attribut und jede Subsection wird ein Makro verwendet, das beim Aufruf der Funktion die entsprechende Stelle aus dem XML-File parst.
- Der geparste Wert bzw. die erzeugte Unterklasse wird dann einer Funktion übergeben, die das „Laden“ vollzieht. Auch für das Speichern ist eine solche Funktion vorgesehen, sie gibt den aktuellen Wert des Attributs zurück.
- Der Aufbau und die Funktionsweise der XMLPort-Funktion und der Makros wird in der Dokumentation auf der Wiki ausführlich beschrieben.

XML

Scripting:

- Um die Levels interessanter zu gestalten, kann innerhalb der XML-Files eine Scriptsprache verwendet werden.
- Wir verwenden dazu Lua.
- Der Grund warum wir nicht Tcl verwenden, wie für die Shell, ist einfach: Tcl kann nur Commands aufrufen, aber Lua kann die Attribute der Objekte manipulieren. Hätte Tcl diese Möglichkeit, bzw. würde man Lua in der Shell verwenden, würde dies zu massiven Sicherheitsproblemen führen.
- Wir verwenden also absichtlich zwei unterschiedliche Scriptsprachen, da beide komplett unterschiedliche Verwendungszwecke haben.

XML

Scripting:

- Eine Sektion mit Lua-Code wird in XML wie folgt initiiert:

```
<xml code />
<?lua
    ...lua code...
?>
<xml code />
```

- Es ist aber auch Möglich, durch Lua einen Output mitten in einer XML-Sektion zu generieren:

```
<Klasse attribut="<?lua print(value); ?>" />
```

XML

Scripting:

- Dadurch ist es zum Beispiel auch möglich, ein Asteroidenfeld zufällig zu generieren.
- In Zukunft wird es auch möglich sein, Lua-Code nicht direkt beim Laden des XML-Files zu parsen und den Output auszuwerten, sondern Den Code zu Speichern und während der Laufzeit des Spiels auszuführen. Dadurch ist es möglich, Scriptsequenzen durchzuführen, die z.B. Aktionen auslösen oder Objekte manipulieren.