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:

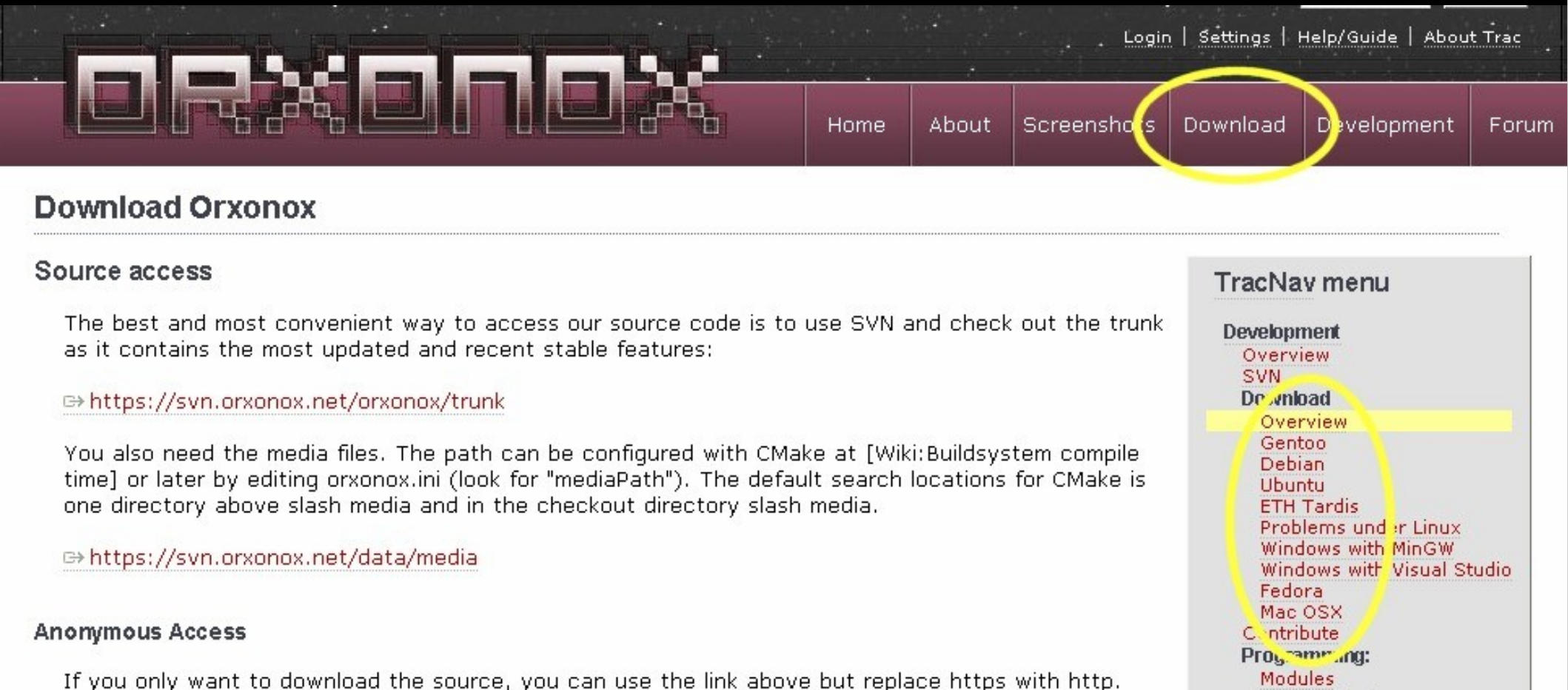
- Wenn man Orxonox nur spielen will, reicht es, ein Paket herunterzuladen, das die Binaries enthält.

- 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:



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.

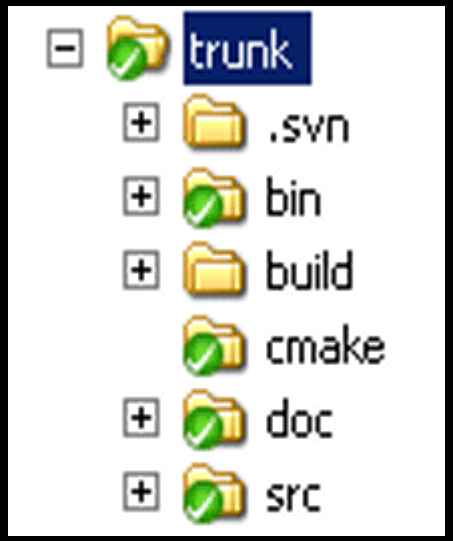
- Für Windows sind Pakete mit allen vorkompilierten Libraries vorhanden. Sie müssen in einem Ordner „dependencies“ abgelegt werden, der im selben Verzeichnis liegt, wie der trunk und das media Verzeichnis.

- 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

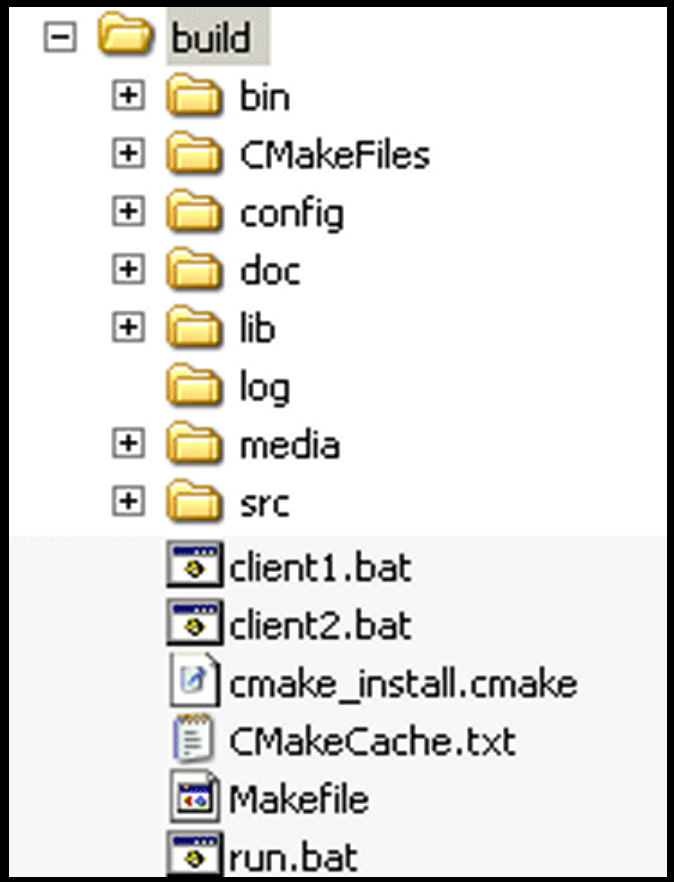
Das Hauptverzeichnis:



- `.svn:` Hilfsverzeichnis für SVN
- `build:` Wurde von euch angelegt, enthält alles was von 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)

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.

```
COUT(1) << „Error: Abcd efg hijklmn opqrs. << std::endl;
```

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:

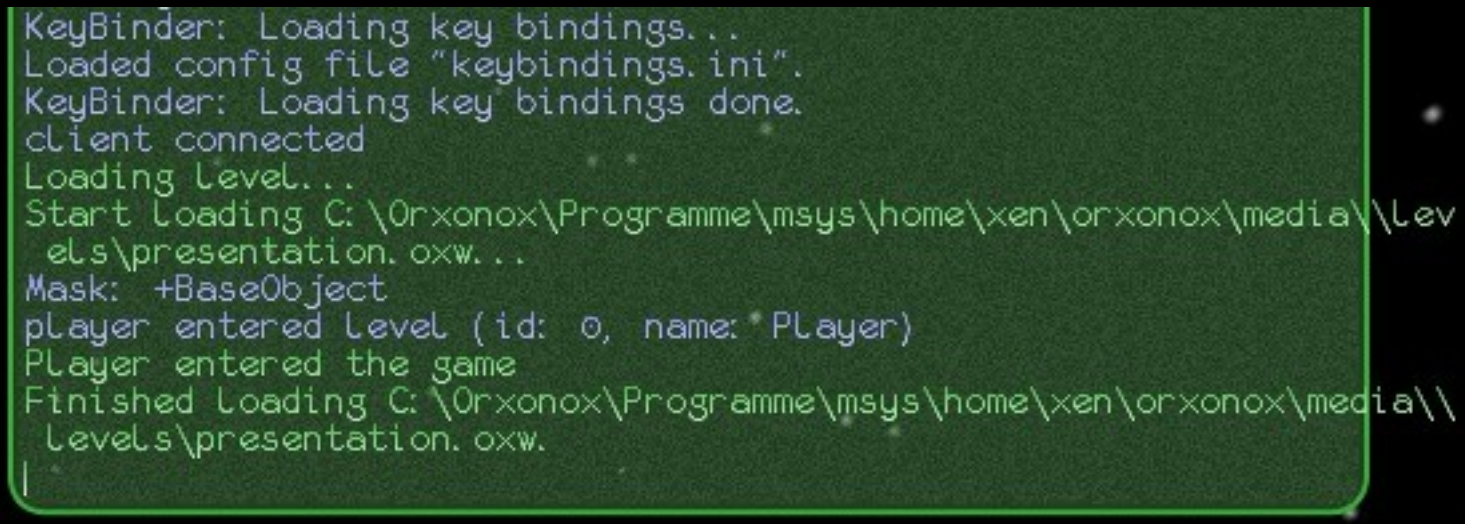
- Orxonox besitzt eine eigene ingame Shell. Sie kann üblicherweise über die Taste unterhalb ESC geöffnet werden. Ist dies nicht möglich, muss die Tastenbelegung in `masterKeybindings.ini` geändert werden.

- 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:



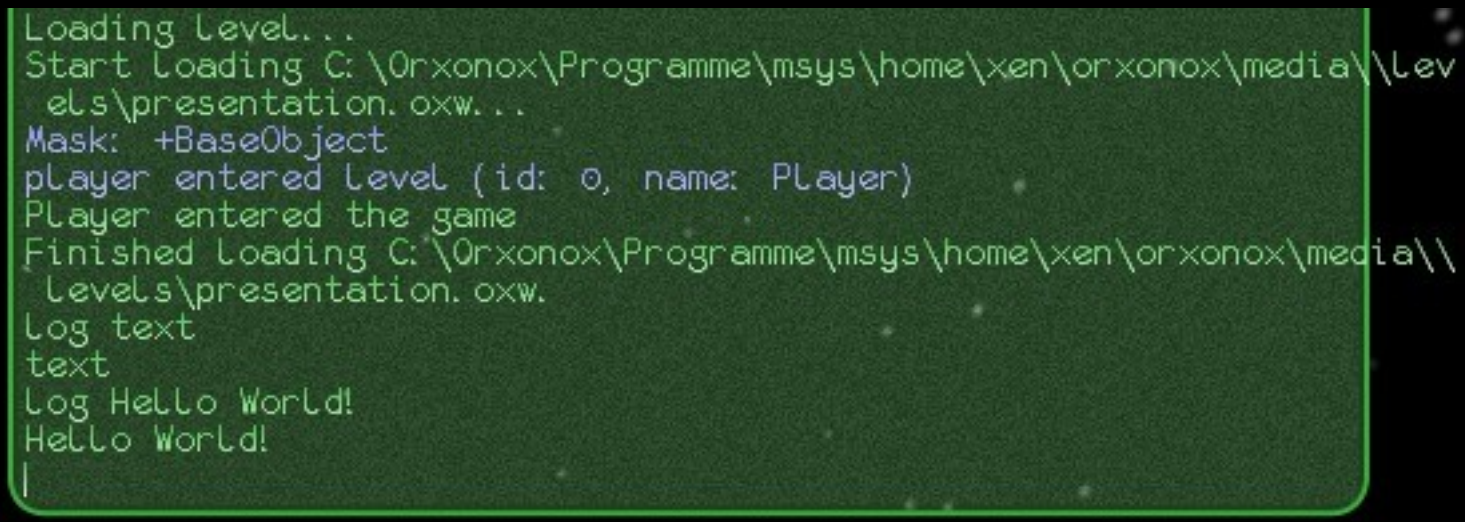
Input und Output

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:



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
```

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.

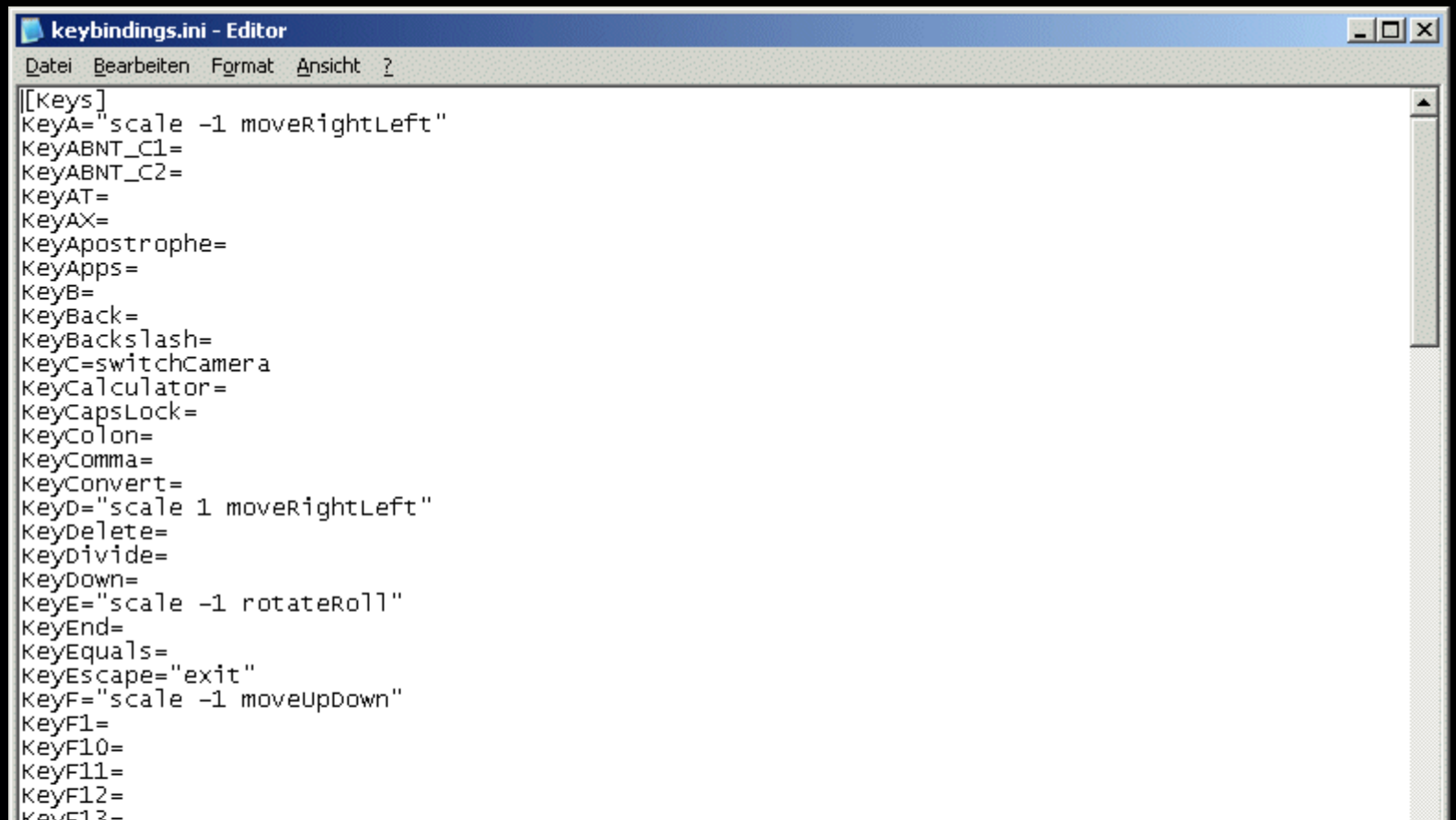
Input und Output

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:



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.

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.
- Factory: Ermöglicht das erzeugen eines Objekts durch die Nennung der Klasse als String.
- 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

Factory:

- Um die Klasse mit einem String verknüpfen zu können, muss man die Factory mittels CreateFactory(Classname) erzeugen. Der Aufruf dieses Makros erfolgt direkt im Sourcefile (ausserhalb der Funktionen).

- Ausserdem muss die Klasse im Core registriert werden, damit sie in den Objectlists erscheint. Dies funktioniert mit RegisterObject(Classname) im Konstructor.

```
CreateFactory(MyClass);

MyClass::MyClass(BaseObject* creator) : Parent(creator)
{
	RegisterObject(MyClass);
}
```

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

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“
```

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 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

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.