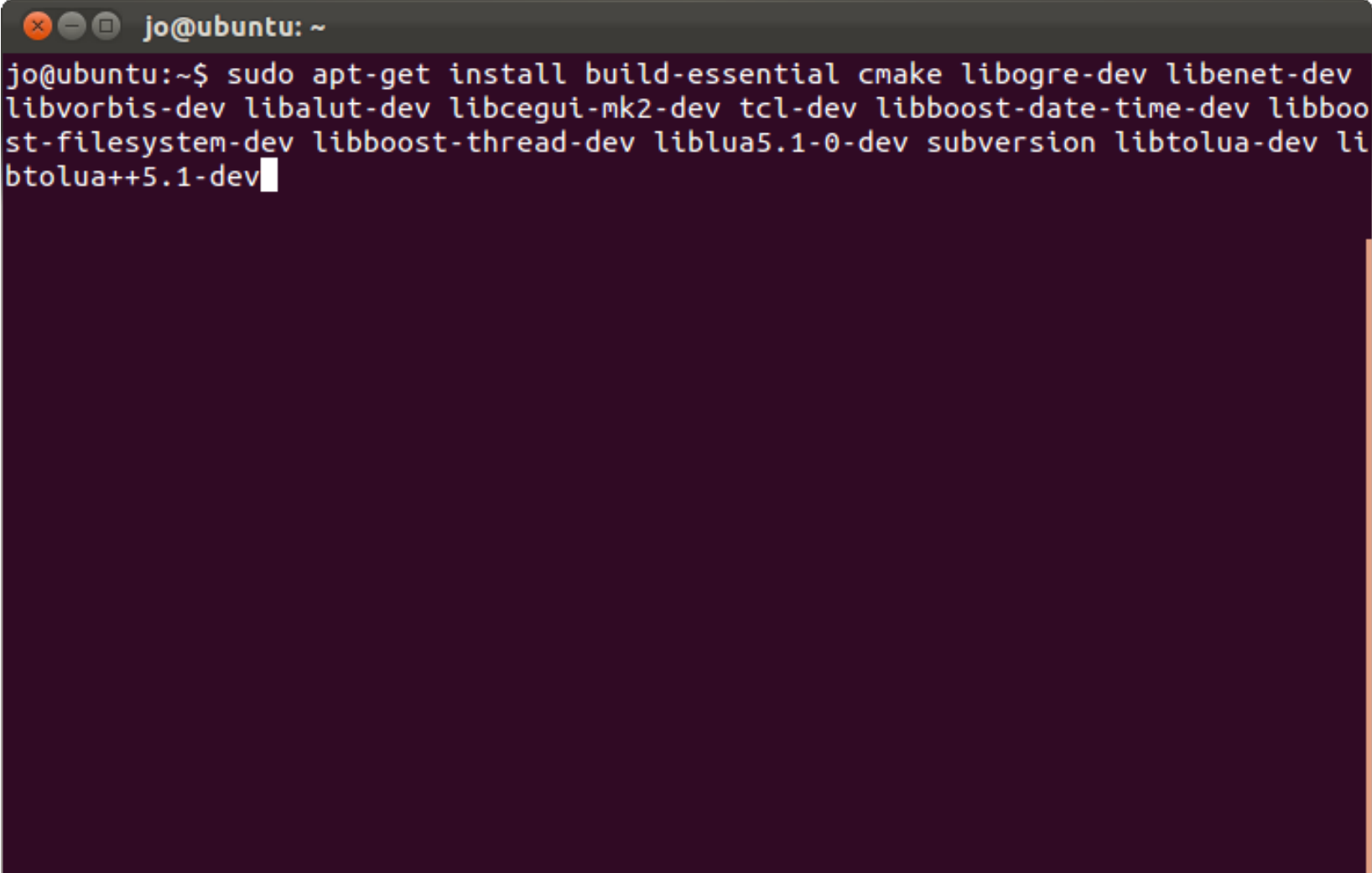


# Framework & Coding

Einleitung in das Framework von Orxonox

# Installation – Libraries



```
jo@ubuntu: ~  
jo@ubuntu:~$ sudo apt-get install build-essential cmake libogre-dev libenet-dev  
libvorbis-dev libalut-dev libcegui-mk2-dev tcl-dev libboost-date-time-dev libbo  
ost-filesystem-dev libboost-thread-dev liblua5.1-0-dev subversion libtolua-dev li  
btolua++5.1-dev
```

# Libraries

Ogre (Grafikengine)



# Libraries

Ogre (Grafikengine)



# Libraries

Ogre (Grafikengine)



# Libraries

Ogre (Grafikengine)



# Libraries

## CEGui (GUI-Engine)

The image shows a screenshot of a CEGui control panel for a table demo. The panel is titled "Demo 6 - Control Panel" and contains several sections for managing a table.

**Table Data:**

Column 1	Column 2	Test.2
Test Data		1234567890
Test item	Another item	Third Column Entry
Abcdefg	More items	

**Column Control:**

ID Code:  Width:  Caption:  (Add)

ID Code:  (Delete Column)

**Row Control:**

Col ID:  Item Text:  (Add)

Row Idx:  (Delete Row)

**Item Modification:**

Row Idx:  Col ID:  Item Text:  (Set)

Current Row Count: 3  
Current Column Count: 3  
Current Selected Count: 2

**Performance Metrics:**

Current FPS: 263.736  
Average FPS: 263.195  
Worst FPS: 47.4543 1616 ms  
Best FPS: 277.168 3 ms  
Triangle Count: 1254

**Buttons:** (Quit This Demo!)

**Logo:** OGRE

# Libraries

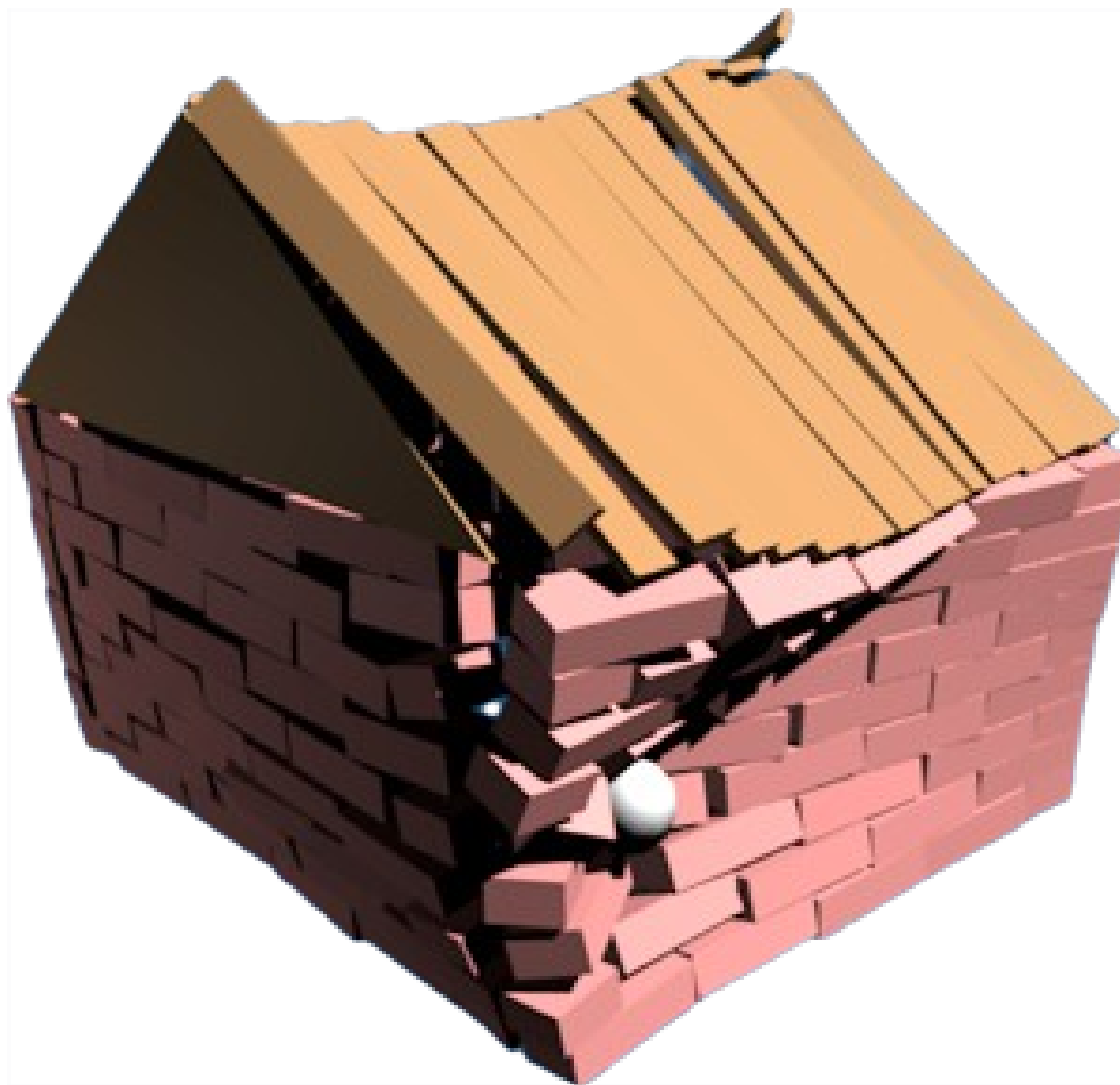
## CEGui (GUI-Engine)



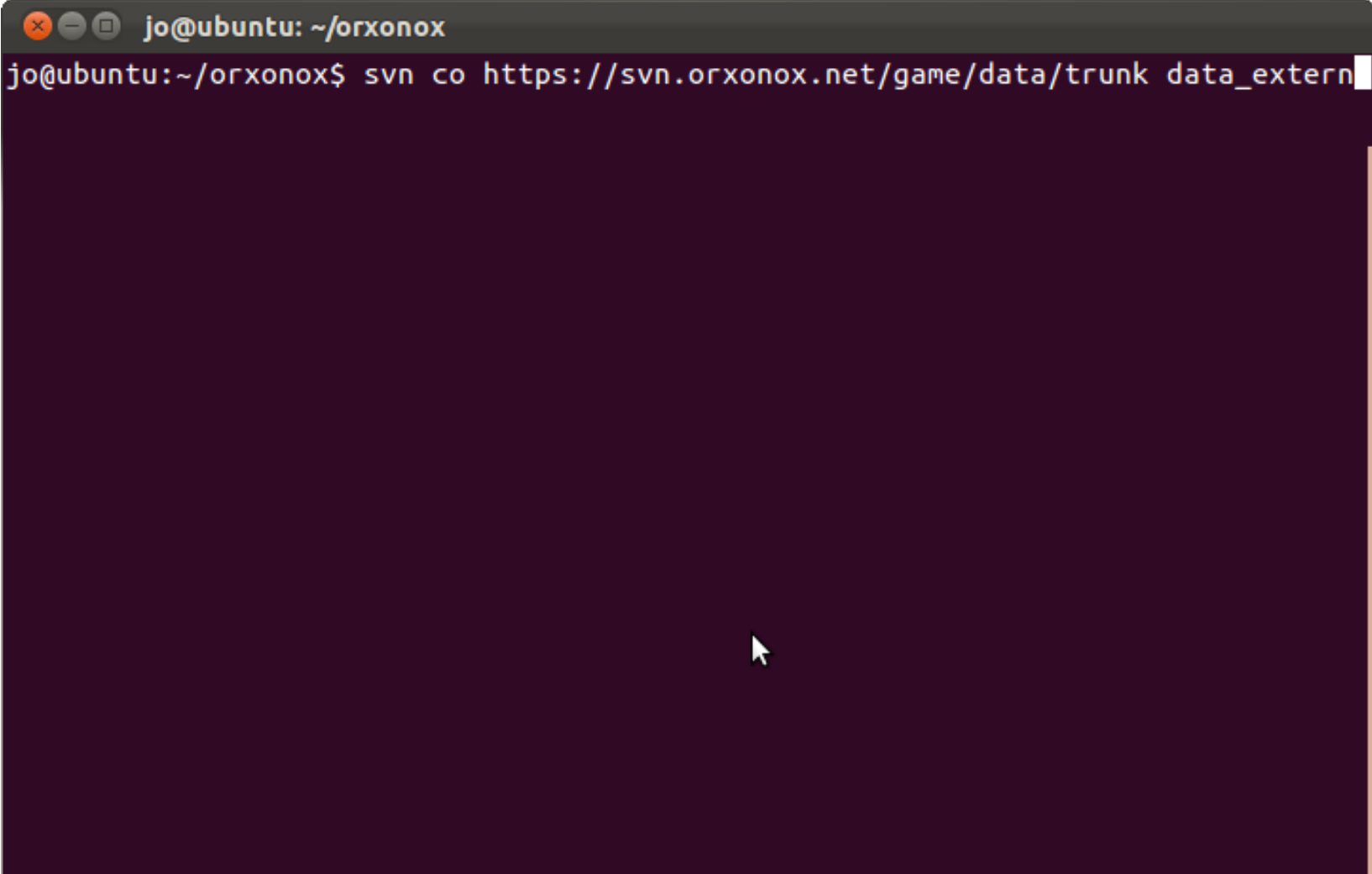


# Libraries

Bullet (Physikengine)



# Installation - Content

A terminal window with a dark purple background. The title bar shows 'jo@ubuntu: ~/orxonox'. The command 'svn co https://svn.orxonox.net/game/data/trunk data\_extern' is entered at the prompt. A mouse cursor is visible at the bottom center of the terminal area.

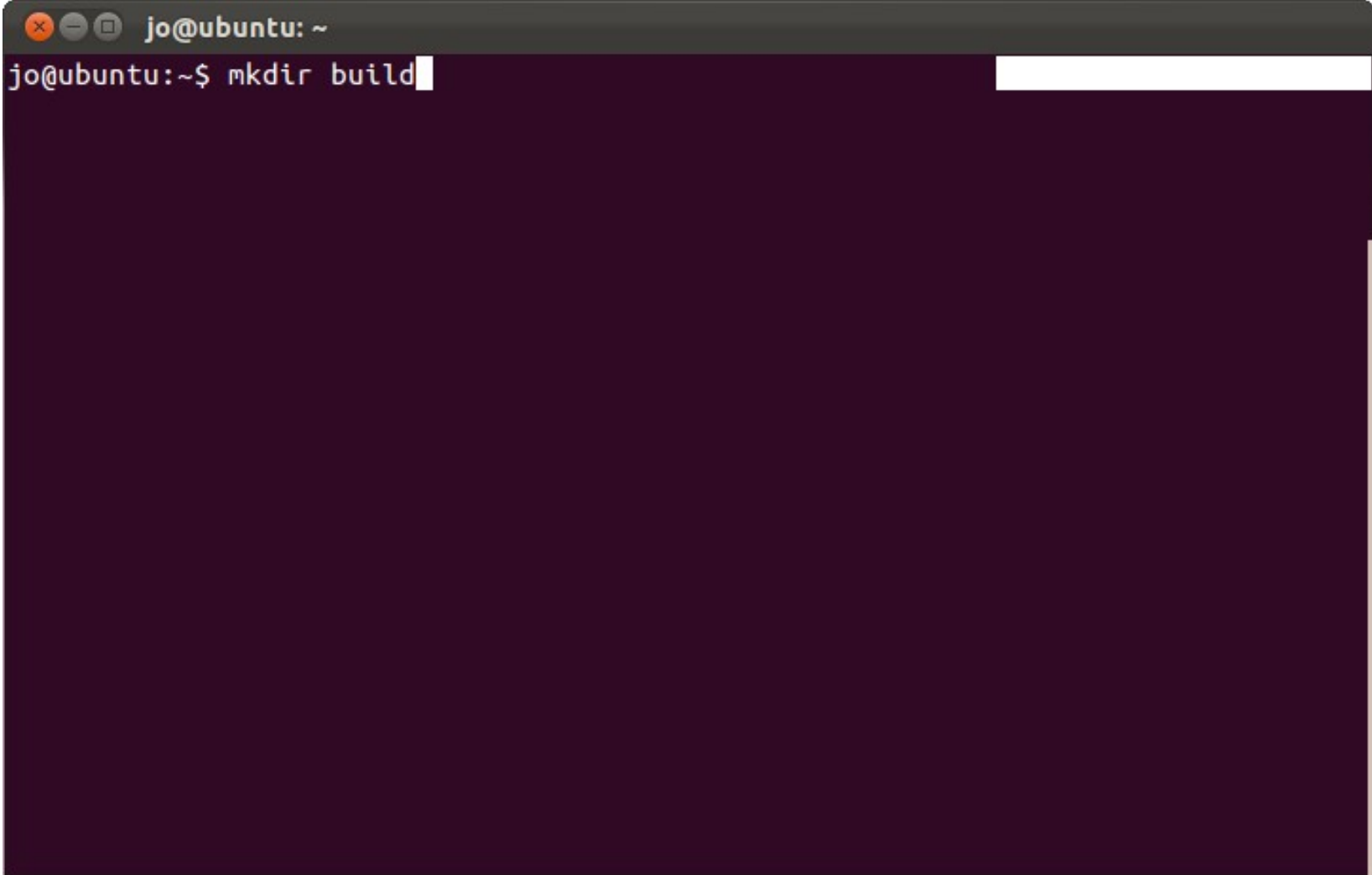
```
jo@ubuntu: ~/orxonox
jo@ubuntu:~/orxonox$ svn co https://svn.orxonox.net/game/data/trunk data_extern
```

# Installation - Code



```
jo@ubuntu: ~/orxonox
jo@ubuntu:~/orxonox$ svn co http://svn.orxonox.net/game/code/trunk
```

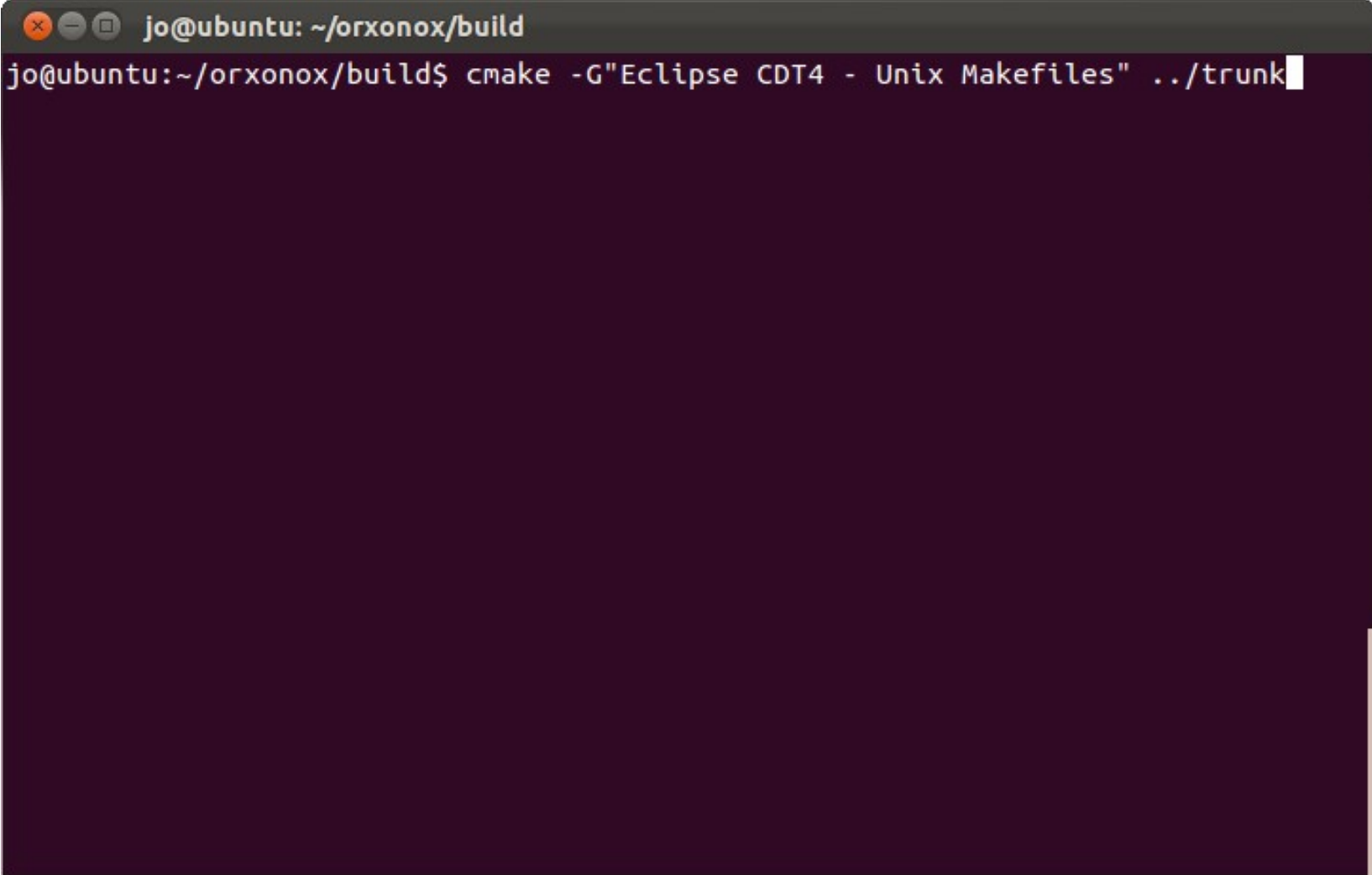
# Installation – Build Ordner

A terminal window with a dark purple background. The title bar at the top shows a window icon, a close button, and the text 'jo@ubuntu: ~'. The main area of the terminal displays the command 'jo@ubuntu:~\$ mkdir build' with a white cursor at the end of the word 'build'.

```
jo@ubuntu: ~  
jo@ubuntu:~$ mkdir build
```

```
jo@ubuntu: ~/orxonox
jo@ubuntu:~/orxonox$ cd build
```

# Installation - CMake


A terminal window with a dark purple background. The title bar shows 'jo@ubuntu: ~/orxonox/build'. The command prompt shows 'jo@ubuntu:~/orxonox/build\$ cmake -G"Eclipse CDT4 - Unix Makefiles" ../trunk' with a white cursor at the end of the line.

```
jo@ubuntu: ~/orxonox/build
jo@ubuntu:~/orxonox/build$ cmake -G"Eclipse CDT4 - Unix Makefiles" ../trunk
```

# CMake

- Findet die benötigten Libraries
- Erstellt ein Makefile
- Kann IDE-Projekt-Dateien erstellen

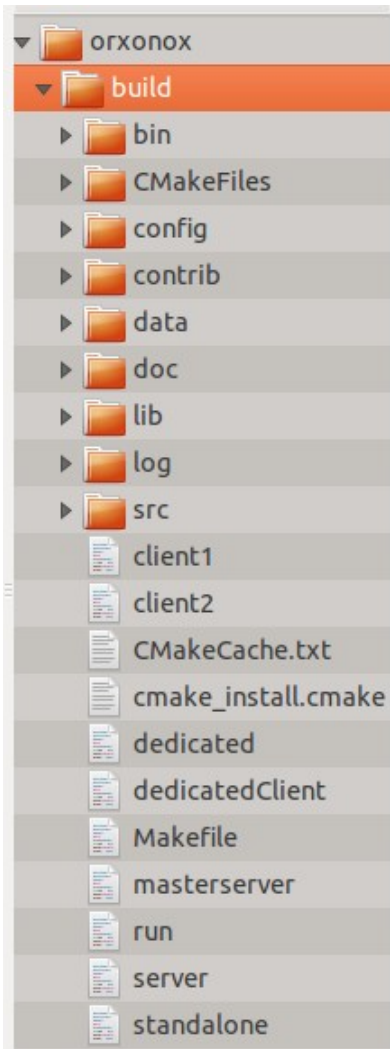
# Installation - Make



```
jo@ubuntu: ~/orxonox/build
jo@ubuntu:~/orxonox/build$ make -j4
```

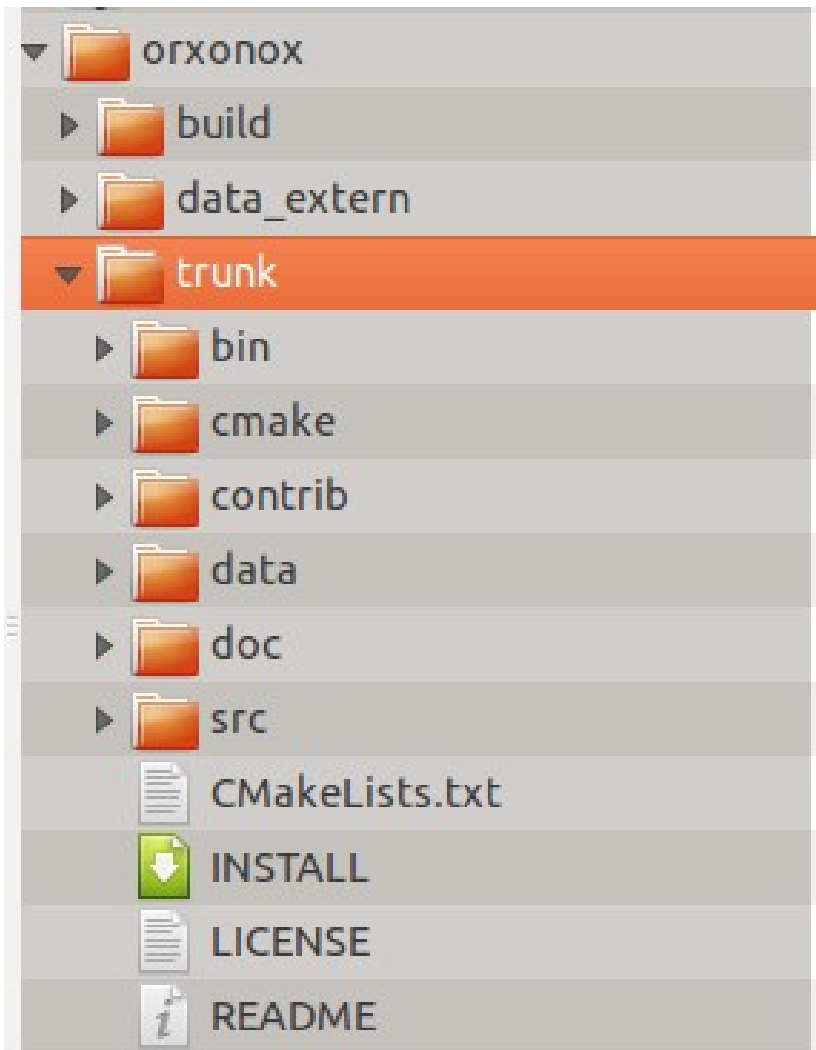


# Verzeichnisstruktur - Build



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

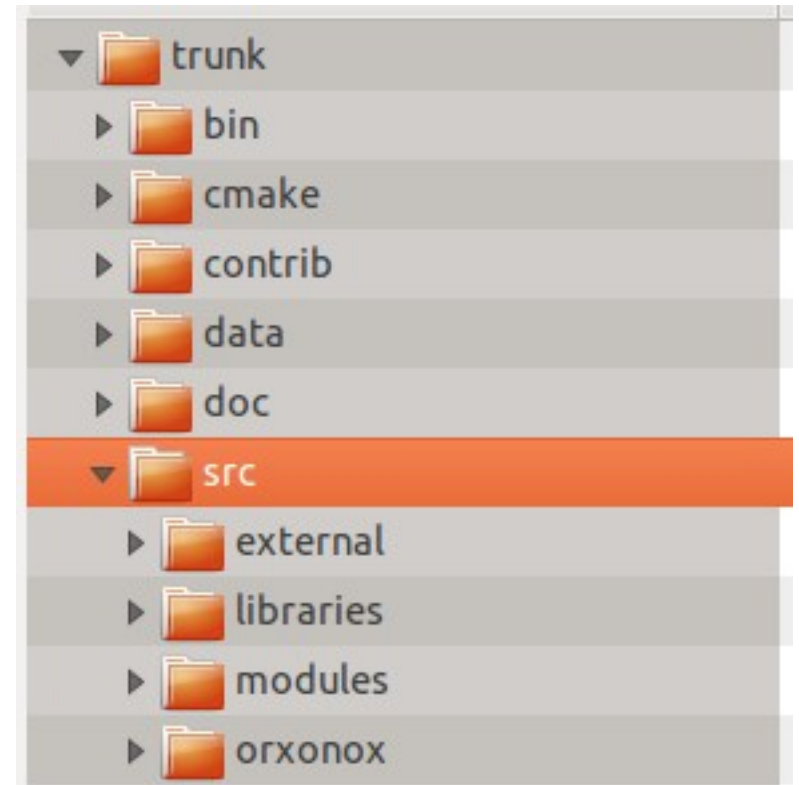
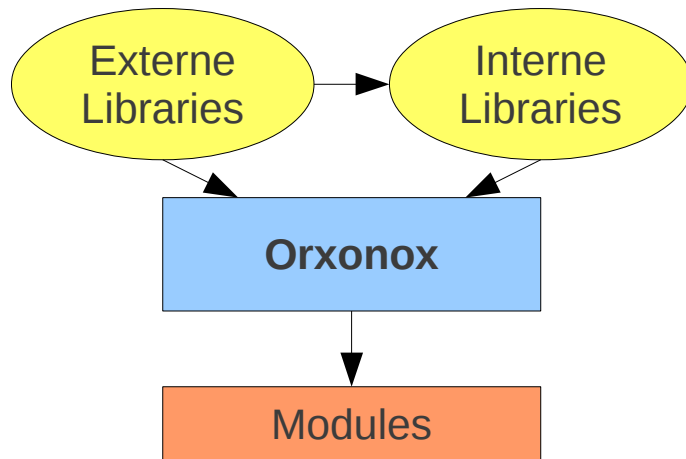
# Verzeichnisstruktur - Trunk



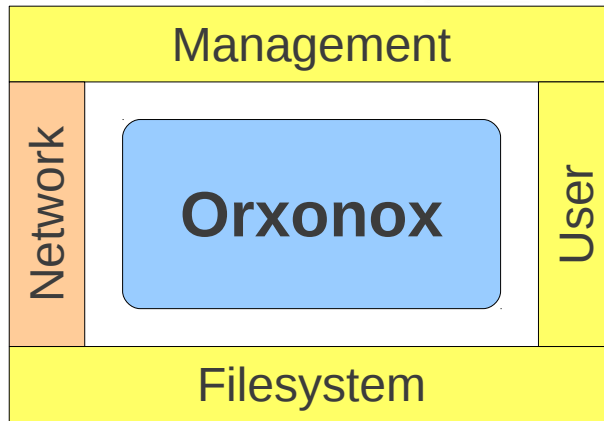
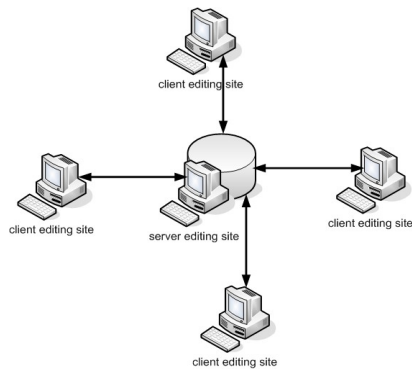
- cmake: CMake Scripts
- data: XML und Lua Scripts
- src: Quellcode

# Framework

## Struktur



# Framework



# Objekteigenschaften - Raum

- jedes Objekt, das im Level einen Ort hat, erbt von der Basisklasse „Worldentity“.
  - jedes Objekt, das man sehen kann
- Worldentity: Punkt und Vektor im Raum
- Worldentities:
  - Statisch: StaticEntity (z.B. eine Spacestation)
  - Beweglich: MovableEntity (z.B. ein Projektil)
  - Kontrolliert: ControllableEntity (z.B. ein Spaceship)

# Framework

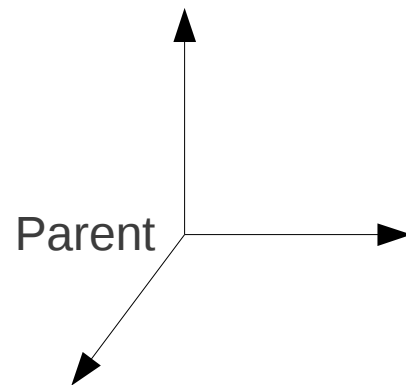
## Worldentities: Definition

- Worldentities können aneinander „attached“ werden, d.h. man kann sie zusammenhängen. Die Position des angehängten Objekts (Child) ist dann relativ zur Position und Rotation des Basisobjekts (Parent).

# Framework

## Worldentities: Attachen

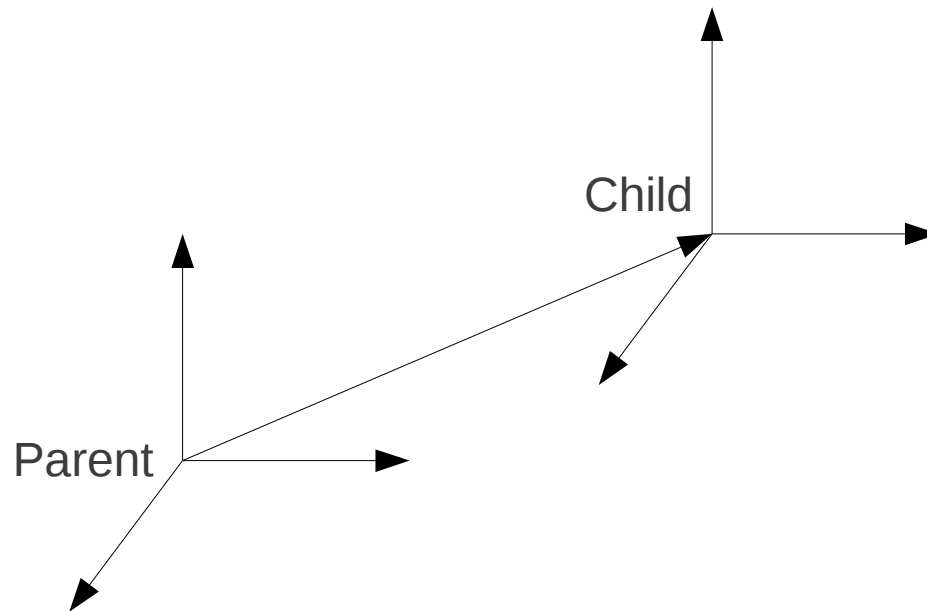
- Absolute Position im Raum (Parent):



# Framework

## Worldentities: Attachen

- Relative Position im Raum (Child):

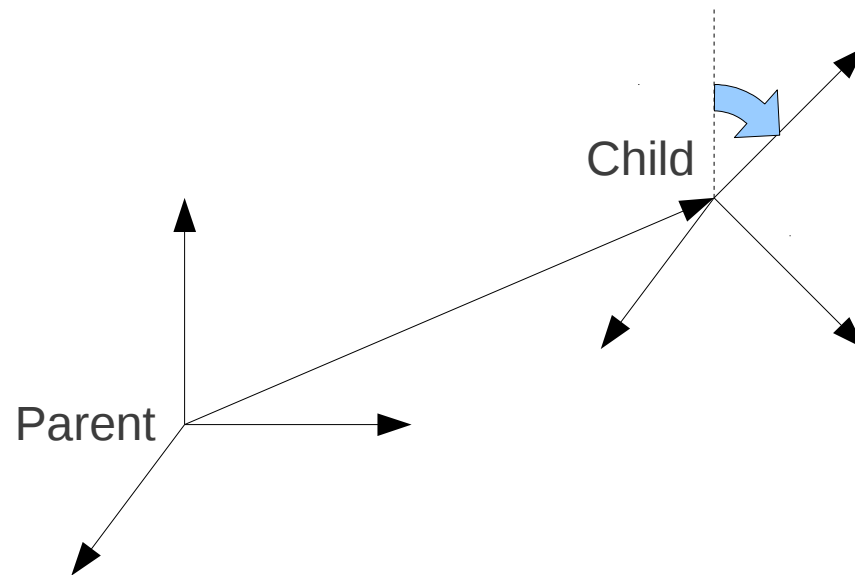




# Framework

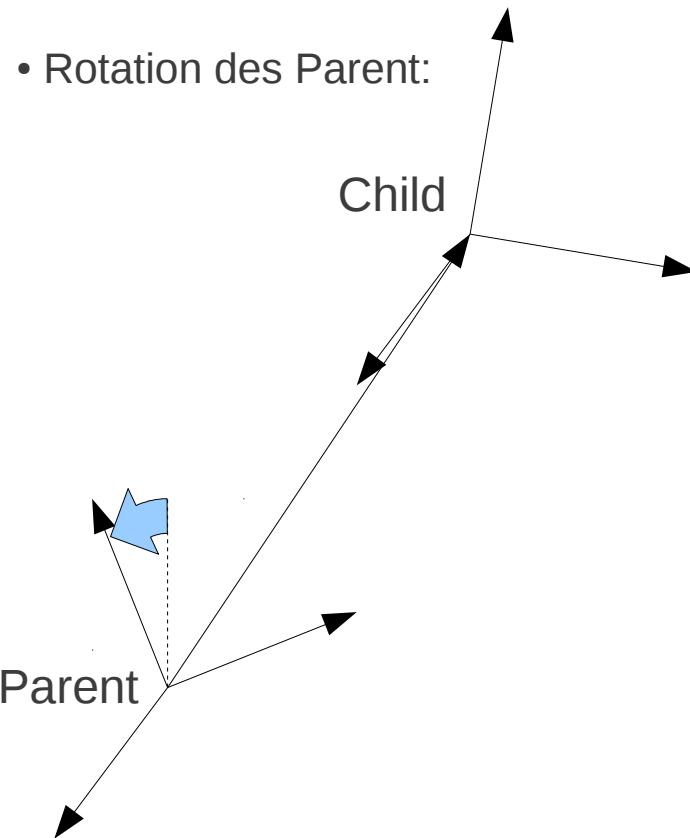
## Worldentities: Attachen

- Rotation des Child:



# Framework

## Worldentities: Attachen



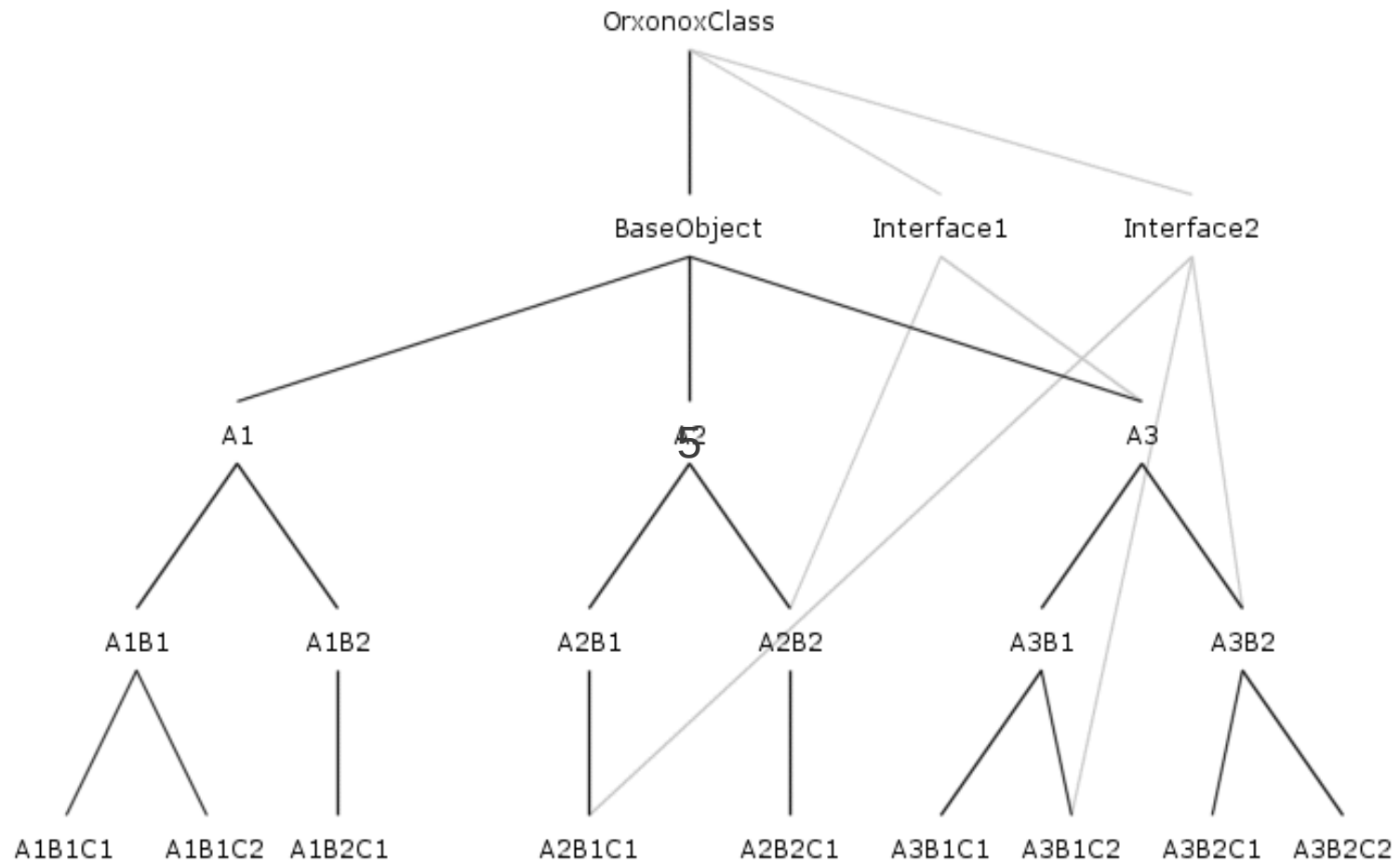
# Objekteigenschaften - Zeit

- ein Objekt ändert sich in der Zeit -> es muss vom Interface Tickable erben.
- Klassen die von Tickable erben, erben die Funktion tick(float dt).
- ein Frame wird gerendert -> tick(dt) wird aufgerufen
- dt: die Zeit seit dem letzten Aufruf von tick(dt)

# Objektmodellierung

- is–a–Relation:
  - ein Spaceship ist ein Worldentity
  - Mechanismus: Vererbung
- has–a–Relation:
  - „ein Spaceship hat Waffen, einen Antrieb, ...“
  - Mechanismus: Pointer auf ein anderes Objekt

# Klassenhierarchie



# Klassenhierarchie - OrxonoxClass

- alle Klassen und Interfaces erben von OrxonoxClass
- notwendig für das Funktionieren des Frameworks -> Servicefunktionalität

# Klassenhierarchie - Baseobject

- Basisklasse aller „Objects“ in Orxonox
- Objects:
  - Können in ein Level geladen werden.
  - Werden am Ende des Levels wieder gelöscht.

# Klassenhierarchie - Baseobject

- Basisklasse aller „Objects“ in Orxonox
- Objects:
  - Können in ein Level geladen werden.
  - Werden am Ende des Levels wieder gelöscht.



# Framework

## Beispielklasse: CMakeLists.txt

- Wir erstellen zwei neue Dateien, MyClass.cc (das Source-File) sowie MyClass.h (das Header-File).
- Im gleichen Ordner in dem wir die Files erstellt haben, suchen wir die Datei „CMakeLists.txt“ und suchen nach einer Liste von anderen Source-Files. Dort Tragen wir MyClass.cc an einer beliebigen Stelle ein.
- Dadurch wird sichergestellt, dass unser neues File kompiliert wird.

# Framework

## Beispielklasse: Header

- Im Header-File Deklarieren wir die neue Klasse:

```
class MyClass : public MovableEntity
{
    public:
        MyClass(BaseObject* creator);
        virtual ~MyClass();

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

- Unsere Klasse erbt also von MovableEntity (ein bewegliches WorldEntity).
- Da MovableEntity ausserdem vom Interface Tickable erbt, erbt auch unsere Klasse die Tick-Funktion.

# Framework

## Beispielklasse: Source

- Im Source-File Implementieren wir das Grundgerüst der neuen Klasse:

```
MyClass::MyClass(BaseObject* creator)
{
}

MyClass::~~MyClass()
{
}

void MyClass::tick(float dt)
{
}
```

# Framework

## Beispielklasse: CreateFactory

- Zuerst müssen wir eine Factory erstellen, damit unsere Klasse vom Framework erkannt und auch über XML geladen werden kann:

```
CreateFactory(MyClass);
```

```
MyClass::MyClass(BaseObject* creator)  
{  
}
```

```
MyClass::~MyClass()  
{  
}
```

```
void MyClass::tick(float dt)  
{  
}
```

# Framework

## Beispielklasse: RegisterObject

- Als nächstes müssen wir direkt zu Beginn des Constructors unser Objekt registrieren:

```
CreateFactory(MyClass);

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

MyClass::~MyClass()
{
}

void MyClass::tick(float dt)
{
}
```

# Framework

## Beispielklasse: Creator

- Ausserdem müssen wir den creator-Pointer an die Basisklasse weitergeben. Er übermittelt den Kontext, in dem ein Objekt erzeugt wurde:

```
CreateFactory(MyClass);
```

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

```
MyClass::~MyClass()
{
}
```

```
void MyClass::tick(float dt)
{
}
```

# Framework

## Beispielklasse: SUPER

- Damit nicht nur die Tick-Funktion von MyClass aufgerufen wird, sondern auch weiterhin der Tick von MovableEntity, müssen wir den Aufruf der Tick-Funktion an die Basisklasse weiterleiten:

```
CreateFactory(MyClass);

MyClass::MyClass(BaseObject* creator) : MovableEntity(creator)
{
    RegisterObject(MyClass);
}

MyClass::~~MyClass()
{
}

void MyClass::tick(float dt)
{
    SUPER(MyClass, tick, dt);
}
```

# Framework

## Beispielklasse: orxout()

- Schlussendlich wollen wir noch etwas (sinnlose) Action in die Klasse bringen, daher geben wir in jedem Tick einen Text in die Konsole aus:

```
CreateFactory(MyClass);

MyClass::MyClass(BaseObject* creator) : MovableEntity(creator)
{
    RegisterObject(MyClass);
}

MyClass::~MyClass()
{
}

void MyClass::tick(float dt)
{
    SUPER(MyClass, tick, dt);

    orxout() << „Hello World“ << endl;
}
```



# Grafik & XML Demo

- Models
- Billboard
- Particle Effects

# Framework

## XML

- XML ist eine textbasierte Sprache, die die Interaktion zwischen verschiedenen Programmen und menschlichen Autoren ermöglicht.
- XML ermöglicht die Beschreibung von Attributen und Objekten.
- XML weist die selbe Form wie HTML auf.
- Wir verwenden XML, um Levels und andere Ansammlungen von Klassen (z.B. HUDs) zu beschreiben.
- Beispiel:

```
<MyClass myvalue="1" myothervalue="Hello World">  
  <subclasses>  
    <OtherClass somevalue="1.111" />  
    <OtherClass somevalue="2.222" />  
  </subclasses>  
</MyClass>
```

# Framework

## XMLPort

- XMLPort ist unser Interface zwischen XML und C++.
- In XMLPort wird definiert, welche Objekte und Attribute in XML beschrieben werden können. Ausserdem werden Funktionen definiert, um diese Attribute lesen und schreiben zu können.
- Für jeden Wert braucht es ein Paar von set- und get-Funktionen. Die set-Funktion setzt den Wert im Objekt, die get-Funktion liest ihn aus.

### • Beispiel:

```
void MyClass::XMLPort(...)
{
    SUPER(MyClass, XMLPort, ...);

    XMLPortParam(MyClass, "myvalue", setValue, getValue, xmlelement, mode);
    XMLPortParam(MyClass, "myothervaluevalue", setOtherValue, get...

    XMLPortObject(MyClass, OtherClass, "subclasses", addSubclass, getSubclass, xmlelement, mode);
}
```

