ORXONOX – Das Wichtigste in Kürze

Allgemein

Orxonox ist in C++ programmiert, verwendet Objektorientierung, Templates, virtuelle Funktionen und andere Features, die C++ bietet. Grundkenntnisse sollten vorhanden sein (Informatik I Vorlesung), die komplizierteren Dinge werden allerdings im Framework versteckt und können über ein einfaches Interface verwendet werden.

Zu C++ gehört auch eine Standardbibliothek mit nützlichen Funktionen und Klassen. Sie beinhaltet unter anderem die „string“ Klasse, sowie Templates für Listen, Maps und mehr. Es wird dringend empfohlen, diese Klassen nach Möglichkeit zu verwenden, anstatt sie selber neu zu implementieren.

Lesenswert:

- http://www.orxonox.net/wiki/c%2B%2B_styleguide – Tipps für lesbaren Code
- http://www.orxonox.net/wiki/PerformanceTips – Tipps für schnellen Code
- http://cplusplus.com/reference/ – Dokumentation der C/C++-Standardbibliothek

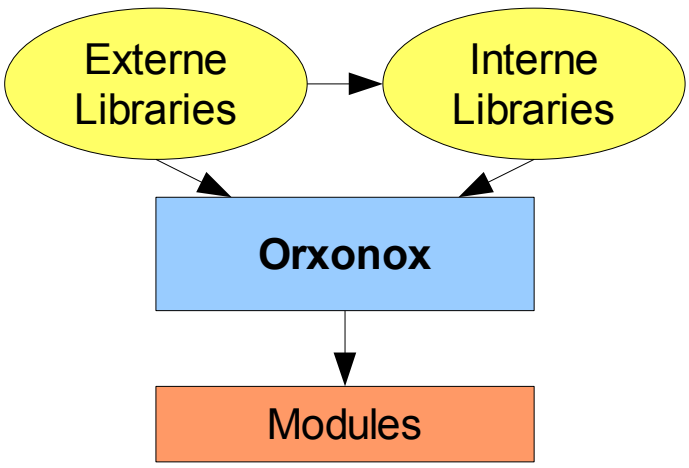
Tipps

Iterativ entwickeln – Schreibt zuerst das Grundgerüst für euren Code, schaut ob alles kompiliert und ob es korrekt ins Game eingebunden wird. Dann könnt ihr die ersten Features hinzufügen, wieder kompilieren, wieder testen, etc. Allgemein gilt, je früher ihr euren Code testen könnt, desto schneller seht ihr Probleme, fehlende Features und allenfalls auch Fehlüberlegungen in der Planung.

Fragen – Egal ob es sich um Probleme mit der C++-Syntax, dem Framework, Linux/Tardis, SVN, Cmake, dem Compiler oder sonst etwas handelt, oder ob ihr euch bei der Planung und dem generellen Vorgehen bei der Implementierung unsicher seid, fragt eure Assistenten. Sie wurden dazu erzogen euch zu helfen und freuen sich über jede Beachtung die man ihnen schenkt. Und über ein Leckerli.

Framework

Das Framework stellt das Grundgerüst von Orxonox dar. Es enthält sowohl externe Libraries (u.a. für Grafik und Physik) als auch interne Libraries, die von uns entwickelt und laufend ausgebaut werden (u.a. Core und Network). Diese Libraries beinhalten die Grundfunktionen und Basisklassen, auf denen das eigentliche Game aufbaut.



Die internen Libraries lassen sich in vier Teilbereiche aufgliedern: Management der Klassen und Objekte, das Userinterface, die Netzwerkkommunikation sowie die Interaktion mit dem Filesystem. Unten folgt eine Liste mit Features des Frameworks, die hier allerdings nicht näher erläutert werden. Wenn ihr denkt ihr könnt eines dieser Features verwenden, fragt einen Assistenten oder schaut in der Wiki nach, respektive adaptiert Code von einer bestehenden Klasse.

Management:

- Objectlists: Für jede Klasse existiert eine Liste, die alle Instanzen dieser Klasse speichert
- Identifier: Jede Klasse lässt sich identifizieren und mit anderen Klassen vergleichen
- Factory: Erstellt ein Objekt einer Klasse, deren Name als String vorliegt
- SmartPtr: Solange ein Smart Pointer auf ein Objekt zeigt, kann es nicht gelöscht werden
- WeakPtr: Wird ein Objekt gelöscht, auf das ein Weak Pointer zeigt, wird der Pointer NULL
- Tickable: Objekte die von diesem Interface erben, werden „getickt“
- Timer: Ruft eine bestimmte Funktion nach Ablauf einer gewissen Zeit auf
- Console-Commands: Aufrufen einer Funktion über die Konsole oder eine Taste
- Config-Values: Konfigurieren von Werten über ein Config-File.
- Util: Sammlung von Funktionen für mathematische Zwecke, Stringmanipulation sowie Konvertierung von Werten unterschiedlichen Typs
- Output: Schreiben von Text in ein Logfile, die Konsole oder eine In-Game-Shell

User-Interface:

- Keybindings: Ermöglichen das Steuern sowie den Aufruf einer Funktion durch Tastendruck
- Shell: Eine Shell die die Eingabe von Commands erlaubt (inkl. der Scriptsprache Tcl)
- GUI: Graphical user interface, z.B. das Hauptmenü oder das Inventar
- Overlays: Anzeige von Health, Fadenkreuz, Textnachrichten, etc.
- 3D-Objekte: Models, Billboards, Lichter, Partikel Effekte, etc.
- Physik: Dynamics (Bewegung, Gravitation, etc.) und Collisions
- 3D-Sound: Waffengeräusche, Triebwerke, Explosionen, etc.
- Hintergrundmusik: Abhänging von der Situation, Menümusik

Filesystem:

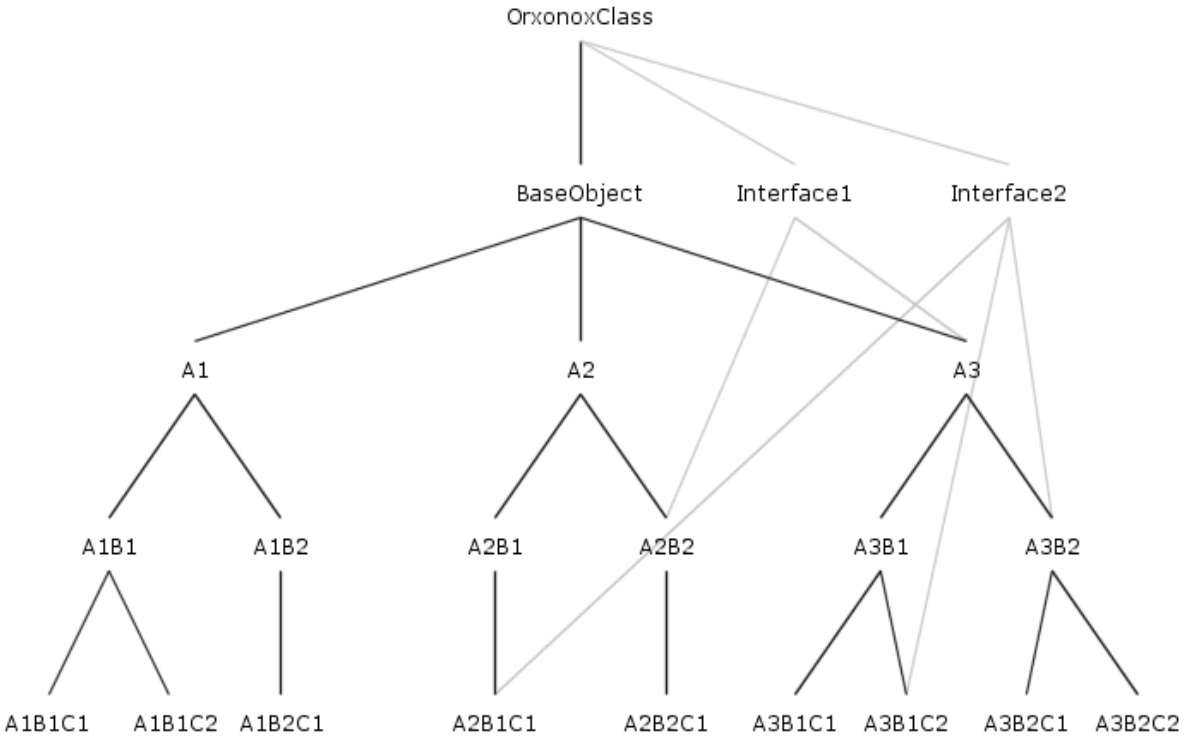
- XML: Laden und Speichern von Attributen und Objekten
- Lua-Scripts: Erlauben Code in einem XML File, z.B. für 1000 zufällige Asteroiden
- Templates: Definieren das „Aussehen“ eines Objekts
- Config-Files: Konfiguration von Parametern, Keybindings, Auflösung, etc.
- Laden von Resourcen: Models, Texturen, Sounds, etc.
- Logfile: Speichert den Programmablauf sowie Informationen im Falle eines Absturzes

Weiterführende Links:

- http://www.orxonox.net/wiki/Documentation – Doku in der Wiki (nicht immer aktuell)
- http://www.ogre3d.org/ – Ogre (Grafikengine)
- http://www.bulletphysics.com/ – Bullet (Physikengine)

Klassenhierarchie

Alle Objekte in Orxonox befinden sich in einer Hierarchie, d.h. spezialisierte Objekte erben von allgemeineren Objekten. An der Spitze befindet sich das BaseObject. Es stellt die Basisklasse aller Klassen dar, die als Objekte im Spiel erscheinen können. Objekte sind Elemente im Spiel, die entweder aktiv das Spiel beeinflussen, oder aber passiv vom Spiel beeinflusst werden. Objekte können ausserdem von sogenannten Interfaces erben. Interfaces können nicht direkt instanziiert werden, sie stellen jedoch gewisse Funktionalitäten zur Verfügung, die von allen erbenden Klassen übernommen werden. BaseObject selbst, sowie alle Interfaces, erben von OrxonoxClass, eine abstrakte Basisklasse die keine Spielrelevanten Eigenschaften hat, sondern lediglich Funktionen, die für das Framework benötigt werden.



Tickable

Tickable ist ein spezielles Interface. Objekte die von Tickable erben (direkt oder indirekt) erben die virtuelle Funktion tick(dt). Diese Funktion wird einmal pro „Tick“ aufgerufen, wobei man unter einem Tick einen kompletten Durchlauf der Game-Logik versteht. Bei jeden Tick wird ausserdem ein Frame gerendert. Die Variable dt enthält dabei die Anzahl Sekunden seit dem letzten Tick (üblicherweise im Bereich von 0.02 Sekunden, was 50 FPS entspricht).

Da Objekte, die von Tickable erben, diese Funktion beliebig implementieren dürfen, können sie in jedem Tick eine bestimmte Aktion vornehmen. Dies ermöglicht Bewegung (z.B. in jedem Tick 1 Unit nach vorne bewegen), künstliche Intelligenz und vieles mehr. Objekte die nicht von Tickable erben, bleiben das gesamte Spiel über unverändert.

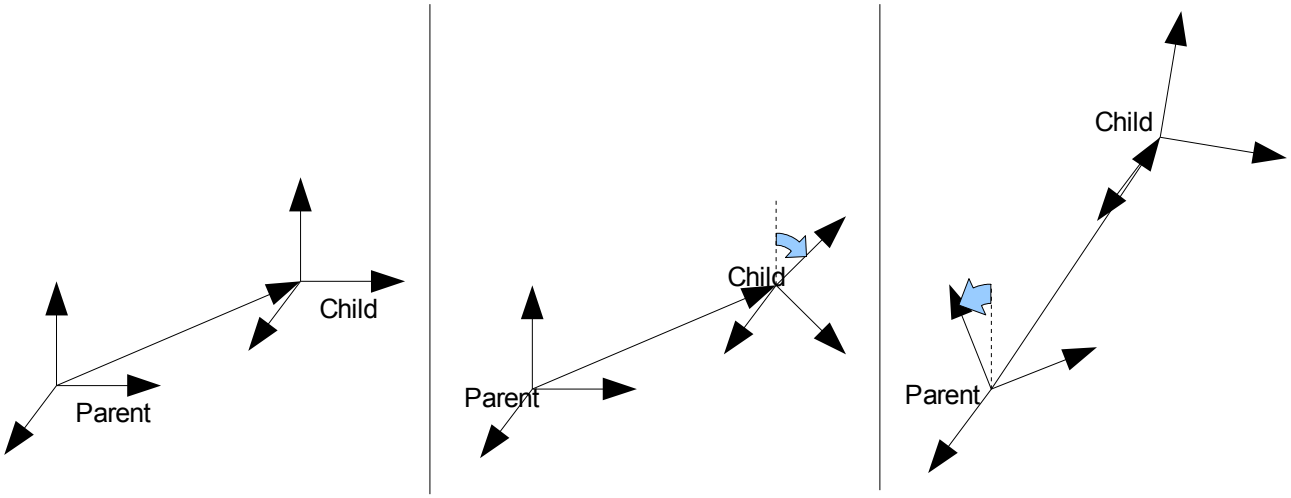
Beispiel (konstante Bewegung):

```
void MyClass::tick(float dt)
{
	static Vector3 speed = Vector3(0, 0, -100);

	Vector3 position = this->getPosition();
	position += (speed * dt);
	this->setPosition(position);
}
```

WorldEntity

Ein ganz besonderes Objekt in der Klassenhierarchie stellt das WorldEntity dar. Es ist die Basisklasse aller Objekte, die eine Position im Raum haben. WorldEntities können positioniert, rotiert und attached werden, wobei man unter attachen das „ankleben“ eines WorldEntities an ein anderes WorldEntity versteht. Die Position und Rotation des Objekts das attached wurde, ist relativ zu seinem Basis-Objekt. Bewegt sich das Basis-Objekt, bewegen sich alle „angeklebten“ Objekte mit.



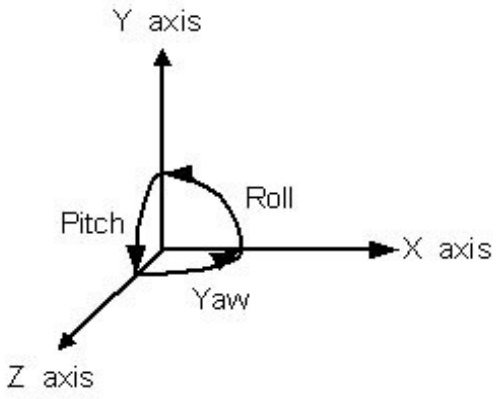
Aus Performancegründen unterscheiden wir zwischen drei unterschiedlichen Subklassen von WorldEntity: **StaticEntity** für unbewegliche Objekte, **MovableEntity** für Objekte die sich gleichförmig bewegen oder rotieren, sowie **ControllableEntity** für Objekte, die sich beliebig bewegen können, wobei die Bewegung von einer Controller Klasse festgelegt wird (z.B. künstliche Intelligenz).

Das Koordinatensystem

Wichtig im Umgang mit Worldentities ist natürlich das Koordinatensystem. In Orxonox ist dies etwas anders als üblich definiert, nämlich x nach rechts und y nach oben (dies erlaubt die Anordnung von 2D-Elementen auf dem Bildschirm in der xy-Ebene). Damit es sich um ein Rechtskoordinatensystem handelt, folgt, dass die z-Achse nach hinten in Richtung des Betrachters schaut. „Vorne“ ist also die **negative** z-Achse.

Rotationen werden gemäss den im Flugverkehr üblichen Bezeichnungen beschrieben, nämlich yaw (Drehung um die y-Achse, „links-rechts“), pitch (Drehung um die x-Achse, „hoch-runter“) und roll (Drehung um die z-Achse, wie ein Propeller). Die Vorzeichen der Drehungen sind so definiert, dass man den Daumen der rechten Hand in Richtung der Achse hält; die restlichen Finger beschreiben dann eine Rotation in die positive Richtung (vergleichbar mit der elektromagnetischen Induktion).

Wichtig: Wenn ihr diese Regeln vergesst, schaut nicht im Internet nach, die meisten Quellen verwenden andere Konventionen. *(Illustration: siehe nächste Seite)*



Grafische Komponenten

Ein normales WorldEntity ist nur ein abstrakter Punkt im Raum, man kann es also nicht sehen. Um ein Objekt sichtbar zu machen, muss man eine der grafischen Komponenten attachen. Bei diesen Komponenten handelt es sich ebenfalls um WorldEntities, d.h. die oben genannten Regeln bezüglich der relativen Position gelten auch hier.

- Model: Ein dreidimensionales Objekt (Mesh) mit Textur (Material)
- Billboard: Eine zweidimensionale Textur die sich immer zum Betrachter ausrichtet
- ParticleEmitter: Erzeugt einen dauerhaften Partikeleffekt (z.B. Feuer, Rauch, Schnee)
- ParticleSpawner: Subklasse des ParticleEmitters, erzeugt den Partikeleffekt nur für eine bestimmte Zeit (z.B. Explosion)
- Light: Wird benötigt um ein Model zu beleuchten. Man unterscheidet drei Typen:
    - Ambient Light: Ein Hintergrundleuchten, wirft keinen Schatten, überall gleich hell
    - Point Light: Vergleichbar mit einer Glühbirne, strahlt in alle Richtungen
    - Spotlight: Wirft einen Lichtkegel

CmakeLists.txt

In beinahe jedem Verzeichnis befindet sich eine Datei namens CMakeLists.txt. Sie enthält eine Liste von Source-Files, die beim Kompilieren von Orxonox berücksichtigt werden sollen. Davon abgesehen enthalten diese Dateien häufig auch noch weitere Anweisungen, etwa das Einbinden von externen Header-Files oder das Erzeugen von Libraries und Executables.

Wenn man ein neues Source-File erstellt, sollte man es als erstes in der CMakeLists.txt, die sich im selben Ordner befindet, eintragen.

CreateFactory

Für jede Klasse die zur Klassenhierarchie gehört (mit Ausnahme von Interfaces, abstrakten Klassen) muss eine Factory erzeugt werden. Eine Factory dient generell dazu, die Klasse beim Framework „anzumelden“. Speziefisch wird die Factory später verwendet, um Instanzen dieser Klasse über ein XML-File erzeugen zu können.

Um eine Factory zu erzeugen, muss `CreateFactory(ClassName);` **ausserhalb** einer Funktion (am besten direkt über dem Constructor) aufgerufen werden.

RegisterObject

Der Zweck von RegisterObject ist vergleichbar mit CreateFactory, nur dass RegisterObject **Objekte** beim Framework registriert, während CreateFactory die **Klasse** registriert. Um dies zu erreichen, muss `RegisterObject(ClassName);` **innerhalb des Constructors** aufgerufen werden. Dabei gilt, dass man diesen Aufruf so früh wie möglich machen sollte, d.h. noch bevor mit dem Aufrufen von Funktionen und Erzeugen von Objekten begonnen wird.

Als wichtige Nebenwirkung von RegisterObject ist zu nennen, dass beim Start von Orxonox von jeder Klasse ein Objekt erzeugt wird, um die Struktur der Klassenhierarchie zu evaluieren. Dabei wird ein Objekt aber nicht vollständig erzeugt, sondern nur bis RegisterObject. Danach wird der Constructor abgebrochen. Deswegen muss man im **De**structor darauf achten, dass man beim Zerstören des Objekts überprüft, ob das Objekt vollständig erzeugt wurde. Dies geht z.B. durch den Aufruf von `if (this->isInitialized())`, man kann aber auch einfach alle Pointer mit NULL initialisieren **bevor** man RegisterObject aufruft.

Creator

Jedes Objekt das von BaseObject erbt, benötigt einen Creator im Constructor. Der Creator ist ein Pointer auf das Objekt, das das neue Objekt erzeugt hat. Dies dient dazu, den Kontext zu übermitteln, in dem ein Objekt erzeugt wird. Der Kontext sagt z.B. zu welchem Level ein Objekt gehört, sowie in welcher Scene es sich befindet.

Der Creator ist jedoch nicht Ausdruck einer expliziten oder impliziten Hierarchie (Parent), er dient lediglich der Übermittlung des Kontexts.

Destroy

Normalerweise kann man Objekte mit `delete myObject;` zerstören. Dies ist natürlich auch in Orxonox möglich, jedoch sollte man dies bei Objekten, die zur Klassenhierarchie gehören, vermeiden. Stattdessen verwenden wir `myObject->destroy();` Dies stellt sicher, dass allfällige Smart Pointer berücksichtigt werden, und das Objekt nur dann zerstört wird, wenn dies auch tatsächlich erlaubt ist. Ein direkter Aufruf von `delete` kann daher zu schwerwiegenden Fehlern führen.

SUPER

Das Makro SUPER leitet den Aufruf einer virtuellen Funktion an die Basisklasse weiter. Im Falle der Tick-Funktion sieht das wie folgt aus:

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

Dies ist äquivalent zum herkömmlichen Aufruf:

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

Der Unterschied ist lediglich, dass man bei der herkömmlichen Methode den Namen der Basisklasse kennen muss, während man bei der Verwendung von SUPER den Namen der **eigenen** Klasse angibt. Dadurch wird der Code unabhängiger von Veränderungen in der Klassenhierarchie.

orxout()

Die Funktion orxout() verhält sich beinahe identisch wie std::cout, d.h. sie gibt Text in die Konsole aus. Allerdings gibt es zwei Unterschiede: Erstens wird der Text nicht bloss in die Konsole ausgegeben, sondern ausserdem in das Logfile und die In-Game-Shell und zweitens unterstützt die orxout() Funktion Levels, die die Wichtigkeit von Output beschreiben.

Syntax: orxout(*level*) << „text“ << endl;

Dabei ist *level* ein Enum, dessen Werte verschiedene Bedeutungen haben:

- debug_output: Temporärer Debug Output, erscheint immer (alternativ orxout() ohne Level)
- user_error, user_warning: Wird für Errors und Warnings verwendet, die den User betreffen
- user_status, user_info: Informiert den User über den Status der Applikation
- internal_error, internal_warning: Errors und Warnings für Developer, nur im Logfile sichtbar
- internal_status, internal_info: Interner Status der Applikation, nur im Logfile sichtbar
- verbose: Ausführlicher Debug Output, der normalerweise verborgen bleibt. Um diesen Output sichtbar zu machen, muss der entsprechende Kontext aktiviert werden.

XML

XML steht für Extensible Markup Language und ist eine simple Formatierungssprache für Daten. XML zeichnet sich dadurch aus, dass es für Menschen lesbar und modifizierbar ist. Syntaktisch ist XML mit HTML verwandt.

Ein XML-Dokument besteht aus verschiedenen Elementen, wobei ein Element verschiedene Attribute haben kann. Ausserdem kann ein Element Subelemente besitzen, die dem umschliessenden Element zugeordnet werden (z.B. Waffen einem Spieler zuordnen).

Beispiel:

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

„MyClass“ stellt in diesem Beispiel das Root-Element dar, „subclasses“ ist eine Subsection innerhalb MyClass (eine Subsection dient nur zur Gliederung und hat keine weitere Bedeutung), welche wiederum zwei Elemente vom Typ „OtherClass“ enthält. Die Attribute „myvalue“ und „myothervalue“ beschreiben Variabeln von MyClass, während „somevalue“ eine Variable von OtherClass darstellt.

Die Syntax von XML schreibt vor, dass jedes Element mit <Elementname> geöffnet wird und mit </Elementname> geschlossen (siehe MyClass). Sofern ein Element keine Subelemente enthält, kann man die beiden „Tags“ zu Einem zusammenfassen: <Elementname /> (siehe Otherclass).

Attribute werden nach dem Elementname aufgelistet, die Werte der Attribute sollten von Anführungszeichen umschlossen sein, wobei diese bei Werten, die aus nur einem Wort (oder einer Zahl) bestehen, fakultativ sind.

Beispiel: <Elementname attributname=1 />, aber: <Elementname attributname=“Hello World“ />

In Orxonox ist es ausserdem möglich, XML-Files mittels der Scriptsprache „Lua“ zu automatisieren. Dies ist vergleichbar mit PHP in HTML Dokumenten und ermöglicht zum Beispiel das Erstellen von 1000 zufallsgenerierten Asteroiden in wenigen Zeilen.

XMLPort

Um XML mit C++ zu verbinden, verwenden wir die Funktion „XMLPort“. Dieser Funktion wird ein XML-Element übergeben. Aus diesem Element werden dann innerhalb der XMLPort-Funktion die verschiedenen Attribute und Subobjekte ausgelesen.

```
virtual void XMLPort(Element& xmlelement, XMLPort::Mode mode);
```

Das Prinzip von XMLPort ist, dass immer zwei Funktionen übergeben werden: Eine Funktion um Attribute oder Objekte zu laden und eine zweite Funktion um sie zu speichern.

XMLPortParam wird verwendet, um XML-Attribute mit Variabeln in C++ zu verbinden:

```
XMLPortParam(MyClass, „attributname“, setFunction, \
                                  getFunction, xmlelement, mode);
```

MyClass ist der Name der Klasse, „attributname“ definiert den Namen des Attributes wie er später in XML verwendet wird. Über die „setFunction“ wird der Parameter an C++ übergeben und über die „getFunction“ wird er von C++ an XML zurückgegeben. Wenn es sich beim Attribut um einen Integer handelt, würden die beiden Funktionen wie folgt aussehen:

```
void setFunction(int value);

int getFunction() const;
```

Die Variable selbst kann als Membervariable in MyClass gespeichert werden, sie kann sich aber theoretisch auch woanders (z.B. in einem Unterobjekt) befinden.

Die Parameter „element“ und „mode“ wurden vom Framework an XMLPort übergeben und werden nun einfach an XMLPortParam weitergeleitet.

Bei Subobjekten verhält es sich fast gleich, ausser dass hier XMLPortObject verwendet wird. Ausserdem muss man dem Framework mitteilen, um welche Klasse es sich bei den Subobjekten handelt:

```
XMLPortObject(MyClass, SubObjectClass, „subsectionname“, \
                addSubobject, getSubobject, xmlelement, mode);
```

SubObjectClass steht hier für die Klasse, der die Subobjekte angehören. Die Subobjekte können theoretisch auch einer Unterklasse von SubObjectClass angehören, allerdings spielt dies für XMLPort keine weitere Rolle.

Wenn man anstelle von „subsectionname“ nur einen leeren String „“ übergibt, fällt die Subsection in XML komplett weg und die Subobjekte können direkt innerhalb des Basiselements hinzugefügt werden. Allerdings lässt dies keine Unterscheidung von mehreren Typen von Subobjekten mehr zu.

In diesem Beispiel würden die set- und get-Funktion wie folgt aussehen:

```
void addSubobject(SubObjectClass* object);

SubObjectClass* getSubobject(unsigned int index) const;
```

Der Index in der get-Funktion dient als Nummerierung der Objekte. Das erste Objekt hat die Nummer 0, das zweite die 1, etc. Beim Speichern eines Objekts wird das Framework wird so lange die get-Funktion mit steigendem Index aufrufen, bis ein NULL-Pointer zurückgegeben wird.

Subobjekte werden typischerweise in einer Liste oder einem ähnlichen Container gespeichert. Das Objekt das die Subobjekte erhält, ist auch für deren Zerstörung verantwortlich.

Lesenswert:

- http://www.orxonox.net/wiki/howto/XMLPort – Eine Kurzanleitung zu XMLPort
- http://www.orxonox.net/wiki/XMLPort – Ausführlichere Beschreibung der Features