Praktikum Design und Entwicklung eines Computerspiels

Netzwerk: Serialisierung von Aktionen und Entities
Version 0.9.2

Thomas Huth

29. Januar 2003

Zusammenfassung

Dieser Artikel entstand während des Praktikums "Design und Entwicklung eines Computerspiels" und beschreibt die Serialisierung von Entities/Aktionen für deren Transport übers Netzwerk bzw. für deren Abspeicherung in einer Datenbank etc.

1 Serialisierung

1.1 Einleitung

Entities und Aktionen in Neverending-ODC müssen komplett und auch teilweise über das Netzwerk übertragen werden können. Hierfür ist es erforderlich, die Entities- und Action-Klassen in eine ,,flache", d.h. serialisierte Struktur zu bringen, so dass die Netzwerkschicht sie leicht in einen Bytestrom zum Übertragen über das Netz umwandeln kann.
Aber nicht nur für die Netzwerkschicht ist so eine Serialisierung angebracht, auch zum Abspeichern der Entities, z.B. in einer Datenbank oder in eine XML-Datei, ist ein eine solche Datenrepräsentation der Objekte hilfreich. Deshalb soll in Neverending-ODC ein möglichst universell einsetzbares System zur Serialisierung auf Entity/Action-Ebene zum Einsatz kommen, erst die entsprechenden Komponenten (Netzwerk, XML-Dateispeicherklasse, etc.) machen dann aus diesem universellen Schema die tatsächlich benötigte Datendarstellung (Bytestrom, XML, etc.).

1.2 Vorgehensweise

Hauptidee des hier vorgestellten Serialisierungsansatzes ist es, dass jedes zu serialisierende Objekt zwei Methoden zur Verfügung stellt, die je einen Vektor mit den gewünschten Daten zurückliefern. Ein Vektoreintrag beinhaltet dann den Namen der entsprechenden Variablen, den Typ der Variablen und natürlich auch noch den Wert der Variablen.
Als Namen für die beiden Methoden werden im Folgenden **getCondensedDataVector** und **getAllCondensedDataVector** verwendet. Die eine Methode dient dabei nur dazu, einen Teil des Objektes zu serialisieren, die andere hat die Aufgabe, das komplette Objekt zu serialisieren. Die erste erwähnte Methode ist deshalb nötig, weil es nicht immer sinnvoll ist, ein komplettes Objekt zu serialisieren. Wenn man z.B. nur ein Positions-Update eines Entities verschicken will, ist es nicht nötig, auch noch die Rotation und weitere Entity-Eigenschaften mit zu verschicken.

1.2.1 Der Serialisierungsvektor

Um die Daten im Rückgabevektor der beiden Methoden sinnvoll weiterverarbeiten zu können, müssen mindestens folgende Informationen in einem Vektoreintrag enthalten sein:

- Name der entsprechenden Variablen
- Typ der entsprechenden Variablen
- Wert der entsprechenden Variablen

Als Typ der Variablen werden folgende Konstanten benutzt, die durch eine Aufzählung (enum) definiert sind:

```
/** This enum declares the constants for types of the serialization
    vector entries */
enum CondensedDataType
{
    INT8,                   /* 8 bit integer */
    INT16,                  /* 16 bit integer */
    INT32,                  /* 32 bit integer */
    FLOAT,                  /* single precission float */
    DOUBLE,                 /* double precission float */
    CSTRING,                /* a C string */
    VECTOR3D                /* a Vector3d object */
};
```

Der Vektor, der die Variablenwerte in der gewünschten Form enthält, wird nun wie folgt definiert:

```
#include <string>
#include <vector>
#include <odcTypes.h>

class CondensedData
{
    string m_varName;           /* The name of the variable */
    CondensedDataType m_type;
    union
    {
        OdcSInt8 m_int8;
        OdcSInt16 m_int16;
        OdcSInt32 m_int32;
        float m_float;
        double m_double;
        char* m_cstring;
        Vector3d* m_vector3d;
    } m_value;
};

typedef std::vector<CondensedData *> CondensedDataVector;
```

Man beachte hierbei, dass nicht die C/C++ Standarddatentypen ,,char", ,,short", ,,int" und ,,long" in *m_value* benutzt werden, sondern größenspezifische Datentypen, da die tatsächliche Bytegröße eines solchen Eintrags sehr wichtig für das Netzwerk ist, aber die C/C++ Standarddatentypen je nach Compiler bzw. Systemarchitektur unterschiedliche Bytegrößen haben.
,,Nicht-primitive" Datentypen wie z.B. *Arrays* oder *Structs*, müssen für die Serialisierung in primitive Datentypen umwandelbar sein. So kann z.B. ein *Arrays* für die Position a la ,,float xyz[3];" in seine drei einzelnen Koordinatenanteile aufgeteilt werden. Auf eine sinnvolle Namensgebung für *m_varName* ist hierbei besonders zu achten .

1.2.2 Die Zugriffsmethoden

Die Prototypen der Zugriffsmethoden sehen wie folgt aus:

```
CondensedDataVector
    getCondensedDataVector(std::vector<string *> f_StrVec);
```

liefert den zugehörigen Datenvektor zu den in *f_StrVec* angegebenen Variablennamen.

```
CondensedDataVector getAllCondensedDataVector(void);
```

liefert einen Vektor mit allen zu dem Objekt gehörigen Daten.
Die Realisierung dieser Methoden geschieht wie folgt: In einer obergeordneten Serialisierungsklasse (namens *DataCondenser*), von der alle Objekte erben müssen, welche die Serialisierungsfunktionen bieten sollen, wird eine ,,map" (bzw. ,,hash_map") definiert, die wie folgt aussieht:

```
#include <hashMap.h>

class CondensedDataRef
{
    CondensedDataType m_type;
    union                    /* The reference to the variable */
    {
        OdcSInt8* m_int8;
        OdcSInt16* m_int16;
        OdcSInt32* m_int32;
        float* m_float;
        double* m_double;
        char** m_cstring;
        Vector3d* m_vector3d;
    } m_ref;
};

MAKE_MAP_CSTR(CondensedDataRef*, CondensedDataRefMap);
```

Im Konstruktor eines serialisierbaren Entities- bzw. Action-Objektes wird dann für jede betroffene Variable ein *CondensedDataRef* erzeugt, *m_type* auf den entsprechenden Typ gesetzt und den Pointer *m_ref* lässt man auf die entsprechende Variable zeigen. Das fertig initialisierte *CondensedDataRef* fügt man dann mit dem Variablennamen als Schlüssel in die *CondensedDataRefMap* des Objektes ein.

Die Zugriffsmethoden *getCondensedDataVector()* und *getAllCondensedDataVector()* können dann einfach und überall auf die gleiche Weise über diese ,,map" die entsprechenden Variablenwerte erreichen, so dass es im Normalfall reicht, diese beiden Zugriffsfunktione in der Eltern-Serialisierungsklasse bereit zu stellen und in den implementierenden Klassen zu benutzen. Der Zugriff funktioniert dann bei allen Erb-Klassen genau gleich über die ,,map". Die Methode *getCondensedDataVector()* kann dabei direkt über die Variablennamen-Strings auf die ,,map" zugreifen und die Methode *getAllCondensedDataVector()* kann die ,,map" einfach mit einem ,,iterator" durchlaufen.
Falls die implementierende Entity- bzw. Aktionsklasse auch die Daten ihres Elternobjektes mitliefern möchte, muss darauf geachtet werden, dass beim Initialisieren der *CondensedDataRefMap* die Daten der Eltern-Klasse(n) erhalten bleiben. Damit es beim Vererben keine Namenskonflike gibt, ist bei der Bildung der Vererbungshierarchie darauf zu achten, dass die zu serialisierenden Klassenvariablennamen eindeutig sind!

1.2.3 Die Methoden zum Setzen

Analog zu den beiden beschriebenen Methoden zum Abfragen von Objektvariablen muss es natürlich auch ein Gegenteil dazu geben, eine Methode, welche die Werte eines Objektes aus einem *CondensedDataVector* ausliest und im Objekt die entsprechenden Variablen auf diese Werte setzt.
Der Prototyp dieser Methode sieht wie folgt aus:

```
int setCondensedData(CondensedDataVector& f_dataVec);
```

Diese Methode kann analog zu *getCondensedDataVector()* auch über die *CondensedDataRefMap* des Objektes leicht realisiert werden.
Falls eine Klasse jedoch darauf angewiesen ist, Änderungen an einem Variablenwert mitzubekommen, muss die Methode *setCondensedData()* in der Klasse überschrieben werden, um den übergebenen Datenvektor auf entsprechende Variablen hin zu überprüfen. Natürlich sollte in diesem Fall in der Methode auch unbedingt die *setCondensedData()* der Elternklasse aufgerufen werden, um den Rest des Datenvektors zu verarbeiten.

1.3 Ausblicke

Der unschönste Teil an diesem Konzept ist bisher die Tatsache, dass die *CondensedDataRefMap* im Konstruktor eines Objektes mühsam ,,von Hand" erstellt werden muss. Ändert man eine Klassenvariable, muss auch die ,,map" im Konstruktor von Hand geändert werden.
Sinnvoll wäre es, hier einen Präprozessor (bzw. eine bestimmte Make-Regel und ein geeignetes Tool wie z.B. Flex?) einzusetzen, der aus den Klassenvariablen (welche z.B. durch einen bestimmten Ausdruck markiert sind) diese *CondensedDataRefMap* im Konstruktor automatisch erstellt.