Martin Burckhardt

Ein kurzes Dll-Turorial, dessen Ziel es ist, sich vom WDL-Skripting zu verabschieden

Eine sehr kurze Einführung

WDL (oder C-Skript) ist eine schöne und leicht verständliche Sprache.

Aber auch C++, sofern man es denn einmal verstanden hat, ist eine überaus schöne und elegante Sprache. Und wenn man sich einmal auf C++ eingelassen hat, ist der Weg zum Gamestudio SDK nicht mehr weit, ist es einem doch hier vergönnt, die Vorzüge einer objektorientierten Programmiersprache auszukosten.

Infolgedessen sollte es einfach sein, mit diesen Werkzeugen eine *dll zu schreiben. Aber dies ist bedauerlicherweise nicht der Fall. Fragt sich nur warum?

Der Grund dafür, daß sich hier Komplikationen einstellen, denen man in C++ nicht begegnet, hat damit zu tun, daß man sich beständig über das Interface im klaren sein muß. Man stelle sich vor, daß ein Sprechakt einen beständigen Übersetzungsvorgang beinhaltetete, daß man also vom Deutschen ins Englische und wieder zurück übersetzen müßte, etwa so

Deutsch -> Englisch -> Deutsch

Gewiß: als Diagramm sieht dies nicht sonderlich kompliziert aus, aber wenn man bedenkt, daß es Wörter gibt, wie sich nicht übersetzen lassen (wie etwa »Weltangst«), verwandelt sich dies in ein mühseliges Unterfangen - und wird mühseligen noch, wenn man sich vergegenwärtigt, daß bei einer derartigen Simultanübersetzung jeder Satz nicht nur einmal, sondern gleich dreimal formuliert werden muß.

Die beste Art und Weise, sich dem Schreiben einer DLL anzunähern, besteht darin, sich vom WDL-Skripten zu verabschieden (in dem Maße, in dem dies möglich ist). So muß man es halten wie Cortes, der Conquistador, der, um das Land der Azteken erobern zu können, sich zunächst daran machte, seine Schiffe zu verbrennen.

I. Wie greife ich auf eine DLL zu?

Was ist eine DLL? Eine DLL ist im wesentlichen ein angehängtes Programm (eine Bibliothek von Befehlen), und infolgedessen ist es nötig, daß sich im Hauptprogramm, das auf die DLL verweist, ein entsprechender Verweis findet.
Dies muß in WDL geschehen. Zunächst bedarf es eines Handles - einer Variable vom no-save Type (definiert man eine ganz gewöhnliche Variable, so wird diese beim Speichern des Spiels mitabgespeichert, was beim erneuten Laden zu einem Fehlverhalten der DLL führen kann):

```
var_nsave  my_dll;
```

Als nächstes gilt es die DLL zu öffnen und ihr den zugedachten Handler zuzuweisen. Dies geschieht sinnvollerweise in der Funktion, die zu Anfang des Spiels aufgerufen wird, also in „main“:

```
my_dll  =  dll_open(“MyDll.dll”);
```

Wenn Sie meine einzige DLL benutzen, ist dieser Prozeß überaus simpel. Sie öffnen die DLL zu Beginn der Sitzung und halten sie, um auf die Befehle der Bibliothek zugreifen zu können, sinnvollerweise offen.

Wenn Sie allerdings verschiedene DLLs benutzen, so müssen Sie Sorge tragen, daß sie jeweils in der ent-

sprechenden DLL (metaphorisch: im jeder passenden Wörterbuch) nachschlagen, und diese ist eine etwas schwierigere Aufgabe.

“Nun,” mögen Sie sagen (ganz im Sinne eines alten Beatles Songs); “I got a driver, but I ain’t got no car.” Und sollten Sie Ihr Gamestudio Projekt öffnen, so wird Ihnen dies die entsprechende Antwort gegeben: daß nämlich die DLL nicht existiert.

2. Wie schreibe ich eine DLL

Setzen wir uns also daran und konstruieren ein Auto. Der große Vorteil: wir wissen, unter welchen Namen es firmiert: MyDll.dll.

Selbstverständlich sollte das Autor seinen Fahrer aufnehmen können - was nichts anderes bedeutet, als daß wir den dazugehörigen DLL-Type wählen müssen.

Wenn Sie (was sich inständig hoffe) mit Visual C++ arbeiten, werden Sie sehen, daß es zwei DLL-Typen gibt:

- eine DLL, die es mit Hilfe der MFC, also der Microsoft Foundation Classes, erzeugt wird (was in diesem Falle nicht hilfreich ist, da sich diese Klassenbibliothek nicht sonderlich gut mit DirectX verträgt)
- der zweite Typus ist nüchtern “win32dynamicLinkLibrary” betitelt.

Genau diesen Typus benötigen wir.

Starten Sie also ein neues Visual C++ DLL Projekt und nennen es „MyDll“, kopieren Sie anschließend alle SDK Dateien in das entsprechende Verzeichnis, binden Sie die *cpp in das Projekt ein, und zuguterletzt kompilieren und erzeugen Sie schließlich die DLL. Sie werden nun die Datei“MyDll.dll” in ihrem *debug* Verzeichnis finden.

Das einzige, was es noch zu tun gilt, ist, diese frisch erzeugte DLL in ihren Gamestudio-Folder zu kopieren.

Jetzt starten Sie das Spiel, und was passiert? Gamestudio beklagt sich nicht mehr darüber, daß es die DLL nicht finden vermag. Das ist wunderbar - endlich Ruhe.

3. Wie bewege ich mich von hier nach da?

In der Tat, das Leben ist kompliziert, und so wird auch das folgende eher ein bißchen bizarr und doppelt gemoppelt ausschauen, als hätte man ständig eine Gebärdenüersetzer neben sich.

Das Ziel, wir erinnern uns, ist es, den Code, den Sie in WDL schreiben müssen, auf ein Minimum zu reduzieren. Aber was wäre das Minimum?

Nun werden Sie schnell begreifen, daß das Minimum im Grunde nicht weniger ist als die komplette WDL Syntax. Wie der Transfer dieser Syntax bewerkstelligt werden kann, werde ich im folgenden zeigen, aber zunächst einmal möchte ich mich darauf konzentrieren, den Mechanismus der WDL-DLL Interaktion zu beschreiben.

Da Conitec so nett war, dem SDK einige Beispielfunktionen hinzuzufügen, haben Sie schon einige Befehle zu Ihrer Verfügung. Bevor Sie allerdings eine DLL Funktion in ihrem Gamestudio Projekt benutzen können,

müssen Sie sie deklarieren (das heißt: Sie machen sie der Engine bekannt).
Um dies zu tun, benutzen Sie den dllfunction Ausdruck:

```
dllfunction PaintEntitiesRed(void_var);
```

Haben Sie dies einmal getan, so können Sie die DLL Funktion in WDL benutzen, so als ob sie ein Teil der Gamestudio-Syntax wäre:

```
PaintEntitiesRed(0);
```

Was aber passiert, wenn WDL diesen Befehl ausführt? Ganz einfach:

1. die Engine greift auf die DLL zu
2. und sucht hier nach der Funktion "PaintEntitiesRed(0)"

Nehmen wir einmal an, daß Sie eine gewöhnliche C++-Funktion dieses Namens definiert hätten (beispielsweise BOOL PaintEntitiesRed(void)), so würde die Gamestudio Engine sie nicht finden.
Aber warum nicht?
Der Grund dafür ist, daß die Engine die DLL nur nach Funktionen des Typus DLLFUNC durchsucht.
Und so werden nur diejenigen Funktionen, die nach dem folgenden Schema gebaut sind, tatsächlich funktionieren:

```
DLLFUNC fixed PaintEntitiesRed(void)
```

Der Rest bleibt im Dunkeln.
DLLFUNC ist das Schlüsselwort, und die ganze Kommunikation zwischen WDL und DLL läuft über diesen Kanal.

Aber was ist DLLFUNC?

Es ist keinesfalls ein Teil der Programmiersprache C++; es stellt vielmehr das Interface, genauer: den Datentyp dar, den Johannes C. L sich für des Interface ausgedacht hat, mit dem dazugehörigen Datentyp fixed.

fixed ist 32 bit und äquivalent zu long.

Lassen Sie uns nun ein bißchen weiter gehen. Nehmen wir einmal an, wir hätten eine sauber und ordentlich geschriebene DLL. Wären wir dann autonom?
Keinesfalls - denn wir werden immer auf einige WDL Kommandos zurückgreifen müssen.

Für ein Gamestudio Spiel brauchen d actions, und C++ selbst kennt keinen Typ dieser Art; tatsächlich weiß C++ überhaupt nichts von der Gamestudio Engine. Hier eine Liste all der Gamestudio-Datentypen, die in C++ unbekannt sind:

```
A4_ENTITY;
A4_STRING;
A4_TEX;
A4_BITMAP;
A4_TEXT;
A4_STRING;
...
```

schauen Sie einmal in die Datei a5dll.h - dort finden sie alles, was eine DLL benötigt, um mit der Engine von Gamestudio in der agieren zu können.

4. Sorry, aber ich bin noch nicht da. Suche noch immer nach einem Fixpunkt, suche einen Pointer

Vielleicht bin ich hartnäckig: ich hätte gerne all die Dinge in meiner DLL zu Verfügung, die in WDL so leicht zu handhaben sind. In anderen Worten: ich möchte WD Funktionen und Entities ebenso einfach handhaben wie ich von WDL aus Dll-Funktionen aufrufen kann. Wir erinnern uns, das ging folgendermaßen:

```
dllfunction PaintEntitiesRed(0);
```

Es gibt nun eine ganz einfache Generallösung: schreiben Sie alle WDL Funktionen auf und importieren das ganzes Set in die DLL.

Sie sollten dies beim Start Ihres Programmes tun, am Ende der main() Funktion.
Das sehr dann folgendermaßen aus.

Der WDL Teil:

```
dllfunction function_import();
```

der DLL Teil:

```
DLLFUNC fixed function_import(void)
```

Nehmen wir an, wir wollen den Player in der Dll benutzen.

Das erste, was er dabei wissen müssen, ist, das wir ein WDL Objekts in der DLL niemals als solches benutzen können; wir haben lediglich Zugriff auf seine Adresse (was in der C++-Syntax ein *Pointer* genannt wird).

Folglich müssen wir einen solchen Pointer erzeugen und ihm die Adresse des dazugehörigen WDL Objekts mitteilen.

Hier sieht man, wie man den Player-Pointer in der DLL deklariert
Aus Einfachheitsgründen erhält der Pointer der DLL den gleichen Namen wie das Objekt in WDL: player.

```
A4_ENTITY *player;
```

Damit ist der Pointer erzeugt, und wir können ihm nun die Adresse des »wirklichen« Player Objekts mitteilen:

```
DLLFUNC fixed function_import(void)
{
player        = (A4_ENTITY *)a5DLL_getwdlobj("player");       //
}
```

Das großartige ist, das ein Pointer keinesfalls eine untergeordnete und weniger funktionstüchtigen Entity ist, sondern daß all die Attribute erbt, die auch das Objekts selbst besitzt.

Wenn Sie das Pointer-Zeichen eintippen (->):

player->

zeigt Ihnen der Compiler all die verfügbaren Attribute, wie alpha, red, green, blue etc.

Diese Operation müssen wir mit all den WDL Objekts vollziehen, wie wir benutzen wollen (und idealerweise mit der gesamten WDL Syntax). So wäre es angebracht, all die nötigen Pointer zu erzeugen und am Ende der main die dazugehörigen Objekte zu importieren.
Dies muß nur ein einziges Mal geschehen.
Dafür gibt es zwei Gründe:

1. wiederholt man diesen Prozeß, so geht dies auf die Rechenzeit. „a5dll_getwdlobj" ist eine sehr rechenintensive Funktion.
2. Wenn Sie wirklich eine Applikation bauen wollen, die komplex wie eine DLL gesteuert wird, sollten Sie die Syntax in einem Schwung einladen. Damit lassen sich die Schwierigkeiten der bidirektionalen Kommunikation und die Probleme, die daraus entstehen kommen, reduzieren.

Jetzt können wir unsere Importfunktionen und Deklarationen ein wenig erweitern.

Unsere Deklarationen könnten jetzt folgendermaßen aussehen:

```
A4_ENTITY           *player;
A4_ENTITY           *YOU;
A4_TEXT             *msg;
A4_STRING           *empty_str;
fixed               *time;              // this is how you declare a variable

wdlfunc1            ang;         // a function with 1 argument
wdlfunc2            vec_set;            //                  2 arguments
wdlfunc3            vec_diff;           //                  3 arguments
```

(Sie werden bemerken, das alles außer den Funktionen Pointer sind. Tatsächlich sind auch die Funktionen Pointer, nur daß sie nicht als solche sichtbar sind).

Unsere Import-Funktion könnte nun folgendermaßen aussehen:

```
DLLFUNC  fixed  function_import(void)
{
player        =  (A4_ENTITY  *)a5dll_getwdlobj("player");
YOU           =  (A4_ENTITY  *)a5dll_getwdlobj("you");
msg           =  (A4_TEXT  *)a5dll_getwdlobj("msg");
empty_str     =  (A4_STRING  *)a5dll_getwdlobj("empty_str");
game_panel    =  (A4_PANEL  *)a5Ddll_getwdlobj("game_panel");
wdltime       =  (fixed  *)a5DLL_getwdlobj("time");
vec_set       =  (wdlfunc2)a5DLL_getwdlfunc("vec_set");
}
```

5. Die DLL übernimmt das Kommando - aber wie?

Nehmen wir einmal an, daß Sie die komplette WDL Syntax eingeladen haben - oder daß Sie zumindest all das importiert haben, was Sie benötigen.

Das erste, was Sie nun ausprobieren wollen, ist, wie man eine Entity in der DLL manipuliert, sie rotieren läßt, ihrer Alpha-Wert verändert usw.

Nehmen wir an, wir haben eine WDL Aktion, etwa:

```
action my_dummy
{
while(1)
     {
     //hier folgt der DLL Aufruf
     wait(1);
     }
}
```

Dies ist der Minimalrahmen (der Kanal), der unerläßlich er ist, wenn wir eine DLL das Kommando übernehmen lassen wollen .

Zunächst einmal deklarieren wir eine neue DLL Funktionen, die unser Objekts anspricht - sagen wir:

```
dllfunction call_my_dummy(ent);
```

In der DLL fügen wir hinzu:

```
DLLFUNC fixed call_my_dummy(long entity)
{
///////jetzt folgt die Aktion
return 0;
}
```

Des weiteren brauchen wir einen Pointer auf unser Objekt, also:

```
A4_ENTITY *dummy;
```

Jetzt können wir die DLL ansprechen, wenn wir die Schleife etwas modifizieren:

```
while(1)
     {
     call_my_dummy(my);
     wait(1);
     }
```

Lassen wir uns analysieren, was passiert, wenn die Schleife den Befehle call_my_dummy(my) ausführt.

Das Objekts wird zunächst der DLL übermittelt. Hier wird es von der Funktion entgegengenommen, freilich nicht das Objekts selbst, sondern abermals nur seine Adresse, der Pointer

Infolgedessen ist eine Konversion nötig, welche den Datentyp long in einen A4_ENTITY pointer zurückverwandelt.

Dies wird durch den folgenden Code bewerkstelligt:

```
DLLFUNC fixed call_my_dummy(long entity)
{
dummy = (A4_ENTITY *)entity;
return 0;
}
```

Wunderbar. Das klappt.
Lassen Sie uns rekapitulieren, was wir bis jetzt erreicht haben.

Wir haben eine WDL Aktion, die eine DLL Funktionen anspricht, die wiederum die Adresse unseres Objektes erhält. Nun können wir unser Objekt über die DLL manipulieren.
Zum Beispiel:

```
DLLFUNC fixed call_my_dummy  (long entity)
{
dummy = (A4_ENTITY *)entity;
return dummy->pan += FLOAT2FIX(0.1);
}
```

In diesem Beispiel würde unser Objekt beginnen, sich zu drehen.
Lassen Sie uns einen Blick auf die hinzugefügte Zeile werfen. Warum benutzen wir diesen merkwürdigen Ausdruck, FLOAT2FIX(0.1)? Warum könnten wir nicht den folgenden Code benutzen?

```
return dummy->pan += 0.1;
```

Der Grund dafür ist, das WDL allein den Datentyp fixed akzeptiert. Wenn wir also Werte zurück zur Engine schicken wollen, müssen wir unsere Datentypen (wenn sie denn eine andere Form haben) zu fixed wandeln.
Es gibt eine Reihe von Makros, die dies für uns erledigen, wie:

```
INT2FIX(1);          //konvertiert einen integer-Wert zu fixed
FLOAT2FIX(1);        //konvertiert einen float-Wert zu fixed
```

Man kann auch den anderen Weg nehmen:

```
FIX2INT(1);
FIX2FLOAT(1);
```

Wenn Sie vorhaben, die DLL als Hauptplatfform ihrer Entwicklung zu benutzen, werden Sie sich wohl oder übel daran gewöhnen müssen, diese Konversionen vorzunehmen. Zunächst erscheint dies als ein überaus mühsamer Prozeß, der ihrer Arbeit verdreifacht: eine simple Variable in WDL muß in den gewünschten Datentyp der DLL konvertiert werden und anschließend wieder zurück in einen fixed-Wert.

Dem man dies als Graphen, so sieht der Prozeß folgendermaßen aus:

WDL -> DLL -> WDL

(was unsere anfängliche Deutsch-> English ->Deutsch Analogie erklärt.)

Sie könnten sich fragen, worin denn, wenn dieser Prozeß so überaus der mühevoll ist, der Vorteil gegenüber einer WDL-Applikation besteht. Aber diese Arbeit zahlt sich letztlich doch aus: denn Sie können nun endlich Klassen benutzen, Sie werden sich niemals mehr darüber beschweren, nur 48 Skills benutzen zu können, zuguterletzt: Sie arbeiten in einer objektorientierten Umgebung.

6. Eine Klasse bauen

Nehmen wir einmal an, daß wir etwas sehr viel komplexeres als bloß das Dummy-Objekt einer Entity konstruieren wollen, nämlich eine Klasse (den Grundbaustein der objektorientierten Programmierung). Dies sähe dann etwa so aus:

```
class smart_dummy
{
public:
int MillionSkills[1000][1000];


A4_ENTITY *ent;

A4_TEXT   *text;
A4_PANEL  *panel;

} SmartGuy;
```

Anstatt einen Club ein Pointer zu erzeugen, haben wir unsere Beispiel-Entity in der Klasse verkapselt. Des weiteren haben einige andere Pointer hinzugefügt. Um nun zu zeigen, wie einfach es jetzt ist, dies Skill-Grenze zu überschreiten, verfügt unsere Entity nun über eine zweidimensionale Liste, die sich mit allerlei Werten füllen läßt.

Wir können unseren Smart Dummy jetzt so einfach aktivieren, wie wir zuvor unseren globalen Dummy-Pointer angesprochen haben:

```
DLLFUNC fixed call_my_dummy(long entity)
{
SmartGuy.ent = (A4_ENTITY *)entity;
return SmartGuy.ent->pan += FLOAT2FIX(0.1);
}
```

Ich hoffe, Sie sehen, wie einfach dies letztlich ist. Wir könnten nun eine main Funktion zu unserer Klasse hinzufügen:

```
class smart_dummy
{
public:

int MillionSkills[1000][1000];


main();       / /call to the smart_dummy main() function

A4_ENTITY *ent;

A4_TEXT   *text;
A4_PANEL  *panel;

} SmartGuy;
```

```
smart_dummy::main()
{
ent->pan += FLOAT2FIX(0.1);   //jetzt wird die Rotation automatisch erzeugt
}
```

Wir haben nun mit Erfolg ein Kommunikationssystem zu unserer DLL aufgebaut. Jedesmal, wenn die WDL Aktion die Schleife durchläuft, ruft sie die DLL auf und wird von da zur Klasse SmartGuy und zur DLL_Funktion main () weitergeleitet.

Jetzt müssen Sie sich nicht mehr um WDL Funktionen bekümmern.

7. WDL-Funktionen benutzen

Nehmen wir an, Sie wollen eine WDL Funktion in ihrer DLL aufrufen. Dies ist eine etwas haarige Angelegenheit, denn der Code sieht überaus kompliziert aus. Wir setzen voraus, daß Sie die entsprechende Funktion (wie zuvor besprochen) deklariert und importiert haben

Lassen Sie uns die nötigen Schritte durchgehen.
Zunächst definieren wir einen Vektor, der die player-Position aufnehmen soll:

```
class  smart_dummy
{
...

public:
..
///das fügen wir jetzt hinzu
fixed  temp[3];

execute_wdl();
...
}
```

```
smart_dummy::execute_wdl()
{
temp[0] = INT2FIX(1000);    //wir müssen mit dem fixed Datentyp arbeiten
temp[1] = INT2FIX(200);
temp[2] = FLOAT2FIX(0.2);

(*vec_set)((long)&(ent->x),(long)temp);
}
```

Die ersten Zeilen des Codes sind selbsterklärend. Wenn wir eine WDL Funktion benutzen wollen, müssen wir mit dem fixed Datentyp arbeiten - folglich ist eine Konversion notwendig.

Gleichwohl, die letzte Zeile macht einen etwas konfusen Eindruck. Lassen Sie uns also einen etwas genaueren Blick darauf werfen.

Warum heißt es (*vec_set) ?

Die Klammern diesen Ausdruck unversehens ihn mit einem *, dem Pointer Zeichen, weil wir keinen unmittelbaren Zugriff auf diese Funktion haben. Die uns zu Verfügung stehende Version von "vec_set" ist lediglich ein Pointer, und da wir der DLL nicht den Wert ihrer Adresse, sondern die Adresse selbst übermitteln wollen, müssen wir den Pointer zunächst dereferenzieren. Dies geschieht in der ersten Klammer.

Wenn wir die Funktion, was im Falle von vec_set vorgegeben ist, mit einem Parameter benutzen wollen, müssen wir diesen in den Datentyp long konvertieren.

Werfen Sie doch einmal einen Blick auf die Deklaration im a5dll header:

```
typedef fixed (*wdlfunc1)(long);
typedef fixed (*wdlfunc2)(long,long);
typedef fixed (*wdlfunc3)(long,long,long);
typedef fixed (*wdlfunc4)(long,long,long,long);
```

Da sehen wir die Prototypen, auf die wir unsere importierten WDL Funktionen einrichten müssen. Umgebung zu genügen, müssen wir unsere ersten fixed Vektor zu long konvertieren, etwa so:

```
(long) temp;
```

Im Falle des ersten Parameters greifen wir nicht auf swn fixed vector[3] zu, sondern lediglich auf die Adresse der betreffenden Entity, also vector[0]. Deshalb schreiben wir:

```
(long)&(ent->x)
```

Dies könnte man übersetzen als: nimm die Adresse und lies sie als einen long-Datentyp ein.
Und zusammenzufassen:

All dieses Hin- und Her-Konvertieren mag eine mühselige Angelegenheit sein, aber vergessenen Sie nicht, daß wir dabei mit den Vorzügen der objektorientierten Programmierung belohnt werden. Sie können ihre action zu jeder gewünschten Komplexität ausarbeiten, und das lohnt die Mühe.

8. Event handling

Die Architektur der DLL wäre unvollständig, wenn wir des Event Handling nicht einbeziehen würden. Und dies zu bewerkstelligen, müssen wir der kurzen Augenblick von unserer DLL abwenden und ein paar Zeilen zu unserer bisherigen WDL Code hinzufügen.

Zunächst gilt es eine DLL Funktion zu deklarieren, die das Interface zum DLL Event-Handling darstellen soll:

```
dllfunction call_my_dummyevent(eventNr);
```

Dann müssen Sie eine Funktion schreiben, die die DLL aufruft:

```
function dummy_event()
{
call_my_dummyevent(event_type, my); // dies triggert die DLL
}
```

```
action my_dummy
{
my.enable_click = ON;
                    //fügen Sie hier all die Eventtypen hinzu,
                    // die Sie benutzen wollen
my.enable_shoot = ON;

my.event = dummy_event; // dies triggered das Event

while(1)
     {
     call_my_dummy(my);
     wait(1);
     }
}
```

Nun, der Code dehnt sich aus. Unsere Action ruft eine WDL Event-Funktion auf, die ihrerseits die DLL Eventfunktion aufruft.

Sie könnten sich über die Variable "event_type" verwundern, die als Argument übergeben wird. Das hat mich (zugegebenermaßen) selbst verwundet - bis ich verstand, daß event_type lediglich das Synonym für eine einfache Zahl ist.

Wenn der Event-Type beispielsweise ein Click ist, wird der DLL nicht die Zeichenkette "click", sondern lediglich eine Zahl übermittelt - in diesem Falle die 16.
Jeder Eventtyp hat eine eindeutige Kennzahl. Wir werden sie uns gleich anschauen.

Wir haben jetzt in WDL eine DLL Funktion deklariert, freilich existiert diese noch nicht in der DLL. Dieses ist also das nächste, was wir erledigen müssen.

```
DLLFUNC fixed call_my_dummyevent(fixed event_type, long entity)
{
//jetzt sollten wir unser SmartDummy triggern
return 0;
}
```

Diese Funktion etabliert unserer Kommunikationskanal. Jedesmal, wenn ein Event sich ereignet, wird die DLL angesprochen. Bis jetzt allerdings hat sich nur nichts ereignet; wir müssen der Funktion bekanntgeben, um welche Entity es sich handelt und darüberhinaus auch der SmartDummy-Klasse eine event_function hinzufügen.
Machen wir uns also an die Arbeit:

```
class smart_dummy
{
...

public:
```

```
..
//dies wird hinzugefügt

call_event(int number);   // “event type” ist lediglich eine Zahl
...
}


smart_dummy::call_event(int number)
{
if (event == 5)                    // Impact
 {
 }

if (event == 9)                    // Shoot
 {
 }

if (event == 16)                   // Touch
 {
 }

if (event == 18)                   // Click
 {
 }

if (event == 19)                   // Right-click
 {
 }

if (event == 8)                    // Scan
 {
 }

if (event == 1)                    // Block
 {
 }

if (event == 3)                    // Stuck
 {
 }

if (event == 2)                    // Entity

 }

if (event == 4)                    // Push
 {
 }

if (event == 17)                   // Release
 {
 }

if (event == 7)                    // Detect
 {
 }

if (event == 10)                   // Trigger
 {
 }
```

```
if (event == 11)                    // Sonar
 {
 }


if (event == 24)                    // Disconnect
 {
 }

}
```

Zugegebenermaßen, es wäre sehr viel eleganter gewesen einen Switch zu schreiben, aber steht Ihnen frei, den Code ein wenig zu verbessern. Das allerwichtigste ist, daß Sie mit den entsprechenden Kennziffern auf die richtigen Events zugreifen können.

Bleibt's nun nicht vielmehr zu tun, als die Interface-Funktion ein wenig zu modifizieren:

```
DLLFUNC fixed call_my_dummyevent(fixed event_type, long entity)
{
SmartGuy.ent = (A4_ENTITY *)entity;
int no = FIX2INT(event_type);           // zu int konvertieren
SmartGuy.call_event(no);
return 0;
}
```

Nun - hier wären wir also: wir haben uns erfolgreich von WDL gelöst und können uns nun ganz und gar der C++-Programmierung widmen.

Ps. Zum Zweck der Übersichtlichkeit habe ich versucht, den Code auf das absolute Minimum zu reduzieren. Wenn Sie sich über fehlende Konstruktoren und Destruktoren verwundern, machen Sie sich keine Gedanken. Sie sind nur deswegen ausgelassen worden, um den Code transparent zu halten.