

Martin Burckhardt

A very short dll tutorial aiming to undock from wdl coding

Very Short Introduction

WDL (or C-Script), like C, is a nice and easy language.

But C++, once you understand it, is a very, *very* nice language. And once you *have* grasped C++, the GameStudio SDK is an incredibly powerful tool.

So it *should* be easy to write a *DLL. But it is not. Why not?

The reason that it seems so complicated is that you have to think about an interface all the time. Imagine how it would be if speaking involved a continuous process of simultaneous translation and retranslation, like this:

German -> English -> German

It looks easy in diagrammatical form, but when you consider words that cannot be translated (such as “Weltangst”), it turns out to be a real pain – and it’s even worse when you think that all of your work must be done more than once: in fact, all of your work is in effect *tripled*.

The best way to approach DLL writing is to say goodbye to WDL altogether (in so far as this is possible). So you have to act like Cortez, the conquistador who decided to burn his ships when he landed in the land of the Aztecs.

1. How do I access a *DLL?

What is a DLL? A DLL is essentially a supplementary program, and there therefore needs to be a reference in your main program (GameStudio) that points to it.

This must be done in WDL. First you need a handle – a variable that should be of the no-save type (otherwise the save operation would cause complications once the file is loaded again):

```
var_nsave my_dll;
```

Next you have to open the DLL and assign it to the handle (which I would do in the main function call):

```
my_dll = dll_open("MyDll.dll");
```

If you have just the one DLL running, this is straightforward. You open the DLL at the beginning of the session and keep it open.

But if you are using various DLLs, you have to take care that you refer to the correct DLL (by using different handles), and this is a more delicate task.

“Fine,” you may say; “I got a driver, but I ain’t got no car.” (And if you try to execute your GameStudio project, it will give you much the same answer.)

2. How do I write a DLL?

Okay, so let’s make a car. Big advantage: I know its name: `MyDll.dll`.

The car should fit the driver – which is to say that we should choose the appropriate type of DLL.

If you have Visual C++ (and I hope you have) there are two types of *DLL:

- the first type is created with the help of MFC (which is, I am afraid to say, no friend)
- the second type is soberly entitled “win32dynamicLinkLibrary”

It is the second type that you need.

Start a new VC++ DLL project and call it “MyDll”, copy all of the SDK files into the folder that is created, integrate the *.cpp files, and finally compile and build the DLL. You will now find “MyDll.dll” in your debug folder.

The only thing left to do is to copy your new DLL into the GameStudio project folder.

Now run your game and what happens? Gamestudio ceases to complain that it cannot find your DLL. That’s nice - silence.

3. How to move from here to there

Yes, this is complicated, and at first much of what follows will seem a confusing mix of skewed logic and DoubleSpeak.

My aim is to reduce the amount of work you need to do on the WDL side of coding to a minimum. But what is the minimum?

In fact you soon realise that the minimum is nothing less than the complete WDL syntax. I will show how this might work below, but first I want to concentrate on understanding the mechanics of WDL-DLL interaction.

Since Conitec has been nice enough to include some example functions, you already have a few commands at your disposal.

Before you can use a DLL function in WDL you have to declare it. To do this, you use the dllfunction declaration:

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

Once you have done this, you can use it in WDL, just like a normal function:

```
PaintEntitiesRed(0);
```

This is what happens when your WDL file executes this command:

1. it accesses the DLL
2. it looks for the function “PaintEntitiesRed(0)”

But let’s say you have a regular C++ function with that name (eg. BOOL PaintEntitiesRed(void)); the Gamestudio engine will not find it.

Why not?

The reason is that the engine only looks for functions of the DLLFUNC type. So only functions that look like this:

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

will work. Anything else will not.

DLLFUNC is the keyword, then, and all communication between WDL and a DLL must use this channel.

But what is `DLLFUNC`?

It is by no means part of C++; it is the interface data type that Johann C. Lotter has defined for us, along with the data type `fixed`.

`fixed` is 32 bit and equivalent to `long`.

Let's go a little deeper though. Let us assume that we have a neat and properly written DLL. Can we be autonomous?

No - we will always need some WDL commands as well.

For a GameStudio game we need `actions`, and C++ itself does not know of any type such as "`action`"; in fact, C++ does not know anything about Gamestudio's engine at all. Here is a sample list of GameStudio types unknown to C++:

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

Take a look at `a5dll.h` - there you will find everything that C++ needs to know in order to interact with GameStudio.

4. Sorry, I am not there yet. Still looking for a point of departure, looking for a pointer

I am stubborn: I want to have access to all the things I already have in WDL that are so easy to handle. In other words, I want to import WDL functions and entities into my DLL as easily as I can import DLL functions into WDL - I can do the latter as easily as this:

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

There is a catch-all solution for this: just write down all the commands and import the whole set into the DLL.

You should do this at the start of your program, at the end of the `main()` function. It would look like this:

the WDL part:

```
dllfunction function_import();
```

the DLL part:

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

Let's assume we want to use the player object in the DLL.

The first thing to note is that we can never, *ever* use a WDL object in a DLL as such - we must always use its address (which is called a pointer in C++ syntax).

Thus we have to create such a pointer and then assign it the address of the WDL object.

Here's how we declare a player pointer in the DLL.

For the sake of simplicity, I will give the DLL pointer the same name as the one it is given in WDL: player.

```
A4_ENTITY *player;
```

This creates the pointer, which can now receive the address of the “real” WDL player-object:

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

The great thing is that this pointer is by no means a minor entity - it inherits all the attributes we know from WDL syntax.

If you type in the pointer sign (->):

```
player->
```

your compiler will show all the attributes available, like **alpha**, **red**, **green**, **blue** etc.

This operation has to be done with all the WDL objects we want to use (and ideally the whole WDL syntax).

So it would be advisable to create all the necessary pointers and then import them at the end of the main file.

This only needs to be done once.

There are two reasons for this:

1. refreshing the import process will cause slowdown.
“a5dll_getwdlobj” is a very time intensive function.
2. if you really want to build an application where everything is done via a DLL, you should transfer the syntax all in one go. This helps to avoid later bi-directional communication and any problems that may arise from this.

Now we can extend our import function and declarations a little bit.

For instance, our declarations might look like this:

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

wldfunc1       ang;                  // a function with 1 argument
wldfunc2       vec_set;              //                2 arguments
wldfunc3       vec_diff;             //                3 arguments
```

(You will notice that everything except for the functions is a pointer.)

And our import function might look like this:

```
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 *)a5dll_getwdlobj("game_panel");
wldtime     = (fixed *)a5DLL_getwdlobj("time");
vec_set     = (wldfunc2)a5DLL_getwldfunc("vec_set");
}
```

5. The DLL takes command - but how?

Let's assume that you have copied the whole WDL syntax - or at that you have at least imported everything you need.

The first thing you will want to know is how to manipulate an entity: to make it rotate, change its alpha value, and so forth.

Okay, so say we have a WDL action, such as:

```
action my_dummy
{
while(1)
{
// call the DLL here
wait(1);
}
}
```

This is the minimum framework (the channel) which is absolutely necessary if we are to let our DLL take over.

First we declare a new dllfunction which triggers our object - let's say:

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

and in our DLL we add:

```
DLLFUNC fixed call_my_dummy(long entity)
{
///////// here comes the action
return 0;
}
```

We also need to add a pointer to our object, like so:

```
A4_ENTITY *dummy;
```

Now we can trigger the DLL by modifying the while loop slightly:

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

Let's analyse what happens when the while loop executes `call_my_dummy(my)`.

It transfers the object to the DLL. Here it is received by the function, but once again, not as an object as such, but as an address.

Thus a conversion is required to change the long entity in the brackets into an `A4_ENTITY` pointer.

This can be achieved by the following code:

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

Fine. That works.

Let us recapitulate what we have done so far.

We have a WDL action that triggers a DLL function, and this function receives the address of our object. Now we can begin to manipulate our object using the DLL.

For example:

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

In this example, our object would begin to rotate.

Let us have a look at the extra line here. Why do we use this strange expression, `FLOAT2FIX(0.1)`? Why can't we just write the following?

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

The reason for this is that WDL can only accept the fixed data type. So if we want to send values back to the engine we have to recast them to fixed beforehand.

There are a few macros that do this for us, such as:

```
INT2FIX(1);           // converts an integer to fixed
FLOAT2FIX(1);         // converts float to fixed
```

And they can also be used the other way around:

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

You must get into the habit of using these conversions if you plan on using a DLL as the main development

platform for your game. At first, this may seem a laborious process, tripling your work: if you have a simple variable, you must convert it to the desired data type and then back again.

Essentially, this consists of the following continuous process:

WDL -> DLL -> WDL

(Which explains our German -> English -> German analogy earlier.)

You might be wondering what advantages this extra work yields over simply coding your game in WDL. But the work does pay off: you will gain the ability to build classes, you will never again have to worry about being limited to 48 skills, and you will be working in a true object oriented environment.

6. Building a class

Now let's say we want to create something more complex, not just a dummy object for a WDL entity. We would start by constructing a new class, like this:

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

A4_ENTITY *ent;

A4_TEXT *text;
A4_PANEL *panel;

} SmartGuy;
```

Instead of creating a global pointer we have encapsulated our example entity in a class. Further, we have added some other pointers. And to show how easy it is to go beyond the skill limit, our entity now has a two-dimensional array with lots of values available to fill.

We can now activate our smart dummy just as easily as we did our global dummy:

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

You should now be beginning to see how easy this is. At this point it would be expedient to write a main function for our class object, which should be part of the class declaration:


```

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);  // now the rotation is triggered automatically
}

```

We have now successfully built up a complete means of communication with our DLL. Every time the WDL loop calls the dllfunction, the object is triggered and routed into the class function main().

So you don't have to worry about WDL calls.

7. Using WDL functions

But let's say you want to use a WDL function inside your DLL. This is tricky because the code looks extremely complicated.

We'll assume that we have already declared the function at the beginning and imported it into the DLL (as explained above).

Now let's go through the necessary steps.

First let us define a vector to receive the player position:

```

class smart_dummy
{
...

public:
..
/// this is what we would add
fixed temp[3];

execute_wdl();
...
}

smart_dummy::execute_wdl()
{

```

```
temp[0] = INT2FIX(1000);    // remember, you have you work with fixed data type
temp[1] = INT2FIX(200);
temp[2] = FLOAT2FIX(0.2);

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

The first few lines are self-explanatory. If we use a WDL function we have to work with the fixed data type; thus conversion is necessary.

However, the last line is confusing at first, so let's take a closer look at it.

Why do we say `(*vec_set)` ?

We do this because we do not have direct access to this function. The version of “`vec_set`” in the DLL is only a pointer, and since we do not want to transfer the address but only the value, we have to reference it. This is done by the first bracket.

Next, if we want to use a parameter we have to convert it to the long data type.

Take a look at the declaration in the a5dll header:

```
typedef fixed (*wdfunc1)(long);
typedef fixed (*wdfunc2)(long,long);
typedef fixed (*wdfunc3)(long,long,long);
typedef fixed (*wdfunc4)(long,long,long,long);
```

We have to adjust our call so that the data types it uses accord with the types declared in this header.

Thus we have to recast our fixed vector temp to long, like this:

```
(long) temp;
```

In the case of the first parameter we do not access a fixed vector[3], but just the address of the entity's vector that is to be affected, vector[0]. Therefore we write

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

This translates as: take the address and read it as a long data type.

To conclude:

All this casting and recasting might be painstaking work, but never forget that it will give you access to all the advantages of object oriented programming. You can expand your action to the desired complexity. It is worth the effort.

8. Event handling

The architecture of the DLL would be incomplete if we did not integrate event handling. For this, we once again have to step back from the DLL and add a few lines to our WDL action.

First, we need to declare a dllfunction which executes the DLL side of the event handling:

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

Then you have to write an event function to trigger the DLL:

```
function dummy_event()
{
  call_my_dummyevent(event_type, my); // this triggers the DLL event
}

action my_dummy
{
  my.enable_click = ON;
                                // add all the event types you want to use
  my.enable_shoot = ON;

  my.event = dummy_event; // this triggers the event

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

Okay, this is a little long-winded. Our action calls a WDL event function, and the WDL event function calls a DLL event function.

You may wonder about the variable “`event_type`”, which is passed as an argument. I wondered about that too - until I understood that `event_type` is just a synonym for a unique number.

For instance, if the event type is Click, it would not pass across the string “click”, but a number – in this case 16.

Every event type has a unique corresponding number. We will look at them shortly.

We have now declared a `dllfunction` in WDL, but have yet to do so in the DLL. So this is what we have to do next:

```
DLLFUNC fixed call_my_dummyevent(fixed event_type, long entity)
{
  // now we should trigger our SmartDummy
  return 0;
}
```

This function establishes a channel of communication. Every time an event occurs the DLL is triggered. Until now nothing happens, though; we have yet to assign the entity and then add another `event_function` to our `SmartDummy` class.

Let’s do that now:

```

class smart_dummy
{
...

public:
..
// this is what we would add

call_event(int number); // remember "event type" is just a number
...
}

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
    {
    }

}

```

Okay, it would have been more elegant to write a switch, but you're free to improve the code as you wish. The most important thing is that you can assign the correct events to the number that has been passed across.

It just remains to modify the interface function:

```

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

```

So there we are – we have now undocked from WDL and are completely free to concentrate on C++ programming.

NB. For the purposes of this tutorial, I have tried to reduce everything to the bare minimum. If you are wondering about missing constructors and destructors, don't worry. I have omitted them for the sake of making the process transparent.