3D GameStudio

Source Development Kit Programmer's Manual

for A5 Engine 5.51

Johann C. Lotter / Conitec February 2003

This manual is protected under the copyright laws of Germany and the U.S. Acknex, A4, A5, and 3D GameStudio are trademarks of Conitec Corporation. Windows, DirectX and Direct3D are trademarks of Microsoft, Inc. Any reproduction of the material and artwork printed herein without the written permission of Conitec is prohibited. We undertake no guarantee for the accuracy of this manual. Conitec reserves the right to make alterations or updates without further announcement.

Contents

The A5 plugin interface

What is a DLL? Basically it's an external library that adds functions to a program. A5 DLLs can be used as "**plugin**" extensions to the engine and to the C-Script language. For creating an A5 plugin, the SDK (source development kit) and a development system like VC++ or Delphi is required. DLL plugins are used for adding new effects, actor AI or new C-Script instructions, as well as for programming a game totally in C++ or Delphi instead of C-Script. Theoretically everything - MP3 or MOD players, a physics engine, another 3D engine or even another scripting language - could be added to the engine this way. Because DLL plugins work with all editions, they can be distributed or sold to other 3D GameStudio users. On our download and link pages at http://www.3dgamestudio.com you can already find a lot of useful plugins created with the SDK by GameStudio users.

While the **Microsoft Visual C++™** (version **6.0** or **.NET**) development system normally is recommended for creating DLL plugins, users have also provided libraries for **Borland C++™**, **C++ Builder™**, or **Delphi™** up to version 6, which are included in the SDK. Please note however that Conitec can not give support for user-provided Delphi and Borland libraries. The DLL SDK contains an example project, which makes it easy to create extensions even for not-so-experienced C or Delphi programmers who have never written a DLLs before. The SDK license includes the right to freely distribute DLLs created with it, as long as the DLL functions only provide application functionality. It is not allowed to distribute DLLs that work as a 'wrapper' for the library by providing functions that allow direct access of the interface structures or the library functions from outside the DLL.

The following documentation contains just the description of the DLL interface to the C-Script API, and some examples. The C-Script functions themselves are described in the **GameStudio C-Script Manual**. C-Script functions are the API for the DLL as well as for scripts, so you'll definitely need both manuals for creating DLL plugins.

Getting started with the SDK

You'll find the SDK either as a ZIP file on your **key disk**, or on a separate **SDK disk**. You can unzip or copy the SDK into a directory of your choice, and open it as a VC++ project. The SDK comes with a DLL source file, **ackdll.cpp**, that contains some typical examples for DLL functions. Examine the examples carefully – it's the best way to see how to program DLL functions and access engine parameters! You can use them as a 'template' for your own DLL.

If you have updated to the most recent SDK version, you'll find the Borland file versions in the **BCPP** and **Delphi** subfolders – read the **readme.txt** for more information about how to create Borland and Delphi projects. We are here using **VC++ .Net** for our following examples. We are also assuming that you have some basic knowledge of C and C-Script. If not, read a C++ book, and read the first chapter of the **C-Script tutorial** before continuing here.

When you create a new project (File->New Project), VC++ offers you a choice of project templates. Don't select "MFC DLL" here – with VC++ .Net it's the plain **Win32 Project**, with VC++ 6.0 the **Win32 Dynamic-Link Library** that you'll want to create. When the Win32 Application Wizard pops up, select **DLL** (.NET) or **Simple DLL** (6.0) in the application settings and you're done. VC++ now creates a new DLL project for you. The main .cpp file will look like this:

```
// plugin.cpp : Defines the entry point for the DLL application.
//
#include "stdafx.h"
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    return TRUE;
}
```

That's the main entry point of your DLL and you can leave that function unchanged. Copy all of the SDK files (except the Borland/Delphi subfolders of course) into the folder that is created by VC++. Now for compiling an A5 plugin DLL, you have to link the **a5dll.lib** (one of the files of the SDK) to the project (Project Properties -> Linker Input -> Additional Dependencies), and include the **a5dll.h** and **a5funcs.h** files to your main .cpp file. These 3 files are what you need for creating A5 DLLs. You can see in the **ackdll.cpp** example how it should look like.

Now you can begin to add functions to the DLL that can then later be called by a script, or by the main function of your game. To be recognized by the engine, all such functions must be of type **DLLFUNC fixed function(...)**, like this:

```
// returns the value of x * 2^n
DLLFUNC fixed ldexp(fixed x,fixed n)
{
   return (FLOAT2FIX(FIX2FLOAT(x)*pow(2.0,FIX2FLOAT(n))));
}
```

This example function just returns an arithmetic expression of its arguments. **DLLFUNC** is not a part of C++ - it's just a convenience shortcut for declaring DLL export functions. **fixed** is the all-purpose numeric variable type of A5 and C-Script - a long integer that can be used either as 22.10 fixed point value, or as a pointer. Both are declared in the **a5dll.h** together with some conversion functions:

```
#define DLLFUNC extern "C" __declspec(dllexport)

typedef long fixed;  // fixed point 22.10 number format used by C-Script
inline fixed INT2FIX(int i) { return i<<10; }
inline int FIX2INT(fixed x) { return x>>10; }
inline double FIX2FLOAT(fixed x) { return ((double)x)/(1<<10); }
inline fixed FLOAT2FIX(double f) { return (fixed)(f*(1<<10)); }
```

The engine will pass and expect all numbers – coordinates, variables, no matter what – in **fixed** type. So convert any number to **fixed** before you return it to the engine, like in the example above.

Ready? Now compile your DLL – let's assume that you named it plugin.dll – and copy it into your work folder. How can we now call our **ldexp** function by a script? We have to do two things: declare the function, and open the DLL. The first is achieved by a **dllfunction** prototype in the script:

```
dllfunction ldexp(x,n); // declaration of a DLL function
```

This makes our **ldexp** function known to C-Script. Before we can call it, we have to open the DLL. A good place to do this is the **main()** function of your script:

```
function main() {
...
dll_open("plugin.dll");
```

and do not forget to close the DLL before exiting the game:

```
...
dll_close(dll_handle);  // just use the default handle as long as there's only one DLL
exit;
```

After this is done, you can now enjoy that C-Script has gotten one extra instruction:

```
...
x = ldexp(y,n); // calculates x = y * 2^n
...
```

For **debugging** your DLL in VC++, set the command in Project Properties -> Debug to the engine EXE path (like "C:\program files\gstudio\bin\acknex.exe"), the command arguments to your script and command line parameters (like "mygame.wdl -wnd") and the working directory to your game directory (like "C:\program files\gstudio\mygame"). In Project Properties -> General, set the output directory and the intermediate directory to . (a period, meaning the working directory) for the engine to find the intermediate files. Then you can compile and debug your DLL by setting breakpoints as usual.

Ok, this was the basics of writing plugin DLLs. Of course, there's a lot more to learn. The methods for exchanging data with the engine are described in the following. All DLL functions can be declared and called in scripts just like each other C-Script function, after having activated the DLL through the **dll_open** and **dll_close** instructions described in the script manual.

Using C-Script objects in a DLL

We have learned how to add new C-Script instructions, but how can we access C-Script variables, objects and functions from within a DLL? We have some library functions to do that. All library functions provided by the SDK are preceded by **a5dll_**. The most often used function is

long a5dll_getwdlobj(char *name);

This function returns the address of the C-Script object or variable with the given name. It can be used to read or write any defined C-Script object from inside a DLL plugin. If the object does not exist, **NULL** is returned and an error message will pop up. Examples for DLL functions that access C-Script objects:

```
// adds the given value to a C-Script vector
fixed AddToVector(fixed value)
{
// get the address of the variable
  fixed *myvector = (fixed *)a5dll_getwdlobj("myvector");

// add the same value to the 3 components
  myvector[0] += value;
  myvector[1] += value;
  myvector[2] += value;

  return value;
}
```

So you can use this function to obtain a pointer to any C-Script object, no matter whether it's predefined by the engine or defined in our script. In this case it's a vector which was defined in C-Script by

```
var myvector[3] = 1,2,3;
```

and **a5dll_getwdlobj** just returns a pointer to this vector, which is, by the way, an array of **fixed**. Why don't we have to convert to/from fixed here? Because **value** is already fixed and we can add two fixed numbers just as we add two integers (however we could not do this with multiplication and division!).

It is not wise to obtain the myvector pointer directly in the function where it's needed, the way it's done here for tutorial purposes. **a5dll_getwdlobj** executes slow because it has to find the address in a list. Rather, you would have a global list of pointers to all WDL objects that you need in your DLL, and obtain the pointers once at the beginning in a startup function by calling **a5dll_getwdlobj** for each of them.

So now we know how to read and change a C-Script variable, but how about more complex objects, like panels, views or entities? Take a look at **a5dll.h** – there you will find declared all structs known to C-Script, like **A4_STRING**, **A4_ENTITY**, **A4_PARTICLE**, **A4_BMAP**, **A4_TEX** etc. All those struct names begin with **A4_** (even if it's the A5 engine meanwhile). You can get a pointer to such a struct the same way like to a variable:

```
// zooms the camera view
fixed ZoomIn(fixed value)
{
  A4_VIEW *camera = (A4_VIEW *)a5dll_getwdlobj("camera");
  return (camera->arc -= value); // change the FOV and return it
}
```

"Camera" is our main predefined **view**. Because **a5dll_getwdlobj** returns a long integer, you have to cast it to the desired type, as through the **(A4_VIEW *)** cast operator here. As you can see in the struct declarations in **a5dll.h**, the **A4_VIEW** struct contains all the parameters that you are used from C-Script, like **arc**, **ambient** etc., and once you have the pointer you can change and read all of them.

So now we know how to access all C-Script objects, and call DLL functions from C-Script. But how do we do the opposite – calling engine and C-Script functions from a DLL?

Using the API

Accessing C-Script functions is done in a similar way like accessing C-Script objects, through:

long a5dll_getwdlfunc(char *name);

This function returns the address of the C-Script instruction with the given name. It can be used to call engine functions from inside a DLL plugin. Not all C-Script instructions are available for DLLs. If the instruction is not available because it makes no sense in a DLL (like **wait()** or **inkey()**), **NULL** is returned and an error message will pop up. Example for an entity AI DLL function that uses C-Script functions for scanning the environment of an entity:

```
// returns free distance in front of MY entity until next obstacle
fixed DistAhead(long p_ent)
{
  if (!my) return 0;

// retrieve the pointer to the given entity
  A4_ENTITY *ent = (A4_ENTITY *)p_ent;

// get the address of some script variables and functions
  fixed *tracemode = (fixed *)a5dll_getwdlobj("trace_mode");
  wdlfunc2 vecrotate = (wdlfunc2)a5dll_getwdlfunc("vec_rotate");
  wdlfunc2 trace = (wdlfunc2)a5dll_getwdlfunc("trace");

  fixed target[3] = { FLOAT2FIX(1000.0),0,0 }; // trace target vector

// rotate vector by entity engles, just as in C-Script
  (*vecrotate)((long)target,(long)&(ent->pan));

// add entity position to target
  target[0] += ent->x;
  target[1] += ent->y;
  target[2] += ent->z;

// set trace_mode, then trace a line between entity and target,
// and return the result
  *tracemode = INT2FIX(TRM_IGNORE_ME + TRM_IGNORE_PASSABLE + TRM_USE_BOX);
  return (*trace)((long)&(ent->x),(long)target);
}
```

Let's examine the important part of the code in detail:

```
  wdlfunc2 vecrotate = (wdlfunc2)a5dll_getwdlfunc("vec_rotate");
```

wdlfunc2 is a convenience typedef for a pointer to a C-Script instruction that takes 2 arguments. Because all C-Script instructions take either 1, 2, 3, or 4 arguments, there are 4 such typedefs in the **a5dll.h**:

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

Once we've gotten the pointer to that instruction – again, it's recommended to retrieve pointers to all used instructions in a startup function – we can just call it:

```
// rotate vector by entity engles, just as in C-Script
  (*vecrotate)((long)target,(long)&(ent->pan));
```

This looks a little different than you are used to call a function in C++! However, it's quite straightforward: We have a pointer to that function, so for calling the function itself we have to use the (*...). And the arguments passed are always fixed or long. We have casted them to long instead of fixed as a convention to indicate that we are passing pointers. All vector instructions expect pointers to fixed.

All this pointer handling and typecasting may seem a little complicated at first, but because it's logical you'll fast get the grip of it.

What if a C-Script instruction expects not a vector or value, but something more complicated like an entity? Well, we'll then just pass the **A4_ENTITY** pointer casted to long. And what if it expects a string – must we really use a pointer to **A4_STRING** or can we just pass **char***? We must use **A4_STRING** I'm afraid. But in the example ackdll.cpp you can find an easy way how to pass a string constant to a C-Script instruction:

```
long pSTRING(char* chars)   // convenience function to make string passing easy
{
   static A4_STRING tempstring;
   static char tempname[256];
   strncpy(tempname,chars,255);
   tempstring.chars = tempname;
   return (long)&tempstring;
}

// example for passing a string to create an entity
DLLFUNC fixed create_warlock(long vec_pos)
{
   wdlfunc3 ent_create = (wdlfunc3)a5dll_getwdlfunc("ent_create");
   return (*ent_create)(pSTRING("warlock.mdl"),vec_pos,0);
}
```

Some special C-Script functions, like keyboard entry, can not be called directly from a DLL. However they can be executed indirectly by calling a script that executes that function. Scripts can be called from a DLL through the following functions:

long a5dll_getscript(char *name);

This function returns an addresss of the user-defined script function with the given name. It can be used to call user defined C-Script actions or functions from inside a DLL plugin. If the function is not found, **NULL** is returned and an error message will pop up.

fixed a5dll_callscript(long script,long p1=0,long p2=0,long p3=0,long p4=0);
fixed a5dll_callname(char *name,long p1=0,long p2=0,long p3=0,long p4=0);

These functions call a user-defined script function with given address or given name. The 4 parameters can be a fixed point number, an array, or a pointer to a C-Script object. If the function expects less than 4 parameters, the superflous ones can just be set a 0.

Example for a DLL function that calls a function that must be defined in the C-Script code:

```
DLLFUNC fixed WDLBeep(fixed value)
{
// get the function
   long beeptwice = a5dll_getscript("beeptwice");
// call it
   return a5dll_callscript(beeptwice,0,0,0,0);
}
```

This DLL function expects the following function in the C-Script which is then called:

```
function beeptwice() { beep; beep; } // in the script
```

Now that we have learned to access every part of C-Script by a DLL and vice versa, let's continue with some special applications for DLL functions.

Using Direct3D functions

The following example shows how easy it is to use Direct3D functions for creating some effects on the screen. As all initialization is done by the engine, it is sufficient just to call the draw functions. All Direct3D functions are accessed through a **LPDIRECT3DDEVICE8** pointer that is available through the DLL. For details refer to the DirectX documentation that is available, along with the DirectX 8.1 SDK, from the Microsoft site.

The example paints a multicolored triangle onto the screen. You'll see the triangle briefly flashing in the upper left corner when you call **PaintD3DTriangle()** once. If you call it in a **wait(1)**-loop, the triangle will be permanently on the screen.

```
#include <d3dx8.h> // from the DIRECTX8.1 sdk

// dllfunction PaintD3DTriangle();
// draws a red/blue/green triangle in D3D mode
DLLFUNC fixed PaintD3DTriangle (void)
{
// get the active D3D device
    LPDIRECT3DDEVICE8 pd3ddev = (LPDIRECT3DDEVICE8) a5dx->pd3ddev8;
    if (!pd3ddev) return 0;

// define a suited vertex struct
    struct VERTEX_FLAT { float x,y,z; float rhw; D3DCOLOR color; };
    #define D3DFVF_FLAT (D3DFVF_XYZRHW | D3DFVF_DIFFUSE)

// define the three corner vertices
    VERTEX_FLAT v[3];

    v[0].x = 10.0; v[0].y = 10.0; v[0].color = 0xFFFF0000;  // the red corner
    v[1].x = 310.0; v[1].y = 10.0; v[1].color = 0xFF0000FF; // the blue corner
    v[2].x = 10.0; v[2].y = 310.0; v[2].color = 0xFF00FF00; // the green corner

    v[0].z = v[1].z = v[2].z = 0.0;           // z buffer - paint over everything
    v[0].rhw = v[1].rhw = v[2].rhw = 1.0;  // no perspective

// begin a scene - needed before D3D draw operations
    pd3ddev->BeginScene();

// set some render and stage states (you have to set some more, normally)
    pd3ddev->SetRenderState(D3DRS_ALPHABLENDENABLE,FALSE);
    pd3ddev->SetTextureStageState(0,D3DTSS_COLORARG2,D3DTA_DIFFUSE);
    pd3ddev->SetTextureStageState(0,D3DTSS_COLOROP,D3DTOP_SELECTARG2);

// now draw the triangle
    pd3ddev->SetVertexShader(D3DFVF_FLAT);
    pd3ddev->DrawPrimitiveUP(D3DPT_TRIANGLEFAN,1,(LPVOID)v,sizeof(VERTEX_FLAT));

// do not forget to do a clean closing of the scene
    pd3ddev->EndScene();
    return 0;
}
```

Note: Depending on the 3D hardware, sometimes A5 has to release and reallocate the Direct3D device when the video output is switched between window and fullscreen mode. If you use the pd3ddev8 for allocating an object, like a texture or a buffer, you have to release the object before switching video, and recreate it afterwards. Otherwise the device can't be released and you'll receive an **Uninitialized Device** error message.

Particle functions

DLL functions can also be used for particles, using the **A4_PARTICLE** struct defined in **a5dll.h**. They can be used the same way as C-Script defined particle functions. A pointer to the particle is the sole argument of a DLL particle function. Example:

```
// examples for a particle effect function
// dllfunction DLLEffect_Explo(particle);
// dllfunction DLLPart_Alphafade(particle);
```

```
// start the particle effect by
// effect(DLLEffect_Explo,1000,my.x,nullvector);

fixed *var_time = NULL;
long func_alphafade = 0;

// helper function: fades out a particle
DLLFUNC fixed DLLPart_Alphafade(long particle)
{
  if (!var_time || !particle) return 0;
  A4_PARTICLE* p = (A4_PARTICLE*) particle;
  p->alpha -= *var_time * 2;
  if (p->alpha <= 0) p->lifespan = 0;
  return 0;
}

// helper function: return a random float
float random(float max)
{
  return (float)(rand()*max)/(float)RAND_MAX;
}

// particle effect: generate a blue explosion
DLLFUNC fixed DLLEffect_Explo(long particle)
{
  if (!particle) return 0;
// initialize time var and alphafade function (must only be done once)
  if (!var_time)
    var_time = (fixed *)a5dll_getwdlobj("time");
  if (!func_alphafade)
    func_alphafade = a5dll_getscript("DLLPart_Alphafade");
  A4_PARTICLE* p = (A4_PARTICLE*) particle;

// initialize particle parameters
  p->flags |= EPF_STREAK|EPF_MOVE|ENF_FLARE|ENF_BRIGHT;
  p->vel_x = FLOAT2FIX(random(10) - 5);
  p->vel_y = FLOAT2FIX(random(10) - 5);
  p->vel_z = FLOAT2FIX(random(10) - 5);
  p->red = 0;
  p->green = 0;
  p->blue = INT2FIX(255);
  p->alpha = FLOAT2FIX(50 + random(50));
  p->function = func_alphafade;
  return 0;
}
```

Sending information over the network

The SDK can use A5's send and receive functions for sending user-defined messages in a multiplayer environment. For this, the **SendPacket** and **ReveivePacket** function pointers are available via the **ENGINE_INTERFACE**:

```
typedef struct {
     byte *save_block;// pointer to block of variables for save/load (not used)
     int  save_size;  // size of block of variables for saving (not used)
     long (*Exec)(long n,long p1,long p2,long p3);     // DLLLIB internal use only
// only available in A5.51 or above - first packet byte must be 17 (0x11) for user defined packets
     void (*SendPacket)(long to,void *data,long size,long guaranteed);     // the send function of the engine
     void (*ReceivePacket)(long from,void *data,long size);      // user provided function
  } ENGINE_INTERFACE;
```

SendPacket sends a user defined packed from the client to the server, or vice versa.

Parameters:

to - Identifier number for the client to receive the message. Set to 0 for sending to all clients.
data - Data packet to be sent. First byte must be 17 (0x11) for identifying a user-defined packet.
size - Size of the packet in bytes.
guaranteed - set to 1 for TCP/IP mode, 0 for UDP mode.

ReceivePacket can be set to a user provided void(long,void*,long) function that receives and interprets messages sent with **SendPacket**.

Parameters:
from – ID Number of the sender. If at 0, the message was received from the server.
data - Data packet to be sent. First byte is always 17 (0x11) for identifying a user-defined packet.
size - Size of the packet in bytes.

Note that the receive function should be very short and mainly just store the message, for not interfering the receive process. It must not send, open a file, render, or do anything time consuming.

Programming a game in C++

Using the **A4_ENTITY** object (see below), a DLL can implement complex AI functions that would be harder to code in C-Script. Even the whole gameplay could be written in a DLL. The following example shows how to change entity parameters through a DLL function.

```
// rolls the given entity by 180 degrees
DLLFUNC fixed FlipUpsideDown(long entity)
{
   if (!entity) return 0;

// retrieve the pointer to the given entity
   A4_ENTITY *ent = (A4_ENTITY *)entity;

// set the entity's roll angle to 180 degrees
   ent->roll = FLOAT2FIX(180);

   return 0;
}
```

This would be called by C-Script through **FlipUpsideDown(my)**. For controlling entities totally through a DLL – for instance, when you intend to write your whole game in C++ or Delphi, instead of C-Script – C-Script dummy actions can be assigned to the entity, like this:

```
var appdll_handle;
dllfunction dll_entmain(entity);
dllfunction dll_entevent(entity);

function main()
{
// open the application DLL
   appdll_handle = dll_open("myapp.dll");
   ...
}

action myent_event {
   dll_handle = appdll_handle;
   dll_entevent(my);    // this DLL function handles all entity events
}

action myentity {
   my.event = myent_event;
   while(1) {
      dll_handle = appdll_handle;
      dll_entmain(my); // this DLL function controls the entity
      wait(1);
   }
}
```

DLL interface structures and special functions

Interface structs are initialized at DLL startup for accessing essential engine variables and pointers. There are three such interfaces, which are defined in the **a5dll.h**: the **WDL_INTERFACE a5wdl** that contains C-Script access functions and he **MY** and **YOU** entity pointers, the **ENGINE_INTERFACE a5eng** that contains basic engine functions, and the **DX_INTERFACE a5dx** that contains pointers to all DirectX devices initialized by A5. Normally you'll only need the last one. For instance, **a5dx->pd3ddev8** will get you the pointer to the Direct3D 8.1 device.

Some utility functions are provided for manipulation of textures and entities:

A4_TEX *a5dll_tex4ent(A4_ENTITY *entity,int frame,int texnum=0);

This function returns the texture of a sprite, model or terrain entity. It can be used to directly access D3D textures (see example below). **Frame** is the frame or skin number, **texnum** the subtexture number if it is split into several subtextures.

A4_ENTITY *a5dll_entnext(A4_ENTITY *entity);

This function enumerates local entities, and can be used to access all entities in a level. If called with NULL, it returns a pointer to the first entity in the level. If called with a level entity pointer, it returns a pointer to the next level entity. If called with a pointer to the last entity or no entity at all, it returns NULL.

Example for a function that uses DirectX 8.1 for painting the textures of model, sprite and terrain entities red:

```
// dllfunction PaintEntitiesRed();
// paints the first mipmap of all sprite and model entities red
DLLFUNC fixed PaintEntitiesRed(void)
{
// find the first entity in the level
   A4_ENTITY *ent = NULL;
   while (1) {
// find the next entity
       ent = a5dll_entnext(ent);
       if (!ent) break;

// we can not be sure that the entity texture exists - it could be purged
       A4_TEX *tex = a5dll_tex4ent(ent,0,0);
       if (!tex) continue;
       LPDIRECT3DTEXTURE8 dx8tex = (LPDIRECT3DTEXTURE8) tex->pd3dtex;
       if (!dx8tex) continue;

// check the texture format
       D3DSURFACE_DESC  ddsd;
       if (FAILED(dx8tex->GetLevelDesc(0,&ddsd))) continue;

// lock the texture and retrieve a pointer to the surface
       D3DLOCKED_RECT d3dlr;
       if (FAILED(dx8tex->LockRect(0,&d3dlr,0,0))) continue;
       byte *pixels = (byte *)(d3dlr.pBits);

// do we have a 16 bit or 32 bit format?
       if (ddsd.Format == D3DFMT_A8R8G8B8)
           for (unsigned y = 0; y < ddsd.Height; y++ ) {
               DWORD *target = (DWORD *)(pixels + y*d3dlr.Pitch);
               for (unsigned x = 0; x < ddsd.Width; x++ )
                   *target++ = 0xFFFF0000; // that's red in 8888
           }
       else if (ddsd.Format == D3DFMT_A4R4G4B4)
           for (unsigned y = 0; y < ddsd.Height; y++ ) {
               WORD *target = (WORD *)(pixels + y*d3dlr.Pitch);
               for (unsigned x = 0; x < ddsd.Width; x++ )
                   *target++ = 0xFF00;  // that's red in 4444
           }
       else if (ddsd.Format == D3DFMT_A1R5G5B5)
           for (unsigned y = 0; y < ddsd.Height; y++ ) {
               WORD *target = (WORD *)(pixels + y*d3dlr.Pitch);
```

```
            for (unsigned x = 0; x < ddsd.Width; x++ )
               *target++ = 0xFC00;  // that's red in 1555
         }
      else if (ddsd.Format == D3DFMT_R5G6B5)
         for (unsigned y = 0; y < ddsd.Height; y++ ) {
            WORD *target = (WORD *)(pixels + y*d3dlr.Pitch);
            for (unsigned x = 0; x < ddsd.Width; x++ )
               *target++ = 0xF800;  // that's red in 565
         }

// Unlock the surface again
      dx8tex->UnlockRect(0);
   }

   a5dll_errormessage("Entities are now red!");
   return 0;
}
```

void a5dll_errormessage(char *text);

This function pops up an Error #1527 message requester with the given text. It can be used to display diagnostic messages, or notify the user of wrong DLL calls, like with an invalid entity pointer.

One final consideration. On accessing system resources like sound, video, joystick and so on, the DLL must take care of possible resource conflicts. The engine shares its resources and expects the same from the code inside the DLL. For instance, code that requires exclusive access to the sound device (like some old MOD players) won't work. Some resources (like the midi player) can't be shared - if midi music is played by the DLL, the engine must not play a midi file at the same time and vice versa.

The A5 Client/Server Protocol

The protocol is optimized for using as less bandwidth as possible. Only parameters that have changed are sent over the network. Sending a player's changed XYZ coordinate from the server to the client, for instance, needs only 12 bytes (including header). A dead reckoning mechanism is used for extrapolating positions and angles between cycles.

The structure of the messages is a single-byte code, followed by code-dependant informations. When describing the content of messages, we use the following conventions:

Byte = an unsigned integer, on one byte.
Short = a signed integer, on two bytes, Big Endian order (Intel order).
Long = a signed integer, on four bytes, Big Endian order (Intel order).
Float = a floating point number, on four bytes, Big Endian order (Intel order).
Fixed = a fixed point number in 22.10 format, on four bytes, Big Endian order (Intel order).
String = a sequence of characters, terminated by 0 ('\0')
Angle = a short, to be multiplied by 360.0/65535.0 to convert it to degrees.
Position = a coordinate packed in three bytes by dividing it by 8
CPosition = either **Position** or **Fixed,** depending on the **pos_resolution** variable
Scale(x) = a value packed into one byte to be multiplied by x/255.0.

Client Messages

The following commands are used for transferring information from a client to the server.

Command	*Bytecode*	*Arguments*	*Description*
cls_fill	**0x01**		Filler byte for inflating UDP messages to a minimum length. Can be ignored.
cls_join	**0x02**	**String Player_Name**	Request for joining the session (TCP).
cls_create	**0x03**	**String File_Name** **Position Start[3]** **Short Action_Index** **Short Identifier**	Request creating an entity with given model name, and link the client to it (TCP).
cls_remove	**0x04**	**Long Entity_Index**	Request removing entity on the server (TCP).
cls_ping	**0x07**		Sent after each client frame (UDP). If a client does not send anything for more than 5 seconds, it is automatically disconnected by the server.
cls_level	**0x09**	**String Level_Name**	Inform server that client has loaded a level (TCP).
cls_var	**0x0a**	**Short Var_Index** **Short Var_Length** **Fixed Var[Var_Length]**	Send a variable or an array (TCP).
cls_string	**0x0b**	**Short String_Index** **String Text**	Send a string (TCP).
cls_skill	**0x0e**	**Short Entity_Index** **Short Struct_Offset** **Fixed Skill**	Send an entity skill (TCP). **Struct_Offset** gives the byte offset of the skill in the **A4_ENTITY** struct.
cls_skill3	**0x0f**	**Short Entity_Index** **Short Struct_Offset** **Fixed Skill[3]**	Send an entity vector skill (TCP).

Server Messages

The following commands are used for transferring information from the server to either a specific client, or to all clients connected. Server-client communication uses the reliable TCP protocol for important messages, and the unreliable UDP protocol for unimportant messages.

Command	*Bytecode*	*Arguments*	*Description*
svc_fill	**0x01**		Filler byte for inflating UDP packets to at least 8 bytes (can be ignored).
svc_create	**0x03**	**Short Entity_Index** **Short Identifier**	Created entity with given index (TCP).
svc_remove	**0x04**	**Short Entity_Index**	Removed entity from server (TCP).
svc_entsound	**0x05**	**Short Entity_Index** **Short Sound_Index** **Scale(2000) Volume** **Long Sound_Handle**	Play an entity sound on the clients (UDP).
svc_effect	**0x06**	**Short Action_Index** **Short Number** **Position Start[3]** **Fixed Vel[3]**	Generate a particle or beam effect on the clients (UDP).
svc_info	**0x07**	**Long 0x11191218** **Byte Protocol_Version** **Float Server_Time** **Float Frame_Time**	Send a sync value and the server time to the clients (TCP). This is sent once a frame.
svc_var	**0x0a**	**Short Var_Index** **Short Var_Length** **Fixed Var[Var_Length]**	Send a variable to the client (TCP).
svc_string	**0x0b**	**Short String_Index** **String Text**	Send a string to the client (TCP).
svc_skill	**0x0e**	**Short Entity_Index** **Short Struct_Offset** **Fixed Skill**	Send an entity skill to the client (TCP). **Struct_Offset** gives the byte offset of the skill in the **A4_ENTITY** struct.
svc_skill3	**0x0f**	**Short Entity_Index** **Short Struct_Offset** **Fixed Skill[3]**	Send an entity vector skill to the client (TCP).
svc_local	**0x12**	**Short Entity_Index** **Short Function_Index**	Start the given function with the given MY entity on the client (TCP).
svc_update1	**0x40..0x7f**	**Short Entity_Index** (Parameters see below)	Update entity parameters 1 (UDP).
svc_update2	**0x80..0xbf**	**Short Entity_Index** (Parameters see below)	Update entity parameters 2 (UDP).
svc_update3	**0xc0..0xff**	**Short Entity_Index** (Parameters see below)	Update entity parameters 3 (UDP).

For the 3 entity parameter update messages, bits 0..5 of the **svc_update** bytecode give the parameter combination to be sent or received, in the order given below. All parameters are sent through the UDP protocol.

Parameter	*Update.Bit*	*Arguments*	*Remarks*
position	**2.0**	**CPosition Pos[3]**	XYZ position
pan	**2.1**	**Angle Pan**	
tilt	**2.2**	**Angle Tilt**	
roll	**2.3**	**Angle Roll**	

Parameter	*Update.Bit*	*Arguments*	*Remarks*
frame	**2.4**	**Short Frame_Int Scale(1) Frame_Frc Short Nextframe**	Frame number, tweening factor, tweening target
flags1	**2.5**	**Short Flags 8..23**	
type	**1.0**	**String File_Name**	Name of the mdl, wmb, pcx, etc. file
scale	**1.1**	**Short Scale[3]**	XYZ scale*0.25
ambient	**1.3**	**Scale(100) Ambient**	
albedo	**1.4**	**Scale(255) Albedo**	
skin	**1.2**	**Byte Skin**	Skin number
lightrange	**3.0**	**Scale(2000) Lightrange**	
color	**3.1**	**Scale(255) Red,Green,Blue**	RBG color packed in 3 bytes
alpha	**3.2**	**Scale(100) Alpha**	
uv	**3.3**	**Fixed U,V**	UV offset or speed for entity textures

For instance, the code sequence

0x83 0x07 0x00 0x80 0x00 0x00 0x00 0x01 0x00 0x80 0x01 0x00 0x80 0x00

updates position and pan angle (**0x83** has bits 0 and 1 set) of entity No. 7 (**0x07 0x00**). The position uses the packed format and is set at coordinates x=1 (**0x80 0x00 0x00)**, y=2 (**0x00 0x01 0x00**) and z=3 (**0x80 0x01 0x00**), and the pan angle is set at 180 degrees (**0x80 0x00**).

The MDL5 model format

Despite the engine uses model files with .MDL extension, it's internal MDL5 format differs from the Quake MDL format. A wireframe mesh, made of triangles, gives the general shape of a model. 3D vertices define the position of triangles. For each triangle in the wireframe, there will be a corresponding triangle cut from the skin picture. Or, in other words, for each 3D vertex of a triangle that describes a XYZ position, there will be a corresponding 2D vertex positioned that describes a UV position on the skin picture.

It is not necessary that the triangle in 3D space and the triangle on the skin have the same shape (in fact, it is normally not possible for all triangles), but they should have shapes roughly similar, to limit distortion and aliasing. Several animation frames of a model are just several sets of 3D vertex positions. The 2D vertex positions always remain the same.

A MDL file contains:
- A list of skin textures in 8-bit palettized, 16-bit 565 RGB or 16 bit 4444 ARGB format.
- A list of skin vertices, that are just the UV position of vertices on the skin texture.
- A list of triangles, which describe the general shape of the model.
- A list of animation frames. Each frame holds a list of 3D vertices.
- A list of bone vertices, which are used for creating the animation frames.

MDL file header

Once the file header is read, all the other model parts can be found just by calculating their position in the file. Here is the format of the .MDL file header:

```
typedef float vec3[3];

typedef struct {
  char version[4];   // "MDL3", "MDL4", or "MDL5"
  long unused1;      // not used
  vec3 scale;        // 3D position scale factors.
  vec3 offset;       // 3D position offset.
  long unused2;      // not used
  vec3 unused3;      // not used
  long numskins ;    // number of skin textures
  long skinwidth;    // width of skin texture, for MDL3 and MDL4;
  long skinheight;   // height of skin texture, for MDL3 and MDL4;
  long numverts;     // number of 3d wireframe vertices
  long numtris;      // number of triangles surfaces
  long numframes;    // number of frames
  long numskinverts; // number of 2D skin vertices
  long flags;        // always 0
  long numbones;     // number of bone vertices (not used yet)
} mdl_header;
```

The size of this header is 0x54 bytes (84).

The **MDL3** format is used by the A4 engine, while the newer **MDL4** and **MDL5** formats are used by the A5 engine, the latter supporting mipmaps. After the file header follow the skins, the skin vertices, the triangles, the frames, and finally the bones (in future versions).

MDL skin format

The model skins are flat pictures that represent the texture that should be applied on the model. There can be more than one skin. You will find the first skin just after the model header, at offset **baseskin = 0x54**. There are **numskins** skins to read. Each of these model skins is either in 8-bit palettized (**type == 0**), in 16-bit 565 format (**type == 2**) or 16-bit 4444 format (**type == 3**). The skin structure in the **MDL3** and **MDL4** format is:

```
typedef byte unsigned char;
typedef struct {
  int skintype;    // 0 for 8 bit (bpp == 1), 2 for 565 RGB, 3 for 4444 ARGB (bpp == 2)
  byte skin[skinwidth*skinheight*bpp]; // the skin picture
} mdl_skin_t;
```

In the **MDL5** format the skin is a little different, because now mipmaps can be stored and the model skins have not necessarily the same size. If the skin contains mipmaps, **8** is added to the **skintype**. In that case the 3 additional mipmap images follow immediately after the skin image. The texture width and height must be divisible by 8. 8 bit skins are not possible anymore in combination with mipmaps.

```
typedef word unsigned short;
typedef struct {
  long skintype;     // 2 for 565 RGB, 3 for 4444 ARGB, 10 for 565 mipmapped, 11 for 4444 mipmapped (bpp = 2),
                     // 5 for 8888 ARGB, 13 for 8888 ARGB mipmapped (bpp = 4)
  long width,height; // size of the texture
  byte skin[bpp*width*height];     // the texture image
  byte skin1[bpp*width/2*height/2];   // the 1st mipmap (if any)
  byte skin2[bpp*width/4*height/4];   // the 2nd mipmap (if any)
  byte skin3[bpp*width/8*height/8];   // the 3rd mipmap (if any)
} mdl5_skin_t;
```

8 bit skins are a table of bytes, which represent an index in the level palette. If the model is rendered in overlay mode, index 0x00 indicates transparency. 16 bit skins in **565** format are a table of unsigned shorts, which represent a true colour with the upper 5 bits for the red, the middle 6 bits for the green, and the lower 5 bits for the blue component. Green has one bit more because the human eye is more sensitive to green than to other colours. If the model is rendered in overlay mode, colour value 0x0000 indicates transparency. 16 bit alpha channel skins in **4444** format are represented as a table of unsigned shorts with 4 bits for each of the alpha, red, green, and blue component. 32 bit alpha channel skins in **8888** format are represented as a table of unsigned long integers with 4 bytes for each of the alpha, red, green, and blue component. Note that the byte order in that case is Intel order: blue, green, red, alpha.

The size width and heights of skins should be a multiple of 4, to ensure long word alignement. When using mipmaps, they must be a multiple of 8. The skin pictures are usually made of as many pieces as there are independent parts in the model. For instance, for a player, there may be several pieces that defines the body, the face, the hands, and the weapon.

Illustration 1: Samurai skin

MDL skin vertices

The list of skin vertices indicates only the position on texture picture, not the 3D position. That's because for a given vertex, the position on skin is constant, while the position in 3D space varies with the animation. The list of skin vertices is made of these structures:

```
typedef struct
{
  short u;  // position, horizontally in range 0..skinwidth-1
  short v;  // position, vertically in range 0..skinheight-1
} mdl_uvvert_t;
```

```
mdl_uvvert_t skinverts[numskinverts];
```

u and v are the pixel position on the skin picture. The skin vertices are stored in a list, that is stored at offset **basestverts = baseskin + skinsize**. **skinsize** is the sum of the size of all skin pictures. If they are all 8-bit skins, then **skinsize = (4 + skinwidth * skinheight) * numskins**. If they are 16-bit skins without mipmaps, then `skinsize =` **(4 + skinwidth * skinheight * 2) * numskins**.

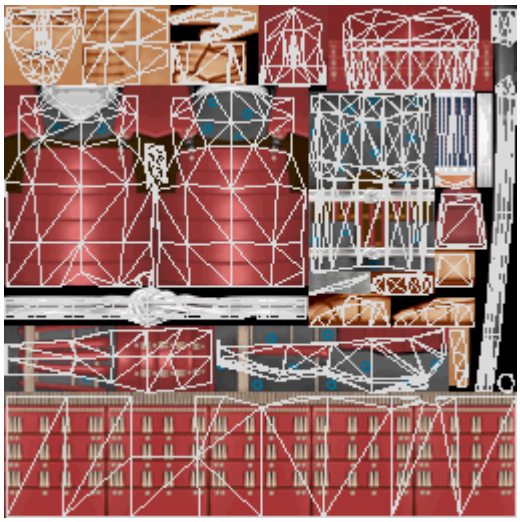

Illustration 2: Samurai skin with uv mapping

MDL mesh triangles

The model wireframe mesh is made of a set of triangle facets, with vertices at the boundaries. Triangles should all be valid triangles, not degenerates (like points or lines). The triangle face must be pointing to the outside of the model. Only vertex indexes are stored in triangles. Here is the structure of triangles:

```
typedef struct {
   short index_xyz[3]; // Index of 3 3D vertices in range 0..numverts
   short index_uv[3];  // Index of 3 skin vertices in range 0..numskinverts
} mdl_triangle_t;

mdl_triangle_t triangles[numtris];
```

At offset **basetri = baseverts + numskinverts * sizeof(uvvert_t)** in the .MDL file you will find the triangle list.

MDL frames

A model contains a set of animation frames, which can be used in relation with the behavior of the modeled entity, so as to display it in various postures (walking, attacking, spreading its guts all over the place, etc). Basically the frame contains of vertex positions and normals. Because models can have ten thousands of vertices and hundreds of animation frames, vertex posistion are packed, and vertex normals are indicated by an index in a fixed table, to save disk and memory space.

Each frame vertex is defined by a 3D position and a normal for each of the 3D vertices in the model. In the MDL3 format, the vertices are always packed as bytes; in the MDL4 format that is used by the A5 engine they can also be packed as words (unsigned shorts). Therefore the MDL4 format allows more precise animation of huge models, and inbetweening with less distortion.

```
typedef struct {
  byte  rawposition[3];   // X,Y,Z coordinate, packed on 0..255
  byte  lightnormalindex; // index of the vertex normal
} mdl_trivertxb_t;

typedef struct {
  unsigned short rawposition[3];   // X,Y,Z coordinate, packed on 0..65536
```

```
  byte  lightnormalindex; // index of the vertex normal
  byte  unused;
} mdl_trivertxs_t;
```

To get the real X coordinate from the packed coordinates, multiply the X coordinate by the X scaling factor, and add the X offset. Both the scaling factor and the offset for all vertices can be found in the `mdl_header` struct. The formula for calculating the real vertex positions is:

```
float position[i] = (scale[i] * rawposition[i] ) + offset[i];
```

The lightnormalindex field is an index to the actual vertex normal vector. This vector is the average of the normal vectors of all the faces that contain this vertex. The normal is necessary to calculate the Gouraud shading of the faces, but actually a crude estimation of the actual vertex normal is sufficient. That's why, to save space and to reduce the number of computations needed, it has been chosen to approximate each vertex normal. The ordinary values of lightnormalindex are comprised between 0 and 161, and directly map into the index of one of the 162 precalculated normal vectors:

```
float  lightnormals[162][3] = {
   {-0.525725, 0.000000, 0.850650}, {-0.442863, 0.238856, 0.864188}, {-0.295242, 0.000000, 0.955423},
   {-0.309017, 0.500000, 0.809017}, {-0.162460, 0.262866, 0.951056}, {0.000000, 0.000000, 1.000000},
   {0.000000, 0.850651, 0.525731}, {-0.147621, 0.716567, 0.681718}, {0.147621, 0.716567, 0.681718},
   {0.000000, 0.525731, 0.850651}, {0.309017, 0.500000, 0.809017}, {0.525731, 0.000000, 0.850651},
   {0.295242, 0.000000, 0.955423}, {0.442863, 0.238856, 0.864188}, {0.162460, 0.262866, 0.951056},
   {-0.681718, 0.147621, 0.716567}, {-0.809017, 0.309017, 0.500000}, {-0.587785, 0.425325, 0.688191},
   {-0.850651, 0.525731, 0.000000}, {-0.864188, 0.442863, 0.238856}, {-0.716567, 0.681718, 0.147621},
   {-0.688191, 0.587785, 0.425325}, {-0.500000, 0.809017, 0.309017}, {-0.238856, 0.864188, 0.442863},
   {-0.425325, 0.688191, 0.587785}, {-0.716567, 0.681718, -0.147621}, {-0.500000, 0.809017, -0.309017},
   {-0.525731, 0.850651, 0.000000}, {0.000000, 0.850651, -0.525731}, {-0.238856, 0.864188, -0.442863},
   {0.000000, 0.955423, -0.295242}, {-0.262866, 0.951056, -0.162460}, {0.000000, 1.000000, 0.000000},
   {0.000000, 0.955423, 0.295242}, {-0.262866, 0.951056, 0.162460}, {0.238856, 0.864188, 0.442863},
   {0.262866, 0.951056, 0.162460}, {0.500000, 0.809017, 0.309017}, {0.238856, 0.864188, -0.442863},
   {0.262866, 0.951056, -0.162460}, {0.500000, 0.809017, -0.309017}, {0.850651, 0.525731, 0.000000},
   {0.716567, 0.681718, 0.147621}, {0.716567, 0.681718, -0.147621}, {0.525731, 0.850651, 0.000000},
   {0.425325, 0.688191, 0.587785}, {0.864188, 0.442863, 0.238856}, {0.688191, 0.587785, 0.425325},
   {0.809017, 0.309017, 0.500000}, {0.681718, 0.147621, 0.716567}, {0.587785, 0.425325, 0.688191},
   {0.955423, 0.295242, 0.000000}, {1.000000, 0.000000, 0.000000}, {0.951056, 0.162460, 0.262866},
   {0.850651, -0.525731, 0.000000}, {0.955423, -0.295242, 0.000000}, {0.864188, -0.442863, 0.238856},
   {0.951056, -0.162460, 0.262866}, {0.809017, -0.309017, 0.500000}, {0.681718, -0.147621, 0.716567},
   {0.850651, 0.000000, 0.525731}, {0.864188, 0.442863, -0.238856}, {0.809017, 0.309017, -0.500000},
   {0.951056, 0.162460, -0.262866}, {0.525731, 0.000000, -0.850651}, {0.681718, 0.147621, -0.716567},
   {0.681718, -0.147621, -0.716567}, {0.850651, 0.000000, -0.525731}, {0.809017, -0.309017, -0.500000},
   {0.864188, -0.442863, -0.238856}, {0.951056, -0.162460, -0.262866}, {0.147621, 0.716567, -0.681718},
   {0.309017, 0.500000, -0.809017}, {0.425325, 0.688191, -0.587785}, {0.442863, 0.238856, -0.864188},
   {0.587785, 0.425325, -0.688191}, {0.688197, 0.587780, -0.425327}, {-0.147621, 0.716567, -0.681718},
   {-0.309017, 0.500000, -0.809017}, {0.000000, 0.525731, -0.850651}, {-0.525731, 0.000000, -0.850651},
   {-0.442863, 0.238856, -0.864188}, {-0.295242, 0.000000, -0.955423}, {-0.162460, 0.262866, -0.951056},
   {0.000000, 0.000000, -1.000000}, {0.295242, 0.000000, -0.955423}, {0.162460, 0.262866, -0.951056},
   {-0.442863,-0.238856, -0.864188}, {-0.309017,-0.500000, -0.809017}, {-0.162460, -0.262866, -0.951056},
   {0.000000, -0.850651, -0.525731}, {-0.147621, -0.716567, -0.681718}, {0.147621, -0.716567, -0.681718},
   {0.000000, -0.525731, -0.850651}, {0.309017, -0.500000, -0.809017}, {0.442863, -0.238856, -0.864188},
   {0.162460, -0.262866, -0.951056}, {0.238856, -0.864188, -0.442863}, {0.500000, -0.809017, -0.309017},
   {0.425325, -0.688191, -0.587785}, {0.716567, -0.681718, -0.147621}, {0.688191, -0.587785, -0.425325},
   {0.587785, -0.425325, -0.688191}, {0.000000, -0.955423, -0.295242}, {0.000000, -1.000000, 0.000000},
   {0.262866, -0.951056, -0.162460}, {0.000000, -0.850651, 0.525731}, {0.000000, -0.955423, 0.295242},
   {0.238856, -0.864188, 0.442863}, {0.262866, -0.951056, 0.162460}, {0.500000, -0.809017, 0.309017},
   {0.716567, -0.681718, 0.147621}, {0.525731, -0.850651, 0.000000}, {-0.238856, -0.864188, -0.442863},
   {-0.500000, -0.809017, -0.309017}, {-0.262866, -0.951056, -0.162460}, {-0.850651, -0.525731, 0.000000},
   {-0.716567, -0.681718, -0.147621}, {-0.716567, -0.681718, 0.147621}, {-0.525731, -0.850651, 0.000000},
   {-0.500000, -0.809017, 0.309017}, {-0.238856, -0.864188, 0.442863}, {-0.262866, -0.951056, 0.162460},
   {-0.864188, -0.442863, 0.238856}, {-0.809017, -0.309017, 0.500000}, {-0.688191, -0.587785, 0.425325},
   {-0.681718, -0.147621, 0.716567}, {-0.442863, -0.238856, 0.864188}, {-0.587785, -0.425325, 0.688191},
   {-0.309017, -0.500000, 0.809017}, {-0.147621, -0.716567, 0.681718}, {-0.425325, -0.688191, 0.587785},
   {-0.162460, -0.262866, 0.951056}, {0.442863, -0.238856, 0.864188}, {0.162460, -0.262866, 0.951056},
   {0.309017, -0.500000, 0.809017}, {0.147621, -0.716567, 0.681718}, {0.000000, -0.525731, 0.850651},
   {0.425325, -0.688191, 0.587785}, {0.587785, -0.425325, 0.688191}, {0.688191, -0.587785, 0.425325},
   {-0.955423, 0.295242, 0.000000}, {-0.951056, 0.162460, 0.262866}, {-1.000000, 0.000000, 0.000000},
   {-0.850651, 0.000000, 0.525731}, {-0.955423, -0.295242, 0.000000}, {-0.951056, -0.162460, 0.262866},
   {-0.864188, 0.442863, -0.238856}, {-0.951056, 0.162460, -0.262866}, {-0.809017, 0.309017, -0.500000},
   {-0.864188,-0.442863, -0.238856}, {-0.951056,-0.162460, -0.262866}, {-0.809017, -0.309017, -0.500000},
   {-0.681718, 0.147621, -0.716567}, {-0.681718, -0.147621, -0.716567}, {-0.850651, 0.000000, -0.525731},
```

```
   {-0.688191, 0.587785, -0.425325}, {-0.587785, 0.425325, -0.688191}, {-0.425325, 0.688191, -0.587785},
   {-0.425325,-0.688191, -0.587785}, {-0.587785,-0.425325, -0.688191}, {-0.688197,-0.587780, -0.425327}
};
```

A whole frame has the following structure:

```
typedef struct {
  long type;      // 0 for byte-packed positions, and 2 for word-packed positions
  mdl_trivertx_t bboxmin,bboxmax;   // bounding box of the frame
  char name[16];  // name of frame, used for animation
  mdl_trivertx_t vertex[numverts];  // array of vertices, either byte or short packed
} mdl_frame_t;
```

The size of each frame is **sizeframe = 20 + (numverts+2) * sizeof(mdl_trivertx_t)**, while **mdl_trivertx_t** is either **mdl_trivertxb_t** or **mdl_trivertxs_t**, depending on whether the type is 0 or 2. In the MDL3 format the type is always 0. The beginning of the frames can be found in the .MDL file at offset **baseframes = basetri + numtris * sizeof(mdl_triangle_t)**.

MDL bones

This is for future expansion of the MDL format, and not supported yet.

Bones are a linked list of 3D vertices that are used for animation in the MDL5 format. Each bone vertex can have a parent, and several childs. If a bone vertex is moved, the childs move with it. If on moving a bone vertex the connection line to his parent rotates, it's childs are rotated likewise about the parent position. If the distance of the bone vertex to its parent changes, the change is added onto the distance between childs and parent. So the movement of the childs is done in a spherical coordinate system, it is a combination of a rotation and a radius change.
Each bone vertex has an influence on one or more mesh vertices. The mesh vertices influenced by a bone vertex move the same way as it's childs. If a mesh vertex is influenced by several bone vertices, it is moved by the average of the bone's movement.

The HMP5 terrain format

A terrain is basically a rectangular mesh of height values with one or several surface textures. It is a simplified version of the GameStudio Model format, without all the data structures that are unnecessary for terrain.

HMP file header

Once the file header is read, all the other terrain parts can be found just by calculating their position in the file. Here is the format of the .HMP file header:

```
typedef float vec3[3];

typedef struct {
  char version[4];   // "HMP4" or "HMP5"; only the newer HMP5 format is described here
  long nu1;          // not used
  vec3 scale;        // heightpoint scale factors
  vec3 offset;       // heightpoint offset
  long nu6;          // not used
  float ftrisize_x;  // triangle X size
  float ftrisize_y;  // triangle Y size
  float fnumverts_x; // number of mesh coordinates in X direction
  long numskins ;    // number of textures
  long nu8,nu9;      // not used
  long numverts;     // total number of mesh coordinates
  long nu10;         // not used
  long numframes;    // number of frames
  long nu11;         // not used
  long flags;        // always 0
  long nu12;         // not used
} hmp_header;
```

The size of this header is 0x54 bytes (84).

The "HMP4" format is used by the A5 engine prior to 5.230, while the new "HMP5" format is used by the A5 engine since version 5.230. The number of vertices in the rectangular mesh can be determined by

```
int numverts_x = (int) fnumverts_x;
int numverts_y = numverts/numverts_x;
```

After the file header follow the textures and then the array of height values.

HMP texture format

The terrain surface textures are flat pictures. There can be more than one texture. By default, the first texture is the terrain skin, and the second texture is the detail map if it has a different size. Further textures are not used yet. You will find the first texture just after the model header, at offset **baseskin = 0x54**. There are **numskins** textures to read. The texture and pixel formats are the same as for MDL skins, and are described in detail in the MDL format description.

HMP height values

A terrain contains a set of animation frames, which each is a set of height values. Normally only the first frame is used, because terrain does not animate. Each mesh vertex is defined by a height value and a normal.

```
typedef byte unsigned char;
typedef word unsigned short;
typedef struct {
  word  z;   // height value, packed on 0..65536
  byte  lightnormalindex; // index of the vertex normal
  byte  unused;       // not used
} hmp_trivertx_t;
```

To get the real Z coordinate from the packed coordinates, multiply it by the Z scaling factor, and add the Z offset. Both the scaling factor and the offset can be found in the `mdl_header` struct. Thus the formula for calculating the real height positions is:

```
float height = (scale[2] * z) + offset[2];
```

The X and Y position of the vertes results of the number of the vertex in the mesh, and thus must not be stored. The lightnormalindex field is an index to the actual vertex normal vector, just like in the MDL format description. A whole frame has the following structure:

```
typedef struct {
  long type;      // always 2
  mdl_trivertx_t bboxmin,bboxmax;   // bounding box of the frame - see mdl description
  char name[16];  // name of the frame, used for animation
  hmp_trivertx_t height[numverts];  // array of height values
} hmp_frame_t;
```