3D GameStudio

Programmer's Manual

for A5 engine 5.10

Johann C. Lotter / Conitec May 2001

Contents

The A5 DLL interface

DLLs can be used as extensions to the engine and to the WDL language, as well as for programming a game in VC++ instead of WDL. The DLL interface is available on all A5 editions. For creating an A5 DLL, the SDK (source development kit) and its DLL interface library is required. SDK owners can create arbitrary DLLs for adding new effects, actor AI or WDL instructions, and distribute or sell them to other 3D GameStudio users.

The Microsoft Visual C++ 6.0 development system is used for creating DLL extensions. The DLL SDK contains an interface library that must be linked to any DLL. An example VC++ project with a DLL template is also provided, which makes it easy to create extensions even for not-so-experienced C programmers who have never used DLLs before.

DLL extensions work bidirectionally: WDL instructions can access DLL functions, and DLL functions can access essential engine functions and variables. On opening a DLL, the engine transfers the pointer to an internal interface structure to the interface library. The interface contains pointers to engine variables and functions, like the frame buffer, the DirectX interface, the network interface, the DirectInput interface, the level, the WDL functions and so on. 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.

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. Also care must be taken that for writing something into the frame buffer it must be locked before, and unlocked afterwards. The interface library provides functions for that.

Getting started with the SDK

The SDK comes with a DLL **ackdll.dll** that contains just a few typical example functions. For testing one of those DLL functions, just copy the compiled **ackdll.dll** into the work folder, and insert the following WDL instructions into a function assigned to a key:

```
dll_open("ackdll.dll");
dll_exec("PaintScreenWhite",0);  // or any other DLL function
dll_close(dll_handle);
```

For creating a new DLL, just unzip the SDK into any directory and open it as a VC++ 6.0 project. The source code of **ackdll.dll** is included. It contains typical examples for all sorts of DLL functions. In the following, some typical DLL uses are described.

All exported DLL functions must be of type **DLLFUNC fixed func(long)** or **DLLFUNC fixed func(fixed)**, while **fixed** is a long integer interpreted by WDL as 22.10 fixed point value. The engine structs and functions accessible from DLL functions are described at the end of this chapers. All DLL functions can be accessed from WDL script through the **dll_open**, **dll_close**, **dll_exec**, and **dll_exec_vec** instructions that are described in the WDL manual.

Implementing new WDL functions

A DLL can contain a library of new arithmetic or other functions that can be accessed by WDL. The following example implements an exp function (which is already available in WDL) just for demonstration purpose:

```
// returns e power n
DLLFUNC fixed Exp(fixed var)
{
   return (FLOAT2FIX(exp(FIX2FLOAT(var))));
}
```

After having openend the DLL, the new function can be used this way:

```
x = dll_exec("Exp",y); // calculates x = e power y
```

Writing to the screen buffer

The following simple example shows how to lock the screen buffer, write into it and unlock it again. It paints the screen all white for one frame. This works in D3D as well as in 8-bit mode. From WDL, activate this function through **`dll_exec("PaintScreenWhite",0)`**. You'll see a short white flash when you call this function once. If you call it in a **`wait(1)`**-loop, the screen will become all white.

```
DLLFUNC fixed PaintScreenWhite (long unused)
{
// retrieve the pointer to the screen buffer
   FRAME_INTERFACE *a5fb = a5->fb;

// lock the screen buffer to get access to it
   (*a5fb->Lock)();

// paint it all white; note the use of a5fb->pitch here
   for (int j=0; j<a5fb->height; j++) {
      byte *buffer = a5fb->bytes + j*a5fb->pitch;
      for (int i=0; i<a5fb->width*a5fb->bpp; i++)
         *buffer++ = 255;
   }

// unlock it so that A5 can use it again
   (*a5fb->Unlock)();

   return 0;
}
```

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 **`IDirect3DDevice7`** pointer that is available through the DLL. For details refer to the DirectX documentation that is available, along with the DirectX 7 SDK, from the Microsoft site.

The example paints a multicolored triangle onto the screen. From WDL, activate this function through **`dll_exec("PaintD3DTriangle",0).`**You'll see the triangle briefly flashing in the upper left corner when you call this function once. If you call it in a **`wait(1)`**-loop, the triangle will be permanently on the screen. This code only works in 16- or 32-bit mode when Direct3D is activated.

```
#include <d3d.h> // from the DIRECTX7 sdk

DLLFUNC fixed PaintD3DTriangle (long unused)
{
// get the active D3D device
   FRAME_INTERFACE *a5fb = a5->fb;
   IDirect3DDevice7 *pd3ddev = (IDirect3DDevice7 *) a5fb->pd3ddev;
   if (!pd3ddev) return 0; // no D3D device in 8 bit mode

// define three corner vertices
   D3DTLVERTEX v[3];

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

   v[0].sz = v[1].sz = v[2].sz = 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 for more complicated operations)
   pd3ddev->SetRenderState(D3DRENDERSTATE_ALPHABLENDENABLE,FALSE);
   pd3ddev->SetTextureStageState(0,D3DTSS_COLORARG2,D3DTA_DIFFUSE);
   pd3ddev->SetTextureStageState(0,D3DTSS_COLOROP,D3DTOP_SELECTARG2);

// now draw the triangle
   pd3ddev->DrawPrimitive(D3DPT_TRIANGLEFAN,D3DFVF_TLVERTEX,(LPVOID)v,3,0);

// Normally we have to store the old render and texture states before, and
// set them back here... but the simple states above do no harm

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

Programming a game in VC++

Using the **`A4_ENTITY`** object (see below), a DLL can implement complex AI functions that would be harder to code in WDL. 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 WDL through **`dll_exec("FlipUpsideDown",my)`**. For controlling entities totally through a DLL – for instance, when you intend to write your whole game in VC++ instead of WDL – WDL dummy actions can be assigned to the entity, like this:

```
var appdll_handle;

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

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

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

WDL object structures

Pointers to WDL objects can be transferred to DLL functions, thus allowing object manipulation. The internal engine format of some important WDL objects is listed here (the structs are defined in the **`a5dll.h`** file).

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

typedef unsigned char byte;

typedef long fixed;  // fixed point 22.10 number format used by WDL
#define INT2FIX(i)   ((i)<<10)
#define FIX2INT(x)   ((x)>>10)
#define FIX2FLOAT(x) (((double)(x))/(1<<10))
#define FLOAT2FIX(f) ((fixed)((f)*(1<<10)))

//////////////////////////////////////////////////////////////////////
// internal A4 / A5 structs

typedef struct {
   long   index; // internal use only
   long   next;  // internal use only
   char   *name; // pointer to WDL name of the object
} WDL_LINK;      // WDL object header

typedef struct {
   WDL_LINK link;
   long   modelkey;  // internal use only
   fixed  x,y,z;     // position of the entity
   fixed  pan,tilt,roll;   // euler angles
   fixed  scale_x,scale_y,scale_z;    // scale factors, 0..255
   long   flags;     // entity property flags, see ENF...
   union {
   fixed  frame;     // frame number for sprites & models
   fixed  u;        // texture x offset for maps
   };
   union {
   fixed  nextframe; // interpolation target frame for models
   fixed  v;        // texture y offset for maps
   };
   fixed  skin;     // skin texture number for models
   fixed  ambient;  // -100..+100
   fixed  albedo;   // 0..100, light reflectivity
   fixed  alpha;    // 0..100, transparency, default 50
   fixed  lightrange;      // dynamic light range in quants
   fixed  red,green,blue;  // dynamic light color, 0..255
   long   emask;    // event enable flags, EVF...
   long   eflags;   // internal status flags
   fixed  min[3],max[3];   // bounding box in quants
   fixed  trigger_range;
   fixed  push;
   fixed  shadow_range;
   fixed  floor_range;
   long   client_id;    // client # that has created this entity
   fixed  skill[48];    // entity skills
} A4_ENTITY;

typedef struct {
   WDL_LINK link;
   char   *text;    // pointer to null terminated string
   long   length;   // allocated length of string (NEVER exceed!)
   long   flags;    // 0 = don't save, 1 = save string at SAVE/LOAD
} A4_STRING;

typedef struct {
   void   *pd3dsurf;    // pointer to the DirectDrawSurface7 (unlock before writing to it)
   long   finalwidth,finalheight;  // allocated size of the d3dsurf (not necessary the size of the bitmap!)
   byte   *pixels;  // pointer to the original pixels
   byte   *palette; // pointer to the original palette when 8 bit
   long   bitspp;   // original bitspp - 8, 16, 24, or 32
   long   width,height; // original size of the bitmap
   long   pitch;    // original size of a horizontal line
   long   flags;
} A4_TEX;

typedef struct {
   WDL_LINK link;
   long   width,height; // size of the bitmap
   long   bpp;      // internal bytes per pixel (1 or 2)
   long   flags;    // 0 = don't save, 1 = save images at SAVE/LOAD
   byte   *pixels8; // pointer to 8 bit image (color indexes)
   byte   *pixels16;    // NULL when palettized, otherwise ptr to 5-6-5 or 4-4-4-4 coded 16-bit image
   A4_TEX *tex;     // pointer to one or more textures the image is split into
} A4_BMAP;

typedef struct {
   WDL_LINK link;
   long   type;     // internal use only
   fixed  layer;    // layer number (read only)
   fixed  pos_x,pos_y;  // screen position in pixels
   long   flags;
   fixed  alpha;    // transparency factor
   fixed  offset;   // vertical offset
   fixed  lines;    // number of lines to display
   long   fontkey;  // internal use only
   long   strings;  // number of strings (read only)
   A4_STRING **pstrings;  // pointer to array of string pointers
} A4_TEXT;

typedef struct {
   WDL_LINK link;
   long   type;     // internal use only
   fixed  layer;    // layer number (read only)
   fixed  pos_x,pos_y;  // screen position in pixels
   long   flags;
   fixed  alpha;    // transparency factor
   A4_BMAP *bmap;   // background bitmap
} A4_PANEL;

typedef struct {
   WDL_LINK link;
   long   type;     // internal use only
   fixed  layer;    // layer number (read only)
   fixed  pos_x,pos_y;  // screen position in pixels
   long   flags;
   fixed  size_x,size_y;   // screen size in pixels
   fixed  x,y,z;    // position of the camera
   fixed  pan,tilt,roll;   // camera angles
   fixed  offset_x,offset_y;   // eye offsets
   fixed  arc;      // camera FOV, used for zooming
   fixed  aspect;   // width to height ratio
   fixed  ambient;  // brightness
   fixed  fog;      // fog strength
   fixed  diameter; // camera size for collision detection
   A4_ENTITY *genius;// gives valid BSP tree leaf for the view
} A4_VIEW;
```

DLL interface structures

Internal structs are handed over to the DLL for accessing internal variables and pointers initialized by the engine.

```
typedef struct {
   byte *bytes;  // pointer to frame buffer (only valid after Lock)
   int    pitch; // size of a horizontal line in bytes (!= width!)
   int    width,height; // width, height of the screen
   int    bpp;          // bytes per pixel of the frame buffer

   BOOL (*Lock)(void);  // lock frame buffer before accessing it
   void (*Unlock)(void); // unlock frame buffer after accessing it
   void   *pd3ddev;     // pointer to the IDirect3DDevice7 initilized by A4/A5
} FRAME_INTERFACE;

typedef struct {
   void *pdi;       // pointer to the IDirect3DDevice7 used by A4/A5
   void *pdimouse;  // pointer to the mouse DirectInputDevice
   void *pdikbd;    // always zero (keyboard doesn't use DirectInput)
   void *pdijoy;    // always zero (joystick doesn't use DirectInput)
   void *pdplay;    // pointer to the DirectPlay4 interface
   void *pdplobby;  // pointer to the DirectPlayLobby2A interface
   DWORD pdplayer;  // player ID
   void *pdpguid;   // multiplayer session GUID
   void *pds;       // pointer to the DirectSound interface
   void *pdsb;      // pointer to the DirectSoundBuffer interface
} DX_INTERFACE;

typedef struct {
   long (*GetObj)(char *name); // internal use only
   long (*GetFunc)(char *name);// internal use only
   A4_ENTITY **my,**you;       // Pointer to MY, YOU entity pointers
} WDL_INTERFACE;


typedef struct {
   long dll_version;
// The version is automatically tested against A5DLL_VERSION
// on opening the DLL. DLLs work with engines with the same or a higher
// version number, but not with a lower version engine.
```

```
   WDL_INTERFACE    *wdl;  // access to MY and YOU pointers
   FRAME_INTERFACE  *fb;   // access to the frame buffer and the Direct3D Device
   DX_INTERFACE     *dx;   // access to directx pointers
} A5_INTERFACE;
```

An **A5_INTERFACE** object named **a5** is initialized on startup of each DLL and can be used for accessing the screen buffer, the Direct3D Device and the **MY** and **YOU** entities. For directly accessing any WDL object from a DLL, the **a5dll.lib** can be used in a way described in the following chapter.

DLL functions

Two functions in the **a5dll.lib** provide DLL access to internal WDL variables, objects and WDL script functions. This way, entity AI can be implemented in a DLL plugin, and can use most WDL script instructions.

long a5dll_getwdlobj(char *name);

This function returns the address of the WDL object or variable with the given name. It can be used to read or write any defined WDL 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 WDL objects:

```
// adds the given value to the sky speed
fixed AddToSkySpeed(fixed value)
{
// get the address of the variable
  fixed *skyspeed = (fixed *)a5dll_getwdlobj("sky_speed");
  if (!skyspeed) return 0;

// add the value to both the x and y components
  skyspeed[0] += value;  // skyspeed X value
  skyspeed[1] += value;  // skyspeed y value

  return INT2FIX(1);
}

// zooms the camera view
fixed ZoomIn(fixed value)
{
  A4_VIEW *camera = (A4_VIEW *)a5dll_getwdlobj("camera");
  if (!camera) return 0;
  return (camera->arc -= value); // change the FOV and return it
}
```

long a5dll_getwdlfunc(char *name);

This function returns the address of the WDL function with the given name. It can be used to execute WDL functions from inside a DLL plugin. Not all WDL functions are available for DLLs. If the function is not available (as can be the case for some special WDL functions, like **wait()** or **inkey()**), **NULL** is returned and an error message will pop up. Example for an entity AI DLL function that uses WDL functions for scanning the environment of an entity:

```
// returns free distance in front of MY entity until next obstacle
fixed DistAhead(long my)
{
  if (!my) return 0;
// retrieve the pointer to the given entity
  A4_ENTITY *ent = (A4_ENTITY *)my;

// get the address of some wdl variables and functions
  fixed *tracemode = (fixed *)a5dll_getwdlobj("trace_mode");
  wdlfunc2 vecrotate = (wdlfunc2)a5dll_getwdlfunc("vec_rotate");
  wdlfunc2 trace = (wdlfunc2)a5dll_getwdlfunc("trace");
  if (!tracemode || !trace || !vecrotate) return 0;

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

// rotate vector by entity engles, just as in WDL
  (*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);
}
```

Global WDL functions, like level loading and keyboard entry, can not be called directly from a DLL. However they can be executed indirectly from a WDL script that calls a DLL function for deciding which operation must be executed. Example:

```
function main_loop {
   while(1) {
      dll_handle = appdll_handle;
      operation = dll_exec("choose_operation",0);  // this function returns a control code
      if (operation == 1) { load_level(<level1.wmb>); }
      if (operation == 2) { load_level(<level2.wmb>); }
      if (operation == 10) { inkey(entry_string); }
      wait(1);
   }
}
```

The A5 Client/Server Protocol

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

```
String = a sequence of characters, terminated by NULL ('\0')
Angle = a short, to be multiplied by 360.0/65535.0 to convert it to degrees.
Position – a position packed in three bytes by dividing it by 8.
Byte   = an unsigned integer, on one byte.
Scale(x) = a value packed into one byte to be multiplied by x/255.0.
Short  = a signed integer, on two bytes, Big Endian order (Intel order).
Long  = a signed integer, on four bytes, Big Endian order (Intel order).
Fixed = a floating point number, on four bytes, Big Endian order (Intel
order).
```

Client Messages

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

Command	*Bytecode*	*Arguments*	*Description*
CLS_JOIN	**0x02**		Request for joining the session
CLS_CREATE	**0x03**		Create entity with given model name, and link client to it
CLS_REMOVE	**0x04**		Remove entity on server
CLS_PING	**0x07**		Sent after each client frame
CLS_LEAVE	**0x08**		Leave the session
CLS_LEVEL	**0x09**		Client has loaded a level
CLS_VAR	**0x0a**		Send a variable
CLS_STRING	**0x0b**		Send a string
CLS_SKILL	**0x0e**		Send an entity skill
CLS_SKILL3	**0x0f**		Send an entity vector skill

Server Messages

The following commands are used for transferring information from the server to either a specific client, or to all clients connected:

Command	*Bytecode*	*Arguments*	*Description*
SVC_CREATE	**0x03**		Created entity with given index
SVC_REMOVE	**0x04**		Removed entity from server
SVC_ENTSOUND	**0x05**		Play an entity sound on the clients
SVC_PARTICLE	**0x06**		Generate a particle effect on the clients
SVC_INFO	**0x07**		Send the server time to the clients
SVC_LEAVE	**0x08**		Server goes down

Command	*Bytecode*	*Arguments*	*Description*
`SVC_VAR`	`0x0a`		Send a variable to all clients
`CLS_STRING`	`0x0b`		Send a string to all clients
`SVC_SKILL`	`0x0e`		Send an entity skill to the entity's client
`SVC_SKILL3`	`0x0f`		Send an entity vector skill to the entity's client
`SVC_UPDATE1`	`0x40`		Update entity parameters 1
`SVC_UPDATE2`	`0x80`		Update entity parameters 2
`SVC_UPDATE3`	`0xc0`		Update entity parameters 3

The MDL model 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 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" or "MDL4"
  long final;        // not used yet
  vec3 scale;        // 3D position scale factors.
  vec3 offset;       // 3D position offset.
  float pad;         // not used yet.
  vec3 eye;          // not used yet.
  long numskins ;    // number of skin textures
  long skinwidth;    // width of skin texture; must be a multiple of 2
  long skinheight;   // height of skin texture
  long numverts;     // number of 3d wireframe vertices
  long numtris;      // number of triangles surfaces
  long numframes;    // number of frames
  long numskinverts; // number of 2D skin vertice
  long flags;        // 0 = normal, 1 = terrain model
  long numbones;     // number of bone vertices (MDL4 only, otherwise 0)
} mdl_header;
```

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

The "MDL3" format is used by the A4 engine, while the "MDL4" format is used by the A5 engine. After the file header follow the skins, the skin vertices, the triangles, the frames, and finally the bones (future expansion).

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 structure 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;
```

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 are a table of 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 are represented as a table of shorts with 4 bits for each of the alpha, red, green, and blue component.

The width of skins should be a multiple of 4, to ensure long word alignement. The skin pictures are usually made of as many pieces as there are independent parts in the model. For instance, for the a player, there may be two pieces that defines the body, and two others that define the gun.

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, then skinsize = **(4 + skinwidth * skinheight * 2) * numskins**.

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  boneindex;        // index of the bone this vertex belongs to
} 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 only for future expansion of the MDL format, and not implemented yet.

Bones are a linked list of 3D vertices that are used for animation in the MDL4 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.