TR WAD FILE FORMAT

Document Version 2.1
By IceBerg, Lisbon, Portugal, European Union
March 31, 2005

Based on Document Version 1.0
By Turbo Pascal, Tegucigalpa, Honduras, Central America
December 11, 2001

Disclaimer

Blank page

Foreword to version 2.1

Mainly correcting typing errors. This document was downloaded much more then I expected, so I decided to give it a general clean-up... ☺

IceBerg
Lisbon, Portugal
2005-03-31

Foreword to version 2.0

When I received an invitation from Turbo Pascal to become the developer of StrPix and the carrier of "the torch", my first reaction was of pleasure and honour.
My second reaction was of panic!
StrPix is widely used in the official TR community, developing a next version was a task of great responsibility. And I knew nothing about TR4 geometry or about the structure of the WAD files. And there was very little information about any of these subjects.

Fortunately, Turbo Pascal had published on-line a draft of the WAD file format, some years ago. Written back in late 2001, that draft developed only the sections used by Turbo Pascal in his Strpix3. Several other sections were not fully explained at the time. So I made some research and experiments with WAD files, went over the document and completed or up-dated where needed.

This document is targeted to programmers and reflects my opinion about the internal structure of the WAD file. There may be errors or misinterpretations in my research, so please use with caution and please do report your own findings at the EZBoard Forum where most Tomb Raider researchers can be found (http://pub84.ezboard.com/ftreditingzonefrm2). Please do not hesitate in making an addenda or errata to this document.

Thanks to all those researchers who have been contributing with bits and pieces of information, here and there, about the TR file formats. I've been taking notes ☺

A very special thanks to Turbo Pascal. For your friendship and for sharing your findings with the TR community. Without your first document this revision which I'm now proposing would have never been started.

By publishing this new version of the TR Wad File Format document, I'm trying to give back to the community some of the precious help I've been receiving from it.
May this effort be useful to other developers.

IceBerg
Lisbon, Portugal
2005-03-03

Blank page

Introduction

This document was split in two chapters: “Quick Reference” and “Exploring the WAD File”. The contents are basically the same, but “Quick Reference” is exactly what it states, and “Exploring…” has a lot of extra comments and illustrations.

A new nomenclature, following recent orientations in the computer industry, is used to describe some integer and real variable types[1]:

sint8	: **S**igned **8**-bits	[-128..127]
sint16	: **S**igned **16**-bits	[-32768..32767]
sint32	: **S**igned **32**-bits	[-2147483648..2147483647]
uint8	: **U**nsigned **8**-bits	[0..255]
uint16	: **U**nsigned **16**-bits	[0..65535]
uint32	: **U**nsigned **32**-bits	[0..4294967295]

single : single-precision floating point **32**-bits
double : double-precision floating point **64**-bits

In the text, used occasionally, references may be made to the traditional **bytes**, **words** and **dwords**, respectively **8**, **16** and **32** bit unsigned integers, mostly to define sizes.

Other changes from tradition:
The description of the Collision Sphere as a Bounding Sphere; renaming the Frames data as Keyframes data; renaming the Mesh Trees as Pivot Links Data and associating them and the Keyframes data with the Animation section instead of the Movables section; renaming the lighting schemes as Normals/Shades; renaming a lot of fields in the Animation Section for clarity.

Softwares:
Turbo Pascal’s StrPix3.95R11, RView5.0R3, FexInspect1.0R0, FexAnim1.1; IceBerg’s HexDump; TRWad’s WadMerger1.95A12; Popov’s TRViewer1.083; Raider Croft’s PixStr2.2R2; were used together as research tools to help me understanding the WAD / TR4 file formats.

Bibliography:
Turbo Pascal’s original version of this document; his source code for StrPix3; his source code for TRUnit3 and other related sources; the original TRosettaStone from 1999 and the revision by Popov in 2003, which includes references to the TR4 file format; explanations and comments published in discussion forums like EZBoard and Lara’s Home; also Michael Jelarcic’s manual for TRwest; TRWad/Michiel’s manual for the Animation Editor in WADMerger.

[1] *See the “Discussion Topic 1 – Variables Nonsenseclature” for my comments on this subject.*

Blank page

INDEX

Blank page

TR WAD FILE FORMAT

Quick Reference

Exploring the WAD File

Discussion Topics

Blank page

QUICK REFERENCE

Blank page

The Tomb Raider WAD file format

Section 1 – Version

File_ID (**uint32**).
A valid Tomb Raider WAD file has a value of **129** in its **File_ID** field.

Section 2 – Textures

Num_Texture_Samples (**uint32**).
Number of texture samples listed in the **Texture_Samples_Table**.

Texture_Samples_Table (**Num_Texture_Samples * 8 bytes**).
Records containing the position, size and attitude of each texture sample stored in the WAD file. Each texture sample is defined by a rectangle whose **anchor corner** is located at its the top-left pixel. Description:

x	(**uint8**)	anchor corner x pixel position.
y	(**uint8**)	anchor corner y pixel position.
page	(**uint16**)	page where the texture sample is stored.
flipX	(**sint8**)	horizontal flip, yes or no, **-1** or **0**.
addW	(**uint8**)	number of pixels to add to the width.
flipY	(**sint8**)	vertical flip, yes or no, **-1** or **0**.
addH	(**uint8**)	number of pixels to add to the height.

Texture_Num_Bytes (**uint32**).
The texture samples are packed into pages packed in a single Texture Map, whose total size in **bytes** is given by **Texture_Num_Bytes** .

Texture_Map_Package (**Texture_Num_Bytes * 1 byte**).
The Texture Map itself is stored as a standard RAW 24bits [R G B] pixel file.
Some dimensions of the texture map are known by default: the width is always 256 pixels, the pixel format takes 3 bytes per pixel. The height of the texture map and the number of pages need to be calculated.

number_of_pages = **Texture_Num_Bytes div** (256 * 256 * 3)

map_height = **Texture_Num_Bytes div** (256 * 3)

Section 3 – Meshes

Num_Mesh_Pointers (**uint32**).
Number of pointers stored in the **Mesh_Pointers_List**.

Mesh_Pointers_List (**Num_Mesh_Pointers * 4 bytes**).
Each record in this list is an offset to a mesh located in the **Mesh_Data_Package**. Description:

offset	(**uint32**)	offset of a mesh in the meshes package.

Meshes_Num_Words (**uint32**).
Size of the **Meshes_Data_Package**, expressed in **word** (**uint16**) units.

Meshes_Data_Package (**Meshes_Num_Words * 2 bytes**).
The mesh data package stores the meshes all together, as a single package. The individual meshes in the package vary in size, but they all have the same internal structure:

Bounding_Sphere (**10 bytes**).
Coordinates of the centre, and the radius, of a bounding sphere used for proximity testing in-game. Used by Movable Models only, Static Models have zeros in these fields. Description:

cx	(**sint16**)	centre's coordinate in x.
cy	(**sint16**)	centre's coordinate in y.
cz	(**sint16**)	centre's coordinate in z.
radius	(**uint16**)	radius of the sphere.
unk	(**uint16**)	unknown.

Num_Vertices (**uint16**).
Number of vertices in the mesh.

Vertices_Table (**Num_Vertices * 6 bytes**).
Table storing the XYZ coordinates of the vertices. Description:

vx	(**sint16**)	vertex coordinate in x.
vy	(**sint16**)	vertex coordinate in y.
vz	(**sint16**)	vertex coordinate in z.

Num_Normals / Num_Shades (**sint16**).
The value stored here has a different meaning depending on its sign. If **positive**, it means **Num_Normals**, if **negative** it means **Num_Shades**.

Normals_Table (**Num_Normals * 6 bytes**).
Table with the XYZ lengths of the normal vectors attached to the vertices of the mesh. Description:

lx	(**sint16**)	vector length in x.
ly	(**sint16**)	vertex length in y.
lz	(**sint16**)	vertex length in z.

To “normalize” this vector we need to divide each component by 16300.

nx	(**single**)	normalized vector length in x	= **lx** / 16300.
ny	(**single**)	normalized vector length in y	= **ly** / 16300.
nz	(**single**)	normalized vector length in z	= **lz** / 16300.

Shades_List (**Num_Shades * 2 bytes) * (-1)**.
The (minus one) is to turn positive a number that is stored as negative.
List of values representing a light intensity, a shade of grey affecting the luminosity or the darkness of its correspondent vertex. Description:

shade	(**sint16**)	shade of grey for the vertex.

grey_intensity = 255 - **shade** * 255 / 8191

Num_Polygons (**uint16**).
Number of polygons in the mesh.

Polygons_Package (**Num_Polygons * variable bytes**).
We know the number of polygons, but that does not tell us the size of the polygons data package because the WAD file stores triangles and quads all mixed up. The structure of each polygon record is the following:

shape	(**uint16**)	a triangle, or a quad.
v1	(**uint16**)	anchor vertex index.
v2	(**uint16**)	next vertex index.
v3	(**uint16**)	next vertex index.
v4	(***uint16***)	*next, only if quad*.
texture	(**uint16**)	index and horizontal flip.
attributes	(**uint8**)	opacity and shine.
unk	(**uint8**)	unused byte.

The **shape** field tells us if the polygon is a triangle (8) or a quad (9).

The **texture** word, treated as a bit field, has the following meaning:

flipped	**texture and $8000**	(**1 bit**)	horizontal flipping.
shape	**texture and $7000**	(**3 bits**)	texture sample shape.
index	**texture and $0FFF**	(**12 bits**)	texture sample index.

If bit [15] is set, as in (**texture and $8000**), it indicates that the texture sample must be **flipped** horizontally prior to be used.

Bits [14..12] as in (**texture and $7000**), are used to store the texture **shape**, given by: (**texture and $7000**) **shr 12**.
The valid values are: **0**, **2**, **4**, **6**, **7**, as assigned to a square starting from the top-left corner and going clockwise: **0**, **2**, **4**, **6** represent the positions of the square angle of the triangles, **7** represents a quad.

Bits [11..0] as in (**texture and $0FFF**), are used to store a texture **index** to be used on the **Texture_Samples_Table** to find the sample's bitmap.

The **attributes** byte, treated as a bit-field, stores the following data:

unk	**attributes and $80**	(**1 bit**) not used?
intensity	**attributes and $7C**	(**5 bits**)shine effect intensity.
shine	**attributes and $02**	(**1 bit**) shine effect flag.
opacity	**attributes and $01**	(**1 bit**) opacity mode.

Bit [7] as in (**attributes and $80**), apparently is not used. I've noticed that, when forced to be set, it disables the shine effect.

Bits [6..2] as in (**attributes and $7C**), are used to store the **intensity** of the shine effect. The highest the value, the more intense the effect is. A field with 5 bits can store 32 values in the range [0..31].

If bit [1] is set, as in (**attributes and $02**), the **shine** effect is on.

Bit [0] as in (**attributes and $01**), is used to flag the **opacity** mode.
If this field is not set (bit value = 0) the texture is considered opaque and the **magenta colour** is used to mark pixels that are to be made fully transparent.
If this field is set (bit value = 1), then colours are treated as translucent, after converting the magenta colour, if any, to black.

Padding (**uint16**).
This field only exists if the number of quads is odd (not a multiple of 2) as given by the logical test:

(number_of_quads **mod** 2) = 1

If this test is TRUE the remaining of the division by 2 is different from zero (or it is equal to one, the only other possibility), meaning that the number is **odd**. If so, a padding word must be added.
If the test is FALSE the remaining of the division by 2 is zero, meaning that the number is **even**, and no padding is added.

And this completes the parsing of a mesh.

Section 4 – Animations

Num_Animations (**uint32**).
Number of animations stored in the **Animations_Table**.

Animations_Table (**Num_Animations * 40 bytes**).
This table stores all the animations for all the Movable Models in the WAD file. These *animations* are in fact segments of the model's *actions*.

keyframeOffset	(**uint32**)	offset in **Keyframes_Data_Package**.
frameDuration	(**uint8**)	engine ticks per frame.
keyframeSize	(**uint8**)	size of the keyframe record, in words.
state_ID	(**uint16**)	ID of the state of this animation.
unknown1	(**sint16**)	unknown 2 bytes.
speed	(**sint16**)	ground speed.
acceleration	(**sint32**)	easy-in and easy-out for the speed.
unknown2	(**sint64**)	unknown 8 bytes.
frameStart	(**uint16**)	[frame-in] index of this animation.
frameEnd	(**uint16**)	[frame-out] index of this animation.
nextAnimation	(**uint16**)	index of the default next animation.
frameIn	(**uint16**)	[frame-in] index of the next animation.
numStateChanges	(**uint16**)	number of animation transitions.
changesIndex	(**uint16**)	index in **State_Changes_Table**.
numCommands	(**uint16**)	number of commands.
commandOffsets	(**uint16**)	offset in **Commands_Data_Package**.

Each animation segment has a certain number of frames, every segment starts at a given frame number, **frameStart**, and stops at a given frame number, **frameEnd**. The number of frames in the animation segment, including *keyframes* and *interpolated*, is given by:

number_of_frames = **frameEnd** - **frameStart** + 1

The number of keyframes-only is *not* stored in the **Animations_Table**. It needs to be deduced from the **keyframeOffset** of the current animation segment, from the **keyframeOffset** of the next segment, and from the **keyframeSize** of the current segment converted to bytes.

num_keyframes = (keyframeOffset[i+1] - keyframeOffset[i]) **div** (2 * keyframeSize[i])

The last animation, the last keyframeOffset, has no next keyframeOffset, so the total size of the **Animations_Table** must be used in the equation above as a "next offset".

Num_State_Changes (**uint32**).
Number of records stored in the **State_Changes_Table**.

State_Changes_Table (**Num_State_Changes * 6 bytes**).
This table determines all the possible transitions between animations of different families, built in by the designer of the game. Description:

state_ID	(**uint16**)	ID of the state of the next animation.
numDispatches	(**uint16**)	number of animation dispatches.
dispatchesIndex	(**uint16**)	index in the dispatches table.

Num_Dispatches (**uint32**).
Number of records stored in the **Dispatches_Table**.

Dispatches_Table (**Num_Dispatches * 8 bytes**).
This table stores all the possible transitions between animations of different families, built in by the designer of the game. Description:

inRange	(**uint16**)	[frame-in] where this range starts, inclusive.
outRange	(**uint16**)	] frame-out [where this range stops, exclusive.
nextAnim	(**uint16**)	index of the next animation in the **Animations_Table**.
frameIn	(**uint16**)	[frame-in] index of the next animation.

Commands_Num_Words (**uint32**).
Size of the **Commands_Data_Package**, expressed in **word** (**uint16**) units.

Commands_Data_Package (**Commands_Num_Words * 2 bytes**).
This package stores all the commands for all the animation segments in the **Animations_Table**. It needs to be parsed. The structure of each command record is the following:

command	(**uint16**)	command's code.
operator1	(***uint16***)	*first operator, if applicable.*
operator2	(***uint16***)	*second operator, if applicable.*
operator3	(***uint16***)	*third operator, if applicable.*

The "*if applicable*" is there precisely because the record has a variable size. Some commands have no operators, some have two or three.

Links_Num_DWords (**uint32**).
Number of integers (**sint32**) in the **Links_Data_Package**.

Links_Data_Package (**Links_Num_DWords * 4 bytes**).
The integers in the package are organized in records, each record being a **Pivot Link** consisting of four **sint32** integers describing the *hierarchy* and the *relative offsets* of the *pivot points* for a 3D model. Description of each record:

opCode	(**sint32**)	stack operation code.
dx	(**sint32**)	mesh offset in x.
dy	(**sint32**)	mesh offset in y.
dz	(**sint32**)	mesh offset in z.

The **opCode** takes the values **0**, **1**, **2**, **3**, where:
0 = stack not used. Link the current mesh to the previous mesh.
1 = pull the parent from the stack. Link the current mesh to the parent.
2 = push the parent into the stack. Link the current mesh to the parent.
3 = read the parent in the stack. Link the current mesh to the parent.

Keyframes_Num_Words (**uint32**).
Size of the **Keyframes_Data_Package**, expressed in **word** (uint16) units.

Keyframes_Data_Package (**Keyframes_Num_Words * 2 bytes**).
This package stores all the bounding boxes, root mesh offsets and pivot angles for all the animation segments in the **Animations_Table**. Description:

bb1x	(**sint16**)	coordinate, bounding box.
bb2x	(**sint16**)	coordinate, bounding box.
bb1y	(**sint16**)	coordinate, bounding box.
bb2y	(**sint16**)	coordinate, bounding box.
bb1z	(**sint16**)	coordinate, bounding box.
bb2z	(**sint16**)	coordinate, bounding box.
offx	(**sint16**)	coordinate, root mesh offset.
offy	(**sint16**)	coordinate, root mesh offset.
offz	(**sint16**)	coordinate, root mesh offset.
keys	(**variable words**)	package of pivot point angles.

The pivot point angles are coded as sets of three rotations [rotateX, rotateY, rotateZ]. In some cases only one axis is rotated, the other rotations being zero [rotateX, 0, 0] or [0, rotateY, 0] or [0, 0, rotateZ]. In all, there are four different possibilities. If the angle set specifies a three-axes rotation, a **uint32** is used to store the set (angle_set = word * $10000 + next_word). If the angle set specifies a one-axis rotation, a **unit16** is used. To detect which is the case, while parsing the angles package, test the value of axes = angle_set **and** $C000.
if axes = $0000 then it is a three-axes rotation and the lower 30 bits of the **uint32** code the 3 rotations, at 10 bits per rotation. If axes = $4000 then it is a rotateX, if axes = $8000 then it is a rotateY, if axes = $C000 then it is a rotateZ. The rotation value is stored in the lower 14 bits of the **unit16**. Having extracted the rotation values, their values in degrees are obtained through the following conversions:
For a three-axes rotation a value of 1024 is equivalent to 360 degrees.
For a one-axis rotation a value of 4096 is equivalent to 360 degrees.

Section 5 – Models

Num_Movables (**uint32**).
Number of Movable Models stored in the **Movables_Table**.

Movables_Table (**Num_Movables * 18 bytes**).
This table is the entry point to access the Movable Models in the WAD. The indexes and offsets it contains are used to fetch data from the other tables and packages. Description:

obj_ID	(**uint32**)	unique ID number for this Movable.
numPointers	(**uint16**)	number of mesh pointers.
pointersIndex	(**uint16**)	index to the pointers list.
linksIndex	(**uint32**)	index to the pivot point links package.
keyframeOffset	(**uint32**)	offset in the keyframes package.
animsIndex	(**sint16**)	index in the animations table.

Num_Statics (**uint32**).
Number of Static Models stored in the **Statics_Table**.

Statics_Table (**Num_Statics * 32 bytes**).
This table is the entry point to access the Static Models in the WAD. Description:

obj_ID	(**uint32**)	unique ID number for this Static.
pointersIndex	(**uint16**)	index of a pointer to the mesh.
vx1	(**sint16**)	coordinate, visibility bounding box.
vx2	(**sint16**)	coordinate, visibility bounding box.
vy1	(**sint16**)	coordinate, visibility bounding box.
vy2	(**sint16**)	coordinate, visibility bounding box.
vz1	(**sint16**)	coordinate, visibility bounding box.
vz2	(**sint16**)	coordinate, visibility bounding box.
cx1	(**sint16**)	coordinate, collision bounding box.
cx2	(**sint16**)	coordinate, collision bounding box.
cy1	(**sint16**)	coordinate, collision bounding box.
cy2	(**sint16**)	coordinate, collision bounding box.
cz1	(**sint16**)	coordinate, collision bounding box.
cz2	(**sint16**)	coordinate, collision bounding box.
flags	(**uint16**)	some flags.

Blank page

EXPLORING THE WAD FILE

Blank page

The Tomb Raider WAD file format

The WAD file stores 3D objects, together with their textures and animations, to be used with the official Tomb Raider Level Editor (TRLE). Whereas the TR4 is a *playable* file format, the WAD is a *storage* file format[2].

For the purposes of this document, the WAD file was divided in 5 main sections, representing the different logical groups contained in this file format:

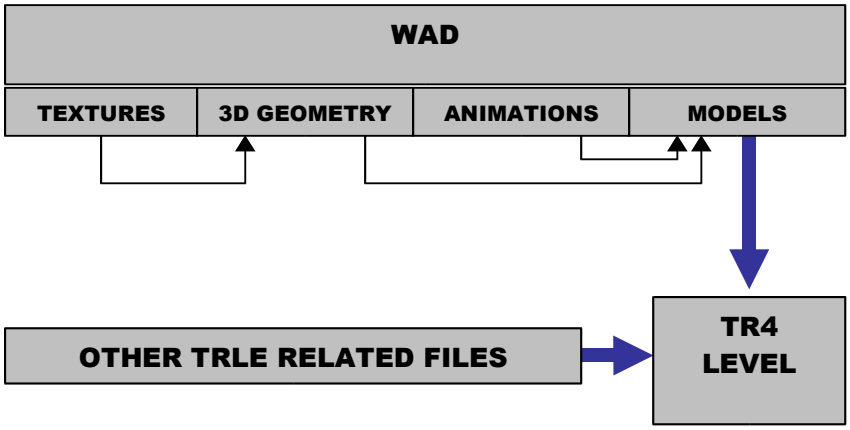
Section 1 – Version

Section 2 – Textures

Section 3 – Meshes

Section 4 – Animations

Section 5 – Models



[2] *Other formats are involved in building up a playable TR4 file: *.CD for soundtracks, *.LAR for Lara's animations, *.SAM and *.SFX for sound files, *.SWD for image files, eventually some *.BMP, *.TGA and *.TXT, there may be *.RAW for sky graphics, *.PRJ for TRLE projects, *.TOM for rendered projects.*
*The *.WAS file contains names for the models in the respective *.WAD file. This file is not needed for TRLE despite of apparently being requested. The WAD file can be loaded directly without the WAS file.*
On the other hand, the OBJECTS.H file is needed for TRLE to work.

Blank page

Section 1 – Version

File_ID (**uint32**).
A valid Tomb Raider WAD file has a value of **129** in its **File_ID** field[3].

FIG. 1 - An exploratory application was developed while the WAD file was being studied. This application grew up as the study progressed and suffered several modifications during the process.

[3] *This value of 129 is not exclusive of *.WAD files. The *.SWD files also have this value in their own **file_id** fields. This value of 129 is in fact the meshes version, OBJECT_VERSION, defined in the OBJECTS.H file that goes with TRLE.*

Blank page

Section 2 – Textures

Num_Texture_Samples (**uint32**).
Number of texture samples listed in the **Texture_Samples_Table**.

Texture_Samples_Table (**Num_Texture_Samples * 8 bytes**).
Each record in the sample's table contains the position, size and attitude of each texture sample stored in the WAD file. Each texture sample is defined by a rectangle whose **anchor corner** is located at its top-left pixel. Description:

x	(**uint8**)	anchor corner x pixel position.
y	(**uint8**)	anchor corner y pixel position.
page	(**uint16**)	page where the texture sample is stored.
flipX	(**sint8**)	horizontal flip, yes or no, **-1** or **0**.
addW	(**uint8**)	number of pixels to add to the width.
flipY	(**sint8**)	vertical flip, yes or no, **-1** or **0**.
addH	(**uint8**)	number of pixels to add to the height.

The sample's (**x**, **y**) pixel coordinates are relative to the top-left corner the **page** where that sample is stored.

The width of a texture sample is given by (width = 1 + **addW**), which includes one pixel for **x** plus the additional pixels **addW**. The height of a texture sample, including a pixel for **y,** is given by (height = 1 + **addH**).

Taking as an example the sample 0001 in the table below, we can see that its (**x**, **y**), its **anchor corner**, is located at (192, 194) of page 0. The values of 15 under **addW** and **addH** are not the width and the height of the texture sample, which are in fact 16, as explained above. The values of -1 under **flipX** and **flipY** indicate that the sample must be flipped before being mapped[4].

sample	x	y	page	flipX	addW	flipY	addH
0000	0	0	0	0	0	0	0
0001	192	194	0	-1	15	-1	15
0002	32	194	0	-1	23	-1	15
0003	12	180	0	-1	29	-1	13
0004	192	194	0	-1	15	-1	15
0005	192	194	0	-1	15	-1	15
0006	136	224	0	-1	7	-1	15
0007	136	224	0	-1	7	-1	15
0008	136	224	0	-1	7	-1	15
0009	56	180	0	-1	5	-1	13

FIG. 2 - Interesting that first line, ALL ZEROS! What is it for? Most likely this is just a NULL texture placeholder.

[4] *More about this in pages 13 and 14.*

The page-relative coordinates can be converted into map-relative coordinates in order to locate the texture sample in the **Texture_Map**. The conversion is given by:

mapX = **x**
mapY = **y** + 256 * **page**

In its own page, the sample's bounding rectangle can be expressed by:

Rect(**x**, **y**, **x** + (1 + **addW**), **y** + (1 + **addH**))

Relative to the **Texture_Map**, the sample's bounding rectangle can be expressed by:

Rect(mapX, mapY, mapX + width, mapY + height)

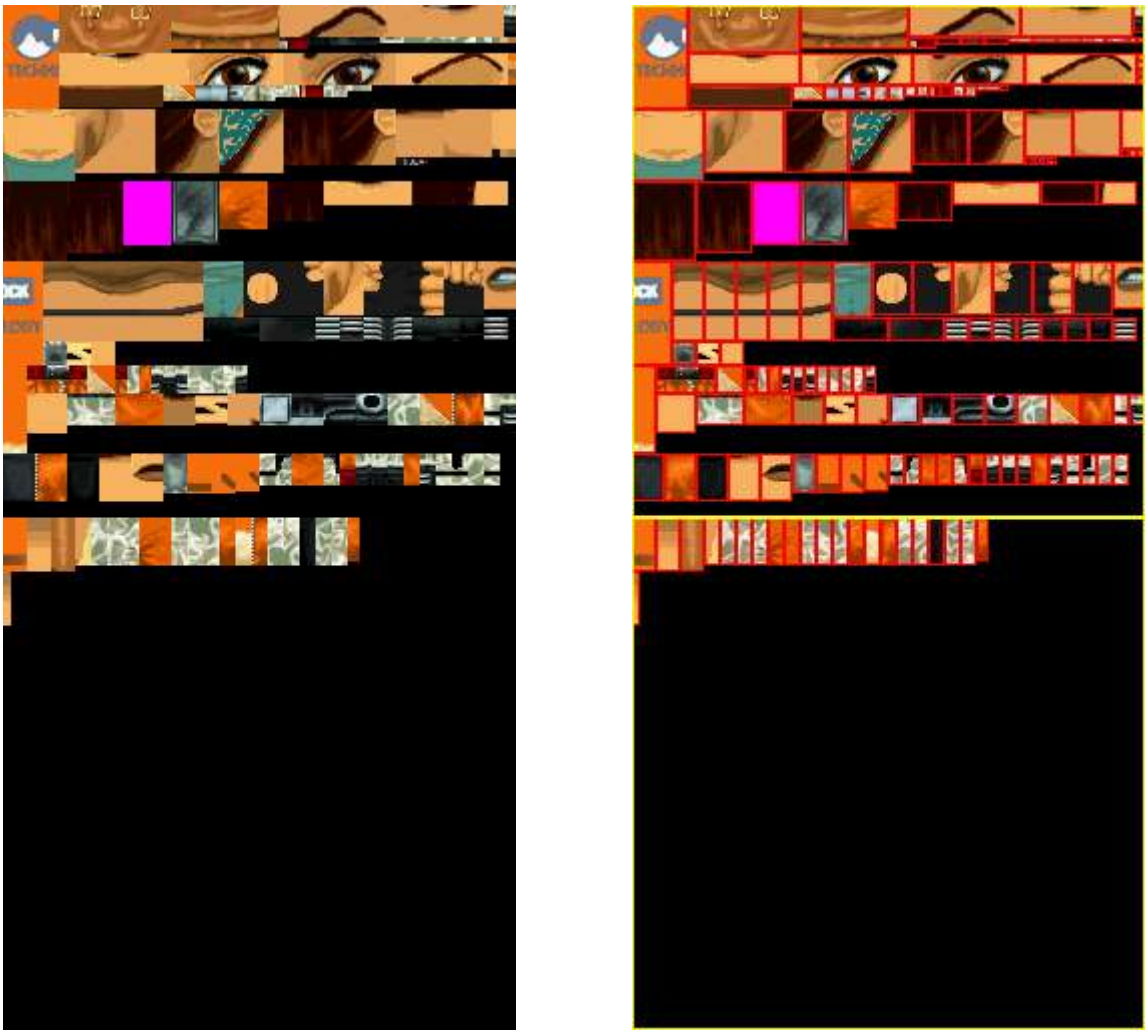


FIG. 3 - The picture on the left shows a texture map with two pages, each page with several samples. The picture on the right shows exactly the same texture map, but with the grid turned on. The yellow 256 x 256 squares mark the pages. In red, the bounding rectangles for each texture sample. These rectangles are generated by my exploratory application using the formulas defined above.

Texture_Num_Bytes (**uint32**).
The texture samples are packed into texture pages. Each texture page is 256 * 256 pixels in size. The texture pages are in turn stored as a single Texture Map, whose total size in bytes is given by **Texture_Num_Bytes**.

Texture_Map_Package (**Texture_Num_Bytes * 1 byte**).
The Texture Map itself is stored as a standard RAW 24bits [R G B] pixel file.
Some dimensions of the texture map are known by default: the width is always 256 pixels, the pixel format always takes 3 bytes per pixel.

The height of the texture map and the number of pages need to be calculated.

```
map_width   =  256

pixel_depth  =  3
```

```
map_height   =  ( number_of_pages * 256 )

Texture_Num_Bytes   =  map_width  *  ( map_height )  *  pixel_depth

                    =  256  *  ( number_of_pages * 256 )  *  3

                    =  number_of_pages  *  ( 256  *  256  *  3 )
```

Given the value of **Texture_Num_Bytes**, the number of pages in the Texture Map can be deduced from the above equation, as well as the height of the texture map in pixels.

```
number_of_pages  =  Texture_Num_Bytes  div  ( 256  *  256  *  3 )
```

```
map_height     =  number_of_pages * 256

               =  Texture_Num_Bytes  * 256 div ( 256 * 256 * 3 )
```

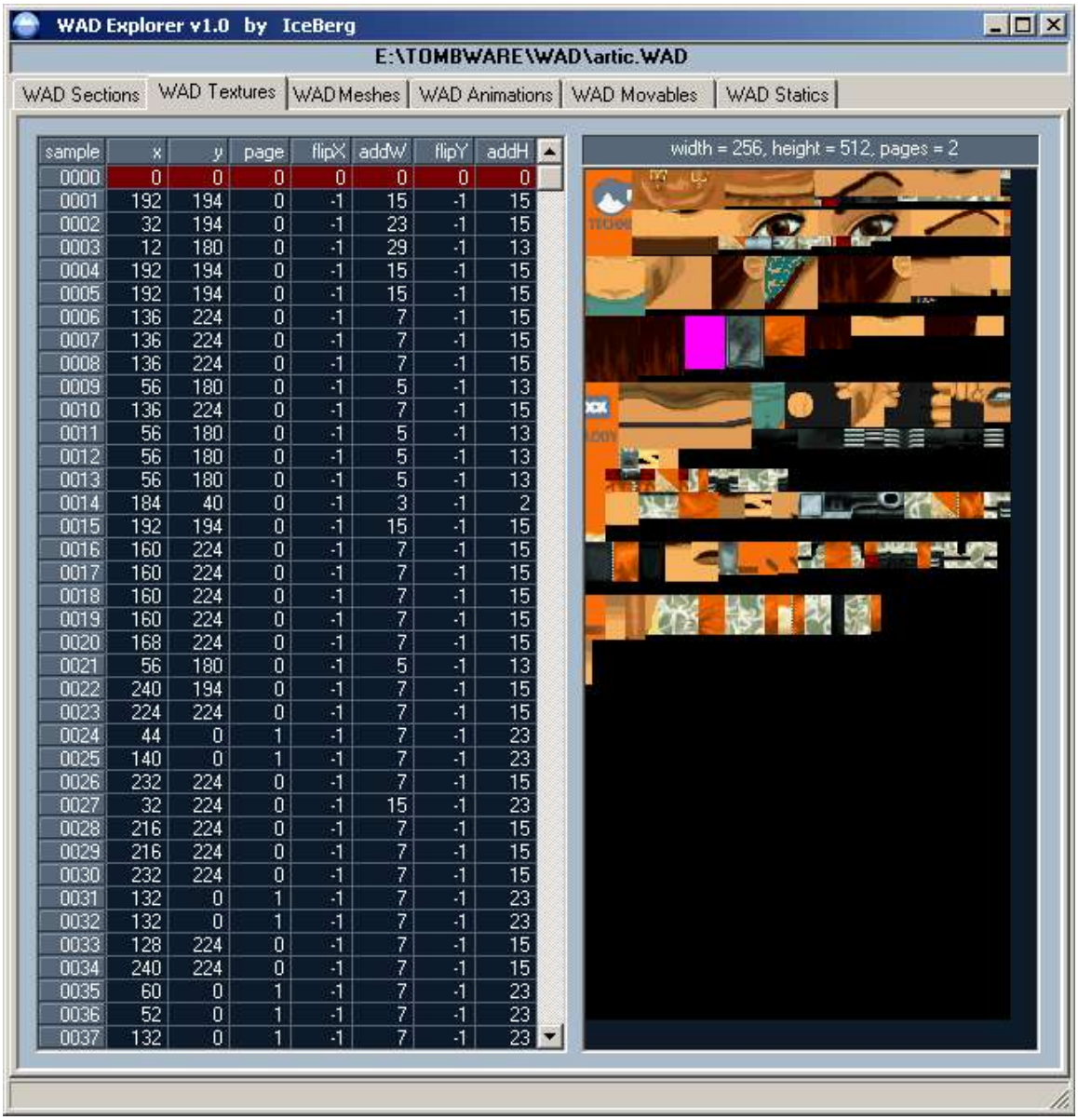
```
map_height  =  Texture_Num_Bytes  div  ( 256  *  3 )
```

The standard RAW file format stores the 24bits RGB data as [R G B] in-buffer.
These values will need to be reversed when transferring data to a standard BMP bitmap, which stores pixel data as [B G R] in-buffer.

The standard RAW file format stores its image as top-down (the first bytes in the file correspond to the first pixels in the top-left corner of the image). The scanlines will need to be reversed when transferring data to a standard BMP bitmap, whose scanlines are organized bottom-up (the first bytes in the file correspond to the pixels in the bottom-left corner of the image).

With a depth of **24 bits**, the Texture Map has *no alpha channel*. To define if a pixel is transparent or not, the **magenta** colour **RGB(255, 0, 255)** is used to paint the transparent pixels. This is a yes-or-no situation, transparent-or-not, there are no translucent levels.

TRLE will convert these **24-bit** textures to its internal **32-bit** textures by adding an extra channel to store alpha information. The **alpha channel** is opaque except where **magenta** was found. After storing the transparency information in the alpha channel the **magenta** colour is discarded, changed into **black** in the RGB colour channels. Internally, TRLE uses the alpha channel to produce transparency, not the magenta colour. In fact, *the 32-bit texture map inside the TR4 level file has no magenta at all*.



sample	x	y	page	flipX	addW	flipY	addH
0000	0	0	0	0	0	0	0
0001	192	194	0	-1	15	-1	15
0002	32	194	0	-1	23	-1	15
0003	12	180	0	-1	29	-1	13
0004	192	194	0	-1	15	-1	15
0005	192	194	0	-1	15	-1	15
0006	136	224	0	-1	7	-1	15
0007	136	224	0	-1	7	-1	15
0008	136	224	0	-1	7	-1	15
0009	56	180	0	-1	5	-1	13
0010	136	224	0	-1	7	-1	15
0011	56	180	0	-1	5	-1	13
0012	56	180	0	-1	5	-1	13
0013	56	180	0	-1	5	-1	13
0014	184	40	0	-1	3	-1	2
0015	192	194	0	-1	15	-1	15
0016	160	224	0	-1	7	-1	15
0017	160	224	0	-1	7	-1	15
0018	160	224	0	-1	7	-1	15
0019	160	224	0	-1	7	-1	15
0020	168	224	0	-1	7	-1	15
0021	56	180	0	-1	5	-1	13
0022	240	194	0	-1	7	-1	15
0023	224	224	0	-1	7	-1	15
0024	44	0	1	-1	7	-1	23
0025	140	0	1	-1	7	-1	23
0026	232	224	0	-1	7	-1	15
0027	32	224	0	-1	15	-1	23
0028	216	224	0	-1	7	-1	15
0029	216	224	0	-1	7	-1	15
0030	232	224	0	-1	7	-1	15
0031	132	0	1	-1	7	-1	23
0032	132	0	1	-1	7	-1	23
0033	128	224	0	-1	7	-1	15
0034	240	224	0	-1	7	-1	15
0035	60	0	1	-1	7	-1	23
0036	52	0	1	-1	7	-1	23
0037	132	0	1	-1	7	-1	23

FIG. 4 - The **flipX** and **flipY** values are *both* mostly -1, with only a few exceptions. This is equivalent to a 180 degrees rotation, that would put the textures upside-down before being mapped.

Section 3 – Meshes

Num_Mesh_Pointers (**uint32**).
Number of pointers stored in the **Mesh_Pointers_List**.

Mesh_Pointers_List (**Num_Mesh_Pointers * 4 bytes**).
Each record in this list is an offset to a mesh located in the **Mesh_Data_Package**. Description:

offset (**uint32**) offset of a mesh in the meshes package.

The meshes are not referenced directly. The **Mesh_Pointers_List** is used instead, and a pointer is extracted from it. This pointer represents the *offset* of the desired mesh from the beginning of the mesh data package, as given by:

absolute_address_of_the_mesh[i] = beginning_of_the_package
+ **Mesh_Pointers_List**[i]

In many WAD files there are several *different pointers* offsetting to the *same mesh*. The "0" offset, for instance, happens to be very much repeated. The same can happen with *other values*. In a general way the number of pointers is greater then the number of meshes, not equal to it.

There is another way for looking at this list.
Given a 3D model made of several meshes, these meshes will be referenced in the **Mesh_Pointers_List** . The pointers that refer to a certain 3D model are grouped together. If a given 3D model has 15 meshes, then the list will have 15 sequencial pointers storing the offsets of those meshes. The meshes themselves may be stored somewhere in a random way, but the pointers will be grouped together. The structure of the list is related to the structure of the 3D model. It fills the model' skeleton with meshes.

Another characteristic of this list is the repetition of the "zero" offset.
It actually points to a complex mesh, most of the times it points to a representation of Lara's left thigh. This makes no sense. No 3D model is made of 15 left thighs. Another approach is facing this "zero" offset as a dummy mesh, a placeholder. The sequence of "zeros" still relate to the structure of the 3D model, but which are the true meshes, then?

In this case the true meshes are given by another 3D model, considered as the main skin model. This introduces the possibility of associating animations to a "dummy structure" and later on assigning a "true skin" to the dummies. In fact, this introduces a skeletal hierarchy.

pointer	to offset
0000	0
0001	0
0002	0
0003	0
0004	0
0005	0
0006	0
0007	0
0008	0
0009	0
0010	0
0011	0
0012	0
0013	0
0014	0
0015	0
0016	0
0017	0
0018	0
0019	1,268
0020	0
0021	0
0022	0
0023	0
0024	0
0025	2,524
0026	0
0027	0
0028	3,068
0029	0
0030	0
0031	3,612
0032	0
0033	0
0034	5,048

FIG. 5 - The mesh pointers list. The first 15 pointers, marked in maroon, belong to Lara. There are 445 animations associated to this group. No meshes, however, as this scheme works like an animated skeleton that must be associated to a skin.

Meshes_Num_Words (**uint32**).
Size of the **Meshes_Data_Package**, expressed in **word** (**uint16**) units. Given that a **word** takes 2 bytes, the total number of bytes used to store the mesh data is given by:

package_size_in_words = **Meshes_Num_Words**

package_size_in_bytes = 2 * **Meshes_Num_Words**

Meshes_Data_Package (**Meshes_Num_Words * 2 bytes**).
The mesh data package stores the meshes all together, as a single package. The WAD file has no indications on *how many* meshes it stores.
The number of mesh pointers, **Num_Mesh_Pointers**, tells us nothing about this.

Some extra work will be needed to find out the number of meshes:

- Known the address of the beginning of the mesh package, known the size of the package, the address of the end of the mesh package can be computed.

  end_of_the_package = beginning_of_the_package
  + package_size_in_bytes

  Then the package can be parsed for each mesh, and a counting of them can be kept while parsing. The parsing finishes when the end_of_the_package is reached, and by then we will know how many meshes were counted.

- Given the list of pointers, **Mesh_Pointers_List**, we can count the number of *unique* pointers, disregarding the repetitions. A separated array can be created to associate a unique mesh index to its mesh pointer (to its offset, in fact). This same array can have some more dimensions and store the size of each mesh, the number of vertices per mesh, the number of normal vectors or lighting shades per mesh, the number of polygons per mesh, and whatever more the programmer needs.

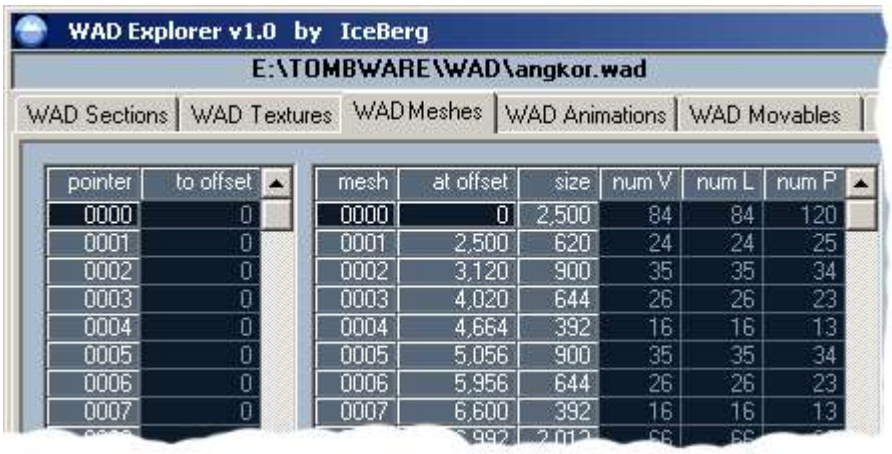


FIG. 6 - An auxiliary table was built into my exploratory application to assist in searching through the meshes: a meshes table. The WAD file does not have such meshes table, it only has the pointers list.

The meshes in the package vary in size, but they all have the same internal structure:

Bounding_Sphere (**10 bytes**).
Coordinates of the centre, and the radius, of a bounding sphere used for proximity testing in-game. Used by Movable Models only, Static Models have zeros in these fields. Description:

cx	(**sint16**)	centre's coordinate in x.
cy	(**sint16**)	centre's coordinate in y.
cz	(**sint16**)	centre's coordinate in z.
radius	(**uint16**)	radius of the sphere.
unk	(**uint16**)	unknown[5].

A Movable Model may have several meshes, each one with its own bounding sphere. Setting to zero the radius of the spheres disables their collision detection capability. If all the spheres in a Movable are disabled, then Lara can move through the object.

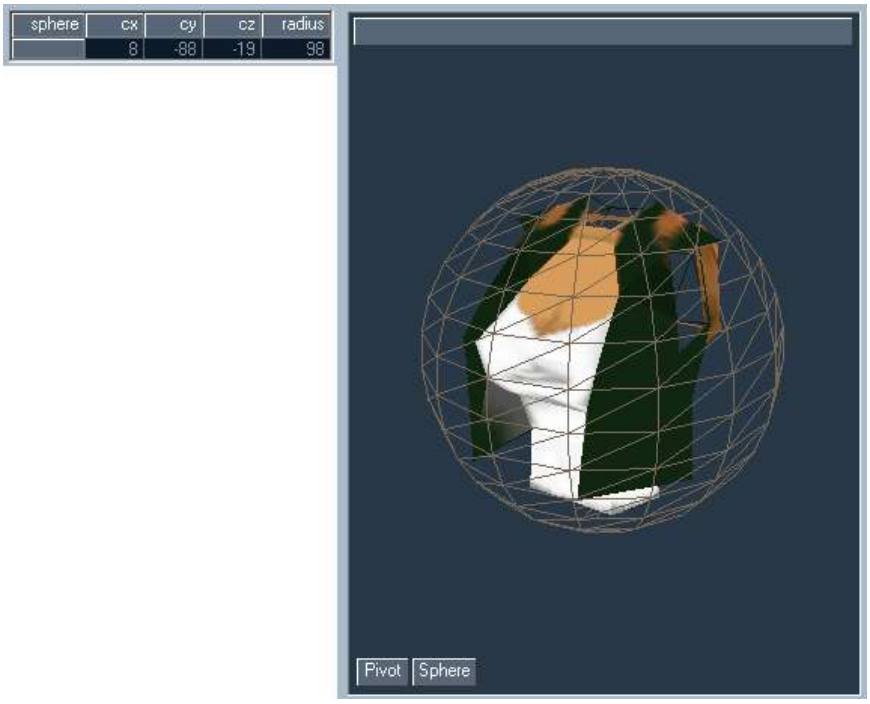


FIG. 7 - The Bounding Sphere is usually placed close to the object's volumetric centre and has a small radius, just enough to envelope the mesh.

The unknown field **unk** can be found in both *Movables* and *Statics*, and only takes the values of **0** and **1**. Most objects have a zero in this field, which seems to be the default value.

[5] *The usual description of the Bounding Sphere states that its radius is a 32-bit integer. I found out that it is not so. The radius is a* ***uint16*** *(2 bytes). This raises a new question: what are the other 2 bytes?*

It is not clear what this **unk** field does. The fact that it takes only two values, **0** and **1**, suggests that it is a flag. The fact that it also shows up in *Statics* suggests that it is *not* related to the **radius** of the sphere, which static models do not have.

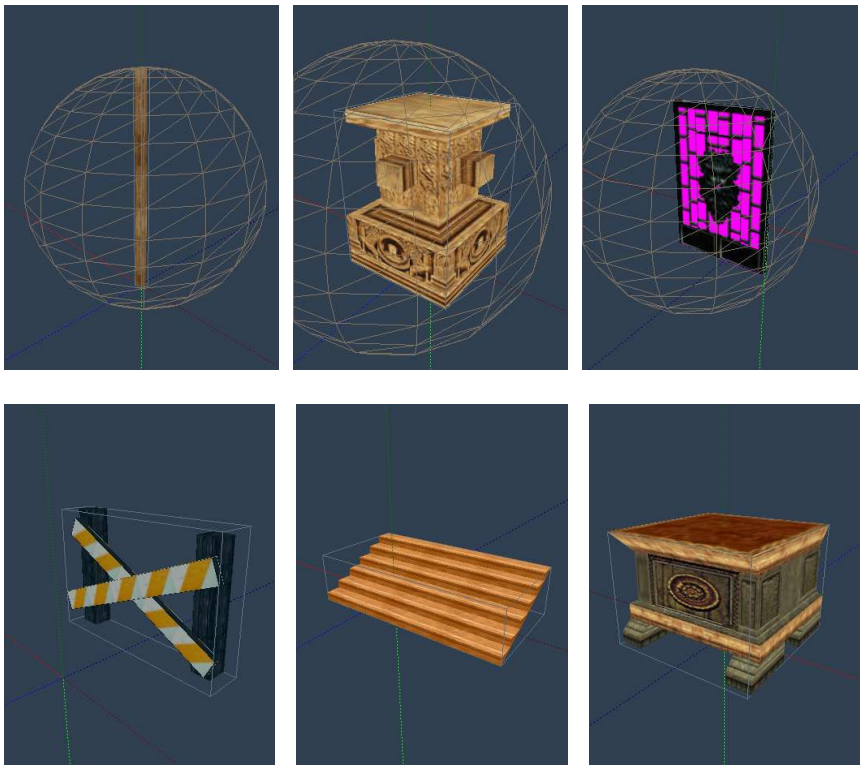


FIG. 8 - Examples of both Movables and Statics which have a **1** in the **unk** field.

These are the **meshes** found in the original WAD files which have a **one** in this field:

ANGKOR: 147, 148, 149, 150, 151, 206, 232, 233, 242, 245, 246, 248.

CATACOMB: 117, 118, 119, 121, 155, 156, 157, 180, 234, 241, 242.

CITY: 115 .. 129, 185, 250, 251, 252, 253, 381, 396, 407.

CLEOPAL: 141, 142, 171, 184, 185, 187, 188, 189, 190, 191, 275, 276, 284.

COASTAL: 122, 147, 184, 185, 330, 336, 350.

GUARD: 130, 226, 236, 256, 395, 397, 400, 402, 410.

KARNAK: 169, 171, 172, 173, 174.

LIBRARY: 153, 213, 214, 215, 216, 323, 324, 329, 334, 339, 340 .. 344, 346, 349.

NEWCITY: 140 .. 154, 206, 271, 272, 273, 274, 402, 417, 428.

SETTOMB: 216, 228, 229, 230, 231, 232, 233, 234, 235, 236, 335.

TITLE: 133, 134, 135.

TUT1: 180, 181, 182, 297.

Num_Vertices (**uint16**).
Number of vertices in the mesh.

Vertices_Table (**Num_Vertices * 6 bytes**).
Table storing the XYZ coordinates of the vertices. Description:

vx	(**sint16**)	vertex coordinate in x.
vy	(**sint16**)	vertex coordinate in y.
vz	(**sint16**)	vertex coordinate in z.

These values are stored as signed 16-bit integers. These values may be too big for the common DirectX, OpenGL or 3D applications to work with. Most Tomb Raider dedicated applications are dividing these values by 100 or by 1000, turning them into single-precision floating-point values.

In the **Texture_Map** section we've seen that most texture samples have *both* flip flags marked as -1, which is equivalent to a 180 degrees rotation. Now we see something similar happening with the geometry. It is upside down!

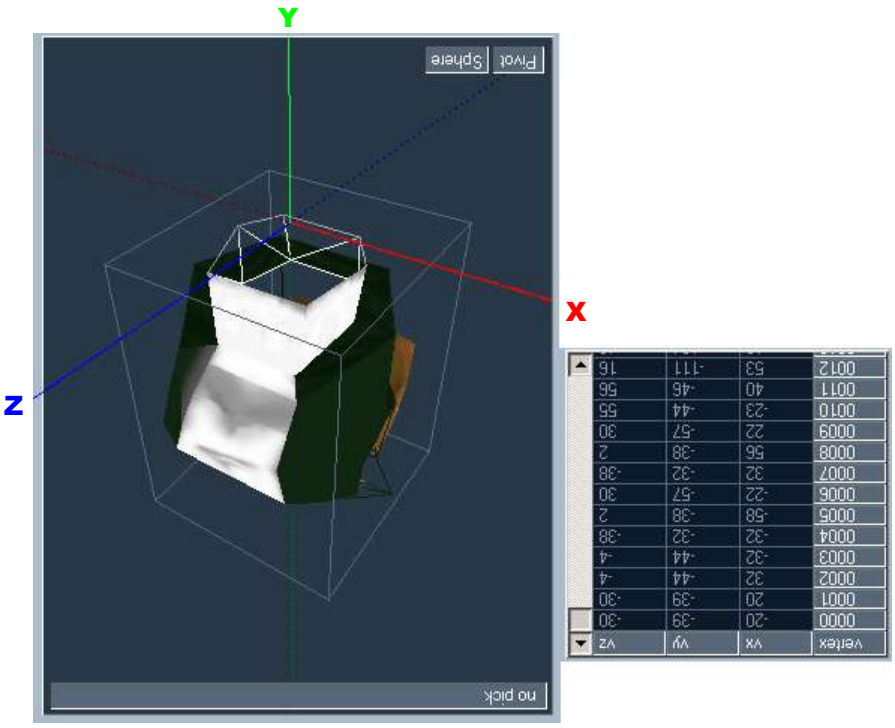


FIG. 9 - The textures go upside-down, the geometry is upside-down, so everything fits together nicely. Now what? Are we supposed to turn ourselves upside-down as well? Where did this idea came from?

If we do not rotate the textures and if we rotate the 3D space instead, we at least get a more reasonable situation to deal with.

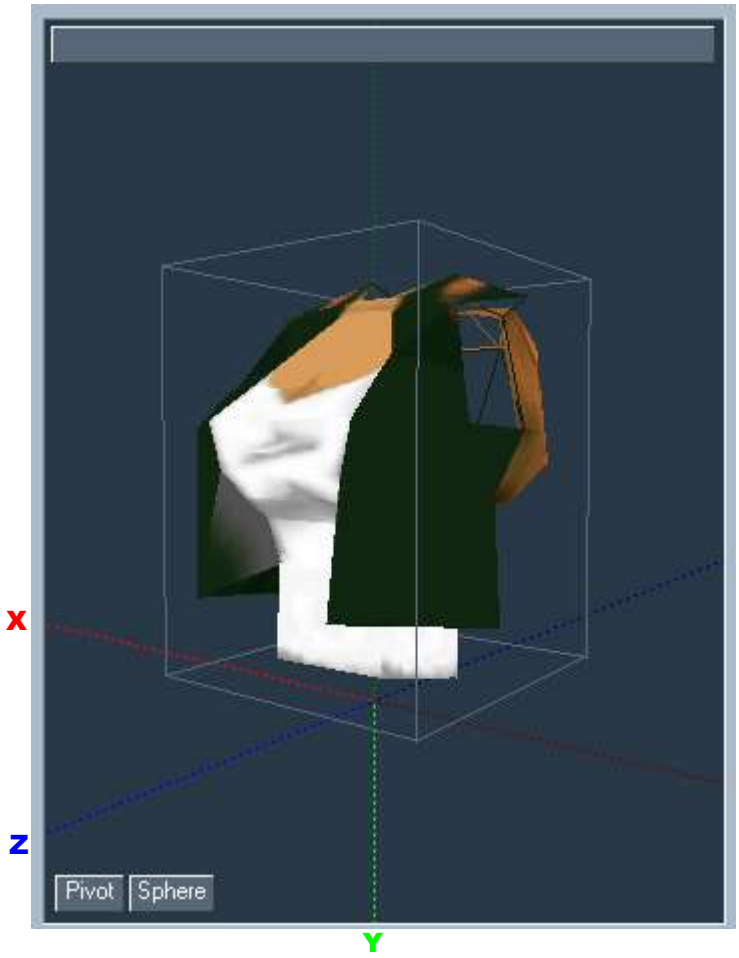


FIG. 10 - The origin of the axes is the **pivot point** of the mesh. The coordinates of the vertices are relative to this point.

Num_Normals / Num_Shades (**sint16**).
The value stored here has a different meaning depending on its sign. If **positive**, it means **Num_Normals**, if **negative** it means **Num_Shades**.

- Number of normal vectors of the mesh, one per vertex, to describe how the faces are affected by external lights. This is the method used by Movables.
- Number of shading values, one per vertex, to describe the vertex's own shading. This is the method used by Statics.

FIG. 11 - Normals vs. Shades: the light projected on the wall is caused by the motorbike's beam. But the Static that lies just in front of the beam remains dark. Statics use their own internal shading values, they do not use external lighting. On the other hand, Lara and the motorbike, which are Movables, show the influence of external lights located somewhere behind and to the right of Lara's position. *(screen capture from TRLE).*

Normals_Table (**Num_Normals * 6 bytes**).
Table with the XYZ lengths of the normal vectors attached to the vertices of the mesh. This normal vector defines a direction. The more this vector points to a light, the brighter the vertex will be. In the vicinity of that vertex, the colour values found on the adjacent faces will be pulled up or down towards the colour values of the external light. Description:

lx	(**sint16**)	vector length in x.
ly	(**sint16**)	vertex length in y.
lz	(**sint16**)	vertex length in z.

A normal vector like this one, which purpose is to define a direction, is expected to have a “normalized” length, to be an “unary” vector :

lx ^ 2 + **ly** ^ 2 + **lz** ^ 2 = (vector_length) ^ 2 = 1

But, applying the above equation to the **lx**, **ly**, **lz** values stored in the WAD file, we get a very different result, we get:

vector_length = 16300

To “normalize” this vector we need to divide each component by 16300.

nx	(**single**)	normalized vector length in x	= **lx** / 16300.
ny	(**single**)	normalized vector length in y	= **ly** / 16300.
nz	(**single**)	normalized vector length in z	= **lz** / 16300.

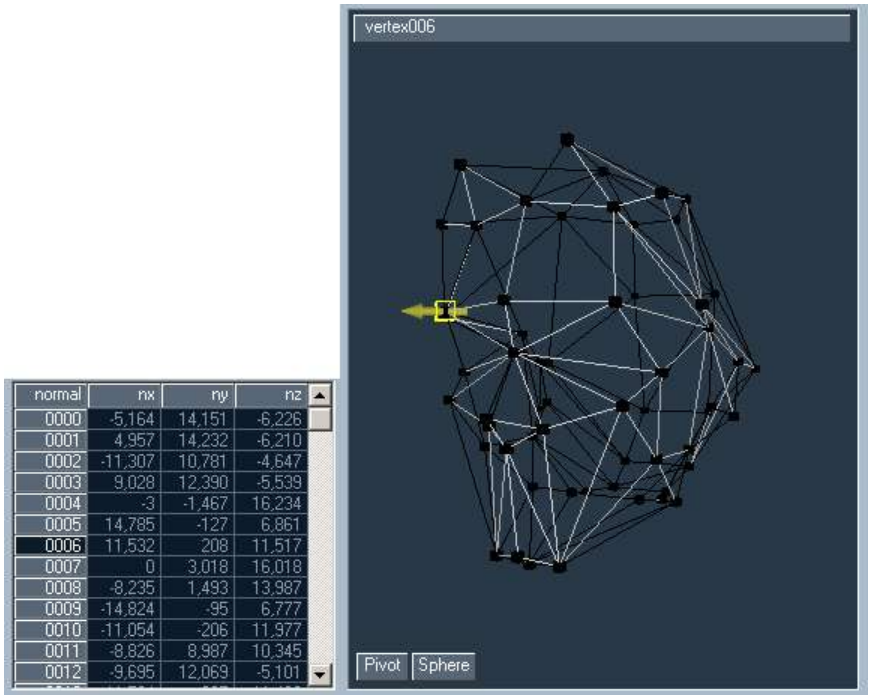


FIG. 12 - The integer values in the **Normals_Table** need to be divided by 16300 to obtain a vector of length = 1. Where is that “16300” coming from?

This “normal vector” business is quite relevant when making custom wads. New geometries may be built with 3D/CAD applications, not every mesh will have its normal vectors correct by default. If the normal vectors are not properly oriented, shading errors will happen in-game.

FIG. 13 - The normal vectors in the head of this custom model are not correct (top insert). They need to be recalculated, which can be done with STRPIX3.95R11 (bottom insert).
(custom wad by L.Croft - AOD Louvre - from Lara's Level Base).

Shades_List (**Num_Shades * 2 bytes**) * (**-1**).
The (minus one) is to turn positive a number that is stored as negative.
List of values representing a light intensity, a shade of grey for each vertex. The shade will affect the luminosity or the darkness of its correspondent vertex. In the vicinity of that vertex, the colour values found on the adjacent faces will be pulled up or down depending on the value of the grey shade of the vertex.
Description:

shade (**sint16**) shade of grey for the vertex.

Representing shades of grey, these values would be expected to lie in a range of [0..255], like the usual RGB colour space does. Nope! The values are stored from darker to lighter in a range of [$1FFF..0]. Darker is $1FFF and lighter is $0000, whereas in the RGB colour space darker is $00 and lighter is $FF. Quite the opposite, and quite a difference in the range of values.

FIG. 14 - The light on the wall is caused by the Lara's flare. But the Static that lies just in front of her remains dark. *(screen capture from TRLE).*

Some extra work will be needed to convert these values.
Given this scheme, that represents the relative position of the values…

Max		**Convert**		**Min**
$FF	.....	**grey**	.....	$00
$0000	.....	**shade**	.....	$1FFF

… the following proportions can be extracted:

$$\frac{(\ \textbf{grey}\ -\ \$00\ )}{(\ \textbf{shade}\ -\ \$1FFF\ )} = \frac{(\ \$FF\ -\ \$00\ )}{(\ \$0000\ -\ \$1FFF\ )}$$

Resolving in order to extract the grey value:

(**grey**) * (- $1FFF) = (**shade** - $1FFF) * ($FF)

grey * $1FFF = $1FFF * $FF - **shade** * $FF

grey = $FF - **shade** * $FF / $1FFF

Replacing those computer-like hexadecimal values by their human-like decimal equivalents:

grey = 255 - **shade** * 255 / 8191

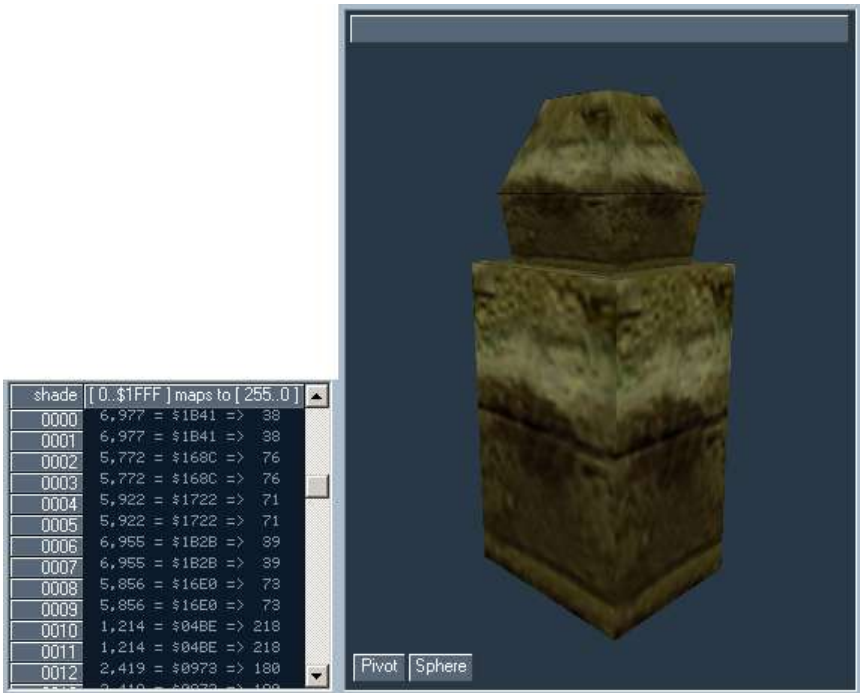


FIG. 15 - This is the Static, textures only, no shading applied.

Num_Polygons (**uint16**).
Number of polygons in the mesh.

Polygons_Package (**Num_Polygons * variable bytes**).
Yes, that's correct. We know the number of polygons, but that does not tell us the size of the polygons data package. That's because the WAD file stores triangles (with 3 vertices) and quads (with 4 vertices) all mixed up. With no warning, each record can have, or not to have, a 4th vertex. The only way for handling this is parsing all the way through and keep counting polygons. While doing it, we need to **keep a separated count of the number of quads**. That will tell us if there must be, or not, an extra **padding Word** at the end of the polygons data package. If the number of quads is **even** (multiple of 2) there is **no padding**. But if such number is **odd**, then there will be **one padding word**.

The structure of each polygon record is the following:

shape	(**uint16**)	a triangle, or a quad.
v1	(**uint16**)	anchor vertex index.
v2	(**uint16**)	next vertex index.
v3	(**uint16**)	next vertex index.
v4	(***uint16***)	*next, only if quad*.
texture	(**uint16**)	index and horizontal flip.
attributes	(**uint8**)	opacity and shine.
unk	(**uint8**)	unused byte.

The **shape** field tells us if the polygon is a triangle (8) or a quad (9), therefore if the record contains 3 or 4 vertex indices. These indices are used on the **Vertices_Data** array to extract the actual XYZ coordinates of the vertex.

Starting from the **anchor vertex v1**, the first vertex listed in the polygon's record, the other vertices are assigned in a clockwise manner, as seen from the visible side of the polygon.

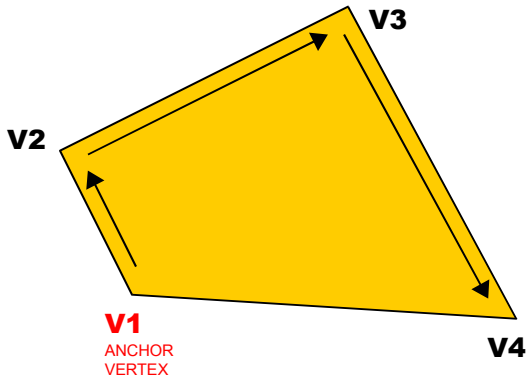


FIG. 16 - The vertices are assigned in a clockwise manner for the polygon to be visible.

The **texture** word, treated as a bit field, has the following meaning:

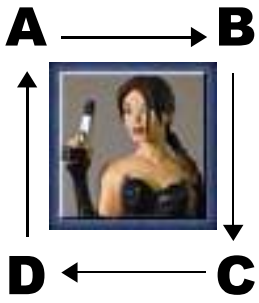
flipped	**texture and $8000**	(**1 bit**)	horizontal flipping.
shape	**texture and $7000**	(**3 bits**)	texture sample shape.
index	**texture and $0FFF**	(**12 bits**)	texture sample index.

If bit [15] is set, as in (**texture and $8000**), it indicates that the texture sample must be **flipped** horizontally prior to be used.

Bits [14..12] as in (**texture and $7000**), are used to store the texture **shape**. More details are given below.

Bits [11..0] as in (**texture and $0FFF**), are used to store a texture **index** to be used on the **Texture_Samples_Table** to find the sample's bitmap.

The texture **shape** is given by: (**texture and $7000**) **shr 12**
The valid values are: **0**, **2**, **4**, **6**, **7**, as shown below.



0 = triangle **DAB**

2 = triangle **ABC**

4 = triangle **BCD**

6 = triangle **CDA**

7 = quad **ABCD**

If **flipped** = 0, the texture would be used like this:

If **flipped** = 1, the texture would be used like this:

FIG. 17 - The purpose of **shape** is to define the shape of the texture sample and how it relates to the polygon's **shape**. The **red** numbers above mark the **anchor corner** of the texture sample for each texture shape.

The texture's **anchor corner** goes to the polygon's **anchor vertex**. Then the next texture corner (a clockwise-next) goes to the next vertex (also a clockwise-next).

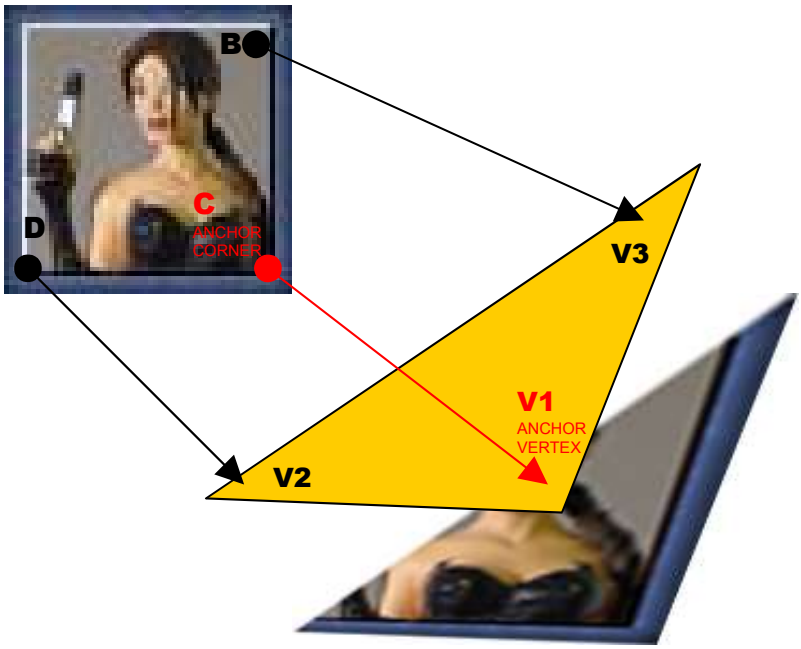


FIG. 18 - Mapping a texture shape 4 on a polygon shape triangle. In practical terms, that's *shape-to-shape*, *anchor-to-anchor*, *next-to-next*. Clockwise.

To **rotate** a texture in the polygon, the sequence of vertices needs to be shifted. If a sequence [v1, v2, v3] shifts to [v2, v3, v1], for example, v2 becomes the anchor vertex and the mapping is rotated clockwise.

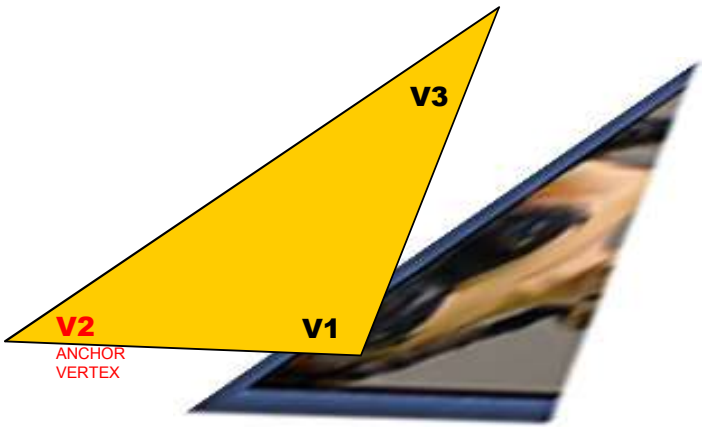


FIG. 19 - The *vertices* themselves did not move, the geometry of the mesh was not modified in any way. It was just their *references* in the polygon record that were shifted. The texture's anchor corner follows the anchor vertex.

The **attributes** byte, treated as a bit-field, stores the following data:

unk	**attributes and $80**	(**1 bit**)	not used?
intensity	**attributes and $7C**	(**5 bits**)	shine effect intensity.
shine	**attributes and $02**	(**1 bit**)	shine effect flag.
opacity	**attributes and $01**	(**1 bit**)	opacity mode.

Bit [7] as in (**attributes and $80**), apparently is not used. I've noticed that, when set, it disables the shine effect. I did not find any examples of this bit being used.

Bits [6..2] as in (**attributes and $7C**), are used to store the **intensity** of the shine effect. The highest the value, the more intense the effect is. A field with 5 bits can store 32 values in the range [0..31].

If bit [1] is set, as in (**attributes and $02**), the **shine** effect is switched on.

Bit [0] as in (**attributes and $01**), is used to flag the **opacity** mode.
If this field is not set (bit value = 0) the texture is considered opaque and the **magenta colour** is used to mark pixels that are to be made fully transparent.
If this field is set (bit value = 1), then colours are treated as translucent, after converting the magenta colour, if any, to black.

Translucent textures, or colours, are processed as *additive*.
In this scheme, a translucent texture, processed on top of an opaque texture, will have its colour values *added* to those of the texture below.
The darker the colour, the more transparent; the brighter the colour, the more opaque. In the limits, black is completely transparent and white is completely opaque. Adding colours makes the final result tend to white, which is the limit.

The Shine Effect can be used with Movables, only[6]. If set with Static Models, the TR4 engine crashes. However, the Intensity can be set, even on Statics[7].

[6] *I've found some objects, like the pistol gun or the laser accessory, which show up in the game's menu, with attributes = **$02**. This makes the shine effect turned on, but the intensity is set to zero! Why is the shine effect flag on if there is no intensity?*

[7] *I've found static models, which cannot use the shine effect, with the flag turned off, as it should, but with an intensity value set! As in attributes = **$40**. Why is the intensity set, if there is no shinning?*

Padding (**uint16**).
This field only exists if the number of quads is odd (not a multiple of 2) as given by the logical test:

(number_of_quads **mod** 2) = 1

If this test is TRUE the remaining of the division by 2 is different from zero (or it is equal to one, the only other possibility), meaning that the number is **odd**. If so, a padding word must be added.
If the test is FALSE the remaining of the division by 2 is zero, meaning that the number is **even**, and no padding is added.

This padding word, apparently, serves no useful purpose. But it is there, so it must be taken in consideration, specially when building up new WAD files.

And this completes the parsing of a mesh.

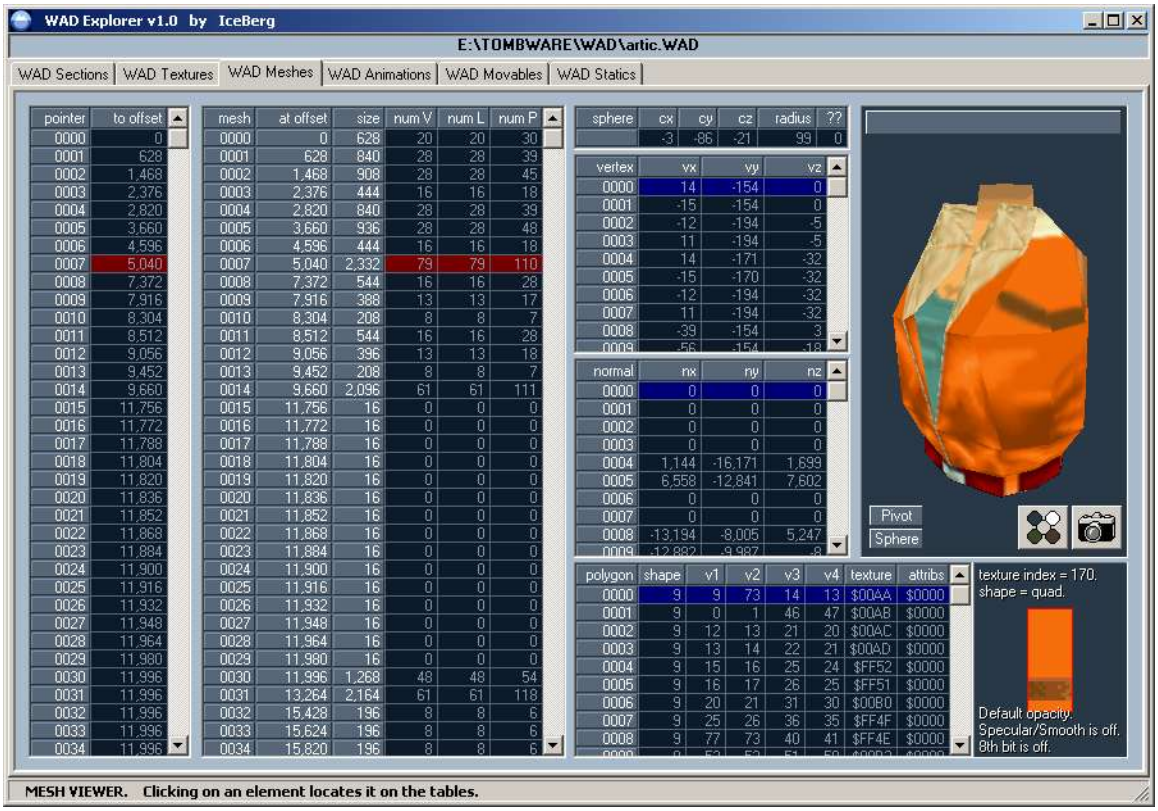


FIG. 20 - Mesh pointers table, auxiliary data about the mesh package, collision sphere, vertices, normal vectors and polygons. OpenGL window displaying one mesh and a texture reference.

Section 4 – Animations

Num_Animations (**uint32**).
Number of animations stored in the **Animations_Table**.

Animations_Table (**Num_Animations * 40 bytes**).
This table is the entry point to access the Animations stored in the WAD file. The indexes and the offsets it contains are used to fetch data from other tables and packages.

These animations are in fact segments of the model's *actions*. For example, a simple action like "running" involves different segments of animation for "start running", "running" and "stop running". The way how the animations are segmented and put together is the most complex subject in the WAD file.

keyframeOffset	(**uint32**)	offset in **Keyframes_Data_Package**.
frameDuration	(**uint8**)	engine ticks per frame.
keyframeSize	(**uint8**)	size of the keyframe record, in words.
state_ID	(**uint16**)	ID of the state of this animation.
unknown1	(**sint16**)	unknown 2 bytes.
speed	(**sint16**)	ground speed.
acceleration	(**sint32**)	easy-in and easy-out for the speed.
unknown2	(**sint64**)	unknown 8 bytes.
frameStart	(**uint16**)	[frame-in] index of this animation.
frameEnd	(**uint16**)	[frame-out] index of this animation.
nextAnimation	(**uint16**)	index of the default next animation.
frameIn	(**uint16**)	[frame-in] index of the next animation.
numStateChanges	(**uint16**)	number of animation transitions.
changesIndex	(**uint16**)	index in **State_Changes_Table**.
numCommands	(**uint16**)	number of commands.
commandOffsets	(**uint16**)	offset in **Commands_Data_Package**.

Animations consist of sequences of *frames*, some of which are *keyframes* that determine the key attitudes of the model, defined by the animator. The other are *interpolated* frames, in-betweens computed from the keyframes.

Not going into details yet, the *keyframe records* that describe an animation are stored in a data package, the **Keyframes_Data_Package,** and can be found by their offsets and record sizes. To find every keyframe that belongs to a given animation, we need to know the **keyframeOffset** and the **keyframeSize** of those keyframe records.

The rate at which the frames are shown is determined by **frameDuration**. The higher the value in this field the slower the animation.

Some animation segments are interrelated, like "running" is related to "start running" and to "stop running", but not related to "jumping", which will in turn be related to "start jumping" and to "stop jumping". The terminology used in the TRosettaStone to designate these families of animations is **state**. The "state of running" comprises a family of animation segments that closely relate to the "action of running". Each animation segment in the **Animations_Table** belongs to one and only one family. Each animation segment has a **state_ID** to identify its family.

The keyframe records in the **Keyframes_Data_Package** store the attitudes of the Movable Model and its elevations relative to the ground level, but not how the model travels through the scenery. The trajectory of the model is decided by the player at playing-time. However, the *ground speed*

and the *ground acceleration* of the model depend on the design of the animations. Some animation segments may depict an action with zero ground speed. Lara is steady and points the shotgun, for example. Other may have a ground speed but no ground acceleration, like Lara running. Other may have no ground speed and have an acceleration, like Lara starting running. These values are stored in **speed** and **acceleration**. The **sint32** that stores the value of the acceleration is in fact a pseudo floating point which codes its integer part in the upper **sint16** and its decimal part in the lower **uint16**.

Floating_point_value = **acceleration** / 65536

Each animation segment has a given number of frames, some of which will be keyframes and some will be interpolated frames. Every segment starts at a given frame number, **frameStart**, and stops at a given frame number, **frameEnd**. The total number of frames in the animation segment, including *keyframes* and *interpolated*, is given by:

number_of_frames = **frameEnd** - **frameStart** + 1

The number of keyframes-only is not stored in the **Animations_Table**. It needs to be deduced from the **keyframeOffset** of the current animation segment, from the **keyframeOffset** of the next segment[8], and from the **keyframeSize** of the current segment converted to bytes.

num_keyframes = (keyframeOffset[i+1] - keyframeOffset[i]) **div** (2 * keyframeSize[i])

Animation segments are displayed in sequence to produce natural actions. Some segments will naturally follow other segments. When Lara is running, the most natural next action is to keep on running. These *natural next animation segments* are the defaults if the game's engine finds no reason to do something else. These defaults are stored in **nextAnimation** and its entry point is the **frameIn**.

But things may be different in-game. The player decides how to play. The game's engine runs the artificial intelligence of the NPC (non-player characters). Accidents happen. The state of the animation may change from running to falling, or something else unexpected. The sequence of animation segments no longer corresponds to the defaults, **nextAnimation** cannot help here.

The engine will know which is the "next animation" from the interpretation it makes from what's happening in the game. Now the engine needs to find out which are the animation segments that make the transition from the current state to the next state requested by the game.

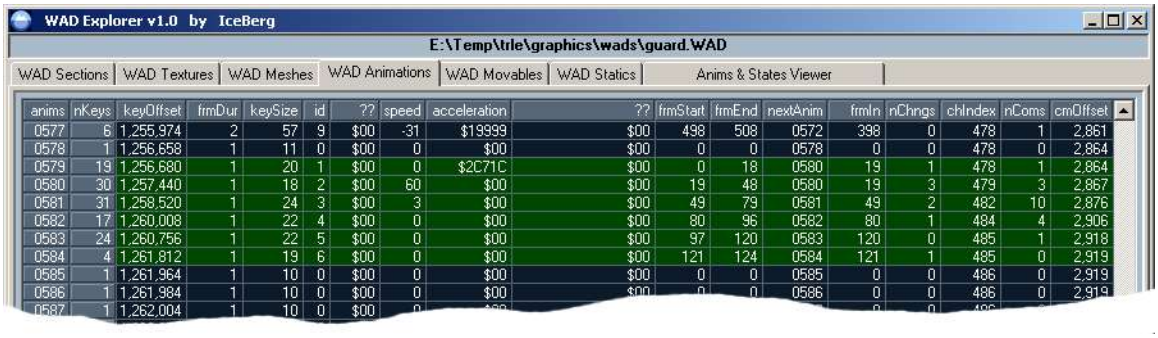
By design, there is only a certain number of possibilities for changing from the current state to another state. Not all the combinations are available. The possibilities are stored in another table, the **State_Changes_Table**.

In order to use it, we need to fetch some info from the **Animations_Table**. We need to know how many state changes to investigate, **numStateChanges**, and where are they stored, as given by their index, **changesIndex**.

Finally, while a given action is happening, some noises will be produced, some bubbles will be emitted, some various additional special effects may happen. There are special commands associated to those extra functions. These commands are packed into a **Commands_Data_Package** and can be read knowing its number, **numCommands**, and the location of the first command related to the current animation segment, **commandsOffset**. The rest needs to be parsed.

[8] *The last animation, the last keyframeOffset, has no next keyframeOffset. The total size of the animation table needs to be used instead.*

Given the complexity of this **Animation_Table** with its complex fields and their complex relations with other tables and packages, we will now follow an example: the **Bat**, movable model **ID# 90** in the **GUARD** WAD file.



anims	nKeys	keyOffset	frmDur	keySize	id	??	speed	acceleration	??	frmStart	frmEnd	nextAnim	frmIn	nChngs	chIndex	nComs	cmOffset
0577	6	1,255,974	2	57	9	$00	-31	$19999	$00	498	508	0572	398	0	478	1	2,861
0578	1	1,256,658	1	11	0	$00	0	$00	$00	0	0	0578	0	0	478	0	2,864
0579	19	1,256,680	1	20	1	$00	0	$2C71C	$00	0	18	0580	19	1	478	1	2,864
0580	30	1,257,440	1	18	2	$00	60	$00	$00	19	48	0580	19	3	479	3	2,867
0581	31	1,258,520	1	24	3	$00	3	$00	$00	49	79	0581	49	2	482	10	2,876
0582	17	1,260,008	1	22	4	$00	0	$00	$00	80	96	0582	80	1	484	4	2,906
0583	24	1,260,756	1	22	5	$00	0	$00	$00	97	120	0583	120	0	485	1	2,918
0584	4	1,261,812	1	19	6	$00	0	$00	$00	121	124	0584	121	1	485	0	2,919
0585	1	1,261,964	1	10	0	$00	0	$00	$00	0	0	0585	0	0	486	0	2,919
0586	1	1,261,984	1	10	0	$00	0	$00	$00	0	0	0586	0	0	486	0	2,919
0587	1	1,262,004	1	10	0	$00	0										

FIG. 21 - The Bat and its location in the **Animations_Table**, highlighting its 6 animation segments.
Only the body of the table is stored in the WAD file. The two left columns are implemented by my exploratory application.

First we locate the Bat in the **Movables_Table**, then we read its **animIndex**, which is an index to the **Animations_Table**. The index has the value **579**. Now we read the **animIndex** of the model *after* the Bat, which is **585**. The number of animation segments assigned to the Bat is given by **585 - 579 = 6** or, expressing this as a range, **animBat = [579 .. 584]**.

Now we locate the 6 animation segments **[579 .. 584]** in the **Animations_Table** (see FIG. 21).

Looking at the column for the state **id** we find out that there are 6 different states for the Bat's animations (see FIG. 22).

anims	id
0577	9
0578	0
0579	1
0580	2
0581	3
0582	4
0583	5
0584	6
0585	0
0586	0
0587	

FIG. 22 - Part of the **Animations_Table** of the Bat model, highlighting its 6 animation segments and concentrating in the six **state_ID** for the Bat's animations.

What are these *states*, what do they represent? Using my exploratory application to snapshot some frames from the animations, we can see what those animations are about.

animation segment # **579**  **(1) Start flying**.

animation segment # **580**  **(2) Flying**.

animation segment # **581**  **(3) Attacking**.

animation segment # **582**  **(4) Falling**.

animation segment # **583**  **(5) Dying**.

animation segment # **584**  **(6) Sleeping**.

FIG. 23 - The six states for the Bat's animations, displaying some frames from the animation segments, the respective state id, and an interpretation of what the animation is.

Each one of these animation segments has a natural next animation. Most likely "Sleeping" will carry on "Sleeping", "Start flying" will go to "Flying", "Flying" will carry on "Flying", eventually it will go to "Attacking", which in turn will eventually go to "Falling" and "Dying". Exactly how all this happens is stored in the **Animations_Table** under **nextAnimation** and **frameIn**.

anims	nextAnim	frmIn
0577	0572	398
0578	0578	0
0579	0580	19
0580	0580	19
0581	0581	49
0582	0582	80
0583	0583	120
0584	0584	121
0585	0585	0
0586	0586	0
0587		

FIG. 24 - Part of the **Animations_Table** of the Bat, highlighting its 6 animation segments and concentrating in the **nextAnimation** and **frameIn** fields.

Inspecting the **nextAnimation** column we see that the animation segment **579** goes to the default next animation **580**. We also see that **580** goes to **580**, it goes to itself. And so do the other, they go to themselves.

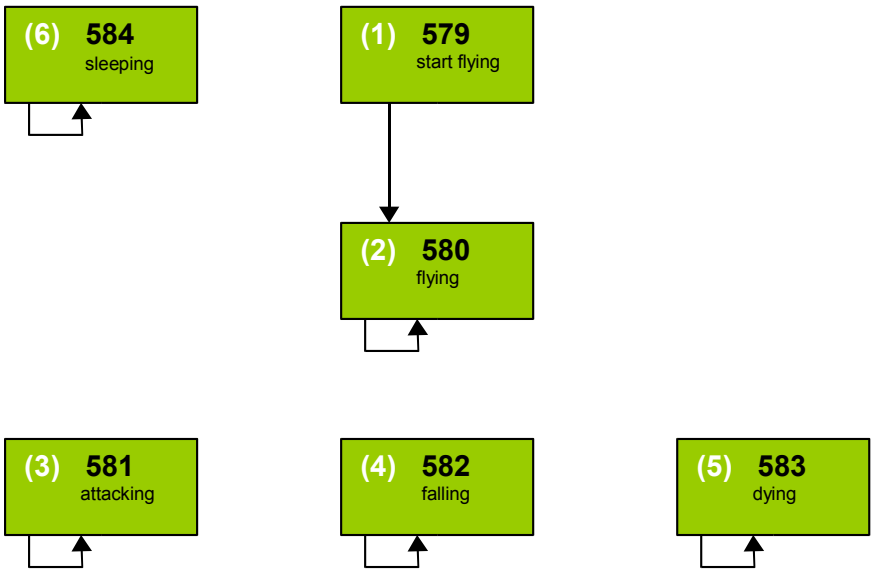


FIG. 25 - A representation of the six animation segments of the Bat with their **state_ID** numbers, and showing the default next animations for each segment, pointed to by the black arrows.

So now we know what the **nextAnimation** is about. But what is the meaning of the **frameIn** field? Like in the animation segment **580**, "Flying", whose default next animation is **580** itself, and whose **frameIn** is **19**. What is this **19**? It is the entry frame into the animation segment **580**, but where is that **19** coming from? To answer this we need to look into two other fields, **frameStart** and **frameEnd**.

anims	frmStart	frmEnd
0577	498	508
0578	0	0
0579	0	18
0580	19	48
0581	49	79
0582	80	96
0583	97	120
0584	121	124
0585	0	0
0586	0	0
0587		

FIG. 26 - Part of the **Animations_Table** of the Bat, highlighting its 6 animation segments and concentrating in the **frameStart** and **frameEnd** fields.

Inspecting these two columns we see that the animation segment **579** starts at frame **0** and ends at frame **18**, we see that the animation segment **580** starts at frame **19** and ends at frame **48**, etc. Crossing this with the "**19**" from FIG. 24, concerning the animation segment **580**, we now see that "**19**" is the *first frame* of the sequence. The complete interpretation of FIG. 24 is that the animation segment **580** goes to itself by default, to its first frame. In other words, this animation *loops*.

From FIG. 26 we can find out how many frames each segment has. Segment **579** has frames in the range **[0 .. 18]**, segment **580** has frames in the range **[19 .. 48]**, and so on, until segment **584** which has frames in the range **[121 .. 124]**. We've seen that the number of frames is given by the equation:

number_of_frames = **frameEnd** - **frameStart** + 1

With this equation we find that segment **579** has 19 frames (18 - 0 + 1), that segment **580** has 30 frames (48 - 19 + 1), and so on, until segment **584** which has 4 frames (124 - 121 + 1).

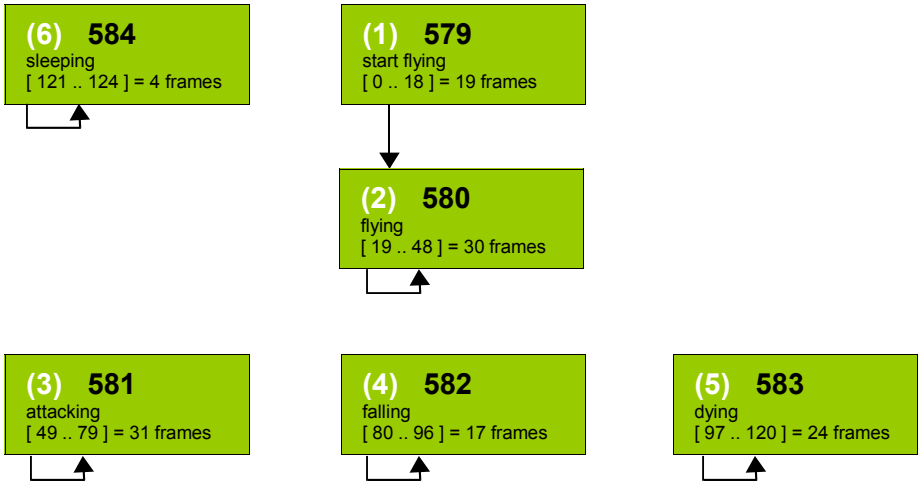


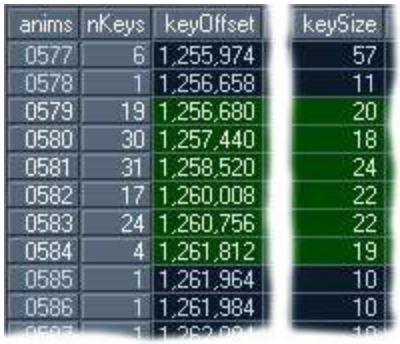
FIG. 27 - A representation of the six animation segments of the Bat with their **state_ID** numbers and each segment's number of frames, and showing the default next animations for each segment, pointed to by the black arrows.

We now have a better description of the animation segments, including their location and size, the associated state id, and the default next animations and their entry frames. But we still don't know how the Bat changes from "Flying" to "Attacking", we still don't know how the several segments connect with each other. To find out about this we need to look at two other fields, the **numStateChanges** and the **changesIndex**.

FIG. 28 - Part of the **Animations_Table** of the Bat, highlighting its 6 animation segments and concentrating in the **numStateChanges** and **changesIndex** fields.

This is as far as we can go from within the **Animations_Table**. This storyline is to be continued in another table, the **State_Changes_Table**, referenced by the two fields above.

Another complexity with the **Animations_Table** is how to access the keyframe records in the **Keyframes_Data_Package**. To find the keyframes for the animation segment **579**, “Start flying”, we need to find the **keyframeOffset** for the segment **579**, the **keyframeOffset** for the next segment, and the **keyframeSize** for the segment **579**.



anims	nKeys	keyOffset	keySize
0577	6	1,255,974	57
0578	1	1,256,658	11
0579	19	1,256,680	20
0580	30	1,257,440	18
0581	31	1,258,520	24
0582	17	1,260,008	22
0583	24	1,260,756	22
0584	4	1,261,812	19
0585	1	1,261,964	10
0586	1	1,261,984	10

FIG. 29 - Part of the **Animations_Table** of the Bat, highlighting its 6 animation segments and concentrating in the **keyframeOffset** and **keyframeSize** fields. Not included in the WAD file, my exploratory application adds a new column, labelled nKeys, number of keyframes, showing the number of keyframes for each animation segment.

Inspecting those columns in the **Animation_Table** we see that the keyframe records for the animation segment **579** are located at 1,256,680 the keyframe records for the next segment are located at 1,257,440 and the size of the keyframe records for the segment **579** is 20 words.

We’ve seen that the number of keyframes is given by the equation:

num_keyframes = (keyframeOffset[i+1] - keyframeOffset[i]) **div** (2 * keyframeSize[i])

Resolving this equation with the values found above, we have:

num_key_records_579 = (1257440 - 1256680) **div** (20 * 2) = 19

This is as far as we can go from within the **Animations_Table**. This storyline is to be continued in another table, the **Keyframes_Data_Package**, using the offsets and the number of records obtained above.

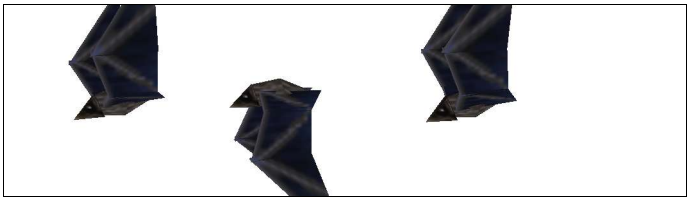
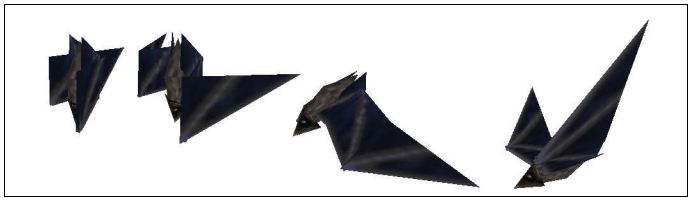


FIG. 30 - The “Flying” segment has a ground **speed** of 60 units.

The **speed** field is self-explanatory. The only segment with an important **speed** value, **60**, is the “Flying” animation. “Attacking” has a value of **3**, just enough to keep some pressure on the target.

The **acceleration** field calls for some considerations. The only segment with a value in this field is “Start flying”, animation segment **579**. In 19 frames, [0 .. 18], or rather in 20 frames if we include the first frame of the “Flying” segment **580**, the Bat goes from an initial speed of 0 to a final speed of 60. This gives an average increment of speed of 3 units per frame (60 **div** 20 = 3).



anims	speed	acceleration
0577	-31	$19999
0578	0	$00
0579	0	$2C71C
0580	60	$00
0581	3	$00
0582	0	$00
0583	0	$00
0584	0	$00
0585	0	$00
0586	0	$00

FIG. 31 – The “Start flying” animation segment **579** and part of the **Animations_Table** of the Bat model, highlighting its 6 animation segments and concentrating in the **speed** and **acceleration** fields.

If we now look into the value stored in the **Animation_Table** for segment **579**, we find the value of $2C71C, that can be converted to a floating point value by dividing it by $10000 (or 65536). This gives us a value of 2.777771, close to the value of 3 that we calculated, but not equal to it.
I’ve no idea why this difference exists.

Num_State_Changes (**uint32**).
Number of records stored in the **State_Changes_Table**.

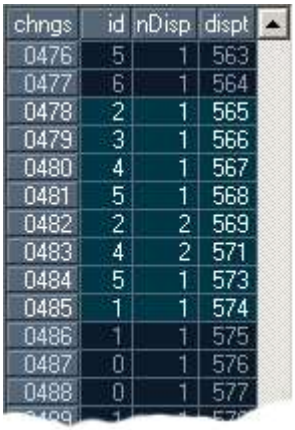
State_Changes_Table (**Num_State_Changes * 6 bytes**).
This table determines all the possible transitions between animations of different families, that were built in by the designer of the game. Description:

state_ID	(**uint16**)	ID of the state of the next animation.
numDispatches	(**uint16**)	number of animation dispatches.
dispatchesIndex	(**uint16**)	index in the dispatches table.

The **State_Changes_Table** is accessed from the **Animations_Table** only, through the two fields **numStateChanges** and **changesIndex**. The reason for accessing this table was to look for a way for changing from the *current state* to a new *target state*. The **state_ID** field is scanned for the desired target state. If it is found, processing carries on, otherwise this line of processing is abandoned and no state changes will occur.

The **State_Changes_Table** is an intermediate table. The actual target animations associated to the target state are stored in another table, the **Dispatches_Table**, which is referenced by the **numDispatches** and by the **dispatchesIndex** fields.

Continuing with the storyline started in the **Animations_Table**, using the Bat as an example, we can now look into the **State_Changes_Table** accessed from FIG. 28. The indices that were stored in the **changesIndex** column of the **Animations_Table** can now be found here, in this state changes table.



chngs	id	nDisp	dispt
0476	5	1	563
0477	6	1	564
0478	2	1	565
0479	3	1	566
0480	4	1	567
0481	5	1	568
0482	2	2	569
0483	4	2	571
0484	5	1	573
0485	1	1	574
0486	1	1	575
0487	0	1	576
0488	0	1	577

FIG. 32 - **State_Changes_Table** of the Bat, highlighting the entries referenced from the **Animations_Table**, and in turn referencing the **Dispatches_Table**.

One first detail to note is that, in the **Animations_Table**, under **numStateChanges**, we found the values 1,3,2,1,0,1 as the number of state changes to investigate. This makes it a total of 8 state changes, as given by 1 + 3 + 2 + 1 + 0 + 1, in the range **[478 .. 485]**. That's precisely what we have highlighted here in the **State_Changes_Table**.

Let's keep in mind that we are trying to find out how to change from the current state to another one. From the **Animations_Table** we gathered all the existing states for the Bat.
From the **State_Changes_Table** we gather the possible transitions to another state.
A graphical representation may help:

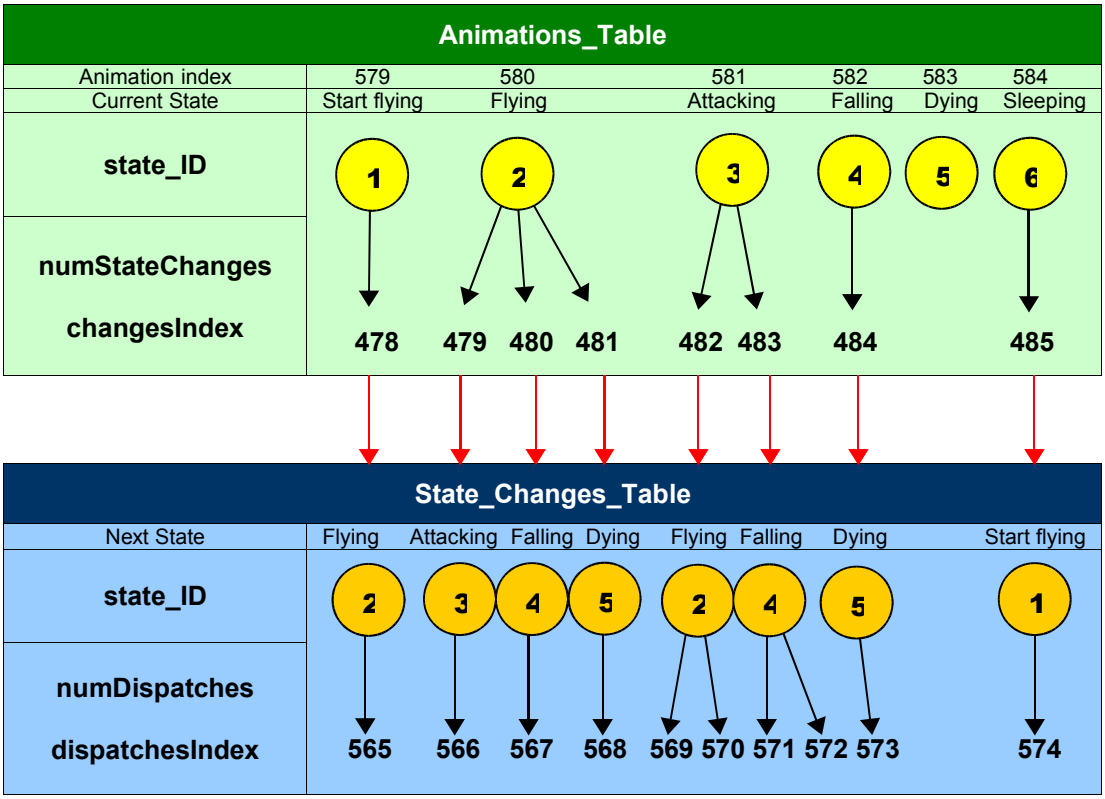


FIG. 33 - Interrelations between the **Animations_Table** and the **State_Changes_Table**, concerning the state changes.

One table contains indices to access the next table, but what matters here is the relationship between the initial states and the possible state transitions. For example, the "Flying" animation segment **580** has a **state_ID** of **2**, and has three possible transitions through indices 479, 480 and 481, leading to three possible target states, **3**, **4** and **5**. These are "Attacking", "Falling" and "Dying" (see FIG. 23). While the game was being played, something happened that caused the engine to call for a new state. For example, the Bat was flying but Lara shot him down. The engine detects the event and calls for "Falling" because the Bat was flying high above the ground. The engine starts from **state_ID** = 2 and looks for a path to **state_ID** = 4. If it finds a path, the transition will occur, otherwise the initial state is kept. In this case the engine can find a path through **changesIndex** = 480, and processing will continue to the **Dispatches_Table** through **dispatchIndex** = 567.

This is quite a complex process, but traceable.

This is as far as we can go now from within the **State_Changes_Table**. The next step involves the **Dispatches_Table**, which is described next.

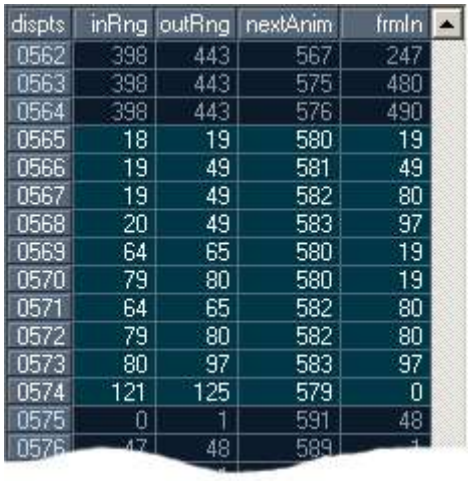
Num_Dispatches (**uint32**).
Number of records stored in the **Dispatches_Table**.

Dispatches_Table (**Num_Dispatches * 8 bytes**).
This table determines the animation segments associated to the possible next states.
This table is accessed from the **State_Changes_Table**, through the **numDispatches** and the **dispatchesIndex** fields. Description:

inRange	(**uint16**)	[frame-in] where this range starts, inclusive.
outRange	(**uint16**)	]frame-out[where this range stops, exclusive.
nextAnim	(**uint16**)	index of the next animation in the **Animations_Table**.
frameIn	(**uint16**)	[frame-in] index of the next animation.

The **Dispatches_Table** defines a range of frames belonging to the *current animation segment*, defined as **[inRange, outRange [**, where the frame-in is inclusive and the frame-out is exclusive.
If the engine reached this point, that's because a path to the desired next state was found, and the engine is trying to find the index to the **nextAnim** and its **frameIn**.
However, there is one more obstacle in the engine's way.
In order to obtain the **nextAnim** from the **Dispatches_Table**, it is necessary that the *current frame* of the *current animation segment* be within the limits of **[inRange, outRange [**.
If it is, then the game will change to the new animation, otherwise the game will carry on with the same current animation and its default next animation, as defined in the **Animations_Table**.

Continuing with the storyline that uses the Bat as an example, we can now look into the **Dispatches_Table** accessed from FIG. 32. The indices that were stored in the **dispatchesIndex** column of the **State_Changes_Table** can now be found here, in this dispatches table.



dispts	inRng	outRng	nextAnim	frmIn
0562	398	443	567	247
0563	398	443	575	480
0564	398	443	576	490
0565	18	19	580	19
0566	19	49	581	49
0567	19	49	582	80
0568	20	49	583	97
0569	64	65	580	19
0570	79	80	580	19
0571	64	65	582	80
0572	79	80	582	80
0573	80	97	583	97
0574	121	125	579	0
0575	0	1	591	48
0576	47	48	589	1

FIG. 34 – **Dispatches_Table** of the Bat model, highlighting the entries referenced from the **State_Changes_Table**, and in turn referencing the **Animation_Table**.

It must be noted that, in the **State_Changes_Table**, under **numDispatches**, we found the values of 1,1,1,1,2,2,1,1 as the number of dispatches for each state change. This makes it a total of 10 dispatches in the range **[565 .. 574]**. That's precisely what we have highlighted here.

The graphical representation can now be completed:

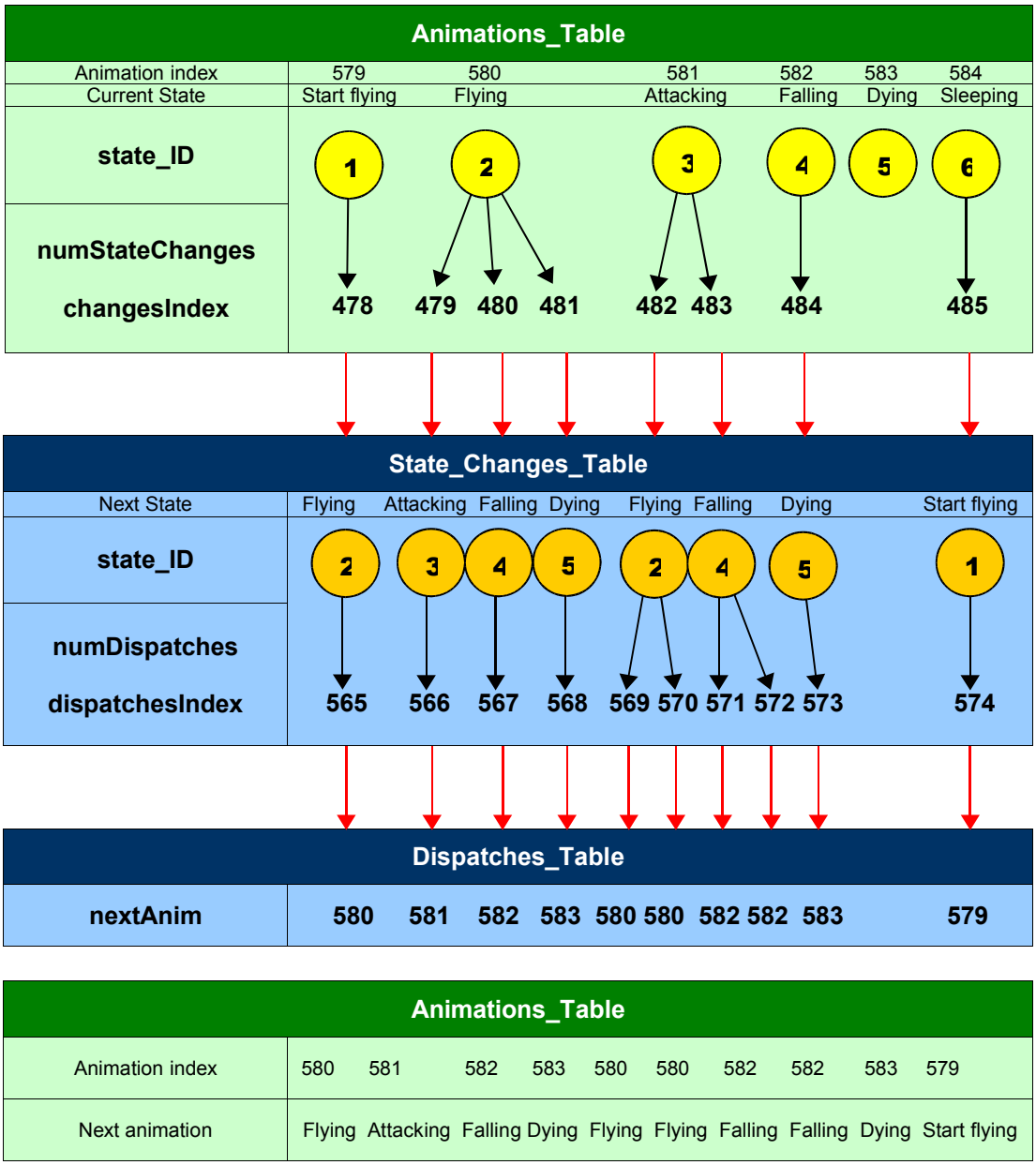


FIG. 35 - Interrelations between the **Animations_Table**, the **State_Changes_Table**, and the **Dispatches_Table**, concerning the state changes and their related animation segments.

So the Bat was using the “Flying” animation segment **580**, which has a **state_ID** = **2** and Lara shot him down. Detecting the event, the engine issues out a request for the “Falling” animation, which has a **state_ID** = **4**. That’s because the Bat was flying high and must fall down before dying on the ground. Another possibility would be the Bat flying low, in which case the engine would ask for the “Dying” animation segment, **state_ID** = **5**.

Looking into the **Animations_Table**, locating the proper entry for the Bat where **state_ID** = **2**, the current state, the engine finds three possibilities with indices from 479. Scanning through these three possibilities in the **State_Changes_Table**, the first one is rejected because it leads to a

state_ID = **3**, when the engine is looking for a **4**. Then the engine scans the second possibility and finds a **state_ID** = **4** under index 480. That's a match.

The **State_Changes_Table** has one single dispatch associated to index 480. This dispatch is to be found at index 567 of the Dispatches_Table.

Carrying on to the **Dispatches_Table**, the engine looks into index 567 of that table.
It finds a range of frames, **[19 .. 49 [**, and compares the current frame of the current animation against that range. If the current frame is within that range, the engine gets to **nextAnim** = **582** and to **frameIn** = 80. The animation segment **582** is the "Falling" animation. If the current frame is outside that range, the comparison test yields FALSE and the current animation continues until a comparison test yields TRUE.

After a successful change of state, the new animation becomes the current animation and the new state becomes the current state, and the storyline starts all over again.

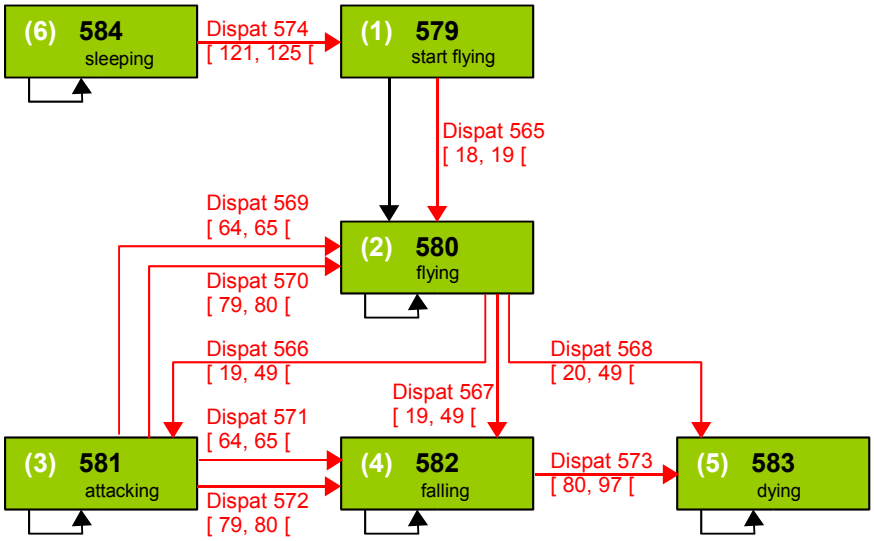


FIG. 36 – A complete description of all the possible transitions between the states and animation segments of the Bat. The default next animations are represented by the **black arrows**, the dispatched next animations are represented by the **red arrows**.

Commands_Num_Words (**uint32**).
Size of the **Commands_Data_Package**, expressed in **word** (**uint16**) units. Given that a **word** takes 2 bytes, the total number of bytes used to store the commands data is given by:

package_size_in_words = **Commands_Num_Words**

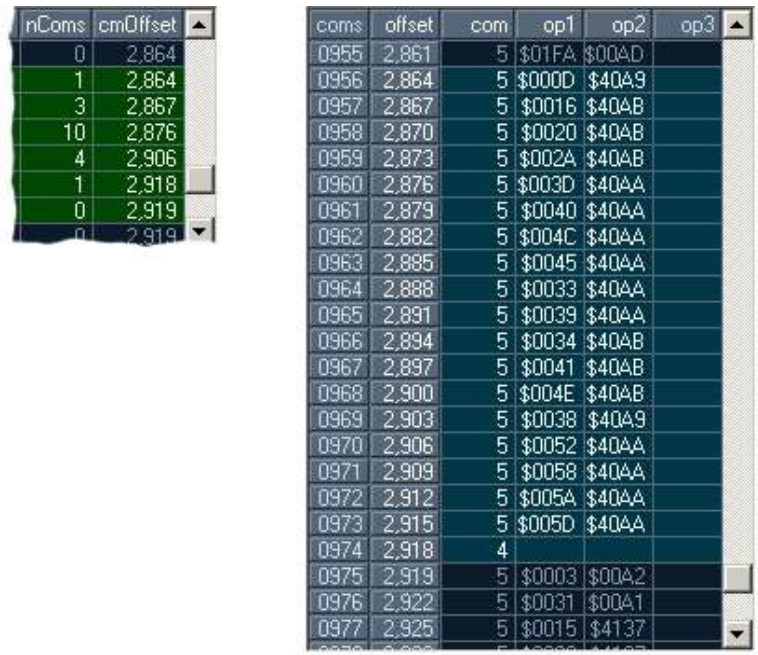
package_size_in_bytes = 2 * **Commands_Num_Words**

Commands_Data_Package (**Commands_Num_Words * 2 bytes**).
This package stores all the commands for all the animation segments in the **Animations_Table**. By design, certain events are associated to certain frames. These events are coded into records with different structures, depending on the event itself. The records having a variable size, there is no way to tell how many commands the package has. It needs to be parsed.
The structure of each command record is the following:

command	(**uint16**)	command's code.
operator1	(***uint16***)	*first operator, if applicable.*
operator2	(***uint16***)	*second operator, if applicable.*
operator3	(***uint16***)	*third operator, if applicable.*

The "*if applicable*" is there precisely because the record has a variable size. Some commands have no operators, some have two or three.

A quick run made with my exploratory application revealed that the valid commands are [**1**, **2**, **3**, **4**, **5**, **6**] as reported in the TRosettaStone. All the original WAD files were inspected. My exploratory application has a built-in alert in case any unknown commands are found in the **Commands_Data_Package**. The only WAD file that fired the alert was **KARNAK**, which showed a weird command structure. All the other showed a standard structure.



nComs	cmOffset
0	2,864
1	2,864
3	2,867
10	2,876
4	2,906
1	2,918
0	2,919
0	2,919

coms	offset	com	op1	op2	op3
0955	2,861	5	$01FA	$00AD	
0956	2,864	5	$000D	$40A9	
0957	2,867	5	$0016	$40AB	
0958	2,870	5	$0020	$40AB	
0959	2,873	5	$002A	$40AB	
0960	2,876	5	$003D	$40AA	
0961	2,879	5	$0040	$40AA	
0962	2,882	5	$004C	$40AA	
0963	2,885	5	$0045	$40AA	
0964	2,888	5	$0033	$40AA	
0965	2,891	5	$0039	$40AA	
0966	2,894	5	$0034	$40AB	
0967	2,897	5	$0041	$40AB	
0968	2,900	5	$004E	$40AB	
0969	2,903	5	$0038	$40A9	
0970	2,906	5	$0052	$40AA	
0971	2,909	5	$0058	$40AA	
0972	2,912	5	$005A	$40AA	
0973	2,915	5	$005D	$40AA	
0974	2,918	4			
0975	2,919	5	$0003	$00A2	
0976	2,922	5	$0031	$00A1	
0977	2,925	5	$0015	$4137	

FIG. 37 – **Commands_Table** of the Bat, highlighting the commands referenced in the **Animations_Table**.

So far the known commands and respective operators are, as extracted from the TRosettaStone and from TRWad's manual for the Animation Editor in WADMerger:

Command	Operator1	Operator2		Operator3	DESCRIPTION
1	**x**	**y**		**z**	Position reference
2	**x or y**	**y or z**		**-**	Jumping reference
3	**-**	**-**		**-**	Slaved animations, guns
4	**-**	**-**		**-**	Some death animations
5	**frame #**	**sound ID**		**-**	Play sound at frame
6	**frame #**	**$0000**	**0**	**-**	Change direction 180 deg.
	"	**$0001**	**1**	**-**	Soft earthquake
	"	**$0002**	**2**	**-**	Play flooding sound
	"	**$0003**	**3**	**-**	Make bubble
	"	**$0004**	**4**	**-**	End level
	"	**$0005**	**5**	**-**	Activate camera trigger
	"	**$0006**	**6**	**-**	Activate triggers
	"	**$0007**	**7**	**-**	Heavy earthquake
	"	**$0008**	**8**	**-**	Get crowbar
	"	**$000B**	**11**	**-**	Soft earthquake
	"	**$000C**	**12**	**-**	Disable guns
	"	**$000E**	**14**	**-**	Get right gun
	"	**$000F**	**15**	**-**	Get left gun
	"	**$0010**	**16**	**-**	Fire right gun
	"	**$0011**	**17**	**-**	Fire left gun
	"	**$0012**	**18**	**-**	MESHSWAP1
	"	**$0013**	**19**	**-**	MESHSWAP2
	"	**$0014**	**20**	**-**	MESHSWAP3
	"	**$0015**	**21**	**-**	?
	"	**$0016**	**22**	**-**	?
	"	**$0017**	**23**	**-**	Hide object
	"	**$0018**	**24**	**-**	Show object
	"	**$001A**	**26**	**-**	Remove ponytail
	"	**$0020**	**32**	**-**	?
	"	**$0025**	**37**	**-**	?
	"	**$0026**	**38**	**-**	?
	"	**$002B**	**43**	**-**	Get waterskin
	"	**$002C**	**44**	**-**	Put back waterskin
	"	**$4020**	**16416**	**-**	Play step sound, land
	"	**$8020**	**-32736**	**-**	Play step sound, water

Links_Num_DWords (**uint32**).
Number of integers (**sint32**) in the **Links_Data_Package**.

total_size_in_bytes = **Links_Num_DWords** * 4

Links_Data_Package (**Links_Num_DWords * 4 bytes**).
The integers in the package are organized in records, each record being a **Pivot Link** consisting of *four* **sint32** integers describing the *hierarchy* and the *relative offsets* of the *pivot points* for a 3D model. Description of each record:

opCode	(**sint32**)	stack operation code.
dx	(**sint32**)	mesh offset in x.
dy	(**sint32**)	mesh offset in y.
dz	(**sint32**)	mesh offset in z.

The **opCode** takes the values **0**, **1**, **2**, **3**, where:

0 = stack not used. Link the current mesh to the previous mesh.
1 = pull the parent from the stack. Link the current mesh to the parent.
2 = push the parent into the stack. Link the current mesh to the parent.
3 = read the parent from the stack. Link the current mesh to the parent.

A stack of meshes is used to “push”, “pull” or “read” meshes, to remember which is the parent to the current mesh. The **opCode** defines what to do with the stack. Both “push” and “pull” modify the stack, “read” does not.

The **dx**, **dy** and **dz** offsets define where the pivot point of the current mesh is placed, relative to the pivot point of the parent mesh. For each movable, the sequence of the link records is taken from the **Links_Data_Package** and the sequence of meshes is found through the sequence of pointers in the **Mesh_Pointers_List**.

The first mesh found through the pointers list is taken as the *root mesh*, the one to which all the other ones are connected, directly or indirectly. The world-offset of the root mesh is not included in the **Links_Data_Package**, it is defined elsewhere in the **Keyframes_Package**.

These links are referenced in the **Movables_Table** by the index of their first integer, not by their byte address. At four integers per record, the total number of records is given by[9]:

num_data_indices = **Links_Num_DWords**

num_records = num_data_indices **div** 4

[9] *I’ve found custom WAD files, processed by WadMerger, where the sequence of records is broken and the formula above is not valid. At the end of a sequence of records there is some garbage, which moves the next records further down, placing them off the “multiple of 4” table scheme. The records can still be found and used correctly by TRLE because their indices are still properly stored in the* ***Movables_Table****, which is the proper entrance door to the* ***Links_Data_Package****.*
The original Eidos/Core WAD files do not have this problem, and the formula above is valid with them.

How does it work? Taking Lara as an example, her body is made of **15 Body Meshes**, *not* including the PONYTAIL which is a separate object:

HIPS (root)
- **L**EFT THIGH
  - **L**EFT SHIN
    - **L**EFT FOOT
- **R**IGHT THIGH
  - **R**IGHT SHIN
    - **R**IGHT FOOT
- **T**ORSO
  - **R**IGHT INNER ARM
    - **R**IGHT OUTER ARM
      - **R**IGHT HAND
  - **L**EFT INNER ARM
    - **L**EFT OUTER ARM
      - **L**EFT HAND
  - **H**EAD

These **15** meshes are connected through **14 Links**.
There are also **Joint Meshes**, special meshes designed to fill the spaces left open between the 15 Body Meshes. We would expect these Joint Meshes to be 14, taking the same locations as the 14 Links themselves. But they are in fact 15. There is one more with the same coordinates as the HIPS, to function as a root for the other Joint Meshes.

HIPS (root)
- **L**EFT HIP
  - **L**EFT KNEE
    - **L**EFT ANKLE
- **R**IGHT HIP
  - **R**IGHT KNEE
    - **R**IGHT ANKLE
- **P**ELVIS
  - **R**IGHT SHOULDER
    - **R**IGHT ELBOW
      - **R**IGHT WRIST
  - **L**EFT SHOULDER
    - **L**EFT ELBOW
      - **L**EFT WRIST
  - **N**ECK

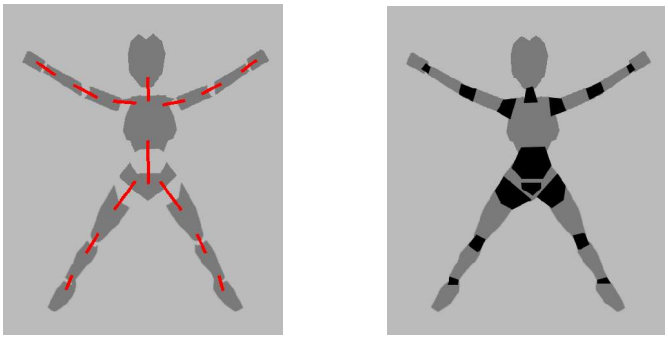


FIG. 38 – The 15 **Body Meshes**, the 14 **Links**, the **15 Joint Meshes** in Lara's body.

The **Links_Data_Package** and the **Mesh_Pointers_List**, taken together, allow us to define a hierarchical skeleton for the 3D models in the **Movables_Table**.

From the movables table we get the number of mesh pointers. This tells us how many meshes the model has, and how many links (links = meshes – 1). Through the mesh pointers list we get to the meshes themselves. Through the links data package we get the hierarchical information for a skeletal representation of the model.

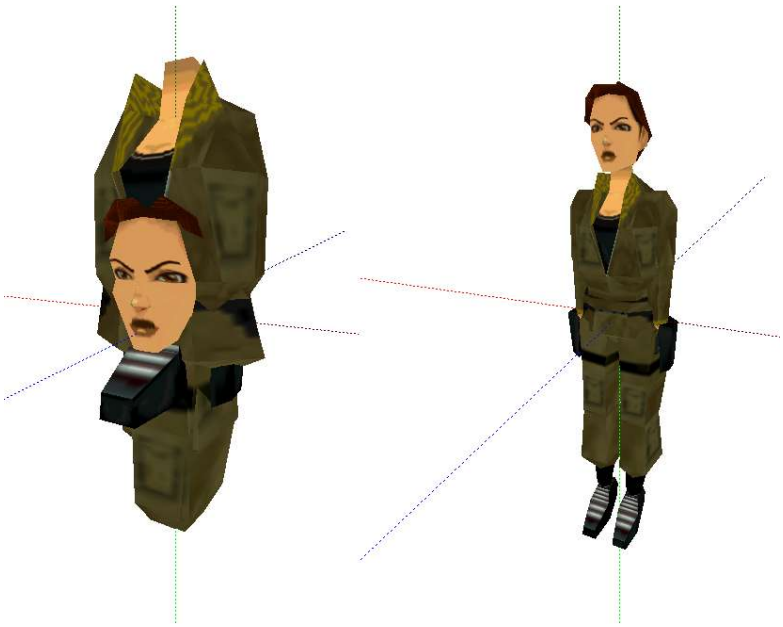


FIG. 39 – The **Mesh_Pointers_List** supplies the meshes, but it is the **Links_Data_Package** that places them in their proper locations, connected the proper way. *(custom wad by Litepulsar - Stargate - from Lara's Level Base).*

links	index	op	dx	dy	dz
0112	0448	2	-42	20	2
0113	0452	0	9	185	3
0114	0456	0	1	192	-8
0115	0460	3	43	20	2
0116	0464	0	-10	185	2
0117	0468	0	0	193	-2
0118	0472	1	-1	-49	11
0119	0476	2	59	-142	-12
0120	0480	0	10	95	-2
0121	0484	0	1	101	-1
0122	0488	3	-57	-143	-12
0123	0492	0	-9	98	-2
0124	0496	0	-1	99	-1
0125	0500	1	2	-198	-23
0126	0504	2	-42	20	2
0127	0508	0	9	185	3
0128	0512	0	1	192	-8
0129	0516	3	43	20	2
0130	0520	0	-10	185	2
0131	0524	0	0	193	-2
0132	0528	1	-1	-49	11

FIG. 40 – The **Links_Data_Package** is addressed by the index of the first **sint32** of the link's record. The record stores the hierarchical operation code followed by the three coordinates of the **pivot point** relative to its parent.

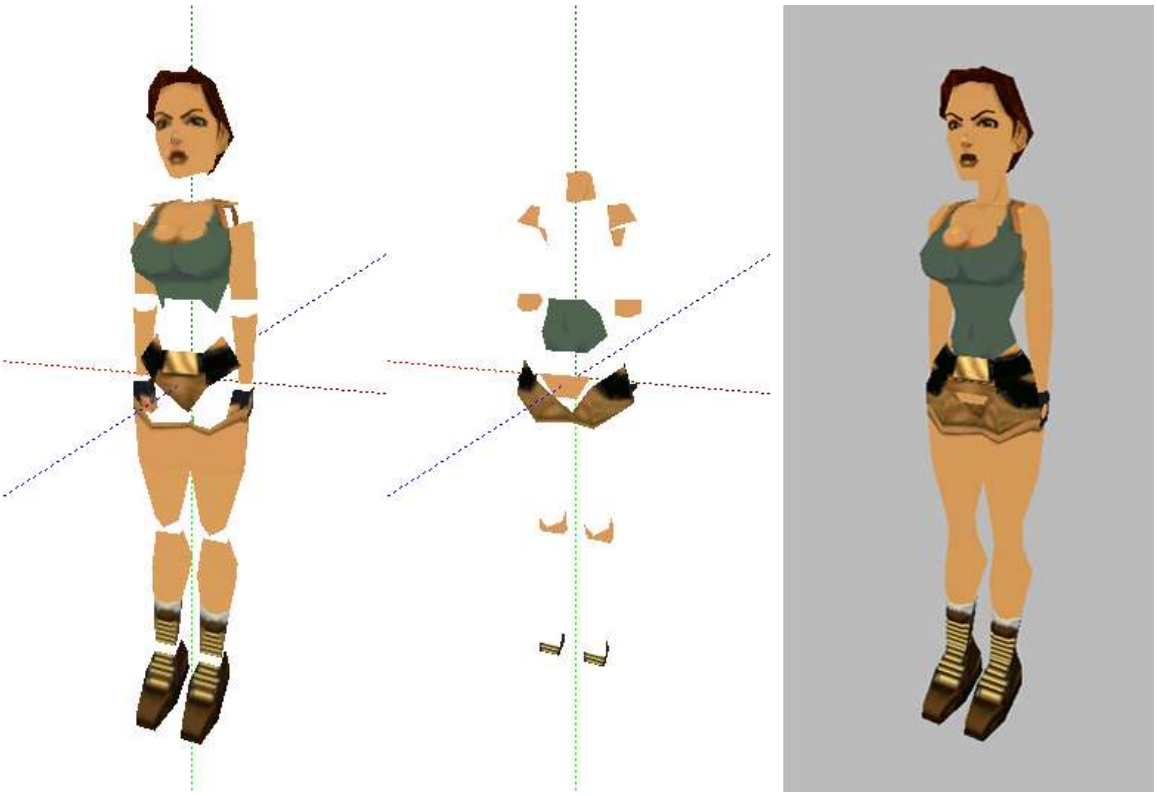


FIG. 41 – Lara's **Body Meshes** and the correspondent **Joint Meshes**. Combined together they constitute the final **Skin**.

Considering only the sequence of **14** **opCodes**, a typical sequence for any of the above **links** is:

2 0 0 3 0 0 1 2 0 0 3 0 0 1

Translation of this sequence, applied to the **15** Lara's **Body Meshes**:

- Take the HIPS mesh and use it as the root mesh.
- **2** = push the HIPS into the stack. Link the LEFT THIGH to the HIPS and offset (dx, dy, dz).
- **0** = stack not used. Link the LEFT SHIN to the LEFT THIGH and offset (dx, dy, dz).
- **0** = stack not used. Link the LEFT FOOT to the LEFT SHIN and offset (dx, dy, dz).
- **3** = read the HIPS from the stack. Link the RIGHT THIGH to the HIPS and offset (dx, dy, dz).
- **0** = stack not used. Link the RIGHT SHIN to the RIGHT THIGH and offset (dx, dy, dz).
- **0** = stack not used. Link the RIGHT FOOT to the RIGHT SHIN and offset (dx, dy, dz).
- **1** = pull the HIPS from the stack. Link the TORSO to the HIPS and offset (dx, dy, dz).
- **2** = push the TORSO into the stack. Link the RIGHT INNER ARM to the TORSO and offset (dx, dy, dz).
- **0** = stack not used. Link the RIGHT OUTER ARM to the RIGHT INNER ARM and offset (dx, dy, dz).
- **0** = stack not used. Link the RIGHT HAND to the RIGHT OUTER ARM and offset (dx, dy, dz).
- **3** = read the TORSO from the stack. Link the LEFT INNER ARM to the TORSO and offset (dx, dy, dz).
- **0** = stack not used. Link the LEFT OUTER ARM to the LEFT INNER ARM and offset (dx, dy, dz).
- **0** = stack not used. Link the LEFT HAND to the LEFT OUTER ARM and offset (dx, dy, dz).
- **1** = pull the TORSO from the stack. Link the HEAD to the TORSO and offset (dx, dy, dz).

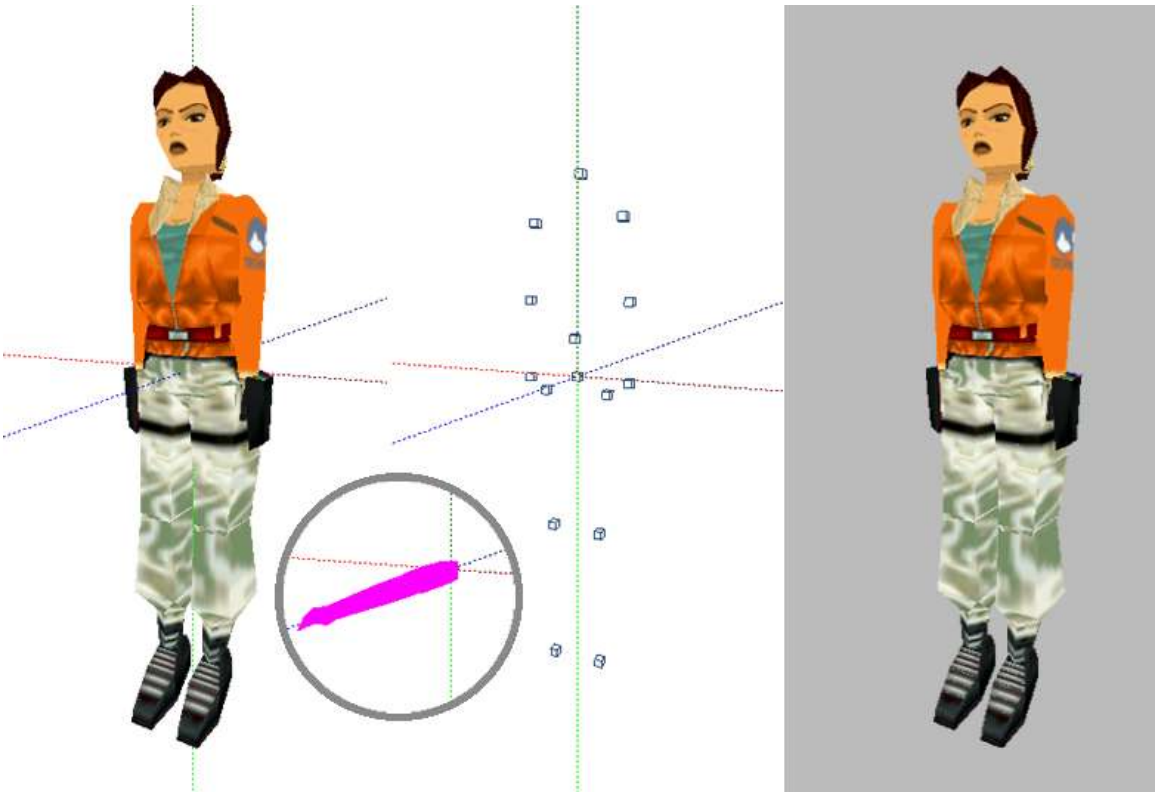


FIG. 42 - Another solution for Lara's body: The **Body Meshes** also cover the **Joints**, the **Joint Meshes** are NULL, in the insert the **Ponytail** is shown textured with **magenta**.

There are other solutions for the **Links_Data_Package**. The image above shows a custom WAD file displaying an **ARTIC** outfit. In this case the **Body Meshes** are built in such a way that they also cover the **Joints**, no need for extra **Joint Meshes**. However, the **Joint Meshes** must be present in the **Links_Data_Package**. One way for solving this is to declare the **Joint Meshes**, even assigning a Bounding Sphere, but insert them as zero-vertices meshes. As dummies, as NULL meshes. In the outfit above we can see that this NULL solution was implemented. Another possibility would be to maintain the **Joint Meshes** but map them with **magenta**, therefore making them invisible. As we can see in the insert in the outfit above, this other solution was implemented for the Ponytail. Lara still has it, but it is invisible.

We've seen how a complex Lara's body is handled by the stack of pivot points.
But what happens if the Movable has *one mesh only*? And there are plenty of those.
The one-mesh is considered to be the root mesh and has no entry in the **Links_Data_Package**. No children, no joints. Quite logical.
15 meshes have 14 joints, 2 meshes have 1 joint, 1 mesh has zero joints.
One-mesh-movables do not need entries in the links package, but the **linkIndex** field still exists in the **Movables_Table**, so something, just something, must be put there. It will not use it, anyway!
A variation of this reasoning is found in custom WAD files processed by WadMerger: it places a *zero* in the **linkIndex** of the **Movables_Table**. This is also valid, because the index is not used.

One more issue, about the internal structure of the **Links_Data_Package**. The original WAD files from Eidos / Core Design have a well-behaved internal structure. The value of **Links_Num_DWords** is a multiple of four integers, the **Link** records are located at indices which are a multiple of four. But the custom WAD files processed by WadMerger reveal another possibility. TRLE accepts that the link be located at any index, it does not expect a multiple of four. Where the original WAD file has a link at index 56, a custom WAD file may have the same link at index 59.

The several **Links** for a given Movable are still packed as contiguous, but the starting point for each pack may be located at some random index and there may be garbage in-between the packs.

links	index	op	dx	dy	dz
0000	0000	2	-42	20	2
0001	0004	0	9	185	3
0002	0008	0	1	192	-8
0003	0012	3	43	20	2
0004	0016	0	-10	185	2
0005	0020	0	0	193	-2
0006	0024	1	-1	-49	11
0007	0028	2	59	-142	-12
0008	0032	0	10	95	-2
0009	0036	0	1	101	-1
0010	0040	3	-57	-143	-12
0011	0044	0	-9	98	-2
0012	0048	0	-1	99	-1
0013	0052	1	2	-198	-23
	0056	2	-42	20	
0014	0059	2	-42	20	2
0015	0063	0	9	185	3
0016	0067	0	1	192	-8
0017	0071	3	43	20	2
0018	0075	0	-10	185	2
0019	0079	0	0	193	-2
0020	0083	1	-1	-49	11
0021	0087	2	59	-142	-12
0022	0091	0	10	95	-2
0023	0095	0	1	101	-1
0024	0099	3	-57	-143	-12
0025	0103	0	-9	98	-2
0026	0107	0	-1	99	-1
0027	0111	1	2	-198	-23
	0115	2	-42	20	
0028	0118	2	-42	20	2
0029	0122	0	9	185	3
0030	0126	0	1	192	-8
0031	0130	3	43	20	2
0032	0134	0	-10	185	2

FIG. 43 – To accommodate the WadMerger's variation, which leaves three extra bytes between the Links, my exploratory application creates an extra slot *without* a link number. It only shows the link index for the extra bytes.

Building up the **opCodes** for the **Links_Data_Package** calls for some caution. The following examples, extracted from **KARNAK**, illustrate the subject:

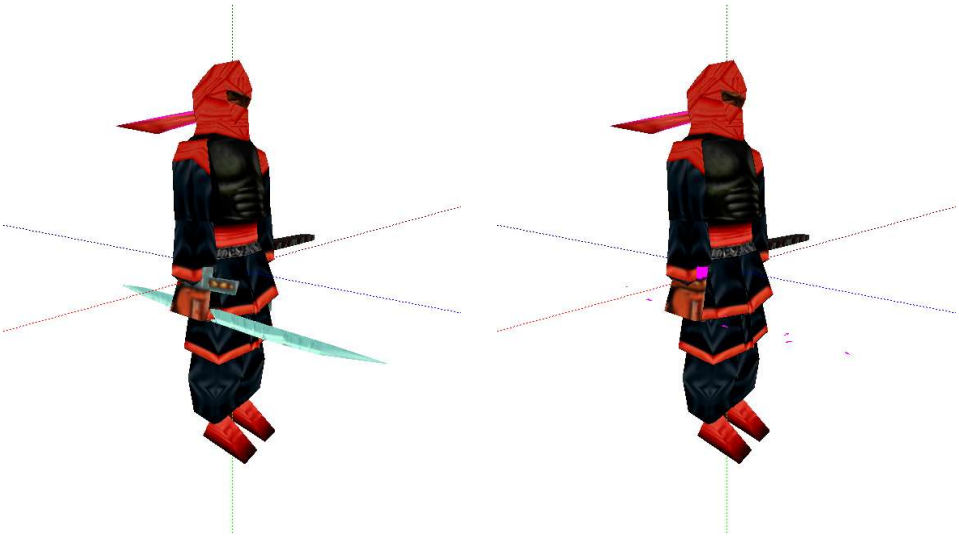


FIG. 44 - The same baddie, with and without weapons. They are in fact exactly the same model, but in the no-weapons baddie the geometry for the weapons was reduced to a few triangles, and even those were mapped with magenta, rendering them invisible. No sword, no uzi. The Links for these baddies could be the same. However, Movable 043 (on the left) has 27 mesh pointers and Movable 418 (on the right) has 28 mesh pointers. The no-weapons baddie has one extra mesh.

Considering only the sequence of **26** opCodes, of Movable 043 :

2 0 0 3 0 0 3 1 2 0 0 3 0 0 2 0 3 0 3 0 1 0 1 0 0 0

This is a well-behaved package. The number of PULL equals the number of PUSH.
This is not always the case, as we can see next.
Now considering only the sequence of **27** opCodes, of Movable 418 :

2 0 0 3 0 0 3 1 2 0 0 3 0 0 2 0 3 0 3 0 1 0 1 0 0 0 1

This is *not* a well-behaved package. There is an extra PULL in the end of the sequence.
It pulls what? Application developers must be aware of this situation. Existing this extra PULL, the stack must be able to keep track of the root mesh to supply it in cases like this.

Now, a skeleton for a four-legged model. The example below is Movable 051 in **COASTAL**.

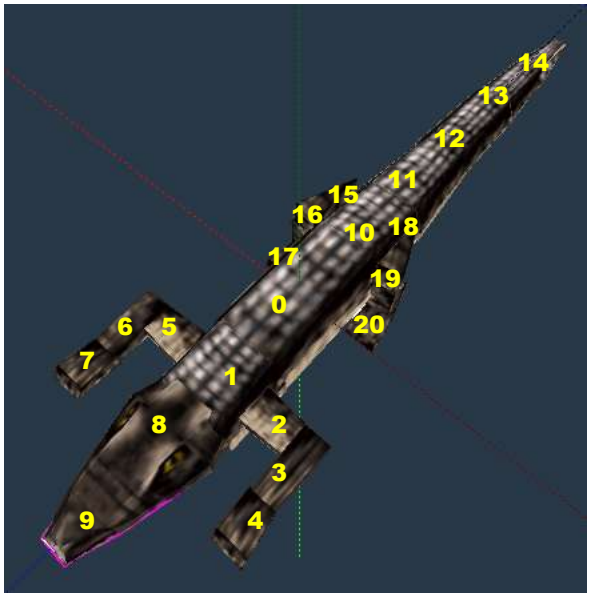


FIG. 45 - The numbers represent the steps for the construction of the body of the crocodile, with 21 meshes [0 .. 20] and 20 joints.

Considering the sequence of opCodes:

- = Take the piece(**0**) and use it as the root mesh.

2 = **PUSH** piece(**0**) into the stack and link piece(1) to it.

2 0 0 = **PUSH** piece(**1**) into the stack and link piece(2) to it, then chain (3) and (4).

3 0 0 = Read piece(**1**) from the stack and link piece(5) to it, then chain (6) and (7).

1 0 = **PULL** piece(**1**) from the stack and link piece(8) to it, then chain (9).

1 = **PULL** piece(**0**) from the stack and link piece(10).

2 0 0 0 = **PUSH** piece(**10**) into the stack and link piece(11) to it, chain (12), (13) and (14).

3 0 0 = Read piece(**10**) from the stack and link piece(15) to it, then chain (16) and (17).

1 0 0 = **PULL** piece(**10**) from the stack and link piece(18) to it, chain (19) and (20).

Keyframes_Num_Words (**uint32**).
Size of the **Keyframes_Data_Package**, expressed in **word** (**uint16**) units. Given that a **word** takes 2 bytes, the total number of bytes used to store the keyframes data is given by:

package_size_in_words = **Keyframes_Num_Words**

package_size_in_bytes = 2 * **Keyframes_Num_Words**

Keyframes_Data_Package (**Keyframes_Num_Words * 2 bytes**).
This package stores all the bounding boxes, root mesh offsets and pivot angles for all the animation segments in the **Animations_Table**. Description:

bb1x	(**sint16**)	coordinate, bounding box.
bb2x	(**sint16**)	coordinate, bounding box.
bb1y	(**sint16**)	coordinate, bounding box.
bb2y	(**sint16**)	coordinate, bounding box.
bb1z	(**sint16**)	coordinate, bounding box.
bb2z	(**sint16**)	coordinate, bounding box.
offx	(**sint16**)	coordinate, root mesh offset.
offy	(**sint16**)	coordinate, root mesh offset.
offz	(**sint16**)	coordinate, root mesh offset.
keys	(**variable words**)	package of pivot point angles.

Yes, another unfortunate situation in the WAD file format. Another one. The sequence of angles has an undefined size because some angles will be coded into one **word**, some will be coded into a **dword**. We know the number of angles from the number of meshes in the movable model, we know the total number of words taken by the angles from the animation table, but we do not know how that number of angles uses the available space. The angles package needs to be parsed. Most likely there will be some padding after the angles, some unused words. My exploratory application shows such padding. Probably the extra space is just a safeguard against a worst case of angles coding.

This package can be accessed from the **Movables_Table** or from the **Animations_Table**. Being a package, it needs to be accessed by offset.

keyframes	at offset	bytes	nMsh	links	nAng	nPad	bb1x	bb2x	bb1y	bb2y	bb1z	bb2z	offx	offy	offz	data01	data02	data03	data04	d
16421	1,256,544	114	25	1164	25	0	-333	320	-683	16	-914	814	-11	-440	251	$001BF417	$3EFFEC6B	$0803FC00	$008C2F8D	$0010
16422	1,256,658	22	2	1260	2	0	-35	35	-40	-4	-55	51	0	0	0	$C000	$C000	-	-	
16423	1,256,680	40	8	1264	8	2	-35	37	-180	-82	-74	7	0	-140	-14	$302057EB	$CB71	$C6EE	$C88F	$
16424	1,256,720	40	8	1264	8	2	-35	36	-180	-82	-76	7	0	-140	-14	$302057EB	$CB68	$C6F7	$C8DC	$
16425	1,256,760	40	8	1264	8	2	-34	36	-180	-82	-78	7	0	-140	-14	$302057EB	$CB52	$C70F	$C90C	$
16426	1,256,800	40	8	1264	8	2	-33	35	-180	-82	-76	7	0	-140	-14	$302057EB	$CB37	$C72B	$C8DD	$
16427	1,256,840	40	8	1264	8	2	-33	34	-180	-82	-73	8	0	-140	-14	$302057EB	$CB1F	$C745	$C89A	$
16428	1,256,880	40	8	1264	8	2	-32	34	-180	-82	-73	8	0	-140	-14	$302057EB	$CB11	$C755	$C89A	$
16429	1,256,920	40	8	1264	8	2	-32	34	-180	-82	-74	8	0	-140	-14	$302057EB	$CB14	$C750	$C8BC	$
16430	1,256,960	40	8	1264	8	2	-33	34	-180	-82	-75	7	0	-140	-14	$302057EB	$CB32	$C731	$C89A	$
16431	1,257,000	40	8	1264	8	2	-35	37	-180	-82	-74	7	0	-140	-14	$302057EB	$CB71	$C6EE	$C89A	$
16432	1,257,040	40	8	1264	8	1	-61	62	-180	-82	-77	7	0	-140	-14	$302057EB	$CCFD	$3FF0015B	$CA32	$
16433	1,257,080	40	8	1264	8	2	-109	110	-179	-82	-56	13	0	-140	-14	$302057EB	$CEA4	$C382	$CBB7	$
16434	1,257,120	40	8	1264	8	0	-116	119	-179	-81	-66	4	0	-140	-14	$302057EB	$CEFE	$000FFC90	$3FFFFF1F	$
16435	1,257,160	40	8	1264	8	0	-107	113	-161	-65	-93	3	0	-122	-14	$322FF008	$CF14	$000FF83F	$3FFFFB47	$
16436	1,257,200	40	8	1264	8	0	-88	95	-142	-51	-115	9	0	-105	-14	$341FF40C	$CF0B	$000FF7F6	$3FFFFB69	$
16437	1,257,240	40	8	1264	8	0	-72	79	-120	-29	-122	17	0	-87	-14	$360FFC0F	$CF07	$000FF7BF	$3FFFF79B	$
16438	1,257,280	40	8	1264	8	0	-81	83	-98	-3	-112	25	0	-70	-14	$38000411	$CF2C	$000FF7A2	$000FF7B2	$
16439	1,257,320	40	8	1264	8	0	-125	120	-75	9	-84	32	0	-52	-14	$39F00C14	$CF8B	$000FF7B0	$000FF7DE	$
16440	1,257,360	40	8	1264	8	0	-153	152	-54	-5	-51	37	0	-34	-14	$3BE01817	$C00B	$000FFBE7	$000FF80C	$
16441	1,257,400	40	8	1264	8	0	-103	129	-126	2	-53	41	0	-17	-14	$3DE0281B	$C09A	$000FFC2D	$000FFC3B	$
16442	1,257,440	36	8	1264	8	0	-42	61	-131	14	-53	41	0	1	-14	$3FD0341F	$C121	$C1AF	$C1AB	$
16443	1,257,476	36	8	1264	8	0	-85	124	-119	8	-53	42	1	-5	-14	$00104824	$C027	$C100	$C1A4	
16444	1,257,512	36	8	1264	8	0	-142													

FIG. 46 – Keyframes Package for a Bat, highlighting some keyframes. Only the body of the table is stored in the WAD file. The seven left columns are implemented by my exploratory application.

Most of the data in the **Keyframes_Data_Package** is quite easy to decode. The bounding box and the root mesh offset are self-explanatory. The angles data is not so obvious. Several steps are needed to decode the angles. First we go to the **Movables_Table** to find out about the number of meshes, stored under **numPointers**. Then we go to the **Animations_Table** to find out about the size of the keyframe record, in **word** units, stored under **keyframeSize**. From the number of words of the keyframe we subtract 9 words for the bounding box and the root mesh offset. The remaining words will be shared by the coded angles and by the padding words.

Up to this point we only know how many they are and where they are, per keyframe record.
Being angles related to pivot points, the angles are coded as sets, coding the rotation in X,Y,Z.

```
set  =  [ rotateX, rotateY, rotateZ ]
```

It can be expected that decoding the angle sets will return those three values. In fact this is one of the two ways for coding the angle sets. But there are situations where only one of those rotations can happen, therefore only one of those components changes and the other components have a value of zero. As a mere example, if the knee can only rotate in X, then Y and Z are zero and there is no need to code them. For similar cases there is another way for coding angle sets that assumes that only one rotation is coded.

```
set  =  [ rotateX, 0, 0 ]
set  =  [ 0, rotateY, 0 ]
set  =  [ 0, 0, rotateZ ]
```

We have four different possibilities. From all this derives the following: if the angle set specifies a three-axes rotation, a **uint32** is used, if the angle specifies a one-axis rotation, a **uint16** is used. When reading through the angles package, one word at a time, we need to check the most significant two bits of the word, to find the proper axes type.

```
axes  =  angle_set  and  $C000
```

If axes = **$0000** then it is a three-axes rotation, uses a **uint32**.
If axes = **$4000** then it is a one-axis rotation in X, uses a **uint16**.
If axes = **$8000** then it is a one-axis rotation in Y, uses a **uint16**.
If axes = **$C000** then it is a one-axis rotation in Z, uses a **uint16**.

If axes = **$0000**, a three-axes rotation that uses a **uint32**, then we must read the next word and combine it with the one we already have.

```
angle_set  =  angle_set  *  $10000  +  next_word
```

Out of these 32 bits, 2 were used to code the axes, the remaining 30 bits are used to code the three rotations. 10 bits per rotation. Given angle_set as a **uint32**, we extract the rotations like this:

```
rotationZ      =  angle_set  and  $3FF
angle_set      =  angle_set  shr  10
rotationY      =  angle_set  and  $3FF
angle_set      =  angle_set  shr  10
rotationX      =  angle_set  and  $3FF
```

Considering the other situations, one-axis rotation only, given angle_set as a **uint16**, out of these 16 bits, 2 were used to code the axis, the remaining 14 bits are used to code the rotation:

rotation = angle_set **and** $3FFF

The meaning of this value depends on the two most significant bits, as shown before.
Finally, these values must be converted into regular angle values. The conversion factor is:

1024 = 360 degrees for a three-axes rotation
4096 = 360 degrees for a one-axis rotation

So if we have a three-axes rotation, the final corrections are:

rotationX = rotationX * 360 / 1024
rotationY = rotationY * 360 / 1024
rotationZ = rotationZ * 360 / 1024

In the case of a one-axis rotation, the final correction is, the other two rotations being zero:

rotation = rotation * 360 / 4096

And this completes the parsing of the Animation Section.

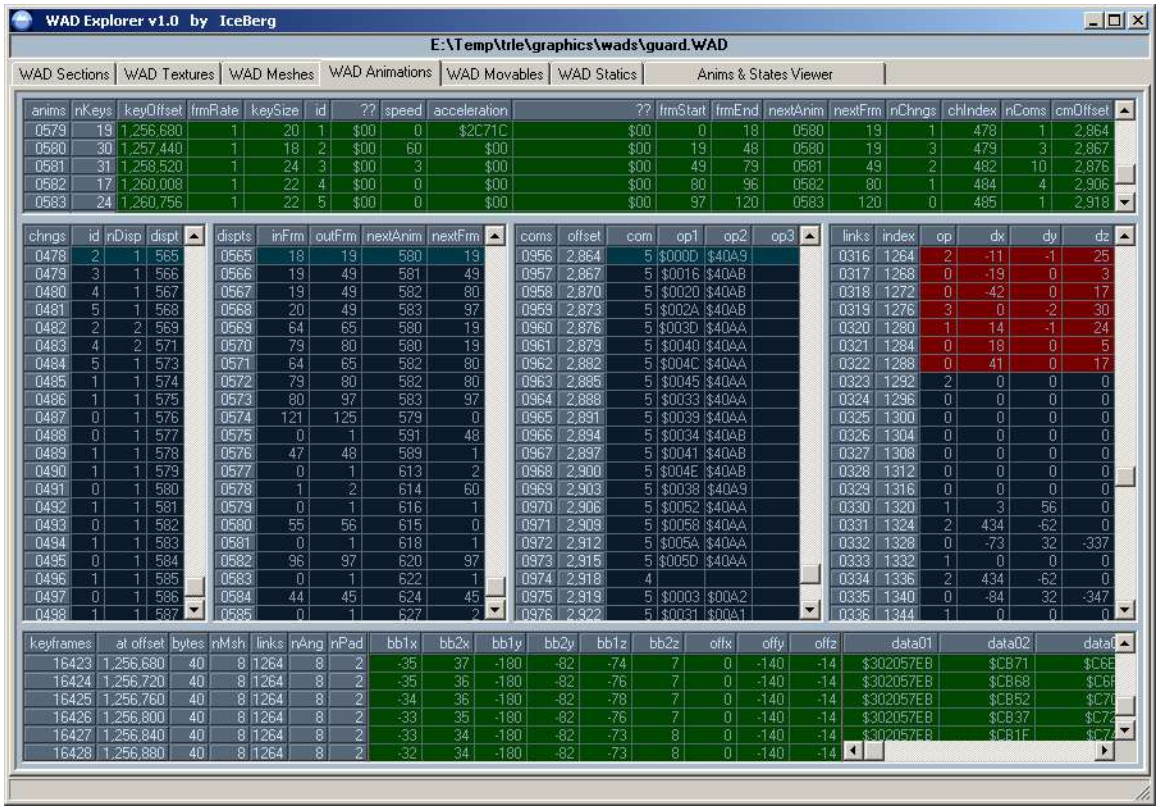


FIG. 47 - Animations table, states changes and dispatches, commands, pivot links. Keyframes package with additional columns introduced by my exploratory application.

Blank page

Section 5 – Models

Num_Movables (**uint32**).
Number of Movable Models stored in the **Movables_Table**.
Most Movable Models are made of a single mesh associated to some animations, or by several meshes linked by articulated and animated joints, organized as a skeleton.

Movables_Table (**Num_Movables * 18 bytes**).
This table lists all the Movable Models stored in the WAD file. The **Movables_Table** holds links to the **Mesh_Pointers_List**, to the **Links_Data_Package**, to the **Keyframes_Data_Package** and to the **Animations_Table**.

This table is the entry point to access the Movable Models in the WAD file. The indexes and the offsets it contains are used to fetch data from other tables and packages. Description:

obj_ID	(**uint32**)	unique ID number for this Movable.
numPointers	(**uint16**)	number of mesh pointers.
pointersIndex	(**uint16**)	index in the pointers list.
linksIndex	(**uint32**)	index of the pivot point links package.
keyframeOffset	(**uint32**)	offset in the keyframes package.
animsIndex	(**sint16**)	index in the animations table.

model	id	numPointers	pointerIndex	linksIndex	keyfrmOffset	animsIndex
0000	0	15	0	0	0	0
0001	1	15	15	56	1,086,194	445
0002	2	15	30	112	1,088,636	449
0003	3	15	45	168	1,091,078	453
0004	4	15	60	224	1,100,704	458
0005	5	15	75	280	1,110,466	463
0006	6	15	90	336	1,119,500	469
0007	7	15	105	392	1,122,774	473
0008	8	15	120	448	1,130,598	478
0009	9	15	135	504	1,131,338	479
0010	10	15	150	560	1,131,466	480
0011	11	15	165	616	1,131,514	481
0012	12	15	180	672	1,131,588	482
0013	13	17	195	728	1,131,658	483
0014	14	17	212	792	1,131,798	484
0015	15	17	229	856	1,131,938	485
0016	16	17	246	920	1,132,078	486
0017	28	15	263	984	1,131,526	-1
0018	29	15	278	1,040	1,132,218	487
0019	30	6	293	1,096	1,134,984	490
0020	31	11	299	1,116	1,135,014	491
0021	33	15	310	1,156	1,146,950	520
0022	87	1	325	1,212	1,168,478	549
0023	90	8	326	1,212	1,168,498	550
0024	95	15	334	1,240	1,173,782	556
0025	96	15	349	1,296	1,197,920	586
0026	101	15	364	1,352	1,197,980	587
0027	103	4	379	1,408	1,202,050	588
0028	107	1	383	1,420	1,202,076	589
0029	115	1	384	1,420	1,202,096	590
0030	116	1	385	1,420	1,202,116	591
0031	120	31	386	1,420	1,202,136	592
0032	121	17	417	1,540	1,202,216	593
0033	122	1	434	1,604	1,202,268	594
0034	127	1	435	1,604	1,205,080	598

FIG. 48 – Movable models table, containing the ID of the movable and indexes or offsets to some other tables.

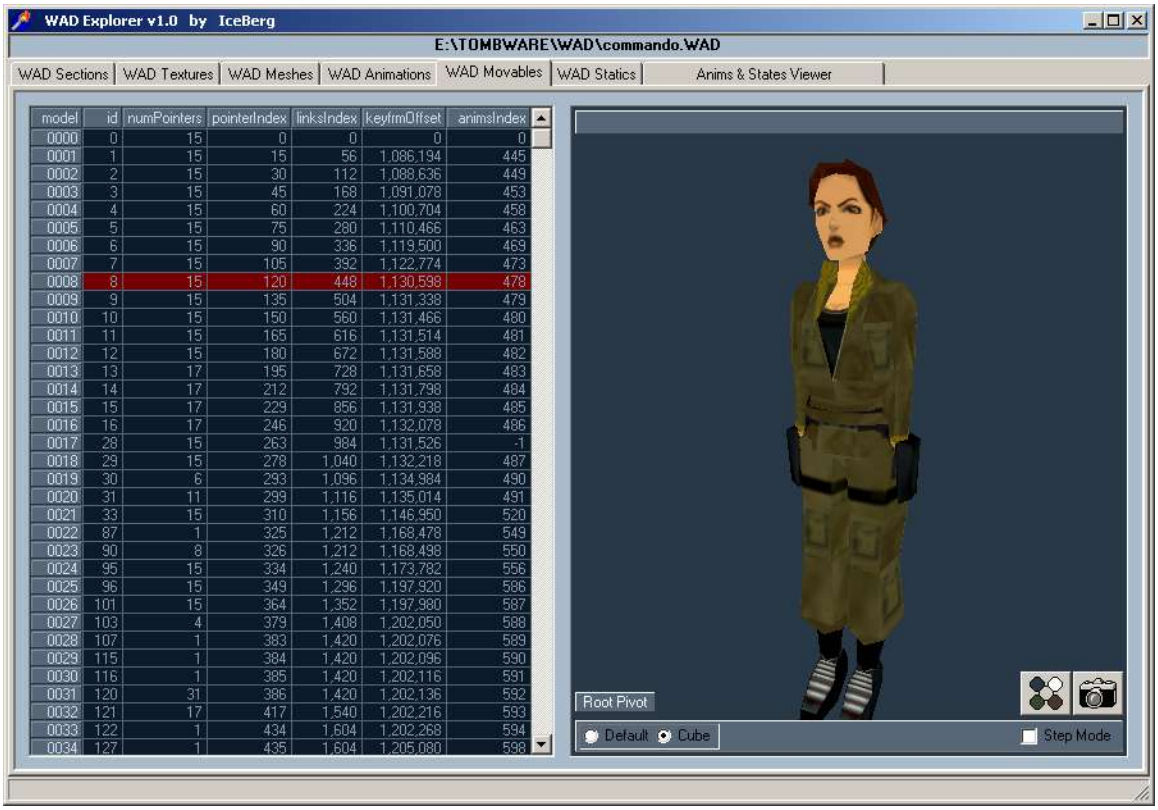
The meshes that compose a movable model are stored in a random access package and are accessed by offset. The offsets of the meshes are stored in a sequencial access list. This list is, in turn, accessed by index, through the **pointersIndex**. The number of meshes is given by the number of pointers, in **numPointers**.

The skeleton associated to a movable model consists of a sequence of hierarchical links. The links are stored in a random access package and accessed by the index of its first integer value, as in **linksIndex**. The number of links is not stored in the movable's table, but can be deduced from the number of meshes (num_links = **numPointers** - 1). Deducing the number of links from the movable's table would be possible if the links package was treated as a sequencial access package. Two consecutive indices, subtracted and divided by four, would yield the correct number. Unfortunately not all applications produce well-behaved link packages. Better use the number of pointers minus one, instead.

The field **keyframeOffset** points to the first keyframe of the current model in the huge keyframes package. Nothing else can be deduced from here, further analysis of the keyframes must be done in the **Animations_Table**.

The entry in the **animsIndex** is usually a positive number, an index to a table, but it may be **-1**, which means that there is no animation for the Movable. The engine will handle the Movable. Examples: Lara's *crowbar* animation and the *bullet cartridges* expelled by the weapons. Other models, like Lara's *ponytail*, have an index to the **Animations_Table** but then the animation is empty and the ponytail is handled by the engine.

The number of animations in the current model is obtained subtracting two consecutive values of the **animsIndex** field, ignoring the **-1** values.



model	id	numPointers	pointerIndex	linksIndex	keyfrmOffset	animsIndex
0000	0	15	0	0	0	0
0001	1	15	15	56	1,086,194	445
0002	2	15	30	112	1,088,636	449
0003	3	15	45	168	1,091,078	453
0004	4	15	60	224	1,100,704	458
0005	5	15	75	280	1,110,466	463
0006	6	15	90	336	1,119,500	469
0007	7	15	105	392	1,122,774	473
0008	8	15	120	448	1,130,598	478
0009	9	15	135	504	1,131,338	479
0010	10	15	150	560	1,131,466	480
0011	11	15	165	616	1,131,514	481
0012	12	15	180	672	1,131,588	482
0013	13	17	195	728	1,131,658	483
0014	14	17	212	792	1,131,798	484
0015	15	17	229	856	1,131,938	485
0016	16	17	246	920	1,132,078	486
0017	28	15	263	984	1,131,526	-1
0018	29	15	278	1,040	1,132,218	487
0019	30	6	293	1,096	1,134,984	490
0020	31	11	299	1,116	1,135,014	491
0021	33	15	310	1,156	1,146,950	520
0022	87	1	325	1,212	1,168,478	549
0023	90	8	326	1,212	1,168,498	550
0024	95	15	334	1,240	1,173,782	556
0025	96	15	349	1,296	1,197,920	586
0026	101	15	364	1,352	1,197,980	587
0027	103	4	379	1,408	1,202,050	588
0028	107	1	383	1,420	1,202,076	589
0029	115	1	384	1,420	1,202,096	590
0030	116	1	385	1,420	1,202,116	591
0031	120	31	386	1,420	1,202,136	592
0032	121	17	417	1,540	1,202,216	593
0033	122	1	434	1,604	1,202,268	594
0034	127	1	435	1,604	1,205,080	598

FIG. 49 – Movable models table, OpenGL viewer displaying my favourite custom outfit.
("Stargate" by Litepulsar, available from Lara's Level Base)

Num_Statics (**uint32**).
Number of Static Models stored in the **Statics_Table**.
The Static Models are made of one single mesh, they have no hierarchies, no Links, they have no animations. Like every other mesh, Statics are just a collection of polygons. They are used for props, ornaments, architectural elements, etc. If for some reason a static model is broken down in several pieces, those pieces need to be handled separately as if they were individual models. This happens a lot with big statues, which are broken down into smaller pieces.

Statics_Table (**Num_Statics * 32 bytes**).
This table lists all the Static Models stored in the WAD file. Each record in this table is 32 bytes long and contains an index to the **Mesh_Pointers_List**, which in turn will have a pointer to the mesh itself in the **Mesh_Data_Package**. The record also stores two bounding boxes, one for visibility and the other for collision purposes. Finally, the record stores some flags. Description:

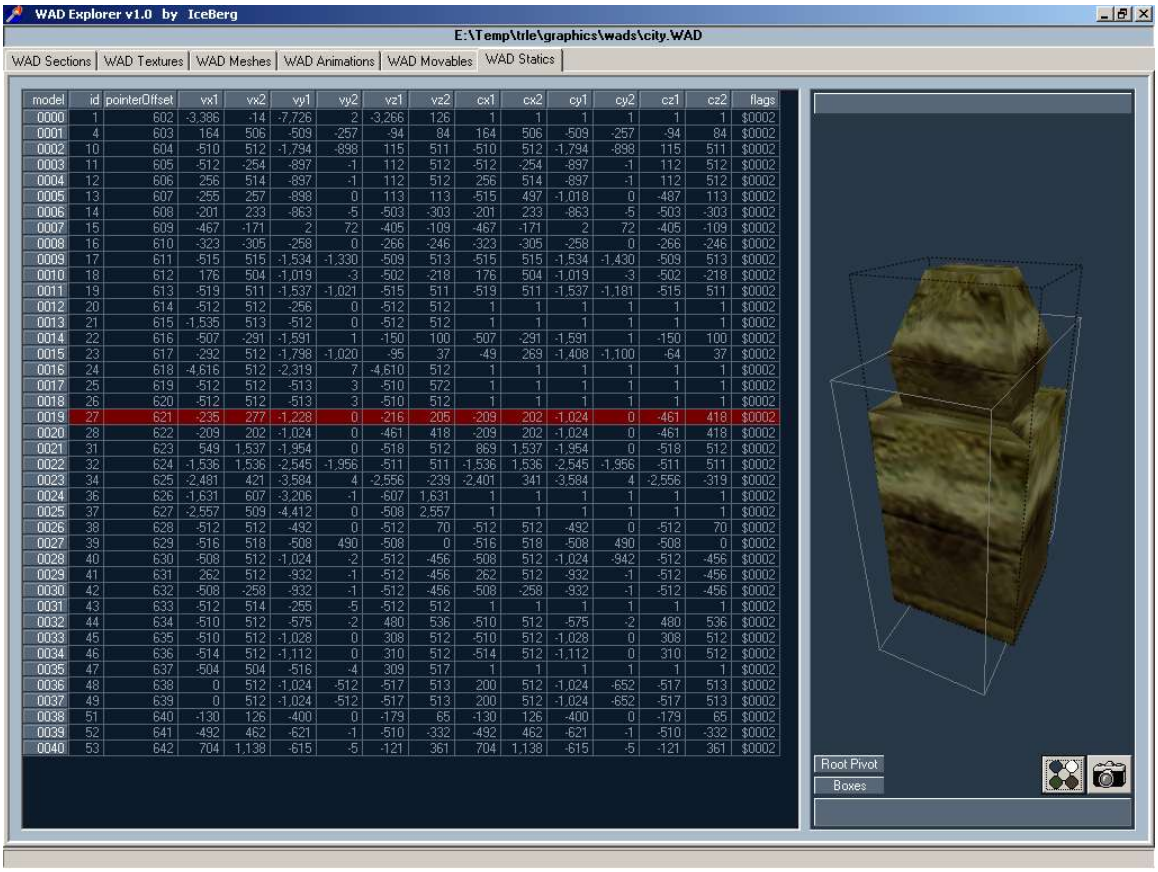
obj_ID	(**uint32**)	unique ID number for this Static.
pointersIndex	(**uint16**)	index of a pointer to the mesh.
vx1	(**sint16**)	coordinate, visibility bounding box.
vx2	(**sint16**)	coordinate, visibility bounding box.
vy1	(**sint16**)	coordinate, visibility bounding box.
vy2	(**sint16**)	coordinate, visibility bounding box.
vz1	(**sint16**)	coordinate, visibility bounding box.
vz2	(**sint16**)	coordinate, visibility bounding box.
cx1	(**sint16**)	coordinate, collision bounding box.
cx2	(**sint16**)	coordinate, collision bounding box.
cy1	(**sint16**)	coordinate, collision bounding box.
cy2	(**sint16**)	coordinate, collision bounding box.
cz1	(**sint16**)	coordinate, collision bounding box.
cz2	(**sint16**)	coordinate, collision bounding box.
flags	(**uint16**)	some flags.

The **visibility bounding box** is probably used for rendering tests in-game, to check if the model is to be rendered or not, the **collision bounding box** is used for collision detection in-game.

The **flags** field always takes the value **2** in the original WAD files for TRLE. Non-official WAD files extracted directly from TR1,2,3 levels[10] show other values : **0**, **3**, but not very often. The meaning of the **flags** field is unknown.

FIG. 50 – From **SETTOMB**, an example of a static model made of two different pieces.

[10] *Extracted by Wee Bald Man, mid-2002 using TR2WAD. These WAD files can be found on-line at http://www.earthacademy.net/TR-Classic/*



model	id	pointerOffset	vx1	vx2	vy1	vy2	vz1	vz2	cx1	cx2	cy1	cy2	cz1	cz2	flags
0000	1	602	-3,386	-14	-7,726	2	-3,266	126	1	1	1	1	1	1	$0002
0001	4	603	164	506	-509	-257	-94	84	164	506	-509	-257	-94	84	$0002
0002	10	604	-510	512	-1,794	-898	115	511	-510	512	-1,794	-898	115	511	$0002
0003	11	605	-512	-254	-897	-1	112	512	-512	-254	-897	-1	112	512	$0002
0004	12	606	256	514	-897	-1	112	512	256	514	-897	-1	112	512	$0002
0005	13	607	-255	257	-898	0	113	113	-515	497	-1,018	0	-487	113	$0002
0006	14	608	-201	233	-863	-5	-503	-303	-201	233	-863	-5	-503	-303	$0002
0007	15	609	-467	-171	2	72	-405	-109	-467	-171	2	72	-405	-109	$0002
0008	16	610	-323	-305	-258	0	-266	-246	-323	-305	-258	0	-266	-246	$0002
0009	17	611	-515	515	-1,534	-1,330	-509	513	-515	515	-1,534	-1,430	-509	513	$0002
0010	18	612	176	504	-1,019	-3	-502	-218	176	504	-1,019	-3	-502	-218	$0002
0011	19	613	-519	511	-1,537	-1,021	-515	511	-519	511	-1,537	-1,181	-515	511	$0002
0012	20	614	-512	512	-256	0	-512	512	1	1	1	1	1	1	$0002
0013	21	615	-1,535	513	-512	0	-512	512	1	1	1	1	1	1	$0002
0014	22	616	-507	-291	-1,591	1	-150	100	-507	-291	-1,591	1	-150	100	$0002
0015	23	617	-292	512	-1,798	-1,020	-95	37	-49	269	-1,408	-1,100	-64	37	$0002
0016	24	618	-4,616	512	-2,319	7	-4,610	512	1	1	1	1	1	1	$0002
0017	25	619	-512	512	-513	3	-510	572	1	1	1	1	1	1	$0002
0018	26	620	-512	512	-513	3	-510	512	1	1	1	1	1	1	$0002
0019	27	621	-235	277	-1,228	0	-216	205	-209	202	-1,024	0	-461	418	$0002
0020	28	622	-209	202	-1,024	0	-461	418	-209	202	-1,024	0	-461	418	$0002
0021	31	623	549	1,537	-1,954	0	-518	512	869	1,537	-1,954	0	-518	512	$0002
0022	32	624	-1,536	1,536	-2,545	-1,956	-511	511	-1,536	1,536	-2,545	-1,956	-511	511	$0002
0023	34	625	-2,481	421	-3,584	4	-2,556	-239	-2,401	341	-3,584	4	-2,556	-319	$0002
0024	36	626	-1,631	607	-3,206	-1	-607	1,631	1	1	1	1	1	1	$0002
0025	37	627	-2,557	509	-4,412	0	-508	2,557	1	1	1	1	1	1	$0002
0026	38	628	-512	512	-492	0	-512	70	-512	512	-492	0	-512	70	$0002
0027	39	629	-516	518	-508	490	-508	0	-516	518	-508	490	-508	0	$0002
0028	40	630	-508	512	-1,024	-2	-512	-456	-508	512	-1,024	-942	-512	-456	$0002
0029	41	631	262	512	-932	-1	-512	-456	262	512	-932	-1	-512	-456	$0002
0030	42	632	-508	-258	-932	-1	-512	-456	-508	-258	-932	-1	-512	-456	$0002
0031	43	633	-512	514	-255	-5	-512	512	1	1	1	1	1	1	$0002
0032	44	634	-510	512	-575	-2	480	536	-510	512	-575	-2	480	536	$0002
0033	45	635	-510	512	-1,028	0	308	512	-510	512	-1,028	0	308	512	$0002
0034	46	636	-514	512	-1,112	0	310	512	-514	512	-1,112	0	310	512	$0002
0035	47	637	-504	504	-516	-4	309	517	1	1	1	1	1	1	$0002
0036	48	638	0	512	-1,024	-512	-517	513	200	512	-1,024	-652	-517	513	$0002
0037	49	639	0	512	-1,024	-512	-517	513	200	512	-1,024	-652	-517	513	$0002
0038	51	640	-130	126	-400	0	-179	65	-130	126	-400	0	-179	65	$0002
0039	52	641	-492	462	-621	-1	-510	-332	-492	462	-621	-1	-510	-332	$0002
0040	53	642	704	1,138	-615	-5	-121	361	704	1,138	-615	-5	-121	361	$0002

FIG. 51 – Static models table, OpenGL viewer displaying a static model from **CITY**. The visibility bounding box is represented with dotted dark line, the collision bounding box is represented with a solid clear line.

DISCUSSION TOPICS

Blank page

Discussion Topic 1 – Variables Nonsenseclature

Programming languages are supposed to be addressed to intelligent and inventive persons.
If so, how is it possible that the nomenclatures used by different languages to define their variable types be so dumb and irrational? How is it possible that from language to language the definitions be so different, even contradictory? How is it possible that such a simple and basic foundation of the programming languages, apparently so easy to define, ends up being just another problem?

There was a time when a **char** would range from **0** to **127** because that was the range for the characters in a font, and vice-versa. As a consequence, the range from **128** to **255** could not be assigned to a **char**. Instead, that range was interpreted as **negative**, as **-128** to **-1**, and **char** was considered as a **signed** type.

There was a time when a **byte** would take care of the **0** to **255** range, always being interpreted as **positive** and considered as an **unsigned** type.

And this was enough for the old 8-bit computer architectures.

There was a time when computers evolved to 16-bit architectures, which called for a new type definition for 16-bit numbers. An **int** was capable of storing **positive** and **negative** numbers in 16 bits, and its range was therefore **-32768** to **32767**. This was a **signed** type.

There was a time when the **unsigned** 16-bit counterpart of an integer was a **word**, whose range would go from **0** to **65535**. A **word** could also be considered as a sequence of two bytes, whose order, in-buffer, would be [b0,b1] for Intel microprocessors or [b1,b0] for the Motorola ones.

Whatever happened to that simplicity?

The arrival of the 32-bit architectures called for a definition of a 32-bit **integer** type. The **int** type was already related to a 16-bit environment, so what name could we give to a 32-bit integer? That's when the big confusions started up. Poor thinking ahead and poor imagination led to the renaming of the old 16-bit **int** to something else and "upgrading" the old **int** to a new 32-bit **int**. Worse, different programming languages did those changes using different nomenclatures. The final result was a complete disaster, unfit of the intelligence and inventiveness that programmers are supposed to have.

The arrival of the 64-bit architectures poses the same problem again. The announcement of the 128-bit architectures poses it yet "another" again. The future etc-bits architectures will inevitably pose this same problem several other "again's". Inevitably? Maybe not. Maybe programmers are not in the mood to carry on supporting the present silliness.

I know I'm not, that's why I'm writing this Discussion Topic. Let's have a look at the problem, as it is at the present time. Basically, in a 64-bit environment, we need to deal with these integers:

Signed **8**-bits	[-128 .. 127]
Signed **16**-bits	[-32,768 .. 32,767]
Signed **32**-bits	[-2,147,483,648 .. 2,147,483,647]
Signed **64**-bits	[-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807]
Unsigned **8**-bits	[0 .. 255]
Unsigned **16**-bits	[0 .. 65,535]
Unsigned **32**-bits	[0 .. 4,294,967,295]
Unsigned **64**-bits	[0 .. 18,446,744,073,709,551,615]

For the records, a shorter way of writing those numbers would be:

Signed **8**-bits	[-2^7 .. 2^7 - 1]
Signed **16**-bits	[-2^15 .. 2^15 - 1]
Signed **32**-bits	[-2^31 .. 2^31 - 1]
Signed **64**-bits	[-2^63 .. 2^63 - 1]
Unsigned **8**-bits	[0 .. 2^8 - 1]
Unsigned **16**-bits	[0 .. 2^16 - 1]
Unsigned **32**-bits	[0 .. 2^32 - 1]
Unsigned **64**-bits	[0 .. 2^64 - 1]

Well, my first reaction to this is that those numbers are growing up into ridiculous sizes.
A signed 32-bit integer holds numbers up to 2 gigabytes. The unsigned equivalent holds numbers up to 4 gigabytes. As for the 64-bit, I don't even know what to call them.

Now lets see the nomenclatures that several programming languages are using now-a-days:

C/C++				**C#**
Signed **8**-bits	signed small int	signed char	small	sbyte
Signed **16**-bits	signed short int		short	short
Signed **32**-bits	signed long int		long	int
Signed **64**-bits	signed hyper int		hyper, LONGLONG	long
Unsigned **8**-bits	unsigned small int	unsigned char	BYTE, UCHAR	byte
Unsigned **16**-bits	unsigned short int	wchar_t	USHORT, WORD	ushort, char
Unsigned **32**-bits	unsigned long int		ULONG, DWORD	uint
Unsigned **64**-bits	unsigned hyper int		DWORDLONG	ulong

One obvious reaction of the C/C++ programmers against this craziness was to use some of the shortcuts or macros described in the 3rd column. The 4th column shows a simpler notation used by C# where the references to signed-unsigned types are included in the type as a suffix. We see, however, incoherencies building up already. In C/C++ a **long** is a **32-bit** integer, whereas in C# a **long** is a **64-bit** integer. In C/C++ **int** can be omitted, in C# it becomes a signed 32-bit integer.

DELPHI			
Signed **8**-bits	Shortint		
Signed **16**-bits	Smallint		
Signed **32**-bits	Longint		Integer
Signed **64**-bits	Int64		
Unsigned **8**-bits	Byte	Char, AnsiChar	
Unsigned **16**-bits	Word	WideChar	DWord
Unsigned **32**-bits	Longword	wchar_t	Cardinal
Unsigned **64**-bits	-		

Delphi goes along with an **integer** being a signed 32-bit type, and uses traditional notations like **byte** or **word** (Delphi is not case-sensitive). At first sight it looks similar to the C/C++ notation for the **signed** integers, but a second look shows a problem. In both the reference to "**long**" designates a 32-bit type, but the references to "**small**" and "**short**" are crossed over.
Confusion keeps on building up.

Another reference language is Java, which brings us yet another source of confusions. It only defines **signed** types for the integers. To define a pseudo-unsigned type, we must use the next bigger signed type and mask the redundant top bits. The only true **unsigned** integer type is **char**, a **16-bits** type. This could easily enter a dictionary as a definition for “brainless”.

JAVA	
Signed **8**-bits	byte
Signed **16**-bits	short
Signed **32**-bits	int
Signed **64**-bits	int64
Unsigned **8**-bits	-
Unsigned **16**-bits	char
Unsigned **32**-bits	-
Unsigned **64**-bits	-

Java also defines an **int** as a signed 32-bits integer. The use of **byte** as a designation for the **signed 8-bits** integer is quite unfortunate, uninspired and misplaced.
Where the other languages consider a **byte** as an **unsigned** type ranging from **0** to **255**, along with tradition, Java “innovates” and redefines a byte to a range from **-128** to **127**. This only adds to the confusion already installed. Software portability? What a joke!

All this makes me wonder if the Programming Language designers are not just mocking us all. What is it with the nomenclature of integer types that makes it so dumb?

However, the basic issue is quite simple.
The notation proposed in the TRosettaStone is an example of a simple solution.

Signed **8**-bits	**bit8**
Signed **16**-bits	**bit16**
Signed **32**-bits	**bit32**
Unsigned **8**-bits	**bitu8**
Unsigned **16**-bits	**bitu16**
Unsigned **32**-bits	**bitu32**

Somewhat similar to this notation, some languages are using **int64** as a description of the **64-bit** integers. In some discussion forums where this issue was brought up, the tendency was to name the 64-bits, 128-bits, etc, within these lines. Something like **int8**, **int16**, **int32**, **int64** could have been adopted long time ago, and avoid the nonsense we see now-a-days. A nomenclature easy to understand, expandable for future computer architectures, no need to redefine older types, very clear in concern to the signed-unsigned issue, could and should have been adopted by all programming languages. Why was it not?

INTEGERS	
Signed **8**-bits	**sint8**
Signed **16**-bits	**sint16**
Signed **32**-bits	**sint32**
Signed **64**-bits	**sint64**
Unsigned **8**-bits	**uint8**
Unsigned **16**-bits	**uint16**
Unsigned **32**-bits	**uint32**
Unsigned **64**-bits	**uint64**

Blank page