Crash Bandicoot 2 and 3 –

Model Entry Format

Described by warenhuis

In this document, a description is given of a model type that can be found in Crash Bandicoot 2: Cortex Strikes Back. There appear to be 2 model types in the game: the first consists out of 5 items, the second of 6 items. There are a lot of similarities between the two.

The content should be sufficient to give you a basic understanding of the generation of the model geometry, the vertex coloring and the UV map. Feel free to contact me if there is anything unclear or incorrect in this document *(warenhuis1@hotmail.com)*.

All the information that is needed to generate a 5-item model, is in two entries:

T1 Entry: frame (Animationframe)

Model Entry: Item0 (Header)

Model Entry: Item1 (Triangle specifications)

Model Entry: Item2 (Colorlist)

Model Entry: Item3 (Textureinfo)

Model Entry: Item4 (Animationinfo)

A 6-item model has the same items and an additional item:

Model Entry: Item 5 (4D Delta's)

Note: a 6-item model has a compressed animationframe

Animation frames

There are corresponding animation frames (T1 entries). These contain the unique 3D positions (not the model's vertices, these are specified by item 1 in the model entry based on the positions in this frame). The animation frames have the following format:

Header: Crash Bandicoot 2

0x00: Offset X (signed little endian 16-bit field)

0x02: Offset Z (signed little endian 16-bit field)

0x04: Offset Y (signed little endian 16-bit field)

0x06: Unknown

0x08: Amount of 3D positions

0x0C: Flag for collision

0x10: Corresponding model ID

0x14: Header size in bytes

If the frame has collision, the header continues as follows:

0x18: Unknown

0x1C: Collision offset X

0x20: Collision offset Y

0x24: Collision offset Z

0x28: Collision(box?) X1

0x2C: Collision(box?) Y1

0x30: Collision(box?) Z1

0x34: Collision(box?) X2

0x38: Collision(box?) Y2

0x3C: Collision(box?) Z2

Header: Crash Bandicoot 3 (differences)

The frameheader is somewhat different from the Crash Bandicoot 2 one and has to be examined further.

For now, the relevant differences are:

0x10: Amount of 3D positions

0x18: Header size in bytes

3D positions

Uncompressed frame

After the header come the 3D positions. From here on the 5-item and 6-item models differ from each other. Frames for the 5-item model specify 3D POSITIONS in 3-byte structures:

XXXX XXXX YYYY YYYY ZZZZ ZZZZ

XXXX XXXX: X (unsigned)

YYYY YYYY: Y (unsigned)

ZZZZ ZZZZ: Z (unsigned)

Compressed frame

Frames for compressed 6-item model specify ADDITIONAL 3D DELTA'S in a varying format:

X (XXX XXXX) Z (ZZZ ZZZZ) Y (YYY YYYY)

X (XXX XXXX) X_temporal + (0 to 7 optional extra bits) (signed)

Z (ZZZ ZZZZ) Z_temporal + (0 to 7 optional extra bits) (signed)

Y (YYY YYYY) Y_temporal + (0 to 7 optional extra bits) (signed)

Item 5 in the model entry specifies how many optional bits each of the XZY-delta gets. There is a delta for every vertex.

After the frameheader, the bytes in every 32-bit word need to be read in reverse (ABCD becomes DCBA). When one places the 32-bit words one after another, there is a bitstream of various sized-XZY delta's that can be read properly (also see illustration 3).

Item 0: Header

This is the first item in a model. The header starts with 3 scale values for the model. To get the dimensions right, one must flip either one of the 3 dimensions (for example: -x, y, z).

After this there are 8 ID's for texturechunks (in this document referred to as textureslot 0-7). Typically, most slots will be left unused.

The slots are followed by the sizes of various items and animation frames. There are also some unknown values as of yet.

0x00: Scale X (signed little endian 32-bit field)

0x04: Scale Z (signed little endian 32-bit field)

0x08: Scale Y (signed little endian 32-bit field)

0x0C: Textureslot 0 ID

0x10: Textureslot 1 ID

0x14: Textureslot 2 ID

0x18: Textureslot 3 ID

0x1C: Textureslot 4 ID

0x20: Textureslot 5 ID

0x24: Textureslot 6 ID

0x28: Textureslot 7 ID

0x2C: Amount of 4-byte structures in item 1 (various structures)

0x30: Unknown

0x34: Amount of 12-byte structures in item 3 (texture info's)

0x38: Amount of 3-byte structures in the animation frames (3d positions)

0x3C: Amount of 4-byte structures in item 2 (colors)

0x40: Unknown

0x44: Amount of triangles in item 1?

0x48: Amount off animation indices in item 4

0x4C: 3D positions offset

Item 1: Triangle specifications

Quick overview

The list in this illustration (1) represents an example of the raw data from item 1, but ordered in 4 bytes per row for clarity's sake. Every 4 bytes represent a “structure” of different types. The grey text structures represent “colorslots”. The black text structures represent “triangle specifications”. The last structure is a “footer”.

The blue column represents the “structure type identifier” for the “triangle specifications”, which is either an “original” or duplicate”.

The red column represents the “triangle types”.

The orange column represent “pointers”, which will make sense later on.

The purple column represents the index for the vertexcolor in the colorlist (item 2, in multiples of 2).

The green column represents the index for the textureinfo In item 3.

Item 1				structure:
04	00	00	00	colorslot
01	02	02	B8	original
01	02	03	B8	original
01	02	57	B8	original
03	00	00	00	colorslot
02	00	04	18	original
03	00	57	38	original
07	02	00	00	colorslot
04	02	02	7C	duplicate
05	02	05	18	original
03	00	00	00	colorslot
06	00	04	5C	duplicate
0B	04	00	00	colorslot
04	04	57	38	original
03	00	00	00	colorslot
07	00	03	1C	duplicate
0F	06	00	00	colorslot
06	06	02	3C	duplicate
08	06	57	78	original
0B	04	00	00	colorslot
01	04	05	7C	duplicate
07	02	00	00	colorslot
09	02	02	1C	duplicate
FF	FF	FF	FF	

Illustration 1: The list

The structures

Item 1 consists of multiple types of 4-byte structures. These structures work together to make a recursive list. This list will specify the vertexcolors and geometry for the model. There are 2 types of triangle specifications and 1 colorslot ype. In this document, the structure types are given the following names:

“Original” (triangle specification)

Specifies 3 vertices, 3 colors and a (un)textured triangle. The structure contains an OWN reference to 1 vertex color and 1 3D position and uses previous structures in the list for the 2nd and 3rd colors and positions.

AAAA AAAA BBBB BBBF CCCC CCCC DDDD EEEE

AAAA AAAA: Index of texture info in item 3, starting with '0x1'. '0x0' means untextured

BBBB BBB: Index of color in item 2, times 2!

F: Animationflag. If true, this also causes the index to refer to a 4-byte structure in item 4 (animationinfo) instead of 12-byte structure in item 3 (texinfo)

CCCC CCCC:

A “pointer” for an original 3d position in the animation frame. This value is refered to by a “Duplicate” structure to reuse the same 3d position.

When this “pointer” has value '0x57', it is not refered to by a “Duplicate” structure.

Note that this is NOT the index in the animation frame! The index is explained later on in the document.

DDDD: Triangle type

EEEE: Structure type identifier. For an “Original” structure, this must be either '0' or '8'

“Duplicate” (triangle specification)

Specifies 3 vertices, 3 colors and a (un)textured triangle

AAAA AAAA BBBB BBBF CCCC CCCC DDDD EEEE

AAAA AAAA: Index of texture info in item 3, starting with '0x1'. '0x0' means untextured

BBBB BBB: Index of color in item 2, times 2!

F: Animationflag. If true, this also causes the index to refer to a 4-byte structure in item 4 (animationinfo) instead of 12-byte structure in item 3 (texinfo)

CCCC CCCC:

A “pointer” for a second-hand 3d position in the animation frame. The “Duplicate” structure finds the first previous “Original” structure with the same “pointer” and uses the same 3d position that was used by that “Original”.

DDDD: Triangle type

EEEE: Structure type identifier. For a “Duplicate” structure, this must be either '4' or 'C'

“Colorslot”

Injects 2 extra color indices in the list, in case different colors are needed for a triangle than can be found in the previous triangle structures.

The “Colorslot” is identified by the 3rd and 4th byte having value 0x00.

AAAA AA00 BBBB BBBA 0000 0000 0000 0000

AAAA AAA: index of color in item 2

BBBB BBB: index of color in item 2

Footer

At the end of item 1. Identified by all 4 bytes being 0xFF.

The triangle types

In the 'DDDD' fields of the “triangle specifications”, there are 3 triangle types, and 3 x 4 subtypes (4 for each type). Each triangle type has unique behavior inside item 1. The subtypes range from 0x0 to 0xB and determine the face's direction (the normal is computed automatically). The following table displays the behavior in regards to specifying triangles:

TriSubtype	Tri Type	Forward face	Vertex A	Vertex B	Vertex C
0x0	“AA”	Double sided	3D position Index	3D position Index-1	3D position Index-2
0x1		Counter clockwise			
0x2					
0x3		Clockwise			
0x4	“BB”	Double sided	3D position Index	3D position Index-1	1st previous “CC”-type 3D position Index / 1st previous “AA”-type 3D position Index-2
0x5		Counter clockwise			
0x6					
0x7		Clockwise			
0x8	“CC”	Double sided	3D position Index	3D position Index+1	3D position Index+2
0x9		Counter clockwise			
0xA					
0xB		Clockwise			

Table 1: Triangletypes and their vertices (note: 'index – 1' means: the index before that etc.)

Table 2 displays the behavior in regards to specifying vertexcolors for the triangle (warning: this table is not fully accurate yet, expect unpredictable results when testing!):

Tri Type	Vertexcolor A	Vertexcolor B	Vertexcolor C
“AA”	Color Index (x 2)	If 1st previous structure is “Colorslot”, 1st Color index (x 4) thereof / Color Index-1 (x 2)	If 1st previous structure is “Colorslot”, 2nd Color index (x 2) thereof / If 2nd previous structure is “Colorslot”, 1st Color index (x 4) thereof / Color Index-2 (x 2)
“BB”	Color Index (x 2)	If 1st previous structure is “Colorslot”, 1st Color index (x 4) thereof / Color Index-1 (x 2)	See table 3
“CC”	Color Index (x 2)	Color Index+1 (x 2) (If valid)	Color Index (x 2)+1 (If valid)

Table 2: Triangletypes and their vertexcolors. Multiples of the index are indicated with '(x N)'

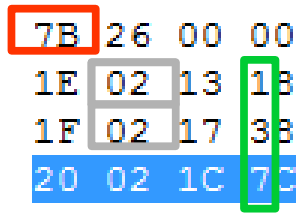
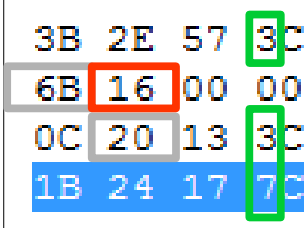
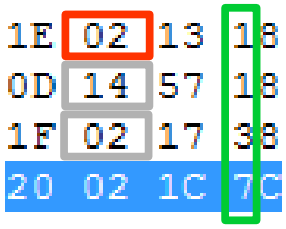
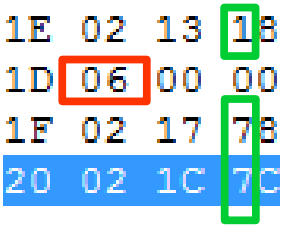
Scenario 1: colorslot is two structures behind 1st previous non-”BB” type structure	Scenario 2: colorslot is one structure behind 1st previous non-”BB” type structure	Scenario 3: intermediate steps get no interference by colorslots	Scenario 4: 1st previous non-”BB” type structure is behind colorslot (no steps, target defaults to 2nd index in colorslot)
			

Table 3: Vertexcolor C scenarios for triangle type “BB” (red marker: target index (either (x 4) or (x 2)), grey markers: intermediate steps, green marker: triangle types)

Validity of type “CC” triangles

For some reason, not all “CC”-type triangles should be drawn. Of course, when a “triangle specification” is at the end of item 1 it can not refer to indices (index +1, +2, etc..) that are beyond the item's reach.

A “CC”-triangle should only be 'valid (i.e. drawn)' if the following conditions are met:

- it is NOT in the last or second-last “triangle-specification” in item 1

- it is NOT preceded by another valid “CC”-triangle within its 2 preceding structures, if it exists

The list

Now that all the referencing mechanics are (sort of) known, it all comes together in the list. Because here the 3D position indices are specified that will be used by the triangles. The triangles will get a vertex that is either an original 3d position or a duplicate 3d position.

The 3D position indices are defined as follows (also see illustration 2):

1. Start with first structure in item 1

2. Start the index-counting with '3D positions offset' (0x4C in the header)

3. Go to next structure in item 1 and read field 'EEEE'

    a) if it is an “Original” (i.e. 0 or 8) add +1 for the current index of this structure. Note that by this logic, the total sum of “Originals” must be equal to the amount of 3D positions in the animationframe minus the '3D positions offset'.

    b) if it is a “Duplicate” (i.e. 4 or C), take its pointer and find 1st previous “Original” with same pointer. Then read the index for that structure and reuse it.

    c) always ignore the “Colorslots”

4. Repeat step 3 until it comes across the item footer (0xFFFF)

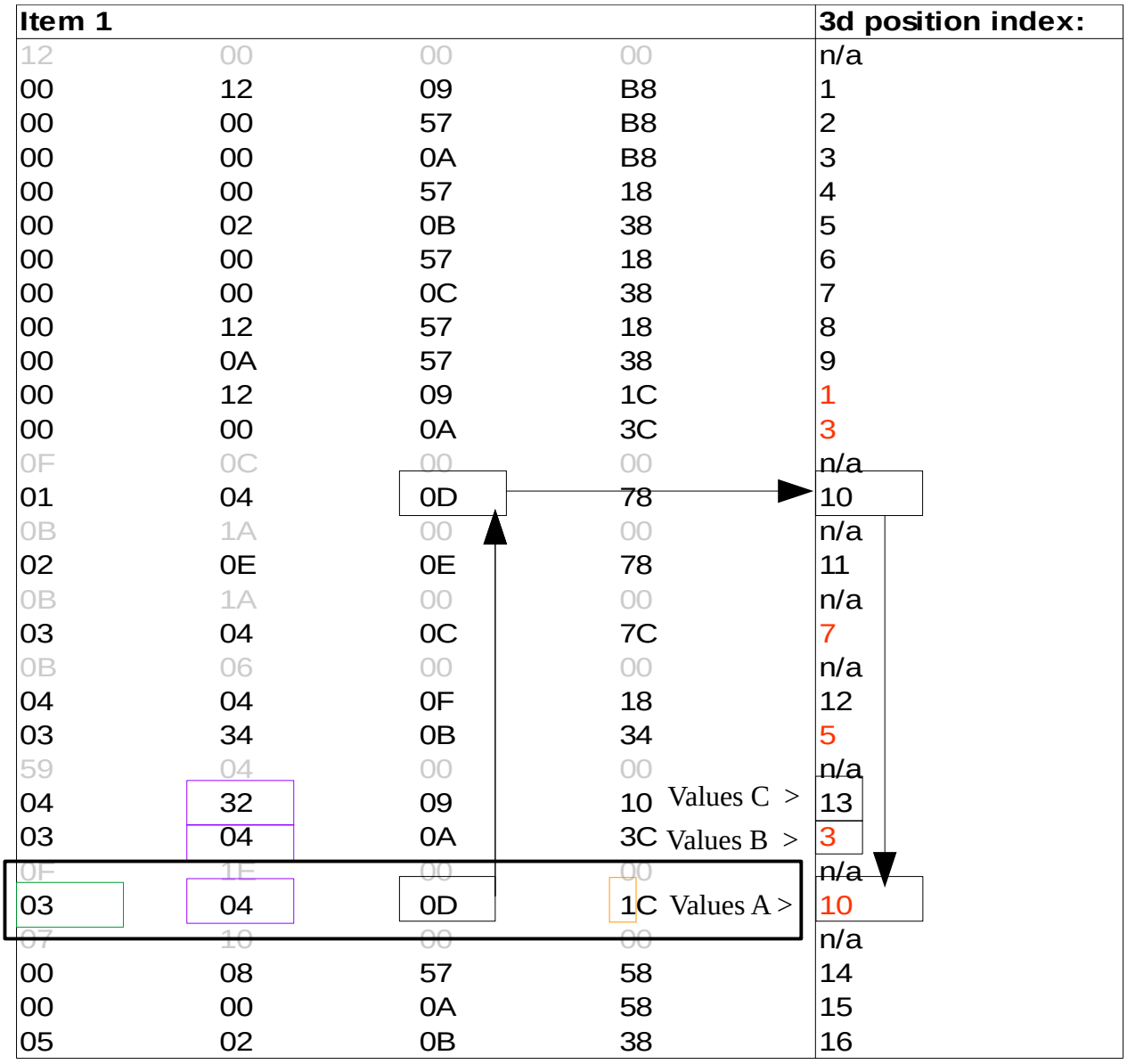


Illustration 2: Example of the specification of 3D position indices (duplicate indices in red, colorindices in purple, texinfo in green, triangle subtype in orange, colorslots grayed out)

The 3d position index list will be used by the triangle types to define the VALUES for vertex A, B and C.

In conclusion, a “triangle specification” specifies the following information:

- textureinformation

- direction of face

- values for vertex A, B, C

- vertexcolor A, B, C.

Item 2: Colorlist

Contains a list with colors of the format:

RRRR RRRR GGGG GGGG BBBB BBBB AAAA AAAA

RRRR RRRR: red

GGGG GGGG: blue

BBBB BBBB: green

AAAA AAAA: unknown

Item 3: Textureinfo

This item is refered to by texture indices in item 1, if the animation flag 'F' is false.

Contains a list of 12 byte structures with the following information:

AAAA AAAA BBBB BBBB CCCC DDDD EEEE EEEE FFFF FFFF GGGG GGGG

NHHH IIII JJJJ JJJJ KKKK KKKK LLLL LLLL MMMM MMMM MMMM MMMM

AAAA AAAA:

for type “AA” and “BB”, mapping point 3 x

for type “CC”, mapping point 1 x

BBBB BBBB:

for triangle type “AA” and “BB”, mapping point 3 y

for triangle type “CC”, mapping point 1 y

CCCC: part of CLUT-line origin y coordinate (CLUT: Color LookUp Table, contain the colors for a specific texture)

DDDD: CLUT-line origin x coordinate

EEEE EEEE: part of CLUT-line origin y coordinate

The origin [x,y] for the CLUT-line is placed in a 16-bit interpretation of the texturechunk. The values can be calculated as follows:

CLUT-line origin x = 16 * DDDD

CLUT-line origin y = 4 * EEEE EEEE + CCCC

FFFF FFFF: mapping point 2 x

GGGG GGGG: mapping point 2 y

NHHH: Colormode. Values 0x0 to 0x7 specify a 4-bit texture. Values 0x8 to 0xF specify an 8-bit texture.

The colormodes in the texinfo are as follows: FHHH
N = 4 / 8 bit flag (1 = 8-bit, 0 = 4-bit)
CCC = colormode (0 to 7)
0 or 1 = translucent face
2 or 3 = additive face
4 or 5 = subtractive face
6 or 7 = standard face

4-bit textures can have 2^4 = 16 different colors. Therefore the corresponding CLUT-line should always be 16 pixels long (in 16-bit!).

8-bit textures can have 2^8 = 256 different colors. Therefore the corresponding CLUT-line should always be 256 pixels long (in 16-bit!).

IIII: Texturechunk segment index (0, 1, 2 or 3)

Each texturechunk in the original Crash games consists out of 4 segments.

The 16-bit size of a texturechunk is 256 x 128

The 8-bit size of a texturechunk is 512 x 128

The 4-bit size of a texturechunk is 1024 x 128

Therefore, the segments in 16-, 8- and 4-bit mode are respectively:

- 64 x 128,

- 128 x 128

- 256 x 128

The segment index refers to the origin of the segment. The mapping points work relative from this origin. Note that the values of the mapping points are not effected by 8- or 4-bit mode. The x,y values range from 0 to 255 and are fixed.

JJJJ JJJJ: refer to an offset from 0x0C in the header, for which textureslot to use (0-7 (i.e. 0x0 – 0x40))

KKKK KKKK:

for triangle type “AA” and “BB”, mapping point 1 x

for triangle type “CC”, mapping point 3 x

LLLL LLLL:

for triangle type “AA” and “BB”, mapping point 1 y

for triangle type “CC”, mapping point 3 y

MMMM MMMM MMMM MMMM: unknown

Item 4: Animationinfo

This item is refered to by texture indices in item 1, if the animation flag 'F' is true.

An animated texture is typically a series of multiple12-byte textureinfo's placed one after another in item 3. These will be played like a 'flipbook'.

Also note that the flipbook runs independent of the model's animation frames. So for example, a model animation with only one animation frame can STILL have animated textures.

Since models with animated texture can have multiple “flipbooks” in item 3, the amount of textureinfo's can go into the THOUSANDS. Item 4 specifies an offset in item 3 for where to find the desired start of the “flipbook”.

Item 4 contains a list of 4 byte structures with the following information:

AAAA BBBB CCCC DDDD EEEE_EEEE EEEE_EEEE

AAAA: Part of the offset for item 3

BBBB: Part of the offset for item 3

CCCC: Length of the “flipbook” in frames (a lower length causes the animation to go slower, so probably all frames must be played within a fixed timespan)

DDDD: Part of the offset for item 3

The offset can be calculated as follows:

Offset in item 3 = DDDD * 256 + AAAA * 16 + BBBB * 1

EEEE_EEEE EEEE_EEEE: Unknown

Item 5: Header (6-item models only)

4D-Delta's

This item contains '4D-Delta's' in 4 byte structures. There is a delta for every vertex in the model. Together with the animation frame, the resulting 3D positions are specified using a 'temporal delta (i.e. the 'time' dimension depending on current frame).

Every delta adds to the previous one, the first delta starting at [0,0,0] in a 255 x 255 x 255 continuum. When the cumulative value of delta's exceed 255 in any dimension, the resulting 3D position will be the remainder of the cumulative value / 255 (for example, 320 would become 65) (or just think of how Pacman comes out on the other end when leaving the level).

You can get a sense of how the temporal values are stored and referenced from in the frame in illustration 3.

Stored as:

AABB BCCC YYYY YYYA ZZZZ ZZZY XXXX XXXZ

To be read as:

XXXX XXXZ ZZZZ ZZZY YYYY YYYA AABB BCCC

'Temporal delta size specification'

AAA: specified amount of EXTRA bits for the x-delta in the animationframe (0 to 7)

BBB: specified amount of EXTRA bits for this y-delta in the animationframe (0 to 7)

CCC: = specified amount of EXTRA bits for this y-delta in the animationframe (0 to 7)

The temporal x, z or y-delta's have 1 to 8 (7+1) bits. With this information the temporal X, Z and Y values can be read in the animationframe.

X-axis

XXXX XXX = X-axis -increment for this delta (0 to 254, multiples of 2)

Y-axis

YYYY YYYY = Y-axis -increment for this delta (0 to 255)

Z-axis

ZZZZ ZZZZ = Z-axis -increment for this delta (0 to 255)

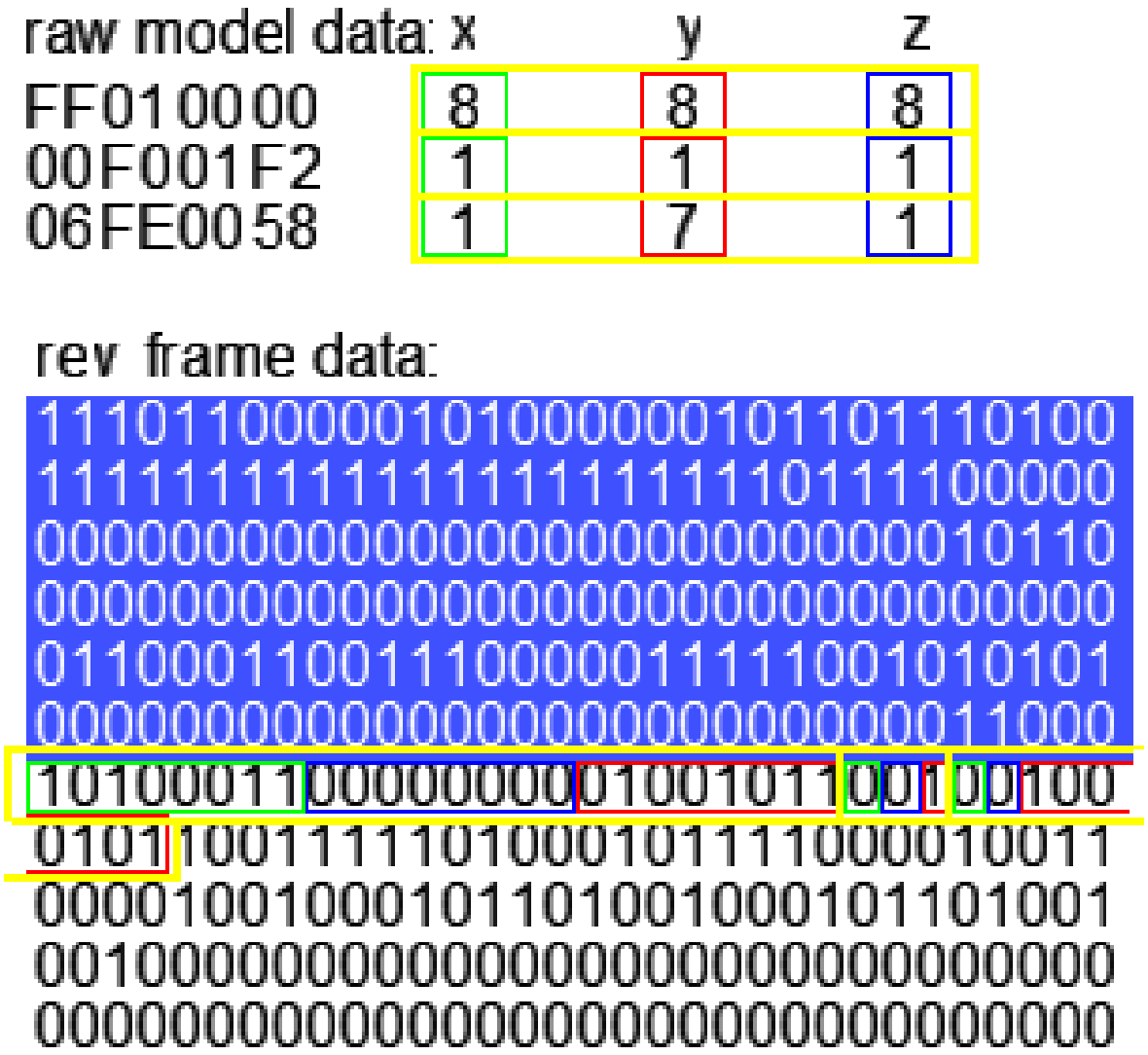


Illustration 3: How to read temporal values in the frame (frameheader is highlighted). Note that the 32-bit words have already been reversed in this example!

Calculate 3D positions

Calculation

Item 5 and a corresponding animation frame is all one needs to calculate the 3D positions.

Let say we have the following values:

X-axis: 56

Z-axis: 72

Y-axis: 234

X-temporal: -70

Z-temporal: -22

Y-temporal: 60

note: the temporal values are signed, so these can also be negative!

The resulting values of the 3D position would be:

X-axis + X-temporal = 56 + -70 = -26

Y-axis + Y-temporal = 72 + -22 = 50

Z-axis + Z-temporal = 234 + 60 = 294

The position [-26 , 50, 294] must be between the bounds of 255 x 255 x 255.

So the values must be edited so that they become remainders of divisions by 255

(i.e. [-26, 50, 294] mod 256 → [230, 50, 38]).

The same values can also be used to add the next delta to, as long as the values stay within a 255 x 255 x 255 continuum.

The '8-bit' reset

When AAA, BBB or CCC = '111', one of the temporal values has 8 bits. In this case the calculation goes different because the corresponding axis value will be ignored and RESET.

For example, when AAA= '111', the X-axis is reset.

With the same values the 3D position would not be:

X-axis + X-temporal = **56** + -70 = **-26** [**-26**, 50, 294] mod 256 → [**230**, 50, 38]

but instead:

X-axis + X-temporal = **0** + -70 = **-70** [**-70**, 50, 294] mod 256 → [**186**, 50, 38]