All values are in Little Endian. Later pages are more readible, sorry for the mess.

Model:

Model Signature

Total Vertex Count (u32)

| Offset(h) | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000000 | 00 | 10 | 00 | 00 | 80 | 9A | 01 | 00 | E4 | 09 | 00 | 00 | 00 | 00 | 00 | 00 |

Sometimes Total Vertex Count is 0, align to 0x10 for future models and continue

Material:

Material Signature: 0xFFFFFFFE, always aligned with 0x0C column

Material Index: u32 / 2

| Offset(h) | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00002880 | 00 | 80 | 04 | 6C | 03 | 00 | 00 | 80 | 00 | 00 | 00 | 00 | FE | FF | FF | FF |
| 00002890 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 20 | EA | 9E | BE | E4 | EE | 3B | BE |

Triangle Strip:

| Offset(h) | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00002960 | 00 | 80 | 04 | 60 | 04 | 00 | 00 | 00 | 41 | 00 | 00 | 00 | 00 | 40 | 1E | 30 |
| 00002970 | 00 | C0 | 1E | 30 | 05 | 01 | 00 | 01 | 00 | 00 | 00 | 20 | 40 | 40 | 40 | 40 |
| 00002980 | 04 | 80 | 04 | 78 | 69 | C3 | 58 | C0 | 1C | 5D | 7F | 3F | F2 | 75 | A4 | BE |
| 00002990 | AF | 8B | 57 | C0 | 93 | 9A | 80 | 3F | C1 | 5C | 94 | BE | E7 | 59 | 57 | C0 |
| 000029A0 | 30 | 9A | 77 | 3F | E8 | CB | 9E | BE | 84 | 93 | 3D | C0 | 2B | 72 | 76 | 3F |
| 000029B0 | 0F | C3 | 95 | BE | 05 | 80 | 04 | 7E | 84 | 0A | 18 | 00 | C7 | 04 | 71 | 00 |
| 000029C0 | C6 | 95 | 23 | 00 | CE | CC | 67 | 00 | 06 | C0 | 04 | 6E | 80 | 80 | 80 | 80 |
| 000029D0 | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 07 | 80 | 04 | 64 |
| 000029E0 | EE | 1C | D9 | BE | 80 | 19 | B8 | 3C | 90 | 6B | D9 | BE | 60 | B6 | E8 | 3C |
| 000029F0 | EE | CF | D7 | BE | 20 | 51 | F0 | 3C | 08 | 97 | D7 | BE | 24 | 78 | 14 | 3E |
| 00002A00 | 04 | 04 | 00 | 01 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00002A10 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 04 | 00 | 00 | 14 |

0 - Triangle Strip Signature?
Triangle Strip Vertex Count: u8
Triangle Strip is aligned to 0x10
Original program checks for 0x40404040 end,
I check last eight bytes.
Some other variables are likely to change.
Unknown, Sometimes 0x3E instead of 0x1E
maybe sometimes something else.

1 - Vertex Position Header
Vertex Position Count: u8
Vertex Position:
x: f32, y: f32, z: f32
Vertex Position is done
Vertex Position Count times.

2 - Vertex Normal Header
Vertex Normal Count: u8
Vertex Normal:
x: u8, y: u8, z: u8
Vertex Normal is done
Vertex Normal Count times
with 1 byte of padding between each.
Vertex Normal should be normalised by
dividing Vertex Normal by 255.

3 - Vertex Unknown Header
Vertex Unknown Count: u8
Vertex Unknown: MOSTLY 0x80808080
Vertex Unknown is done
Vertex Unknown Count times.

4 - Vertex Texture Coordinates Header
Vertex Texture Coordinates Count: u8
Vertex Texture Coordinate:
u: f32, v: f32
Vertex Texture Coordinates is done
Vertex Texture Coordinate Count times.

5 - Footer Begin
Padding 0x00 until 0x10 aligned
Footer End

Read the Model Signature and Total Vertex Count, and then repeat finding and reading of Material OR Triangle Strip until Vertex Count has been reached.
There might be an order to the Material or Triangle Strip, I do not know it. After Vertex Count has been reached, search for the next Model Signature.
See: `read_from_bin()` in `model.rs` or `OutputModels()` in Nights Image Tool.

Texture Formats:

| Offset(h) | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00019C10 | 06 | 00 | 00 | 10 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019C20 | 04 | 00 | 00 | 00 | 00 | 00 | 00 | 10 | 0E | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019C30 | 00 | 00 | 00 | 00 | 00 | 00 | 02 | 00 | 50 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019C40 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 51 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019C50 | 80 | 00 | 00 | 00 | 80 | 00 | 00 | 00 | 52 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019C60 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 53 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019C70 | 00 | 10 | 00 | 00 | 00 | 00 | 00 | 08 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019C80 | 00 | 10 | 00 | 30 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019C90 | 06 | 00 | 00 | 10 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019CA0 | 04 | 00 | 00 | 00 | 00 | 00 | 00 | 10 | 0E | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019CB0 | 00 | 00 | 00 | 00 | 00 | 01 | 01 | 02 | 50 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019CC0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 51 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019CD0 | 10 | 00 | 00 | 00 | 10 | 00 | 00 | 00 | 52 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019CE0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 53 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019CF0 | 20 | 80 | 00 | 00 | 00 | 00 | 00 | 08 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019D00 | 20 | 00 | 00 | 30 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00019D10 | 00 | 00 | 00 | 60 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

Find first Texture Signature in file: 06 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00
Double Size Flag: u8 == 0
Size:
 x: u32, y: u32
if Double Size Flag then
 multiply Size by 2
No texture is should be larger than 1024 x 1024.
(I think, I've seen 512 x 512 and 1024 x 256)

Color Depth: u8 == 0 ? 32-bit : 16-bit

Pixel Encoding: u8 == 2 ? 4-bit : 8-bit
if Pixel Encoding == 4-bit and Double Size Flag then
 multiply Size.y by 2? I am not sure why.

Read Texture Footer.
If Texture Footer == 00 00 00 60 00 00 00 00 00 00 00 00 00 00 00 00 then
 you have finished reading Texture Formats.
Else if Texture Footer == Texture Signature then
 repeat with the Footer as the next Signature.

Calculate the Texture locations afterwords.
I do not know how these calculations work.
See: https://github.com/mg35/NightsImageTool/blob/74c481069f16b4a609284d8f0412732bdb453900/main.cpp#L936
or find tex_location in texture_format.rs of my own project.

16-bit Color Palette:

| Offset(h) | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00029E00 | 18 | EB | 18 | DF | 18 | B7 | 18 | 8B | DF | D8 | BD | D4 | 78 | CC | 12 | D4 |
| 00029E10 | 16 | E7 | 39 | EB | 39 | 9F | F7 | 8A | D9 | 96 | 9A | D0 | 17 | EB | 37 | EB |
| 00029E20 | 11 | D4 | F5 | E2 | 9D | D8 | 35 | D4 | 39 | 8B | DE | D4 | 9B | D4 | 32 | D0 |
| 00029E30 | 38 | A2 | 98 | B5 | 58 | B9 | D8 | C4 | 36 | EF | 98 | 96 | F8 | A9 | 98 | C4 |

and so on...

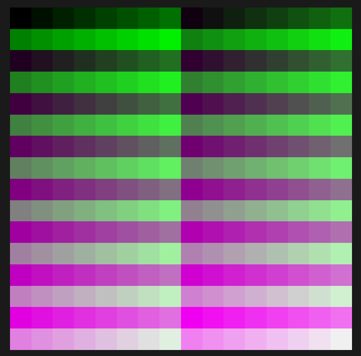
32-bit Color Palette:

| Offset(h) | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00040200 | 00 | 00 | 00 | 7F* | 0C | 00 | 13 | 7F | 14 | 00 | 21 | 7F | 1C | 00 | 2E | 7F |
| 00040210 | 25 | 00 | 3D | 7F | 2B | 00 | 4D | 7F | 39 | 00 | 6B | 7F | 47 | 00 | 83 | 7F |
| 00040220 | 39 | 00 | 62 | 7F | 30 | 00 | 5A | 7F | 29 | 00 | 46 | 7F | 6B | 00 | C2 | 7F |
| 00040230 | 6F | 00 | CD | 7F | 72 | 00 | D0 | 7F | 68 | 00 | D0 | 7F | 62 | 00 | C2 | 7F |

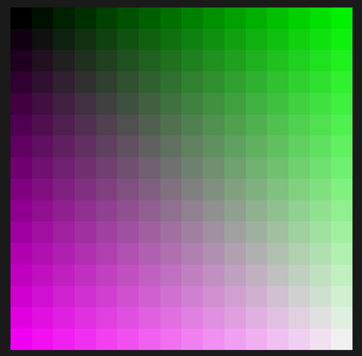
and so on...

8-bit Pixel Encoding's Color Palette Swizzling:

Encoded



Decoded



Color Palettes:

Each Texture has a Color Palette associated with it.
The Color Palette for a texture can be found after the pixel data:
Texture's Location + Width * Height * Pixels per Byte (2 pixels for 4-bit, 1 pixel for 8-bit).
The Color Palette can store up to 256 different Colors in it for 8-bit Pixel Encoding.
The Color Palette can store up to 16 different Colors in it for 4-bit Pixel Encoding.

Storing and handling the Color is subjective. Both Warden and I convert values to 8-bits per channel, but use different calculations for slightly different colors.
My 16-bit colors will be slightly brighter than Warden's, and I store an Alpha channel for the 32-bit colors. In general (keep in mind endian weirdness):

Color (16-bit):
0bABBBBBGGGGGRRRRR

Color (32-bit):
0xAA*BBGGRR

Alpha channel: [0x00, 0x7F]

Texture's that have an 8-bit Pixel Encoding will need to have their Color Palettes swizzled.
Fun.
Assuming a 16 x 16 grid of Colors, every even row (start from 0) should have the right 8 columns swapped with the next row's left 8 columns.

Palette Texture

The palette texture is an array of indexes to colors in the Color Palette.
Once the Texture Format has been read, the Palette Texture can be found at the Texture Location in the BIN file.
If this texture is 8-bit encoded, then there is one index per byte, so just read one byte at a time and extract the index.
If this texture is 4-bit encoded, then there is one index per nibble, read one byte at a time and extract two indexes.
After this you can perform the decoding.

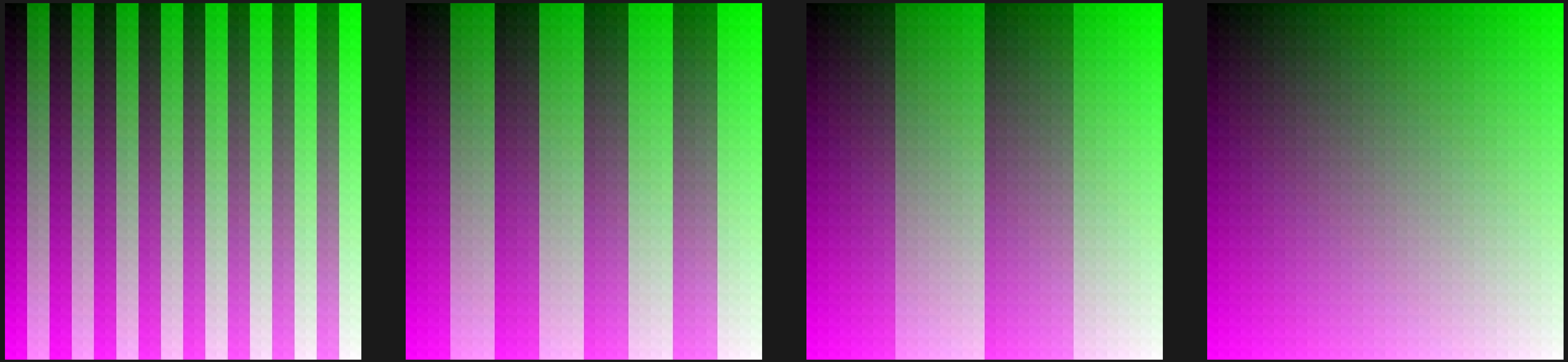
## Texture Decoding - 8-Bit

Textures were scrambled up like a jigsaw puzzle to make advantage of the PS2 hardware. They are a pain to unscramble. I am not too sure how Warden's C++ descrambles 8-bit images, I could not be bothered to learn how visualize it. I did find my own way to unscramble them though.
Warden's solution: https://github.com/mg35/NightsImageTool/blob/74c481069f16b4a609284d8f0412732bdb453900/main.cpp#LL1372C8-L1372C8
My solution (`decode()` in `convert_8bit.rs`) is probably not efficient, so as a fun puzzle to the reader, find your own way to unscramble these images.

Textures with a width of 8 should already be unscrambled.
(note that these images use more than 256 colors, and are NOT flipped vertically like the textures usually are when exported in my program)


1: For every 32 x 4 Chunk:
Split into two 16 x 4 Chunks by Column Parity. That is: one Chunk from all the Even Columns, one Chunk from all Odd Columns.
Join the chunks Vertically, with the Even Chunk on top of the Odd Chunk.
2: Then for the 16 x 8 Chunk:
Split into two 8 x 8 Chunks by Column Parity.
Join the chunks Horizontally, with the Even Chunk left of the Odd Chunk.
3: Then for the 16 x 8 Chunk:
Split into two 16 x 4 Chunks by Row Parity.
Join the chunks Horizontally, with the Even Chunk left of the Odd Chunk.
4: If the Width is NOT 32 then the middle quarters of columns needs to be swapped. I think if I were smarter this step would not be needed*.
Width:
== 16: Swap the middle quarter of columns. That is: swap columns 4, 5, 6, 7 with 8, 9, 10, 11.
== 64: Swap the middle quarter of columns.
\> 64: Swap the middle quarter of columns for each 64 width chunk, then swap middle quarter of columns for each128 width chunk.
Repeat this pattern until you reach the width of the texture. A 256 x 256 example is on the next page.
5: For every OTHER chunk of 4 rows, beginning on the 2nd row (start at 0):
For each 8x4 chunk, swap the left 4 columns with the right 4 columns.
* I found the 1 2 3 pattern on a 32 x 32 image, which is why it might not be adjusted. There might be an order that does not need step 4. If so I'd love to know it.

## Texture Encoding - 8-Bit: Step 4 Example with 256 x 256 Texture



## Texture Encoding - 8-Bit

Do the same but backwards. Steps 1 2 and 3 will need you to perform the inverse, ie split vertically/horizontally and join by interweaving.
See `encode` in `convert_8bit.rs` if you want to see this explicitly writen out.

## Texture from Palette Texture and Color Palette

The texture can be made by accessing the Color Palette at the value of each index in the Palette Texture.

## Texture Decoding - 4-bit

I don't know. I couldn't be bothered figuring out what was going on, and I can not be bothered visualising it just now. It takes time and I do not want to yet.
See https://github.com/mg35/NightsImageTool/blob/74c481069f16b4a609284d8f0412732bdb453900/main.cpp#LL1291C9-L1291C9
or `convert_4bit.rs`.
The steps are individual, so it shouldn't be too hard to visualize the existing solution by returning early then writing an image, easier than the 8-bit one for sure.
At the bottom right are some tiny images, the left is encoded although I can't visualise it like the 8-bit images, since 4-bit encoding has not been implemented yet. Do some magic to make the left into the right image.
4-bit images with a width of 32 call the same function for each 32x32 chunk vertically.
4-bit images not of size 32 x ?, or 64 x 64 can not be decoded yet.