Phantasy Star Online ver.2

By Kion

Table of Contents

PRS

PRS is a commpression algorithm based off LZ77 compression. The Nodejs source code to decompress a buffer is given below.

```
function prs_decompress(buffer) {

	var self = {
		ibuf: buffer,
		obuf: [],
		iofs: 0,
		bit: 0,
		cmd: 0,
		getByte: function() {
			var val = self.ibuf[self.iofs];
			self.iofs += 1;
			return parseInt(val);
		},
		getBit: function() {
			if (self.bit == 0) {
				self.cmd = self.getByte()
				self.bit = 8
			}
			var bit = self.cmd & 1;
			self.cmd >>= 1;
			self.bit -= 1;
```

```
                        return parseInt(bit);
                }
        }

        var t, a, b, j, cmd;
        var offset, amount, start;

        while (self.iofs < self.ibuf.length) {
                cmd = self.getBit();
                if (cmd) {
                        self.obuf.push(self.ibuf[self.iofs]);
                        self.iofs += 1;
                } else {
                        t = self.getBit();
                        if (t) {
                                a = self.getByte();
                                b = self.getByte();

                                offset = ((b << 8) | a) >> 3;
                                amount = a & 7;
                                if (self.iofs < self.ibuf.length) {
                                        if (amount == 0)
                                                amount = self.getByte() + 1;
                                        else
                                                amount += 2;
                                }
                                start = self.obuf.length - 0x2000 + offset;
                        } else {
                                amount = 0;
                                for (j = 0; j < 2; j++) {
                                        amount <<= 1;
                                        amount |= self.getBit();
                                }
                                offset = self.getByte();
                                amount += 2;

                                start = self.obuf.length - 0x100 + offset;
                        }
                        for (j = 0; j < amount; j++) {
                                if (start < 0)
                                        self.obuf.push(0);
                                else if (start < self.obuf.length)
                                        self.obuf.push(self.obuf[start]);
                                else
                                        self.obuf.push(0);

                                start += 1;
                        }
                }
        } //end while

        return new Buffer(self.obuf);

}
```

PRSD

PRSD is an encryption scheme which uses properties of unsigned integers to mask values in such a way that can be decrypted later. The decrypted buffer is then decompressed with prs.

```
function prc_decompress(buf){

        var cxt = {
                "stream" : new Uint32Array(56),
                "key" : null,
                "pos" : null
        };

        var unc_len = buf[0] | (buf[1] << 8) | (buf[2] << 16) | (buf[3] << 24);
        var key = buf[4] | (buf[5] << 8) | (buf[6] << 16) | (buf[7] << 24);
```

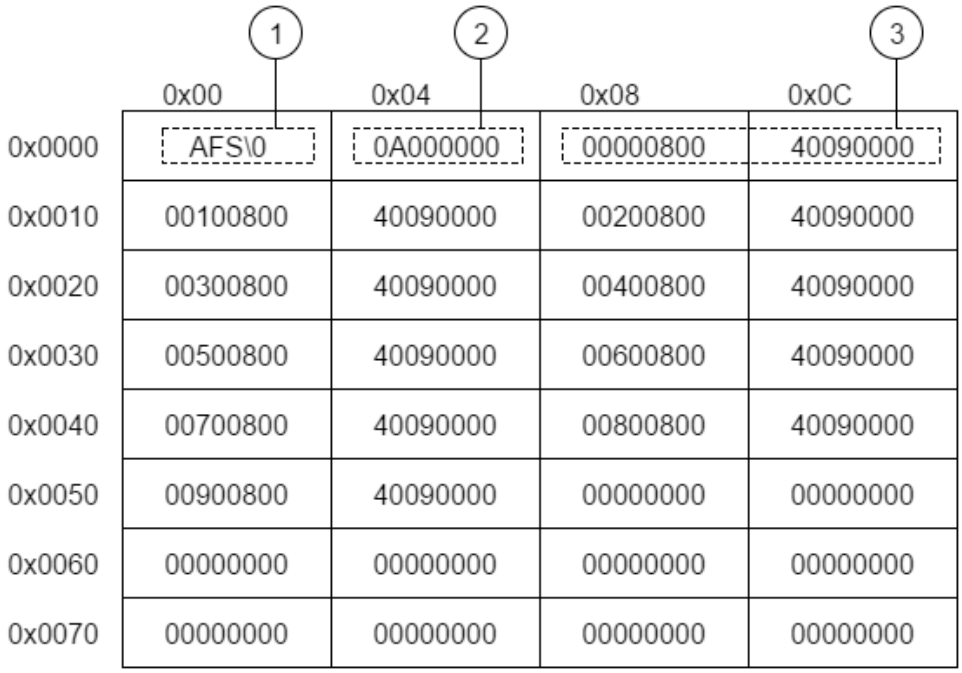
```
        cxt.stream[55] = key;
        cxt.key = key;

        var idx;
        var tmp = 1;

        for(var i = 0x15; i <= 0x46E; i += 0x15) {
                idx = i % 55;
                key -= tmp;
                cxt.stream[idx] = tmp;
                tmp = key;
                key = cxt.stream[idx];
        }

        mix_stream(cxt);
        mix_stream(cxt);
        mix_stream(cxt);
        mix_stream(cxt);

        cxt.pos = 56;

        var tmp, pos, dword;
        var cmp_buf = buf.slice(8)
        var len = (cmp_buf.length + 3) & 0xFFFFFFFC;

        pos = 0;
        while(len > 0){
                dword = cmp_buf.readUInt32LE(pos);
                tmp = crypt_dword(cxt, dword);
                cmp_buf.writeUInt32LE(tmp, pos);

                pos += 4;
                len -= 4;
        }

        return prs_decompress(cmp_buf);
}

function mix_stream(cxt){
        var ptr;

        ptr = 1;
        for(let i = 24; i; --i, ++ptr){
                cxt.stream[ptr] -= cxt.stream[ptr + 31];
        }

        ptr = 25;
        for(let i = 31; i; --i, ++ptr){
                cxt.stream[ptr] -= cxt.stream[ptr - 24];
        }
}

function crypt_dword(cxt, data){
        if(cxt.pos === 56){
                mix_stream(cxt);
                cxt.pos = 1;
        }

        var res = new Uint32Array(1);
        res[0] = data ^ cxt.stream[cxt.pos++];

        return res[0];
}
```

AFS Archive

AFS is a simple file archive which generally stores files that are all of the same type. This is because AFS does not store filenames, so it would likely be impractical to utilize these archives differently. This document will describe archives in two segments, the header, which defines the layout of the files contained in the archive, and the body which contains the files themselves. The diagram below displays the header of an AFS

archive file.

AFS Header



	0x00	0x04	0x08	0x0C
0x0000	AFS\0	0A000000	00000800	40090000
0x0010	00100800	40090000	00200800	40090000
0x0020	00300800	40090000	00400800	40090000
0x0030	00500800	40090000	00600800	40090000
0x0040	00700800	40090000	00800800	40090000
0x0050	00900800	40090000	00000000	00000000
0x0060	00000000	00000000	00000000	00000000
0x0070	00000000	00000000	00000000	00000000

The header is very simple. The first dword in the file, listed as ①, contains the header 'AFS\0'. The second dword ,listed as ②, gives the number of files contained in the archive as an unsigned int. Following that is an array of structs that describes the offset in the file, and the length of the bytes that file contains. A single one of these structs is labeled as ③ in the figure above. The struct trype is as defined below.

```
typedef struct {
        unsigned int offset;
        unsigned int length;
} AFS_ARCHIVE_ENTRY;
```

AFS Body

The body of an AFS archive starts at offset 0x800000, and all of the space inbetween the header and body are empty dwords. The exact size and offset for each file contained within the archive is contained within the header. The offset for each file entry is set so it matches the next offset divisible by 0x0100. Files inside the archive are PRS compressed, so they will need to be decompressed in order to extract the contents.

As filenames are not contained inside the header it is often neccessary to create a naming scheme for files extracted from these archives. The basename of the extracted files should be the name of the archive the file are extracted from. And as there are commonly more than one hundred entries in an AFS archive, and often not as many as a thousand, the archive basename should be followed by a three digit number with leading zeros indicating the file number in the archive. File extensions are also lost, so those must be determined by looking at the interchange file format header of the extracted files.

AFS Source Code

This page contains sample source code for extracting an AFS archive. PRS decompression is used in this function, so please refer to the PRS section of this guide

for that source.

```
"use strict";

var path = require("path");
var filepointer = require("filepointer");

function extract_afs(filename) {
	//Create file pointer to data
	var fp = new FilePointer(filename);


	//Check file format for 'AFS' format
	var iff = fp.readIff();
	if(iff != 'AFS'){
		return null;
	}

	//Get basename of AFS filename
	filename = filename.toLowerCase();
	var basename = path.basename(filename, ".afs");

	//Read number of entries
	var nbEntries = fp.readUInt();

	//For each entry, read struct to array
	var entries = [];
	for(let i = 0; i < nbEntries; i++){
		entries.push({
			"offset" : fp.readUInt(),
			"length" ; fp.readUInt()
		});
	}

	//Seek to and decompress each entry
	for(let i = 0; i < nbEntries; i++){
		var entry = entries[i];

		//Seek to offset
		fp.seek_set(entry.offset);

		//Read copy file data to new buffer
		var data = fp.copy(entry.length);

		//Decompress buffer
		data = prs_decompress(data);


		//Determin extension type
		var ext = ".bin";
		var iff = data.toString("ascii", 0, 4);
		switch(iff){
			case "BIX":
				ext = ".bix";
			break;
			case "NJCM":
			case "NJTL":
				ext = ".nj";
			break;
			case "GBIX":
			case "PVRT":
				ext = ".pvr";
			break;
			case "PVMX":
			case "PVMH":
				ext = ".pvm";
			break;
		}

		entries[i] = {
			"filename" : basename + "_" + i + ext,
			"buffer" : data
		};
	}
```

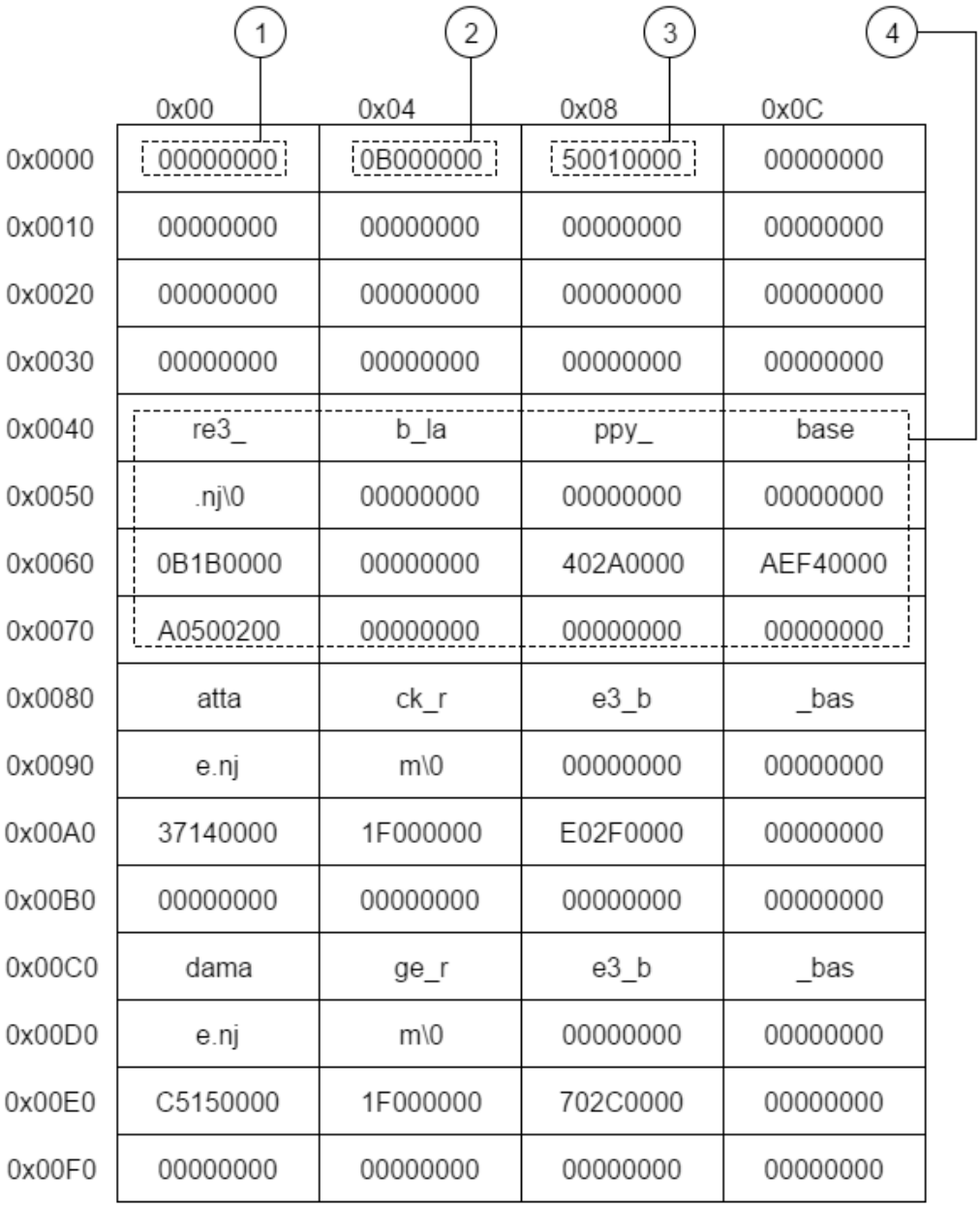
```
        return entries;
}
```

BML Archive

BML is an archive which groups of similar models with their textures and animations. For instance, the model for the rappy and pal rappy will be in the same BML archives, with their own respective textures and grouped with animations which are applicable to both models.

The fileformat itself consists of a header which includes the filename, but not the offset of the files contained within. Also, the header contains a PVM entry, where PVM texture archives are defined as a sub-entry to the model they are defined with, and not a independent entry on their own.

BML Header



The figure above depicts the header of a BML archive. The first dword, labeled as ①, is always zero. The next dword, labeled as ②, gives the number of files contained within

the archive. The third dword, labeled as ③, is a constant 0x0150, which can be used in place of the standard interchange file format.

The actual entry definitions, with the first labeled as ④, start at offset 0x40, and each have a length of 0x40 bytes. Each one contains a filename, length, decompressed length, filetype identifier, pvm archive length and pvm archive decompressed length. The last three dwords on the end of the archive are null values used to adjust the length of each entry, such that it starts at multiples of 0x40.

BML Archive Structure

The structure definition for each BML archive header entry is as given below.

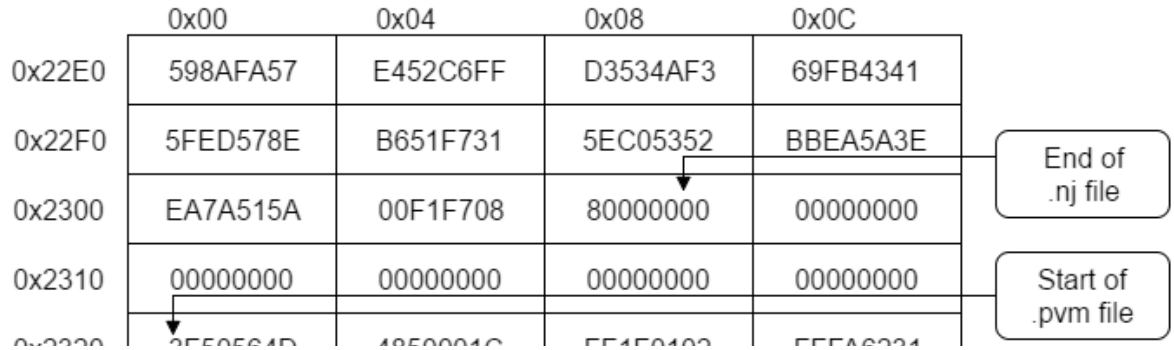
```
typedef struct {
        char filename[0x20];
        unsigned int compressed_length;
        unsigned int filetype_identifier;
        unsigned int decompressed_length;
        unsigned int pvm_compressed_length;
        unsigned int pvm_decompressed_length;
} BML_ARCHIVE_ENTRY;
```

BML Body

The body of BML files will most often start at 0x800. However in some cases, there are so many files declared in the header causing the header to go past this offset. In these cases The offset will be next whole offset of 0x1000 plus 0x0800. The source code for this is included below.

```
var currentOfs = fp.tell();
var bodyOffset = 0x800;
if(currentOfs >= 0x800){
        currentOfs += 0x800;
        bodyOffset = 0xFFFFF800 & currentOfs;
}
fp.seek_set(bodyOffset);
```

Another aspect of BML's is that there is no constant length between two entries. Often with archives where the offset is not defined in the header, the next file in the archive will start immediately after the last, or at some predefined offset. With BML, while each file in the archive starts at a position divisible by 0x10, there is no indication on how far from the last that will take place. Often there will be a row of unsused space between the two, often the next will start immediately at the next number divisble by 0x10, in some rare occasions there will be a massive ammount of unsised space between them. The best approach seems to be to find the first non-zero byte after the previous entry for the start position of the next file. A figure of this is as depicted below.



0x2320	3F50504D	4850001C	FF1F0102	FFFA0231
0x2330	365F7265	6CFD00F8	FF130101	1100A803
0x2340	841EF82C	DA3235C8	FE181CDA	6600A99F
0x2350	DA4D444C	4EE0A8D2	443A5CFF	536F6674
0x2360	696D6167	E165F64F	4654FF33	445F332E

Note that while the end of the previous file in the archive ends at offset 0x230B, rather than continuing the next file immediately from the next multiple of 0x10, there is an entire row left open before the start of the next file. Also each file in BML is compressed with PRS, so the PRS decompress method is required to extract these files.

BML Source Code

Example source code for extracting a BML archive is provided below.

```
"use strict";

var FilePointer = require("filepointer");

function extract_bml(filename){
        //Create filepointer to data
        var fp = new FilePointer(filename);

        //Read number of entries and seek to start of header
        fp.seek_set(0x04);
        var nbEntries = fp.readUInt();
        fp.seek_set(0x40);

        //Read all of the entries into an array
        var entries = [];
        for(let i = 0; i < nbEntries; i++){
                entries.push({
                        "filename" : fp.readString(0x20),
                        "compressed_length" : fp.readUInt(),
                        "filetype_identifier" : fp.readUInt(),
                        "decompressed_length" : fp.readUInt(),
                        "pvm_compressed_length" : fp.readUInt(),
                        "pvm_decompressed_length" : fp.readUInt()
                });
                fp.seek_cur(0x0C);
        }

        //Seek to start of body
        var currentOfs = fp.tell();
        var bodyOffset = 0x800;
        if(currentOfs >= 0x800){
                currentOfs += 0x800;
                bodyOffset = 0xFFFFF800 & currentOfs;
        }
        fp.seek_set(bodyOffset);

        //Read through and parse all files in the archive
        var files = [];
        for(let i = 0; i < nbEntries; i++){
                var entry = entries[i];
                var data = fp.copy(entry.compressed_length);
                data = prs_decompress(data);

                files.push({
                        "filename" : entry.filename,
                        "buffer" : data;
                });

                //Seek to next non-zero byte
```

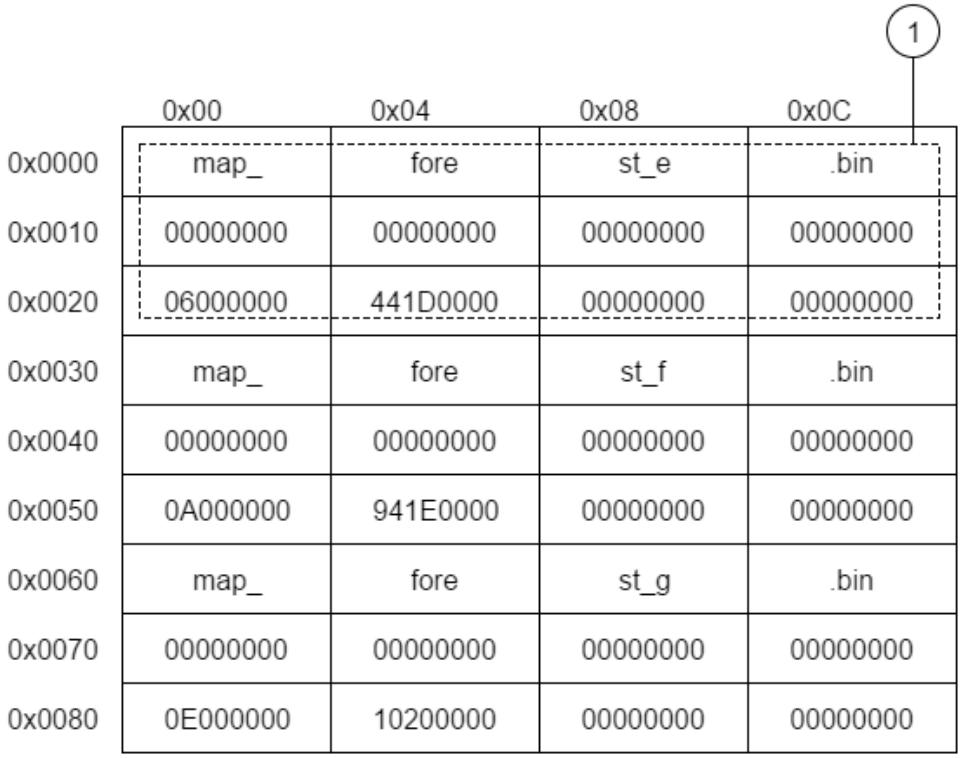
```
                fp.seek_next();

                //Goto next pvm entry if not defined
                if(!entry.pvm_compressed_length){
                        continue;
                }

                //Copy pvm content
                var data = fp.copy(entry.pvm_compressed_length);
                data =  prs_decompress(data);

                //Add PVM file to files array
                files.push({
                        "filename" : entry.filename.split(".")[0] + ".pvm",
                        "buffer" : data
                });
        }

        //Return list of files
        return files;
}
```

GSL Archive

GSL archives are archives where files are grouped by Area. For instance, the file gsl_forest01.gsl, contains the enemy models found in the forest one area, such as boomas and rappies. In addition is also contains messages, fences, boxes, and other objects all found within that area, and not content which is not included in that area.

As for the file format, GSL does not employ the use of an interchange file format header. The archive entries begin right at the beginning of the file. As such there is also no number of entries given for the file. So either the offset for the first entry, or a null dword should be used to determine the end of the archive header.

GSL Header



The header of an GSL file is as depicted above. As stated before, there is no interchange file format header, and the number of entries in the header is not declared. Each entry in the header has a length of 0x30, and the first entry is labeled as ①. The structure for

each entry is as given below.

```
typedef struct {
        char filename[20];
        unsigned int offset;
        unsigned int length;
} GSL_ARCHIVE_ENTRY;
```

On the end of each header entry is two unused dwords, which are simply used to align the start of the next entry to the next multiple of 0x30.

One thing that's deceptive about AFS archives, is that the offset is not a direct offset, but that the offset is obtained by multiplying the number in the header entry by 2048.

Lastly, GSL are not an archive in the sense that files are compressed to save space, but rather a grouping of files by scene. As such files in GSL archives are not compressed with PRS like other archives. This means that files contained within the archive take up more space than if no archive were used at all, but that doesn't seem to be the point, as mainly it seems to be a method to load all required models into memory by scene.

GSL Source Code

Source code for extracting files from a GSL archive is as given below.

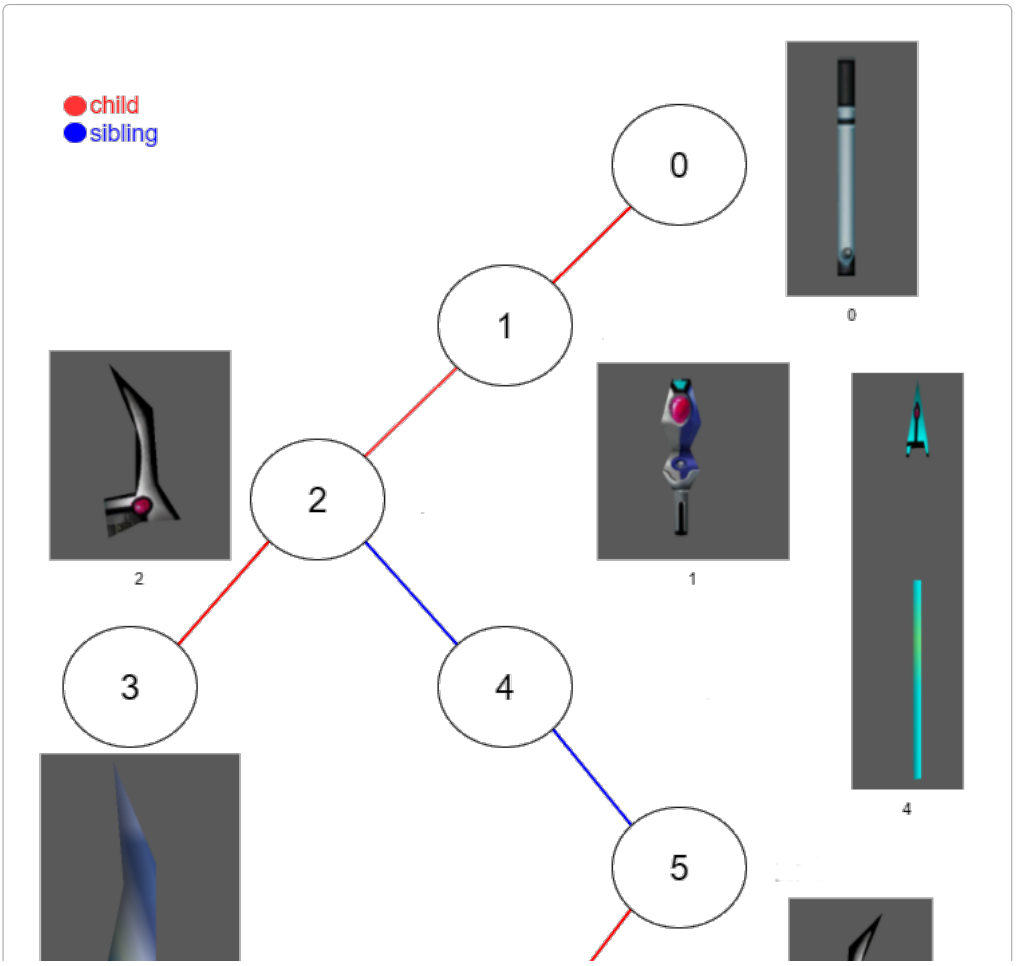
```
"use strict";

var FilePointer = require("filepointer");

function extract_gsl(filename){

        //Set filepointer to data
        var fp = new FilePointer(filename);

        //Offset adjustment constant
        const OFFSET_ADJUST = 2048;

        //Variable for first offset
        var firstOffset = 0;

        //Read each entry in the file header
        var entires = [];

        //Loop as long as the first dword is not zero
        while(!fp.isZero()) {

                var entry = {
                        "filename" : fp.readString(0x20),
                        "offset" : fp.readUInt() * OFFSET_ADJUST,
                        "length" : fp.readUInt()
                };

                //Seek past unused space
                fp.seek_cur(0x08);

                entries.push(entry);

                //Set offset of first file
                if(!firstOffset){
                        firstOffset = entry.offset;
                }

                //Check for distance from first file
                if(fp.tell() > firstOffset - 0x30){
                        break;
                }
        }
```

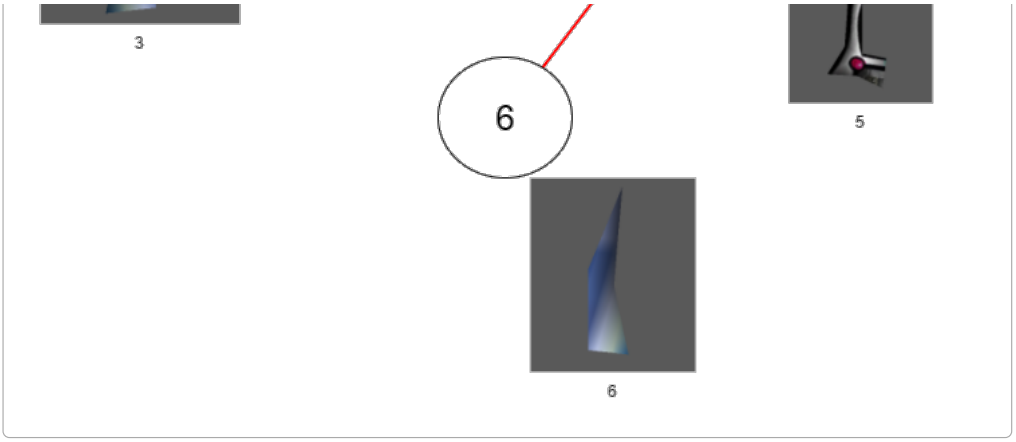
```
        //Read all of the content into an array
        for(let i = 0; i < entries.length; i++){
                var entry = entires[i];

                fp.seek_set(entry.offset);
                var data = fp.copy(entry.length);

                //Overwrite current entry
                entries[i] = {
                        "filename" : entry.filename,
                        "buffer" : data
                };

        }

        //Return archive files
        return entries;
}
```

Ninja Chunk Model

Ninja chunk models, or .nj models are the primary model format used in Phantasy Star Online version 2. Each model consists of a group of nodes. Each node represents a bone that may or may not have mesh data attached to it. These nodes are arranged in a tree structure child-sibling relationship such that the combination of the meshes defined create a single model.

Each mesh is drawn in the following manner. The current matrix state is pushed to the matrix stack. The matrix is then rotated, scaled, at positioned so that the mesh is drawn at the origin. The matrix stack is then popped to return it to its state before the model is drawn, thus placing it in the appropriate state inside the scene.





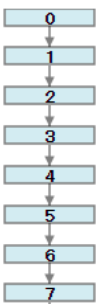
The figure above shows an example of each specific mesh inside the model of the weapon known as the psychowand.

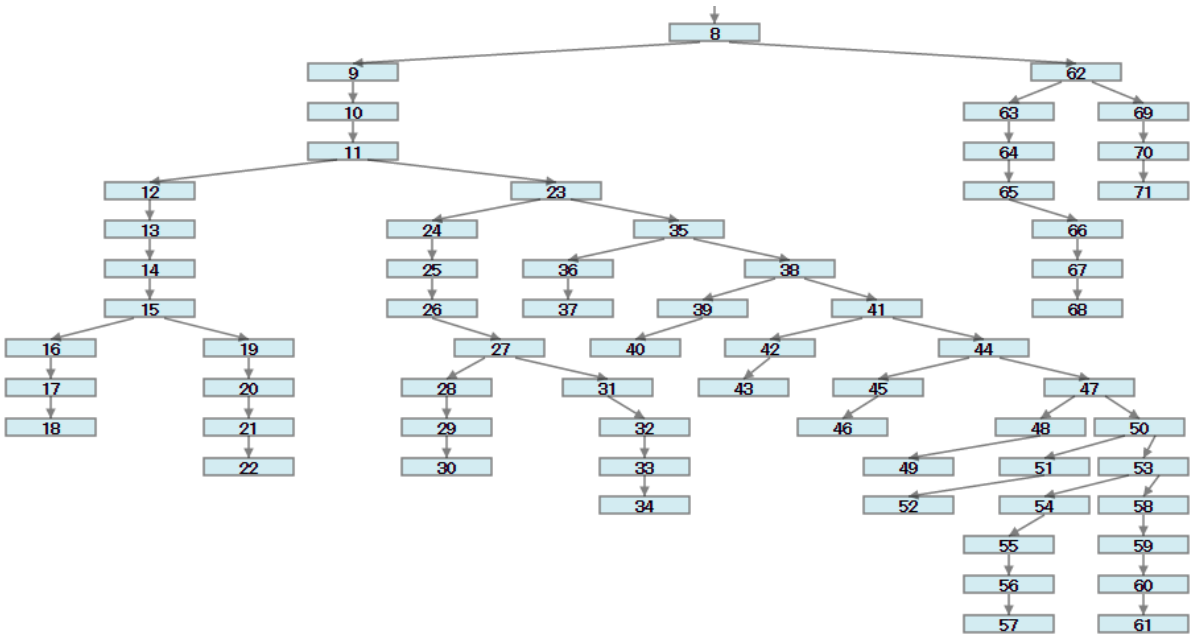
Ninja Chunk Definition

This section defines the necxessary information required to parse information out of an .nj file. We will be using the sample class below and from here on in this documentation for code samples.

```
function NinjaChunkModel(filename){
        this.fp = new FilePointer(filename);

        //List of texture filenames
        this.texureList = [];
        //List of unique materials in the model
        this.materials = [];
        //List of meshes (material Id with faces)
        this.meshList = [];
        //List of bones (as objects)
        this.boneList = [];

        //List of vertices with normals, colors, uv
        this.vertexStack = [];
        //Map vertex indices to stack
        this.vertexMap = [];
        //Map UV coordinates to stack
        this.uvMap = [];

}
```

The way this documentation will parse .nj models is as follows. We will read the Ninja Texture List at the top of the file which is a reference to filenames of all of the textures applied to the model. We will then continue to the Ninja Chunk definition. The Ninja Chunk definition is a tree node child-sibling structure in which bones are defined to create a structure for the model. Meshes can be defined which will be attached to the bone index they are delcared with. The figure below is a depiction of the tree structure implemented in the Ninja Graphics API.





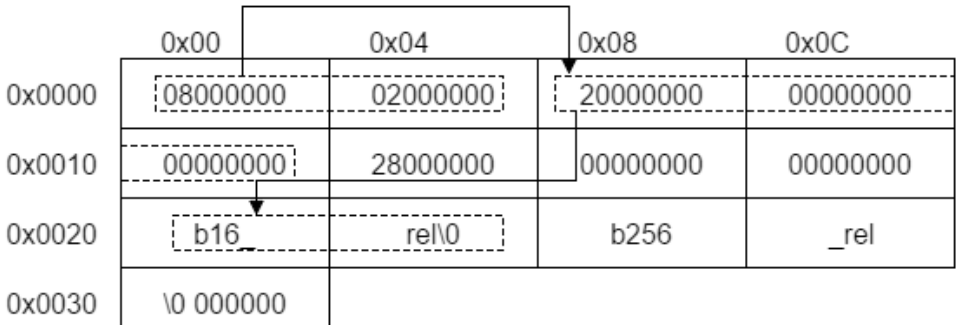
Ninja Texture List

Ninja chunk models will typically have a texture list declared at the top of the model. The texture list is a list of texture names, with empty space for modifiers to be applied to during run time. The texture list is simply a list of texture names, as textures are loaded into memory separately along with a Global Index which is a unique value used to identify each texture as to never be loaded into memory more than once. The texture names are cross referenced with the texture list in memory and are rendered accordingly at run time.

Exceptions for when a texture file are not included is in such cases as the player model. As the player model can be customized, another file in memory dictates which texture should be applied to which model. Other occasions are boss models which maybe split over several .nj files, or the model uses only materials to define its appearance.

Offset(h)	00	04	08	0C	
00000000	4E4A544C	34000000	08000000	02000000	NJTL4.....
00000010	20000000	00000000	00000000	28000000	.....(...
00000020	00000000	00000000	6231365F	72656C00	.....b16_rel.
00000030	62323536	5F72656C	00000000	504F4630	b256_rel....POF0

The texture list starts out with the interchange file format header "NJTL" along with the length of the texture list segment (in this case 0x34). At the end of the segment is the terminator POF0, which indicates that the pointers inside the segment are relative, so with the reference to the pointers, the section is as displayed below.



The first two dwords is a struct, defined as NJS_TEXLIST below, which hass a pointer to the texture list, and a value for the number of textures in that list. Each texture in the list is a three dword long struct, defined as NJS_TEXNAME, has a void pointer to a string, and two dwords for attributes which default to 0. The structure for the first two dwrods is as defined below.

```
typedef struct {
        NJS_TEXNAME *textures;
        unsigned int nbTexture;
} NJS_TEXLIST;
```

The struct for the list of textures is as defined below.

```
typedef struct {
        void *filename;    /* Texture file name */
        unsigned int attr; /* Texture attributes */
        void *texaddr;     /* Texture memory address */
} NJS_TEXNAME;
```

Each string in the texture list ends with a null chartacter, '\0'. As such the length is not declared, only the starting pointer of the string. When parsing these files it is necessary to read one byte at a time and stop at a null character to avoid reading past the end of the file.

Ninja Texture List Source

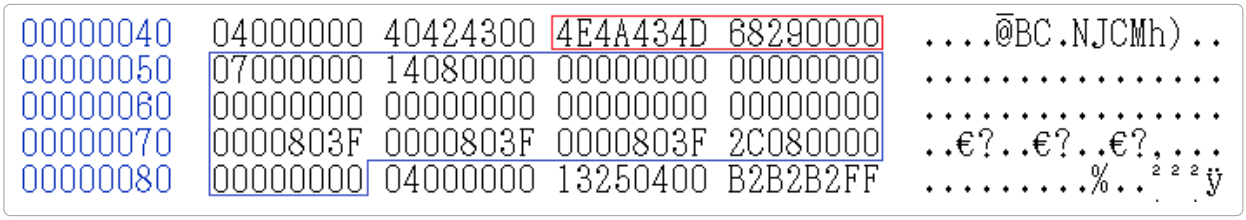
The code below is sample code for parsing the texture list of a Ninja Chunk Model

```
NinjaChunkModel.prototype.readTextureList = function(){

        //Check for NJTL interchange format header
        if(!this.fp.find("NJTL")){
                return;
        }

        var iff = fp.readIff();
        var len = fp.readUInt();

        //Set seek offset to current position
        fp.trim();

        //Read NJS_TEXLIST struct
        var texList = {
                "textures" : fp.readUInt(),
                "nbTexture" : fp.readUInt()
        };

        //Seek to texture offset
        fp.seek_set(texList.textures);

        //Read list of NJS_TEXNAME
        var addrList = [];
        for(let i = 0; i < texList.nbTexture; i++){
                addrList.push({
                        "filename" : fp.readUInt(),
                        "attr" : fp.readUInt(),
                        "texaddr" : fp.readUInt()
                });
        }

        //Read each string into texture list array
        for(let i = 0; i < texList.nbTexture; i++){
                var addr = addrList[i].filename;
                fp.seek_set(addr);
                this.textureList.push(fp.readString());
        }
```

```
}
```

Ninja Chunk Node

As stated in the introduction of this section, Ninja Chunk Models are comprised of nodes, which act as bones which are arranged in a child-sibling tree struct. Each node contains scale, rotation and translation data and pointers to other nodes, one of which can be either a child or a sibling. If the node has either vertex and mesh data, it will have a pointer to a model structure. The Figure below shows the relationship of one of nodes with structs and pointers.



NJCM is the name of the interchange file format shich starts the beginning of the chunk mode data. Similar to the Ninja Texture list, the internal pointers for the segemtn are reset to zero. Following that is a node entry which is of struct type NJS_CNK_OBJECT which is described as a struct below.

```
typedef struct cnkobj {
        unsigned int    evalflags;  /* evalation flags              */
        NJS_CNK_MODEL   *model;     /* model data pointer           */
        float           pos[3];     /* translation                  */
        int             ang[3];     /* rotation                     */
        float           scl[3];     /* scaling                      */
        struct cnkobj   *child;     /* child object                 */
        struct cnkobj   *sibling;   /* sibling object               */
} NJS_CNK_OBJECT;
```

Each node is a set of instructions for creating a bone. The eval flags is an interger value which contains flags for evaluating whether to skip evaluations for scale, rotation and translation for each bone segment. The bit flags as defined in the Katana SDK are as follows.

```
#define NJD_EVAL_UNIT_POS   BIT_0 /* Motion can be ignored */
#define NJD_EVAL_UNIT_ANG   BIT_1 /* Rotation can be ignored */
#define NJD_EVAL_UNIT_SCL   BIT_2 /* Scale can be ignored */
#define NJD_EVAL_HIDE       BIT_3 /* Do not draw model */
#define NJD_EVAL_BREAK      BIT_4 /* Break child trace */
#define NJD_EVAL_ZXY_ANG    BIT_5 /* Specification for evaluation */
#define NJD_EVAL_SKIP       BIT_6 /* Skip motion */
#define NJD_EVAL_SHAPE_SKIP BIT_7 /* Skip shape motion */
```

The description for each bone is described below.

NJD_EVAL_UNIT_POS When this bitflag is true, translation is not applied to the bone

NJD_EVAL_UNIT_ANG When this bitflag is true, rotation is not applied to the bone

NJD_EVAL_UNIT_SCL When this bitflag is true, scale is not applied to the bone

NJD_EVAL_HIDE Mesh data attached to this bone is not to be drwan (such as switching weapons)

NJD_EVAL_BREAK When this flags is true, the child trace operation is terminated

NJD_EVAL_ZXY_ANG When this flag is true, rotation data is evaluated in ZXY order

NJD_EVAL_SKIP During motion execution, matrix processing is carried out using the bone structure value

NJD_EVAL_SHAPE_SKIP Indicates that this node does not include shape motion data.

Ninja Chunk Model

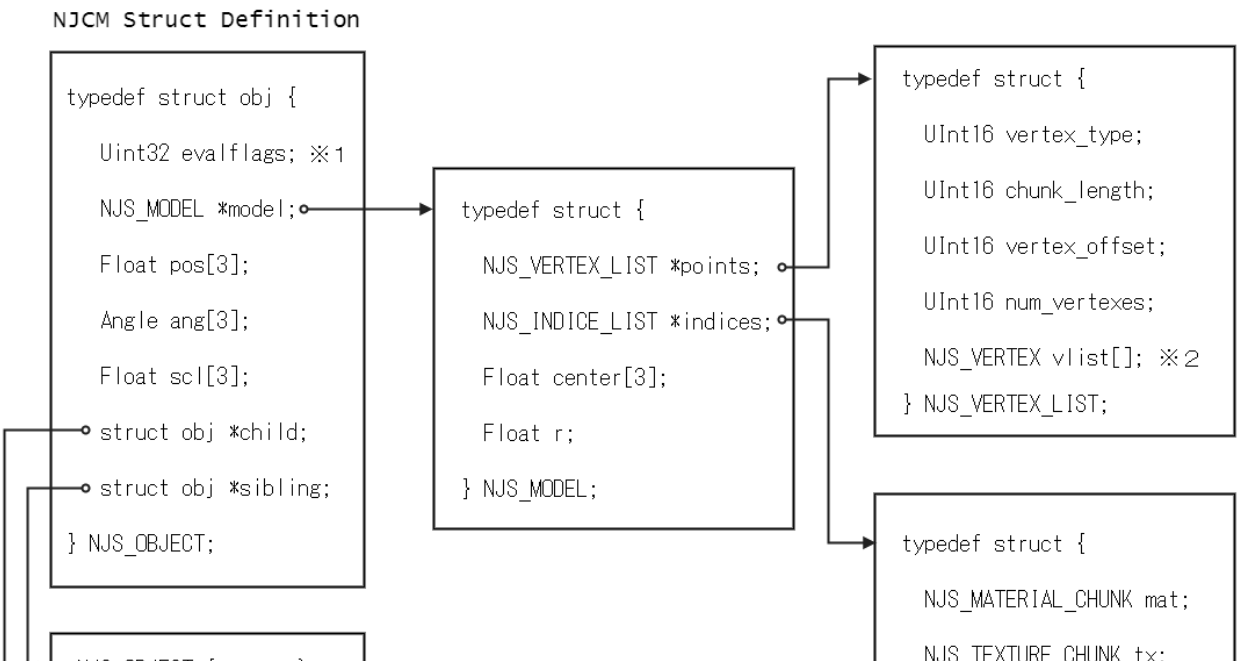
In a ninja chunk model, not all bones will have mesh data attached to them. In such cases, the model pointer from the node will be zero. When mesh data is being defined, the node will include a pointer to the model struct. The model struct acts a pointer which serves three functions. It will point to a vertex list, which gets pushed to the vertex stack, it will have a polygon pointer to define faces which are made up of indexes in the vertex stack, and it declares a center and radius for the mesh data.

```
00000810  FF000000 00000000 34000000 00008034  ÿ.......4.....€4
00000820  D6291F41 FEFFAFBF 00000000 0E000000  Ö).Aþÿ¯¿........
```

The figure above shows the model struct when viewed in a hex editor. The definition for the struct is as given below.

```
typedef struct {
    void                *vlist;    /* vertex list                */
    void                *plist;    /* polygon list               */
    float               center[3]; /* model center               */
    float               r;         /* radius                     */
} NJS_CNK_MODEL;
```

The vertex list is a void pointer which points to a chunk section. This is similiar with the polygon section. The reason for these pointers being void pointers and not structures is because these two sections are not defined structures. The vertex list can have a variable amount of vertices defined, and even multiple vertex lists defined with the same pointer. Similarly with the polygon list, the polygon list is actually made up of multiple chunk segments which must be parsed one after the other to create meshes from the current vertex list.



The relation between nodes, model, polygon list and vertex list is as displayed in the figure above.

Ninja Chunk Node Source Code

Provided below is sample source code for how to read the NJS_CNK_OBJECT, and create a matrix in a recursize loop structure.

```
NinjaChunkModel.prototype.readNode = function(parentBone){

	//Create a new bone object
	var bone = new NinjaChunkBone();

	//Read NJS_CNK_OBJECT struct
	var node = {
		"evalflags" : this.fp.readUInt(),
		"model" : this.fp.readUInt(),
		"pos" : this.fp.readVec3(),
		"rot" : this.fp.readRot3(),
		"scl" : this.fp.readVec3(),
		"child" : this.fp.readUInt(),
		"sibling" : this.fp.readUInt()
	};

	//Evaluate bone scale
	if(!(node.evalflags & BIT_2)){
		bone.scale(node.scl);
	}

	//Evaluate bone rotation
	if(!(node.evalflags & BIT_1)){
		bone.rotate(node.rot, node.evalflags & BIT_5);
	}

	//Evaluate translation
	if(!(node.evalflags & BIT_0)){
		bone.translate(node.pos);
	}

	//Multiply by parent bone
	if(parentBone){
		bone.multiply(parentBone);
	}

	//Parse model
	if(node.model){
		//NJS_CNK_MODEL struct
		var model = {
			"vlist" : this.fp.readUInt(),
			"plist" : this.fp.readUInt(),
			"center" : this.fp.readVec3(),
			"radius" : this.fp.readFloat()
		};

		if(model.vlist){
			this.fp.seek_set(model.vlist);
			this.readVertexList(bone);
		}

		if(model.plist){
			this.fp.seek_set(model.plist);
			this.readPolygonList();
		}
	}
```

```
        //Recursive loop for child
        if(node.child){
                this.fp.seek_set(node.child);
                this.readNode(bone);
        }

        //Recursive loop for sibling
        if(node.sibling){
                this.fp.seek_set(node.sibling);
                this.readNode(parentBone);
        }

}
```

Ninja Vertex List

The ninja vertex list is a chunk which defines a list of vertices to be pushes to the current vertex buffer. The reason this section is refered to as a chunk is because it has no definitely length. The vertex list chunk declares a type which declares the format of the vertexes being used in the list. The chunk also declares the offset of where the vertexes in the list are to be written to in the vertex buffer.

The figure above depicts a Ninja vertex list as depicted in a hex editor. The first two dwords are the header for the chunk section and give the vertex type, flags, length of the section, start index of the vertex list in the vertex buffer and the number of vertexes

being defined. The block definition for these two dwords is as defined below.

```
[Chunkhead (0-7) | Chunkflags (8-15) | Short Size(16-31)] [Vertex Offset (0-15) | Number of Vertex (16-31)]
```

Chunkhead is a byte which defines the type of vertexes being declared in the list.

Chunkflags is a byte which defines the vertexes in an animation state.

Short Size is a short value which declares the length of the chunk is terms of short length.

Vertex Offset is a short value which declares the start index of the vertex list in the stack.

Number of Vertex is a short value which defines the number of vertexes in the list.

Ninja Vertex Type

The Chunkhead is a constant value type which defines the type of vertex being defined in the list. Each vertex data has an x, y, z, float coordinate, but can also include vertex normal data as well as vertex color data. Each of these types along with the cooresponding value for that defined type is provided in the list below.

```
#define NJD_VERTOFF            0x20    /* chunk vertex offset             */

/* optimize for SH4 */
#define NJD_CV_SH     (NJD_VERTOFF+0)  /* x,y,z,1.0F, ...                 */
#define NJD_CV_VN_SH  (NJD_VERTOFF+1)  /* x,y,z,1.0F,nx,ny,nz,0.0F,...    */

/* vertex */
#define NJD_CV        (NJD_VERTOFF+2)  /* x,y,z, ...                      */
#define NJD_CV_D8     (NJD_VERTOFF+3)  /* x,y,z,D8888,...                 */
#define NJD_CV_UF     (NJD_VERTOFF+4)  /* x,y,z,UserFlags32, ...          */
#define NJD_CV_NF     (NJD_VERTOFF+5)  /* x,y,z,NinjaFlags32,...          */
#define NJD_CV_S5     (NJD_VERTOFF+6)  /* x,y,z,D565|S565,...             */
#define NJD_CV_S4     (NJD_VERTOFF+7)  /* x,y,z,D4444|S565,...            */
#define NJD_CV_IN     (NJD_VERTOFF+8)  /* x,y,z,D16|S16,...               */

 /* vertex with normals */
#define NJD_CV_VN     (NJD_VERTOFF+9)  /* x,y,z,nx,ny,nz, ...             */
#define NJD_CV_VN_D8  (NJD_VERTOFF+10) /* x,y,z,nx,ny,nz,D8888,...        */
#define NJD_CV_VN_UF  (NJD_VERTOFF+11) /* x,y,z,nx,ny,nz,UserFlags32,...  */
#define NJD_CV_VN_NF  (NJD_VERTOFF+12) /* x,y,z,nx,ny,nz,NinjaFlags32,... */
#define NJD_CV_VN_S5  (NJD_VERTOFF+13) /* x,y,z,nx,ny,nz,D565|S565,...    */
#define NJD_CV_VN_S4  (NJD_VERTOFF+14) /* x,y,z,nx,ny,nz,D4444|S565,...   */
#define NJD_CV_VN_IN  (NJD_VERTOFF+15) /* x,y,z,nx,ny,nz,D16|S16,...      */

/* vertex with normal 32 */
#define NJD_CV_VNX    (NJD_VERTOFF+16) /* x,y,z,nxyz32, ...               */
#define NJD_CV_VNX_D8 (NJD_VERTOFF+17) /* x,y,z,nxyz32,D8888,...          */
#define NJD_CV_VNX_UF (NJD_VERTOFF+18) /* x,y,z,nxyz32,UserFlags32,...    */
```

There are four general types of vertexes defined in the Ninja library. 1) Optimized for SH4, 2) Vertex, 3) Vertex with Normals, 4) Vertex with Normal32. Types 1 (optimized for SH4) and 4 (Vertex with Normal32) and not used in Phantasy Star Online Version 2 and will not be described in this document.

The two types that are used, Vertex and Vertex with Normals follow the same general set of rules. Each one defines an X, Y, Z vertex point as floating point values. In the case normals are present there will be three floating points for Nx, Ny, and Nz. Lastly if depending on the Chunkhead, there will be an addition dword. D8888 represents the Vertex Color and will have one byte for each value, rgba, decribing the vertex diffuse

color. Userflags are not used in Phantasy Star Online 2. NinaFlags represents a value which represents an adjustment to the possitioning of the declared vertex in the vertex buffer.

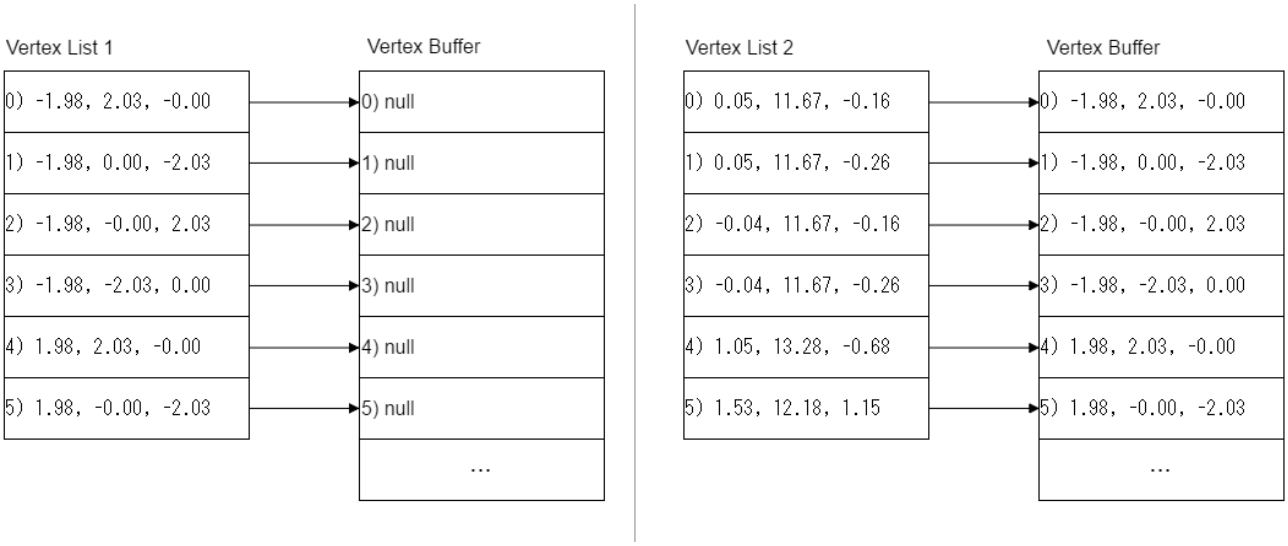
The last three values provide a specular value along side diffuse color in the Vertex Color. However most interfaces only support diffuse color for Vertex color and this data is often truncated when converting models to a current format. D565 and D4444 represent color formats for 16 bit colors. D565 is RGB565 and D4444 is RGBA4444. It is uncretain what format is being defined by D16.

Transforming Vertex Points

Each vertex point defined in the vertex list is defined for creating a mesh at the origin with no rotation, translation or scale information attached. Meaning that to place each Vertex where it exists within the model, the vertex and normal needs to be multiplied by the bone to place it where it needs to be with respect to the model. The position needs to be multiplied by the bone's transformation matrix. Normals need to be multiplied by the inverse transpose of the rotation data of the bone's transformation matrix.

Mapping Vertex Indexes

Vertex indices in Ninja models are required to be mapped when converting them to other format. Ninja models draw models in such a way that vertexes are pushes to a vertex buffer. When meshes are drawn they reference the current values in the vertex buffer. This means that indices inside the vertex buffer will be overwritten with new values as the model is drawn. This structure does not correspond with most 3d format which require all vertexes to be defined in one list to be referenced from. This means that two lists are required to be maintained when parsing Ninja models. One list which acts as a stack in which all of the values are pushes to sequentially and a second vertex which maps the current value of the vertex buffer to its possition in the stack.

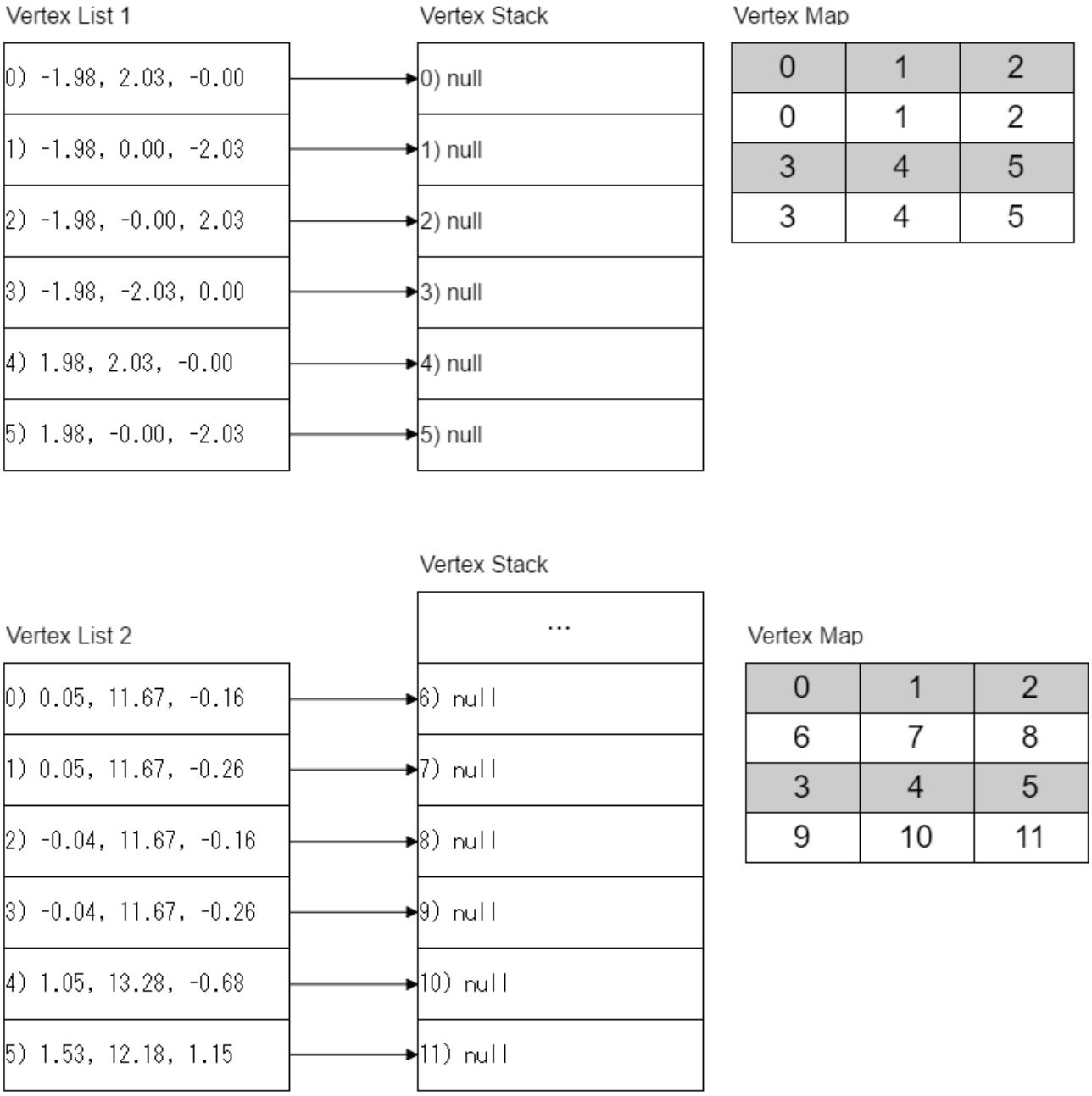


The above image represent how the Dreamcast manages vertex data. On the left side is a vertex list 1, which defines vertexes 0 - 5. To the right of it is the vertex buffer, which currently does not have any values in it. Each one of these values is pushed to the buffer for their corresponding indexes. Any meshes declared will use the values defined in vertex list 1 for indexes 0 - 5. On the left shows a second vertex list, Vertex List 2 defined later in the file, which uses the same indices. We see that the values from Vertex List 1 are currently written to the buffer. In this instance, the values from Vertex List 2 overwrite the values from vertex list 1 in the vertex buffer. Any meshes declared from this point will reference the values from Vertex list 2 for the indices 0-5.

As previously stated, this functionality is not desireable for most 3d software or formats which require vertexes to be defined as a static list. In such cases such as OpenGL, it might be possible to create a library which reads vertex values from an .nj file, pushes those values to a vertex buffer object, and renders each segment sequentially. However, even in these cases, the way UV values are declared does not align with OpenGL specifications and will likely causes problems. As such it is necessary to create a vertex stack which holds all of the values, and create a map for which value is located in that stack for a given vertex buffer index.

Vertex Index Mapping

The figure below displays this document's suggested method of mapping vertex locations to a single vertex stack.



The above figure displays a Vertex List 1 which declares the values for vertex indeices 0 - 5. Each of these values is pushes to the current vertex stack. In this case the stack is empty, so the locations decribed in the vertex list align with where they exist in the

stack. On the right shows the Vertex Map, which maps the current vertex buffer locations with their corresponding values in the stack.

Normal the second Vertex List which descibes the same index locations of 0-5 would overwrite the values from Vertex 1 in the vertex buffer. In this case, we actually push the values to the end of the vertex stack to preserve all of the values in the file. We then overwrite the stack locations in the vertex map, to respresent the new emulated values in the vertex stack. When meshes are declared, we use the vertex map as a lookup table to find the location of the values for which the indices are declared.

Vertex Ninja Flag

With the standard definition vertex lists are an order list of vertex points for a given index. An example being that if the vertex list has an index offset of 75 and 10 points are defined, then the vertex list will write the points to the vertex buffer from index 75 to 84. However, in some instances following this pattern will cause clipping, vertex assignment problems, and null values to be found where indices are declared for a given mesh. This is because in some instances the list is actually not in sequential order. In some cases Ninja Flags are used to adjust the offset of the vertex buffer index definition.

```
#define NJD_VERTOFF            0x20    /* chunk vertex offset             */
#define NJD_CV_NF     (NJD_VERTOFF+5)  /* x,y,z,NinjaFlags32,...          */
#define NJD_CV_VN_NF  (NJD_VERTOFF+12) /* x,y,z,nx,ny,nz,NinjaFlags32,... */
```

Ninja Flags are used when the Chunkheader of the vertex is is defined as 0x25 or 0x2C. The NinjaFlags itself if a 32 bit value which is made up of two short values. The first is the index for the vertex with respect to the the current index offset. The second short value is the distance from the end of the current vertex buffer size.

Index Offset	Relative Value	Distance From End	Vertex Buffer Position
75	0	154	75
76	2	152	77
77	4	150	79
78	8	146	83
79	9	145	84
80	10	144	85
81	11	143	86
82	12	142	87
83	13	141	88
84	21	123	96

The above table shows a sample of this relationship for a vertex list defined with the index offset of 75 with the length of 10 vertices. The first column, Index Offset, value on the left shows the standard sequential position of where each vertex would normally be defined in the vertex buffer. The second column, Relative Value, shows the value for the first short in the Ninja Flags. The third column, Distance From End, shows the value for the distance from the end of the current vertex buffer size. And the last column, Vertex Buffer Position, shows the actual position where the vertex is being declared in the vertex buffer.

Two things to note is that the sum of the Relative Value and the Distance from end for every entry in the vertex list should be a constant number, 154, in this case. The second thing to note is that Ninja Flags values overwrite their standard index offset definitions and do not interact with them. To give a specific example, for Index Offset 83, the relative value is 13. The 13 is added to the defined index offset of the vertex list, or 75 in

this case. It is not applied to the current relative position in which the vertex is normally being delcared sequentially.

Chunk Flags

```
/*--------------*/
/* Chunk Vertex */
/*--------------*/
/* Flag Weight status (NF only) */
#define NJD_FW_SHIFT     8
#define NJD_FW_START    (0 << NJD_FW_SHIFT)      /* Start  */
#define NJD_FW_MIDDLE   (1 << NJD_FW_SHIFT)      /* Middle */
#define NJD_FW_END      (2 << NJD_FW_SHIFT)      /* End    */

#define NJD_FW_MASK     (0x2 << NJD_FW_SHIFT)

/* Flag Vertex */
#define NJD_FV_SHIFT     8
#define NJD_FV_CONT     (0x80 << NJD_FV_SHIFT)   /* vertex calculation continue */
```

By default the chunk flag value for Vertex Lists should be 0. The Ninja Guide included SDK doesn't make any further mention than this. However in practical application, in addition to 0, there seem to be three other possible values: 0x80, 0x81, 0x82. Currently it is unknown the effect of these flags.

Ninja Vertex Source Code

Sample code for parsing a ninja vertex list is as defined below.

```
NinjaChunkModel.prototype.readVertexList = function(bone){

        //Read chunk header
        var cnkHead = this.fp.readByte();
        var cnkFlag = this.fp.readByte();
        var cnkLen  = this.fp.readUShort();

        //Return null if header value is 0xFF
        if(cnkHead == 0xFF){
                return 0;
        }

        //Read index offset and number of indexes
        var indexOfs = this.fp.readUShort();
        var nbIndex = this.fp.readUShort();

        //Declare temporary variables
        var pos, norm, clr, indexPos;

        //Loop through number of vertexes for each variable
        for(let i = indexOfs; i < indexOfs + nbIndex; i++){

                //Create object to store vertex values
                var vertex = {};

                //Read x,y,z coordinate
                pos = this.fp.readVec3();
                vertex['pos'] = bone.transformPoint(pos);

                //Map vertex to current bone
                vertex['skinIndex'] = bone.id;
                vertex['skinWeight'] = 1.0;

                //Vertex Normal definition
                if(cnkhead > 0x28 && cnkhead < 0x30){
                        norm = this.fp.readVec3();
                        vertex['norm'] = bone.transformNormal(norm);
                }

                //Vertex Color definitions
```

```
                if(cnkhead == 0x23 || cnkhead == 0x2A){
                        clr = this.fp.readColor('D8888');
                        vertex['clr'] = this.toThreeColor(clr);
                }else if(cnkhead == 0x26 || cnkhead == 0x2D){
                        clr = this.fp.readColor('D565|S565');
                        vertex['clr'] = this.toThreeColor(clr);
                }else if(cnkhead == 0x27 || cnkhead == 0x2E){
                        clr = this.fp.readColor('D4444|S565');
                        vertex['clr'] = this.toThreeColor(clr);
                } else if(cnkhead == 0x28 || cnkhead == 0x2F){
                        clr = this.fp.readColor('D16|S16');
                        vertex['clr'] = this.toThreeColor(clr);
                }

                //Ninja Flag definitions
                if(cnkhead == 0x25 || cnkhead == 0x2c){
                        var startOfs = this.fp.readUShort();
                        var endOfs = this.fp.readUShort();

                        indexPos = indexOfs + startOfs;
                }else{
                        indexPos = i;
                }

                //Do not push to stack if flag is declared otherwise
                if(cnkFlag > 0x80){
                        continue;
                }

                //Map index position to vertex stack
                this.vertexMap[indexPos] = this.vertexStack.length;
                this.vertexStack.push(vertex);
        }

        //Recall recursizely incase of multiple vertex lists
        this.readVertexList(bone);
}
```

Ninja Polygon List

The Polygon List is a group of chunks which defines the texture, material, and faces of a mesh. Each mesh is first defined by a material which gives the mesh attributes on which color it reflects, and its transparency. If a texture is applied to the meaterial,it will also be defined here. The actual faces of each mesh are defined by a strip list contained with in the Polygon List. In this manner the polygon list can declare on or more meshes.

```
00000540  30080000 13250400 FFFFFFFF 7F7F7FFF  0....%..ÿÿÿÿ...ÿ
00000550  08340040 41024200 0400FBFF 01003C00  .4.@A.B...ûÿ..<.
00000560  0D000900 21000000 00002000 1F000500  ....!..... .....
00000570  04000E00 0B000400 2F00FBFF 0B000400  ......../.ûÿ....
00000580  2F000600 20003F00 00002000 1F000300  /... .?... .....
00000590  3C003000 01003C00 0D00FBFF 09004100  <.0...<...ûÿ..A.
000005A0  AD000100 5B00BB00 0A004100 C7000300  ....[.»...A.Ç...
000005B0  5B00D300 06004100 E100FBFF 06004100  [.Ó...A.á.ûÿ..A.
000005C0  E1000B00 2700D300 07004100 C7000500  á...'.Ó...A.Ç...
000005D0  2700BB00 09004100 AD000000 41032000  '.»...A.....A. .
000005E0  0100F6FF 0700A500 C4000800 C700DE00  ..öÿ..¥.Ä...Ç.Þ.
000005F0  0600C700 C4000400 A500DE00 0A00A500  ..Ç.Ä...¥.Þ...¥.
00000600  C4000C00 8100DE00 09008100 C4000200  Ä.....Þ.....Ä...
00000610  A500DE00 0700A500 C4000800 C700DE00  ¥.Þ...¥.Ä...Ç.Þ.
00000620  FF000000 29004F00 00000D00 00000000  ÿ...).O.........
```

The image above shows a polygon list as seen in a hex editor. Similar to the vertex list,

the Polygon list has no definite length. The list is further divided into chunks. Each chunk defines a different attribute which is used to define a mesh. Each chunk is proceeded by a Chunk Head which is a Byte value that defines what each Chunk defines. Each chunk needs to be processed in order until a chunk header Byte value 0xFF is reached indicating the end of the chunk list.

Constant Name	Constant Value	Chunk Name	Chunk Head Size
NJD_NULLOFF	0	null chunk	16 bit
NJD_BITSOFF	1	chunk bits offset	16 bit
NJD_SAVEOFF	4	chunk save offset	16 bit
NJD_SEEKOFF	5	chunk seek to save offset	16 bit
NJD_TINYOFF	8	chunk tiny offset	32 bit
NJD_MATOFF	16	chunk material offset	32 bit
NJD_BUMPOFF	24	chunk bump map offset	32 bit
NJD_VERTOFF	32	chunk vertex offset	32 bit
NJD_STRIPOFF	64	chunk strip offset	
NJD_ENDOFF	255	end chunk offset	16 bit

The table above shows the different chunk types. To identify which chunk is being parsed, the first Byte in each chunk has a specific "offset" to define its type. By checking range, we are able to determine which chunk, and further more it's attributes. Each chunk has a defined Chunk Header size which defines the attributes of the chunk. Each header must be parsed to determine all of the attributes to obtain both the chunk size and the content of the chunk.

Stateful API

Like many other drawing programs, each polygon list is "stateful", meaning that settings such as diffuse color are carried over if multiple meshes are declared in the same polygon list. This point is often delivered moot, as often all of the attributes for each mesh are specifically defined before each strip list.

Ninja Polygon List Source Code

Source code for iterating over a Ninja Polygon list is as given below. Further information for parsing each chunk type will be given on subsequent pages.

```
NinjaChunkModel.prototype.readPolygonList = function(){

        //Single material for tracking state
        var mat = {};

        //Variable to save position when seeking
        var save = null;

        //Continually loop until 0xFF breaks loop
        while(true){

                //Read chunkflag and header
                var cnkhead = this.fp.readByte();
                var cnkflag = this.fp.readByte();

                //NJD_ENDOFF break
                if(cnkhead == 0xFF && save == null){
                        break;
                }

                //NJD_BITSOFF currently not supported
                else if(cnkhead >= 0 && cnkhead <= 3){
```

```
                        continue;
                }

                //NJD_SAVEOFF save position and exit
                else if(cnkhead == 4){
                        this.polygonMap[cnkflag] = this.fp.tell();
                        break;
                }

                //NJD_SEEKOFF seek to previouslt saved offset
                else if(cnkhead == 5){
                        save = this.fp.tell();
                        var seekPos = this.polygonMap[cnkflag];
                        this.fp.seek_set(seekPos);
                }

                //NJD_TINYOFF read texture id
                else if(cnkhead == 8 || cnkhead == 9){
                        this.readTexId(cnkhead, cnkflag, mat);
                }

                //NJD_MATOFF parse material for mesh
                else if(cnkhead >= 16 && cnkhead <= 23){
                        this.readMaterial(cnkhead, cnkflag, mat);
                }

                //NJD_BUMPOFF currently not supported
                else if(cnkhead == 24){
                        var cnklen = self.fp.readUShort();
                        this.fp.seek_cur(cnklen);
                }

                //NJD_STRIPOFF parse strip list for mesh
                else if(cnkhead >= 64 && cnkhead <= 75){
                        this.readStripList(cnkhead, cnkflag, mat);
                }

                //NJD_ENDOFF seek back to saved position
                else if(cnkhead == 255){
                        this.fp.seek_set(save);
                        save = null;
                }

                //Throw error to debug problems with parsing chunks
                else{
                        throw "Unknown chunk head detected: 0x" + chunkhead.toString(16);
                }

        }
}
```

Ninja Chunk 16 bit Definitions

This page provides defitions for the 16 bit chunk types defined in the Ninja Library.

0x00 Null Chunk

The null chunk is a 16 bit constant value: 0x0000. As most chunk content values are defined with shorts, the null chunk is used to align the next chunk to the next whole 32 bit boundrary after the end of a previous chunk if needed.

0x01 Bit Chunk

```
[ ChunkHead 0-7 | ChunkFlag (8-15) ]
```

Bit chunks is a 16 bit chunk which modifies the SRC and DST alpha values for alpha blending. The first byte, the ChunkHead, is used to identify the ChunkType. The flags for alpha blending are contained within the ChunkFlag. The ChunkFlag uses three bits to define an instruction for the Source and Destination alpha to define how they should be

blended. Definition for the ChunkFlag is as defined below.

```
[ DST Alpha Instruction (0-2) | DST Alpha Instruction (3-5) | NOP (6-7) ]
```

Values for each instruction type for Source and Destination are defined in the table below.

Instruction	Filed Value	Values Returned
Zero	0	(0, 0, 0, 0)
One	1	(1, 1, 1, 1)
'Other' Color	2	('Other' Red, 'Other' Green, 'other' Blue, 'Other' Alpha)
Inverse 'Other' Color	3	(1-'Other' Red, 1-'Other' Green, 1-'Other' Blue, 1-'Other' Alpha)
SRC Alpha	4	(SRC Alpha, SRC Alpha, SRC Alpha, SRC Alpha)
Inverse SRC Alpha	5	(1-SRC Alpha, 1-SRC Alpha, 1-SRC Alpha, 1-SRC Alpha)
DST Alpha	6	(DST Alpha, DST Alpha, DST Alpha, DST Alpha)
Inverse DST Alpha	7	(1-DST Alpha, 1-DST Alpha, 1-DST Alpha, 1-DST Alpha)

0x04 Save Chunk

The Save Chunk is used to define an offset inside the file and terminate Polygon list processing for that node. That position can then be traced back to and parsed from a later node. It is not known why this functionality is used, though it's implementation often seem to be used to declare a polygon list at the begining of the file, and return it to it after all of the vertex indices have been read into the vertex buffer.

```
[ Chunkhead (0-7) | ChunkFlag (8-15) ]
```

The Chunk is divided into two bytes. The first byte, the ChunkHead is used to define the Chunk and has a constant value of 0x04. The second value is the ChunkFlag which declares an index to save the current file offset to. This index is later used from the Seek Chunk to go back to the saved index. Polygon List parsing for that node is stopped when a Save Chunk is implemented. The saved offset is understood to be the position immediately after the end of the Save Chunk.

0x05 Seek Chunk

The Seek Chunk is used to seek to a previously defined offset in the file. It is comprised of two bytes.

```
 [ Chunkhead (0-7) | ChunkFlag (8-15) ]
```

The Chunkhead is a constant 0x05 defining the Chunk type. The second Byte, the ChunkFlag, is an index value used to distinguish which location to seek back to. For instance, if offset 0x158 were set to position 0, and offset 0x214 were set to position 1, a value of 1 as the ChunkFlag would seek back to offset 0x214. The program them parses the Polygon list at the prvious offset before encountering a 0xFF end chunk. Parsing then continues back at the original position of the Seek Chunk, though Seek Chunk are most often followed by end chunks, so this functionality can be considdered moot.

0x08 Tiny Chunk

The Tiny Chunk is a fixed 32 bit size chunk. It's purpose is to set the texture id for the defined mesh in the Polygon List, and clamp and flip flags assosiated with the texture.

```
[ ChunkHead (0-7) | ChunkFlag (8-15) | ChunkBody (16-31) ]
```

The ChunkHead is a Byte value used to identify the Chunk. The value is 0x08 for the first mesh defined in a polygon list and can be 0x09 when used to overwrite the state for subsequent meshes defined in the list. The ChunkFlags is a byte value which contains boolean flags for Mipmap depth, Flip U, Flip V, Clamp U and Clamp V accordingly. The structure of the ChunkFlag is as given below.

```
[ Mipmap Depth Adjust (0-3) | Clamp U (4) | Clamp V (5) | Flip U (6) | Flip V (7) ]
```

The ChunkBody is a short value which is used to set the texture id number for the mesh being defined in the Polygon list. However, there are sampling flags set in the top three bits, so only the first thirteen bits are used for texture, giving a max texture id of 8191. Bit 13 is a boolean flag for Super Sampling and the top two bits are used to set the texture filter mode. The structre for the ChunkBody is as defined below.

```
[ Texture Id (0-12) | Super Sample (13) | Filter Mode (14-15) ]
```

The Values for the Filter mode are as follows:
0 - Point Sampled
1 - Bilinear Filter
2 - Trilinear Filter

Tiny Chunk Source Code

Sample source code for parsing the Tiny Chunk is as defined below.

```
NinjaChunkModel.prototype.readTexId = function(cnkhead, cnkflag, mat){

        //Read Chunk Body
        var cnkBody = this.fp.readUShort();

        //Mask to get texture Id
        var texId = cnkBdy & 0x1FFF;

        //Super Sample
        var isSuperSample = (cnkBody >> 13) & BIT_1;

        //FilterMode
        var filterMode;
        switch(cnkBody >> 14){
                case 0:
                        filterMode = 'pointSampled';
                break;
                case 1:
                        filterMode = 'bilinearFilter';
                break;
                case 2:
                        filterMode = 'trilinearFilter';
                break;
        }

        //Set diffuse map to texture name
        mat['mapLight'] = this.texList[texId];

        //Get mipmap depth
        var mipDepth = cnkflag & 0xF;

        //Parse Clamp and flip flags from Flag
```

```
        mat['clampU'] = cnkflag & BIT_4;
        mat['clampV'] = cnkflag & BIT_5;
        mat['flipU']  = cnkflag & BIT_6;
        mat['flipV']  = cnkflag & BIT_7;
}
```

0x10 Material Chunk

The Material Chunk is used to define the 'material', or reflective color of the mesh being defined in the Polygon List. Similar to real life physics, colors on a mesh are not inheriently drawn on the screen. Rather they reflect light from a light source. The type of light and color they reflect are defined as diffuse, specular and ambient. Each one of these values is defined based upon whether a bit flag is placed in the ChunkHead. The structure for the Chunkhead is as given below.

```
[ ChunkHead (0-7) | ChunkFlag (8-15) | Short Length (16-31) ]
```

The ChunkHead is a single Byte value which defines the type of the chunk as a Material Chunk (0x10) width boolean flags for which reflection types (ambient, specular, diffuse) are defined in the chunk. The ChunkFlag is a Byte value which defines the Souce and Destination alpha for the mesh color defined in the Polygon Strip. Lastly the Short Length is a Short value which defines the length of the chunk in terms of short. For example, each of the reflective types ambient, specular and diffuse are defined with 32 bit values. Including all three would be a length of 12 bytes, or 6 shorts.

```
#define NJD_MATOFF             16
#define NJD_CM_D    (NJD_MATOFF+1)  /* 1 0 0 0 1 (17) Diffuse              */
#define NJD_CM_A    (NJD_MATOFF+2)  /* 1 0 0 1 0 (18) Ambiance             */
#define NJD_CM_DA   (NJD_MATOFF+3)  /* 1 0 0 1 1 (19) Diffuse + Ambiance   */
#define NJD_CM_S    (NJD_MATOFF+4)  /* 1 0 1 0 0 (20) Specular             */
#define NJD_CM_DS   (NJD_MATOFF+5)  /* 1 0 1 0 1 (21) Diffuse + Specular   */
#define NJD_CM_AS   (NJD_MATOFF+6)  /* 1 0 1 1 0 (22) Ambiance + Specular  */
#define NJD_CM_DAS  (NJD_MATOFF+7)  /* 1 0 1 1 1 (23) D + A + S            */
```

The header snippet above defines the boolean bytes used in the ChunkHead for parsing diffuse, specular and ambient respectively. Definition for the ChunkFlag is as defined below.

```
[ DST Alpha Instruction (0-2) | DST Alpha Instruction (3-5) | NOP (6-7) ]
```

Values for each instruction type for Source and Destination are defined in the table below.

Instruction	Filed Value	Values Returned
Zero	0	(0, 0, 0, 0)
One	1	(1, 1, 1, 1)
'Other' Color	2	('Other' Red, 'Other' Green, 'other' Blue, 'Other' Alpha)
Inverse 'Other' Color	3	(1-'Other' Red, 1-'Other' Green, 1-'Other' Blue, 1-'Other' Alpha)
SRC Alpha	4	(SRC Alpha, SRC Alpha, SRC Alpha, SRC Alpha)
Inverse SRC Alpha	5	(1-SRC Alpha, 1-SRC Alpha, 1-SRC Alpha, 1-SRC Alpha)
DST Alpha	6	(DST Alpha, DST Alpha, DST Alpha, DST Alpha)

Instruction	Filed Value	Values Returned
Inverse DST Alpha	7	(1-DST Alpha, 1-DST Alpha, 1-DST Alpha, 1-DST Alpha)

Material Chunk Source Code

Sample code for parsing the material chunk is as given below.

```
NinjaChunkModel.prototype.readMaterial = function(cnkhead, cnkflag, mat){

	//Read Chunk Size
	var cnkLen = this.fp.readUShort();

	//Read alpha instruction from cnkunk flag
	var dstAlpha = cnkflag & (BIT_0|BIT_1|BIT_2);
	cnkflag = cnkflag >> 3;
	var srcAlpha = cnkflag & (BIT_0|BIT_1|BIT_2);

	//Delcare temporary variables
	var n, r, g, b, a, e;

	//Diffuse
	if(cnkhead & BIT_0){
		g = this.fp.readByte() / 255;
		b = this.fp.readByte() / 255;
		r = this.fp.readByte() / 255;
		a = this.fp.readByte() / 255;

		//Set material diffuse values
		mat['colorDiffuse'] = [r, g, b];
		mat['opacity'] = a;
	}

	//Ambience
	if(cnkhead & BIT_1){
		g = this.fp.readByte() / 255;
		b = this.fp.readByte() / 255;
		r = this.fp.readByte() / 255;
		n = this.fp.readByte();

		//Set material ambient values
		mat['ambient'] = [r, g, b];
	}

	//Specular
	if(cnkhead & BIT_2){
		g = this.fp.readByte() / 255;
		b = this.fp.readByte() / 255;
		r = this.fp.readByte() / 255;
		e = this.fp.readByte() & 0xF;

		//Set material specular values
		mat['colorSpecular'] = [r, g, b];
		mat['specularCoef'] = e;
	}

}
```

0x40 Strip Chunk

The Strip Chunk is, exactly as the name implies, a Chunk in which strips are declared to define the vertex positions of the mesh. The chunk header is 32 bits in width and defined below.

```
[ ChunkHead (0-7) | ChunkFlag (8-15) | Short Length (16-31) ]
```

The ChunkHead is a Byte value which defines the Chunk as a Strip Chunk, and a

modifier which indicates the data format inside the chunk. Each strip chunk is made up of multiple strip. Each strip is defined by a list of indices. Each indices is defined by a short value which decribes the location in vertex buffer of the point of where the triangle face for the mesh is defined. In addition to indices, texture mapping, normals, and face colors can also be defined in this section, though only texture mapping is applied in practive. The full list of possibilities is as defined below.

```
#define NJD_STRIPOFF 64
#define NJD_CS          (NJD_STRIPOFF+0)  /* index   */
#define NJD_CS_UVN      (NJD_STRIPOFF+1)  /* index, UN, VN   */
#define NJD_CS_UVH      (NJD_STRIPOFF+2)  /* index, UH, VH   */

#define NJD_CS_VN       (NJD_STRIPOFF+3)  /* index, vnx, vny, vnz   */
#define NJD_CS_UVN_VN   (NJD_STRIPOFF+4)  /* index, UN, VN, vnx, vny, vnz   */
#define NJD_CS_UVH_VN   (NJD_STRIPOFF+5)  /* index, UH, VH, vnx, vny, vnz   */

#define NJD_CS_D8       (NJD_STRIPOFF+6)  /* index, AR, GB   */
#define NJD_CS_UVN_D8   (NJD_STRIPOFF+7)  /* index, UV, VN, AR, GB   */
#define NJD_CS_UVH_D8   (NJD_STRIPOFF+8)  /* index, UH, VH, AR, GB   */
```

The most common values are 0x40, index only which is used when only material is applied to the mesh and no texture is applied. 0x41, which is index and 'low resolution' mapping, or a signed short value which is divided by 255 to give the UV mapping value. 0x42 which is an index and 'high resolution' mapping, a signed short value which is divided by 1023 for the mapping value. The ChunkFlag is a Byte value which defines boolean flags for interpreting the mesh strip. The flags are defined as follows.

```
#define NJD_FST_IL        BIT_0 /* IL  : Ignore light      */
#define NJD_FST_IS        BIT_1 /* IS  : Ignore specular   */
#define NJD_FST_IA        BIT_2 /* IA  : Ignore ambient    */
#define NJD_FST_UA        BIT_3 /* UA  : Use alpha         */
#define NJD_FST_DB        BIT_4 /* DB  : Double side       */
#define NJD_FST_FL        BIT_5 /* FL  : Flat shading      */
#define NJD_FST_ENV       BIT_6 /* ENV : Environment       */
```

The short length defines the length of the Chunk in terms of shorts, as both the index value, U value, and V value, and strip length are all short values, it can be summarized as the number of values contained in the strip. Length with respect to bytes can be found by multiplying the Short Length by two.

Format

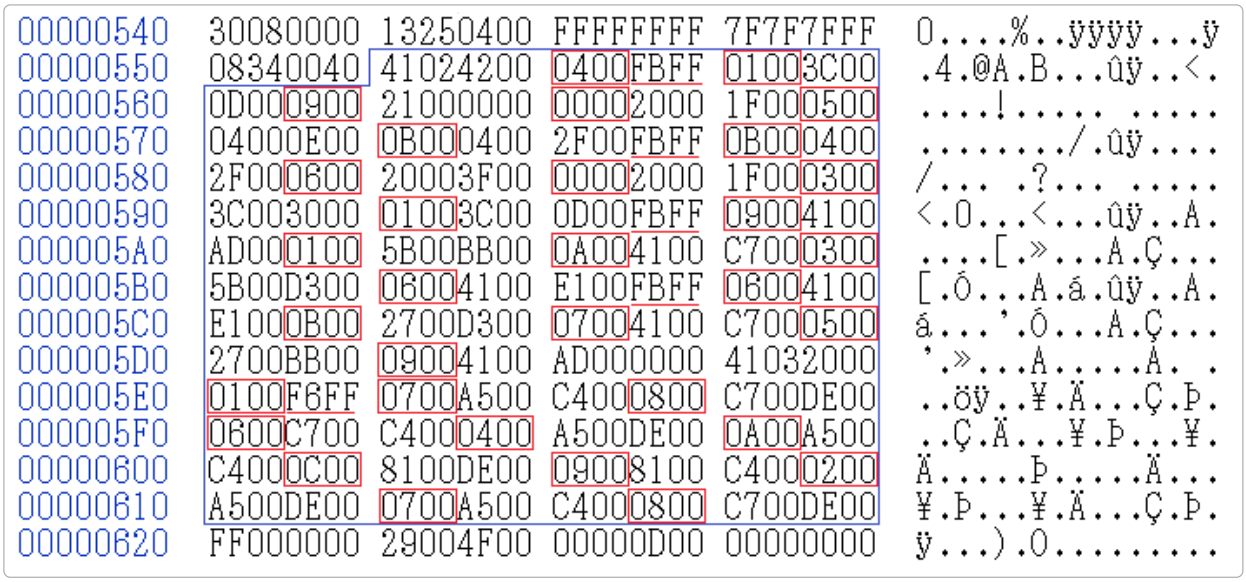
The Strip Chunk is a list of strips which defines the face triangles for a mesh beign defined in the polygon list. The first short in the content of the Strip Chunk is the number of strips, the first short preceeding a strip is the number of indices in the strip, followed by the list of indices. A simplified view of this is displayed below.

```
Number Strips (4)
Strip Length (-5) 1, 9, 0, 5, 11
Strip Length (-5) 11, 6, 0, 3, 1
Strip Length (-5) 9, 1, 10, 3, 6
Strip Length (-5) 6, 11, 7, 5, 9
```

As defining triangles in a mesh often trace back over the same indices repeatedly, it is a common methodlogy to express them as strips. Strips are abreivated lists of triangles, that are later expanded into an array of A,B,C values indicating each face. Strips can be defined as clockwise or counter clockwise depending on the required draw order. Strips are to be drawn counter clockwise have the length expressed with a negative number (signed), whereas clockwise strips are expressed with a possitive number.

0x40 Strip Chunk Parse Example

The following figure is an example of how to parse a strip chunk.



```
00000540  30080000 13250400 FFFFFFFF 7F7F7FFF  0....%..ÿÿÿÿ...ÿ
00000550  08340040 41024200 0400FBFF 01003C00  .4.@A.B...ûÿ..<.
00000560  0D000900 21000000 00002000 1F000500  ....!..... .....
00000570  04000E00 0B000400 2F00FBFF 0B000400  ......../.ûÿ....
00000580  2F000600 20003F00 00002000 1F000300  /... .?... .....
00000590  3C003000 01003C00 0D00FBFF 09004100  <.0...<...ûÿ..A.
000005A0  AD000100 5B00BB00 0A004100 C7000300  ....[.»...A.Ç...
000005B0  5B00D300 06004100 E100FBFF 06004100  [.Ó...A.á.ûÿ..A.
000005C0  E1000B00 2700D300 07004100 C7000500  á...'.Ó...A.Ç...
000005D0  2700BB00 09004100 AD000000 41032000  '.»...A.....A. .
000005E0  0100F6FF 0700A500 C4000800 C700DE00  ..öÿ..¥.Ä...Ç.Þ.
000005F0  0600C700 C4000400 A500DE00 0A00A500  ..Ç.Ä...¥.Þ...¥.
00000600  C4000C00 8100DE00 09008100 C4000200  Ä.....Þ.....Ä...
00000610  A500DE00 0700A500 C4000800 C700DE00  ¥.Þ...¥.Ä...Ç.Þ.
00000620  FF000000 29004F00 00000D00 00000000  ÿ...).O.........
```

The ChunkHead value is 0x41 indicating this is a strip chunk with index and 'low resolution' UV texture maps defined. The ChunkFlag is 0x02, or BIT_1, the boolean flag to ignore specular. The Short Length is 0x0042, indicating there are 66 short values contained in the chunk.

Following that is the strip definition, the first short 0x0004 defines the number of strips to be defined in this strip chunk. Following that, the underlined value 0xFFFB is a signed short, or -5 in decimal, expressing a counter clockwise strip of length 5. As the ChunkHead is defined as 0x41, index with 'low res' UV, each index is followed by a signed short for U, and likewise, a signed short for V. This means that between each index there is a 4 byte gap. Thus for indices, we parse out 0x0001, 0x0009, 0x0000, 0x0005 and 0x000B for strip indices, before coming to the next strip length definition.

This proceeds until all four strips are defined in the chunk and we end at offset 0x05DB, in which we encounter a Null Chunk to adjust the offset of the next chunk definition header to the next full 32 bit width. Following that, there is immediately another Strip Chunk Definition with 0x41 as the ChunkHead, 0x03 for the ChunkFlag indicating to ignore light and ignore specular, and a short length of 0x0020. There is only one strip declare with a length of 0xFFF6 or 10 counter clockwise indices.

With this example, we are able to observe the statefulness of the Ninja Graphics Library. Rather than having to declare a new material and texture id for another Strip Chunk, only the required flags, (ignore light and ignore specular) in the subsequent ChunkHead we rewritten and the API continues to render everything else with the current values set in the library.

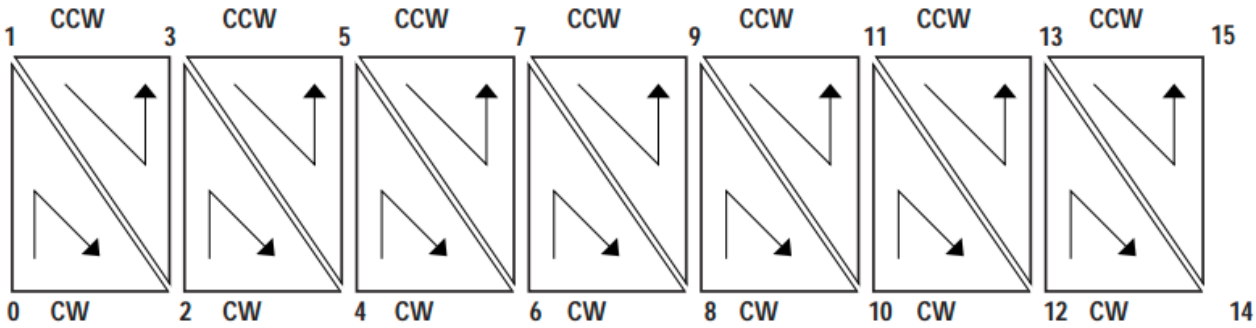
```
Strip Chunk
Format: Index, U, V
Flags: Ignore Specular
Length: 66 Short Values
Number Strips (4)
Strip Length (-5) 1, 9, 0, 5, 11
Strip Length (-5) 11, 6, 0, 3, 1
Strip Length (-5) 9, 1, 10, 3, 6
Strip Length (-5) 6, 11, 7, 5, 9

Strip Chunk
Format: Index, U, V
Flags: Ignore Light, Ignore Specular
Length: 32 Short Values
Number Strips (1)
```

```
Strip Length (10) 7, 8, 6, 4, 10, 12, 9, 2, 7, 8
```

A sample debug output for parsing this chunk is described above.

0x40 Strip Chunk Triangle Order

The figure below illustrates the draw order of a strip in the Ninja Graphics API.



Triangles in strips are evaluated in alternating clockwise, counter clockwise manner. As such whether a strip is clockwise or counter clockwise, only defines the direction of the first triangle. All following triangles then follow by alternating the direction for the next triangle and the next triangle onward until the end of the strip. This is different from other graphics utilities which often restrict strips to one direction or the other. Strips can be converted into a list of triangles from the following function.

Each triangle require three indices to declare. This means that the total number of triangles decalre in a strip should be the strip length minus two. A strip length of three only defines one triangle, a strip length of four defines two triangles, strip length of five with be three triangles and so on.

Source Code

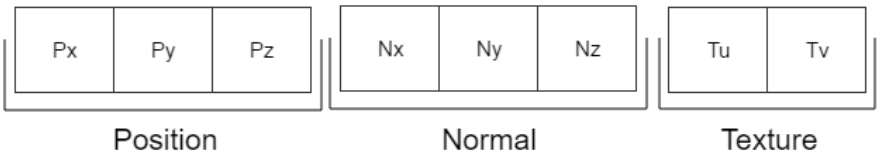
Below is example code for converting strips to raw triangles. For the five indexes (1, 9, 0, 5, 11), and counter clockwise true, the output should be three triangles with indices defined as: (1, 0, 9), (9, 0, 5), (0, 11, 5).

```
NinjaChunkModel.prototype.stripToTriangle = function(stripList){


        //Create a list for A, B, C, values
        var triangleList = [];

        //Define temporary variables
        var a, b, c;

        //Loop through all the strips for the mesh being defined
        for(let i = 0; i < stripList.length; i++){

                //Map strip to local name
                var strip = stripList[i];

                //Direction is saved as a boolean in the first array slot
                var ccw = strip.shift();

                for(let k = 0; k < strip.length - 2; k++){

                        //Three indices to define each triangle
                        a = strip[k + 0];
                        b = strip[k + 1];
                        c = strip[k + 2];

                        //if counter-clockwise, when k is even, otherwise when k is odd
```

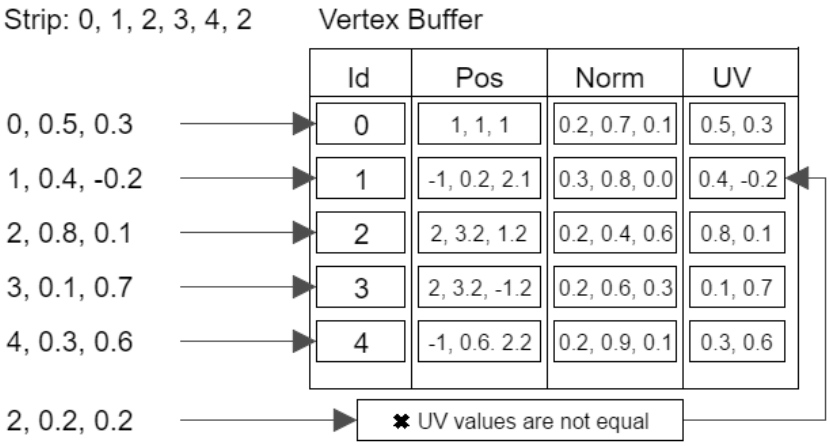
```
                        if ((ccw && !(k%2)) || (!ccw && k%2)){
                                triangleList.push([a, c, b]);
                        } else {
                                triangleList.push([a, b, c]);
                        }
                }

        }

        //Return the triangle list
        return triangleList;
}
```

0x40 Strip Chunk Vertex Mapping

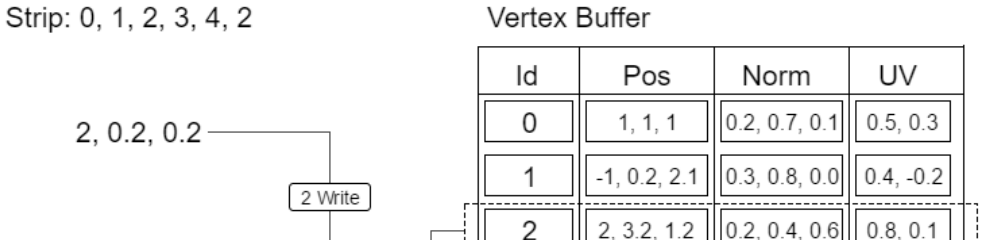
The Ninja Graphics API defines texture mapping defines UV coordinates in the strip list. For instance the strip list may use the same indice multiple times and declare a new UV value each time the value is used. This is counter intuitive to most recent generating 3d libraries.

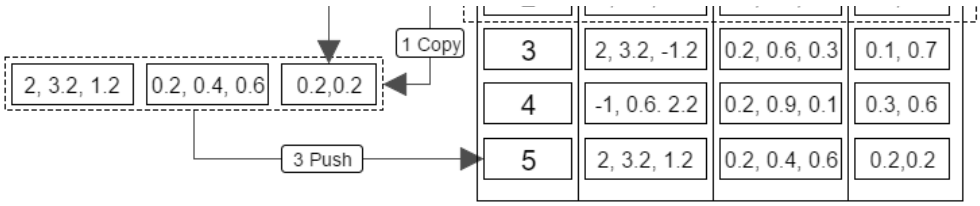


The image above displays how most vertexes are grouped as a vertex buffer object. Each vertex position is paired with its vertex normal and its texture mapping coordinates. This means that for a situation such as the Ninja Graphics Library where texture mapping is declared in the index, we need to clone the position and normal values for a given vertex, change the UV value and then push that 'new' vertex to the vertex stack.



The above image shows this relationship. We have a strip delcared as 0, 1, 2, 3, 4, 2 with UV values listed for each index. When parsing this strip, we look at all of the values in the vertex stack for uv values. If no UV value has been set for a given vertex stack index, then we simply set the listed value of the strip index to the stack. However in this example the second strip index 2 has a different value from the first 2 index listed in the strip.





We copy the vertex value from the 2 index from the vertex stax, we overwrite the uv value with the uv value for the second index, and then push it to the vertex stack as a new vertex. We then need to remap the second 2 index to the new vertex stack location 5. So the new strip indexes should be 0, 1, 2, 3, 4, ~~2~~ 5. In order to avoid duplicating the same vertex too many times, it is suggested to create a vertex map for every uv index, so that each different index is checked against the current map, and only appended as a new vertex to the stack if none of the other indexes match. Also it is recomended this is done before generating triangles.

0x40 String Chunk Vertex Map Source Code

The code provided below is an example for how to map UV values to a vertex stack.

```
NinjaChunkModel.prototype.mapUvToIndex(index, u, v, mat){

	//Check for flip flags
	if(mat['flipU']){
		u = -u;
	}

	if(mat['flipV']){
		v = -v;
	}

	//Check for clamp flags
	if(mat['clampU']){
		u = u < 0 ? 0 : u;
		u = u > 1 ? 1 : u;
	}

	if(mat['clampV']){
		v = v < 0 ? 0 : v;
		v = v > 1 ? 1 : v;
	}

	//Check if the current index is not set in map
	if(!this.uvMap[index]){

		//If not set, then set first index to current index
		this.uvMap[index] = [index];

		//Set the UV coords of the vertex to the ones provided
		this.vertices[index]['uv'] = [u, v];

		//Return the current index
		return index;
	}



	//If set, then iterate over the current vertex map for index
	for(let i = 0; i < this.uvMap[index].length; i++){

		//Read the mapped index from the uv map
		var mappedIndex = this.uvMap[index][i];

		//Create a local variable for the mapped Vertex
		var vertex = this.vertexStack[mappedIndex];

		//Check the provided UV values agains the one set
		if(vertex['uv'][0] == u && vertex['uv'][1] == v){
```

```
                        //If they match then return the mapped index
                        return mappedIndex;
                }
        }

        //When value is not found, we add a new vertex to the stack
        var vertex = this.vertiexStack[index];

        //Clone to avoid passing values by reference
        var newVertex = clone(vertex, false);

        //Overwrite UV value with current value
        vertex['uv'] = [u, v];

        //Save the current stack length
        var stackLength = this.vertexStack.length;

        //Add new vertex to end of stack
        this.vertexStack.push(newVertex);

        //Add new vertex index to uv vertex map
        this.uvMap[index].push(stackLength);

        //Return the mapped index
        return stackLength;
}
```

0x40 Strip Chunk Material Id

The Ninja Graphics API is a stateful library in which any changes to the API are set and remembered for concurrent calls. This presents a problem with materials, as while this was standard for that API, it is not the case with most current libraries, it is necessary to try to combine materials as often as possible to avoid major perforance hits later.

Each 3d graphics API and format is likely going to support, or not support different functionalities for materials. If the destination format does not support any of the following functions then it would be a good idea to comment them out as to reduce drawing calls. As for the Ninja Graphics API, the full list of material flags is as follows.

1	Diffuse Color	RGBA Value
2	Ambient Color	RGB Value
3	Specular Color	RGB Value
4	Specular Coefficient	Intger 0 - 15
5	SRC Alpha Instruction	Instruction
6	DST Alpha Instruction	Instruction
7	Clamp U	true / false
8	Clamp V	true / false
9	Flip U	true / false
10	Flip V	true / false
11	Texture Name	String ('.png')
12	Ignore Light	true / false
13	Ignore Specular	true / false
14	Ignore Ambient	true / false
15	Use Alpha	true / false
16	Double Side	true / false
17	Flat Shading	true / false
18	Environment Mapping	true / false

Also make note of any functionality such as double-sided, or flipping UV which can be implemented by applying them directly when parsing as to reduce the number of materials.

```
NinjaChunkModel.prototype.getMaterialId = function(mat){

	//Clone to avoid changes by reference
	var newMat = clone(mat, false);

	//Remove unused flags
	delete newMat['doubleSide'];
	delete newMat['flipU'];
	delete newMat['flipV'];

	//Loop through  current list of materials
	for(let i = 0; i < this.materials.length; i++){

		//Set reference to local variable
		var compareMat = this.materials[i];

		//If the two json object values are the same
		if(equals(newMat, compareMat){
			//Return material index
			return i;
		};
	}

	//get current material length
	var matId = this.materials.length;
	//Push to the current list
	this.materials.push(newMat);

	//Return material index
	return matId;
}
```

0x40 Strip Chunk Source Code

The following is example source code for how to parse a strip chunk.

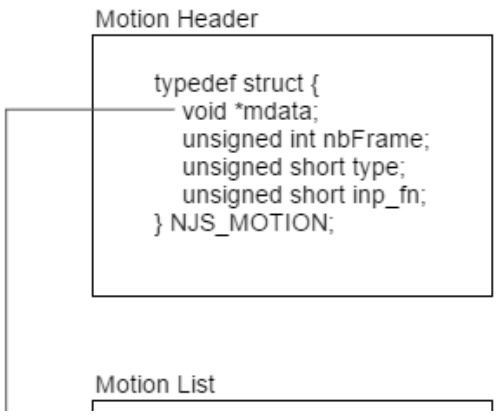
```
NinjaChunkModel.prototype.readStripList = function(cnkhead, cnkflag, mat){

	//Read Short length of chunk
	var shortLength = this.fp.readUShort();

	//Read Number of strips
	var nbStrip = this.fp.readUShort();

	//Parse boolean flags for chunk flag
	mat['ignoreLight']    = cnkflag & BIT_0;
	mat['ignoreSpecular'] = cnkflag & BIT_1;
	mat['ignoreAmbient']  = cnkflag & BIT_2;
	mat['useAlpha']       = cnkflag & BIT_3;
	mat['doubleSide']     = cnkflag & BIT_4;
	mat['flatShading']    = cnkflag & BIT_5;
	mat['envMapping']     = cnkflag & BIT_6;

	//Create object to hold mesh definition
	var mesh = {
		'matId' : this.getMaterialId(mat),
		'triangleFaces' : []
	};

	//iterate over all of the strips
	for(let i = 0; i < nbStrips; i++){

		//Create strip
		var strip = [];

		//Read strip length as signed short
		var stripLength = this.fp.readShort();
```

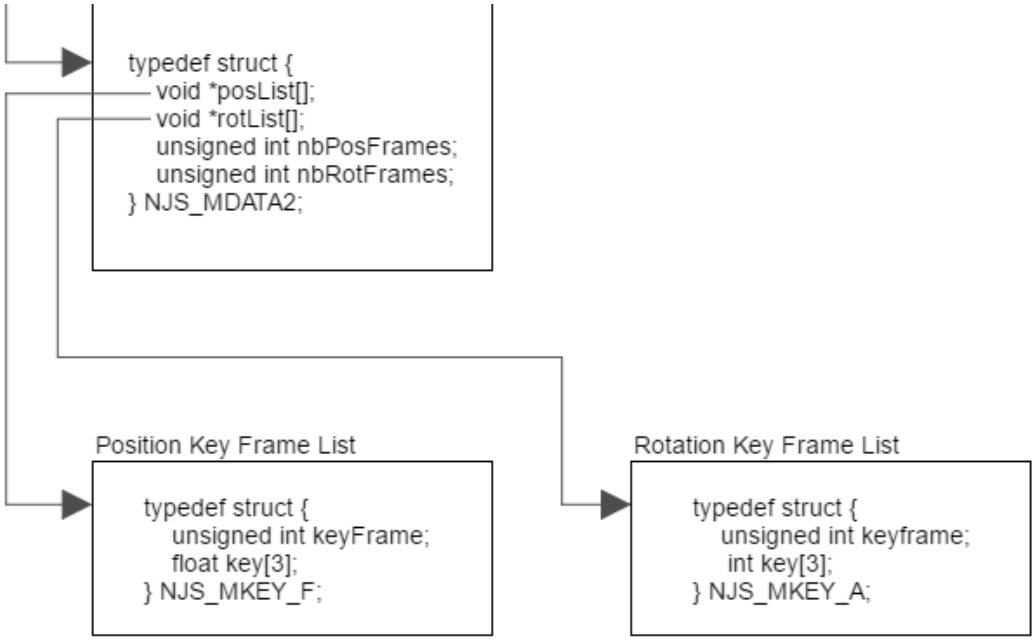
```
                //Add counter-clockwise boolean
                strip.push(stripLength < 0);
                stripLength = Math.abs(stripLength);

                //Loop over each index in the strip
                for(let j = 0; j < stripLength; j++){
                        //Read index
                        var index = this.fp.readUShort();
                        //Map to vertex stack index
                        index = this.vertexMap[index];

                        //Read UV coords and map to vertex indices
                        if(cnkhead == 0x41){
                                var u = this.fp.readShort() / 255;
                                var v = this.fp.readShort() / 255;
                                index = this.mapUvToIndex(index, u, v, mat);
                        }else if(cnkhead == 0x42){
                                var u = this.fp.readShort() / 1023;
                                var v = this.fp.readShort() / 1023;
                                index = this.mapUvToIndex(index, u, v, mat);
                        }

                        //Add index to current strip
                        strip.push(index);
                }

                //Convert strip to a list of triangle faces
                var triangleList = this.stripToTriangle(strip);

                //Add triangle faces to current mesh
                for(let j = 0; j < triangleList.length; j++){
                        mesh.triangleFaces.push(triangleList[j]);
                }
        }

        //Add mesh to current mesh list
        this.meshList.push(mesh);
}
```

It depends on the implementation of the intended target format, however this implementation create a list of triangle faces which are mapped to vertexes with the correct vertex normal, color and uv, and should be adaptable to almost any format.

Ninja Motion

Ninja Motion or .njm, is the file format used to animation ninja chunk models. Each animation consists of a number of key frames in which the animation takes place. Then for each bone the model, and for each frame in the bone, the file defines a position and a rotation. The position and rotation for each matrix are multiplied into a transformation matrix. Every vertex in the bone is then multiplied by that trnasformation matrix to position each bone segment for the animation. When key frames are skipped linear interpolation is used to calculate the frames between each keyframe.





The image above shows the overall structure for this file format. We have a header which defines the number of frames and type, which points to the Motion List. The Motion List is an array which defines pointers arrays translation and rotation keyframe data for each bone. Each keyframe provides a corresponding keyframe number and value which is used to transform every vertex attached to the bone to produce an animation.

```
00000000  4E4D444D FC0A0000 FC090000 14000000  NMDMü...ü.......
00000010  03000200 00000000 00000000 00000000  ................
```

Similar to most other file format inside Phantasy Star Online Version 2, the file starts off with an interchange file format header, in this case "NMDM" to signify the file contains Ninja Motion Data content. Followed by the length of the content, in this case, 0x0AFC. Again similar to other file formats in PSO Version 2, the internal pointers of this section are set to zero internally. This documentation will focus only on the file specification.

Ninja Motion Header

The figure below shows the Motion Header with the interchange file format header removed to adjust for the internal file pointers being set to zero. The Motion Header is outlined.

```
00000000  FC090000 14000000 03000200 00000000  ü...............
00000010  00000000 00000000 00000000 01000000  ................
```

The struct definition for the Motion Header is as follows.

```
typedef struct {
        void *mdata;            /* Array for object tree                        */
        unsigned int nbFrame;   /* Number of motion frames                      */
        unsigned short type;    /* Motion element bit string                    */
        unsigned short inp_fn;  /* Interpolation method and number of elements */
} NJS_MOTION;
```

The first dword is mdata, which is a void pointer to the Motion List. The motion list is an array of NJS_MDATA struct, the length of nodes for the model for which motion is being

defined.

The next dword is nbFrame, or the number of frames in which this animation takes place. In the example above, the length is 0x14 in hex, or 20 frames. For every motion for every motion defined in the file, there will be a key frame number attached to it.

The next word defined is the type. The type is defined by bit flags which set the type of motion being defined in the animation. The Bit flags are defined below.

```
NJD_MTYPE_POS_0   BIT_0
NJD_MTYPE_ANG_1   BIT_1
NJD_MTYPE_SCL_2   BIT_2
NJD_MTYPE_SHAPE_3 BIT_3
NJD_MTYPE_VEC_4   BIT_4
NJD_MTYPE_ANG_X_5 BIT_5
```

For PSO version 2, this number should always be 0x03, or (NJD_MTYPE_POS_0 | NJD_MTYPE_ANG_1). This means that for every NJS_MDATA instance in the Motion List will have two pointers, one for translational and one for rotation key framed values. The code snippet below gives a the definition for the struct used in the Motion List.

```
typedef struct {
    void *posList;
    void *rotList;
    unsigned int nbPosFrames;
    unsigned int nbRotFrames;
} NJS_MDATA2;
```

The last word in the header, inp_fn, has two purposes. Firstly it provides a confirmation for the number of pointers used in the NJS_DATA structure. In this case 0x02, which corresponsed to one for the translation, and one for the rotational motion. The second purpose is to define the interpolation function between each frame.

```
#define NJD_MTYPE_LINER  0x0000 /* Linear interpolation       */
#define NJD_MTYPE_SPLINE 0x0040 /* Spline interpolation       */
#define NJD_MTYPE_USER   0x0080 /* User function interpolation*/
#define NJD_MTYPE_MASK   0x00c0 /* Sampling mask              */
```

The three possible interpolation types are linear, spline and user defined. The mask value for indentifying these types is 0x0c. For Phantasy Star Online Version 2, the value is always defined as zero for linear interpolation.

Ninja Motion List Types

The Motion List is an array of the struct NJS_MDATA for each node is the model motion is being defined for. Each type of motion is defined in the 'type' variable defined in the NJS_MOTION header. The types are as defined below.

```
NJD_MTYPE_POS_0   BIT_0
NJD_MTYPE_ANG_1   BIT_1
NJD_MTYPE_SCL_2   BIT_2
NJD_MTYPE_SHAPE_3 BIT_3
NJD_MTYPE_VEC_4   BIT_4
NJD_MTYPE_ANG_X_5 BIT_5
```

There are sinx types in total, but for any given animation, there can only be one type of anglular (rotation), method defined. This means that the NJS_MDATA struct can have anywhere from one pointer to five. Each type of NJS_MDATA struct is numbered from 1 to 5 accordingly. As such the types are as follows.

```
//NJS_MDATA1
```

```
typedef struct {
        void *p[1];         /* Motion pointer     */
        unsigned int nb[1]; /* Number of keyframes*/
} NJS_MDATA1;

//NJS_MDATA2
typedef struct {
        void *p[2];         /* Motion pointer     */
        unsigned int nb[2]; /* Number of keyframes*/
} NJS_MDATA2;

//NJS_MDATA3
typedef struct {
        void *p[3];         /* Motion pointer     */
        unsigned int nb[3]; /* Number of keyframes*/
} NJS_MDATA3;

//NJS_MDATA4
typedef struct {
        void *p[4];         /* Motion pointer     */
        unsigned int nb[4]; /* Number of keyframes*/
} NJS_MDATA4;

//NJS_MDATA5
typedef struct {
        void *p[5];         /* Motion pointer     */
        unsigned int nb[5]; /* Number of keyframes*/
} NJS_MDATA5;
```

As previously stated, in application for PSO version 2, the format will always be for position and roation using NJS_MDATA2. This data has been included for the event other file formats needed to be cross referenced against this one.

Pointer and number of frames order will always be in the order in which the bitflags are evaluated. In the case of Phantasy Star Online 2, the order is NJD_MTYPE_POS_0 followed by NJD_MTYPE_ANG_1. This produced the psudo-struct format which was presented on previous pages.

```
typedef struct {
    void *posList;
    void *rotList;
    unsigned int nbPosFrames;
    unsigned int nbRotFrames;
} NJS_MDATA2;
```

This is the format this document will use as it better illustrates relationships between structures. A word of caution is adviced as it might not hold true for other titles. Another thing to mention is that that *p pointer defined in the NJS_MDATA struct changes depending on what kind of data it is pointing to. More information on this will be covered on the next page.

Ninja Motion List Key Frame Types

The Ninja Graphics API defines six different kind of motions: position, angle, scale, shape, vector, and angleX. As each animation definition is different, so to is the keyframe value being defined. Each key frame contains a key frame number which defines which key frame the value is ato be applied to, and the value itself. Each value type depends on the definition of the motion type.

Float Type

The float type defines a keyframe number and three floats. This key frame struct applies to NJD_MTYPE_POS_0 (position), NJD_MTYPE_SCL_2 (scale), and NJD_MTYPE_VEC_4 (vector).

```
typedef struct {
        unsigned int keyframe; /* Keyframe number*/
        float key[3];          /* Float type key value (array 3)*/
} NJS_MKEY_F;
```

Angle Type

The angle type is used for NJD_MTYPE_ANG_1(rotation) and NJD_MTYPE_ANG_X_5 (rotation X). As NJD_MTYPE_ANG_X_5 does not occur within PSO version 2 data, the current implementation is not tested. The author conjects that this may refer to the ZXY data format used for Lightwave evaluated models, though there is no confirmation of this in the official documentation. The struct type itself is a keyframe combined with an array of three signed ints. The interger values work the same as defined in the Ninja Chunk Model node rotation. 0x0000 is 0 degrees, and 0xFFFF corresponds to 360. As such the value for each integer values needs to be multiplied by (360 / 0xFFFF) to get the angle value.

```
typedef struct {
        unsigned int keyframe; /* Keyframe number*/
        int key[3];            /* Angle type key value (array 3)*/
} NJS_MKEY_A;
```

Shape Motion

The shape motion modifier is not used in Phantasy Star Online Version 2, so it is unknown how this animation works in practical application. One thing to note is that there is a Bitflag in the evalflags definition of the NJS_OBJECT struct defined in the Ninja Chunk Model specification. That boolean cooresponds with this section for whether to evaluate or not. The following format is used when the NJD_MTYPE_SHAPE_3 (shape) boolean flag is set in the type field of the NJS_MOTION structure.

```
typedef struct {
        unsigned int keyframe; /* Keyframe number               */
        unsigned int key;      /* Unsigned int32 type key value */
} NJS_MKEY_UI32;
```

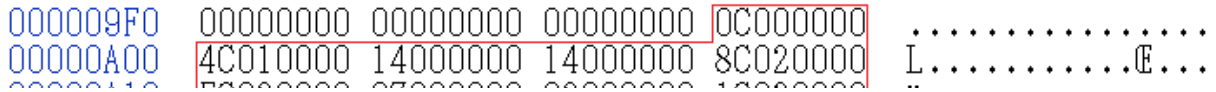
Evaluating Key Frames

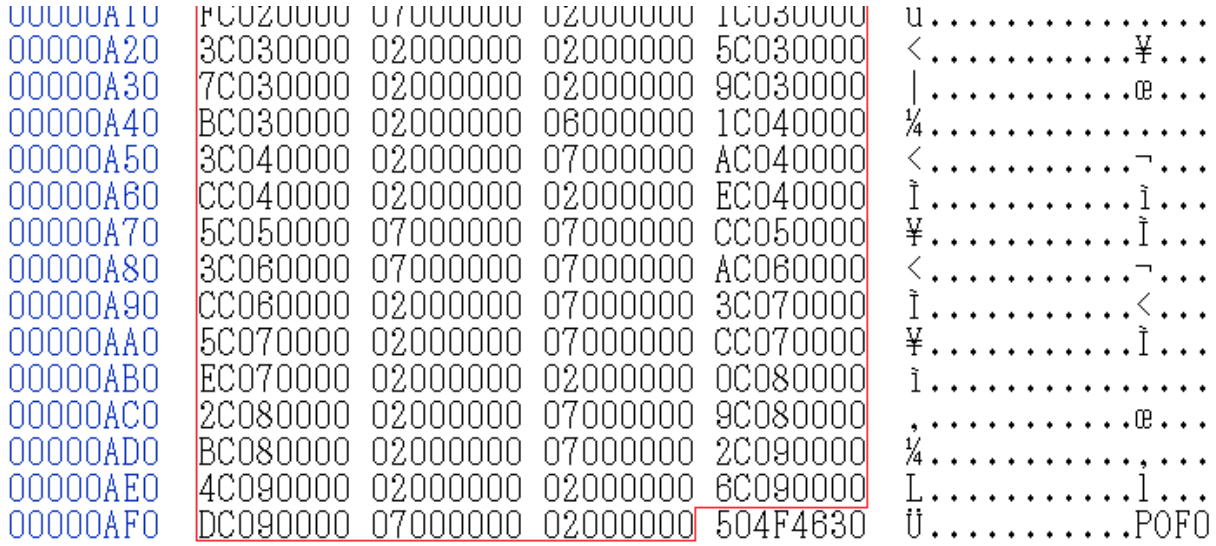
By default, PSO version uses the two Key Framed structures NJS_MKEY_F for translation data, and NJS_MKEY_A for the rotation data. Each of these is used similar to the bone definition in the node type for the Ninja Chunk Model definition. The translation data is multiplied by the rotation data to create a transformation matrix which is multilied by the base value of each group of vertices defined by the node it's modifying.

In more practical terms, this means that the the bone position is defined by the NJS_MKEY_F, and rotation depends on the format or library being ported to. Most libraries evaluated rotation as an isolated quaternion value. This means that rotation will often be managed independent of the position. So the transformation matrix will may be required to be decomed into scale, rotation, and position to keep other values from interfering with the quaternion values.

Ninja Motion List

The sample motion list for a Ninja Motion file is displayed in hex in the figure below.





Above pictured is the Motion List for the Candine's enemy's change animation. The Candine enemy model has 16 bones defined. The struct definition for each NJS_MDATA is as defined below.

```
typedef struct {
    void *posList;
    void *rotList;
    unsigned int nbPosFrames;
    unsigned int nbRotFrames;
} NJS_MDATA2;
```

The size of NJS_MDATA2 is posList(4 bytes), rotList(4 bytes), nbPosFframes(4 bytes), nbRotFrames(4 bytes), or 16 bytes for each entry in the Motion Data list. We can and need to confirm that the motion list has the same number of nodes as in the model it's defined. We can do this by taking the length of the file defined in the interchange file format (0x0AFC) and subtract it by pointer to mdata list defined by *mdata in the NJS_MOTION strucutre (0x09FC) to get 0x100. From there we can divide by the size of the NJS_MDATA2 struct (0x10) to get the number of nodes, 0x010. For for the Candine, we have 16 nodes, and 16 entries, so this is correct. This process can be used for error checking to make sure incompatible animations are not being applied to the wrong model.

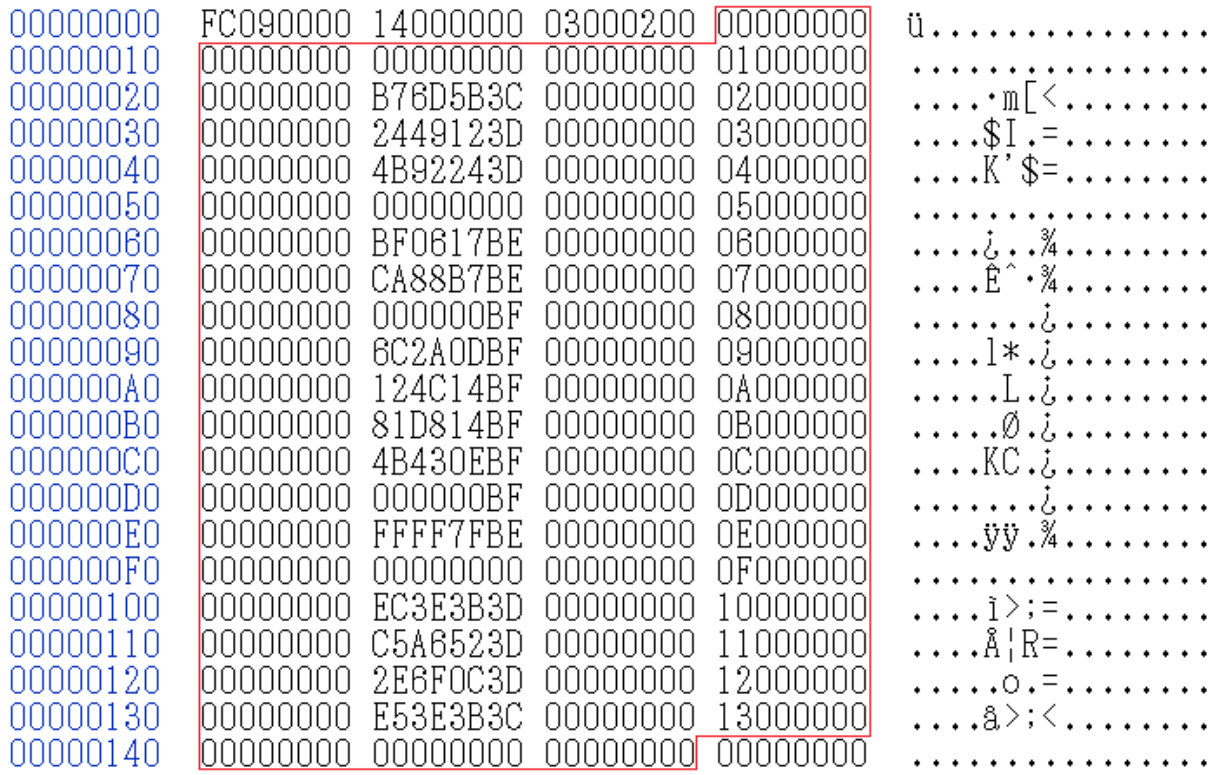
For each bone there is a pointer to a list of position key frames, a pointer to a list of rotation key frames, an integer specifiying the number of keyframes defining position, and an integer defining the number of rotation key frames. The maximum number of frames frames is defined in the NJS_MOTION header. So if the number of frames is defined as 0x14 in this case, the list will contained values from 0x00 to 0x13 for each frame defined in the animation. In other cases the are no frames defined for a bone. In these cases both the pointer and number of frames is defined as 0 and these values are skipped.

When the number of frames defined is less that the number of key frames, then the key frame numbers will be declared next to their values, with some frames being skipped. In these cases the programmer will need to use linear interpolation to smooth the transformation matrices for each frame to produce the correct animation. Though for most cases, these equations are done by either the library or file type, and keyframes can simply be ported as is.

Ninja Motion Key Framed Values (Position)

The Motion List points to two kinds of Key Frames values, float values for position, and

integer values for rotation. The figure below shows position values as shown in a hex editor.



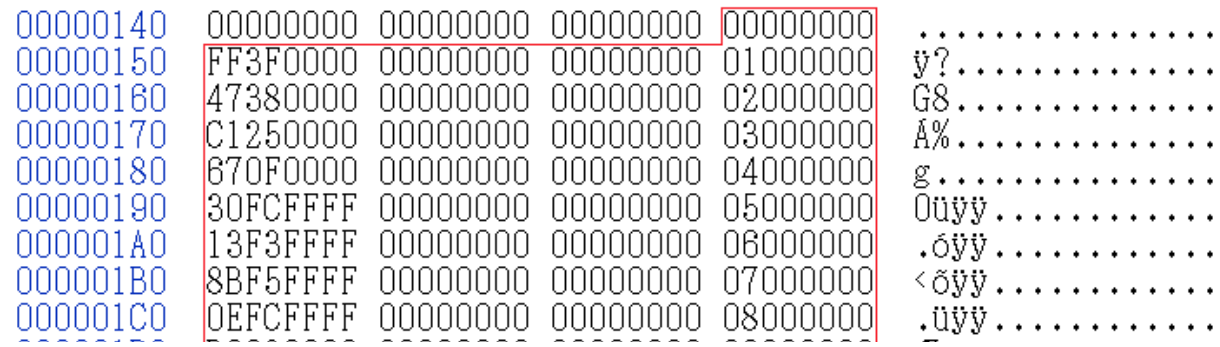
The first dword for each entry is the keyframed number. In this figure we can see on the right side, each keyframe is clearly labeled from 0x00 - 0x13. We can tell the other values are floats as there's no clear indication they are any other values. The structure for each key framed value is as defined below.

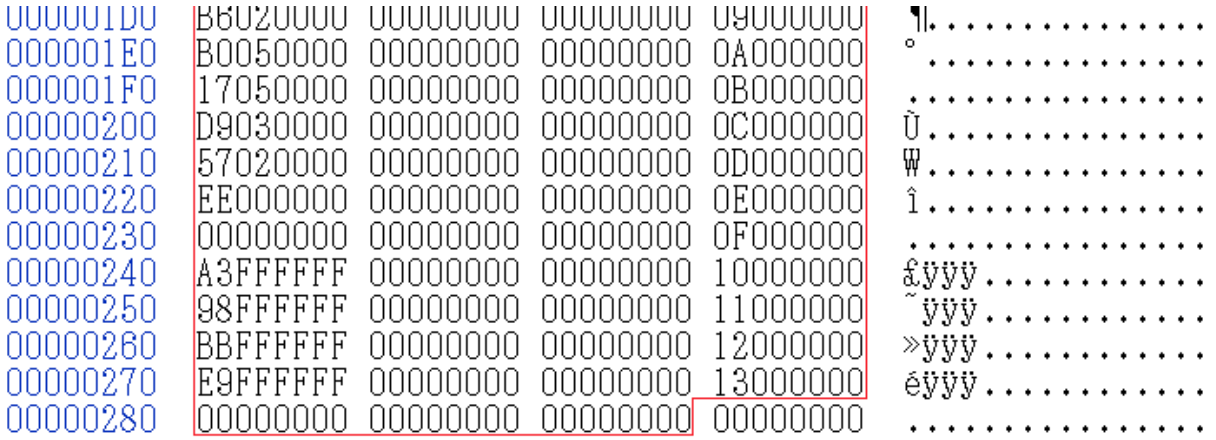
```
typedef struct {
        unsigned int keyframe; /* Keyframe number*/
        float key[3];          /* Float type key value (array 3)*/
} NJS_MKEY_F;
```

In practice, the position, or translational values are multiplied by each vertex to give it its location for the key framed animation. As each node is drawn at the origin and then multiplied by these values, the (x, y, z) float for each key framed value can be refered to as the bone's (x, y, z) value for that given frame.

Ninja Motion Key Framed Values (Rotation)

The Motion List points to two kinds of Key Frames values, float values for position, and integer values for rotation. The figure below shows rotation values as shown in a hex editor.





Rotation data in the Ninja Graphics API has one defining feature. As the values are 0x000 for 0 degrees and 0xFFFF for 360 degrees, the value appears as a short for most values. Such as the first value 0x3FFF, we can see that is is followed by four zeros. On the referse side, because the value is signed, it can go the other way to indicate nagetive values. However, negative values share the similar attribute of being followed by 0xFFFF, as 360 degrees does not go beyond the first two bytes as seen in the dword at offset 0x0190.

```
typedef struct {
        unsigned int keyframe; /* Keyframe number*/
        int key[3];            /* Angle type key value (array 3)*/
} NJS_MKEY_A;
```

Similar to the position based key framed values, we can see that the first dword for each entry has the key framed number. In this case numbered from 0x00 to 0x13 on the right hand side. Following the key framed value is an angle based int indicating the rotation for a given axis (x, y, z). In practical application, these values are simply mutliplied by every vertex in the node's defined vertex list to produce animation, however other libraries and file formats vary in their rotation implementation, and may require aditional equations to produce the original effect of the defined animation.

Ninja Motion Source Code

The code provided below provides a sample implementation for parsing the values of a Ninja Motion file.

```
NinjaChunkModel.prototype.readAnimFile = function(filename){

        //Set filepointer reference to filename
        var fp = new FilePointer(filename);
        //Read interchange file format
        var iff = bs.read_iff();

        //Stop if interchange file format header
        if(iff != 'NMDM'){
                return null;
        }

        //Read total length of animation file
        var len = fp.readUInt();
        fp.trim();

        //Read offsrt to motion list
        var motionOfs = bs.readUInt();
        //Read Number of frames in the animation
        var nbFrames = bs.readUInt() - 1;
        //Calculate the number of bones defined in the animation
        var nbBones = (len - motionOfs)/16;
```

```
        //Make sure number of bones matches model
        if(nbBones != this.boneList.length){
                return null;
        }

        //Seek to start of motion list
        fp.seek_set(motionOfs);

        //Parse Motion List
        var motionList = [];
        for(let i = 0; i < nbBones; i++){
                motionList.push({
                        'posOfs' : bs.readUInt(),
                        'rotOfs' : bs.readUInt(),
                        'nbPosFrame' : bs.readUInt(),
                        'nbRotFrame' : bs.readUInt()
                });
        }
        //Variable to hold animation data
        var animData = [];

        //Iterate over every entry in the data
        for(let i = 0; i < motionList.length; i++){
                var motion = motionList[i];

                //Set animation for current bone
                animData[i] = [];

                //Seek and parse positions
                 if(motion['nbPosFrame']){
                        fp.seek_set(motion['posOfs']);
                        for(let k = 0; k < motion['nbPosFrame']; k++){
                                var time = fp.readUInt();
                                animData[i][time] = animData[i][time] || {};
                                animData[i][time]['pos'] = fp.readVec3();
                        }
                }

                //Seek and parse rotations
                if(motion['nbRotFrame']){
                        fp.seek_set(motion['posOfs']);
                        for(let k = 0; k < motion['nbPosFrame']; k++){
                                var time = fp.readUInt();
                                animData[i][time] = animData[i][time] || {};
                                animData[i][time]['rot'] = fp.readRot3();
                        }
                }
        }

        //Add animation data to animation list
        this.animList.push(animData);
}
```

Ninja Scenery Format

Sonic Team developed their own stage format for stages. The file format is defined as rel. It can be hard to distinguish which files are actually maps because there doesn't seem to be consistent file extension, as rel extensions are used for several different file type. For the purpose of this documentation, we will refer to Stage Files as the "Ninja Scenery Format", which corresponds to n.rel and d.rel files in the SCENE folder of the data.

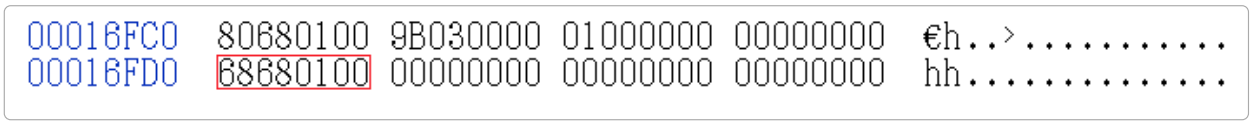
The format itself is actually decpetively simple. The model data is represented with internal Ninja Chunk Models inside the file. This documentation will use following class to define how to parse the code in the file.

```
function NinjaSceneFormat(filename){
        this.fp = new FilePointer(filename);
```

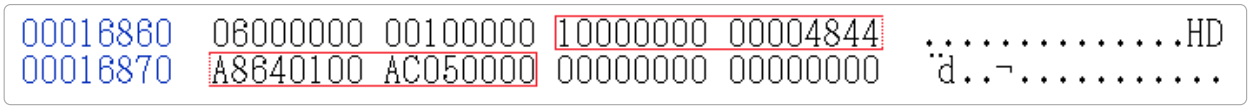
```
        this.vertexStack = [];
        this.vertexMap = [];

        this.materials = [];
        this.triangleFaces = [];

        this.texList = [];
}
```

Ninja Scenery Format Header

The Ninja Scenery file format is parsed from the bottom of the file to the top. The pointer to the header is located at EOF (end of file), minus 16 bytes.



The header is location at the position defined by the pointer at the end of the file. The figure below depicts the header as seen in a hex editor.



The head itself is four dwords. The first dword gives the number of sections in the file. The functionality of the second dword is unknown. In most cases it seems to be a constant 0x00HD, but the significance of this is unknown. The third dword is a pointer to the section list. The last dword is a pointer to the texture list. The structure for the Ninja Scenery Format header is given by the code snippet below.

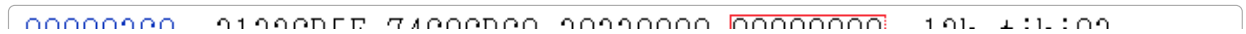
```
typedef struct {
        unsigned int nbSections;
        unsigned int typeDef;
        void* sectionList;
        void* texAddr;
} NJS_SCENE_HEADER;
```

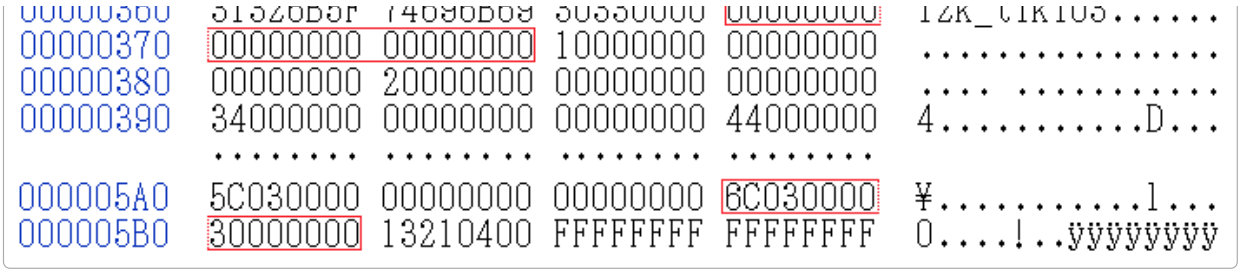
The Ninja Scenery Format uses sections to define an area of a map. A section is a group of models which can be loaded and unloaded in the scene depending on draw distance. Each scene has a list of animated and static models which make up that section. Each section has a center, Y-rotation and translation offset. Where each scene is drawn at the orgin, rotated in place in the left or right direction and then translated to its location on the map.

The texture list defined in the stages uses the same structures as defined in the Ninja Chunk Model, Ninja Texture list definition. The texAddr pointer defined in the header points to the NJS_TEXLIST struct which points to the texture list, and defines how many items are in that list. Each item in the list is a struct of NJS_TEXNAME, which contains a void pointer to each texture's filename.

Ninja Scenery Format Texture List

Sections will be covered on the next page. This documentation will first address the texAddr value. The texAddr is a pointer which points to the texture list. The texture list shares the same definition as the Ninja Texture List defined in the Ninja Chunk Model section of this document. The purpose is to define of list of textures used by models inside the scene.





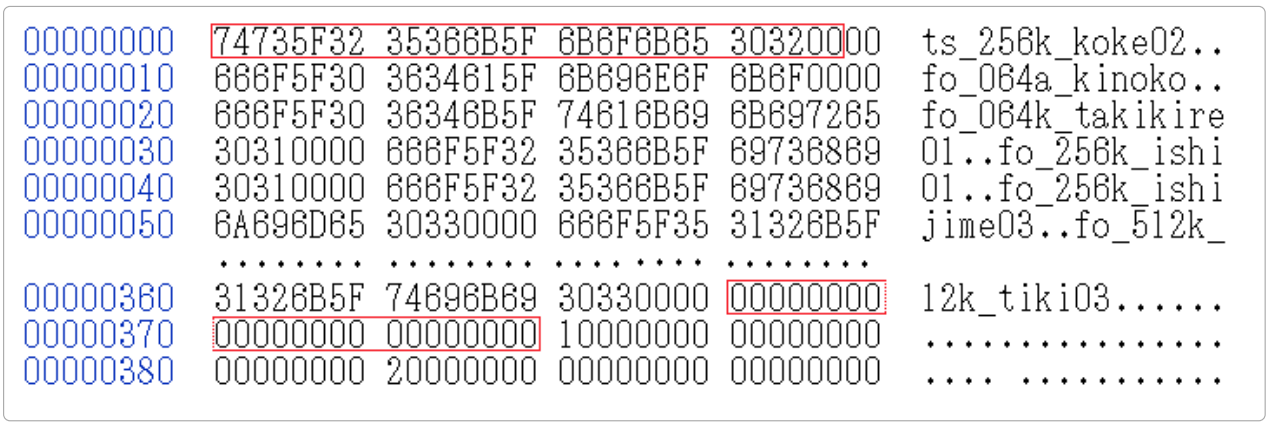
The figure above deptice the texture list. The structures are identical to the Nija Texture List declared earlier in this document. The two dwords at 0x05AC define the texture list for pointer to the begining of the list and the number of elements in the list as defined below.

```
typedef struct {
        NJS_TEXNAME *textures;
        unsigned int nbTexture;
} NJS_TEXLIST;
```

In the struct defined above, textures points to the begining of the texture list. In this case 0x036C. The next value is the number of textures defined for the scene models, in this case 0x30.

```
typedef struct {
        void *filename;         /* Texture file name      */
        unsigned int attr;      /* Texture attributes     */
        void *texaddr;          /* Texture memory address */
} NJS_TEXNAME;
```

Each struct in the texture list is defined as type NJS_TEXNAME. Each texture name is defined as a void pointer to the first character of the filename's String. The second dword defines attributes, such as clamp, flip or otherwise. And then third dword defines the textures location in memory. Both of these values are set into mememory during runtime, and can be ignored when parsing the file.

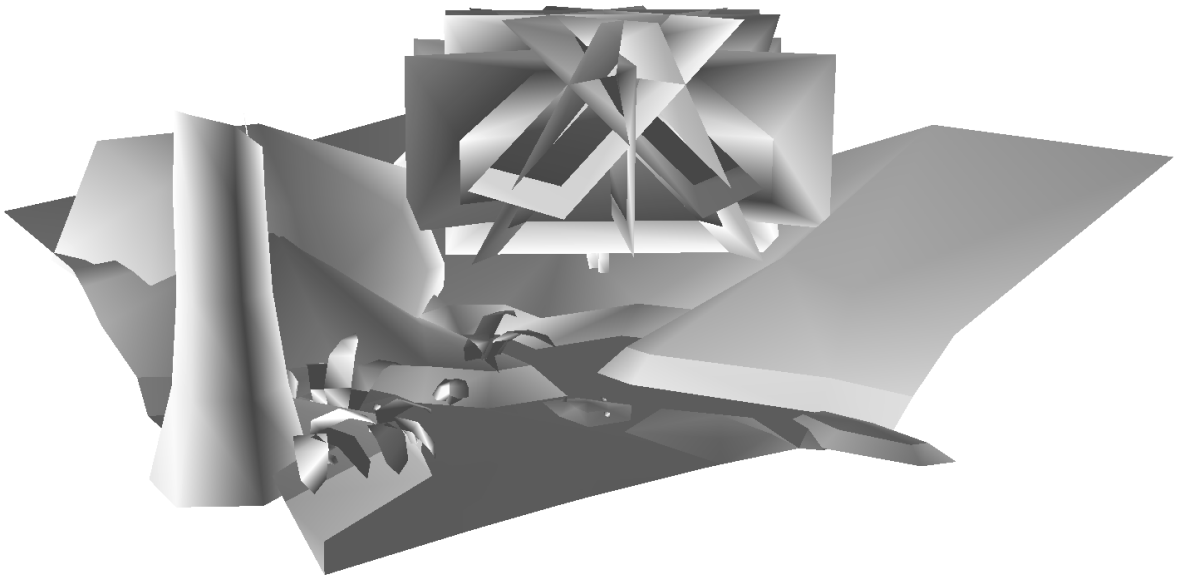


Each filename is a standard C formated string, where the end of the string is signified by a 0x00 null character. While it is possible to parse this section by reading from the start pointer of one filename until the start of the following entry start point, it is simply easier to read the from the starting point until the terminating character.

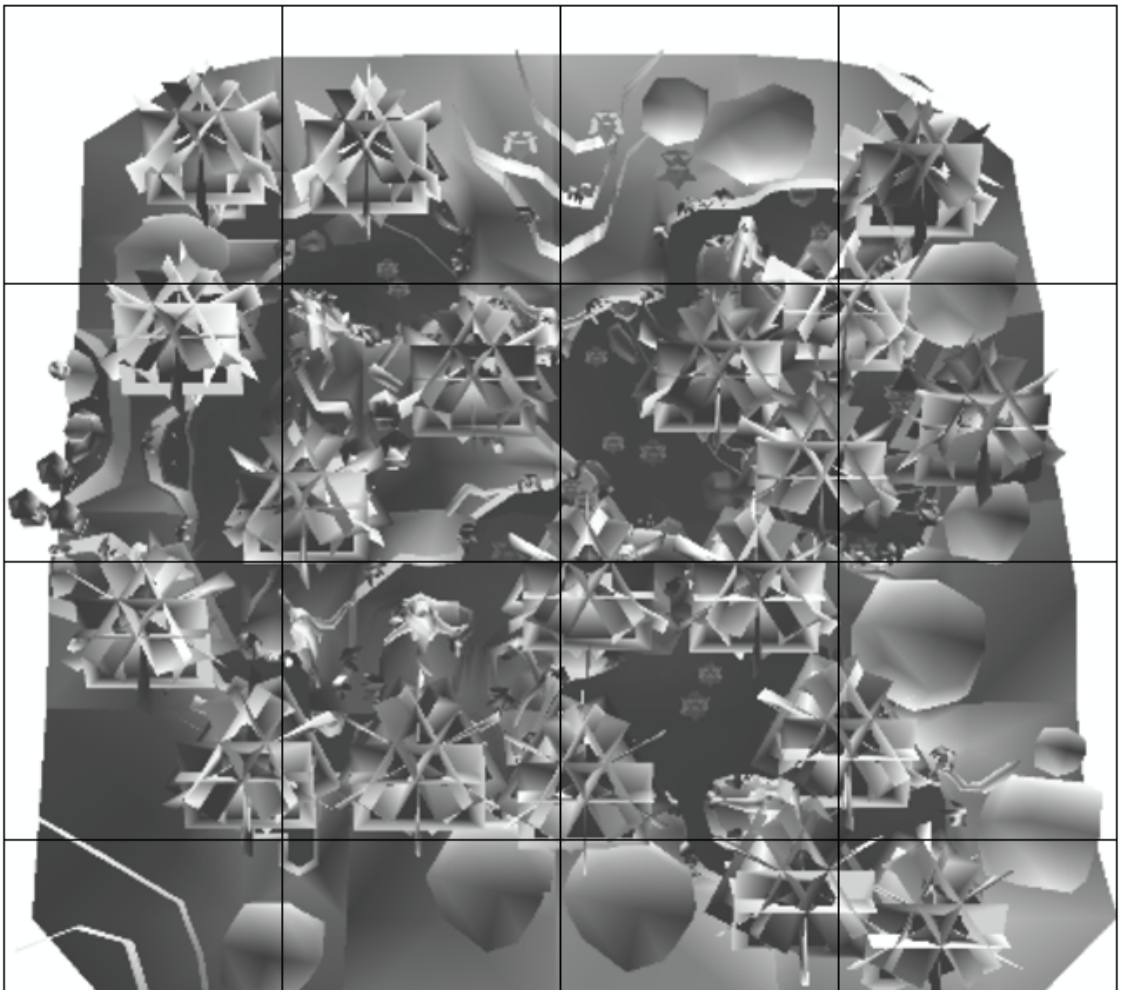
Ninja Scenery Format Section List

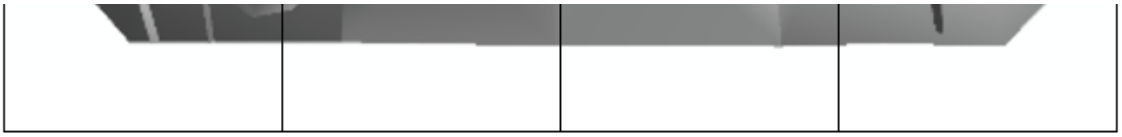
The Ninja Scenery Format defines a section list which contain pointers to all of the model which are used to draw the stage. Sections themselves have a Y-rotation and translation. Each section has a list of models that are drawn at the origin, rotated left or

right and then translated to their defined location inside the map.



This figure demonstrates one of the stage sections rendered by itself. The section contains many different models which are defined within the section.

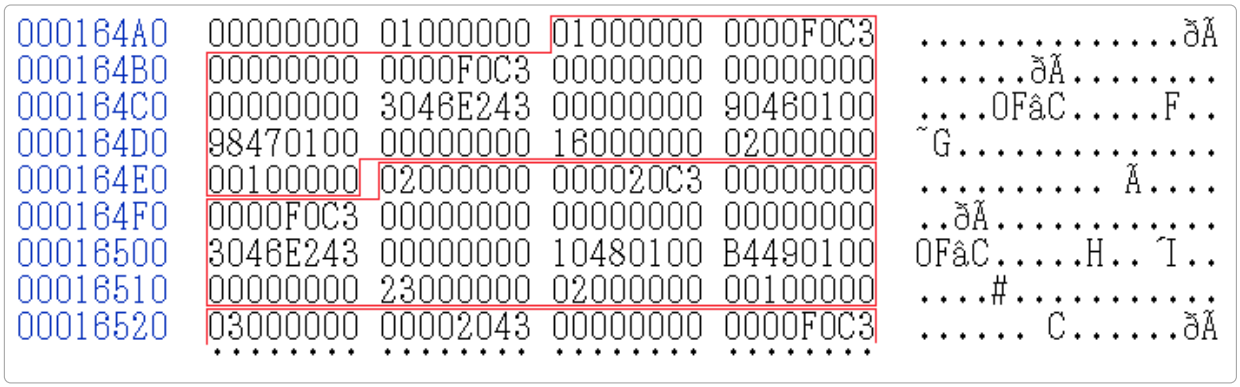




The figure displays the relationship for these sections for the forest 01 stage. In the forest 01 stage there are 16 sections. Each section represents a cross section of the map as shown in the grid above. The solo defined section is the shown from the top, in the first row from the top, and the first cell from the left.

Ninja Scenery Format Section List

The figure below shows what the section list of a stage file looks like when viewed in a hex editor.



The example being used is a stage with 16 sections. The dots at the bottom indicates that the list continues. Only the first two entries are seen in their entirety. The first dword in each section is its section number. This can be seen in the image above with 0x01, 0x02, and 0x03 being the first values to appear in their respective sections. The structure definition for the Ninja Scenery Format section is shown in the code snippet below.

```
typedef struct {
        unsigned int sectionId;
        float pos[3];
        int rot[3];
        float radius;
        void* staticModelList;
        void* attributeList;
        void* animModelList;
        unsigned int nbStaticModel;
        unsigned int nbAttributes;
        unsigned int nbAnimModels;
        unsigned int endSignal;
} NJS_SECENE_SECTION;
```

The first dword is an int which defines the section number. This number is an easy indicator to locate the start of each section structure when debugging this file format. The next three dwords are float values for the translation of the section. For the first section the values are 0xC3F00000, 0x00000000, 0xC3F00000, or -480.0, 0.0, -480.0. So the section will be drawn at the origin and then moved in to the global position with respesct to the origin. The next three dwords are signed integer values which represent the rotation of the section. In practical application, only the Y-direction is set to turn the section to the left or right to adjust it's orientation before translation is applied.

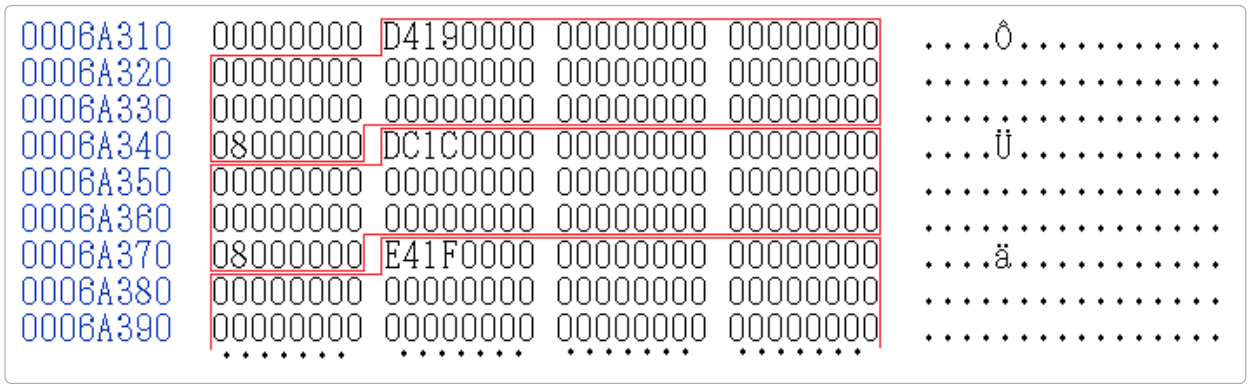
The next three values are pointers to arrays with the respective length of each array listed afterwards. The staicModelList is a pointer to the list of static models found in the

array. The nbStaticModel attribute defines the number of of static models in that list. The same thing occurs for the attriubuteList and animModelList. The contents of the attributeList is currently unknown. It is assumed to be flags for stage related attributes such as footsteps, collision detection or other possible flags. These attributes have no impact on the 3d structure of the stage, and this documentation makes no attempt to describe this section. The staticModelList and animModelList provide similar function. Each is a list which contains a pointer to internal Ninja Chunk Model structures. The static list is only non-moving model and the animated models also include a pointer into an internal Ninja Motion structure to define the movement of the model to be animated. In the case in which a section has either no static models or no animated models, the number of and pointer value will be defined as zero.

On the end of each model is a constant value of 0x1000. It is unknown as to what purpose this value holds and as it does not have any apparent effect of the 3d structure of the Ninja Scenery Format, this documentation makes no attempt to describe its function.

Ninja Scenery Format Static Model List

The list for static models as seen in a hex editor is shown in the figure below.

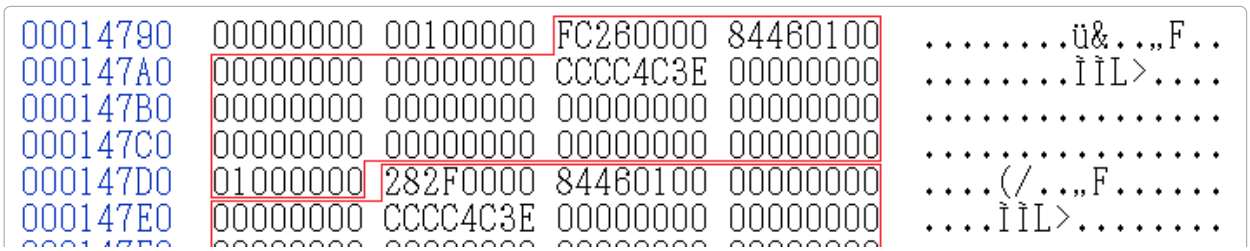


The first three entries of a static model list are displayed above. Each list entry has a length of 0x30 bytes. The first dword is the pointer to the Ninja Chunk Model definition for the model. Following that is 10 dwords whose usage is completely unknown. The last dword in each static model entry is 0x08, however the significance of this is also unkown. As these unkown values have no effect on the structure of the stage, this document will not make any attempt at describing their functionality. The code snippet below dfefines the struct used in this array.

```
typedef struct {
        void* modelAddr;
        unsigned int unknown[10];
        unsigned int endSignal;
} NJS_STATIC_MODEL_OBJECT;
```

Ninja Scenery Format Animated Model List

The list for animated models as seen in a hex editor is shown in the figure below.



```
00014790  00000000 00000000 00000000 00000000  ................
00014800  00000000 00000000 00000000 01000000  ................
```

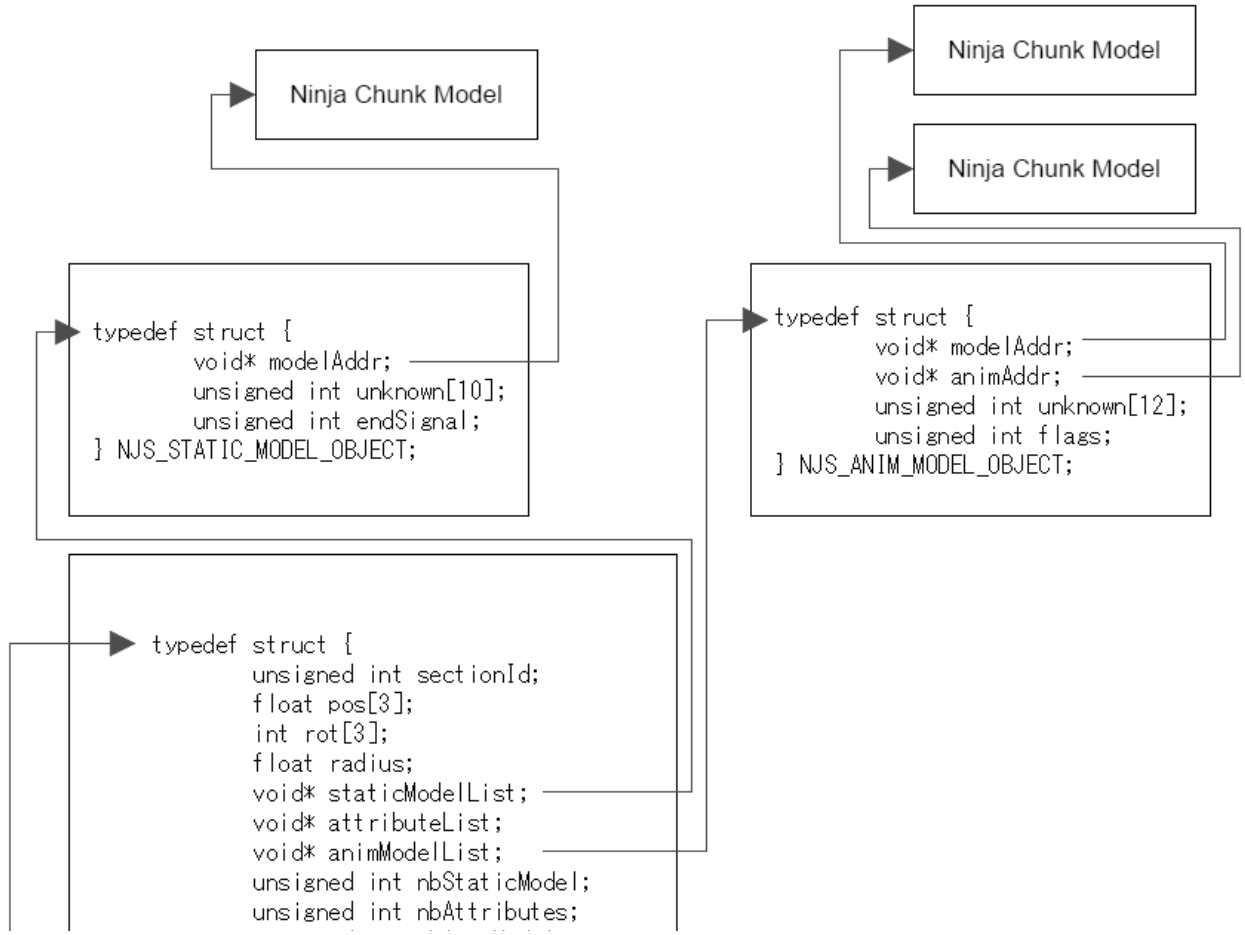
The figure above shows two entries in an animation model list. The first dword is a pointer to the Ninja Chunk Model definition for the model. The second dword is a pointer to the Ninja Motion structure which defines the animation for the model being defined. The functionality of the remaining thirteen dwords is unknown, and as it seems has no apparent effect on the structure of the 3d model, this document makes no attempt to describe their functionality. Though it should be noted that the data does seem contain some float data, and some sort of possible attribute or flag on the end of each entry.

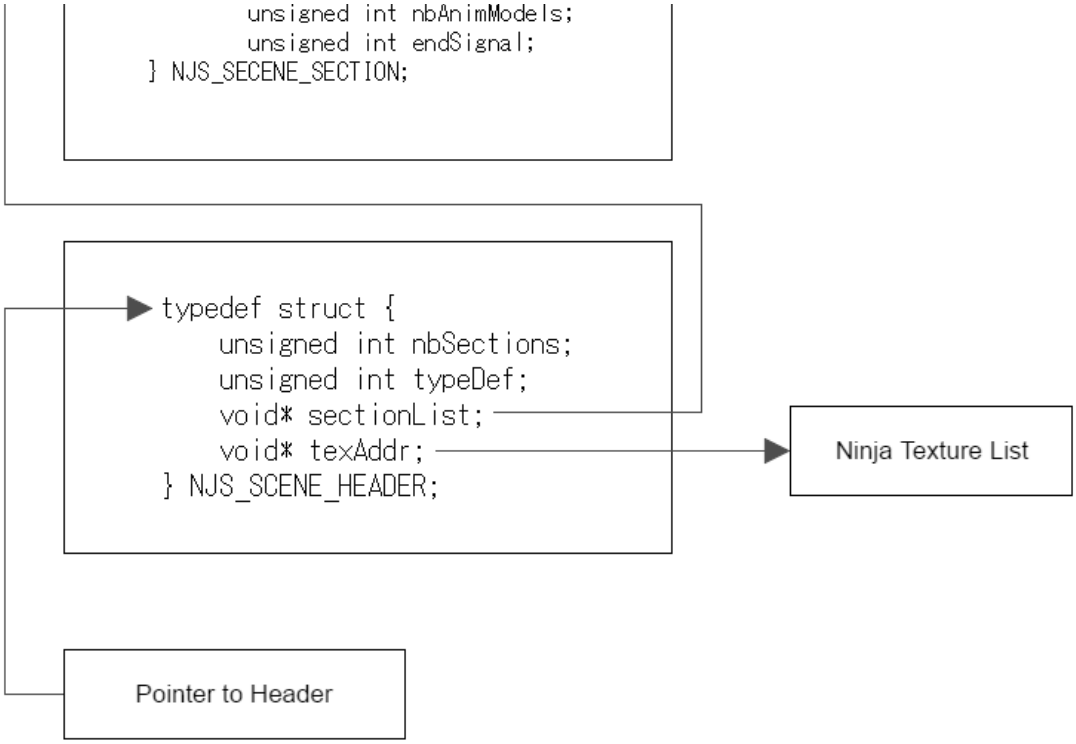
```
typedef struct {
        void* modelAddr;
        void* animAddr;
        unsigned int unknown[12];
        unsigned int flags;
} NJS_ANIM_MODEL_OBJECT;
```

Each pointer to the Model and Animation correspond to the Ninja Chunk Model definition and Ninja Motion definitions described in this document. As there are no changes to their structure, this document will not cover that content again in this section.

Ninja Scenery Format Structure Summary

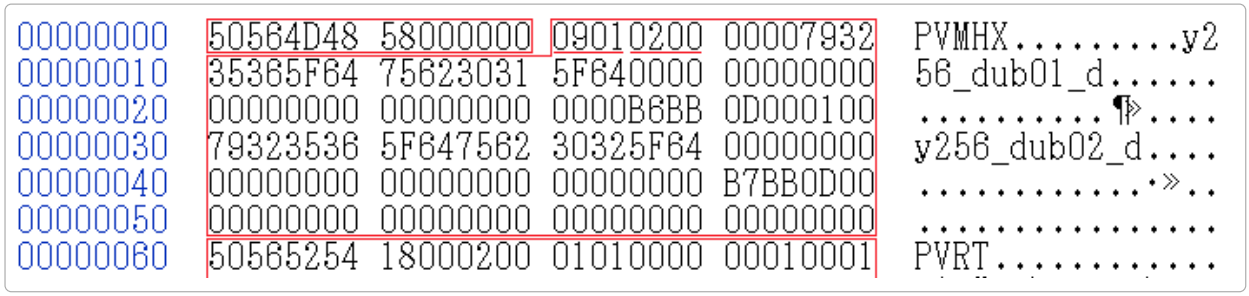
The figure below outlines the over all file structure of the Ninja Scenery File Format. A pointer at the end of the file points to the header. The header points to the texture list and a list of sections. Each section points to a list of model pointers for static and animated models. Each model is defined by a pointer to a Ninja Chunk Model structure.





Power VR Archive .PVM

PowerVR is the name of the graphics card computer used in the Sega Dreamcast. The texture formats capable with the Dreamcast are proccessed through the graphics card natively. PVM is the extension for archiving multiple PVR textures into one file. Each PVM file has a header which defines the content of the file and an offset to the first texture. The second dword in every texture defines the file length and each file starts with the interchange file format header, 'PVRT'. Files in the archive are not compressed, and the only type of file in the archive is PVR textures. The header for a PVM file is shown in the figure below.

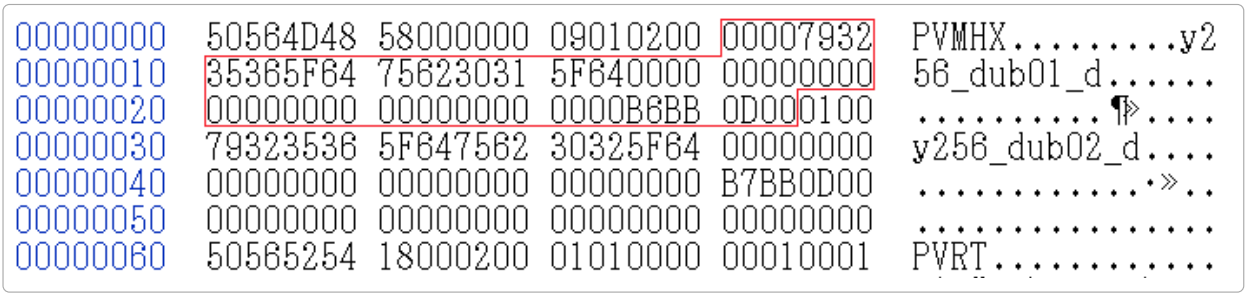


The figure above shows the header of a PVM archive as seen in an Hex editor. The interchange file format is defined by the four characters 'PVMH', followed by the seek offset to the fist texture (shown by PVRT in the image). The content of the header is bound by a rectangular outline. The format for the content of the header is not constant and is defined by reading flags in the header.

The first word in the PVM header defines the flags for the format of each entry in the archive. The second word is the number of images in the archive. The flags are defined by the code snippet below.

```
#define PVR_GLOBAL_INDEX        BIT_0 //Include global index (4 bytes)
#define PVR_TEXTURE_DIMENTSIONS BIT_1 //Defines texture dimensions
#define PVR_PIXEL_DATA_FORMAT   BIT_2 //Defines pixel and data formats
#define PVR_FILENAMES           BIT_3 //Filenames are included
```

In this case, the flags value is 0x0109, which means BIT_0 and BIT_3 are set to include the global index and filename.



A single PVM header entry is highlighted in the figure above. The format of the header is given in the table below.

Datatype	Description	Bit Flag
unsigned short	Entry Number in the archive	
char[0x1C]	Filename for the texture entry	BIT_3
unsigned short	Texture's pixel format followed by data format	BIT_2
unsigned short	Texture's width (bits 0-3) and height (bits 4-7)	BIT_1
unsigned int	Global index (unique number to indentify texture)	BIT_0

As a general pattern, the width, height, pixel format and data format for each PVR image are defined inside the PVR image itself. Therefor, it is very rare for these two flags to be set in the header. Most often the texture will contain the filename and the global index.

Power VR Archive .PVM Source

The following is sample source code for parsing the header of a PVM archive and extracting the contents.

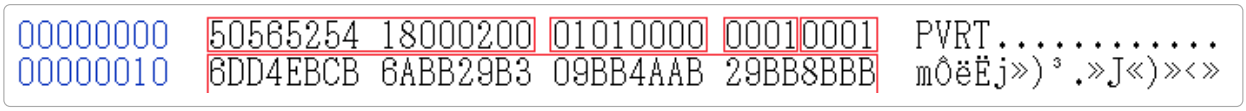
```
"use strict";

var FilePointer = require('filepointer');

function extractPVMH(filename) {

	//Filepointer to .pvm data
	var fp = new FilePointer(filename);

	//Check first four bytes for header
	if(fp.read_iff() != 'PVMH'){
		throw new Error('Incorrect file type provided');
	}

	//Read length of header section
	var iffLen = fp.readUInt();

	//Read flags and number of textures
	var flags = fp.readUShort();
	var nbTextures = fp.readUShort();

	//Create array to hold data
	var pvmTextures = [];

	//Iterate over file head
	for(let i = 0; i < nbTextures; i++) {

		pvmTextures.push({

			//Read texture id from archive
```

```
                        'id' : fp.readUShort(),

                        //Read the filename if BIT_3 is set
                        'filename' : (flags & BIT_3) ? fp.readString(0x1C) : null,

                        //Read format is BIT_2 is set
                        'format' : (flags & BIT_2) ? fp.readUShort() : null,

                        //Read dimensions if BIT_1 is set
                        'dimensions' : (flags & BIT_1) ? fp.readUShort() : null,

                        //Read global index if BIT_0 is set
                        'gbIndex' : (flags & BIT_0) ? fp.readUInt() : null

                });
        }

        //Iterate over file to copy texture data
        for(let i = 0; i < nbTextures; i++){

                //Seek to next instance of 'PVRT'
                if(!fp.find('PVRT')){
                        throw new Error('Inconsistent data for PVM header');
                }

                //Read header and file length
                var pvr = fp.read_iff();
                var pvrLen = fp.readUInt() + 0x08;

                //Seek back to start of PVRT definition
                fp.seek_cur(-8);

                //Copy content of data to array
                pvmTextures[i].data = fp.copy(pvrLen);

        }

        //Return data extracted from archive
        return pvmTextures;
}
```

Power VR Textures

PowerVR is the name of the graphics card used in the Dreamcast. The file formats for the textures used in the Dreamcast were designed to be processed by the graphics card to avoid wasting CPU cycles on decoding textures. Thus, textures are referrder to 'PVR' textures and have a .PVR file extension.

PVR textures are defined by several attributes, width, height, data format, pixel format, global index, and filename. Width and height refer to the dimensions in pixels of the texture for width and height. Data format refers to the encoding method of the texture. PVR has several methods of encoding data which call all be decoded by the graphics card, these will be covered in depth extensively in this section. The pixel format refers to the scheme in which color is stored for each pixel. The global index is an unsigned int (4 byte) value assigned to every unique texture for identification. This value helps the Dreamcast determine if a texture is already loaded into video memory to avoid loading it more than once. Lastly the filename, which is not always defined depending on the archive or circumstances and may require a naming scheme, or searching through the game's executable data for string values which may have the correct texture names.



The image above shows the header of a PVR file as seen in a hex editor. The first dword is the interchange file format header 'PVRT', the second dword is the length of the

texture in bytes. The next dword contains two flags. The first byte is the pixel format, with the following byte being the data format. The upper two bytes are not used. The next word value is with width, followed by the next word value being the height (in pixels). The header takes up the first 0x10 bytes of the file. Everything following is the body of the data which is used to define the image. This is the data that must be decoded.

```
#define KM_TEXTURE_ARGB1555 (0x00000000)
#define KM_TEXTURE_RGB565   (0x00000001)
#define KM_TEXTURE_ARGB4444 (0x00000002)
#define KM_TEXTURE_YUV422   (0x00000003)
#define KM_TEXTURE_BUMP     (0x00000004)
```

Pixel formats are defined by the code snippet above. Each pixel color data is defined by a 16 bit format. Each format is a tradeoff between alpha and color depth. the format ARGB_1555 defines one bit for alpha. Meaning the pixel is visible when on and invisible entirley when off. RGB_565 forgoes an alpha bit entirely in the interest of increased color depth. Textures which use this pixel format often attain an alpha effect by defining black areas in the texture and making them transparent with alpha blending. ARGB_4444 uses only 4 bits for depth color but allows for more definition with translucent colors. YUV422 and Bump are not covered in this document.

```
#define KM_TEXTURE_TWIDDLED           (0x00000100)
#define KM_TEXTURE_TWIDDLED_MM        (0x00000200)
#define KM_TEXTURE_TWIDDLED_RECTANGLE (0x00000D00)
#define KM_TEXTURE_VQ                 (0x00000300)
#define KM_TEXTURE_VQ_MM              (0x00000400)
#define KM_TEXTURE_SMALLVQ            (0x00000F00)
#define KM_TEXTURE_SMALLVQ_MM         (0x00001000)
#define KM_TEXTURE_PALETTIZE4         (0x00000500)
#define KM_TEXTURE_PALETTIZE4_MM      (0x00000600)
#define KM_TEXTURE_PALETTIZE8         (0x00000700)
#define KM_TEXTURE_PALETTIZE8_MM      (0x00000800)
#define KM_TEXTURE_RECTANGLE          (0x00000900)
#define KM_TEXTURE_STRIDE             (0x00000B00)
#define KM_TEXTURE_TWIDDLED_MM_ALT    (0x00001200)
```

The code snippet above defines the different data types found in PVR files. Each file format on its own is relatively simple. What complicates matters is that all of the the types are minor variations on three main flags, "twiddled", "vector quantized", and "mipmaps". Twiddled is a data order which can be processed by the graphics processor in parallel improving performance for loading textures. Vector quantized is a compression which uses a 'Codebook' to define pixel information and the actual data refers back to entries in this codebook to define the image data. Mipmaps refers to a technique to minimize wasted rendering. Textures further away from the camera in a scene do not need as much detail as textures close to the camera. Mipmaps are a method of changing the size of a given texture for a mesh depending on its distance in the scene. In addition to these flags there are also external pallets, an in-order rectandgle format, and a stride format.

Power VR Textures Sample Code

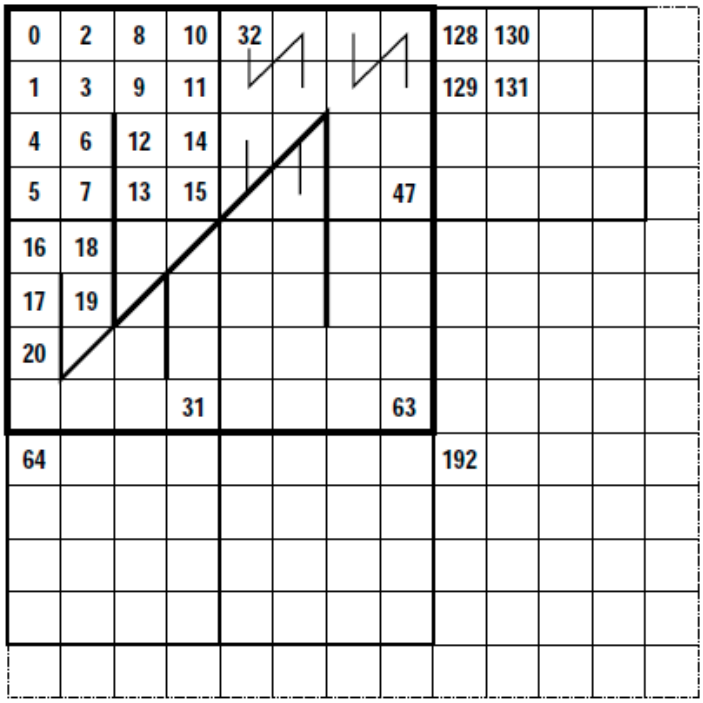
The following is sample code for parsing the header of a PowerVR texture, to extract flags which will be used for converting the texture to a bitmap.

```
"use strict";

var FilePointer = require("filepointer");

//Pixel Type definitions
```

```
const PIXEL_ARGB1555         = 0x00;
const PIXEL_RGB565           = 0x01;
const PIXEL_ARGB_4444        = 0x02;
const PIXEL_YUV422           = 0x03;
const PIXEL_BUMP             = 0x04;

//Data Type definitions
const DATA_TWIDDLED          = 0x01;
const DATA_TIDDLED_MM        = 0x02;
const DATA_TIDDLED_MM_ALT    = 0x12;
const DATA_TWIDDLED_RECT     = 0x0D;
const DATA_VQ                = 0x03;
const DATA_VQ_MM             = 0x04;
const DATA_SMALL_VQ          = 0x0F;
const DATA_SMALL_VQ_MM       = 0x10;
const DATA_PALETTIZE4        = 0x05;
const DATA_PALETTIZE4_MM     = 0x06;
const DATA_PALETTIZE8        = 0x07;
const DATA_PALETTIZE8_MM     = 0x08;
const DATA_RECTANGLE         = 0x09;
const DATA_STRIDE            = 0x0B;

const VECTORQ  = [ DATA_VQ, DATA_VQ_MM, DATA_SMALL_VQ, DATA_SMALL_VQ_MM ];
const TWIDDLED = [ DATA_TWIDDLED, DATA_TIDDLED_MM, DATA_TIDDLED_MM_ALT, DATA_TWIDDLED_RECT ];
const MIPMAP   = [ DATA_TIDDLED_MM, DATA_TIDDLED_MM_ALT, DATA_VQ_MM, DATA_SMALL_VQ_MM, DATA_PALETTIZE8_MM ];
const PALLET   = [ DATA_PALETTIZE4, DATA_PALETTIZE4_MM, DATA_PALETTIZE8, DATA_PALETTIZE8_MM ];

function pvrToBitmap(filename) {

        //File pointer to data
        var fp = new FilePointer();

        //Check interchange file format type and length
        if(fp.read_iff() != 'PVRT'){
                throw new Error('Incorrect file type supplied');
        }
        var pvrLen = fp.readUInt();

        //Read data and pixel types
        var pixelType = fp.readByte();
        var dataType  = fp.readByte();

        //Skip over unused short
        fp.seek_cur(0x02);

        //Read width and height dimensions
        var width = fp.readUShort();
        var height = fp.readUShort();

        //Check if file has mipmaps
        var isMipmap = MIPMAP.indexOf(dataType) != -1;

        var bitmap;
        //Functions to convert data to bitmap array
        if(TWIDDLED.indexOf(dataType) != -1) {
                bitmap = twiddleFormat(fp, width, height, isMipmap);
        } else if(VECTORQ.indexOf(dataType) != -1){
                bitmap = vqFormat(fp, width, height, isMipmap, dataType >= 0x0F);
        } else if(PALLET.indexOf(dataType) != -1) {
                bitmap = palletFormat(fp, width, height, isMipmap)
        } else if(dataType != DATA_RECTANGLE) {
                bitmap = rectangleFormat(fp, width, height);
        } else if(dataType != DATA_STRIDE) {
                bitmap = strideFormat(fp, width, height);
        } else {
                throw new Error("Unknown data type");
        }

        return colorFormat(bitmap, pixelType);
}
```

Power VR Textures Twiddled

Twiddled refers to the order in which most of the data inside PVR textures is stored. Rather than being in texture order, the data is organized in such a way that can be read in parallel by the graphics card to increase the performance of loading textures.



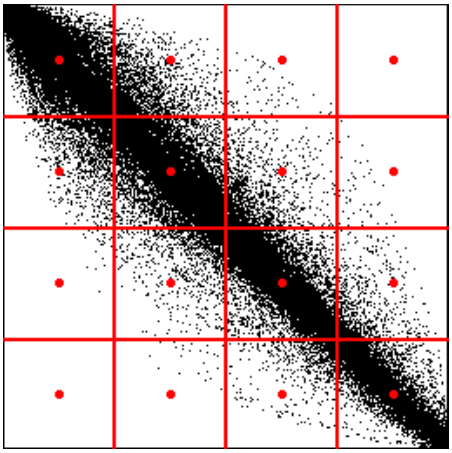
The pattern is recursive, so it can be difficult to discren the pattern from the image above. The code snippet below shows the algorythm used to iterate over a twiddled encoded texture.

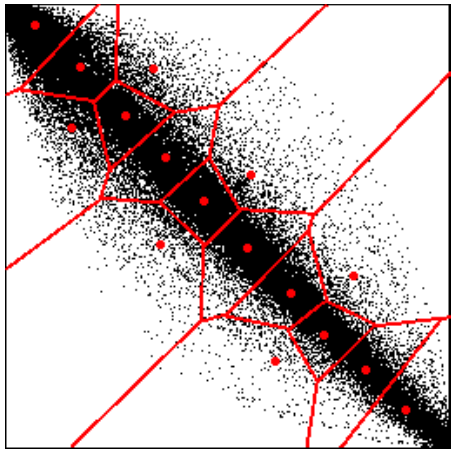
```
function square_twiddled(twiddled, imgsize){
  var ptr = -1;
  var detwiddled = [];
  subdivide_and_move(0, 0, imgsize, 0);
  return detwiddled;

  function read_pixel(){
    ptr++;
    return twiddled[ptr];
  }

  function subdivide_and_move(x1, y1, size, op) {
        if (size == 1) {
           detwiddled[y1*imgsize+x1] = read_pixel();
        } else {
                var ns = size/2;
                subdivide_and_move(x1, y1, ns);
                subdivide_and_move(x1, y1+ns, ns);
                subdivide_and_move(x1+ns, y1, ns);
                subdivide_and_move(x1+ns, y1+ns, ns);
        }
  }
}
```

The algorythm uses a recursive formula the divides the width into four sub-divisions on each call until the location is narrowed down to a single pixel. The order in which the recursive calls narrow down a pixel is the order in which the image pixels are drawn in sequentially. One thing to note is that this function assumes that the width and height are the same size. In the case that the texture is rectanglular, this function will produce out of bounds exception errors.

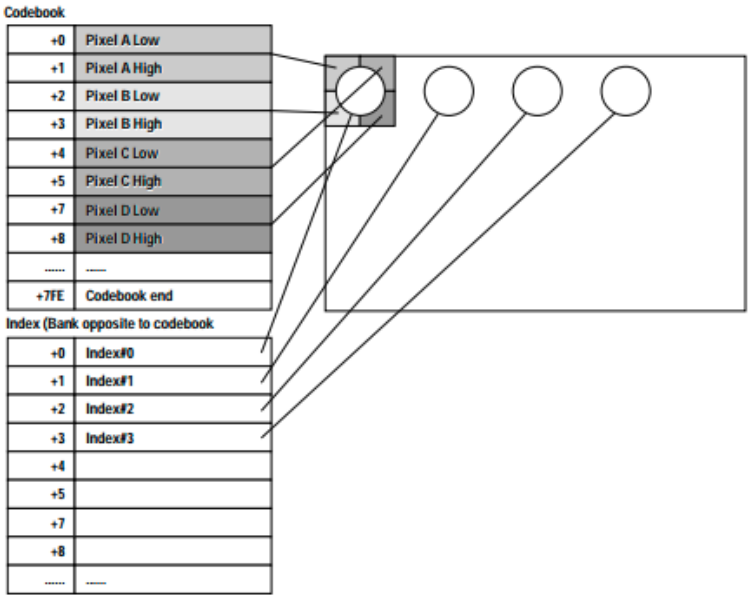
Power VR Textures Vector Quantized

Vector Quantized is a technique for compressing PVR textures. The technique works on the following concept presented in the two figures below.





The dots in these two figures represent grey scale value of pixel that occur in a monochrome digital image. If we only had a limited amount of space or bits to represent this data, we'd need to define a scheme to be able to represent different colors with less information. The left image shows that if we were to just take incremental values from 0 to 255, we wouldn't have a pallet that would accurately represent the information in the image. The right image shows what Vector Quantization does, it defines a pallet which better represents the range of vlaues in the image to be refered to. The term 'Vector Quantized' is actually somewhat of a red herring. While the tecnique for extracting colors to be reffered to in image processing is generally refered to as a 'color pallet'.



The figure above gives a better depiction of the actual file format. Vector Quantized textures have a codebook at the top of the file. Each entry in the codebook has information for four pixels, left high, left low, right high, right low, which represent four pixels in a square. The length of the codebook is generally 256 values by default, or shorter if the flag is defined in the header. In other words, each entry in the codebook can be refered to by a single byte value, which is exactly what the body of the data is. The body of the data, or the part that defined the pixel data is a list of bytes which refers to entries in the code book. Each entry in the codebook defines four pixels. As each codebook entry defines two pixels in the horizontal direction and two pixels in the

vertical direction, the width and height respectively have have the number of bytes to respresent them.

Power VR Textures Mipmaps

The term 'mipmap' gives very little insite into its functionality. Mipmaps is the concept of having multiple image sizes to reduce the number of draw calls. For example, if there is a texture with the size of 512x512, when upclose the texture will look detailed, but when the mesh is far away from the camera, then the draw calls used on a texture that is barely visible to the player is wasted. Mipmaps are used for this reason.

The figure above illustrates how mipmaps work. The full size of the texture for the box is 256x256 pixels. As the box gets further and further away from the camera, the texture will be switched to a lower resolution texture to cut down on draw times. Each times the width and height are halved to 128x128, 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, and 1x1. In this way mipmaps can be though of a function which returns a different bitmap for a given width and height, also referred to as a "mipdepth".

```
00000000  50565254 B8AA0200 01020000 00010001  PVRT¸ª..........
00000010  0000AC6B CC6BAC6B AC6BAC6B CD73CD6B  ..¬kÌk¬k¬k¬kÍsÍk
00000020  CD738C6B 8C6BCD73 AC6BAC6B ED73AC6B  ÍsŒkŒkÍs¬k¬kís¬k
00000030  AC6B8C6B AC6BED73 6C63AC6B AC6BCD73  ¬kŒk¬kíslc¬k¬kÍs
00000040  0D74AC6B CD738C6B ED730D74 ED73ED73  .t¬kÍsŒkís.tísís
00000050  AC6BCD6B 4B63CD73 CD738B6B 6B638C6B  ¬kÍkKcÍsÍs‹kkcŒk
00000060  AC6B6B63 AC6BAC6B CD730D74 AC6B8C6B  ¬kkc¬k¬kÍs.t¬kŒk
00000070  8C6BED73 6B63ED73 ED738C6B CC6BED73  ŒkískcísísŒkìkís
```

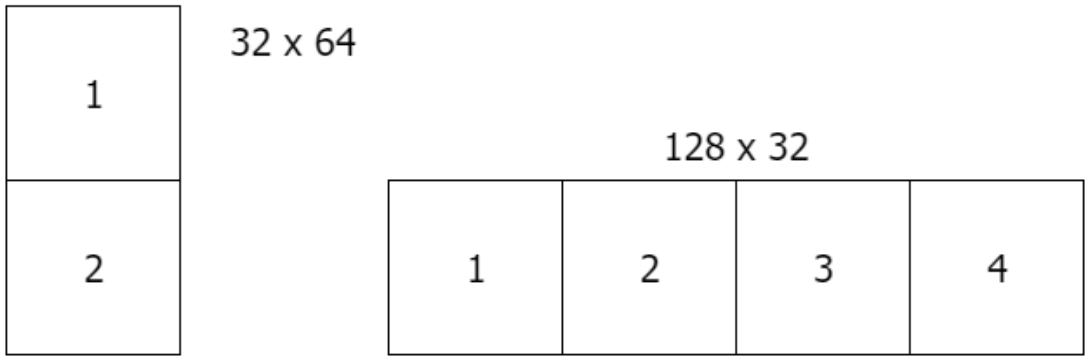
The figure above shows what a square twiddled PVR texture with mipmaps looks like in a hex editor. After the header, the body starts of with two unused bytes followed by a word value for the 1x1 bitmap. As every bitmap here will align with the 32 bit range the two usued bits are used to align bit boundaries so they line up correctly. Following that is 8 bytes for the 2x2 texture, 32 bytes for the 4x4 texture and 128 bytes for the 8x8 texture. The pattern continues like this until the texture size defined in the header. In this case that value is 256x256.

MipDepth	Dimensions	Twiddled Size	Twiddled Offset	VQ Size	VQ Offset
	Offset Adjust	0x02	0x00	0x01	0x00
0	1 x 1	0x02	0x02	0x00	0x01
1	2 x 2	0x08	0x04	0x01	0x01
2	4 x 4	0x20	0x0C	0x04	0x2
3	8 x 8	0x80	0x2C	0x10	0x6

MipDepth	Dimensions	Twiddled Size	Twiddled Offset	VQ Size	VQ Offset
4	16 x 16	0x200	0xAC	0x40	0x16
5	32 x 32	0x800	0x2AC	0x100	0x56
6	64 x 64	0x2000	0xAAC	0x400	0x156
7	128 x 128	0x8000	0x2AAC	0x1000	0x556
8	256 x 256	0x20000	0xAAAC	0x4000	0x1556
9	512 x 512	0x80000	0x2AAAC	0x10000	0x5556
10	1024 x 1024	0x200000	0xAAAAC	0x40000	0x15556

Power VR Textures Rectangle

The twiddled order format for textures is intended for square textures. Passing a buffer into the algorythm and parsing it when the width does not equal the length will result in out of bounds errors where the algorithm tries to found the color value for a pixel that is outside the length of the file. However PowerVR does define rectanglular textures that utilize the twiddled data format. The textures used for these will always have a width or height that is a multiple of the other. The figure below illustrates this property.



In each case the texture is divided up into squares of even height and width and each one is taken from a twiddled state into a sequential one. Each square generates a new sequential array. These arrays need to be meged back together to define a bitmap. In the case of tall rectangles where the height is greater than the width, arrays can simply be concatinated one after the other. In the case of wide rectangles, where the width is greater than the height, the process is more involved. Each row from each one of the arrays needs to be added one at a time in a way that the arrays mesh together to create a new bitmap.

Small Vector Quantized

Vector Quantized textures have a codebook size that is 256 entires by default. For 256 entries, with each pixel color definition being 2 bytes, and each entry having pixels defined inside of it, that's 2048 (0x800) bytes used for the codebook by default. In the cases of smaller textures, this is overkill as for smaller textures such as 32x32 the standard twiddled size is 2048 (0x800) bytes, which is the size of the codebook alone. In cases like these PVR textures have a data format defined as "small VQs". These textures have a smaller codebook size. The size code the codebook depends on the width of the texture and whether it uses mipmaps or not.

Size	Uses Mipmaps	Codebook Entries	Byte Size of Codebook
8 x 8	no	16	0x80
16 x 16	no	16	0x80
32 x 32	no	32	0x100
64 x 64	no	128	0x400

Size	Uses Mipmaps	Codebook Entries	Byte Size of Codebook
8 x 8	yes	16	0x80
16 x 16	yes	16	0x80
32 x 32	yes	64	0x200
64 x 64	yes	256	0x800

Pallet and Stride

The two formats pallet and stride are not covered in this documentation. But maybe included in a later revision. The pallet format represents a challenge because it utilizes a .pvp file which represpents an external pallet in addition to the .pvr which defines the pixel data. The appropriate approach seems to scan the entire asset contents for .pvp files and any .pvr files which contain the external pallet flag and attempt to find any patterns that would indicate they are connected, such as filename or format compatibility.

Stride represents a different challenge as it is a rarely used format does not seem to be present in the data on Phnatasy Star Online version 2 for the Dreamcast.

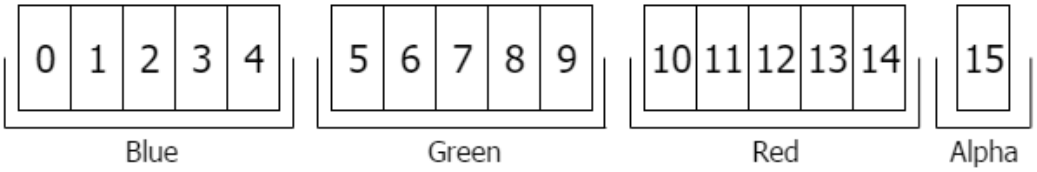
Power VR Textures Color Formats

PowerVR textures use 16 bit color formats to represent pixel data. These types are listed below.

```
#define KM_TEXTURE_ARGB1555 (0x00000000)
#define KM_TEXTURE_RGB565   (0x00000001)
#define KM_TEXTURE_ARGB4444 (0x00000002)
#define KM_TEXTURE_YUV422   (0x00000003)
#define KM_TEXTURE_BUMP     (0x00000004)
```

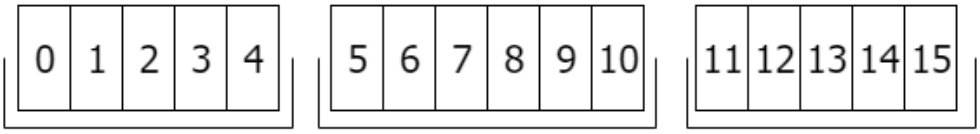
In practical application YUV422 does not occur and will not be covered in this document. BUMP represents a texture that indicates 'bumps' on a model to make it appear uneven, even though the surface of the model is actual flat. This format has not yet been analized and will not be covered in this documentation. The three prmary formats used in Phantasy Star Online Version 2 for the dreamcast are ARGB1555, RGB565 and ARGB4444. Each one presents advantages and disadvantages and each one is chosen depending on the desired effect.

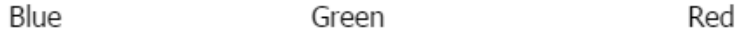
ARGB 1555




```
function ARGB1555 (short){
        var a = (short & (1<<15)) ? 0xff : 0;
        var r = (short >> 7)) & 0xf8;
        var g = (short >> 2) & 0xf8;
        var b = (short << 3) & 0xf8;
        return [r,g,b,a];
}
```

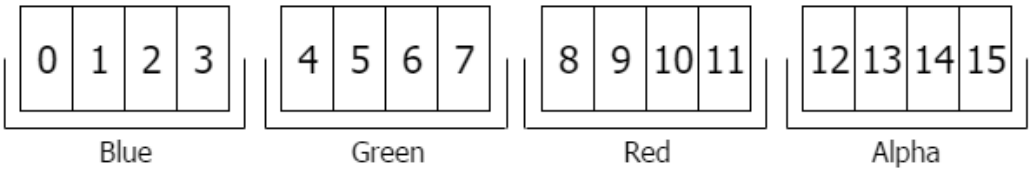
RGB 565






```
function ARGB1555 (short){
        var a = 0xff;
        var r = (v >> 8) & (0x1f<<3);
        var g = (v >> 3) & (0x3f<<2);
        var b = (v << 3) & (0x1f<<3);
        return [r,g,b,a];
}
```

ARGB 4444




```
function ARGB1555 (short){
        var a = (short >> 8) & 0xf0;
        var r = (short >> 4) & 0xf0;
        var g = (short >> 0) & 0xf0;
        var b = (short << 4) & 0xf0;
        return [r,g,b,a];
}
```