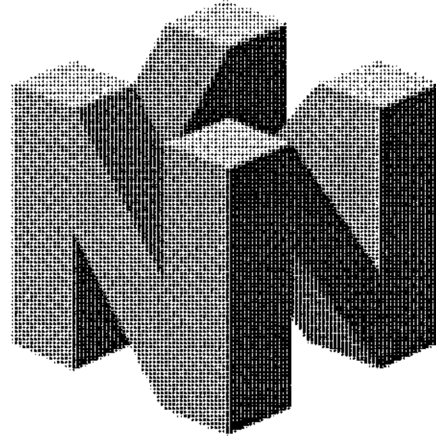


Z-Sort Microcode

User's Guide

NUS-06-0164-001A
Released: 1/9/98

D.C.N. NUS-06-0164-001 REV A

"Confidential"

This document contains confidential and proprietary information of Nintendo and is also protected under the copyright laws of the United States and foreign countries. No part of this document may be released, distributed, transmitted or reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from Nintendo.

© 1998 Nintendo

TM® and the "N" logo are trademarks of Nintendo

Table of Contents

Chapter 1 Introduction and Installation 5
Introduction to Z-Sort Microcode 5
Installation 5
Confirm Package Installation 5
For IRIX 5.3, 6.2, 6.3 6
For Partner-N64PC (Windows95/NT) 6
Chapter 2 Z-Sort Microcode Functions 7
Drawing Flow Using Z-Sort 7
Drawing and Arithmetic Operations 8
RSP Processing Installation Methods 8
2-Task Processing 9
2-Pass Parallel Processing 11
Chapter 3 Drawing 13
Drawable Objects (ZObject) 13
ZObject List Processing 13
Z-Sort Processing 14
Preparation of gSPZObject Array 14
Array initialization 15
Array Registration According to Screen Depth of each ZObject 15
ZObject Data Formats 16
zShTri Structure 16
zShQuad Structure 17
zTxTri Structure 20
zTxQuad Structure 21
zNull Structure 23
Controlling RDP Commands with RDPcmd Parameters 23
Clear Screen and Other Drawing Processing 26
gSPZRdpCmd (Gfx *gp. Gfx *rdpcmd) 26
Chapter 4 Arithmetic Operations 27
Display Objects and Arithmetic Operations 27
(Operation A) --- gSPZMultMPMtx 27
(Operation B) --- gSPZLight / gSPZLightMaterial 27
(Operation C) --- gSPZLight/gSPZLightMaterial 27
Work Area for Operations in DMEM 27
GBI List 29

Table of Contents (Continued)

GBI Functions 29
gSPZSetUmem (Gfx *gp, u32 umem, u32 size, u64 *adrs) 29
gSPZGetUMem (Gfx *gp, u32 umem, u32 size, u64 *adrs) 29
gSPZSetUMtx (Gfx *gp, u32 mid. Mtx *mptr) 29
gSPZGetUMtx (Gfx *gp, u32 mid. Mtx *mptr) 30
gSPZMtxCat (Gfx *gp, u32 mids, u32 midt, u32 midd) 30
gSPZMtxTrnsp3x3 (Gfx *gp, u32 mid) 30
gSPZViewPort (Gfx *gp, Vp *vp) 30
gSPZMultMPMtx (Gfx *gp, u32 mid, u32 src, u32 num, u32 dest) 31
gSPZSetAmbient (Gfx *gp, u32 umem, Ambient *ambient); 33
gSPZSetDefuse (Gfx *gp, u32 umem, u32 lid, Light *defuse); 33
gSPZSetLookAt (Gfx *gp, u32 umem, u32 lnum, LookAt *lookat) 34
gSPZXfmLights (gfx *gp, u32 mid, u32 lnum, u32 umem) 34
gSPZLight (Gfx *gp, u32 nsrc, u32 num, u32 cdest, u32 tdest) 35
gSPZLightMaterial (Gfx *gp, u32 msrc, u32 nsrc, u32 num, u32 cdest, u32 tdest) 35
gSPZMixS16 (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor) 36
gSPZMixS8 (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor) 36
gSPZMixU8 (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor) 36

Chapter 5 Other Processing 37

GBI List 37

GBI Functions 37
gSPZSetSubDL (Gfx *gp, Gfx *subdl) 37
gSPZLinkSubDL (Gfx *gp) 37
gSPZSendMessage (Gfx *gp) 37
gSPZWaitSignal (Gfx *gp, zSignal *sig, u32 param) 38

Chapter 6 Compatibility With Other Microcodes 39

About GBIs 39

Common GBIs 39
gSPZSegment (Gfx *gp, u32 seg, u32 base) 39
gSPZPerspNormalize (Gfx *gp, u16 persp) 39

Chapter 7 CPU Support Library 41

Chapter 8 Sample Programs 43

zonetri/ 43

cubes-1 43

Chapter 1 Introduction and Installation

Introduction to Z-Sort Microcode

Z-Sort Microcode was developed to delete obscured screens at the Nintendo 64 (N64) hardware level using a Z-sort. Z-Sort creates screens using a procedure which sorts all the graphics to be displayed on the screen in order of their depth on the screen and then draws them in order from back to front.

The N64 OS/Library supports obscured screen processing using the Z-Buffer. This processing method judges whether or not a graphic is visible on a pixel-by-pixel basis. Compared with Z-Sort, this has the advantage being able to accurately express the relationship before and after the graphic is displayed. On the other hand, access to RAM increases. With Z-Sort, although the relationship before and after display cannot be processed to the same extent as with the Z-Buffer, the amount of RAM access per graphic decreases. Thus, the amount of graphics displayed on the screen within a specific time increases compared to the Z-Buffer method.

The advantage of Z-Sort is that the improved RAM band makes the RDP processing load lighter. In many applications, the time required to perform RDP processing causes a bottleneck. Thus, lighter processing load is ideal when the volume of graphics is high.

One note of caution, however. RSP processing load does not change significantly. RDP processing load changes according to the size of the area to be filled. With a drawing in a small area in particular, RDP processing ends sooner than RSP processing. Because there are many small drawing areas, RDP processing waits for RSP processing to end, during which time the processing capacity does not change with Z-Sort or with Z-Buffer. When the drawing area is somewhat larger, however, the Z-Sort method is effective. Z-Sort Microcode cannot do everything. Carefully consider the screen to be drawn before using Z-Sort.

Installation

This description pertains to installation of Z-Sort Microcode when it is distributed as a separate package. If it is already included in the N64 OS/Library, these operations are not necessary.

Confirm Package Installation

This microcode runs on N64 OS/Library version 2.0H or later. When using 2.0H, confirm that the following packages have been installed. If they are not installed, install them first.

ultra	N64 OS/Library Version 2.0H
patchNmisc_082297	- Patch Nmisc_082297: miscellaneous patches for N64 OS/Library version 2.0H

The Z-Sort package includes the following patch and, therefore, it need not be obtained separately. If the following patch is already installed, install the Z-Sort package as instructed above.

patchNgbi_040997	- Patch Ngbi_040997: patch for gSP1Quandrangle() in gbi.h for N64 OS/Library version 2.0H

NUS-06-0164-001A
Released: 1/9/98

For IRIX 5.3, 6.2, 6.3

The Z-Sort Microcode package is formatted as follows.

patchNuZST_mmddyy (mmddyy is the release date)

Install this patch using the Software Manager or the inst command. This will install the following files. For details on the microcode, see the README file.

`/usr/src/PR/doc/gfxucode.Z-Sort/README`	README file
`/usr/lib/PR/gspZ-Sort.fifo.o`	Z-Sort Microcode
`/usr/lib/PR/gspZ-Sort.pl.fifo.o` improved	Z-Sort Microcode (version with arithmetic operations)
`/usr/include/PR/gbi.h`	Z-Sort include file
`/usr/include/PR/gZ-Sort.h`	Z-Sort include file
`/usr/include/PR/rcp.h`	Z-Sort include file
`/usr/src/PR/gZ-Sort/*`	Z-Sort sample programs

For Partner-N64PC (Windows95/NT)

The Z-Sort Microcode package is formatted as follows.

Z-SORTxxx.EXE (xxx is the release number)

This file is self-extracting. When executed, the user will be asked for the installation destination. Input the ROOT directory of the N64 OS/Library. The default is `c:\ultra`. The file opens under the specified directory just as with the IRIX version.

Chapter 2 Z-Sort Microcode Functions

Drawing Flow Using Z-Sort

Z-Sort Microcode supports triangle areas, quadrangle areas, and texture and fill rectangles using RDP commands. In this manual, all areas to be drawn by the RDP are called `ZObjects`.

In Z-Sort Microcode, for each ZObject, one screen depth value is found to represent the drawing area. Each ZObject is then sorted by that screen depth and obscured screen processing is executed by drawing the ZObjects in order from the back to the front.

The processing flow for ZObject drawing is as follows.

1. Multiply model matrix by perspective transformation matrix, etc.
2. Calculate coordinate transformation/perspective transformation/screen depth for model vertices.
3. Determine whether there are vertices in the screen.
4. Determine clipping/back plane.
5. Construct ZObject data.
6. Create ZObject list.
7. Draw in order of ZObject list (drawing processing).

In order to draw a ZObject, the information concerning how the ZObject will be drawn must be prepared as data. With conventional Fast3D Microcode, the Vertex and Tri commands were combined to draw triangles, while with Z-Sort Microcode, drawing is performed by creating ZObject structures.

Not all of these processes are available in Z-Sort Microcode. The major difference between Z-Sort and other graphics microcodes is that Z-Sort Microcode does not function by itself; the CPU must perform some of the processing related with drawing.

For example, the function of sorting ZObjects in order of screen depth is not available as microcode. Since the CPU does not perform sorting, that function must be handed over to the RSP.

At the very least, the CPU must perform the following processes.

- Clipping/back screen determination
- ZObject data construction
- ZObject list creation

Z-Sort Microcode currently offers the following main functions. Each process is controlled by the Display List (DL) comprised of one or more GBI commands.

- Multiplication of model matrix by perspective transformation matrix
- Calculation of coordinate transformation/perspective transformation/ screen depth for model vertices
- Creating flags for whether or not vertices are in the screen
- Drawing in order of ZObject list (drawing processing)

Naturally, matrix multiplication and coordinate transformation (here, called arithmetic operation processing) could also be performed by the CPU. Dividing these tasks between the CPU and the RSP according to available processor capacity is best. For the remainder of the explanation, however, it is assumed that the RSP will perform arithmetic operation processing. If the CPU is to perform operation processing, read about the arithmetic operation processing explained in chapter 4.

Drawing and Arithmetic Operations

When the RSP performs the arithmetic operations, Z-Sort Microcode processing uses two passes. Since the coordinate transformation of the before and after ZObject is not completed, the final Z-Sort results cannot be obtained. This means that data cannot flow in a pipeline like it does with other microcodes. It is necessary to temporarily hold all of the Zobject information.

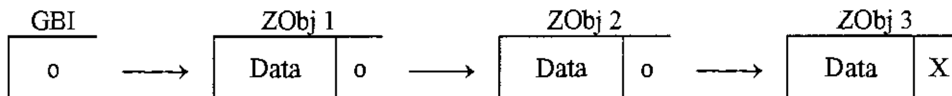
Thus, the following functions related to coordinate transformation are called arithmetic operation processing and are performed on the first pass. These processes are "vertex" coordinate transformations, so ZObject plane data is not created at this time. Note that the CPU creates actual ZObject plane data from the results of vertex coordinate transformation.

- Multiplication of model matrix by perspective transformation matrix
- Calculation of coordinate transformation/perspective transformation/ screen depth for model vertices
- Determination of whether there are vertices in the screen

The next process following Z-Sorting by the CPU, is called "drawing processing" and is performed on the second pass of the RSP.

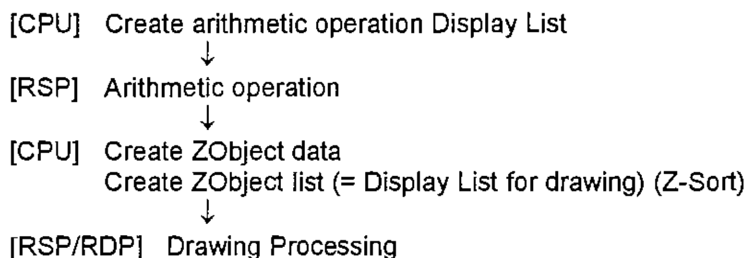
- Drawing in order of ZObject list

This ZObject list is a chained data string similar to that below, in which ZObject data are linked in the form of a list in order from the back of the screen. The X of ZObj 3 below signifies the end of the chain.



It is necessary that the CPU create this ZObject list. Since Z-Sort Microcode supports ZObjects in list format, the cost of substituting in data when sorting can be kept to a minimum. Any sorting algorithm may be used. Incidentally, in the sample program of this microcode, packet sorting divided into 1024 steps between far and near planes is performed by creating multiple ZObject lists.

Once the above is complete, the processing flow continues as follows.



RSP Processing Implementation Methods

It was discussed above that the RSP processing is divided into two passes. The methods for implementing this will be explained here.

- Implementation method A) 2-task processing
- Implementation method B) 2-pass parallel processing

A detailed explanation follows. Since A and B each has advantages and disadvantages, select the implementation method carefully.

2-Task Processing

The 2-task processing method starts by dividing the tasks into arithmetic operation processing and drawing processing. This should be an easy method to understand since it resembles the starting methods for other microcodes. The principle must first be understood.

Implementation Method A

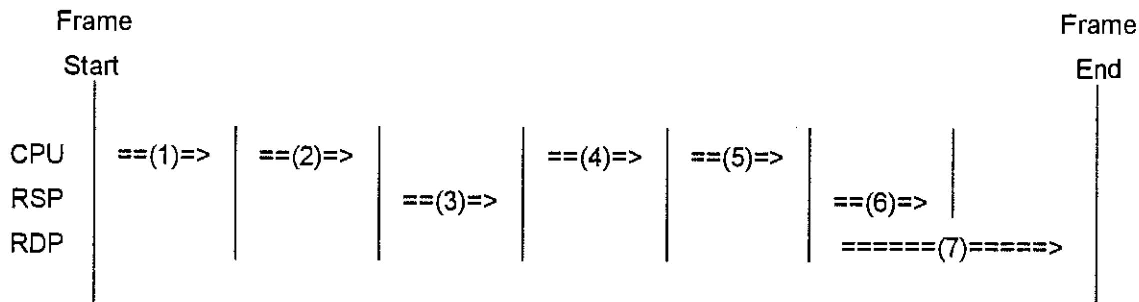
This is the simplest 2-task processing method. It is listed below.

1. Create the Display List for arithmetic operations.
2. Start the first task of the RSP (Display List for arithmetic operations).
3. The RSP performs calculations and the CPU waits until the RSP is done.
4. a. Create ZObject data using the calculation results.
   b. Create ZObject processing links by sorting.
   c. Create the Display List for drawing processing.
5. Start the second task of the RSP (Display List for drawing processing).
6. The RSP performs drawing calculations and the CPU waits until the RDP is done.
7. The RDP performs drawing.

This is the simplest method and, therefore, the easiest to understand. It is effective when shortening the time between key input and screen response. Also, since a single buffer is sufficient as the buffer for developing ZObject data, the amount of memory that should be reserved is decreased.

True of all implementation methods, constructing ZObjects using the CPU in (4) a~c above, requires a considerable computation cost. Differences in the number of ZObjects that can be drawn per frame appear in the ways in which this portion is implementated. If possible, it is recommended that you use "assembly language" instead of C language for this part of the implementation.

The operating status of the CPU, RSP, and RDP for each process is shown below. The numbers in parentheses correspond to those above.

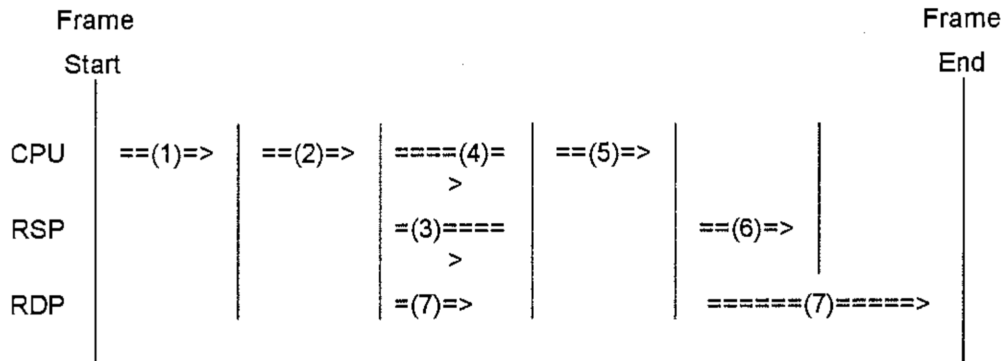


Implementation Method B

One of the problems with implementation method A is that there are no places where the CPU and RSP can operate in parallel. This leaves openings in both CPU and RSP processing. Pipelining processes (3) and (4), in method A, would eliminate some of the space. To create data for a certain number of ZObjects, creation must begin at vertex data points. To support this, Z-Sort Microcode contains a GBI command to send this message to the CPU. When this message is inserted midway through the arithmetic operation processing GBI command, the RSP sends the message to the CPU when the command is processed. When the CPU receives the message, it knows that arithmetic operations prior to the command that sent the message have been completed.

Also, the RDP does nothing during processes (1) to (5) of method A. Thus, the RDP's idle time means reduced drawing performance. Needless to say, to save RDP processing time, it is best that RDP drawing processes that do not require RSP operations, such as screen clearing, be performed within the first RSP pass.

Merely as an example, if the above points are improved, the following results.



In the third stage, each processor performs the following processing.

CPU: Creates ZObject data from RSP coordinate calculation data and sorts it. Also, creates the DL for the second pass.

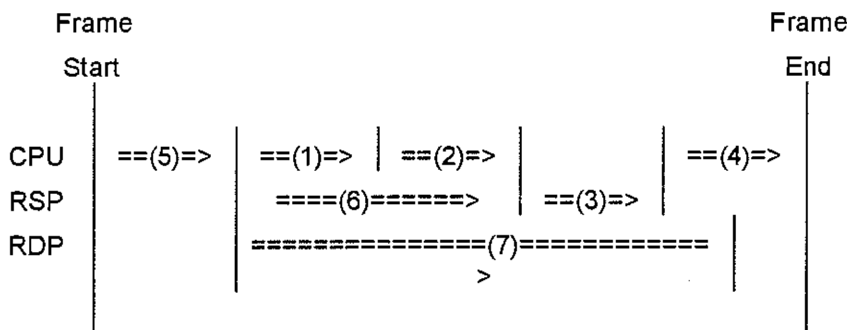
RSP: Performs coordinate calculations and sends a message to the CPU every time a vertex data point necessary to create a certain amount of ZObject data is obtained.

RDP: Primarily performs processing that does not require RSP operations, such as screen clearing

Implementating this processing system to perform the above is more complex than system A. There is no significant difference between the difficulty of this processing and that of 2-pass parallel processing described below. Since the performance gain resulting from serial processing (3) and (4) is generally not that great, a different method should be used when reducing the delay in response time after key input and reducing the memory footprint are not important.

Implementation Method C

If the delay in key input response time is acceptable, the following implementation method may be used. The processes (5) through (7) are carried over to the next frame.



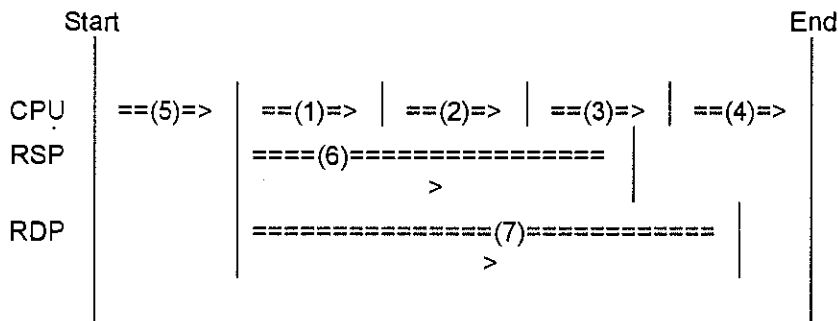
In this case, the time between key input and screen response slows, lengthening the RDP processing time.

Since processes (3) and (4) must wait until (6) has ended, the processing time of (6) in the RSP and the processing time of (4) in the CPU must be as short as possible. Since the time the RSP must wait for RDP drawing decreases when the FIFO buffer is enlarged, the processing time of (6) normally shortens, boosting the performance of this processing system. When numerous small ZObjects appear, however, the RSP processing time becomes longer than the RDP's. Since the RDP waits for the RSP, performance does not improve even when the FIFO buffer in enlarged. Thus, it would appear that (4) should be implemented using assembly language.

Considering the ease of implementation and performance, this method appears to be the most balanced among the 2-task processing methods.

Implementation Method D

In the rare event that implementation and sufficient performance in (3) can be obtained using the CPU instead of the RSP, problem-free parallel processing would be possible, as shown below. However, since (6) and (4) sometimes overlap, ZObject data and the DL must be processed using a double buffer.



Whether or not this implementation improves performance depends on the extent to which (3) can be performed faster. If possible, use assembly language for this part of the implementation as was done with (4).

2-Pass Parallel Processing

In graphics processing, the RDP processing time rarely matches the RSP processing time. The FIFO buffer exists to absorb this difference. When the RDP processing time exceeds the RSP processing time, the End Processing RDP command is stored in the FIFO buffer. Since the FIFO buffer size is limited, if the wait is too long, the buffer becomes full.

In other microcodes (Fast3D, F3DEX, S2DEX), when the buffer is full, the RSP waits until space opens up in the FIFO buffer. Merely waiting for RDP processing needlessly consumes the calculation capacity of the RSP.

To eliminate this waste in Z-Sort Microcode, the RSP can perform other DL processing (mainly, arithmetic operation processing) while waiting for RDP processing. This combines arithmetic operation processing and drawing processing into a single task for a pseudo-parallel processing called 2-pass parallel processing.

In 2-pass parallel processing, the DL processed within the RSP stand-by time is called the Sub Display List (Sub DL). Here, as in conventional microcodes, the normal DL is called the Main DL to distinguish it from the Sub DL. Just like the Main DL, the Sub DL has 18 dedicated DL stacks. Since the Sub DL is processed while the RSP is waiting for RDP processing, the GBI commands that can be processed by the Sub DL are limited. Naturally, commands using the RDP cannot be executed. Only commands using the RSP can be used. If GBI commands using the RDP are included in the Sub DL, a malfunction will result. Specific GBI commands which can be included in the Sub DL will be explained later. Mainly arithmetic operation commands can be used.

In actual processing, the RDP processing time usually is not longer than the RSP processing time, and if the RDP drawing area is small, the wasted RSP time mentioned above disappears. When this happens, the Sub DL cannot be processed until expressly called by the Main DL.

The specifications for this microcode assume that there will be inconveniences. Since the RDP drawing area varies depending on the scene to be drawn, the RSP stand-by time in which the Sub DL can be processed is not constant. RSP arithmetic operation processing must end within a certain time to ensure the CPU's ZObject creation time. This is why Sub DL processing even outside the RSP stand-by time is so desirable.

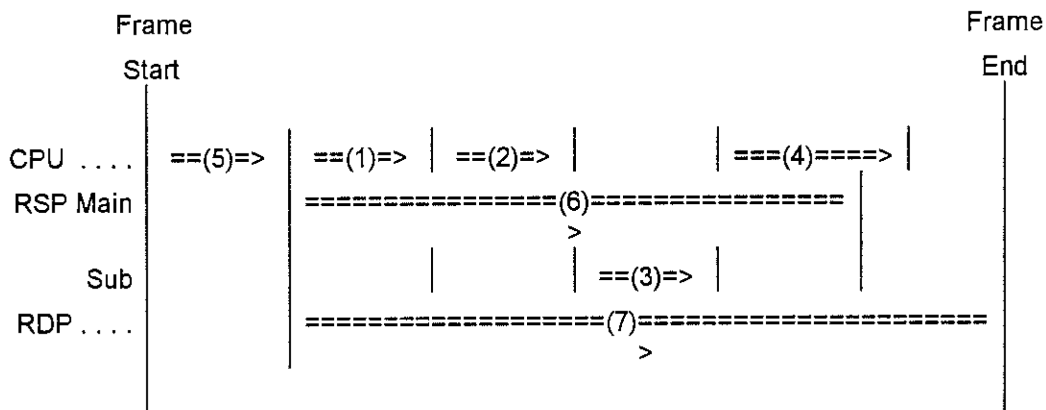
For the above reasons, a microcode `gspZ-Sort.pl.fifo.o` (`Z-Sort.pl ucode`) has been prepared that starts each GBI command in the Sub DL, one at a time, each time a certain amount of ZObject processing is completed; even outside the RSP stand-by time. The timing for calling the Sub DL commands differs depending on the type of ZObject drawn. For polygon ZObjects, one Sub DL command is required for every two to four ZObjects.

In contrast to `Z-Sort.pl ucode`, the microcode `gspZ-Sort.fifo.o` (`Z-Sort ucode`) is for Sub DL processing only during RSP stand-by.

Since this additional processing is performed by `Z-Sort. pl ucode`, the overhead becomes larger than in `Z-Sort ucode`. Therefore, `Z-Sort ucode` offers slightly better RDP drawing performance. These two types of microcode are identical except for the difference in calling the Sub DL and the larger overhead. Select the type desired according to the circumstances.

The 2-pass parallel processing implementation is as follows. Here, (3) and (6) are processed in parallel.

Installation E



Chapter 3 Drawing

Drawable Objects (ZObject)

As explained earlier, in Z-Sort Microcode, graphics are drawn in drawing areas called ZObjects. The drawing parameters for each type of ZObject are defined below, according to the corresponding structure.

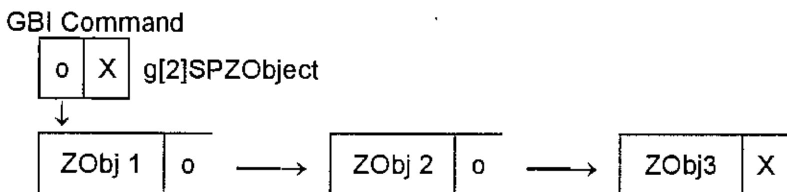
`zShTri`	triangle with smooth shading
`zShQuad`	quadrangle with smooth shading
`zTxTri`	triangle with textured smooth shading
`zTxQuad`	quadrangle with textured smooth shading
`zNull`	other drawing areas using RDP commands (used for Fill Rectangle and Texture Rectangle)

Unfortunately, due to size limitations, Z-Sort Microcode does not provide ZObjects for drawing triangles and quadrangles with flat shading. To draw these, specify the same color for all vertices.

Although the microcode supports only these simple types of graphics, every imaginable type of graphic can be drawn using the libraries in the CPU. For details, refer to the sample programs.

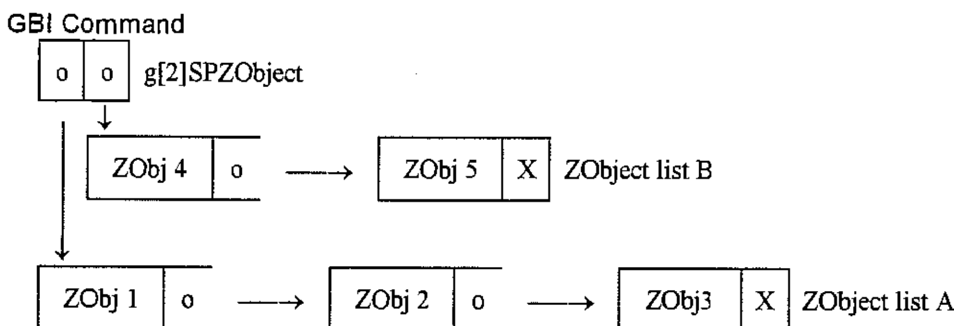
ZObject List Processing

Since ZObjects can be put into a list format, pointer data for the next ZObject and the type ID for the next ZObject can be saved at the head of the structure. The 4 bytes at the head of all ZObject structures are reserved as the header area. ZObjects can be formatted as a list depending on the values of these 4 bytes.



When the pointer and ZObject type ID in the head ZObject (ZObj 1 in the figure above) in the list are specified by the GBI command `g[s]SPZObject`, the RSP draws in order according to this list.

From 0 to 2 lists can be processed by the GBI command `g[s]SPZObject`. In other words, two ZObject lists A and B can be drawn by one GBI command.

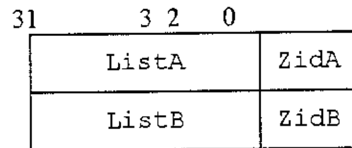


The minimum size of a GBI command is 8 bytes, which is equal to two pointer data of 4 bytes each. If fewer than two processing lists are being drawn, write the end value (= `G_ZOBJ_NONE`) in the empty space.

The data format of the GBI command `g[s]SPZObject` is as follows, with the front and back halves being the same.

```
* gSPZObject (Gfx *gp, u32 listA, u32 listB)

      listA Link parameter of ZObject link A = ZHDR (ListA, ZidA)
      listB Link parameter of ZObject link B = ZHDR (ListB, ZidB)
```



`ListA/ListB`	The head 8 bits from bit 31 to bit 3 of the pointer to the ZObject list must be 0×80. (Normally 0×80)
`ZidA/ZidB`	The ZObject type ID of head of the ZObject list

`ZHDR` (pointer, type) has been provided as a macro for setting these data (32 bits), and can be used as follows.

```
gSPZObject (gfx, ZHDR (ptr_listA, ZH_SHTRI), ZHDR (ptr_listB, ZH_TXTRI));
```

To change only processing link A or B, the direct value may be substituted in as shown below.

```
*((u32 *) gfx) = ZHDR (ptr_listA, ZH_SHTRI);
```

`ZH_xxxxx` is the ZObject type ID and takes the following five values.

`ZH_SHTRI`	triangle with smooth shading
`ZH_SHQUAD`	quadrangle with smooth shading
`ZH_TXTRI`	triangle with texture map and smooth shading
`ZH_TXQUAD`	quadrangle with texture map and smooth shading
`ZH_NULL`	other drawing areas using RDP commands

Although only `gSPZObject` has been explained here, `gsSPZObject` also exists. Further GBI command explanations follow in later chapters; however, as with this GBI, `gsSPZ***` explanations will be omitted.

Z-Sort Processing

The GBI command `g[s]SPZObject` is a structure listing only the pointer for the ZObject list and type ID of a ZObject. When this command is arrayed in multiple lists, however, three or more ZObject lists can be processed. For each ZObject list, a ZObject list of ZObjects with nearly the same screen depth is created. By listing them in order from the ZObject list at the back of the screen using `g[s]SPZObject`, they can easily be packet sorted.

The processing procedure is as follows. In this example, processing is performed dividing the screen depth for each ZObject into 1024 steps.

Preparation of `gSPZObject` Array

Since there is one ZObject list per screen depth step, 512 commands (=1024/2) are required as the `gSPZObject` array size. Since this array becomes part of the DL and is processed directly, as is, by the RSP, `gSPEndDisplayList` is added to the very end of the `gSPZObject` array. As a result, the required size becomes 513 commands (=512 + 1).

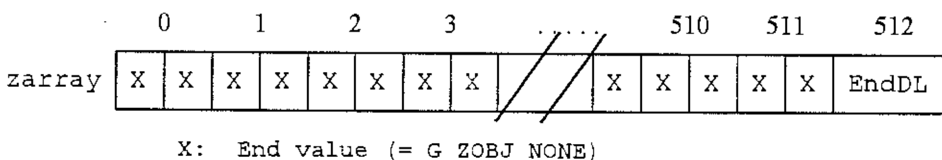
```
|
|     Gfx zarray [1024/2+1]
|
```

Array initialization

Substitute the end value (G_ZOBJ_NONE = 0x80000000) to all array elements to initialize the array. Write EndDL at the very end, as shown below.

```
Gfx *zp = zarray + 512;

gSPEndDisplayList (zp);
while (zp != zarray) {
      gSPZObject (-- zp, G_ZOBJ_NONE, G_ZOBJ_NONE);
}
```



Array Registration According to Screen Depth of each ZObject

Calculate the screen depth for each ZObject. Although the RSP can calculate the value of the screen depth at each point, the decision as to which value to use as the screen depth for the ZObject is up to the user. Here are some examples of screen depth values.

Examples of screen depths in triangle ZObjects

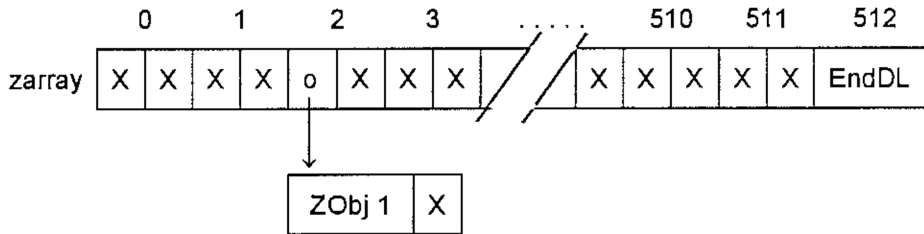
- Smallest value for distance from 3 vertices
- Largest value for distance from 3 vertices
- Average value for distance from 3 vertices
- Median value between largest and smallest values for distance from 3 vertices

Note: The inverse of the distance can also be used. The sample programs use the average of the inverse values.

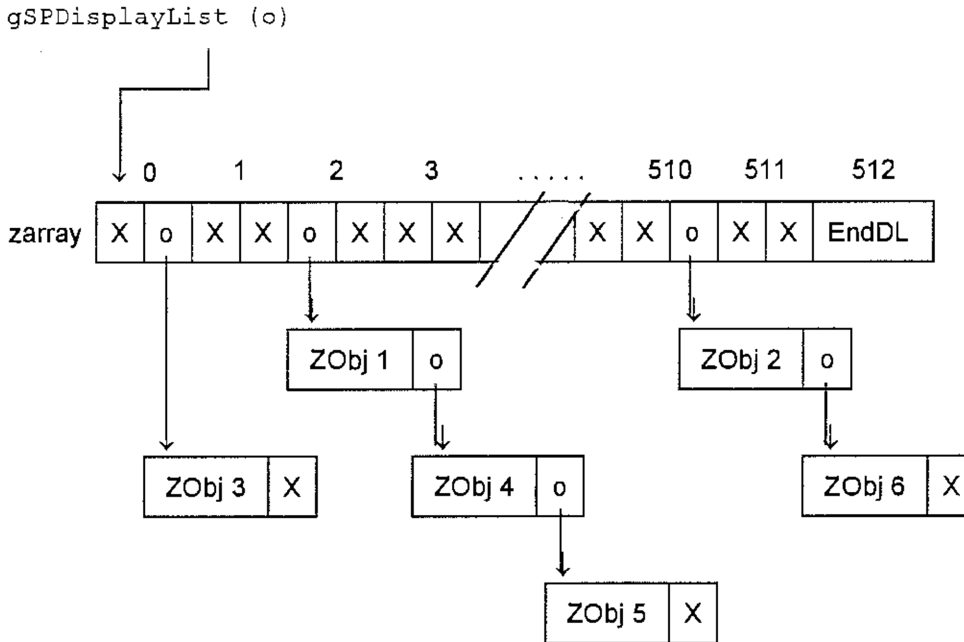
This value is normalized between 0 and 1023 and is the number of the array element to register. Store this number in the header of the ZObject in which the pointer and ZObject type ID that originally existed in the applicable array element are registered, and write the pointer to the ZObject structure data and the ZObject type ID to the corresponding array element.

```
s32    zid;                             /* No. of array element to register
*/
zHeader *zhptr:                         /* ZObject pointer */
u32    ztype;                           /* ZObject type ID */

for ( each ZObject ) {
      Calculate zid from the screen depth;
      if ( zid < 0 )          zid = 0;          /* Clamp zid */
      if ( zid > 1023)        zid = 1023;
      zhptr->t.header = *(uzarray+zid);         /* Set next node */
      *(uzarray+zid) = SHDR (zhptr, ztype);     /* Register to zarray */
}
```



Drawing processing can be performed when this process is performed on all ZObjects and the completed arrays are called by `gSPDisplayList`.



In the above example, drawing is performed in the order ZObj 3 → ZObj 1 → ZObj 4 → ZObj 5 → ZObj 2 → ZObj 6.

ZObject Data Formats

Z-Sort Microcode supports five types of ZObjects and the data required to draw each differs. The five types of structures for storing each type of ZObject data are explained below.

`zShTri` Structure

The `zShTri` structure is used for drawing a triangle with smooth shading and no texture. The following three groups of `zShVtx` vertex data are necessary for specifying this shape.

```
typedef struct {
  s16      x,     y;                   /* Vertex screen coordinates (s10.2) */
  u8       r,     g,     b,     a;     /* Each color in vertex 0. .255 */
}     zShVtx;
```

The `zShTri` structure has the following data format.

	+0		+4			+7
	Hdr		RDP cmd			
V0	X	Y	R	G	B	A
V1	X	Y	R	G	B	A
V2	X	Y	R	G	B	A

```
typedef struct          {
  zHeader               *header;        /* Information on next ZObject */
  Gfx                   *rdpcmd1;       /* Pre-processing DP command */
  zShVtx                v[3];           /* Vertex data */
}       zShTri_t;

typedef struct          {               /* Structure for word access */
  zHeader               *header;
  Gfx                   *rdpcmd1;
  u32                   xy0, clr0;
  u32                   xy1, clr1;
  u32                   xy2, clr2;
}       zShTri_w;

typedef union           {
  zShTri_t              t;
  zShTri_w              w;
  u64                   force_structure_alignment;
}     zShTri;
```

A triangle formed from the three vertices specified by this structure is drawn. At this point, the back side of the triangle is not taken into consideration. The triangle will be drawn regardless of the direction it faces. When a triangle facing the back is not desired, after the CPU determines the front and back when it creates ZObject data, draw only the ZObjects facing the front. The front/back determination is the same as for other polygon ZObjects.

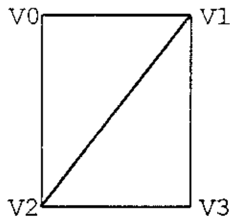
When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject.

The member variable `rdpcmd1` is used to change the current RDP processing mode. Specify the RDP command DL string to be sent to the RDP before drawing the ZObject. For details on `rdpcmd1`, see, "Controlling RDP Commands with `RDPcmd` Parameters" on page 23.

`zShQuad` Structure

The `zShQuad` structure is used for drawing a quadrangle with smooth shading and no texture. The four groups of `zShVtx` vertex data necessary for specifying this shape are given below.

With zShQuad, a quadrangle is drawn by drawing the two triangles V0-V1-V2 and V1-V2-V3.



The zShQuad structure has the following data format.

	+0		+4			+7
	Hdr		RDP cmd			
V0	X	Y	R	G	B	A
V1	X	Y	R	G	B	A
V2	X	Y	R	G	B	A
V3	X	Y	R	G	B	A

```
typedef struct      {
  zHeader           *header;      /* Information on next ZObject */
  Gfx               *rdpcmd1;     /* Pre-processing DP command */
  zShVtx            v[4];         /* Vertex data */
}    zShQuad_t;

typedef struct      {             /* Structure for word access */
  zHeader           *header;
  Gfx               *rdpcmd1;
  u32               xy0, clr0;
  u32               xy1, clr1;
  u32               xy2, clr2;
  u32               xy3, clr3;
} zShQuad_w;

typedef union       {
  zShQuad_t         t;
  zShQuad_w         w;
  u64               force_structure_alignment;
}    zShQuad;
```

Memory requirements differ for drawing the same quadrangle using one zShQuad function or two zShTri functions. Using zShQuad requires less memory, a significant advantage.

In addition, RDP drawing performance can be greatly improved by using the CPU to dramatically change the quadrangle's dividing line to better suit RDP drawing. Specifically, compare the absolute value of the Y coordinate of the V0-V3 diagonal (ABS Y0-Y3) to the absolute value of the Y coordinate of the V1-V2 diagonal (ABS Y1-Y2). Then, substitute in the ZObject data so that the quadrangle can be drawn as two triangles along the diagonal with the smaller absolute value. Refer to the following algorithm.

```
zShQuad *zquad;

if (ABS (Y0-Y3) > ABS (Y1-Y2)) {
     /* Divide at diagonal V1-V2, divide into V0-V1-V2 and V1-V2-V3 */
     zquad->t.v[0] = V0; zquad->t.v[1] = V1;
     zquad->t.v[2] = V2; zquad->t.v[3] = V3;
} else {
     /* Divide at diagonal V0-V3, divide into V1-V0-V3 and V0-V3-V2 */
     zquad->t.v[0] = V1; zquad->t.v[1] = V0;
     zquad->t.v[2] = V3; zquad->t.v[3] = V2;
}
```

However, since the diagonal to be selected as the dividing line is unknown at this time, the four specified vertices must be in the same plane so that whichever diagonal is selected, the division of the triangles is problem free. Also, when using texture or smooth shading, the texture coordinate value (`s, t`) or color value (`r, b, g, a`) must be set to avoid contradictions. (A poor example is included in the sample program `cubes-1`.)

For a specific explanation of texture map use. When the vectors `V1, V2,` and `V3` are defined as:

```
V1 = (v1 - v0), V2 = (v2 - v0), V3 = (v3 - v0)
```

in `v0 (x0, y0, z0, s0, t0), v1 (x1, y1, z1, s1, t1), v2 (x2, y2, z2, s2, t2),` and `v3 (x3, y3, z3, s3, t3)`, the actual factors a and b must exist to satisfy:

```
V2 = a * V1 + b * V3.
```

Geometrically, the four vertices in the 5-dimensional coordinate space that included s and t must exist in the same plane.

Similarly when smooth shading and lighting are used, the color value (`r, g, b`) or the normal ray vector (`nx, ny, nz`) must satisfy the above relationship.

For example, the vertices in the `onetri` demo, below, are not good for a quadrangle.

```
onetri/static.c:
  static Vtx shade_vtx[] = {
  {    -64,     64,    -5,    0,    0,    0,      0,  0xff,     0,  0xff },
  {     64,     64,    -5,    0,    0,    0,      0,     0,     0,  0xff },
  {     64,    -64,    -5,    0,    0,    0,      0,     0,  0xff,  0xff },
  {    -64,    -64,    -5,    0,    0,    0,   0xff,     0,     0,  0xff },
  };
```

This part

There would be no problem, however, if this were expressed as illustrated below.

```
static Vtx shade_vtx[] = {
  {    -64,     64,    -5,    0,    0,    0,      0,  0xff,     0,  0xff },
  {     64,     64,    -5,    0,    0,    0,      0,     0,     0,  0xff },
  {     64,    -64,    -5,    0,    0,    0,      0,     0,  0xff,  0xff },
  {    -64,    -64,    -5,    0,    0,    0,      0,  0xff,  0xff,  0xff },
  };
```

In other words, pay close attention when the values between the vertices continuously change. No problem exists with Flat Shading in which the color values between the vertices do not change.

`zShQuad` does not crimp the back of the quadrangle as was done with other ZObjects. Plan for this when the CPU creates ZObject data.

When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject. If there is no next ZObject, substitute the end value `G_ZOBJ_NONE`.

The member variable `rdpcmd1` is used to change the current RDP processing mode. Specify the RDP command DL string to be sent to the RDP before drawing the ZObject. For details on `rdpcmd1`, see, "Controlling RDP Commands with RDPcmd Parameters" on page 23.

`zTxTri` Structure

The `zTxTri` structure is for drawing textured triangles with smooth shading. The three groups of `zTxVtx` vertex data necessary for specifying this shape are given below.

```
typedef struct      {
     s16            x, y;           /* Vertex screen coordinates (s10.2) */
     u8             r, g, b, a;     /* Each color in vertex 0. .255 */
     s16            s, t;           /* Texture coordinates in vertex (s10.5) */
     s32            invw;           /* Texture pass vective correction parameter 1/W
                                       (s15.16) (proportion to inverse of distance from
                                        perspective) */
}    zTxVtx;
```

The member variable `invw` is found as shown below from coordinate value `(X, Y, Z, W)` `W` after multiplying the coordinate value of each vertex (`x, y, z, w`) by the MP matrix. However, `perspNorm` is the parameter for normalizing the perspective transformation that can be obtained by the `guPerspective` function.

```
invw = (1<<30)/(perspNorm * W):
```

The RDP uses this value to correct the texture perspective. In the microcode's arithmetic operation processing GBI, this value can be found in the same way as perspective transformation.

The `zTxTri` structure has the following data format.

	+0		+4				+8		+c +f
	Hdr		RDP cmd 1				RDP cmd 2		RDP cmd3
V0	X	Y	R	G	B	A	S	T	invW
V1	X	Y	R	G	B	A	S	T	invW
V2	X	Y	R	G	B	A	S	T	invW

```
typedef struct      {
  zHeader           *header;        /* Information on next ZObject */
  Gfx               *rdpcmd1;       /* Pre-processing DP command 1 */
  Gfx               *rdpcmd2;       /* Pre-processing DP command 2 */
  Gfx               *rdpcmd3;       /* Pre-processing DP command 3 */
  zTxVtx            v[3];           /* Vertex data */
}    zTxTri         t;

typedef struct      {               /* Structure for word access */
  zHeader           *header;
  Gfx               *rdpcmd1;
  Gfx               *rdpcmd2;
  Gfx               *rdpcmd3;
  u32               xy0, clr0, st0, invw0;
  u32               xy1, clr1, st1, invw1;
  u32               xy2, clr2, st2, invw2;
}    zTxTri         w;

typedef union       {
  zTxTri_t          t;
  zTxTri_w          w;
  u64               force_structure_alignment
} zTxTri;
```

zTxTri does not crimp the back of the triangle as was done with other ZObjects. Plan for this when the CPU creates ZObject data.

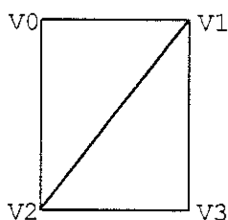
When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject. If there is no next ZObject, assign the end value G_ZOBJ_NONE.

The member variables rdpcmd1, 2, and 3 are used to change the current RDP processing mode to load the texture. Specify the three RDP command DL strings to be sent to the RDP before drawing the ZObject. For details on rdpcmd1, 2, and 3, see, "Controlling RDP Commands with RDPcmd Parameters" on page 23.

zTxQuad Structure

The zTxQuad structure is for drawing textured quadrangles with smooth shading. The four groups of zTxVtx vertex data necessary for specifying this shape are given below.

With zTxQuad, a quadrangle is drawn by drawing the two triangles V0-V1-V2 and V1-V2-V3.



The zTxQuad structure has the following data format.

	+0		+4				+8		+c +f
	Hdr		RDP cmd 1				RDP cmd 2		RDP cmd3
V0	X	Y	R	G	B	A	S	T	invW
V1	X	Y	R	G	B	A	S	T	invW
V2	X	Y	R	G	B	A	S	T	invW
V3	X	Y	R	G	B	A	S	T	invW

```
typedef struct      {
  zHeader           *header;      /* Information on next ZObject */
  Gfx               *rdpcmd1;     /* Pre-processing DP command 1 */
  Gfx               *rdpcmd2;     /* Pre-processing DP command 2 */
  Gfx               *rdpcmd3;     /* Pre-processing DP command 3 */
  zTxVtx            v[4];         /* Vertex data */
}    zTrQuad_t;

typedef struct      {             /* Structure for word access */
  zHeader           *header;
  Gfx               *rdpcmd1;
  Gfx               *rdpcmd2;
  Gfx               *rdpcmd3;
  u32               xy0, clr0, st0, invw0;
  u32               xy1, clr1, st1, invw1;
  u32               xy2, clr2, st2, invw2;
  u32               xy3, clr3, st3, invw3;
} zTxQuad_w;

typedef union       {
  zTxQuad_t         t;
  zTxQuad_w         w;
  u64               force_structure_alignment
} zTrQuad;
```

For the advantages of using zTxQuad and performance enhancing techniques, see the explanation of "zShQuad Structure" on page 17.

zTxTri does not crimp the back of the triangle as was done with other ZObjects. Plan for this when the CPU creates ZObject data.

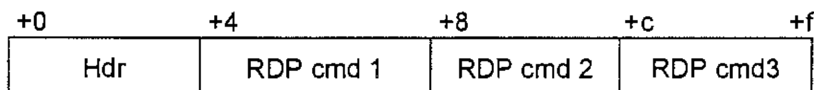
When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject. If there is no next ZObject, assign the end value G_ZOBJ_NONE.

The member variables rdpcmd1, 2, and 3 are used to change the current RDP processing mode to load the texture. Specify the three RDP command DL strings to be sent to the RDP before drawing the ZObject. For details on rdpcmd1, 2, and 3, see, "Controlling RDP Commands with RDPcmd Parameters" on page 23.

zNull Structure

The zNull structure is not for drawing so-called polygons like triangles and quadrangles. It is for drawing rectangle areas drawn by sending direct commands to the RDP (e.g., FillRectangle, TextureRectangle).

Not only the command for drawing the rectangle areas but the type of RDP command can be specified. As a result, a ZObject can be created merely by changing the Fog and Primitive colors and not actually drawing anything.




```
typedef struct      {
  zHeader           *header;
  Gfx               *rdpcmd1;
  Gfx               *rdpcmd2;
  Gfx               *rdpcmd3;
} zNull_t

typedef union       {
  zNull_t           t;
  u64               force_structure_alignment;
} zNull;
```

When the ZObjects are lined up by the list structure, the member variable header holds the pointer to the next ZObject. If there is no next ZObject, assign the end value G_ZOBJ_NONE.

Specify the three RDP command DL strings to be sent to the RDP before drawing the ZObject. For details on rdpcmd1, 2, and 3, see, "Controlling RDP Commands with RDPcmd Parameters" on page 23.

Controlling RDP Commands with RDPcmd Parameters

Each ZObject structure has one or three RDP Cmd areas. The status of the RDP during ZObject drawing processing can be changed by the member variable.

To change the RDP status, use the dedicated DL that lists the GBI commands. This is called the RDP command string.

The RDP command string can contain primarily only commands for controlling the status of the RDP. In other words, the GBI commands that can be used as the RDP command string are limited. The RDP command string and the possible GBIs are shown below. The operation of the GBI commands below is the same as in the Fast3D-compatible microcode. GBI commands not listed below may not work correctly.

GBI Commands Usable in RDP Command Strings

gSPNoOp	gDPNoOp
gSPEndDisplayList	
gDPFillRectangle	
gSPTextureRectangle	gSPTextureRectangleFlip
gDPSetColorImage	gDPSetDepthImage
gDPSetTextureImage	gDPSetScissor
gDPSetFillColor	gDPSetEnvColor
gDPSetFogColor	gDPSetBlendColor
gDPSetPrimColor	gDPSetPrimDepth

gDPSetCombineMode
gDPSetConvert
gDPSetKeyR
gDPSetKeyGB

gDPSetOtherMode
gDPPipelineMode(*)
gDPSetCycleType(*)
gSPSetTexturePersp(*)
gDPSetTextureDetail(*)
gDPSetTextureLOD(*)
gDPSetTextureLUT(*)
gDPSetTextureFilter(*)
gDPSetTextureConvert(*)
gDPSetCombineKey(*)
gDPSetColorDither(*)
gDPSetAlphaDither(*)
gDPSetAlphaCompare(*)
gDPSetDepthSource(*)
gDPSetRenderMode(*)

gDPSetTile
gDPSetTileSize

gDPLoadBlock
gDPLoadTextureBlcok
gDPLoadTextureBlockS
gDPLoadTextureBlock_4b
gDPLoadTextureBlock_4bs
gDPLoadTextureBlockYuv
gDPLoadTextureBlockYuvs
gDPLoadMultiBlock
gDPLoadMultiBlockS
gDPLoadMultiBlock_4b
gDPLoadMultiBlock_4bS

gDPLoadTile
gDPLoadTextureTile
gDPLoadTextureTile_4b
gDPLoadTLUT_pal16
gDPLoadTLUT_pal256

gDPLoadSync
gDPPipeSync
DPTileSync
gDPFullSync

One important note here regarding the inability to use `gSPSegment`. Although the segment address can be used for `gDPSetColorImage`, and the like, the value cannot be set with the RDP command string. Also note that `gSPBranchDL` and `gSPDisplayList` cannot be used.

It is assumed that the three RDP Cmd areas `rdpcmd1`, `rdpcmd2`, and `rdpcmd3` will be used as follows.

rdpcmd1: for setting RDP rendering mode

rdpcmd2: for loading to TMEM (mainly, loading to total TMEM/front half of TMEM)

rdpcmd3: for help in loading to TMEM (mainly, loading to TLUT/back half of TMEM)

Given this assumption, use only `rdpcmd1` for drawing graphics without texture (`zShTri`, `zShQuad`). All three may be specified when drawing textured graphics (`zTxTri`, `zTxQuad`).

Z-Sort Microcode is different from the microcode using the Z Buffer function, in that it draws in order from the back to the front. Thus, it cannot continuously draw only polygons with the same texture. Therefore, when using Z-Sort Microcode, ZObjects must be provided with texture information. However, Z-Sort Microcode is equipped with a mechanism for minimizing the waste that results when a texture that is already loaded to the TMEM is loaded again.

The pointer to the just-processed RDP command string is memorized. This is compared to the pointer to the RDP command string to be processed by the current ZObject and is sent to the RDP only when it is different.

The microcode contains RDP command pointer memory areas for the three RDP commands `rdpcmd1`, `rdpcmd2`, and `rdpcmd3` in DMEM (tentatively called `rdpcmd1_save`, `rdpcmd2_save`, and `rdpcmd3_save`). The algorithm for each process is written on the following page in C language.

For `zShTri`, `zShQuad` (one RDP Cmd area):

```
if (rdpcmd1 != rdpcmd1  save) {
      Processing of RDP command string displayed by rdpcmd1;
      rdpcmd1  save = rdpcmd1;
}
Drawing of ZObject;
```

The RDP command string for switching to the RenderMode is usually set to `rdpcmd1`. A sample of an RDP command string specific to `rdpcmd1` is given below.

`gsDPSetOtherMode` is the GBI for setting a number of DP mode settings at once. Since many RDP commands can be processed with a single instruction, using this command accelerates the processing speed. The commands marked (*) in the above table of GBI Commands Usable in RDP Command Strings, can be processed collectively by `gDPSetOtherMode`.

```
#define OTHERMODE_A(cyc)                (G_CYC_##cyc##|G_PM_1PRIMITIVE|G_TP_PERSP|¥
                                        G_TD_CLAMP|G_TL_TILE|G_TT_NONE|G_TF_BILERP|¥
G_TC_FILT|G_CK_NONE|G_CD_DISABLE|G_AD_DISABLE)
#define OTHERMODE_B(rm)
(G_AC_NONE|G_ZS_PRIM|G_RM_##rm##|G_RM_##rm##2)

/*------ Shade Triangle mode switching ------*/
Gfx  modeShTri[] = {
      gsDPPipeSync(),
      gsDPSetOtherMode (OTHERMODE_A (1CYCLE), OTHERMODE_B (RA_OPA_SURF)),
      gsDPSetCombineMode (G_CC_SHADE, G_CC_SHADE),
      gsSPEndDisplayList(),
};
```

For `zTxTri` and `zTxQuad` (three RDP Cmd areas)

```
if (rdpcmd1 != rdpcmd1_save) {
      rdpcmd1_save = rdpcmd1;
      Processing of RDP command string displayed with rdpcmd1;
}
if (rdpcmd2 != rdpcmd2_save) {
      rdpcmd2_save = rdpcmd2;
      Processing of RDP command string displayed with rdpcmd2;
}
if (rdpcmd3 != rdpcmd3_save) {
      rdpcmd3_save = rdpcmd3;
      if (rdpcmd3 != NULL) {
            Processing of RDP command string displayed by rdpcmd3;
      }
}
```

As with `zShTri` and `zShQuad`, the RDP command string for switching to the RenderMode is set to `rdpcmd1`. A sample of an RDP command string specific to `rdpcmd1` is given below. Palette-switching in the 4b CI texture, etc., can also be included here.

```
/*------ Textured Triangle mode switching ------*/
Gfx  modeTxTri[] = {
      gsDPPipeSync(),
      gsDPSetOtherMode (OTHERMODE_A (1CYCLE), OTHERMODE_B (RA_OPA_SURF)),
      gsDPSetCombineMode (G_CC_MODULATERGB, G_CC_MODULATERGB),
      gsSPEndDisplayList(),
};
```

Set the RDP command for loading the texture to `rdpcmd2`. A sample of an RDP command string specific to `rdpcmd2` is given below.

```
Gfx  modeTxTri2[] = {
     gsDPPipeSync(),
     gsDPLoadTextureBlock      (brick, G_IM_FMT_RGBA, G_IM_SIZ_16b, 32, 32, 0,
                               G_TX_WRAP | G_TX_MIRROR, G_TX_WRAP | G_TX_MIRROR,
                               5, 5, G_TX_NOLOD, G_TX_NOLOD),
     gsSPEndDisplayList(),
};
```

Make settings to `rdpcmd3` the same way as `rdpcmd2`. Although `rdpcmd3` is presumably used for TLUT loading, it can also be used for texture loading.

If `rdpcmd3` is unnecessary, assign `NULL (= 0x00000000)`. At this time, `rdpcmd3_save` is *cleared* by `NULL` and the RDP command displayed by `rdpcmd3` is not processed.

For `zNull` (three RDP command areas):

```
     if (rdpcmd1 != NULL && rdpcmd1 != rdpcmd1_save) {
          rdpcmd1_save = rdpcmd1;
          Processing of RDP command displayed by rdpcmd1;
     }
     if (rdpcmd2 != NULL && rdpcmd2 != rdpcmd2_save) {
          rdpcmd2_save = rdpcmd2;
          Processing of RDP command displayed by rdpcmd2;
     }
     if (rdpcmd3 != NULL && rdpcmd3 != rdpcmd3_save) {
          rdpcmd3_save = rdpcmd3;
          Processing of RDP command displayed by rdpcmd3;
     }
```

There are no particular assumptions regarding `zNull`, so it may be used freely. As can be seen from the above algorithms, when `NULL (=0x00000000)` is set to `rdpcmd1`, RDP commands are not processed. The value of the corresponding `rdpcmd_save` at this time is *saved*.

Note: *Save* at `NULL` specification differs from the `rdpcmd3` *clear* processing with `zTxTri` and `zTxQuad`.

Clear Screen and Other Drawing Processing

One important note regarding the use of Z-Sort Microcode is the inability to write direct RDP commands to a normal Display List. This is due to its being internally divided into SP command processing and DP command processing. This determines the number of microcode instructions and processing speed.

Normally, background filling processes, such as Clear Screen, are necessary for drawing all ZObjects. In Fast3D-compatible microcodes, such a GBI string is usually created in a static area and is called from the Display List side.

However, since the RDP command string for controlling such DP operations as screen clearing is called from the normal Display List, Z-Sort Microcode contains the following GBI commands. The GBI commands that can be used for the RDP command string are limited, as are which ones can be used during ZObject drawing. Refer to the preceding table. For specific examples, refer to the sample program `cubes-1`.

`gSPZRdpCmd` (Gfx *gp. Gfx *rdpcmd)

This is a pointer to the `rdpcmd` RDP command string.

Process the RDP command string. The RDP commands that can be called, however, are limited. (Refer to the table "GBI Commands Usable in RDP Command Strings" on page 23.)

Chapter 4 Arithmetic Operations

Display Objects and Arithmetic Operations

As explained previously, Z-Sort Microcode, can draw four types of polygons, `zShTri`, `zShQuad`, `zTxTri`, and `zTxQuad`. Though this initially appears to be a small number, many more shapes can be drawn by combining these basic four. This microcode offers the following three principal processing operations.

(Operation A) --- `gSPZMultMPMtx`

```
Model coordinate vertex data +
MxP matrix                  ==> Screen coordinate vertex data
```

(Operation B) --- `gSPZLight / gSPZLightMaterial`

```
Normal ray vector data  +
Material data           +
Light data              +
ModelView matrix        ==> Color data
```

(Operation C) --- `gSPZLight/gSPZLightMaterial`

```
Normal ray vector data        +
Line of sight (LookAt) data   +
ModelView matrix              ==> Texture coordinate (environment map) data
```

In all polygon ZObjects, (Operation A) must be performed to find the screen coordinate vertex data. Also, (Operation B) is required when processing light and (Operation C) is required when processing the environment map.

Each GBI used to perform operations A, B, and C (`gSPZMultMPMtx`, `gSPZLight/gSPZLightMaterial`), however, is insufficient by itself. The vertex data and transformation parameters (matrixes, etc.) must be prepared and the DMEM in the RSP must be loaded before the GBI that performs the operations. In addition, the operation results must be written and returned to the DRAM from the DMEM.

Work Area for Operations in DMEM

Z-Sort Microcode has a GBI for specialized arithmetic operations to perform transformation processing to the 3D model screen coordinate system, lighting calculations, and matrix operations using the RSP.

By combining multiple operations, such values as coordinate and color values necessary to draw ZObjects to the screen can be obtained.

For example, the following GBI commands are combined to transform model coordinates to screen coordinates.

	Command	Description
1.	gSPZViewPort	Sets VIEWPORT.
2.	gSPZPerspNormalize	Sets pass normalization factor.
3.	gSPZSetMtx	Loads PROJECTION matrix to work area in DMEM.
4.	gSPZSetMtx	Loads MODELVIEW matrix to work area in DMEM.
5.	gSPZMtxCat	Multiplies PROJECTION and MODELVIEW matrixes.
6.	gSPZSetUMem	Loads model coordinate values inside DRAM to work area in DMEM.
7.	gSPZMultMPMtx	Transforms model coordinate values to screen coordinate values.
8.	gSPZGetUMem	Outputs screen coordinate values to DRAM.

In Z-Sort Microcode, the work areas used in processing arithmetic operations are stored in DMEM. There are two types of work areas, one for general purpose use and one for matrices, each with the following sizes. Also, the general purpose work area is called the user area.

General purpose work area:			
(User area)	Total	2048 bytes	
Matrix work area	Total	192 bytes	
	(Breakdown) ModelView		64 bytes
	Projection		64 bytes
	M x P		64 bytes

The user area occupies address 0 to 2047. The application creator determines how this area is to be used.

In `libZ-Sort` of the sample program `cubes-1`, the user area is used as follows. Though the areas overlap, this does not cause a problem because they differ in terms of time sequence. Refer to the user area.

1200-1919:	stores source of model coordinate values	(Can hold up to 120 groups)
0-1919:	stores results of screen coordinate value calculations	(Can hold up to 120 groups)
0-383:	stores source of normal ray vectors	(Can hold up to 128 groups)
512-1023:	stores source of material colors	(Can hold up to 128 groups)
512-1023:	stores results of lighting calculations	(Can hold up to 128 groups)
1024-1535:	stores results of environment texture map coordinate calculations	(Can hold up to 128 groups)
1920-2047:	stores light data	(3 DEFUSE lights + 1 AMBIENT + environment map)

The user can divide up and freely use the user area. Since the matrix area has been prepared for storing matrix data, however, it cannot normally be used for other purposes. The user area can also be divided up in detail by specifying a particular address; in the matrix area, basically one of the `areas (GZM_MMTX, GZM_PMTX, or GZM_MPMTX)` is specified. However, address 0~63 at the head of the user area and address 64~127 can be used for the matrix area. Therefore, the five following areas can be used for matrices. Note that the matrix areas have been named `ModelView/Projection/MxP` for ease of understanding; their functions, however, are identical. If there is any confusion, the `ModelView` matrix can be assigned to the `MxP` matrix area.

`GZM_MMTX`	ModelView matrix area
`GZM_PMTX`	Projection matrix area
`GZM_MPMTX`	MxP matrix area
`GZM_USER0`	User area address 0~63
`GZM_USER1`	User area address 64~127

GBIs used for arithmetic operations operate with either the Main DL or the Sub DL. Thus, pay attention when reading and writing the user area by either DL. When parallel processing by the Main and Sub DLs, Sub DL GBIs sometimes destroy the data calculated by Main DL GBI. Accessing the user area via either DL, therefore, is not recommended. Also, it is better to determine which DL will perform arithmetic operations.

GBI List

This is the list of GBIs for arithmetic operations.

gSPZSetUMem	Writes data to user area
gSPZGetUMem	Reads data in user area
gSPZSetMtx	Writes matrix
gSPZGetMtx	Reads matrix
gSPZMtxCat	Multiplies matrixes
gSPZMtxTrnsp3x3	Inverts 3x3 element of matrix
gSPZViewPort	Sets VIEWPORT
gSPZMultMPMtx	Transforms model coordinate values to screen coordinate values
gSPZSetAmbient	Writes Ambient light (environment light)
gSPZSetDefuse	Writes Defuse light (diffused light)
gSPZSetLookAt	Writes LookAt structure data
gSPZXfmLights	Performs light parameter pre-processing
gSPZLight	Performs light calculations
gSPZLightMaterial	Performs light calculations taking matrix into consideration
gSPZMixS16	Performs s16 numeric interpolation
gSPZMixS8	Performs s8 numeric interpolation
gSPZMixU8	Performs u8 numeric interpolation

GBI Functions

This sections explains the GBIs for arithmetic operations.

`gSPZSetUmem` (Gfx *gp, u32 umem, u32 size, u64 *adrs)

```
umem            user area address for write destination (0~2040)
size            write size (8~2048)
adrs            pointer to write source in DRAM
```

This GBI writes data to the user area. `umem` and `size` must be multiples of 8. Also, `adrs` has an 8-byte boundary. If 10 bytes of data are needed, specify 16 bytes.

`gSPZGetUMem` (Gfx *gp, u32 umem, u32 size, u64 *adrs)

```
umem            user area address for read destination (0~2040)
size            read size (8~2048)
adrs            pointer to read destination in DRAM
```

This GBI reads data from the user area. `umem` and `size` must be multiples of 8. Also, `adrs` has an 8-byte boundary.

`gSPZSetUMtx` (Gfx *gp, u32 mid. Mtx *mptr)

```
mid             matrix area for write destination
mptr            pointer to write source in DRAM
```

This GBI writes matrix data in DRAM to the matrix area. Generally, one of `GZM_MMTX`, `GZM_PMTX`, or `GZM_MPMTX` is specified to `mid`. However, the 128 bytes at the head of the user area can also be used. If so, specify `GZM_USER0` and `GZM_USER1`. This allows address 0~63 at the head of the user area and address 64~127 to be used for the matrix area.

`gSPZGetUMtx` (Gfx *gp, u32 mid. Mtx *mptr)

```
mid                 matrix area for write destination
mptr                pointer to write source in DRAM
```

This GBI writes matrix data in DRAM to the matrix area. Generally, one of `GZM_MMTX`, `GZM_PMTX`, or `GZM_MPMTX` is specified to `mid`. However, the 128 bytes at the head of the user area can also be used. If so, specify `GZM_USER0` and `GZM_USER1`. This allows address 0~63 at the head of the user area and address 64~127 to be used for the matrix area.

`gSPZMtxCat` (Gfx *gp, u32 mids, u32 midt, u32 midd)

```
mids                matrix area S
midt                matrix area T
midd                matrix area D
```

This GBI calculates `(Matrix D) = (Matrix S) + (Matrix T)`. Generally, one of `GZM_MMTX`, `GZM_PMTX`, or `GZM_MPMTX` is specified to `mids`, `midt`, and `midd`. However, the 128 bytes at the head of the user area can also be used. If so, specify `GZM_USER0` and `GZM_USER1`. This allows address 0~63 at the head of the user area and address 64~127 to be used for the matrix area.

When matrix T and matrix D areas are the same, however, the operation may not perform as expected. There is no problem when areas S and D or S and T are the same.

`gSPZMtxTrnsp3x3` (Gfx *gp, u32 mid)

```
mid                 matrix area to be transposed
```

This GBI transposes the 3x3 element of the matrix (`x, y, z`). When the matrix is rotating, the transposed result means the reverse rotation of the source matrix. This transposed matrix is used mainly for light processing.

```
|00  01  02  03|           |00  10  20  30|
|10  11  12  13|    →      |01  11  21  13|
|20  21  22  23|           |02  12  22  23|
|30  31  32  33|           |30  31  32  33|
```

One of `GZM_MMTX`, `GZM_PMTX`, or `GZM_MPMTX` is specified to `mid`. However, the 128 bytes at the head of the user area can also be used. If so, specify `GZM_USER0` and `GZM_USER1`. This allows address 0~63 at the head of the user area and address 64~127 to be used for the matrix area.

`gSPZViewPort` (Gfx *gp, Vp *vp)

```
vp                  pointer to VIEWPORT data
```

This GBI is roughly the same as the `gSPViewPort` GBI in F3DEX. Although it sets the VIEWPORT, there are differences in the VIEWPORT data parameters. In Z-Sort Microcode, the parameter to control Fog is specified to the Vp structure member variables `vscale`, `vscale[3]` of `vtrans` and `vtrans[3]` using the following macro,

```
vp->vp.scale[3] = GZ_VIEWPORT_FOG_S (in, out);
vp->vp.trans[3] = GZ_VIEWPORT_FOG_T (in, out);
```

where:

`in:`	Fog start distance
`out:`	Fog end distance

A negative value must be set for the `vscale[1]` value to make the top part of the screen positive, i.e., the right, top, front direction (clockwise system).

Start Fog from a distance of 3000 from the perspective. When specifying so that the background color is uniform at a distance of 4000, initialize as follows.

```
Vp   viewport = {
    SCREEN_WD*2, *SCREEN_HT*2, G_MAXZ/2, GZ_VIEWPORT_FOG_S (3000, 4000)
    SCREEN_WD*2, *SCREEN_HT*2, G_MAXZ/2, GZ_VIEWPORT_FOG_S (3000, 4000)
} ;
```

`gSPZMultMPMtx` (Gfx *gp, u32 mid, u32 src, u32 num, u32 dest)

```
mid              MxP matrix
src              user area head address that stores vertex model coordinate values
num              number of vertices to be processed
dest             head address in user area that stores vertex screen coordinate
                 values after coordinate transformation
```

This GBI regards the data at the user area's `src` position as the 16-bit x, y, z value. This is multiplied by the 4x4 matrix specified by `mid` and that result (X, Y, Z, W) is normalized by W=1. The screen coordinate value is then obtained by transforming ViewPort to the obtained coordinates. Also, at this time, the flags for the FOG parameter and clipping processing are calculated and that data is output to the `dest` position. Next, 6 is added to `src` and 16 to `dest`, and the process proceeds to the next vertex. The `num` vertices are processed continuously.

The formats of the coordinate values to be input and output at this point are defined as follows as the `zVtxSrc` and `zVtxDest` structures in the header file `gZ-Sort.h`.

```
typedef struct      {
  s16  x, y, z;             /* Vertex model coordinate values (s10.2) */
}    zVtxSrc;               /* Size 6 bytes */

typedef struct      {
  s16  sx, sy;              /* Vertex screen coordinate values (s10.2) */
  s32  invw;                /* Texture pass vective correction parameter 1/W (s15.16)
/*
  s16  xi, yi;              /* X, Y values before normalization (integers only) */
  u8   cc;                  /* Flag for clip processing determination */
  u8   fog;                 /* FOG factor */
  s16  wi;                  /* W value (integers only) */
}    zVtxDest;              /* Size total 16 bytes */
```

Since the size of the `zVtxSrc` structure is 6 bytes, pay special attention to the 8-byte alignment when transferring DMA using `gSPZSetUMem`. When the transfer size must be a multiple of 8, the DMA transfer size must be rounded off to a multiple of 8.

Since the size of the `zVtxDest` structure is 16 bytes, only the 128-byte area in the 2048-byte user area can be protected. As a result, the `num` range is from 1 to 128. (In actuality, since light and other processes are performed, the range is usually smaller than this.) At this time, the `num` * 16-byte area from the `dest` address can be rewritten; the exception is when `num` is 3 or less. In this case, the 64-byte area from the position specified by `dest` is overwritten.

For example, when `num` is 3 and `dest` is 0, the correct value after transformation can be stored at address 0~47 and meaningless data can be written to address 48~63. Be careful here because the value of the source of address 48~63 will be destroyed. This specification is necessary for improving the calculation speed.

NUS-06-0164-001A
Released: 1/9/98

The routine for this GBI is illustrated below. Be sure that the unprocessed `src` is not overwritten by the `dest` output to allow the `src` and `dest` areas to be overlapped. In `libZ-Sort` of the sample program `cubes-1`, with `src` = 1200~1919 and `dest` = 0~1919, a maximum of 120 vertices can be processed.

```
for (i = 0; i < num; i ++) {
     *dest = MultMP (*src);
     src += 6;
     dest += 16;
}
```

The member variable `invw` is found as shown below from coordinate value (X, Y, Z, W) W after multiplying the coordinate value of each vertex (x, y, z, 1) by the MxP matrix. However, `perspNorm` is the parameter for normalizing the perspective transformation set by `guPerspNormalize`.

```
invw = (1<<30)/(perspNorm * W):
```

The `invw` value can be used, as is, to set `zTxTri/zTxQuad` for the ZObject.

When creating the MP matrix using `guPerspective` and `guLookAt`, the `wi` value usually indicates the distance from the perspective point. Z-Sort can also be performed by selecting this value as the screen depth.

Also, `xi, yi` is the non-normalized coordinate value before perspective transformation. This value can be used mainly for clipping processing. Z-Sort Microcode does not support clipping processing using the microcode. However, clipping can be performed using the `xi, yi, wi` value in the CPU program. The details are explained later in this manual.

By checking the value of the clipping processing determination flag `cc`, it can easily be determined whether that vertex is in ViewPort (visible area). The following explains each `cc` flag.

`GZ_CC_LEFT`	X coordinate is left of Left Plane of visible area
`GZ_CC_RIGHT`	X coordinate is right of Right Plane of visible area
`GZ_CC_TOP`	Y coordinate is above Top Plane of visible area
`GZ_CC_BOTTOM`	Y coordinate is below Bottom Plane of visible area
`GZ_CC_NEAR`	Z coordinate is closer than Near Plane of visible area
`GZ_CC_FAR`	Z coordinate is further from Far Plane of visible area

To determine whether the triangle comprised of the vertices v0, v1, and v2 is completely outside the screen, do an AND for the `cc` value of each vertex as shown below and check to see if the result is 0. If the result is not 0, it means that the entire triangle area is outside at least one of the six clip planes. If this is the case, the processing can be stopped at that point, since the triangle is outside the screen.

```
if (v0. cc & v1. cc & v2. cc) {
     Processing stopped because triangle is outside screen;
}
```

To determine whether the triangle v0, v1, v2 intersects the Near Plane, use the above formula to determine whether the triangle is outside the Near Plane and then perform OR processing. This can be used to determine whether clipping processing is being performed at the Near Plane.

```
if ((v0. cc | v1. cc | v2. cc) & GZ_CC_NEAR) {
     Perform Near clipping processing;
}
```

`fog` is used when performing FOG processing. Using the `fog` value for A in RGBA enables FOG processing. In Z-Sort Microcode, FOG is adjusted by ViewPort's Vp structure parameter. For details, refer to the sample program.

In this GBI, obtainable vertex data is actually used as shown below. The numeric values actually to be assigned to each ZObject structure are the `sx, sy, fog,` and `invw` values. The `invw` or `wi` value can be used as the screen depth value for Z-Sort processing.

```
[ For zShTri structure ]

zVtxDest          *v0, *v1, *v2;
zShTri            *shtri;

/* Screen coordinate setting */
shtri->t.v[0].x = v0->sx;      shtri->t.v[0].y = v0->sy;
shtri->t.v[1].x = v1->sx;      shtri->t.v[1].y = v1->sy;
shtri->t.v[2].x = v2->sx;      shtri->t.v[2].y = v2->sy;

/* The settings below apply only when using Fog */
shtri->t.v[0].a = v0->fog;
shtri->t.v[1].a = v1->fog;
shtri->t.v[2].a = v2->fog;
[ For zTxTri structure ]

zVtxDest          *v0, *v1, *v2;
zTxTri            *txtri;

/* Screen coordinate setting */
txtri->t.v[0].x = v0->sx;      txtri->t.v[0].y = v0->sy;
txtri->t.v[1].x = v1->sx;      txtri->t.v[1].y = v1->sy;
txtri->t.v[2].x = v2->sx;      txtri->t.v[2].y = v2->sy;

/* Texture correction parameter setting */
txtri->t.v[0].a = v0->fog;
txtri->t.v[1].a = v1->fog;
txtri->t.v[2].a = v2->fog;
```

gSPZSetAmbient (Gfx *gp, u32 umem, Ambient *ambient);
gSPZSetDefuse (Gfx *gp, u32 umem, u32 lid, Light *defuse);

```
umem              head address for light data protection area
ambient           pointer to Ambient light structure
lid               Defuse light number (0, 1, .....)
defuse            pointer to Defuse light structure
```

These GBIs write Ambient light (environment light) data or Defuse light (planar diffused light) data to the user area. The light data area is protected in advance in the user area. Its size depends on the number of Defuse lights and whether the environment is mapped. It is calculated as follows.

```
(Light data area size) =
     8 + 24 * (number of Defuse lights) + ((environment mapping)? 48 : 0));
```

In `libZ-Sort` of the sample program `cubes-1`, since three Defuse lights and environment mapping are used, the 128 bytes from 1920 to 2047 are reserved for the lights.

Fast3D macros can be used to set the Ambient and Defuse structures. When there are two Defuse lights, they are set using `gdSPDefLights2`, as shown in the example below.

```
/*--- Light parameter ---*/
static Lights2 scene_light =
gdSPDefLights2 (  0x20, 0x20, 0x20,                  /* Ambient */
                  0xe0, 0xe0, 0xe0, 0, 40, 80,          /* Defuse 0 */
                  0x40, 0x00, 0x00, 0, 80, 40 );        /* Defuse 1 */

/*--- Load light parameter ---*/
gSPZSetAmbient (gp++, 1920,    &scene_light. a);
gSPZSetDefuse (gp++, 1920, 0, &scene_light. l[0]);
gSPZSetDefuse (gp++, 1920, 1, &scene_light. l[1]);
```

`gSPZSetLookAt` (Gfx *gp, u32 umem, u32 lnum, LookAt *lookat)

`umem`	head address for light data protection area
`lnum`	number of Defuse lights
`lockat`	pointer to LookAt structure

This GBI writes the `LookAt` structure data that constitutes the parameter for environment mapping to the light data area. The light data area is protected in advance in the user area. Refer to the explanation for `gSPZSetAmbient/gSPZSetDefuse` on page 33 for further details.

Z-Sort Microcode supports `TEX_GEN_LINEAR` in the `TEX_GEN` and `TEX_GEN_LINEAR` processing modes of the Fast3D-compatible microcode for environment map processing. It is already set up for `TEX_GEN` processing.

Although the functions `guLookAtReflect` and `guLookAtHilite` in the `gu` library of the N64 OS can be used to set the `LookAt` structure, part of it differs from Z-Sort. Since the macro `guZFixLookAt` is available for correction, correct using this after setting `LookAt` using the library functions.

Shown below is the data write processing when `gdSPDefLights2` is used for two Defuse lights. The lights are set and the reflection is mapped using `guLookAtReflect`.

```
/*--- Light parameter ---*/
static Lights2 scene_light =
gdSPDefLights2 (  0x20, 0x20, 0x20,                         /* Ambient */
                  0xe0, 0xe0, 0xe0, 0, 40, 80,              /* Defuse 0 */
                  0x40, 0x00, 0x00, 0, *80, 40 );           /* Defuse 1 */

/*--- Make reflection parameter ---*/
guLookAtReflect (&dynamicp->viewing, &dynamicp->lookat,
        0, 0, 1000, 0, 0, 900, 0, 1, 0);

guZFixLookAt (&dynamic;*>lookat);

/*--- Load light parameters ---*/
gSPZSetAmbient (gp++, 1920,    &scene_light. a);
gSPZSetDefuse  (gp++, 1920, 0, &scene_light. l[0]);
gSPZSetDefuse  (gp++, 1920, 1, &scene_light. l[1]);

/*--- Load reflection parameters ---*/
gSPZSetLookAt (gp++, 1920, 2, &dynamicp*>lookat);
```

`guZFixLookAt` is defined in `gZ-Sort.h` as shown below.

```
#define guZFixLookAt(lp)
     { (lp)->l[1].l.col[1] = (lp)->l[1].l.colc[1]l = 0x00; }
```

This is because two elements have been cleared to 0. (0x80 has been assigned by the `gu` library.) If you want to optimize your processing time, refer to the source file of the `gu` library function in the N64 OS under the `/libultra/gu` directory in the libultra sample program, to correct and replace the library function.

`gSPZXfmLights` (gfx *gp, u32 mid, u32 lnum, u32 umem)

`mid`	matrix with 3x3 element at upper left of ModelView matrix inverted
`lnum`	number of lights to be processed
`umem`	head address in user area that holds light data

This GBI performs lighting pre-processing. The GBI must be called after one or both of the light data and ModelView matrix has been changed and by the time `g*SPLight` or `g*SPLightMaterial` is called. This enables pre-processing in which light data can be used in light calculations by `gSPZLight` and `gSPZLightMaterial`. Since light data will rarely change in one scene, this GBI is called when the ModelView matrix changes.

To execute this GBI, the reverse rotation matrix of the ModelView matrix is necessary. For this, the matrix with the 3x3 element at upper left of ModelView matrix inverted can usually be used. (The shading is sometimes off when scaling only certain axes, but this is not a notable problem.) `gSPZMtxTrnsp3x3` is used for the inversion.

The number of `lnum` basically is the number of Defuse lights. This does not include the Ambient lights. Also, `lnum` cannot be set to 0 in Z-Sort Microcode. To process only Ambient lighting, specify one black (RGB=0, 0, 0) Defuse light using a dummy.

When processing the environment map, use two so-called Defuse lights. When expressing highlighting and reflection, load the environment map parameter to the light parameter area and set (2 for the number of Defuse lights) to `lnum`.

When using only the environment map without using lights (no Defuse lights), the dummy Defuse light described above is unnecessary. Specify 2 to `lnum` and call the GBI.

Shown below is a processing example with the changed ModelView matrix, the light data area head at address 1920, two Defuse lights, and the environment map.

```
/*--- Set ModelView and MxP matrix ---*/
gSPZSetMtx (gp++, GZM_MMTX, &dynamicp->modeling[i]);
gSPZMtxCat (gp++, GZM_MMTX, GZM_PMTX, GZM_MPMTX);

/*--- Xfm light data ---*/
gSPZMtxTrnsp3x3 (gp++, GZM_MMTX);
gSPZXfmLight (gp++, GZM_MMTX, 1920, 4);
```

`gSPZLight` (Gfx *gp, u32 nsrc, u32 num, u32 cdest, u32 tdest)

`gSPZLightMaterial` (Gfx *gp, u32 msrc, u32 nsrc, u32 num, u32 cdest, u32 tdest)

`msrc`	head address in user area that stores material color data (color of vertices)
`nsrc`	head address of user area that stores normal ray vector data
`num`	number of normal ray vector data to be processed (multiple of 2)
`cdest`	head address of user area that stores color value of vertices after light calculation
`tdest`	head address of user area that stores texture and coordinate values of vertices after environment map calculation

This GBI regards the data from the `nsrc` address in the user area as the signed 8-bit normal ray vector value (nx, ny, nz). It calculates the lighting using the light parameters specified by `gSPZXfmLight`. This provides the light color that corresponds to the normal ray vectors. The vertex color is obtained by multiplying this light color and the material color, which is the color of the vertex itself, by each r, g, b element. These calculated color values are stored at the `cdest` address in the user area.

With `gSPZLight`, (r, g, b, a) = (255, 255, 255, 255) is used as the material color. As with Fast3D microcode, this indicates vertex coloring using light color. Also, with `gSPZLightMaterial`, use data from the `msrc` address in the user area as (r, g, b, a), in order, as the unsigned 8-bit color data.

In addition, when `LookAt` structure data is set as the light data, lighting calculation and environment map calculation are performed simultaneously. Texture coordinate values (S, T = 0.00~32.00) are output to the `tdest` address in the user area as the calculation results. Even when `LookAt` structure data is not set, an undefined value is output to `tdest` so be careful that the (`num * 4`) bytes area is not destroyed.

After `cdest` and `tdest` are output, 3 is added to `nsrc` and 4 is added to `msrc, cdest,` and `tdest,` and the process proceeds to the next vertex. Although the `num` normal ray vectors are processed

continuously, `num` must be an even number. If it is odd, (`num+1`) is output to `cdest` and `tdest` to make an even number. Meaningless data will be output to the output data position of `num+1`.

The formats of the data values to be input and output using this GBI are defined as follows as the `zNorm`, `zColor`, and `zTxtr` structures in the header file `gZ-Sort.h`.

```
typedef struct      {
  s8                nx, ny, nz;
  }  zNorm;

typedef struct      {
  u8                r, g, b, a;
  } zColor_t;

typedef union       {
  zColor_t          n;
  u32               w;
  }  zColor;

typedef struct      {
  s16               s, t;
  }  zTxtr_t;

typedef union       {
  zTxtr_t           n;
  u32               w;
  }  zTxtr;
```

Since the size of each structure is 3 or 4 bytes, pay special attention to the 8-byte alignment when transferring DMA using `gSPZSetUMem`. When the transfer size must be a multiple of 8, the DMA transfer size must be rounded off to a multiple of 8.

The routine of this GBI is basically as follows. If you are careful not to overwrite the unprocessed `nsrc` and `msrc` with the output of `cdest` and `tdest`, it is possible to overlap these areas.

```
for (i = 0; i < num; i ++) {
     (*cdest, tdest) = CalcLight (*nsrc, *msrc);
     nsrc    += 3;
     msrc    += 4;
     cdest   += 4;
     tdest   += 4;
}
```

`gSPZMixS16` (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor)
`gSPZMixS8` (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor)
`gSPZMixU8` (Gfx *gp, u32 src1, u32 src2, u32 num, u16 factor)

`src1`	head address 1 in user area where data to be interpolated is stored and head address (common) in user area from which interpolation results are to be output
`src2`	head address 2 where data to be interpolated is stored
`num`	number of data (multiple of 8)
`factor`	mixed factors (u. 15 format 0x0000~0x7fff value)

These GBIs perform linear interpolation on two numbers using the formula below. The s16, s8, and u8 data types are handled by the respective GBI.

```
(*src1) = (*src1)*factor + (*src2)*(1.0*factor);
```

In `gSPZMixS16`, `src1` and `src2` combined are limited to 16 bytes. Also, in `gSPZMixS8` and `gSPZMixU8`, `src1` and `src2` combined are limited to 8 bytes.

`num` must be a multiple of 8. If a number which is not a multiple of 8 is specified, meaningless data will be output to `src1` until the number becomes a multiple of 8.

Chapter 5 Other Processing

GBI List

This is a list of other GBIs.

`gSPZSegment`	Sets segment
`gSPZSetSubDL`	Registers/starts Sub DL
`gSPZLinkSubDL`	Processes unprocessed Sub DL
`gSPZSendMessage`	Sends message to CPU
`gSPZWaitSignal`	Waits for signal from CPU

GBI Functions

This chapter explains the remaining GBIs.

`gSPZSetSubDL` (Gfx *gp, Gfx *subdl)

`subdl`	`Sub DL head address`

This GBI registers the Sub DL and can only be processed in the Main DL. If a Sub DL has already started, a second Sub DL may not function properly, if entered. Register a Sub DL only after the processing of any Sub DL already registered by `gSPZLinkSubDL` is completed.

`gSPZLinkSubDL` (Gfx *gp)

This GBI processes the Sub DL remaining to be processed and can only be processed in the Main DL. If a Sub DL has already ended, nothing happens when the Sub DL are not registered.

`gSPZSendMessage` (Gfx *gp)

This GBI sends a SP_BREAK message to the CPU to inform the CPU of the status of Display List execution.

When the DL execution status is unknown, the CPU cannot determine whether or not processing has been completed, forcing it to wait until RSP processing has ended (until the RSP message is received).

Display List

| ZObject A vertex calculation
| ZObject B vertex calculation
| ZObject C vertex calculation
↓ At this point, end message is sent to CPU

If the Display List is prepared as shown below using this GBI, the CPU can know whether or not the vertex calculation for each ZObject has ended and can immediately build ZObjects.

Display List

| ZObject A vertex calculation
| gSPZSendMessage → message sent to CPU
| ZObject B vertex calculation
| gSPZSendMessage → message sent to CPU
| ZObject C vertex calculation
↓ gSPZSendMessage → message sent to CPU

Given the overhead resulting from actually sending and receiving messages for each Zobject, as explained above, it may be better to send messages for multiple ZObjects rather than for each object. This is up to the user.

Just as with the delivery of normal messages, for the CPU to receive the `SP_BREAK` message sent from the RSP, a message queue is used. Get the message queue for the `SP_BREAK` message and connect it to `OS_EVENT_SP_BREAK` using `osSetEventMesg`. Also, although it is safer to set the size of the queue to greater than the number of `gSPZSendMessage` in the Display List, this is not necessary. As long as the number of `SP_BREAK` messages can be controlled, a smaller size presents no problem.

In conventional microcode, `rmonThread` used this `SP_BREAK` message. Originally, the message was prepared for microcode `BREAK POINT` processing when using the GameShop DEBUGGER. This function currently is not used significantly, so it was left up to the user. As a result, when `rmonThread` is not used, no problem occurs. When it is used, note that the `SP_BREAK` message queue must be set after creating or starting `rmonThread` (execute `osStartThread` to `rmonThread`).

`gSPZWaitSignal` (Gfx *gp, zSignal *sig, u32 param)

`sig`	pointer to Signal buffer
`param`	Signal value (u32)

This GBI waits until the CPU Signal value exceeds the `param` value. Since the Signal value from the CPU is updated through an RDRAM buffer, that buffer must be contained in the application itself. During execution of this GBI, the RSP determines whether or not the CPU has rewritten the buffer's Signal. If so, the Signal buffer on the RDRAM is DMA transferred to DMEM and compared to the `param`.

The following is a macro for rewriting the CPU's Signal value.

```
GZ_SENSIGNAL (zSignal *sig, u32 val)

sig  pointer to Signal buffer
val  new Signal value
```

After the Signal value is rewritten to `val`, notice that the change that has occurred is sent to the RSP.

Since the Signal value is an unsigned 32-bit variable, the smallest value is 0.

So far in the microcode, the Display List is handed over to the RSP after it is complete. In other words, the RSP cannot process until the Display List has been completely created. However, even if the Display List is not completely created, this GBI can send any created portion to the RSP, i.e., the RSP can be made to wait until the rest of the Display List is created. When this `gSPZWaitSignal` and the earlier output `gSPZSendMessage` are combined, simple synchronicity occurs between CPU and RSP processing, demonstrating the great power of serial processing of the Display List.

Chapter 6 Compatibility With Other Microcodes

About GBIs

Z-Sort Microcode is not compatible with other Fast3D-compatible microcodes. However, some GBIs will be shared to allow switching by the microcode and self-loading of the F3DEX system. This section explains those GBIs that will likely belong to both microcodes.

The names of the GBIs explained here basically have the new prefix `gSPZ` instead of the corresponding prefix `gSP` of the GBI macro in F3DEX.

Z-Sort Microcode GBIs include a subset of F3DEX GBI Level 2. This F3DEX GBI Level 2 is a new and improved GBI set offering faster RSP processing speeds in F3DEX Microcode and will be adopted in the upcoming F3DEX Microcode release.

As a result, Level 2 is not compatible at the binary level with the GBIs adopted in F3DEX Microcode Version 1.23 or earlier. Thus, performing such processing as the microcode and self-loading in the F3DEX microcode system is difficult.

Since Z-Sort Microcode uses F3DEX GBI Level 2, when using Z-Sort Microcode, `F3DEX_GBI_2` must be defined by the `#define` statement or compile option D.

Note: At the present time, `F3DEX_GBI` must also be defined.

Common GBIs

`gSPZSegment`	Sets segment
`gSPZPerspNormalize`	Sets perspective correction value

`gSPZSegment` (Gfx *gp, u32 seg, u32 base)

`seg`	segment number (0~15)
`base`	segment base address

This GBI sets the segment. Although processing by either the Main DL or a Sub DL is possible, when the same segment number has been rewritten in the Main DL or a Sub DL, problems can be expected when parallel processing is started. To avoid these problems, try as much as possible not to overlap the segments to be used in the Main and Sub DL.

`gSPZPerspNormalize` (Gfx *gp, u16 persp)

`persp`	pass correction value

This GBI sets the perspective correction value. It is the same as the `gSPPerspNormalize` GBI in F3DEX.

Chapter 7 CPU Support Library

In Z-Sort Microcode, building plane data from the vertex data on the screen, i.e., ZObject data, depends on the CPU. Using arithmetic operation GBI commands, 3D coordinate vertices can be transformed into screen coordinate vertices. The CPU's role is to connect these vertices to build polygons. The CPU performs other processing as well and, therefore, a CPU library must be created by the user to perform this processing. The library used in the sample program `cubes-1` is explained below to provide a sample library.

- Multiply model matrix by perspective transformation matrix
- Calculate coordinate transformation/perspective transformation/screen depth for model vertices
- Determine whether there are vertices in the screen
- Determine clipping/back plane
- Construct ZObject data
- Create ZObject list

Chapter 8 Sample Programs

The sample programs are installed under the `/usr/src/PR/gZ-Sort` directory.

`zonetri/`

This displays one quadrangle and is the simplest application of Z-Sort.

`cubes-1`

A wide variety of polygons can be drawn using Z-Sort Microcode. The general-purpose library `libZ-Sort` is created and data is sent to it for drawing. Near clipping and other processes are performed in the library. Its 2-pass processing, however, hinders performance.