PlayStation®2 EE Library Overview
Release 2.4.2

Graphics Libraries

Publication date: December 2001

Sony Computer Entertainment Inc.
1-1, Akasaka 7-chome, Minato-ku
Tokyo 107-0052, Japan

Sony Computer Entertainment America
919 E. Hillsdale Blvd.
Foster City, CA 94404, U.S.A.

Sony Computer Entertainment Europe
30 Golden Square
London W1F 9LD, U.K.

Summary Table of Contents

About This Manual

This is the Runtime Library Release 2.4.2 version of the *PlayStation®2 EE Library Overview - Graphics Libraries* manual.

The purpose of this manual is to provide overview-level information about the PlayStation®2 EE graphics libraries. For related descriptions of the PlayStation®2 EE graphics library structures and functions, refer to the *PlayStation®2 EE Library Reference - Graphics Libraries.*

Changes Since Last Release

Chapter 1: High Level Graphics Library Service Function Groups

- The description of the packet buffer has been changed in the "Memory Consumption" section of "DMA Service Functions."
- The following items have been added to "DMA Service Functions."

  Dynamic/Static

  sceHiDMAWait

  Double Buffering
- The description of "ID problem: ID conflict restrictions" has been removed from the "Restrictions" section of "DMA Service Functions."

Chapter 5: High Level Graphics Plugin Library

- A description of the Tim2 plugin has been added to "Library Overview."
- A "Tim2 Plugin" section has been added to "Functional Overview for Each Plugin."

Related Documentation

Library specifications for the IOP can be found in the *PlayStation®2 IOP Library Reference* manuals and the *PlayStation®2 IOP Library Overview* manuals.

Note: the Developer Support Web site posts current developments regarding the Libraries and also provides notice of future documentation releases and upgrades.

Typographic Conventions

Certain Typographic Conventions are used throughout this manual to clarify the meaning of the text:

Convention	Meaning
`courier`	Indicates literal program code.
italic	Indicates names of arguments and structure members (in structure/function definitions only).
medium bold	Indicates data types and structure/function names (in structure/function definitions only).
blue	Indicates a hyperlink.

Developer Support

Sony Computer Entertainment America (SCEA)

SCEA developer support is available to licensees in North America only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
In North America:	*In North America:*
Attn: Developer Tools Coordinator Sony Computer Entertainment America 919 East Hillsdale Blvd. Foster City, CA 94404, U.S.A. Tel: (650) 655-8000	E-mail: PS2_Support@playstation.sony.com Web: http://www.devnet.scea.com/ Developer Support Hotline: (650) 655-5566 (Call Monday through Friday, 8 a.m. to 5 p.m., PST/PDT)

Sony Computer Entertainment Europe (SCEE)

SCEE developer support is available to licensees in Europe only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
In Europe:	*In Europe:*
Attn: Production Coordinator Sony Computer Entertainment Europe 30 Golden Square London W1F 9LD, U.K. Tel: +44 (0) 20 7859-5000	E-mail: ps2_support@scee.net Web: https://www.ps2-pro.com/ Developer Support Hotline: +44 (0) 20 7859-5777 (Call Monday through Friday, 9 a.m. to 6 p.m., GMT)

Chapter 1: High Level Graphics Library Service Function Groups

Library Overview

SCEI provides the following basic service function groups for developers of high level graphics library plugins.

- DMA service functions (HiG DMA Service) for creating and managing DMA packets
- Memory management service functions (HiG Memory Service) for managing the heap area allocated by the user
- GS management service functions (HiG GS Service) for configuring and managing the drawing and display environments that used libgraph, management of GS register contents and settings, and management of GS local memory.

Although these service function groups are provided as part of the high level graphics library, they are basically designed so that they can operate independently of the library.

Related Files

Since the service function groups are provided as part of the high level graphics library, the related files are the same as those of the high level graphics library.

Table 1-1

Category	File
Library file	libhig.a
Header file	libhig.h

Memory Management Service Functions

Overview

HiG provides a series of memory management service function groups to enable user programs to understand the memory area that is used by libhig and libhip. These function groups are provided to ensure that plugins do not consume the memory heap arbitrarily.

Therefore, each plugin must be designed so that the required memory area is obtained by this memory management service function group. Also, the amount of memory that is used must be publicly presented as a specification.

Errors

Most of the memory management service functions, which differ from the series of HiG functions because they emphasize usage efficiency or execution efficiency, have not been designed to return an sceHiError type error. The specifications of Alloc, Free, Align, Realloc, and Calloc are almost identical to the specifications of the corresponding functions of the general library.

Therefore, when the memory management service functions are used, the caller must check whether or not the allocated memory is appropriate.

If an inappropriate return value is received, an insufficient heap size condition can be expected to occur.

Each plugin must return information indicating this condition as an error.

The operations performed when allocation fails or other conditions occur are shown below for each memory management service function.

sceHiMemInit()	sceHiErr type is returned
sceHiMemAlloc()	NULL is returned
sceHiMemFree()	No processing is performed
sceHiMemAlign()	NULL is returned
sceHiMemRealloc()	NULL is returned
sceHiMemCalloc()	NULL is returned

DMA Service Functions

Overview

The DMA service functions are a group of functions for preparing data transfers and providing accompanying buffer memory management support, so that the processing of the DMA transfers required for drawing graphics can be easily performed.

When DMA service functions are used to create transfer data (packet chain), an ID is assigned to that data, and the DMA transfer can be performed by specifying that ID.

Part of the buffer memory used for transferring the data is allocated when the DMA service functions start to be used, and part is dynamically allocated internally. Since both buffer areas are allocated with memory allocation functions that are specified from the user program, the user can indirectly control the memory area that is used. When a plugin is created, a function for acquiring memory under HiG management is specified. However, if a normal memory function is specified, DMA service functions can be used for uses separate from HiG.

To work together with other DMA chain creation programs, service functions are also provided for managing DMA chains that have not been created by DMA service functions.

Explanation of Terms

Table 1-2

Term	Meaning
Packet	One DMA/VIF transfer request that was created by a DMA service function
Chain	A linked series of packets for consecutive DMA/VIF transfer requests that were created by multiple DMA service functions

Processing Flow

DMA service functions are generally used by carrying out the following steps.

1. **Initialization step**

   Use the sceHiDMAInit() function to specify the Packet Buffer size and the memory allocation function to be used for acquiring that amount of memory.

   This step must be executed before all other DMA service functions.

2. **Packet chain creation step**

   First call sceHiDMAMakeChain_Start(), then use a function such as sceHiDMAMake_LoadMicro() to

create a packet chain. Calling sceHiDMAMakeChain_End() completes one packet chain and assigns an ID that represents that packet chain. Repeat this for the required number of packet chains.

Packet chains are created in the Packet Buffer.

Also, since creating packet chains consumes a certain amount of CPU power, this step should be performed as much as possible during the initialization stage of each scene of the application.

3. **Packet chain transfer registration step**

   Use sceHiDMARegist() to register a packet chain to be transferred during that frame. You can select and register the required number of packet chains from among the packet chains that had been created beforehand.

   Although the DMA service function prepares for transferring packet chains in the order they were registered, the transfer is not yet performed.

4. **Packet chain transfer step**

   Registered packet chains are actually transferred by calling sceHiDMASend().

   A transferred packet chain is returned to an unregistered state.

5. **Repeat steps 3 and 4 for each frame.**

Memory Consumption

The two memory areas that are required by the DMA service functions are the Packet Buffer and the chain maintenance list.

Packet Buffer

The packet buffer is used to maintain packet chains that were created during the packet chain creation step.

The buffer is allocated when sceHiDMAInit() is called and has a size that was specified at the time of the call.

The buffer management method changed starting with library release 2.4.

In earlier releases, the buffer was managed as a single contiguous buffer, and the user called the Purge function after the packet chain was freed. However, since purging took time, a Slice management method was adopted.

This management method allocates memory in units of slices, where each slice has a fixed memory size (currently 64 qwords). Therefore, a memory area of at least one slice is required even for a chain consisting of only one packet.

When a slice becomes full during chain creation, a Next Tag is inserted at the end of the slice and a new slice is allocated. Unused slices can be allocated, and when a slice is freed, it is marked "unused." Therefore, the user does not need to call Purge regularly, as was required with the previous DMA Service.

Chain Maintenance List

This is an area of the list structure for maintaining information about created packet chains. It is dynamically allocated using the memory allocation function that was specified by sceHiDMAInit().

16 bytes (one qword) are required per packet chain.

Dynamic/Static

Starting with library release 2.4, two types of packet buffers can be used. These types are known as Dynamic and Static packet buffers.

With previous releases, a chain that was created between sceHiDMAMake_ChainStart() and sceHiDMAMake_ChainEnd() was created as a Static chain, packets could be maintained after they were transferred using sceHiDMASend(), and the chain could be managed using a chain ID.

Starting with library release 2.4, sceHiDMAMake_DynamicChainStart() and sceHiDMAMake_DynamicChainEnd() can be used to create a Dynamic chain, which can be used in cases when the packets change each time. A Dynamic chain is automatically deleted from the packet buffer after sceHiDMASend() is executed. Also, after a Dynamic chain is created, an action equivalent to sceHiDMARegist() is automatically performed to register the transfer. Note that since registration and deletion are performed automatically, a Dynamic chain has no chain ID.

SceHiDMAWait

The DMA Wait method changed due to the introduction of Dynamic chains.

With previous releases, a user-dependent DMA wait had to be used. However, beginning with library release 2.4, a DMA Wait can be set with sceHiDMAWait. Proper operation is not guaranteed if the DMA wait is performed without using sceHiDMAWait().

Double Buffering

DMA Double Buffering can be performed beginning with Release 2.4. Transfer registration can have two phases according to sceHiDMASwap().

The general DMA Double Buffering procedure is shown below.

```
void foo(void)
{
      while (1) {
        sceHiDMASend();         /* DMA Send */
        sceHiDMASwap();         /* Buffer Change */

       MY_DMA_MAKING_FUNCTION(); /* Packets are created and
                                    registered here */
                                 /* Transfer is performed in the
                                  background of creation processing */


        sceHiDMAWait();        /* DMA Wait */

          _MY_DATA_CHANGE_FUNCTION(); /* To change the data to be
                                         transferred, a Wait is
                                         executed */
                                   /* If the transfer has not
                                      completed, the data cannot be
                                      changed. A single state occurs
                                      here */
      }
}
```

In the example shown above, DMA transfer is performed in the background of _MY_DMA_MAKING_FUNCTION().

Also, when changing the data to be transferred, the data will not be synchronized unless the transfer of that portion is completed. Therefore, you must not change the data before the Wait has completed. In the example shown above, this corresponds to the position of _MY_DATA_CHANGE_FUNCTION().

Special Operations

Various modes can be set with regard to DMA/VIF. However, since setting and using these modes would be complicated if these actions were implemented as functions, the following internal variable of the DMA service functions has been made publicly available.

sceHiDMAState_t sceHiDMAState

This is an internal variable of the DMA service functions declared as extern in libhig.h.

DMA/VIF settings can be directly changed by directly overwriting values in this structure. The new settings are applied from the DMA service function immediately after they are overwritten.

For details, refer to the reference document.

Size Due to Unpack

The size after the transfer data is unpacked by VIF depends on the format.

If sceHiDMAState.pack_active (see higservice.h) is -1, this size is set so that the DMA service functions use the transfer data to its maximum extent.

Restriction

pack_active restriction

If at least 32 bits of data with 32-bit alignment is left over when unpacking with a usage size specification for which pack_active was used, it will end up being interpreted as VIF CODE. This phenomena is caused by DMA/VIF specifications.

GS Management Service Functions

Overview

Three types of service functions for using the GS are currently provided as HiG service functions: functions which are used to manage GS local memory, functions for configuring and managing GS registers, and functions for configuring the display environment, which used libgraph.

Functions

GS local memory management accepts a library area from a user, then uses a required frame buffer, depth buffer and texture buffer to draw and display in that area. This is a structure and function group that starts with the prefix sceHiGsMem.

GS register management manages GS registers in a unified manner, sets register contents, and creates, updates, and transfers transfer packets. GS register service functions are divided into three groups according to the type of register.

The display environment is a higher level library which uses libgraph functions. This is a structure and function group that starts with the prefix sceHiGsDisplay.

GS Register Management Service

Overview

The GS register management service functions are a group of service functions that allow GS register settings to be handled in a unified manner as ports, and that support the setting of GS registers so that the drawing environment can be easily configured.

These functions allow you to change simple GS registers, and to easily create sub-windows within a screen or draw to a texture area.

The GS register management service functions consist of three groups.

- Context register setting group

This group enables you to set registers corresponding to two contexts (registers having _1 and _2 appended to the end of the register name).

The names of these functions start with the prefix sceHiGsCtx.

- Environment register setting group

  This group enables you to configure general-purpose registers that do not correspond to a context.

  The names of these functions start with the prefix sceHiGsEnv.

- Display (special) register setting group

  This group has not been implemented yet.

Additional functions that perform common operations for services have names which start with the prefix sceHiGsService.

The GS register management service functions of each group are organized as follows.

Management Structure

One management structure is acquired for each port to hold the configured contents. The structure has four elements: the setting area, transfer area, transfer register specification, and internal management area.

Create and Delete Functions

These functions get, initialize, end and free the management structure for an individual port.

Set Functions

These functions allow values to be set in each register of a port, in the setting area within the management structure.

The SetReg function sets register contents directly. Other Set functions can set contents that span multiple registers or can specify a value in a form that is easier to use.

Update Function

This function reflects the values that were set for a port in the transfer area within the same management structure according to the transfer register specification.

Regist and Send Functions

The Regist function registers the transfer area of a port in order to transfer it to the GS, and the Send function immediately transfers the transfer area to the GS.

Swap Function

This function updates the transfer area directly for values that generally must change every frame such as double buffering or interlace settings. The environment register group does not have this function.

Standard Port

One pointer to the standard port is provided as a global variable for each group. It is created in the sceHiGsServiceInit() function.

SetDefault and GetDefault functions

One default port can be maintained for each group.

These functions set and get the default port.

Immediately after initialization, the standard port is set as the default port.

Other Functions

Several other service functions are also provided. For example, the Copy function exactly copies the contents of one port to another and sets everything the same other than the transfer area.

The FCache function writes back the cache line corresponding to the transfer area, to prepare for a DMA transfer.

The SetMax function enables you to change the maximum number of ports that can be acquired (initial value: 256). (the number of ports is unlimited for the environment register group.)

The following are also provided as generally applied to GS register management service functions.

sceHiGsServiceInit() and sceHiGsServiceExit()

These functions initialize and exit the GS register management service functions. The standard port for each group is also created during initialization.

Since sceHiGsServeceInit() is called within sceHiGsDisplaySize() and sceHiGsServiceExit() is called within sceHiGsDisplayEnd(), the user must not call sceHiGsServiceInit() and sceHiGsServiceExit() independently when sceHiGsDisplaySize() and sceHiGsDisplayEnd() are being used.

Other Functions

Functions are provided for independently setting the functions to be used for registering transfers.

Processing Flow

The general procedure for using GS register management service functions is as follows.

GS register management service function initialization

Use the sceHiGsDisplaySize() function or call the sceHiGsServiceInit() function directly to initialize all GS register management service functions. At this time, the standard port for the various groups is also created. When the sceHiGsDisplaySize() function is used, the standard port for the context register group will match the display settings.

Initialization step

Use the Create function to acquire one new port. Memory allocation and initialization processing are performed.

Configuration step

Use the Set or Copy function to configure the setting area within the management structure of the port. If there is an element for which the setting is to be changed, it should be changed at this time.

Transfer area update step

Use the Update function to set the transfer area within the management structure. The value that was set in the setting area and the transfer register settings are used at this time. If any setting is changed by the Set or Copy functions, the Update function must be called once before the Swap or Transfer steps.

Swap step (every frame if necessary)

Use the Swap function to overwrite the double buffering or interlace offset within the transfer area. Then, use the FCache function to write back the cache line if necessary.

Transfer step (every frame or each time drawing environment is switched)

Use the Regist function to register a DMA transfer to the GS, or use the Send function to immediately transfer the transfer area directly to the GS.

Completion step

Use the Delete function to perform port completion processing and to free memory.

Exit GS register management service functions

Use the sceHiGsDisplayEnd() function or call the sceHiGsServiceExit() function directly to exit all GS register management service functions.

Memory Consumed

The memory area that is dynamically allocated by the GS register management service functions is as follows.

sceHiMemAlign() is used internally for memory allocation.

Area Used by All Services

When the sceHiGsServiceInit() function is called to initialize the GS register management service functions, 1752 bytes are used, including that which is required for the standard port. This area is freed when completion processing is performed with the sceHiGsServiceExit() function.

Area Used by the Context Register Setting (sceHiGsCtx) Group's Port

Each time one port is created, 480 bytes are used.

Area Used by the Environment Register Setting (sceHiGsEnv) Group's Port

Each time one port is created, 32+(24*N) bytes are used, where N is the specified number of transfer registers. In other words, 56 bytes are used if there is only one transfer register. Since there are 9 standard ports, 248 bytes are used altogether. When all 20 registers are specified, which is the maximum number, 512 bytes will be used.

Registers that cannot be handled with GS register management service functions

Among the general-purpose registers of the GS, the following registers cannot be set with GS register management service functions: PRIM, RGBAQ, ST, UV, XYZF2, XYZ2, XYZF3, XYZ3, and HWREG.

In addition, the display (special) register configuration group for handling special registers is currently not implemented. The specifications for this group and when it will be made publicly available have not yet been decided.

Plugin Function Template

This section presents an example of using a plugin function template when developing a high level graphics library plugin.

The identification codes are assumed to have been set as follows.

Table 1-3

Repository:	TEMPLATE
Project:	MYPROJECT
Category:	MYCATEGORY
PluginID:	MYPLUGIN

Identification Codes

Define individual identification codes.

Since repository identifiers must be prevented from being duplicated, to make an internally created plugin available externally, you must report its repository identifier. SCEI provides official identifiers.

Any constant can be defined for the other identifiers.

```
#define TEMPLATE    255
#define MYPROJECT   1
#define MYCATEGORY  1
#define MYPLUGIN    1
```

Function Template

A plugin function template is shown below.

In this example, SCE_HIG_PRE_PROCESS gets flags from the user, and SCE_HIG_POST_PROCESS gets values of MyArgs type.

```
typedef struct{
     int    myint;
     float  myfloat;
}MyArgs;

typedef struct{
     int    myint;
     float  myfloat;
}MyStruct;

sceHiErr MyPlugin(sceHiPlug *plug, int process)
{
    MyStruct      *mystruct;
    MyArgs  *myargs;
    u_int   *myflag;

    switch(process){
      case SCE_HIG_INIT_PROCESS:

      /*     Allocate structure memory  */
      mystruct = (MyStruct *)sceHiMemAlign(16,sizeof(MyStruct) * 1);
      if(mystruct == NULL) return SCE_HIG_NO_HEAP;

      /*     Initialization processing  */
      ....

      /*     Stack structure     */
      plug->stack = (u_int)mystruct;

      break;

       case SCE_HIG_PRE_PROCESS:

      /*     Pop structure */
      mystruct = (MyStruct *)plug->stack;
      if(mystruct == NULL) return SCE_HIG_INVALID_DATA;

      /*     Get arguments from user as flags  */
```

```
        myflag = plug->args;

        /*     Process using flags       */
        .....

        break;

         case SCE_HIG_POST_PROCESS:

        /*     Pop structure */
        mystruct = (MyStruct *)plug->stack;
        if(mystruct == NULL) return SCE_HIG_INVALID_DATA;

        /*     Get arguments from user as a structure  */
        myargs = (MyArgs *)plug->args;

        /*     Process using arguments    */
        .....

        break;

         case SCE_HIG_END_PROCESS:

        /*     Destroy structure   */
        mystruct = (MyStruct *)plug->stack;
        if(mystruct == NULL) return SCE_HIG_INVALID_DATA;
        sceHiMemFree((u_int *)Paddr(mystruct));
        plug->stack = NULL;

        break;

         default:
        break;
       }

       return SCE_HIG_NO_ERR;
}
```

Sample Data

Reflect the plugin in the data.

The following data is an example in which the template plugin is added at the end of the frame plugins.

```
head_top:
.word HiG_VERSION, 0, 0, (head_end - head_top)/16

.ascii "Frame\0\0\0\0\0\0\0"
.word Frame_plug_top - head_top

....

.ascii      "MyPlug\0\0\0\0\0\0"
.word My_plug_top - head_top

head_end:

My_plug_top:
.byte TEMPLATE, MYPROJECT, MYCATEGORY, PLUGIN_STATUS
.word MYPLUGIN | (MYREVISION<<24), 0, (My_plug_end -
My_plug_top)/16
```

```
 .byte 0, 0, 0, 0
 .word 0, 0, 0
 My_plug_end:


 Frame_plug_top:
 .byte SCE_HIP_COMMON, SCE_HIP_FRAMEWORK, SCE_HIP_FRAME,
 SCE_HIP_PLUGIN_STATUS
 .word SCE_HIP_FRAME_PLUG | (SCE_HIP_REVISION<<24), 0,
(Frame_plug_end -
 Frame_plug_top)/16

 ....

 .byte TEMPLATE, MYPROJECT, MYCATEGORY, SCE_HIG_PLUGIN_STATUS
 .word MYPLUGIN | (MYREVISION<<24), 0, (My_plug_end -
 My_plug_top)/16

 Frame_plug_end:
```

Sample Code

This section shows sample code of an application in which the above template function and data are used.

The call types on the application side use the same procedures as those used for the other plugin functions.

```
static sceHiPlugTable ptable[] = {
 {{SCE_HIP_COMMON, SCE_HIP_FRAMEWORK, SCE_HIP_MICRO,
SCE_HIG_PLUGIN_STATUS, SCE_HIP_MICRO_PLUG, SCE_HIP_REVISION},
  (sceHiErr *)sceHiPlugMicro },
       {{SCE_HIP_COMMON, SCE_HIP_FRAMEWORK, SCE_HIP_TEX2D,
SCE_HIG_PLUGIN_STATUS, SCE_HIP_TEX2D_PLUG, SCE_HIP_REVISION},
  (sceHiErr *)sceHiPlugTex2D },
       {{SCE_HIP_COMMON, SCE_HIP_FRAMEWORK, SCE_HIP_SHAPE,
SCE_HIG_PLUGIN_STATUS, SCE_HIP_SHAPE_PLUG, SCE_HIP_REVISION},
  (sceHiErr *)sceHiPlugShape },
       {{SCE_HIP_COMMON, SCE_HIP_FRAMEWORK, SCE_HIP_HRCHY,
SCE_HIG_PLUGIN_STATUS, SCE_HIP_HRCHY_PLUG, SCE_HIP_REVISION},
  (sceHiErr *)sceHiPlugHrchy },
       {{SCE_HIP_COMMON, SCE_HIP_FRAMEWORK, SCE_HIP_ANIME,
SCE_HIG_PLUGIN_STATUS, SCE_HIP_ANIME_PLUG, SCE_HIP_REVISION},
  (sceHiErr *)sceHiPlugAnime },
       {{SCE_HIP_COMMON, SCE_HIP_FRAMEWORK, SCE_HIP_SHARE,
SCE_HIG_PLUGIN_STATUS, SCE_HIP_SHARE_PLUG, SCE_HIP_REVISION},
  (sceHiErr *)sceHiPlugShare },


 };
 /*   Add template plugin to table            */
 {{TEMPLATE, MYPROJECT, MYCATEGORY, SCE_HIG_PLUGIN_STATUS, MYPLUGIN,
 MYREVISION}, (sceHiErr *)MyPlugin },

 };


 int main(){
      sceHiErr     err;
      sceHiPlug    *frameP, *tempP;
      u_int        myflag;
      MyArgs       myargs;
```

```
        /* Initialize memory management service functions */
        sceHiMemInit(memalign(64, HIG_HEAP_SIZE), HIG_HEAP_SIZE);

        /* Initialize DMA service functions:  Use HiG functions for
           malloc and free functions  */
        sceHiDMAInit(NULL, NULL, HIDMA_BUF_SIZE);

        /* Register plugin table */
        sceHiRegistTable(ptable, NTABLE);


        /* Get frame plugin */
        err = sceHiGetPlug((u_int *)DATA_ADDR, "Frame", &frameP);

        /* Get template plugin */
        err = sceHiGetPlug((u_int *)DATA_ADDR, "MyPlug", &tempP);


        /* Initialization Processing */
        err = sceHiCallPlug(frameP, SCE_HIG_INIT_PROCESS);

        while(1){

              /* Assign flag */
              tempP->args = flag;

              /* Template function pre-processing is called */
              err = sceHiCallPlug(frameP, SCE_HIG_PRE_PROCESS);

              /* Assign MyArgs type */
              tempP->args = (u_int *)(&myargs);

              /* Template function post-processing is called */
              err = sceHiCallPlug(frameP, SCE_HIG_POST_PROCESS);

              /* Drawing */
              sceHiDMASend();
        }

        /* Termination processing */
        err = sceHiCallPlug(frameP, SCE_HIG_END_PROCESS);
}
```

Chapter 2:
Basic Graphics Library

Library Overview

Functional Overview

libgp is a library that enables GS drawing to be easily performed.

When a GS primitive is drawn, libgp creates a procedure for setting the proper values in GS registers so that data can be transferred to the GIF. libgp creates "packets" corresponding to individual drawing functions so there is no need to be aware of the actual GS register manipulation sequence. In addition, the transfer order can be easily controlled by managing multiple packets as a "chain." This transfer order control mechanism can also be applied to Z-sorting and material sorting.

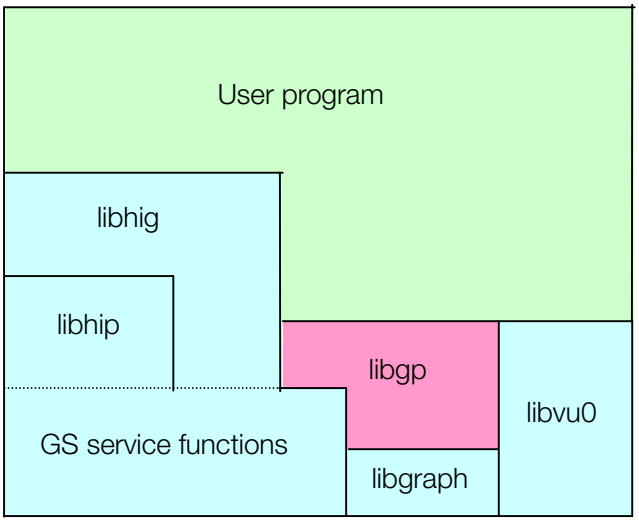
Other special features provided by libgp include transferring other kinds of memory data or DMA chains by assembling them together using "special packets." More specifically, data can be transferred to VU1 by assembling it together in the same chain.

Relationship to Other Graphics Libraries

As described above, libgp can be used for drawing once the drawing environment has been initialized and calculations such as perspective transformations have been completed.

libgraph or HiG GS services should be used to set up the drawing environment. Calculations such as perspective transformation can be performed independently with libvu0. If the high-level graphics library is used, all processing from calculation to drawing can be performed in a consistent manner.

Figure 2-1



Reference

libgp is located in a similar position as that of libgpu in the PlayStation libraries. However, among the functions that were part of libgpu, those functions that are included in libgraph, as well as those functions that are related to updating the drawing area or display area, and those related to fonts are not implemented in libgp.

Related Files

The following files are required to use libgp.

Table 2-1

Category	Filename
Header file	libgp.h
Library file	libgp.a

Sample Programs

The following libgp sample programs are provided for reference.

- **ee/sample/graphics/gp/balls/**
- **ee/sample/graphics/gp/filter/**
- **ee/sample/graphics/gp/multiwin/**
- **ee/sample/graphics/gp/withhig/**
- **ee/sample/graphics/gp/texsort/**
- **ee/sample/graphics/gp/zsort/**

Data Structures Used by libgp

Chain Management Structure and Ordering Table

A chain is a management mechanism that collects together multiple packets so they can be transferred as a single unit. It is defined as the sceGpChain structure shown below and is initialized by calling the sceGpInitChain() function.

```
typedef struct _sceGpChain {
    u_long128 *ot; /* Ordering table address */
    u_long128 *pKick;    /* DMA transfer start address */
    u_long128 *pEnd;
    int resolution;
} sceGpChain;
```

*ot is the ordering table that indicates the transfer order (drawing order) of packets within the chain. When a chain is initialized, "resolution" specifies the number of levels used to control the drawing order. When a packet is added to the chain, the drawing order for that packet within the chain is indicated by the level. During a transfer, packets that were added at lower levels are transferred first, and among packets at the same level, those that were added last are transferred first.

The *ot itself is also considered data that is transferred using DMA. Within the ordering table, pKick keeps the DMA transfer start address and pEnd keeps the DMA transfer end address.

Packet Types

Although many different types of packets are used, depending on the type of drawing that is performed, packets can be divided into three main types, namely drawing packets, environment packets, and special packets. These are defined below.

- Drawing packets: Packets corresponding to GS primitives.

- Environment packets: Packets that are used to prepare for drawing by performing alpha blending or texture control and texture or CLUT transfers.
- Special packets: Packets for including other DMA chains in chains or that are used to directly control GS registers.

When packets are created, suitable memory is allocated and a packet initialization function is used to set up required values. At this time, a general-purpose initialization function can be called with the type specified in an argument or a dedicated function for initializing a specific type of packet can be called.

Symbolic constants have been defined for specifying the type in the general-purpose initialization function, and these symbolic constants are also used for the packet type name.

Drawing Packets

Drawing packets correspond to individual GS primitives. Formats for these packets include the 64-bit wide R-format and the 128-bit wide P-format. The drawing packet type is specified by adding a subtype indicating the texture and shading of the corresponding GS primitive.

R-format (SCE_GP_PRIM_R) corresponds to the GIF's REGLIST mode. Although individual data values are comprised of 64 bits, bit field processing is necessary when setting values. This format is useful when data has been prepared in advance, or when you wish to reduce memory consumption or transfer time more than processing time. P-format (SCE_GP_PRIM_P), on the other hand, corresponds to the GIF's PACKED mode. With this format, 128 bits are needed to represent each data value, but the processing for setting a value is simpler. This format is useful when using values calculated by the EE Core or VU0, or when you want to reduce processing time more than memory consumption or transfer time.

The subtype is specified by combining the following three types of specifications.

Table 2-2

Subtype Indicating a GS Primitive	Remark
Point	Cannot be combined with "G" shading
Line	---
LineStrip	---
Tri	---
TriStrip	---
TriFan	---
Sprite	Cannot be combined with "G" shading

Table 2-3

Subtype Indicating Shading	Meaning
G (Gouraud)	Has color information for each vertex and performs Gouraud shading. Cannot be combined with "Point" or "Sprite" GS primitives.
F (Flat)	Has color information for each polygon and performs flat shading.
FM (Flat Monochrome)	Has color information of 0 or 1 for each packet and performs flat shading. Cannot be combined with "TS" texture mapping.

Table 2-4

Subtype Indicating Texture Mapping	Meaning
(No subtype specification)	No texture mapping
TU	UV value Texture mapping for which (texel coordinates: no perspective correction) are used.
TS	STQ value Texture mapping for which (texel coordinates: with perspective correction) are used. Cannot be combined with "FM" shading.

For example, the specification of an R-format packet for drawing a TriangleStrip with STQ-value texture mapping and Gouraud shading would be "SCE_GP_PRIM_R | SCE_GP_TRISTRIP_GTS." Either of the following function calls can be used to get the index of the Xyzf value for this packet.

```
sceGpIndexXyzf (SCE_GP_PRIM_R | SCE_GP_TRISTRIP_GTS, n); // General-purpose
function
sceGpIndexXyzfTriStripGTS (n); // Dedicated function
```

For a list of packet types and dedicated functions, see Section 4, "Packet Types and Dedicated Functions."

Environment Packets

Environment packets are used for performing set-up related to alpha blending or textures, and for transferring texture data. The following packet types are available (other types may be added in future releases).

Table 2-5

Packet Type (Structure)	Meaning
SCE_GP_ALPHAENV (sceGpAlphaEnv)	Alpha environment setting packet. Performs settings related to alpha values such as alpha blending.
SCE_GP_TEXENV (sceGpTexEnv)	Texture environment setting packet (without mipmap). Switches texture to be used.
SCE_GP_TEXENVMIPMAP (sceGpTexEnvMipmap)	Texture environment setting packet (with mipmap).
SCE_GP_LOADIMAGE (sceGpLoadImage)	Image transfer packet. Transfers image data (only texel data, only CLUT data, etc.) from main memory to GS local memory.
SCE_GP_LOADTEXELCLUT (sceGpLoadTexelClut)	Texture transfer packet with CLUT. Transfers texture data (set of texel data and CLUT data) from main memory to GS local memory.

The various environment packets are also defined as structures.

Special Packets

Special packets are used for including data in a chain that is not handled by the main libgp functions. Special packets are also used for independently setting GS registers. The following packet types are available.

Table 2-6

Packet Type (Structure)	Meaning
SCE_GP_CALL (sceGpCall)	call packet. Transfers an arbitrary DMA chain (one that ends with a RET tag).
SCE_GP_REF (sceGpRef)	ref packet. Transfers arbitrary data in memory.
SCE_GP_AD (sceGpAd)	General-purpose register setting packet. Sets an arbitrary value in an arbitrary GS general-purpose register.

Although arbitrary DMA chains or data can be transferred with these packets it goes without saying that the destination is responsible for the proper handling of the contents and format.

Packet Structures

The structure of each packet is represented by a structure that is unique for each type of packet. For information about these structures, refer to the appropriate reference documents. Each data structure contains a sequence of commands for the GS to which have been added an appropriate GIF tag and DMA tag. Hence, these data structures can be DMA-transferred asis to the GIF. Please see the DMA and GIF chapters of the "EE User's Manual" or the "GS User's Manual" to gain a better understanding of libgp functions and operations. Several specific points are given below.

Packets must be 16-byte aligned

Since packets are data that are DMA-transferred, they must be located at 16-byte boundaries. This is also true for ordering tables, so be sure to allocate memory carefully.

The contents of a packet cannot be changed during a transfer

If the contents of a packet are changed after a transfer begins, it is indeterminate whether the contents before or after the change will be transferred.

It may be possible to use surplus registers

Although userreg in a drawing packet is used as a packet color setting area for monochrome packets, in other cases (when no packet color is used, for a monochrome packet), userreg can be used for setting an arbitrary GS register.

For example, if it is used for setting the tex0 register, SCE_GP_TEXENV packets can be omitted.

High-order part of DMA tag

Although the high-order 64-bits of a DMA tag are empty, per the structure definition, libgp uses the high-order part of the DMA tag at the beginning of a packet. The third word of the DMA tag at the beginning of a packet (corresponding to bits 95-64) contains a VIFcode NOP, and the IMMEDIATE field is used for libgp internal management. Also, the fourth word (corresponding to bits 127-96) contains a VIFcode DIRECT for transfers via PATH2.

Drawing Procedure

The overall libgs drawing takes the form:

- Prepare packets and a chain
- Register packets in the chain
- Transfer the chain

Preparing Chains

The procedure for preparing a chain is as follows. The memory areas to be used for the sceGpChain structure and ordering table are allocated, and the resolution is specified when the chain is initialized. A program example is shown below.

```
sceGpChain chain;
u_long128 *ot;
ot = memalign(16, sceGpChkChainSize(resolution)*sizeof(u_long128));

sceGpInitChain(&chain, ot, resolution);
```

Since the ordering table is a data entity that is transferred with DMA, memory must be allocated with 16-byte alignment. Since the size depends on the resolution, sceGpChkChainSize() should be used to obtain the size, as shown above.

The resolution is a value indicating the number of levels to be used by that chain for controlling the drawing order. For example, in actual use, the resolution would be set beforehand to the number of textures to be used. Later, the texture number would be specified as the drawing order (level) when the packet was registered. Also, each texture environment setting and data transfer packet should be registered at the end of the chain at the corresponding level. If this is done, the texture and the GS primitives that use that texture are assembled together and transferred as a unit, which improves GS drawing efficiency. This method can also be used for Z-sorting. The resolution of a Z-value can be suitably determined from the contents of the associated image, and the chain is initialized in advance using that value for the resolution. Since packets are transferred in order of increasing level, if the level is specified for the corresponding Z-value when a drawing packet is registered, Z-sorting will be performed.

When there is no particular need to control the drawing order, the chain should be initialized with resolution set to 1, and all packets should be registered with the level set to 0. In this case, the order in which the packets are transferred will be the reverse of the order in which they were registered.

Preparing Packets

The sequence of steps for preparing individual packets is to allocate the respective memory areas, perform initialization corresponding to each packet type, then set parameters.

1. Allocating memory area

Since the size of a drawing packet or general-purpose register setting packet differs according to the contents, the memory area should be allocated as follows.

```
type = SCE_GP_PRIM_R | SCE_GP_TRISTRIP_GTU;  // Packet type
arg = packet-parameters;
ppacket = memalign(16, sceGpChkPacketSize(type,
arg)*sizeof(u_long128));
sceGpInitPrimR(ppacket, type, arg);
```

Since the size of an environment packet, call packet, or ref packet is fixed, memory can be allocated using the structure that corresponds to the packet type.

```
sceGpTexEnv packet;
sceGpInitPacket(&packet, SCE_GP_TEXENV, arg)
```

In any case, since the packet is a data entity that is transferred with DMA, memory must be allocated with 16-byte alignment.

2. Initializing packets

A packet can be initialized either by using a general-purpose initialization function and specifying the packet type as an argument or by using a dedicated function for each packet type.

The following program example uses a general-purpose initialization function to initialize the drawing packet that was allocated in the previous example.

```
sceGpInitPacket(ppacket, type, arg);
```

The following program example uses a dedicated initialization function to initialize the texture environment setting packet that was allocated in the previous example.

```
sceGpInitTexEnv(&packet, ctxt);
```

3. Setting parameters

For drawing packets, the procedure for setting parameters is somewhat complicated because the coordinates and color values for multiple vertices must be set in the packet as parameters. First, a parameter index is obtained. To do this, either a general-purpose index acquisition function or a dedicated function for each packet type can be used as follows. A dedicated function is better in terms of execution speed.

```
/* i-th color value in TriangleStrip packet */
index = sceGpIndexRgba(type, i); // General-purpose function
index = sceGpIndexRgbaTriStripGTU(i);   // Dedicated function
```

Next, the acquired index is used to set the parameters. To do this, either a general-purpose parameter setting function or a dedicated function for each packet type can be used as follows.

```
/* Set color values for TriangleStrip packet */
sceGpSetRgba(ppacket, type, index, r, g, b, a);      // General-purpose
function
sceGpSetRgbaTriStripGTU(ppacket, index, r, g, b, a); // Dedicated function

/* Set coordinate values for TriangleStrip packet */
sceGpSetXyzf(ppacket, type, index, x, y, z, f);      // General-purpose
function
sceGpSetXyzfTriStripGTU(ppacket, index, x, y, z, f); // Dedicated function
```

These functions are implemented as macros, so the type and value are not checked. Note that the contents of the packet and the memory area outside the packet will be destroyed if a function that does not match the packet type is called, if a function is called which casts to the wrong type, or if the wrong index is assigned.

Another method that can be used is to set the members of the structure directly, as shown below.

```
/* Set color values in bit fields of structure members */
ppacket->reg[index].rgba.R = r;
ppacket->reg[index].rgba.G = g;
ppacket->reg[index].rgba.B = b;
ppacket->reg[index].rgba.A = a;
```

```
/* Set u_long type color value directly */
u_long rgba[DATANUM];
ppacket->reg[index].ul= rgba[i];
```

For environment packets and special packets, dedicated parameter setting functions are provided for each packet type. The following program example sets parameters for a texture environment setting packet.

```
sceGpSetTexEnv(&packet, &texarg, ctxt);
```

A texture parameter structure has been defined for setting texture-related parameters, and functions for using Tim2 data to set parameters are also provided.

Registering Packets in a Chain

Once the packets have been prepared, they are registered in a chain as follows.

```
sceGpAddPacket(&chain, level, ppacket);
```

level is used to specify the drawing order for that packet. The range of values that can be specified for level are integers >= 0 that are less than the resolution. Packets registered at a smaller level are transferred first. However, only when a "reverse reset" is performed, packets registered at a larger level will be transferred first. In both cases, packets at the same level will be transferred such that those that were registered last will be transferred first.

Since link information in a packet is rewritten when the packet is registered in the chain, a packet that has already been registered cannot be reregistered. To repeatedly use a packet with the same contents, create a subchain and register it by using sceGpCallChain().

Transferring Chains

After the packets have been registered in the chain, the transfer can be performed as follows.

```
sceGpKickChain(&chain, path);
```

path indicates the transfer destination of the chain. Either SCE_GP_PATH3 (to the GIF) or SCE_GP_PATH2 (to VIF1) can be specified. When SCE_GP_PATH2 is used, the data to be sent to VU1 can be included in the chain in advance by using a call packet.

The sceGpKickChain() function, which begins the transfer, returns without waiting for the end of the transfer. Since a problem may occur if the packet contents are changed during transfer processing, use a function such as sceGsSyncPath() to detect the end of the transfer. When the transfer ends, sceGpResetChain() can be called to return the chain to its initial state (state in which no packets have been registered).

```
sceGsSyncPath(0, 0);       // Wait for end of transfer
sceGpResetChain(&chain);
```

Packet Types and Dedicated Functions

List of Packet Types

Table 2-7: Drawing packets

Packet Type	Description
SCE_GP_PRIM_R	R-format drawing packet
	Mainly uses GIF REGLIST mode
SCE_GP_PRIM_P	P-format drawing packet
	Uses GIF PACKED mode

Drawing packets have subtypes, which are also added to function arguments.

Example: R-format Gouraud Line (no texture)
(SCE_GP_PRIM_R | SCE_GP_LINE_G)

Table 2-8

Subtype	Description
SCE_GP_POINT_FM	Point monochrome
SCE_GP_POINT_FMTU	Point monochrome UV texture
SCE_GP_POINT_F	Point
SCE_GP_POINT_FTU	Point UV texture
SCE_GP_POINT_FTS	Point STQ texture
SCE_GP_LINE_FM	Line flat monochrome
SCE_GP_LINE_FMTU	Line flat monochrome UV texture
SCE_GP_LINE_F	Line flat
SCE_GP_LINE_FTU	Line flat UV texture
SCE_GP_LINE_FTS	Line flat STQ texture
SCE_GP_LINE_G	Line Gouraud
SCE_GP_LINE_GTU	Line Gouraud UV texture
SCE_GP_LINE_GTS	Line Gouraud STQ texture
SCE_GP_LINESTRIP_FM	LineStrip flat monochrome
SCE_GP_LINESTRIP_FMTU	LineStrip flat monochrome UV texture
SCE_GP_LINESTRIP_F	LineStrip flat
SCE_GP_LINESTRIP_FTU	LineStrip flat UV texture
SCE_GP_LINESTRIP_FTS	LineStrip flat STQ texture
SCE_GP_LINESTRIP_G	LineStrip Gouraud
SCE_GP_LINESTRIP_GTU	LineStrip Gouraud UV texture
SCE_GP_LINESTRIP_GTS	LineStrip Gouraud STQ texture
SCE_GP_TRI_FM	Triangle flat monochrome
SCE_GP_TRI_FMTU	Triangle flat monochrome UV texture
SCE_GP_TRI_F	Triangle flat
SCE_GP_TRI_FTU	Triangle flat UV texture
SCE_GP_TRI_FTS	Triangle flat STQ texture

Subtype	Description
SCE_GP_TRI_G	Triangle Gouraud
SCE_GP_TRI_GTU	Triangle Gouraud UV texture
SCE_GP_TRI_GTS	Triangle Gouraud STQ texture
SCE_GP_TRISTRIP_FM	TriangleStrip flat monochrome
SCE_GP_TRISTRIP_FMTU	TriangleStrip flat monochrome UV texture
SCE_GP_TRISTRIP_F	TriangleStrip flat
SCE_GP_TRISTRIP_FTU	TriangleStrip flat UV texture
SCE_GP_TRISTRIP_FTS	TriangleStrip flat STQ texture
SCE_GP_TRISTRIP_G	TriangleStrip Gouraud
SCE_GP_TRISTRIP_GTU	TriangleStrip Gouraud UV texture
SCE_GP_TRISTRIP_GTS	TriangleStrip Gouraud STQ texture
SCE_GP_TRIFAN_FM	TriangleFan flat monochrome
SCE_GP_TRIFAN_FMTU	TriangleFan flat monochrome UV texture
SCE_GP_TRIFAN_F	TriangleFan flat
SCE_GP_TRIFAN_FTU	TriangleFan flat UV texture
SCE_GP_TRIFAN_FTS	TriangleFan flat STQ texture
SCE_GP_TRIFAN_G	TriangleFan Gouraud
SCE_GP_TRIFAN_GTU	TriangleFan Gouraud UV texture
SCE_GP_TRIFAN_GTS	TriangleFan Gouraud STQ texture
SCE_GP_SPRITE_FM	Sprite monochrome
SCE_GP_SPRITE_FMTU	Sprite monochrome UV texture
SCE_GP_SPRITE_F	Sprite
SCE_GP_SPRITE_FTU	Sprite UV texture
SCE_GP_SPRITE_FTS	Sprite STQ texture

Table 2-9: Environment packets

Packet Type	Description
SCE_GP_ALPHAENV	Alpha environment setting
SCE_GP_TEXENV	Texture environment setting (without MIPMAP)
SCE_GP_TEXENVMIPMAP	Texture environment setting (with MIPMAP)
SCE_GP_LOADIMAGE	Texture transfer
SCE_GP_LOADTEXELCLUT	Texture transfer with CLUT

Table 2-10: Special packets

Packet Type	Description
SCE_GP_CALL	General-purpose DMA chain transfer
SCE_GP_REF	General-purpose data transfer
SCE_GP_AD	General-purpose register setting

Dedicated Functions and Structures for Drawing Packets

Drawing packet dedicated functions (common to all drawing packets)

Set parameter (vertex coordinates):	sceGpSetXyzf, sceGpSetXyz, sceSetXy
Set parameter (vertex/polygon color):	sceGpSetRgb, sceGpSetRgba
Set parameter (packet color):	sceGpSetRgbaFM
Set parameter (texel coordinates):	sceGpSetUv
Change drawing attribute:	sceGpSetFog, sceGpSetAa1, sceGpSetCtxt, sceGpSetAbe

SCE_GP_PRIM_R (R-format drawing packet)

Structure:	sceGpPrimR
Initialize packet:	sceGpInitPrimR
Set parameter (texture coordinates):	sceGpSetStq_R

SCE_GP_PRIM_P (P-format drawing packet)

Structure:	sceGpPrimP
Initialize packet:	sceGpInitPrimP
Set parameter (texture coordinates):	sceGpSetStq_P

Dedicated Index Acquisition Functions for Each Subtype

SCE_GP_POINT_FM (Point monochrome)

Get index (vertex coordinates):	sceGpIndexXyzfPointFM

SCE_GP_POINT_FMTU (Point monochrome UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfPointFMTU
Get index (texture coordinates):	sceGpIndexUvPointFMTU

SCE_GP_POINT_F (Point)

Get index (vertex coordinates):	sceGpIndexXyzfPointF
Get index (polygon color):	sceGpIndexRgbaPointF

SCE_GP_POINT_FTU (Point UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfPointFTU
Get index (polygon color):	sceGpIndexRgbaPointFTU
Get index (texture coordinates):	sceGpIndexUvPointFTU

SCE_GP_POINT_FTS (Point STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfPointFTS
Get index (polygon color):	sceGpIndexRgbaPointFTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqPointFTS_P
(For R-format)	sceGpIndexStPointFTS_R
	sceGpIndexQPointFTS_R

SCE_GP_LINE_FM (Line flat monochrome)

Get index (vertex coordinates):	sceGpIndexXyzfLineFM

SCE_GP_LINE_FMTU (Line flat monochrome UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineFMTU
Get index (texture coordinates):	sceGpIndexUvLineFMTU

SCE_GP_LINE_F (Line flat)

Get index (vertex coordinates):	sceGpIndexXyzfLineF
Get index (polygon color):	sceGpIndexRgbaLineF

SCE_GP_LINE_FTU (Line flat UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineFTU
Get index (polygon color):	sceGpIndexRgbaLineFTU
Get index (texture coordinates):	sceGpIndexUvLineFTU

SCE_GP_LINE_FTS (Line flat STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineFTS
Get index (polygon color):	sceGpIndexRgbaLineFTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqLineFTS_P
(For R-format)	sceGpIndexStLineFTS_R
	sceGpIndexQLineFTS_R

SCE_GP_LINE_G (Line Gouraud)

Get index (vertex coordinates):	sceGpIndexXyzfLineG
Get index (vertex color):	sceGpIndexRgbaLineG

SCE_GP_LINE_GTU (Line Gouraud UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineGTU
Get index (vertex color):	sceGpIndexRgbaLineGTU
Get index (texture coordinates):	sceGpIndexUvLineGTU

SCE_GP_LINE_GTS (Line Gouraud STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineGTS
Get index (vertex color):	sceGpIndexRgbaLineGTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqLineGTS_P
(For R-format)	sceGpIndexStLineGTS_R
	sceGpIndexQLineGTS_R

SCE_GP_LINESTRIP_FM (LineStrip flat monochrome)

Get index (vertex coordinates):	sceGpIndexXyzfLineStripFM

SCE_GP_LINESTRIP_FMTU (LineStrip flat monochrome UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineStripFMTU
Get index (texture coordinates):	sceGpIndexUvLineStripFMTU

SCE_GP_LINESTRIP_F (LineStrip flat)

Get index (vertex coordinates):	sceGpIndexXyzfLineStripF
Get index (polygon color):	sceGpIndexRgbaLineStripF

SCE_GP_LINESTRIP_FTU (LineStrip flat UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineStripFTU
Get index (polygon color):	sceGpIndexRgbaLineStripFTU
Get index (texture coordinates):	sceGpIndexUvLineStripFTU

SCE_GP_LINESTRIP_FTS (LineStrip flat STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineStripFTS
Get index (polygon color):	sceGpIndexRgbaLineStripFTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqLineStripFTS_P
(For R-format)	sceGpIndexStLineStripFTS_R
	sceGpIndexQLineStripFTS_R

SCE_GP_LINESTRIP_G (LineStrip Gouraud)

Get index (vertex coordinates):	sceGpIndexXyzfLineStripG
Get index (vertex color):	sceGpIndexRgbaLineStripG

SCE_GP_LINESTRIP_GTU (LineStrip Gouraud UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineStripGTU
Get index (vertex color):	sceGpIndexRgbaLineStripGTU
Get index (texture coordinates):	sceGpIndexUvLineStripGTU

SCE_GP_LINESTRIP_GTS (LineStrip Gouraud STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfLineStripGTS
Get index (vertex color):	sceGpIndexRgbaLineStripGTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqLineStripGTS_P
(For R-format)	sceGpIndexStLineStripGTS_R
	sceGpIndexQLineStripGTS_R

SCE_GP_TRI_FM (Triangle flat monochrome)

Get index (vertex coordinates):	sceGpIndexXyzfTriFM

SCE_GP_TRI_FMTU (Triangle flat monochrome UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriFMTU
Get index (texture coordinates):	sceGpIndexUvTriFMTU

SCE_GP_TRI_F (Triangle flat)

Get index (vertex coordinates):	sceGpIndexXyzfTriF
Get index (polygon color):	sceGpIndexRgbaTriF

SCE_GP_TRI_FTU (Triangle flat UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriFTU
Get index (polygon color):	sceGpIndexRgbaTriFTU
Get index (texture coordinates):	sceGpIndexUvTriFTU

SCE_GP_TRI_FTS (Triangle flat STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriFTS
Get index (polygon color):	sceGpIndexRgbaTriFTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqTriFTS_P
(For R-format)	sceGpIndexStTriFTS_R
	sceGpIndexQTriFTS_R

SCE_GP_TRI_G (Triangle Gouraud)

Get index (vertex coordinates):	sceGpIndexXyzfTriG
Get index (vertex color):	sceGpIndexRgbaTriG

SCE_GP_TRI_GTU (Triangle Gouraud UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriGTU
Get index (vertex color):	sceGpIndexRgbaTriGTU
Get index (texture coordinates):	sceGpIndexUvTriGTU

SCE_GP_TRI_GTS (Triangle Gouraud STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriGTS
Get index (vertex color):	sceGpIndexRgbaTriGTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqTriGTS_P
(For R-format)	sceGpIndexStTriGTS_R
	sceGpIndexQTriGTS_R

SCE_GP_TRISTRIP_FM (TriangleStrip flat monochrome)

Get index (vertex coordinates):	sceGpIndexXyzfTriStripFM

SCE_GP_TRISTRIP_FMTU (TriangleStrip flat monochrome UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriStripFMTU
Get index (texture coordinates):	sceGpIndexUvTriStripFMTU

SCE_GP_TRISTRIP_F (TriangleStrip flat)

Get index (vertex coordinates):	sceGpIndexXyzfTriStripF
Get index (polygon color):	sceGpIndexRgbaTriStripF

SCE_GP_TRISTRIP_FTU (TriangleStrip flat UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriStripFTU
Get index (polygon color):	sceGpIndexRgbaTriStripFTU
Get index (texture coordinates):	sceGpIndexUvTriStripFTU

SCE_GP_TRISTRIP_FTS (TriangleStrip flat STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriStripFTS
Get index (polygon color):	sceGpIndexRgbaTriStripFTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqTriStripFTS_P
(For R-format)	sceGpIndexStTriStripFTS_R
	sceGpIndexQTriStripFTS_R

SCE_GP_TRISTRIP_G (TriangleStrip Gouraud)

Get index (vertex coordinates):	sceGpIndexXyzfTriStripG
Get index (vertex color):	sceGpIndexRgbaTriStripG

SCE_GP_TRISTRIP_GTU (TriangleStrip Gouraud UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriStripGTU
Get index (vertex color):	sceGpIndexRgbaTriStripGTU
Get index (texture coordinates):	sceGpIndexUvTriStripGTU

SCE_GP_TRISTRIP_GTS (TriangleStrip Gouraud STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriStripGTS
Get index (vertex color):	sceGpIndexRgbaTriStripGTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqTriStripGTS_P
(For R-format)	sceGpIndexStTriStripGTS_R
	sceGpIndexQTriStripGTS_R

SCE_GP_TRIFAN_FM (TriangleFan flat monochrome)

Get index (vertex coordinates):	sceGpIndexXyzfTriFanFM

SCE_GP_TRIFAN_FMTU (TriangleFan flat monochrome UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriFanFMTU
Get index (texture coordinates):	sceGpIndexUvTriFanFMTU

SCE_GP_TRIFAN_F (TriangleFan flat)

Get index (vertex coordinates):	sceGpIndexXyzfTriFanF
Get index (polygon color):	sceGpIndexRgbaTriFanF

SCE_GP_TRIFAN_FTU (TriangleFan flat UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriFanFTU
Get index (polygon color):	sceGpIndexRgbaTriFanFTU
Get index (texture coordinates):	sceGpIndexUvTriFanFTU

SCE_GP_TRIFAN_FTS (TriangleFan flat STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriFanFTS
Get index (polygon color):	sceGpIndexRgbaTriFanFTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqTriFanFTS_P
(For R-format)	sceGpIndexStTriFanFTS_R
	sceGpIndexQTriFanFTS_R

SCE_GP_TRIFAN_G (TriangleFan Gouraud)

Get index (vertex coordinates):	sceGpIndexXyzfTriFanG
Get index (vertex color):	sceGpIndexRgbaTriFanG

SCE_GP_TRIFAN_GTU (TriangleFan Gouraud UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriFanGTU
Get index (vertex color):	sceGpIndexRgbaTriFanGTU
Get index (texture coordinates):	sceGpIndexUvTriFanGTU

SCE_GP_TRIFAN_GTS (TriangleFan Gouraud STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfTriFanGTS
Get index (vertex color):	sceGpIndexRgbaTriFanGTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqTriFanGTS_P
(For R-format)	sceGpIndexStTriFanGTS_R
	sceGpIndexQTriFanGTS_R

SCE_GP_SPRITE_FM (Sprite monochrome)

Get index (vertex coordinates):	sceGpIndexXyzfSpriteFM

SCE_GP_SPRITE_FMTU (Sprite monochrome UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfSpriteFMTU
Get index (texture coordinates):	sceGpIndexUvSpriteFMTU

SCE_GP_SPRITE_F (Sprite)

Get index (vertex coordinates):	sceGpIndexXyzfSpriteF
Get index (polygon color):	sceGpIndexRgbaSpriteF

SCE_GP_SPRITE_FTU (Sprite UV texture)

Get index (vertex coordinates):	sceGpIndexXyzfSpriteFTU
Get index (polygon color):	sceGpIndexRgbaSpriteFTU
Get index (texture coordinates):	sceGpIndexUvSpriteFTU

SCE_GP_SPRITE_FTS (Sprite STQ texture)

Get index (vertex coordinates):	sceGpIndexXyzfSpriteFTS
Get index (polygon color):	sceGpIndexRgbaSpriteFTS
Get index (texture coordinates):	
(For P-format)	sceGpIndexStqSpriteFTS_P
(For R-format)	sceGpIndexStSpriteFTS_R
	sceGpIndexQSpriteFTS_R

Special-Purpose Functions and Structures for Environment Packets

SCE_GP_ALPHAENV (alpha environment setting)

Structure:	sceGpAlphaEnv
Initialize packet:	sceGpInitAlphaEnv
Set parameter:	sceGpSetAlphaEnv

SCE_GP_TEXENV (texture environment setting (without MIPMAP))

Structure:	sceGpTexEnv
Initialize packet:	sceGpInitTexEnv
Set parameter:	sceGpSetTexEnv
	sceGpSetTexEnvByTim2
	sceGpSetTexEnvByArgTim2

SCE_GP_TEXENVMIPMAP (texture environment setting (with MIPMAP))

Structure:	sceGpTexEnvMipmap
Initialize packet:	sceGpInitTexEnvMipmap
Set parameter:	(Currently not provided)

SCE_GP_LOADIMAGE (texture transfer)

Structure:	sceGpLoadImage
Initialize packet:	sceGpInitLoadImage
Set parameter:	sceGpSetLoadImage
	sceGpSetLoadImageByTim2
	sceGpSetLoadImageByArgTim2

SCE_GP_LOADTEXELCLUT (texture transfer with CLUT)

Structure:	sceGpLoadTexelClut
Initialize packet:	sceGpInitLoadTexelClut
Set parameter:	sceGpSetLoadTexelClut,
	sceGpSetLoadTexelClutByTim2
	sceGpSetLoadTexelClutByArgTim2

SCE_GP_CALL (general-purpose DMA chain transfer)

Structure:	sceGpCall
Initialize packet:	sceGpInitCall
Set parameter:	sceGpSetCall

SCE_GP_REF (general-purpose data transfer)

Structure:	sceGpRef
Initialize packet:	sceGpInitRef
Set parameter:	sceGpSetRef

SCE_GP_AD (general-purpose register setting)

Structure:	sceGpAd (since packet size is uncertain, cannot be used to allocate memory)
Initialize packet:	sceGpInitAd
Set parameter:	sceGpSetAd

Chapter 3:
GS Basic Library

Library Overview

libgraph is a basic library for controlling the Graphics Synthesizer (GS).

The library hides differences between different GS chip versions and can convert environment information needed for displaying and drawing the appropriate data structures that can be transferred to the GS. In addition, display and drawing functions performed by the GS can be controlled by adding appropriate tags to the data generated by libgraph functions and sending this data to the GS via the GIF.

Other features provided by libgraph include: synchronization control with Vsyncs and data transfers; support for double buffering in GS local memory; and light-weight image transfers.

Related Files

The following files are required for libgraph:

Table 3-1

Category	Filename
Library file	libgraph.a
Header file	libgraph.h

List of Structures

As described above, many of the libgraph functions can be categorized as either functions that set up data in structures or functions that make settings to the GS based on this data. The list below shows combinations of structures and functions.

Table 3-2

Structure	Setting function	Issuing function
sceGsDispEnv	sceGsSetDefDispEnv	sceGsPutDispEnv
sceGsDrawEnv1	sceGsSetDefDrawEnv	sceGsPutDrawEnv
sceGsDrawEnv2	sceGsSetDefDrawEnv2	sceGsPutDrawEnv
sceGsTexEnv	sceGsSetDefTexEnv	
sceGsTexEnv2	sceGsSetDefTexEnv2	
sceGsAlphaEnv	sceGsSetDefAlphaEnv	
sceGsAlphaEnv2	sceGsSetDefAlphaEnv2	
sceGsClear	sceGsSetDefClear	
sceGsClear	sceGsSetDefClear2	
sceGsDrawEnv1	sceGsSetHalfOffset	
sceGsDrawEnv2	sceGsSetHalfOffset2	
sceGsDBuff	sceGsSetDefDBuff	sceGsSwapDBuff
sceGsDBuffDc	sceGsSetDefDBuffDc	sceGsSwapDBuffDc
sceGsLoadImage	sceGsSetDefLoadImage	sceGsExecLoadImage
sceGsStoreImage	sceGsSetDefStoreImage	sceGsExecStoreImage

Sample Programs

Libgraph functions are used by many sample programs. See, for example, sce/ee/sample/basic3d.

Usage Notes

Sample Initialization Sequence

When using libgraph, almost all programs will follow identical steps to perform initialization and double buffer switching. Typical program examples are shown below.

Initialization Sequence for FRAME Mode

```
// initialize DMAC
sceDmaReset(1);
// initialize VIF1, VU1, GIF
sceGsResetPath();
// initialize GS, libgraph
sceGsResetGraph(0, SCE_GS_INTERLACE, SCE_GS_NTSC, SCE_GS_FRAME);
// initialize display, drawing environment
sceGsSetDefDBuff(&db, SCE_GS_PSMCT32, 640, 224, SCE_GS_ZGEQUAL,
    SCE_GS_PSMZ24, SCE_GS_CLEAR);
```

Double Buffer Switching for FRAME Mode

```
// synchronize with drawing completion
if(sceGsSyncPath(0, 0) != 0){
     // Error handling
}
// VSync synchronization
oddeven = !sceGsSyncV(0);
// Half-pixel offset
sceGsSetHalfOffset((frame&1)?(&db.draw1):(&db.draw0),
        2048, 2048, oddeven);
FlushCache(WRITEBACK_DCACHE);
// double buffer switching
sceGsSwapDBuff(&db, frame);
```

Initialization Sequence for FIELD Mode

```
// initialize DMAC
sceDmaReset(1);
// initialize VIF1, VU1, GIF
sceGsResetPath();
// initialize GS, libgraph
sceGsResetGraph(0, SCE_GS_INTERLACE, SCE_GS_NTSC, SCE_GS_FIELD);
// initialize display, drawing environment
sceGsSetDefDBuff(&db, SCE_GS_PSMCT32, 640, 448,
     SCE_GS_ZGEQUAL, SCE_GS_PSMZ24, SCE_GS_CLEAR);
```

Double Buffer Switching for FIELD Mode

```
u_long  scandata[4] __attribute__((aligned(16))) = {
       SCE_GS_SET_TAG(1, 1, 0, 0, 0, 1), 0xe,
       0, SCE_GS_SCANMSK};
// synchronize with drawing completion
if(sceGsSyncPath(0, 0) != 0){
       // Error handling
}
// VSync synchronization
oddeven = !sceGsSyncV(0);
// SCANMASK switching
scandata[2] = ((oddeven&1)?2:3);
DPUT_GIF_FIFO(*(u_long128 *)&scandata[0]);
```

```
DPUT_GIF_FIFO(*(u_long128 *)&scandata[2]);
// double buffer switching
sceGsSwapDBuff(&db, frame);
```

Note: DMA Transfers and Cache Consistency

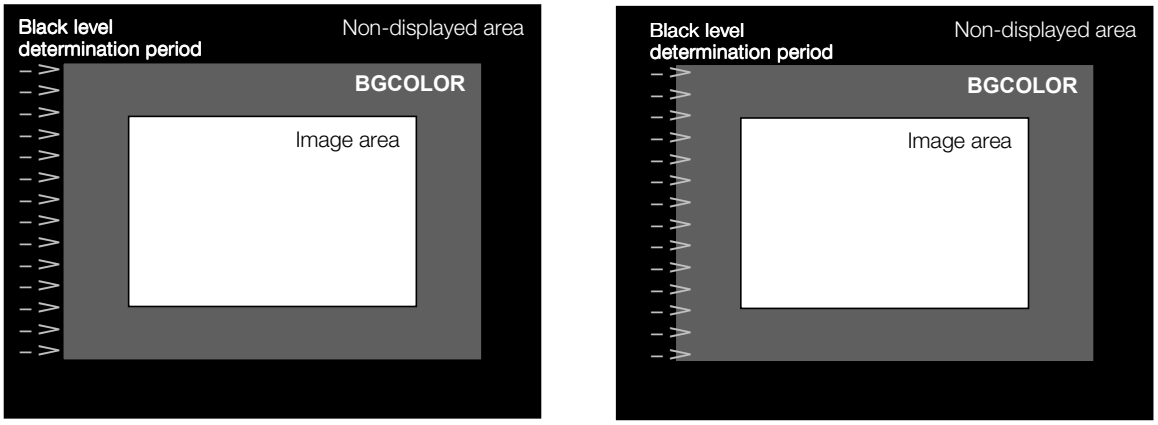
In many of the libgraph functions, when passing a pointer to a structure, data is set up in the structure and returned. This structure can be transferred directly to the GS by adding a GIFtag or a DMAtag, but make sure that the transfer is performed after the data that was set up has been written to memory. In other words, if the structure variable is in a cached area, the data cache must be flushed before the DMA transfer can be performed.

Restrictions: BGCOLOR Settings

Among the GS privileged registers, the value of BGCOLOR must be set to 0, i.e., the background color will be black. This restriction is necessary to make sure that the display is appropriate, regardless of the type of TV monitor being used. Here is a more detailed explanation as to why this restriction is necessary.

TV monitors have a period when the black level is determined, before the actual display period. The input signal is used during this period to determine the reference value for luminance. The length of this period varies from TV to TV, and in some cases the background color may be included as part of this period. In these cases, the reference value for luminance will be wrong unless the background color is black. This will cause the brightness of the entire screen to be very different (darker) compared to when another TV is used. To avoid this, the BGCOLOR register should always be set to 0.

Figure 3-1

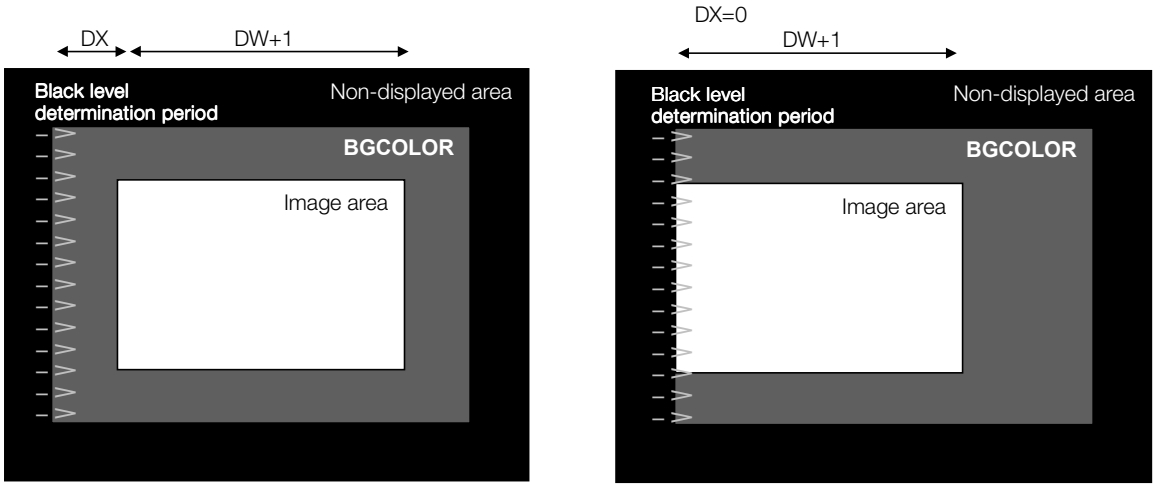


BGCOLOR has no effect on black level

BGCOLOR as an effect on black level

Similarly, setting the GS privileged register DISPLAYn.DX to a very small value such as 0 can also affect the black level.

Figure 3-2



The image area has an effect on the black level

The default value set up in DISPLAYn.DX is indicated in the description for the sceGsSetDefDispEnv function. If a custom setting is to be made, use a value of 536 or greater.

Support for PAL Mode

Basic settings related to video monitors must be changed to support PAL mode.

Uncommenting the '// #define GS_PAL_MODE' line in the following section in the eestruct.h header file will allow SCE_GS_SET_DISPLAY / SCE_GS_SET_DISPLAY_INTERLACE / SCE_GS_SET_DISPLAY_NOINTERLACE to serve as PAL mode settings.

```
#define SCE_GS_SET_DISPLAY SCE_GS_SET_DISPLAY_INTERLACE
// #define GS_PAL_MODE <- uncomment this line
#ifdef GS_PAL_MODE
// PAL, NOINTERLACE
(...)
#endif  // GS_PAL_MODE
```

Description of Structures

Reference Documents

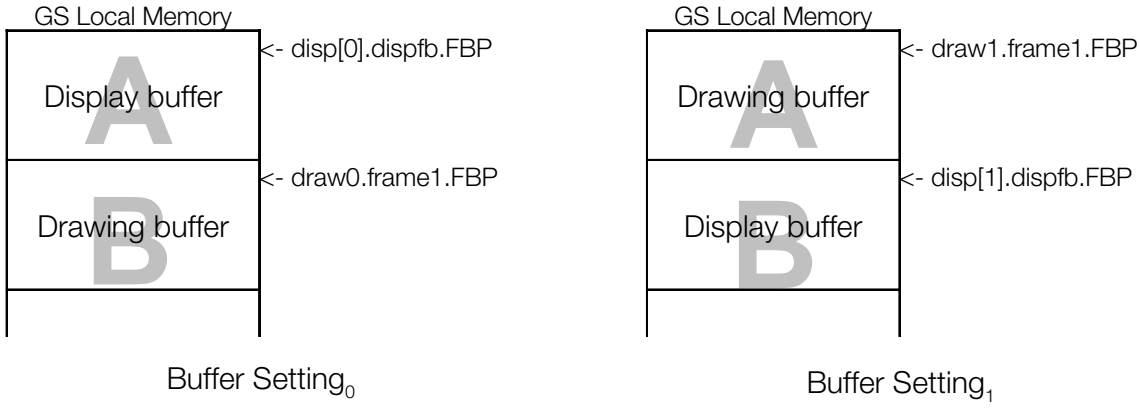
For reference material to help in understanding what libgraph does, see the "GS User's Manual" and "5. The GIF:GS interface" from the "EE User's Manual".

Double Buffering

libgraph supports double buffering in GS local memory to prevent the display of images that are currently being drawn.

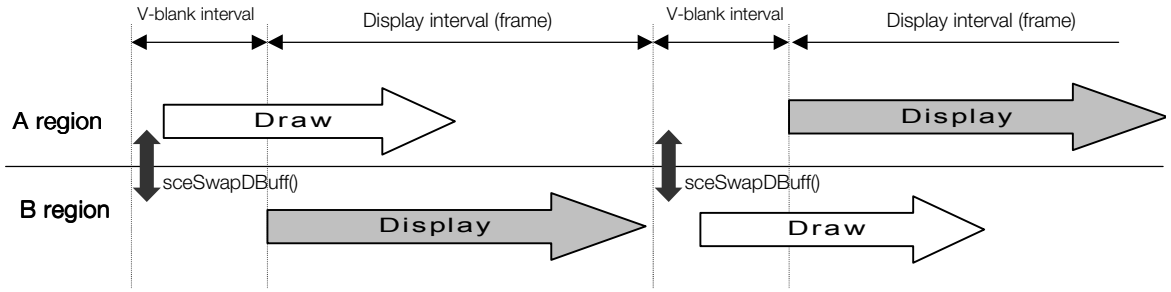
Double buffering is set up using sceGsSetDefDBuff() or sceGsSetDefDBuffDc(). The figure below shows double buffering (for single context) set up using sceGsSetDefDBuff().

Figure 3-3



When sceGsSwapDBuff() is called, buffer settings are swapped and the roles played by the A and B regions are swapped. At the start of a V-blank interval, the buffers are switched and, while the image drawn in the previous frame is displayed, the image for the next frame is drawn in the other buffer.

Figure 3-4



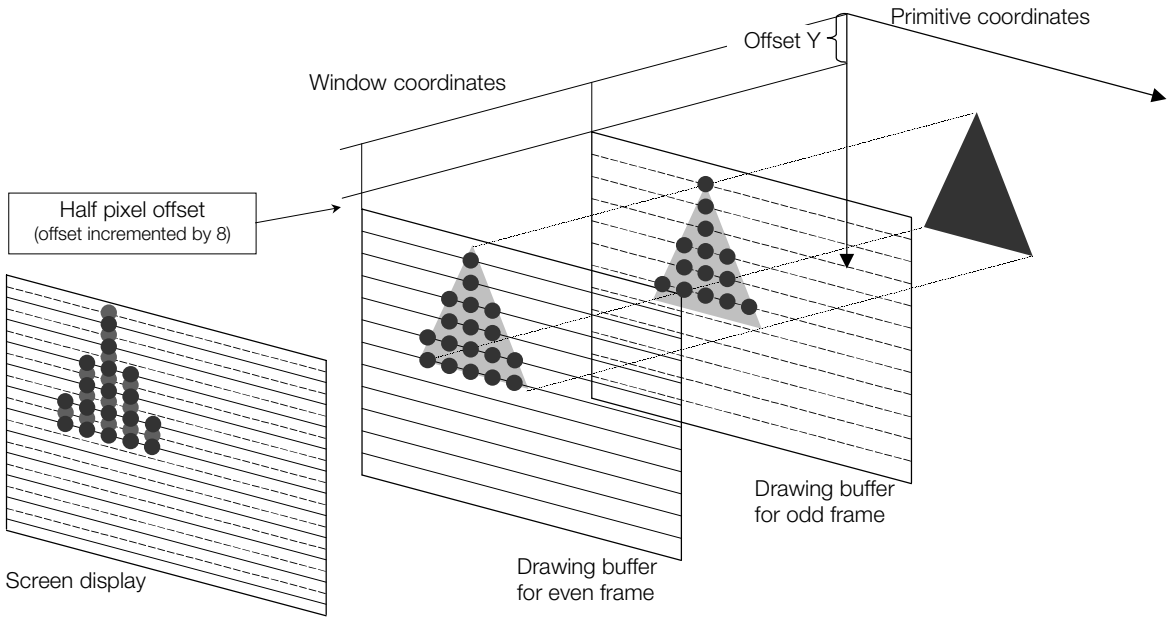
Half-pixel Offsetting

To handle interlaced mode, libgraph can provide a half-pixel offset for the drawing buffer.

In interlaced mode, odd frames and even frames are vertically offset from each other by 1/2 the scan line interval, with the overall image having a vertical resolution of 448 lines. Thus, if the drawing buffer mode is FRAME mode, i.e., the drawing buffers for the frames are separate, then drawing the images with 1/2 pixel offsets between the buffers will provide a vertical resolution of 448 lines.

To draw offset images, different offsets need to be used when converting from primitive coordinates to window coordinates. By specifying the halfoff parameter in sceGsSetHalfOffset() / sceGsSetHalfOffset2(), offsets can be incremented by 8, which corresponds to a half-pixel.

Figure 3-5


Primitive coordinates
Offset Y
Window coordinates
Half pixel offset
(offset incremented by 8)
Drawing buffer
for odd frame
Drawing buffer
for even frame
Screen display

Chapter 4:
High Level Graphics Library

Library Overview

The high level graphics library "HiG" is a graphics library that incorporates a plugin format. The library provides a platform that allows graphics functions that are provided as plugins to be called using a simple, uniform procedure.

SCEI provides plugins as part of the graphics plugin library "HiP". Middleware licensees are also expected to offer plugin libraries that contain independent plugin functions.

In addition, since the HiG library provides a group of service functions that support buffer management and other functions necessary to create plugins, programmers can create their own independent plugins as well.

Related Files

The following files are required to use the high level graphics library.

Table 4-1

Category	File
Library file	libhig.a
Header file	libhig.h

Sample Programs

The following subdirectory contains sample programs that use HiG. You can use these programs for reference.

- sce/ee/sample/graphics/hig

Library Functions

The high level graphics library provides application programmers with a number of service functions for managing multiple plugins as well as service functions related to data formats.

It also provides plugin developers with a high level memory management system and DMA packet management system for performing unified management on the high level graphics library side.

The library functions are classified into several types.

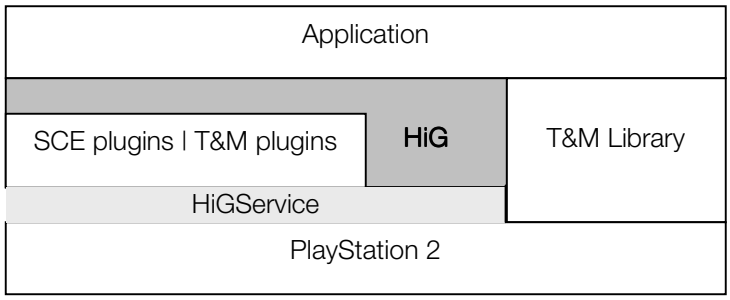
- "Basic functions" for managing plugins
- "Data format functions" for analyzing and obtaining data formats
- "Block manipulation functions" for performing detailed block operations
- "Memory management service functions" for managing memory
- "DMA service functions" for managing DMA packets
- "GS service functions" for managing GS

For details about individual functions, refer to the corresponding reference documents.

Plugin System

The following figure shows the software hierarchy when the high level graphics library is used.

Figure 4-1



Application:	User application
PlayStation 2:	Target hardware
T&M Library:	T&M library
HiG library:	HiG library
SCE plugins:	SCE-provided plugins
T&M plugins:	T&M-provided plugins
HiGService:	Service functions for plugins

Some advantages of using a plugin format are that individual functions can be extended, functions can be replaced, and only required functions need be incorporated.

Common API for Plugins

A plugin is used to aggregate the implementation of certain graphics functions. For example, a plugin might group together functions for describing a series of processes from data analysis to drawing. Although it is an aggregate of functions internally, only a single common-format plugin function which consolidates this set of functions can be accessed externally as an API. The format of this plugin function is as follows.

sceHiErr *plugin*(sceHiPlug **plug*, int *process*)

The first argument is a pointer to the plugin block (data structure used for passing arguments; this data structure will be described later). The second argument specifies the processing contents. The following four constants can appear for this argument.

Table 4-2

Constant	Processing Description
SCE_HIG_INIT_PROCESS	Plugin initialization processing
SCE_HIG_PRE_PROCESS	Pre-processing (first half of processing)
SCE_HIG_POST_PROCESS	Post-processing (second half of processing)
SCE_HIG_END_PROCESS	Plugin termination processing

The specific processing contents differ for each plugin. Many plugins generate graphics data as DMA packets in the post-processing stage. Processing for transferring the generated DMA packets to the GIF for drawing is not performed within the plugin. Since library functions for kicking transfers are provided separately, these are called explicitly from the user program.

Calling a Plugin

A plugin is not directly executed. A data structure called a "plugin block" is executed as a handler. The plugin block contains a pointer to the corresponding plugin function, an area used for passing arguments, a pointer to the data block to be processed by that plugin, and other information. This enables the required data and processing (plugin functions) for drawing a given scene to be collected together and described in advance so that the scene can be drawn by calling that plugin block.

Plugin blocks are differentiated from each other with individual names. To call a plugin block, first use sceHiGetPlug() to obtain a pointer from the name of the plugin block, then use sceHiCallPlug() to call the plugin block by specifying that pointer.

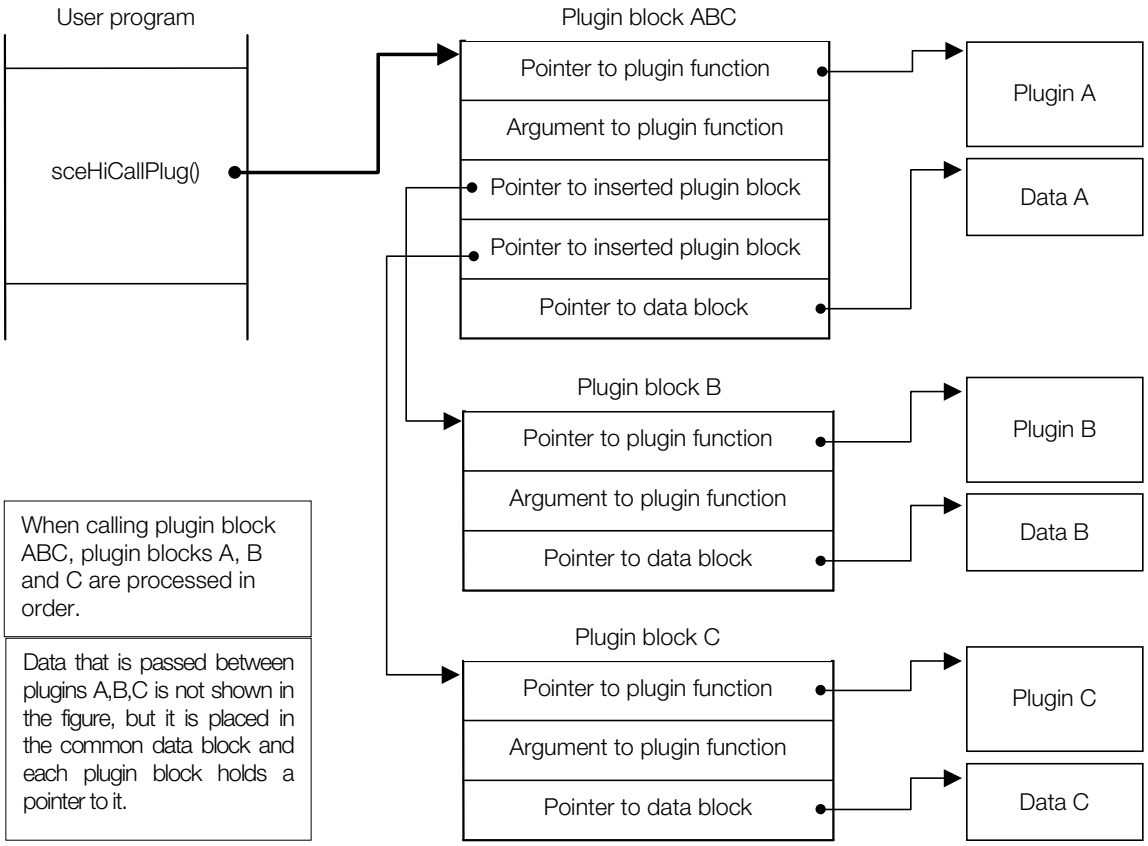
The syntax of sceHiCallPlug() is as follows.

sceHiErr sceHiCallPlug(sceHiPlug **plug*, int *process*)

Inserted Plugin Block

A plugin block is not restricted to just calling individual specific plugin functions, but can also contain a description that sequentially calls other plugin blocks. That is, other plugin blocks can be used as subroutines, and processing that is performed by multiple plugin blocks can be collected in a single plugin block and called only once from a user program. A plugin block that is called like a subroutine is referred to as an inserted plugin block.

Figure 4-2



Type Attribute of a Plugin

A plugin has the following four-level type attribute as an identifier.

Table 4-3

Level	Description
Repository	Plugin provider code (8 bits)
Project	Project code (8 bits)
Category	Category code (8 bits)
Plugin ID	Plugin-specific ID (24 bits)

This type attribute is used to associate a HiG data format plugin block / data block and a plugin. See the general description of the HiG data format for more information.

In addition, revision information is used to indicate the plugin version.

Memory Management Service Functions

The heap area that a plugin will use as a work area is allocated all at once by the user, assigned to the high level graphics library, and managed by the high level graphics library for use by each plugin. The user-side heap area is not destroyed. However, the user must use his discretion to choose and allocate the appropriate amount of memory to be used by the plugin.

Currently, the high level graphics library itself does not consume the heap. For the amount of the heap consumed by each plugin, refer to the respective document.

DMA Service Functions

The high level graphics library provides DMA service functions to efficiently generate DMA packets, which are required for drawing.

For details about DMA service functions, refer to the high level graphics library and service function documents.

GS Service Functions

These functions perform setting and management of drawing environments that used libgraph as well as setting and management of display environments, management of GS general-purpose registers and registers for 2 contexts, generation of PATH2 transfer DMA packets, and GS management service functions that manage GS local memory.

Refer to the documentation for high-level graphics library service functions for details on GS service functions.

Program Example

Processing Flow

A typical processing flow when the high level graphics library is used is shown below. However, since the processing flow depends on the user application, it will not necessarily be as shown below.

1. Initialize service functions
2. Register plugin functions
3. Analyze data format
4. Call plugin function initialization processing
   - The following is performed for each frame -
   - Set up communication between user and plugin function and overwrite data
   - Call plugin function pre-processing
   - Call plugin function post-processing
   - DMA transfer and draw
   - VSync
5. Call plugin function termination processing

Sample Program

```
#include <libhig.h> /* High level graphics library include file */
#include <libhip.h> /* Plugin library include file */

int main(void){
    sceHiPlug    *plug, *micro;      /* Handler to plugin block */
    sceHiMemInit(malloc(HIG_MEM_SIZE*4), HIG_MEM_SIZE*4);
          /* Initialize HiG Memory Service */
    sceHiDMAInit(NULL,NULL,BUFFER_SIZE);/* Initialize HiG DMA Service */
    sceHiRegistTable(ptable, NTABLE);/* Register plugin functions */
    sceHiParseHeader((u_int *)DATA); /* Analyze data format */

    sceHiGetPlug((u_int*)DATA, "Frame", &plug);
                                     /* Get "Frame" plugin block */
```

```
       sceHiGetPlug((u_int*)DATA, "Micro", &micro);
                                        /* Get "Micro" plugin block */

       sceHiCallPlug(plug, SCE_HIG_INIT_PROCESS));          /* Call
initialization processing */

         while(1){
                micro->args = micro_switch;    /* Communicate with "Micro"
plugin function */
                sceHiCallPlug(plug, SCE_HIG_PRE_PROCESS));  /* Call pre-
processing */
                sceHiCallPlug(plug, SCE_HIG_POST_PROCESS)); /* Call post-
processing */
              sceHiDMASend();            /* Perform DMA transfer and drawing
*/
              sceGsSyncPath(0,0)  /* Await end of DMA */
              odev = !sceGsSyncV(0);
            }
            sceHiCallPlug(plug, SCE_HIG_END_PROCESS);       /* Call
termination processing */
        }
```

Chapter 5:
High Level Graphics Plugin Library

Library Overview

SCEI provides "HiP" as a graphics plugin library that supports the high level graphics library "HiG." Up to now, HiP has implemented the graphics methodology that has been provided as graphics sample programs, in the form of plugins. The following plugins are provided by HiP. (There are plans to expand this list from time to time in the future.)

Table 5-1

Plugin Name (Function Name)	Function
Microcode plugin sceHiPlugMicro	Transfers perspective transformation and light source calculation microprogram to VU1
2D texture plugin sce HiPlugTex2D	Transfers texture data to the GS
Tim2 plugin sceHiPlugTim2	Transfers Tim2 format texture data to the GS
Shape plugin sceHiPlugShape	Transfers shape data to VU1 and draws according to the microprogram
Hierarchy plugin sceHiPlugHrchy	Generates matrix for associating hierarchy and coordinate transformation
Animation plugin sceHiPlugAnime	Interpolates between keyframes
Share plugin sceHiPlugShare	Generates shape from shared vertices and shared normal lines
Frame plugin (no function)	Virtual plugin for inserting other plugins to form plugin block
CLUT bump plugin sceHiPlugClutBump	Performs bump effect by calculating CLUT alpha from 256 normal line table and texture CLUT
FishEye plugin sceHiPlugFishEye	Performs scene rendering with a fish eye lens effect. Additional function plugin for Reflection or Refraction plugin.
Reflection plugin sceHiPlugReflect	Performs reflection mapping (environment mapping). Provides reflection on surface effect.
Refraction plugin sceHiPlugRefract	Performs refraction mapping. Provides effect that appears as if background is refracted.
ShadowMap plugin sceHiPlugShadowMap	Provides shadow effects due to shadow texture rendering and shadow mapping onto shadow texture background.
ShadowBox plugin sceHiPlugShadowBox	Bounding box processing for shadow texture. Sub-plugin of ShadowMap plugin
LightMap plugin sceHiPlugLightMap	Projects texture from light information to provide function for calculating lighting effect without light source calculations.

Since each individual plugin is packaged as an independent entity, a plugin block can be flexibly formed according to the scene configuration.

Related Files

The following files are required for the graphics plugin library.

Table 5-2

Category	Filename
Library file	libhip.a
Header file	libhip.h

Sample Programs

The following subdirectories contain sample programs that use the plugin library and sample data. You can use these programs and data for reference.

- sce/ee/sample/graphics/hig
- sce/ee/sample/graphics/hig/data

Data Formats

For an overview of the data formats used by the plugin library, refer to the graphics formats document (gformat.doc).

Data Converter

es2hig is provided for converting graphics data from the eS intermediate file format to HiG format. es2hig is included in the eS package and is provided as an artist tool.

Functional Overview of Each Plugin

This section gives a functional overview of each HiP plugin, as well as the type attributes of the plugin and its required data. For details, refer to the separate reference document.

The repository identifier of the type attribute is COMMON(1) for all plugins. Also, for the current release, the project identifier is FRAMEWORK(1) because the current release is based on the graphics framework sample.

Frame Plugin

The frame plugin is a virtual plugin that allows other plugins to be inserted to form a plugin block. The frame plugin provides no function itself. (Since it has no plugin function, the data block is also unnecessary.) To use several plugin blocks to perform a series of processes, you would first insert them in a frame plugin, then call that frame plugin from a user program.

Microcode Plugin

The microcode plugin generates DMA packets for transferring a perspective transformation and light source calculation microprogram to the VU1. There are functions for registering a microprogram and for selecting the microprogram to be transferred for each process. The actual transfer and execution of the microprogram are performed by calling sceHiDMASend().

2D Texture Plugin

The 2D texture plugin generates packets for transferring texture data to GS local memory. Texture transfer processing is performed by calling sceHiDMASend().

A texture can be made resident by setting resident=TRUE in sceHiPlugTex2dInitArg_t. Since GS memory management is not performed by the current 2D texture plugin, when resident and non-resident textures are used in common, memory must be allocated (by allocating the same Tbl) so that a suitable amount of GS memory is allocated for resident textures and the non-resident texture group shares the remaining GS memory (the amount of GS memory remaining can be found with sceHiGsMemRestSize()).

Example:

```
sceHiPlugTex2dInitArg_t resi1, resi2, nonresi;
sceHiPlug *resiplug1, *resiplug2, *nonresiplug1, *nonresiplug2,
*nonresiplug3;

resi1.tbl = sceHiGsMemAlloc(SCE_HIGS_PAGE_ALIGN, RESI1_TEX_SIZE);
resi1.resident = TRUE;
resi2.tbl = sceHiGsMemAlloc(SCE_HIGS_PAGE_ALIGN, RESI2_TEX_SIZE);
resi2.resident = TRUE;
nonresi.tbl = sceHiGsMemAlloc(SCE_HIGS_PAGE_ALIGN,
sceHiGsMemRestSize());
nonresi.resident = FALSE;
resiplug1->args = (u_int) &resi1;
resiplug2->args = (u_int) &resi2;
nonresiplug1->args = (u_int) &nonresi;
nonresiplug2->args = (u_int) &nonresi;
nonresiplug3->args = (u_int) &nonresi;
```

Tim2 Plugin

The Tim2 plugin generates packets for transferring Tim2-formatted texture data to GS local memory. The textures are transferred by calling sceHiDMASend().

A texture can be made resident by setting resident=TRUE in sceHiPlugTim2dInitArg_t. Since GS memory management is not performed by the current 2D texture plugin, when resident and non-resident textures are used in common, memory must be allocated (by allocating the same Tbl) so that a suitable amount of GS memory is allocated for resident textures and the non-resident texture group shares the remaining GS memory (the amount of GS memory remaining can be found with sceHiGsMemRestSize()). See the Tex2d plugin example shown above.

Shape Plugin

The shape plugin generates packets for transferring shape data to the VU1 and executing microcode, and drawing the shape. Transfer, execution and drawing processing is performed by calling sceHiDMASend(). Shape data includes material, geometry, primitives, vertices, normal lines, texture coordinates, and vertex colors.

Hierarchy Plugin

The hierarchy plugin performs cumulative coordinate transformations of the hierarchical structure of a model to generate a matrix for transforming from local coordinates of the model to world coordinates. The matrix generated here is transferred to VU1 memory by the shape plugin.

Animation Plugin

The animation plugin performs processing for interpolating hierarchical coordinate transformations between keyframes.

The plugin calculates the coordinate transformation parameters for translation, rotation, and zooming according to the assigned interpolation type and writes them in the hierarchy data.

Share Plugin

The share plugin generates a shape based on shared vertices or normal lines using separate coordinate transformations. This plugin has a pseudo-skin deformation effect. The generated shape data is transferred and drawn by the shape plugin as a single set of shape data.

CLUT Bump Plugin

The CLUT bump plugin calculates shading strength from 256 normal line tables and writes it as the texture CLUT alpha value. It produces a bump effect from variations in brightness due to alpha blending, using the texture as a base.

The texture CLUT alpha value is obtained from the shading parameters, normal line tables, and light vector assigned by the CLUT bump arguments and is calculated with the following expression.

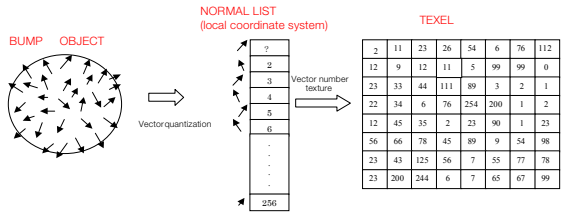
CLUT alpha = ambient alpha + diffuse alpha * (light.normal) + specular alpha *(light.normal)^shininess

Figure 5-1

Bump Mapping

- Bump Mapping using CLUT

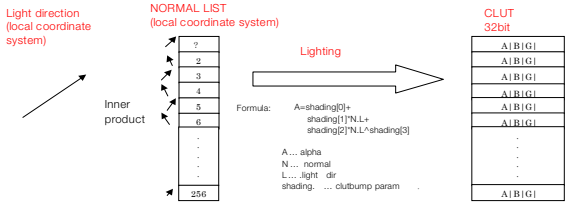
1. Prepare texture



Bump Mapping

- Bump Mapping using CLUT

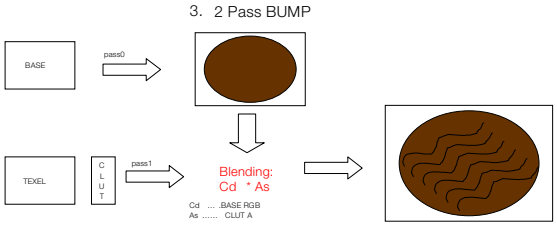
2. Prepare CLUT



Bump Mapping

- Bump Mapping using CLUT

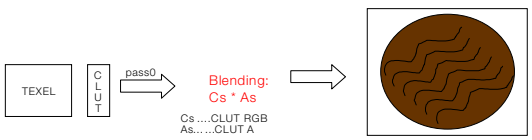
3. 2 Pass BUMP



Bump Mapping

- Bump Mapping using CLUT

4. 1 Pass BUMP



FishEye Plugin

The FishEye plugin generates a spherical texture from scene rendering using the projection transformation of a fish eye lens effect.

It uses the assigned FishEye plugin arguments to generate a transformation matrix. The transformation matrix performs projection transformation processing onto a spherical surface at a distance of length 1 from the viewpoint location.

Fish eye lens projection transformation expression:

M: Field of vision transformation matrix (world_view)

V: Vertex coordinates

texsize: Spherical surface size

ZA: (rmax*near-rmin*far)/(near-far)

ZB: (rmax*near*far-rmin*near*far)/(near-far)

V' = M * V

FishEye.x = 1/|V'| * texsize/2 + 2048

FishEye.y = 1/|V'| * texsize/2 + 2048

FishEye.z = 1/|V'| * ZB + ZA

A suitable drawing environment is needed in order to obtain a suitable rendering image.

Be sure to set correct values for the SCISSOR, XYOFFSET, and other registers.

Reflection Plugin

The reflection plugin uses the simplest and most general spherical reflection mapping among environment mapping techniques.

After calculating the reflection vector from the normal line vector and line-of-sight vector, the ST vector is calculated and a single spherical texture is mapped to reproduce a reflection.

There are two kinds of reflection mappings depending on the mapping texture: static reflection mapping and dynamic reflection mapping.

When the texture is fixed, static reflection mapping is used, which has no rendering cost. Dynamic reflection mapping is implemented by using the background rendering image as a texture. Dynamic reflection mapping can be done by using the FishEye plugin at the same time. However, it has a rendering cost since the background is temporarily rendered. It is effective when you want to implement a more realistic reflection mapping. A suitable drawing environment is required for reproduction. Be sure to configure register settings correctly so that the rendering image is used for the mapping texture. Specify DECAL for the texture function of the mapping texture.

Refraction Plugin

The Refraction plugin is implemented with almost the identical technique as is used with the Reflection plugin.

The differences are that a refraction vector is calculated and a refraction rate parameter (refract_index) is added.

As with reflection mapping, there are also two kinds of refraction mapping, depending on the mapping texture. These are static refraction mapping and dynamic refraction mapping.

Dynamic refraction mapping can be done by using the FishEye plugin at the same time.

A suitable drawing environment is required for reproduction. Be sure to configure register settings correctly so that the rendering image is used for the mapping texture. Specify DECAL for the texture function of the mapping texture.

ShadowMap Plugin

The ShadowMap plugin implements the most flexible shadow effect among several realtime shadowing techniques.

Conceptually, two kinds of processing are performed with two objects.

- Shadow object
- Shadow receiver

The shadow object is the object producing the shadow. It is the rendering target.

The shadow receiver is the object on which the shadow is projected. It is the mapping target.

The shadow object is rendered as a monochrome (black and white) texture. This is set as the shadow texture. Next, the shadow texture is texture mapped onto the shadow receiver. This is done by projection mapping from the light source to the shadow object. Then, subtraction blending is set for the drawing environment. Finally, the color of the shadow mapped shadow receiver is subtracted from the receiver that has been previously rendered in the frame buffer to implement the shadow effect.

For efficiency, the shadow object should have a ShadowBox, which is a bounding box. Make sure that this is passed to a ShadowMap plugin argument or the ShadowMap plugin block has it as data.

Only information for directional light source No. 0 is supported for shadows. Multiple light sources are not supported. When multiple light sources are required, ShadowMap processing should be repeated multiple times and blended.

ShadowBox Plugin

The ShadowBox plugin calculates the world coordinate values of the shadow object's bounding box. It is positioned as a sub-plugin of the ShadowMap plugin.

Only the root layer (first hierarchy data) is supported for the hierarchical structure for which world coordinate transformation is to be performed.

The ShadowBox indicates the bounding box formed from the diagonal line between the maximum and minimum values in the root locale.

When the shadow object has multiple hierarchical structures or animations, a bounding box that takes these into account must be configured. When a single shadow mapping is to be performed for multiple shadow objects, a bounding box that includes the multiple shadow objects must also be configured.

LightMap Plugin

The LightMap plugin implements lighting effects with textures.

A projection mapping from a light source is performed.

Only information for directional light source No. 0 is supported for the light source.

Although information from a directional light source is read, the same quantity of information as for a spotlight is required. The position, vector, color, intensity, and angle should be set.

Chapter 6:
VU0 Matrix Arithmetic Library

Library Overview

libvu0 is a library used to perform vector arithmetic and matrix arithmetic using VU0 macro instructions.

Functions that handle 4-dimensional float vectors include: addition/subtraction, scalar multiplication/division, normalization, interpolation, outer products, inner products, copying, and conversion to and from fixed-point 4-dimensional vectors. Functions that handle 4x4 matrices include general matrix arithmetic such as products, vector products, matrix transposition, matrix inversion, translation, and rotation, as well as operations specific to 3-D graphics processing such as clipping, perspective conversion, and calculations of world-view matrices/normal light matrices/light color matrices/screen view matrices/drop-shadow matrices, etc.

In addition, functions that initialize the VU0 / VIF0 are provided.

Parameter Handling

The libvu0 functions use different numbers and types of parameters based on the type of operations performed, but the function types are all void and the results are returned in the first parameter (vector or matrix). This is true for almost all the functions, with a few exceptions.

The VU0 essentially performs arithmetic with 128-bit (float x 4) values, so the parameters of the functions are essentially represented as 128-bit values. The vectors and matrices used in the parameters must be 128-bit aligned before a function is called.

Registers Used By Each Function

The following is a summary of the VF registers in the VU0 used by each function in libvu0 (starting with Release 1.2). In the table below, the registers marked with "O" will have their contents destroyed when the corresponding function is called, so the register should be saved by the calling program if necessary. Registers marked with "-" are not used so there is no need to save them.

Table 6-1

Function	VF04	VF05	VF06	VF07	VF08	VF09	VF10	VF11	VF12
AddVector	O	O	O	-	-	-	-	-	-
SubVector	O	O	O	-	-	-	-	-	-
MulVector	O	O	O	-	-	-	-	-	-
ScaleVector	O	O	O	-	-	-	-	-	-
ScaleVectorXYZ	O	O	-	-	-	-	-	-	-
DivVector	O	O	-	-	-	-	-	-	-
DivVectorXYZ	O	O	-	-	-	-	-	-	-
Normalize	O	O	O	-	-	-	-	-	-
InterVector	O	O	O	O	O	O	-	-	-
InterVectorXYZ	O	O	O	O	O	O	-	-	-
OuterProduct	O	O	O	-	-	-	-	-	-
InnerProduct	O	O	-	-	-	-	-	-	-
CopyVector	-	-	-	-	-	-	-	-	-
CopyVectorXYZ	-	-	-	-	-	-	-	-	-
ClampVector	O	O	O	-	-	-	-	-	-
FTOI4Vector	O	O	-	-	-	-	-	-	-
FTOI0Vector	O	O	-	-	-	-	-	-	-
ITOF4Vector	O	O	-	-	-	-	-	-	-
ITOF0Vector	O	O	-	-	-	-	-	-	-
ApplyMatrix	O	O	O	O	O	O	-	-	-
MulMatrix	O	O	O	O	O	O	-	-	-
TransposeMatrix	-	-	-	-	-	-	-	-	-
InversMatrix	O	O	O	O	O	O	-	-	-
TransMatrix	O	O	-	-	-	-	-	-	-
CopyMatrix	-	-	-	-	-	-	-	-	-
UnitMatrix	O	O	O	O	-	-	-	-	-
RotMatrixZ	O	O	O	O	O	O	-	-	-
RotMatrixY	O	O	O	O	O	O	-	-	-
RotMatrixX	O	O	O	O	O	O	-	-	-
RotMatrix	O	O	O	O	O	O	-	-	-
RotTransPers	O	O	O	O	O	O	O	-	-
RotTransPersN	O	O	O	O	O	O	O	-	-
CameraMatrix	O	O	O	O	O	O	-	-	-
NormalLightMatrix	O	O	O	-	-	-	-	-	-
LightColorMatrix	-	-	-	-	-	-	-	-	-
ViewScreenMatrix	O	O	O	O	O	O	-	-	-
DropShadowMatrix	O	O	O	O	O	O	-	-	-
ClipScreen	O	O	O	O	-	-	-	-	-
ClipScreen3	O	O	O	O	O	O	-	-	-
ClipAll	O	O	O	O	O	O	O	O	O

* Note: "sceVu0" at the start of function names has been omitted.