DMPGL 2.0 System API Specifications

Version 1.8

Digital Media Professionals Inc.

PROVISIONAL TRANSLATION

The content of this document is highly confidential and should be handled accordingly.

Table of Contents

1 Overview 14

2 Initialization API 15

2.1 API 15

2.1.1 DMPGL Initialization 15

2.1.2 DMPGL Finalization 16

2.1.3 Getting an Allocator 16

2.2 Allocator Information 16

3 Execution Control API 18

3.1 Command List Objects 18

3.1.1 3D Command Buffer 19

3.1.2 Command Requests 19

3.2 Executing Commands 21

3.2.1 Serial Execution Mode 21

3.2.2 Parallel Execution Mode 21

3.2.3 Synchronous Execution Mode 22

3.3 API 23

3.3.1 Generating Command List Objects 23

3.3.2 Deleting Command List Objects 23

3.3.3 Binding Command List Objects 23

3.3.4 Allocating Data Regions for Command List Objects 24

3.3.5 Executing Command List Objects 24

3.3.6 Stopping Command List Objects 25

3.3.7 Scheduling Stops for Command List Objects 25

3.3.8 Splitting the 3D Command Buffer 25

3.3.9 Clearing Command List Objects 26

3.3.10 Clearing Command List Objects and Filling Command Buffers 26

3.3.11 Registering Interrupt Handlers for Command Completion 26

3.3.12 Setting Parameters for Command List Objects 27

3.3.13 Getting the Parameters of Command List Objects 27

3.3.14 Checking for V-Sync Updates 29

3.3.15 V-Sync Synchronization 30

3.3.16 Registering the V-Sync Callback Function 30

3.3.17 Waiting for a Command List Object to Complete Execution 30

3.3.18 Transferring Data via DMA 30

3.3.19 Transferring Block Images with an Anti-Aliasing Filter 31

3.3.20 Image Transfer Requests 31

3.3.21 Setting the Timeout for Waiting to Complete Command List Object Execution 33

3.3.22 Updating Additive Blend Results Rendered with Gas Density Information 33
3.3.23 Transferring a Block Image That Is Converted into a Linear Image 34
3.3.24 Transferring a Linear Image That Is Converted into a Block Image 35
3.3.25 Transferring a Block Image 36
3.3.26 Filling Memory 37

4 Display Control API 40
4.1 Processing Flow from Rendering Through Display 40
4.1.1 Rendering 40
4.1.2 Transferring Rendered Results 40
4.1.3 Displaying 41
4.2 Specifying the Display Area 42
4.3 API 42
4.3.1 Generating Display Buffer Objects 42
4.3.2 Deleting Display Buffer Objects 42
4.3.3 Activating Display Targets 43
4.3.4 Binding Display Buffers 43
4.3.5 Allocating Display Buffers 43
4.3.6 Specifying the Display Area 44
4.3.7 Requesting Transfers of Rendered Results 44
4.3.8 Displaying Rendered Screens (Swapping) 45
4.3.9 Getting Parameters for Display Buffer Objects 46
4.3.10 Display Mode Settings 47

5 Command List Extended API 48
5.1 Saving and Reusing Command List Objects 48
5.1.1 Saving Commands 48
5.1.2 Using Saved Commands 49
5.2 Editing Commands 51
5.3 Other Features 51
5.3.1 Importing and Exporting Command Lists 51
5.3.2 Copying Command List Objects 51
5.3.3 3D Command Buffer Generation 51
5.3.4 Adding 3D Commands 52
5.4 API 52
5.4.1 Start Saving Command Lists 52
5.4.2 Stop Saving Command Lists 52
5.4.3 Using Saved Command Lists 53
5.4.4 Exporting Command Lists 54
5.4.5 Importing Command Lists 57
5.4.6 Getting Command List Information for Exported Data 57
5.4.7 Copying Command Lists 58

5.4.8 Checking the DMPGL State and Generating Commands 58
5.4.9 Updating the DMPGL State 59
5.4.10 Setting the Command Output Mode 59
5.4.11 Getting the Command Output Mode 60
5.4.12 Adding 3D Commands 60
5.4.13 Adding a Copied Command List 61
5.4.14 Getting the Updated DMPGL State 61
5.4.15 Invalidating DMPGL State Updates 62
5.5 State Flags 62
5.5.1 State Flag Types 62
5.5.2 State Flag Dependencies 65
5.5.3 Lookup Table Command Generation 65
5.6 DMPGL Functions That Generate Commands 66
5.7 3D Command Buffer Specifications 68
5.7.1 Basic Specifications 68
5.7.2 Single Access 69
5.7.3 Burst Access 69
5.8 PICA Register Information 70
5.8.1 Render Start Registers 70
5.8.2 Vertex Shader Floating-Point Registers 70
5.8.3 Vertex Shader Boolean Registers 72
5.8.4 Vertex Shader Integer Registers 72
5.8.5 Vertex Shader Starting Address Setting Registers 72
5.8.6 Registers That Set the Number of Input Vertex Attributes 73
5.8.7 Registers That Set the Number of Output Registers Used by the Vertex Shader 73
5.8.8 Registers That Set the Vertex Shader Output Mask 73
5.8.9 Registers That Set Vertex Shader Output Attributes 73
5.8.10 Clock Control Setting Registers for Vertex Shader Output Attributes 75
5.8.11 Vertex Shader Program Code Setting Registers 75
5.8.12 Registers That Map Vertex Attributes to Input Registers 76
5.8.13 Registers That Set Fixed Vertex Attribute Values 77
5.8.14 Registers for Vertex Attribute Array Settings 78
5.8.15 Other Setting Registers Related to the Vertex Shader 86
5.8.16 Texture Address Setting Registers 86
5.8.17 Render Buffer Setting Registers 87
5.8.18 Texture Combiner Setting Registers 88
5.8.19 Registers That Set Fragment Lighting 91
5.8.20 Texture Setting Registers 100
5.8.21 Registers for Gas Settings 109
5.8.22 Fog Setting Registers 112
5.8.23 Fragment Operation Setting Registers 113
5.8.24 Shadow Attenuation Factor Setting Registers 113

5.8.25 w Buffer Setting Registers ... 114
5.8.26 User Clip Setting Registers ... 114
5.8.27 Alpha Test Setting Registers ... 115
5.8.28 Framebuffer Access Control Setting Registers ... 115
5.8.29 Viewport Setting Registers ... 117
5.8.30 Depth Test Setting Registers ... 118
5.8.31 Logical Operation and Blend Setting Registers ... 119
5.8.32 Early Depth Test Setting Registers ... 121
5.8.33 Stencil Test Setting Registers ... 121
5.8.34 Culling Setting Registers ... 122
5.8.35 Scissoring Setting Registers ... 122
5.8.36 Color Mask Setting Registers ... 123
5.8.37 Block Format Setting Registers ... 124
5.8.38 Settings Registers Specific to the Rendering API ... 124
5.8.39 Settings Registers Specific to the Geometry Shader ... 129
5.8.40 Settings Registers When Reserved Geometry Shaders Are Used ... 132
5.8.41 Clearing the Framebuffer Cache ... 142
5.8.42 Commands That Generate Interrupts (Split Commands) ... 142
5.8.43 Settings Information for Otherwise Undocumented Bits ... 142
5.9 Code to Convert Formats for PICA Register Settings ... 144
5.9.1 Converting from float32 to float24 ... 144
5.9.2 Converting from float32 to float16 ... 145
5.9.3 Converting from float32 to float31 ... 145
5.9.4 Converting from float32 to float20 ... 146
5.9.5 Converting a 32-Bit Floating-Point Number into an 8-Bit Signed Fixed-Point Number with 7 Fractional Bits ... 146
5.9.6 Converting a 32-Bit Floating-Point Number into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits ... 147
5.9.7 Converting a 32-Bit Floating-Point Number into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits (Alternate Method) ... 148
5.9.8 Converting a 32-Bit Floating-Point Number into a 13-Bit Signed Fixed-Point Number with 8 Fractional Bits ... 148
5.9.9 Converting a 32-Bit Floating-Point Number into a 13-Bit Signed Fixed-Point Number with 11 Fractional Bits ... 149
5.9.10 Converting a 32-Bit Floating-Point Number into a 16-Bit Signed Fixed-Point Number with 12 Fractional Bits ... 150
5.9.11 Converting a 32-Bit Floating-Point Number into an 8-Bit Unsigned Fixed-Point Number with No Fractional Bits ... 151
5.9.12 Converting a 32-Bit Floating-Point Number into an 11-Bit Unsigned Fixed-Point Number with 11 Fractional Bits ... 151
5.9.13 Converting a 32-Bit Floating-Point Number into a 12-Bit Unsigned Fixed-Point Number with 12 Fractional Bits ... 152

5.9.14 Converting a 32-Bit Floating-Point Number into a 24-Bit Unsigned Fixed-Point Number with 24 Fractional Bits 153
5.9.15 Converting a 32-Bit Floating-Point Number into a 24-Bit Unsigned Fixed-Point Number with 8 Fractional Bits 153
5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer 154
5.9.17 Alternate Conversion from a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer 154
5.9.18 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Signed Integer 154
5.9.19 Converting a 16-Bit Floating-Point Value into a 32-Bit Floating-Point Value 154
5.10 Command Cache Restrictions and Precautions 155
5.11 PICA Register List 155

6 Error Codes 186

Code

Code 5-1 Sample 32-Bit Floating-Point Input 71
Code 5-2 Sample 24-Bit Floating-Point Input 72
Code 5-3 Sample Vertex Shader Definitions 74
Code 5-4 Sample Interleaved Array 82
Code 5-5 Vertex Array Settings for an Interleaved Array 82
Code 5-6 Sample Independent Array 83
Code 5-7 Vertex Array Settings for an Independent Array 84
Code 5-8 Sample Vertex Data Structure with Padding Components 85
Code 5-9 Sample Vertex Data Structure with Automatic Padding 85
Code 5-10 Another Sample Vertex Data Structure with Automatic Padding 86
Code 5-11 Conversion into a 24-Bit Floating-Point Number 144
Code 5-12 Conversion into a 16-Bit Floating-Point Number 145
Code 5-13 Conversion into a 31-Bit Floating-Point Number 145
Code 5-14 Conversion into a 20-Bit Floating-Point Number 146
Code 5-15 Conversion into an 8-Bit Signed Fixed-Point Number with 7 Fractional Bits 146
Code 5-16 Conversion into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits 147
Code 5-17 Alternate Conversion into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits 148
Code 5-18 Conversion into a 13-Bit Signed Fixed-Point Number with 8 Fractional Bits 149
Code 5-19 Conversion into a 13-Bit Signed Fixed-Point Number with 11 Fractional Bits 149
Code 5-20 Conversion into a 16-Bit Fixed-Point Number 150
Code 5-21 Conversion into an 8-Bit Unsigned Fixed-Point Number with No Fractional Bits 151
Code 5-22 Conversion into an 11-Bit Unsigned Fixed-Point Number with 11 Fractional Bits 151
Code 5-23 Conversion into a 12-Bit Unsigned Fixed-Point Number with 12 Fractional Bits 152
Code 5-24 Conversion into a 24-Bit Fixed-Point Number with 24 Fractional Bits 153
Code 5-25 Conversion into a 24-Bit Fixed-Point Number with 8 Fractional Bits 153
Code 5-26 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer 154

Code 5-27 Alternate Conversion of a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer154
Code 5-28 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Signed Integer 154
Code 5-29 Converting a 16-Bit Floating-Point Value into a 32-Bit Floating-Point Value......................... 154

Tables

Table 2-1 Alignments for Each Buffer Type 16
Table 3-1 Parameter List 1 for Command List Objects 27
Table 3-2 Parameter List 2 for Command List Objects 28
Table 3-3 *width* and *height* in `nngxFilterBlockImage` 31
Table 3-4 Color Buffer Formats and `nngxAddMemoryFillCommand` Parameters 38
Table 3-5 Depth/Stencil Buffer Formats and `nngxAddMemoryFillCommand` Parameters 38
Table 4-1 List of Parameters for Display Buffer Objects 46
Table 5-1 State Flag Types.. 62
Table 5-2 State Flag Dependencies 65
Table 5-3 Conditions for Enabling Lookup Tables 65
Table 5-4 Function List 66
Table 5-5 Command Bit Structure 68
Table 5-6 Registers That Set Output Attributes from the Vertex Shader.. 73
Table 5-7 Clock Control Setting Registers for Vertex Shader Output Attributes 75
Table 5-8 Vertex Shader Program Code Setting Registers... 76
Table 5-9 Vertex Shader Swizzle Pattern Setting Registers 76
Table 5-10 Registers That Map Vertex Attributes to Input Registers 76
Table 5-11 Registers for Vertex Attribute Array Settings 78
Table 5-12 Texture Data Address Setting Registers.. 87
Table 5-13 Block Format Setting Registers... 87
Table 5-14 Texture Combiner Setting Registers.. 88
Table 5-15 Texture Combiner Numbers and Starting Registers.. 91
Table 5-16 Registers That Enable or Disable Lighting 91
Table 5-17 Registers That Set Each Color Component 93
Table 5-18 Registers That Set Individual Components of Light Source Coordinates 94
Table 5-19 Registers That Set Individual Components of the Spotlight Direction................................. 94
Table 5-20 Setting Registers for the Bias and Scale with Distance Attenuation 95
Table 5-21 Registers Used by Other Miscellaneous Settings for Individual Light Sources 95
Table 5-22 Registers That Configure Lookup Tables for Fragment Lighting... 96
Table 5-23 Registers That Set the Range of Lookup Table Arguments 97
Table 5-24 Registers That Set Lookup Table Input Values.. 97
Table 5-25 Registers That Set the Output Scaling for Lookup Tables... 98
Table 5-26 Registers for Shadow Attenuation Settings 98
Table 5-27 Registers for Other Miscellaneous Fragment Lighting Settings... 99
Table 5-28 Shadow Texture Setting Registers 100
Table 5-29 Registers That Set the Texture Sampler Type... 100

Table 5-30 Registers for Texture Coordinate Selection 101
Table 5-31 Registers for Procedural Texture Settings 101
Table 5-32 Registers That Configure Lookup Tables for Procedural Textures 103
Table 5-33 Registers That Set the Texture Resolution 105
Table 5-34 Registers for Texture Format Settings 105
Table 5-35 Registers for Texture WRAP Mode Settings 106
Table 5-36 Registers for Texture Filter Mode Settings 107
Table 5-37 Registers for Texture LOD Settings 107
Table 5-38 Registers for Texture Border Color Settings 108
Table 5-39 Registers for Texture LOD Bias Settings 108
Table 5-40 Registers for Gas Settings 109
Table 5-41 Registers That Set the Shading Lookup Table 111
Table 5-42 Fog Setting Registers 112
Table 5-43 Fog Lookup Table Setting Registers 112
Table 5-44 Fragment Operation Setting Registers 113
Table 5-45 Fragment Operation Setting Registers 113
Table 5-46 Fragment Operation Setting Registers 114
Table 5-47 User Clip Setting Registers 114
Table 5-48 Alpha Test Setting Registers 115
Table 5-49 Framebuffer Access Control Setting Registers 116
Table 5-50 Combinations of Framebuffer Access Control Setting Registers 117
Table 5-51 Viewport Setting Registers 117
Table 5-52 Depth Test Setting Registers 118
Table 5-53 Logical Operation and Blend Setting Registers 119
Table 5-54 Early Depth Test Setting Registers 121
Table 5-55 Stencil Test Setting Registers 121
Table 5-56 Culling Setting Registers 122
Table 5-57 Scissoring Setting Registers 122
Table 5-58 Color Mask Setting Registers 123
Table 5-59 Block Format Setting Registers 124
Table 5-60 Register Settings Related to the Rendering API (if the Vertex Buffer Is in Use) 124
Table 5-61 Register Settings Related to the Rendering API (when the Vertex Buffer Is Not in Use) 127
Table 5-62 Geometry Shader Program Code and Swizzle Pattern Data Settings Registers 131
Table 5-63 Miscellaneous Settings Registers When the Geometry Shader Is In Use 131
Table 5-64 Register Setting Values When the Point Shader Is Used 132
Table 5-65: Point Shader Uniforms and Their Corresponding Registers 133
Table 5-66 Register Setting Values When Line Shading Is Used 133
Table 5-67: Line Shader Uniforms and Their Corresponding Registers 135
Table 5-68 Register Setting Values When the Silhouette Shader Is Used 135
Table 5-69: Silhouette Shader Uniforms and Their Corresponding Registers 136
Table 5-70 Register Setting Values When Catmull-Clark Subdivision Is Used 137
Table 5-71: Catmull-Clark Subdivision Shader Uniforms and Their Corresponding Registers 139

Table 5-72 Register Setting Values When Loop Subdivision Is Used 139
Table 5-73: Loop Subdivision Shader Uniforms and Their Corresponding Registers 140
Table 5-74 Register Setting Values When the Particle System Shader Is Used 140
Table 5-75: Particle System Shader Uniforms and Their Corresponding Registers 141
Table 5-76 Otherwise Undocumented Bit Setting Information 143
Table 5-77 PICA Register List 156
Table 6-1 Error Code List 186

Figures

Figure 3-1 Block Diagram of a Command List Object 18
Figure 3-2 3D Command Buffer 19
Figure 3-3 Command Execution in Serial Execution Mode 21
Figure 3-4 Command Execution in Parallel Execution Mode 22
Figure 3-5 Command Execution in Synchronous Execution Mode 23
Figure 3-6 Transferring Partial Image Regions 37
Figure 4-1 Rendering 40
Figure 4-2 Transferring Rendered Results 41
Figure 4-3 Displaying Images After Rendering 41
Figure 4-4 Specifying the Display Area 42
Figure 5-1 Saving Command List Objects 48
Figure 5-2 Using a Copy of a Saved 3D Command Buffer 49
Figure 5-3 Using a Saved 3D Command Buffer Directly 50
Figure 5-4 First Example of Specifying an Export Correctly 55
Figure 5-5 Second Example of Specifying an Export Correctly 56
Figure 5-6 First Example of Specifying an Export Incorrectly 56
Figure 5-7 Third Example of Specifying an Export Correctly 56
Figure 5-8 Fourth Example of Specifying an Export Correctly 57
Figure 5-9 Command Structure for Burst Access 69
Figure 5-10 How to Set 24-Bit Floating-Point Numbers 72

Revision History

Version	Revision Date	Description
1.8	2010/09/16	• Standardized the function argument type `void*` to `GLvoid*`. • Changed the type of the ***srcaddr*** argument to the **`nngxAddVramDmaCommand`** function into `const GLvoid*`. • Removed the restriction that ***srcaddr*** and ***dstaddr*** must be 8-byte aligned in the **`nngxAddVramDmaCommand`** function. • Changed the type of the ***srcaddr*** argument to the **`nngxFilterBlockImage`** function into `const GLvoid*`. • Added **`nngxSetGasAutoAccumulationUpdate`**. • Added **`nngxAddB2LTransferCommand`**. • Added **`nngxAddL2BTransferCommand`**. • Added **`nngxBlockImageCopyCommand`**. • Added **`nngxAddMemoryFillCommand`**. • Added **`nngxGetAllocator`**. • Added a description related to automatic padding for load arrays. • Revised information on gas register settings. • Added a description related to framebuffer access control setting registers. • Revised descriptions of the blend setting register `0x101` and logical operations. • Revised information for bit [0:0] of register `0x25f`, which has settings related to the rendering API. • Added information on the wrapping mode settings for shadow textures.
1.7	2010/08/20	• Revised descriptions of register settings for `dmp_FragOperation.wScale`. • Added information on register settinsg for gas lookup tables and procedural textures. • Added information on shadow and gas settings to the texture format register settings. • Revised the conversion code in section 5.9.7 Converting a 32-Bit Floating-Point Number into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits (Alternate Method). • Changed register `0x289` to `0x28a` in the setting registers related to the rendering API. • Revised information on automatic padding for load arrays.
1.6	2010/07/30	• Deprecated restriction that 2D textures cannot straddle 32 MB boundaries. • Added the **`nngxSetTimeout`** function. • Changed ***buffersize*** restrictions for the **`nngxAdd3DCommand`** function. • Added description of the global ambient register settings. • Added information about bit [7:4] of register `0x1c3`. • Revised information about register `0x227` setting values. • Added setting values for bit [18:18] of register `0x1c4`. • Revised information about bit [16:16] of register `0x25e`. • Clarified the number of items stored in lookup tables.

Version	Revision Date	Description
		• Added details about the commands for the rendering API. • Added information about framebuffer cache clears. • Replaced some images to fix an issue where images were corrupted when creating a PDF version of this document.
1.5	2010/07/13	• Fixed incorrect values. • Added a note about address constraints for cube-map textures.
1.4	2010/07/07	• Revised the notation used for regjster addresses. • Added a table of correspondences between uniforms in reserved geometry shaders and registers. • Changed the setting for register `0x280` in the line shader. • Added a precaution about command generation if **`glUseProgram`** specifies `0`. • Added parameters that can be obtained using **`nngxGetCmdlistParameteri`**. • Added two new functions, **`nngxInvalidateState`** and **`nngxTransferLinearImage`**. • Added register information for global ambients.
1.3	2010/06/04	• Revised the description of **`nngxExportCmdlist`**. • Revised allocator information related to cube-map textures. • Added three new functions: **`nngxClearFillCmdlist`**, **`nngxAddVramDmaCommand`**, and **`nngxFilterBlockImage`**. • Described factors that cause errors to be generated by validation with **`nngxValidateState`**. • Revised argument specifications for **`nngxAdd3DCommand`**. • Added supplementary information on a byte-enable setting of 0 for the command buffer. • Added supplementary information on various registers. • Added information on the binary layout of signed fixed-point numbers. • Added register information for `dmp_Gas.autoAcc`. • Added register information related to clearing the early depth buffer. • Added register information related to the rendering API. • Added register settings that are applicable when a reserved geometry shader is used. • Added register information related to clearing the framebuffer caches. • Added register information related to interrupt commands. • Added a list of PICA registers.
1.2	2010/05/11	• Revised conditions for updating the `NN_GX_STATE_SCISSOR` state as well as dependency relationships. • Added conditions for command generation with `NN_GX_STATE_SHADERPROGRAM`. • Added information on setting registers for fixed vertex attributes. • Added section 5.8.15 Other Setting Registers Related to the Vertex Shader. • Added information on setting registers for the gas shading lookup tables.

Version	Revision Date	Description
		• Fixed typos.
1.1	2010/04/23	• Fixed typos. • Added information on display modes and stereoscopic display. • Added a note about the block format to the specifications of **nngxDisplaybufferStorage**. • Added an error to **nngxTransferRenderImage** related to block-32 mode. • Added a new function, **nngxGetCommandGenerationMode**. • Added details for register settings for vertex shader attributes. • Renamed section 5.8.16 Render Buffer Address Setting Registers to Render Buffer Setting Registers and added register settings related to the render buffer. • Added content in section 5.8.19 Texture Setting Registers. Deleted section 5.12.2.6 Clock Controls for Texture Coordinates and consolidated it with section 5.8.10 Clock Control Setting Registers for Vertex Shader Output Attributes. • Added register information to section 5.8.29 Depth Test Setting Registers. • Added section 5.8.37 Register Settings Related to the Rendering API. • Added section 5.8.38 Register Settings Related to the Geometry Shader. • Added section 5.8.39 Settings for Undocumented Bits. • Added information to section 5.10 Command Cache Restrictions. • Noted that rendering functions generate commands to set registers for the texture sampler type (this was added along with revisions to the implementation). • Added errors for **nngxTransferRenderImage**. • Added information on registers that set gas shading lookup tables. • Revised the description of **nngxStopCmdlist**.
1.0	2010/04/02	• Initial version.

1 Overview

This document describes the system API for the development hardware drivers for DMPGL 2.0. There are four system APIs:

- Initialization API
- Execution Control API
- Display Control API
- Command List Extended API

2 Initialization API

This chapter describes the specifications of the DMPGL 2.0 initialization API.

The initialization API must be called prior to the calling of any other DMPGL API. The initialization API initializes the overall system. Values for the following settings must be passed to it:

- Settings for LCD display
- Memory allocators
- Other extended settings

2.1 API

This section describes the functions in the API.

2.1.1 DMPGL Initialization

```
GLboolean nngxInitialize(
        GLvoid* (*allocator)(GLenum, GLenum, GLuint, GLsizei),
        void (*deallocator)(GLenum, GLenum, GLuint, GLvoid*));
```

Initializes DMPGL. Operation is not guaranteed if any other functions are called prior to this function. It will return `GL_TRUE` if initialization is successful. It will return `GL_FALSE` upon failure. When this function is called again after a successful initialization without first calling the `nngxFinalize` function, it will return `GL_FALSE`.

Specify pointers to the memory allocator and deallocator to the ***allocator*** and ***deallocator*** arguments, respectively. The allocator is used to allocate memory, and the deallocator is used to deallocate memory. The following values are passed to the first argument of the allocator and deallocator functions.

- `NN_GX_MEM_FCRAM` Allocates the FCRAM region
- `NN_GX_MEM_VRAMA` Allocates a region in the A channel in VRAM
- `NN_GX_MEM_VRAMB` Allocates a region in the B channel in VRAM

The following values are passed to the second argument of the allocator and deallocator functions.

- `NN_GX_MEM_SYSTEM` System memory
- `NN_GX_MEM_TEXTURE` Texture memory
- `NN_GX_MEM_VERTEXBUFFER` Vertex buffer memory
- `NN_GX_MEM_RENDERBUFFER` Render buffer memory
- `NN_GX_MEM_DISPLAYBUFFER` Display buffer memory
- `NN_GX_MEM_COMMANDBUFFER` 3D command buffer memory

If the second argument is set to `NN_GX_MEM_TEXTURE`, `NN_GX_MEM_VERTEXBUFFER`, `NN_GX_MEM_RENDERBUFFER`, `NN_GX_MEM_DISPLAYBUFFER`, or `NN_GX_MEM_COMMANDBUFFER`, the name (ID) of the appropriate object will be passed to the third argument of the allocator and deallocator functions.

For the fourth argument to the allocator function, specify the size (in bytes) of the memory area to be allocated. The allocator function will return the address of the area that was allocated. If the allocation failed, it will return zero.

For the fourth argument to the deallocator, specify the address of the area allocated by the allocator. For the first, second, and third arguments to the deallocator, specify the same arguments that you passed to the allocator when the memory was allocated.

The initialization function does not create a default render buffer. The user must create a render buffer explicitly based on the settings being used.

2.1.2 DMPGL Finalization

```
void nngxFinalize(void);
```

Finalizes DMPGL. Some hardware is not reinitialized even if the **nngxInitialize** function is called again after DMPGL finalization.

2.1.3 Getting an Allocator

```
void nngxGetAllocator (
       GLvoid* (**allocator)(GLenum, GLenum, GLuint, GLsizei),
         void  (**deallocator)(GLenum, GLenum, GLuint, GLvoid*));
```

Gets the allocator set by **nngxInitialize**, the DMPGL initialization function. Specify a pointer to a function pointer for both ***allocator*** and ***deallocator*** to get the allocator and deallocator respectively. The allocator and deallocator are not obtained if ***allocator*** and ***deallocator*** are set to 0.

2.2 Allocator Information

Implementation of the allocators set using DMPGL initialization functions must comply with the following address alignment rules.

Table 2-1 Alignments for Each Buffer Type

Buffer Type	Alignment
Texture (2D and environmental mapping)	128 bytes
Vertex buffer	Alignment of each vertex attribute 4 bytes (`GLfloat` type) 2 bytes (`GLshort` and `GLushort` types) 1 byte (`GLbyte` and `GLubyte` types)
Color buffer	32 bytes (for 16-bit colors) 96 bytes (for 24-bit colors) 64 bytes (for 32-bit colors)
Depth buffer (stencil buffer)	32 bytes (for 16-bit depth buffers) 96 bytes (for 24-bit depth buffers) 64 bytes (for 24-bit depth + 8-bit stencil buffers)

Buffer Type	Alignment
Display buffer	16 bytes
3D command buffer	16 bytes
System	4 bytes (when the size allocated is a multiple of four) 2 bytes (when the size allocated is a multiple of two that is not a multiple of four) 1 byte (when the size allocated is not a multiple of two)

These alignments all indicate multiples starting from each address bank (128 MB). For example, a 96-byte alignment would require that the starting addresses of the buffer data be placed at the following positions: (`0x1000_0000`, `0x1000_0060`, `0x1000_00C0`, `0x1000_0120`, …).

Apart from the address alignment rules listed above, you must also implement your allocators with the following specifications of the PICA hardware in mind.

- All six faces of cube-map textures must be contained within 32 MB boundaries.
- Addresses for all six faces of cube-map textures must share the same values for the most-significant 7 bits.
- For cube-map textures, the address of the `GL_TEXTURE_CUBE_MAP_POSITIVE_X` face must be less than or equal to the address of any other face. In other words, the following relationship must be satisfied:

$$(Address\ of\ the\ \mathrm{GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X}\ face) \le (Address\ of\ any\ other\ face)$$

- They must not be located partially in VRAMA and partially in VRAMB.

3 Execution Control API

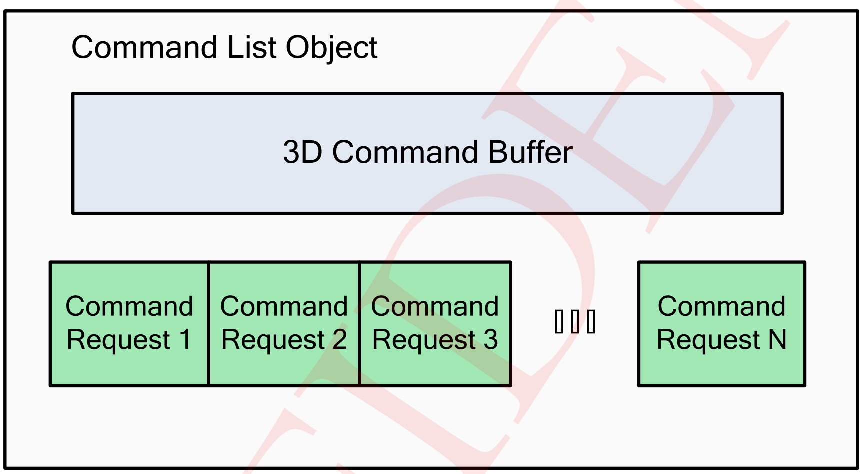
This chapter describes the specifications of the DMPGL 2.0 execution control API. The execution control API lets applications control execution of 3D rendering with a high degree of freedom. It replaces the traditional “one-pipe mode” and “two-pipe mode” mechanisms of execution control.

3.1 Command List Objects

The execution control API introduces a new object called the command list object. This object is treated as the execution unit. A single command list object is made up of the following data.

- 3D command buffer
- Command requests

Figure 3-1 Block Diagram of a Command List Object



The following three actions are performed on command list objects.

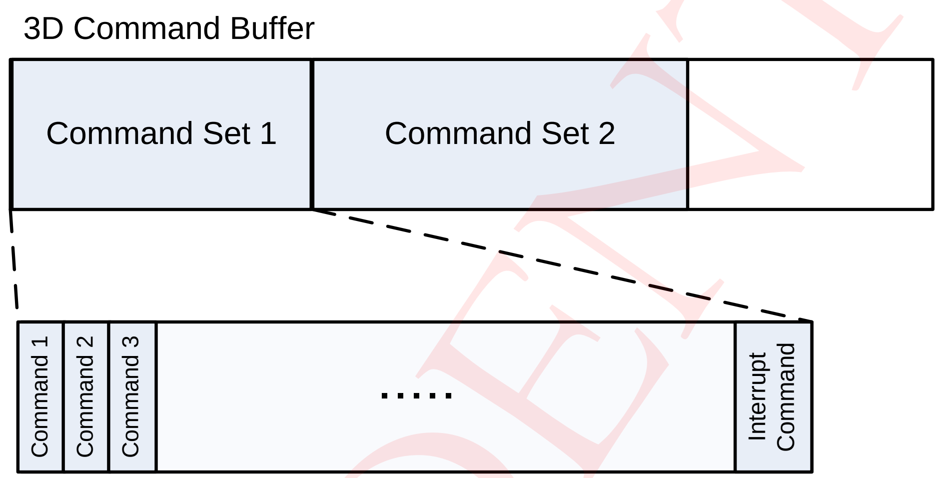
- Accumulating commands
- Executing accumulated commands
- Executing commands immediately while accumulating them

The total accumulatable size of the 3D command buffer and the maximum accumulatable number of command requests are specified using the **`nngxCmdlistStorage`** function. The command list object cannot accumulate any more than these specified limits. A completion interrupt callback is issued to notify the application that accumulated commands have finished executing. Command list objects that have finished execution can be reused by clearing their content with the **`nngxClearCmdlist`** function.

3.1.1 3D Command Buffer

A 3D command buffer is one of the components that make up each command list object. It stores the register write commands to set for PICA. When a 3D execution command from a command request begins, the content of this buffer will be loaded into PICA and executed. The 3D command buffer is caused to accumulate commands by calls to **`glDrawElements`** and other functions in the rendering API.

Figure 3-2 3D Command Buffer



The 3D command buffer stores a number of sequential command sets. Each command set includes multiple register write commands; the last command in each command set is the interrupt generation command. This final command acts as the *command loading completion* command (that is, the command indicating that the loading of commands has completed). All 3D execution commands are executed in command set units.

3.1.2 Command Requests

Command requests include DMA transfer commands, 3D execution commands, memory fill commands, post transfer commands, and render texture transfer commands. Each type of command is queued when certain corresponding functions are issued, and those functions are triggered by specific causes. The details for each type of command are explained below.

3.1.2.1 DMA Transfer Commands

These commands perform a DMA transfer of textures or data in the vertex buffer from FCRAM to SRAM. Functions that allocate texture memory (like **`glTexImage2D`**) and functions that allocate vertex buffers (like **`glBufferData`**) will cause commands to be queued.

3.1.2.2 3D Execution Commands

These commands cause the PICA register write commands that have accumulated in the 3D command buffer to be loaded into PICA and executed. The register write commands for PICA include the *start rendering* command. Each time a 3D execution command is run, a single command set that includes multiple register write commands is executed. When functions like **`glClear`** or **`glCopyTexImage2D`** are called, a *loading complete* command is written to the 3D command buffer to pause the 3D rendering, and the contents of the 3D command buffer up to that point are queued as a single 3D execution command. It is also possible to stop the loading of the 3D command buffer at will by using the **`nngxSplitDrawCmdlist`** function (see section 3.3.8 Splitting the 3D Command Buffer).

3.1.2.3 Memory Fill Commands

These commands use the PICA memory fill feature to fill allocated regions in SRAM with a specified pattern. When the **`glClear`** function is called when attached to a render buffer allocated in SRAM, the command will be queued. Moreover, in order to execute the **`glClear`** function, several PICA registers must be set in addition to the memory fill, so a single 3D execution command will also be queued. In other words, the register write commands for the **`glClear`** function and a 3D command loading complete command are added to the 3D command buffer after the commands that had already accumulated beforehand, one 3D execution command is queued for the sake of loading the 3D command buffer up through that point, and the fill command is queued after that.

3.1.2.4 Post Transfer Commands

These commands take rendered images that were rendered in PICA block format using PICA's post-filters and convert them into a linear format that can be loaded to the LCD. This queues commands using the **`nngxTransferRenderImage`** function in the display buffer control API. As with the **`glClear`** function, this requires that the loading of all commands (such as render commands) up to that point in the 3D command buffer be completed. To do this, a loading complete command is added to the 3D command buffer, and the post transfer command is queued after the 3D execution command is queued. When the 3D command buffer was completed by calling the **`nngxSplitDrawCmdlist`** function immediately beforehand, only the post transfer command will be queued.

3.1.2.5 Render Texture Transfer Commands

These commands copy rendered results from PICA to textures. Commands are queued using **`glCopyTexImage2D`** or other such functions in the texture copying API. As with the **`glClear`** function, this requires that the loading of all commands (such as render commands) up to that point in the 3D command buffer be completed. To do this, a loading complete command is added to the 3D command buffer, and the render texture transfer command is queued after the 3D execution command is queued. When the 3D command buffer was completed by calling the **`nngxSplitDrawCmdlist`** function immediately beforehand, only the render texture transfer command will be queued.

3.2 Executing Commands

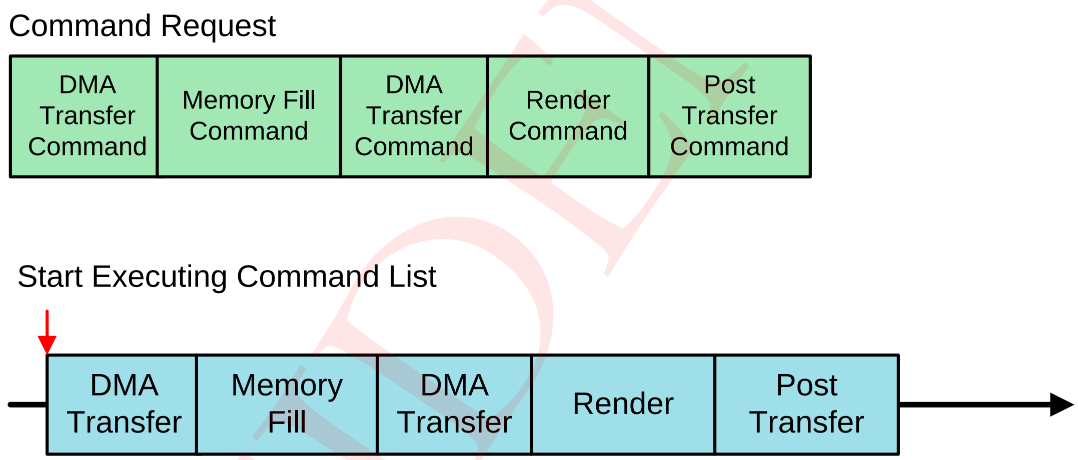
The 3D command buffer and command requests of command list objects can be run in one of three modes: *serial execution mode*, parallel execution mode, or synchronous execution mode. **The current implementation only supports serial execution mode. The other modes are planned to be supported in the future.**

Each mode is explained below.

3.2.1 Serial Execution Mode

Serial execution mode will cause queued command requests to execute in order from start to finish. Each command will be executed after the previous command has finished executing. The figure below shows an example.

Figure 3-3 Command Execution in Serial Execution Mode

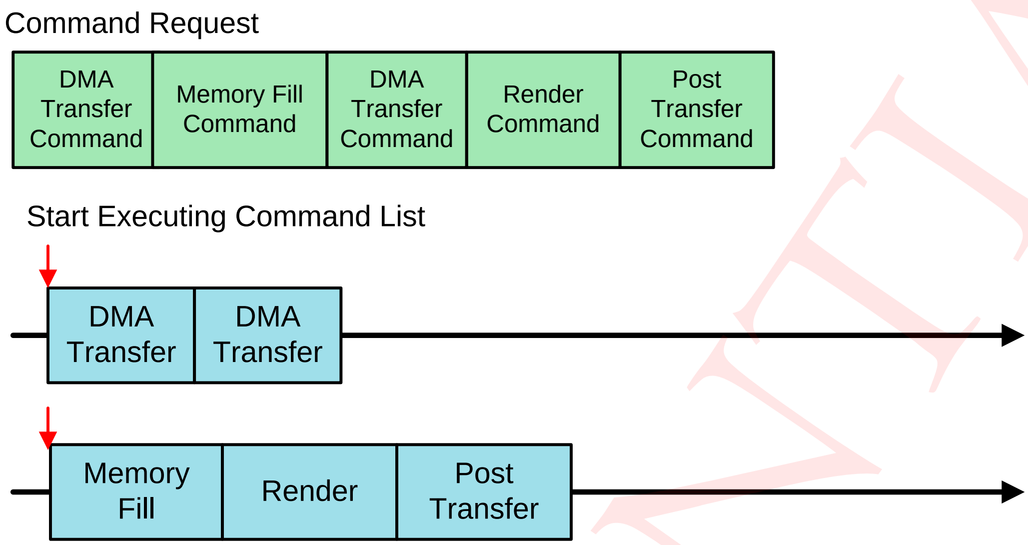


All commands are being executed in order.

3.2.2 Parallel Execution Mode

Parallel execution mode will split the queued command requests into two pipelines and execute them in parallel, one for DMA transfer commands and another for all other commands.

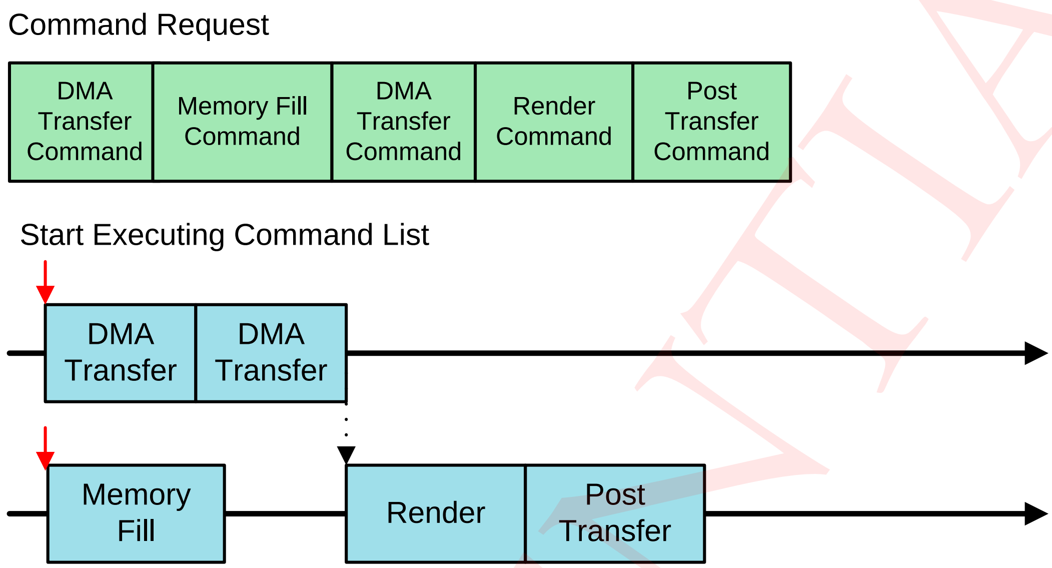
Figure 3-4 Command Execution in Parallel Execution Mode



3.2.3 Synchronous Execution Mode

Synchronous execution mode will split the queued command requests into two pipelines and run them in parallel, one for DMA transfer commands and another for all other commands. However, unlike parallel execution mode, render commands will not execute until any DMA transfer commands that had entered the queue before them have finished executing.

Figure 3-5 Command Execution in Synchronous Execution Mode



3.3 API

This section describes the functions in the API.

3.3.1 Generating Command List Objects

```
void nngxGenCmdlists(GLsizei n, GLuint* cmdlists);
```

Generates a command list object. Specifically, it will create ***n*** command list objects and store the object names in ***cmdlists***. Command list objects have their own namespaces; 0 is reserved for the driver. When a negative value is specified for ***n***, a `GL_ERROR_8000_DMP` error is generated. When memory failed to be allocated for the management region, a `GL_ERROR_8001_DMP` error is generated.

3.3.2 Deleting Command List Objects

```
void nngxDeleteCmdlists(GLsizei n, const GLuint* cmdlists);
```

Deletes command list objects. Specifically, it will delete ***n*** command list objects whose names are stored in the ***cmdlists*** argument. Attempts to delete a command list object that is running causes a `GL_ERROR_8003_DMP` error to be generated. The running command list object will not be affected, but the other command list objects will be deleted. When a negative value is specified for ***n***, a `GL_ERROR_8002_DMP` error is generated.

3.3.3 Binding Command List Objects

```
void nngxBindCmdlist(GLuint cmdlist);
```

Binds the command list object specified in ***cmdlist***. Command list objects that are bound will thereafter accumulate commands. Call the **`nngxRunCmdlist`** function to run command list objects once they are bound. Commands can continue to accumulate in a command list object after that command list has started to execute, but it is also possible to bind another command list object and start accumulating commands there. That said, the order in which commands accumulate in command list objects and the order in which those commands are executed must be the same.

A new command list object is generated when ***cmdlist*** refers to an unused object name. When memory fails to be allocated for the management region at this time, a `GL_ERROR_8004_DMP` error is generated. When you call this function while a command list is being saved, a `GL_ERROR_8005_DMP` error is generated. For details on saving command lists, see section 5.4.1 Start Saving Command Lists.

If no command list object has been bound, or if bound command list objects either have insufficient space in the 3D command buffer or lack available command request slots, calling a DMPGL function that accumulates commands will generate a `GL_INVALID_OPERATION` error.

3.3.4 Allocating Data Regions for Command List Objects

```
void nngxCmdlistStorage(GLsizei bufsize, GLsizei requestcount);
```

Allocates a region for the 3D command buffer of a bound command list object, and also allocates a region for the command request queue. The size of the 3D command buffer (in bytes) will be the size specified to the ***bufsize*** argument. The number of slots allocated in the command request queue will be the value specified to the ***requestcount*** argument. When memory allocation fails, a `GL_ERROR_8006_DMP` error is generated. DMPGL function calls that attempt to add more commands than the command list object is capable of storing (given the specified 3D command buffer size and the command request count) will cause a `GL_INVALID_OPERATION` error to be generated. A `GL_INVALID_OPERATION` error is also generated when a function that generates commands is called on a bound command list object whose data region has not yet been allocated using this function. Execution of this function is ignored when the reserved object 0 is currently bound. If this function is called again on a command list object for which a data region has already been allocated, the existing region will be deallocated, and a new one will be allocated.

It is recommended to allocate more space in the 3D command buffer and more slots in the command request queue than you think you'll require. If necessary, though, you can call the **`nngxGetCmdlistParameteri`** function (described later) to find the actual size used and aim to allocate the optimal size.

When this function is called on a command list object that is executing, a `GL_ERROR_8007_DMP` error is generated. When negative values are specified for ***bufsize*** or ***requestcount***, a `GL_ERROR_8008_DMP` error is generated.

3.3.5 Executing Command List Objects

```
void nngxRunCmdlist(void);
```

Sequentially executes the command requests of command list objects that have been bound using the **`nngxBindCmdlist`** function. Execution of this function is ignored when the reserved object 0 is currently bound.

There are three execution modes for command requests: *serial execution mode*, *parallel execution mode*, and *synchronous execution mode*. See the section 3.2 Executing Commands for more details about each of these modes. The execution mode is set by specifying `NN_GX_CMDLIST_RUN_MODE` for the ***pname*** argument of the **`nngxSetCmdlistParameteri`** command. In the ***param*** argument, specify `NN_GX_CMDLIST_SERIAL_RUN` for serial execution mode, `NN_GX_CMDLIST_PARALLEL_RUN` for parallel execution mode, or `NN_GX_CMDLIST_SYNC_RUN` for synchronous execution mode. Calls to this function are ignored when you bind a command list object while another command list object is running.

Calling this function generates a `GL_ERROR_8009_DMP` error if a region has not been properly allocated for the bound command list object's command buffer and command requests.

3.3.6 Stopping Command List Objects

```
void nngxStopCmdlist(void);
```

Stops the command requests of an executing command list. When the **`nngxRunCmdlist`** function is called, all command requests for command list objects will execute in order. Calling this function (**`nngxStopCmdlist`**) will stop execution after the current command request finishes executing. (It is not possible to interrupt the execution of commands that have already started.) Call the **`nngxRunCmdlist`** function to resume execution.

3.3.7 Scheduling Stops for Command List Objects

```
void nngxReserveStopCmdlist(GLint id);
```

Causes command requests to stop executing automatically after the ***id***th command request has finished executing for a bound command list object. When this is specified for a command list object that is already executing, a `GL_ERROR_800A_DMP` error is generated. When the value specified for the ***id*** argument is zero, negative, or exceeds the maximum command request count, a `GL_ERROR_800B_DMP` error is generated.

3.3.8 Splitting the 3D Command Buffer

```
void nngxSplitDrawCmdlist(void);
```

Adds a *3D command loading complete command* to the 3D command buffer of a bound command list object and queues the resulting 3D execution command in the command requests. If executing commands while accumulating them, the system will execute the 3D commands up to the split point set using this function.

The final command in the 3D command buffer must be a *loading complete command*. A loading complete command will be inserted at the end of the 3D command buffer even when calling functions such as `glCopyTexImage2D` and `glClear`, which require that the 3D command buffer be interrupted.

Calling this function generates a `GL_ERROR_800C_DMP` error when 0 is bound as the current command list. A `GL_ERROR_800D_DMP` error is generated when the maximum number of accumulated command requests has been reached. A `GL_ERROR_800E_DMP` error is generated when, by adding a 3D command loading complete command, the accumulated 3D command buffer exceeds its maximum size.

Some other system functions call this function internally and will output the error codes described in this section if this function causes an error.

3.3.9 Clearing Command List Objects

```
void nngxClearCmdlist(void);
```

Clears a bound command list object. It restores the 3D command buffer and the command request queue to the unused state (they revert to their state right after allocation).

A `GL_ERROR_800F_DMP` error is generated when this function is called on command list objects that are executing.

3.3.10 Clearing Command List Objects and Filling Command Buffers

```
void nngxClearFillCmdlist(Gluint data);
```

Clears a bound command list object. The 3D command buffer and the command request queue return to the unused state. The content of the 3D command buffer is initialized with the value given by ***data***.

A `GL_ERROR_8065_DMP` error is generated when this function is called on command list objects that are executing.

3.3.11 Registering Interrupt Handlers for Command Completion

```
void nngxSetCmdlistCallback(void (*func)(GLint));
```

Registers an interrupt handler that is called when command requests for a bound command list object finish execution. When 0 is specified for the ***func*** argument, the handler will not be called. A `GL_ERROR_8010_DMP` error is generated when this function is called on command list objects that are executing.

```
void nngxEnableCmdlistCallback(GLint id);
void nngxDisableCmdlistCallback(GLint id);
```

If the **`nngxEnableCmdlistCallback`** function is called, the interrupt handler will be called once the ***id***th accumulated command request of a bound command list object has completed. Calls to the interrupt handler can be disabled with the **`nngxDisableCmdlistCallback`** function once they've been enabled with the **`nngxEnableCmdlistCallback`** function. By default, calls to the interrupt handler are disabled. It is also possible to call the interrupt handler on multiple command requests by calling the **`nngxEnableCmdlistCallback`** function multiple times on a single command list object.

When -1 is specified for the ***id*** argument, the interrupt handler will be called for all command requests in the given command list object.

The number of accumulated commands (the value specified for the ***id*** argument) will be passed to the interrupt handler as an argument in order to identify which command request triggers the handler.

The value of ***id*** can be determined when accumulating commands by calling the **`nngxGetCmdlistParameteri`** function to get the current number of accumulated command requests.

Note that this sets a completion interrupt for the ***id***'th command to be accumulated. It does not set a completion interrupt for the ***id***'th command to be executed. In parallel execution mode and synchronous execution mode, the ***id***'th accumulated command request won't necessarily be the same as the ***id***'th executed command request.

It is possible to poll for completed commands even if you're not using interrupt handlers. To get the execution status, specify `NN_GX_CMDLIST_IS_RUNNING` to the **`nngxGetCmdlistParameteri`** function.

A `GL_ERROR_8012_DMP` error is generated when the **`nngxEnableCmdlistCallback`** function is called with the ***id*** argument set equal to 0, a negative number other than -1, or a value that exceeds the maximum command request count.

A `GL_ERROR_8014_DMP` error is generated if the **`nngxDisableCmdlistCallback`** function is called with the ***id*** argument set equal to 0, a negative number other than -1, or a value that exceeds the maximum command request count.

3.3.12 Setting Parameters for Command List Objects

```
void nngxSetCmdlistParameteri(GLenum pname, GLint param);
```

Sets the parameters of a bound command list object. The settings are listed below. Attempting to set parameters for a command list object that is executing will result in a `GL_ERROR_8015_DMP` error. When values not listed in the table below are set for the ***pname*** or ***param*** parameters, a `GL_ERROR_8016_DMP` error is generated.

Table 3-1 Parameter List 1 for Command List Objects

pname	param	Description
`NN_GX_CMDLIST_RUN_MODE`	`NN_GX_CMDLIST_SERIAL_RUN` **(The mode listed above is the only one that is currently supported.)**	Sets the execution mode.

3.3.13 Getting the Parameters of Command List Objects

```
void nngxGetCmdlistParameteri(GLenum pname, GLint* param);
```

Gets the parameters of a bound command list object and stores them in ***param***. The various settings are listed below. When values not listed in the table below are set for the ***pname*** parameter, a `GL_ERROR_8017_DMP` error is generated. When ***pname*** is set equal to a value other than `NN_GX_CMDLIST_BINDING` or when 0 is bound to the current command list, a `GL_ERROR_8018_DMP` error is generated.

Table 3-2 Parameter List 2 for Command List Objects

pname **Value**	**Description**
`NN_GX_CMDLIST_RUN_MODE`	Gets the execution mode that is currently set.
`NN_GX_CMDLIST_IS_RUNNING`	Gets the execution status of the command list. If a value of `GL_TRUE` is obtained, the command list is executing. If a value of `GL_FALSE` is obtained, the command list is not executing.
`NN_GX_CMDLIST_USED_BUFSIZE`	Gets the size (in bytes) of the commands accumulated in the 3D command buffer.
`NN_GX_CMDLIST_USED_REQCOUNT`	Gets the number of command requests that are currently accumulated.
`NN_GX_CMDLIST_MAX_BUFSIZE`	Gets the maximum size of the 3D command buffer. This gets the value that was specified for the ***bufsize*** argument of the **`nngxCmdlistStorage`** function.
`NN_GX_CMDLIST_MAX_REQCOUNT`	Gets the maximum number of command requests. This gets the value that was specified for the ***requestcount*** argument of the **`nngxCmdlistStorage`** function.
`NN_GX_CMDLIST_TOP_BUFADDR`	Gets the starting address of the 3D command buffer.
`NN_GX_CMDLIST_BINDING`	Gets the ID of the command list object that is currently bound.
`NN_GX_CMDLIST_RUN_BUFSIZE`	Gets the size (in bytes) of the 3D command buffer that has already been run.
`NN_GX_CMDLIST_RUN_REQCOUNT`	Gets the number of command requests that have already been run.
`NN_GX_CMDLIST_TOP_REQADDR`	Gets the starting address of the data region for the command request queue.
`NN_GX_CMDLIST_NEXT_REQTYPE`	If command execution is stopped, this will get the command type of the command request that will be run next. If a command is running, this will get the command type of the command request being run. If all command requests have already finished running, nothing will be obtained. The macros below indicate the types of commands that are obtained with this parameter. • `NN_GX_CMDLIST_REQTYPE_DMA`: DMA transfer command • `NN_GX_CMDLIST_REQTYPE_RUN3D`: 3D execution command • `NN_GX_CMDLIST_REQTYPE_FILLMEM`: Memory fill command • `NN_GX_CMDLIST_REQTYPE_POSTTRANS`: Post transfer command • `NN_GX_CMDLIST_REQTYPE_COPYTEX`: Render texture transfer command
`NN_GX_CMDLIST_NEXT_REQINFO`	If command execution is stopped, this will get the parameter information for the command request that will be run next. If a command is running, this will get the parameter information for the command request being run. If all command requests have already finished running, nothing will be obtained. This is only supported if the next command to be run or the command currently running is a 3D execution command. If this parameter is used when any other command is running or up next, nothing will be obtained.

pname Value	Description
	The address of the command buffer will be returned in the first element of ***param***, and the size (in bytes) of the command buffer will be stored in the second element of ***param***.
`NN_GX_CMDLIST_HW_STATE`	Gets a 32-bit value indicating the hardware status. The definitions of each bit are shown below: • `[20]`: Set (has a value of `1`) when a post transfer is executing • `[19]`: Set when a memory fill is executing • `[18]`: Set when a FIFO underrun error occurred for the lower LCD • `[17]`: Set when a FIFO underrun error occurred for the upper LCD • `[16]`: Set when the post-vertex cache is busy • `[15]`: Set when bits `[1:0]` in Register `0x252` are set to the value `1` • `[14]`: Set when vertex processor 3 is busy • `[13]`: Set when vertex processor 2 is busy • `[12]`: Set when vertex processor 1 is busy • `[11]`: Set when vertex processor 0 (which doubles as the geometry shader processor) is busy • `[10]`: Set when bits `[1:0]` in register `0x229` are not `0` • `[9]`: Set when input to the module that loads the command buffer and the vertex array is busy • `[8]`: Set when output to the module that loads the command buffer and the vertex array is busy • `[7]`: Set when the early depth test module is busy • `[6]`: Holds the value of busy signal 2 of the per-fragment operations module • `[5]`: Holds the value of busy signal 1 of the per-fragment operations module • `[4]`: Set when the texture combiners are busy • `[3]`: Set when fragment lighting is busy • `[2]`: Set when the texture units are busy • `[1]`: Set when the rasterization module is busy • `[0]`: Set when triangle setup is busy

3.3.14 Checking for V-Sync Updates

```
GLint nngxCheckVSync(GLenum display);
```

Used to check for V-Sync updates on the screen or screens specified by ***display***. When `NN_GX_DISPLAY0` is specified for ***display***, V-Syncs for screen 0 (the first screen) will be processed. When `NN_GX_DISPLAY1` is specified, V-Syncs for screen 1 (the second screen) will be processed. When `NN_GX_DISPLAY_BOTH` is specified, V-Syncs for both screens will be processed. When any other value is specified for ***display***, a `GL_ERROR_8019_DMP` error is generated. The return value in this case will be undefined.

The driver's internal V-Sync counter will be the return value, and the V-Sync can be checked asynchronously by checking whether this value has been updated. When `NN_GX_DISPLAY_BOTH` is specified for ***display***, the value will be updated by V-Syncs on both screens.

The internal counter for the return value will reset to 0 if the implementation-dependent maximum count is exceeded. This maximum value may be changed without notice in the future when the driver is updated.

3.3.15 V-Sync Synchronization

```
void nngxWaitVSync(GLenum display);
```

Used for V-Sync synchronization on the screen or screens specified by ***display***. When `NN_GX_DISPLAY0` is specified for ***display***, V-Syncs for screen 0 (the first screen) will be processed. When `NN_GX_DISPLAY1` is specified, V-Syncs for screen 1 (the second screen) will be processed. When `NN_GX_DISPLAY_BOTH` is specified, V-Syncs for both screens will be processed. When any other value is specified for ***display***, a `GL_ERROR_801A_DMP` error is generated and control returns immediately.

If this function is called, control will return after waiting for the next V-Sync.

3.3.16 Registering the V-Sync Callback Function

```
void nngxSetVSyncCallback(GLenum display, void (*func)(GLenum));
```

Registers the V-Sync callback. When `NN_GX_DISPLAY0` is specified for ***display***, a V-Sync callback for screen 0 (the first screen) will be registered. When `NN_GX_DISPLAY1` is specified, a V-Sync callback for screen 1 (the second screen) will be registered. When `NN_GX_DISPLAY_BOTH` is specified, a shared V-Sync callback for both screens will be registered. For ***func***, specify a pointer to the callback function. A screen identifier will be passed as an argument to the callback function. When any other value is specified for ***display***, a `GL_ERROR_801B_DMP` error is generated.

3.3.17 Waiting for a Command List Object to Complete Execution

```
void nngxWaitCmdlistDone(void);
```

Waits for an executing command list object to complete. Control returns when all of the accumulated command requests finish executing. 3D execution commands are executed until they reach the location where the command buffer was already split when this function was called. If you want to ensure that the entire 3D command buffer is executed, call the **`nngxSplitDrawCmdlist`** function before calling this function.

This function does not return until command execution has completed. See section 3.3.21 Setting the Timeout for Waiting to Complete Command List Object Execution for details on setting a timeout.

3.3.18 Transferring Data via DMA

```
void nngxAddVramDmaCommand(
        const GLvoid* srcaddr, GLvoid* dstaddr, Glsizei size);
```

Accumulates a DMA transfer command in the current command list. The DMA transfer command transfers ***size*** bytes of data from the address specified by ***srcaddr*** to the address specified by ***dstaddr***.

A `GL_ERROR_8062_DMP` error is generated when a valid command list is not currently bound. A `GL_ERROR_8064_DMP` error is generated when ***size*** is negative.

3.3.19 Transferring Block Images with an Anti-Aliasing Filter

```
void nngxFilterBlockImage(const GLvoid* srcaddr, GLvoid* dstaddr,
              GLsizei width, GLsizei height, Glenum format);
```

Accumulates a transfer command—with an anti-aliasing filter for block images—in the current command list. A *block image* is a rendered image or an image that uses the 8-block addressing format of a texture in the native PICA format. A 2x2 anti-aliasing filter is applied as data is transferred from the address specified by ***srcaddr*** to the address specified by ***dstaddr***. The width and height of the original image are specified by ***width*** and ***height***, respectively, in pixels. The following pixel formats can be specified for ***format***.

- `GL_RGBA8_OES`: 32-bit R8G8B8A8
- `GL_RGB8_OES`: 24-bit R8G8B8
- `GL_RGBA4`: 16-bit R4G4B4A4
- `GL_RGB5_A1`: 16-bit R5G5B5A1
- `GL_RGB565`: 16-bit R5G6B5

Calling this function generates a `GL_ERROR_8068_DMP` error when 0 is bound to the current command list or when the command request queue is too small.

Both ***srcaddr*** and ***dstaddr*** must be 8-byte aligned. A `GL_ERROR_8069_DMP` error is generated when either value is not 8-byte aligned.

The value of ***format*** restricts the values of ***width*** and ***height***, as shown in the following table.

Table 3-3 ***width*** **and** ***height*** **in** `nngxFilterBlockImage`

format	*width*	*height*
• `GL_RGBA8_OES` • `GL_RGB8_OES`	A multiple of 64 that is greater than or equal to 64	A multiple of 8 that is greater than or equal to 64
• `GL_RGBA4` • `GL_RGB5_A1` • `GL_RGB565`	A multiple of 128 that is greater than or equal to 128	A multiple of 8 that is greater than or equal to 128

A `GL_ERROR_806A_DMP` error is generated when the specified values conflict with these restrictions.

A `GL_ERROR_806B_DMP` error is generated when an invalid ***format*** is specified.

3.3.20 Image Transfer Requests

```
void nngxTransferLinearImage(const GLvoid* srcaddr, GLuint dstid, GLenum target);
```

Adds to the current command list a command that transfers the region specified by the ***srcaddr*** argument to the render buffer or texture specified by the ***dstid*** argument.

The ***srcaddr*** argument specifies the address of the source data to transfer. The ***dstid*** argument specifies the object ID of the render buffer or texture where the data should be transferred. When the ***target*** argument is `GL_RENDERBUFFER`, ***dstid*** must indicate a render buffer object. In this case, if ***dstid*** specifies `0`, the data will be transferred to the color buffer attached to the current framebuffer. When the ***target*** argument is `GL_TEXTURE_2D`, ***dstid*** must indicate a 2D texture object. When the ***target*** argument is `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`, ***dstid*** must indicate a cube-map texture object.

The region specified by the ***srcaddr*** argument must store image data that has the same format, width, and height as the render buffer or texture specified by the ***dstid*** argument. The source image data will be converted to block addressing in the native PICA as it is transferred to the destination. When the destination object is a render buffer, the data will be converted to either 8-block addressing or 32-block addressing, depending on the block format setting that was set when this function was called. When the destination object is a texture, the data will be converted to 8-block addressing. This function will only convert the addressing, it will not perform V-flipping or byte-order conversion. Since the render buffer and texture use the native PICA format for images, the source image data must have V-flipping or byte-order conversion done in advance if necessary.

When the commands accumulated in the current 3D command buffer have not been split, a split command is added before the transfer command.

When the destination is in 24-bit format, the source data must be in 32-bit format, and the first byte of each four-byte sequence of the source data will be discarded when the data is transferred. (The hardware does not support transfers from 24-bit format sources to 24-bit format destinations.)

If this function is called when the current command list is bound to `0`, the `GL_ERROR_805B_DMP` error will be generated. When the maximum number of accumulated command requests has been reached in the current command list, the `GL_ERROR_805C_DMP` error will be generated. When the size of the current 3D command buffer is insufficient, the `GL_ERROR_805D_DMP` error will be generated. When the render buffer or texture specified to the ***dstid*** argument does not exist, or when the address has not been allocated, the `GL_ERROR_805E_DMP` error will be generated.

When the 8-block format was configured when this function was called, the width and height of the destination render buffer must be multiples of 8. Likewise, when the 32-block format was configured when this function was called, the width and height of the destination render buffer must be multiples of 32. The width and height must also be greater than or equal to 128. When these restrictions are violated, the `GL_ERROR_805F_DMP` error will be generated.

When an invalid ***target*** is specified, the `GL_ERROR_8060_DMP` error will be generated. When the size of the destination render buffer or texture is anything other than 32, 24, or 16 bits, the `GL_ERROR_8067_DMP` error will be generated.

3.3.21 Setting the Timeout for Waiting to Complete Command List Object Execution

```
void nngxSetTimeout (GLint64EXT time, void (*callback)(void));
```

This function specifies the length of time that the **nngxWaitCmdlistDone** function, which waits for the executing command list object to complete, will wait before timing out. The ***time*** argument specifies the length of time until timeout as a system tick value. The ***callback*** argument specifies a pointer to the callback function to call after timing out.

Once this timeout is set, any call to **nngxWaitCmdlistDone** that does not return before ***time*** has elapsed will result in a call to the function specified in ***callback*** and the completion of the call to **nngxWaitCmdlistDone**, whether command execution has completed or not.

The default value for ***time*** is `0`, which generates no timeout. The default value for ***callback*** is `NULL`, meaning no callback function is called when a timeout occurs.

This timeout feature is only enabled in debug and development builds.

3.3.22 Updating Additive Blend Results Rendered with Gas Density Information

```
void nngxSetGasAutoAccumulationUpdate (GLint id);
```

Updates `INVERTED_ACC_MAX1`, a value related to the results of additive blending when gas density information is rendered. For more details on `INVERTED_ACC_MAX1`, see the *DMPGL 2.0 Specifications*.

When called, the **nngxSetGasAutoAccumulationUpdate** function configures the maximum value of $D1$—a result of additive blending when gas density information is rendered—to be applied to `INVERTED_ACC_MAX1` within the interrupt handler that is invoked upon completion of the ***id***'th command request accumulated in the bound command list object. For example, when ***id*** is 1 this setting affects the first command request, when ***id*** is 2 this setting affects the second command request, and so on. You must specify a command request that is a 3D execution command.

This function is required to implement the functionality of the fragment shader uniform `dmp_Gas.autoAcc` using commands generated by the application. You must clear the maximum value saved for the additive blending result $D1$ to 0 as necessary before you start rendering gas density information. The maximum value is cleared (initialized) with bits [15:0] of register `0x125`. After rendering the gas density information, use this function again to update `INVERTED_ACC_MAX1` before you start gas shading.

`INVERTED_ACC_MAX1` is updated correctly when this function is called on a command request that includes a command to render gas density information. However, note that it is impossible to update `INVERTED_ACC_MAX1` before gas shading when this function is called on a command request that includes both a command to render gas density information and a command to start gas shading. Furthermore, if a value is written to bits [15:0] of register `0x0e5` after this function has updated `INVERTED_ACC_MAX1`, this function's settings are overwritten and invalidated.

A `GL_ERROR_806D_DMP` error is generated when 0 is bound as the command list object. A `GL_ERROR_806E_DMP` error is generated when ***id*** is less than or equal to 0, when ***id*** is greater than the number of accumulated command requests, and when the command request specified by ***id*** is not a 3D execution command.

3.3.23 Transferring a Block Image That Is Converted into a Linear Image

```
void nngxAddB2LTransferCommand(
        const GLvoid* srcaddr, GLsizei srcwidth, GLsizei srcheight,
        GLenum srcformat, GLvoid* dstaddr, GLsizei dstwidth, GLsizei dstheight,
        GLenum dstformat, GLenum aamode, GLboolean yflip, GLsizei blocksize);
```

Adds commands to the command list to convert a block image into a linear image and then transfer it.

This function converts a block image in the rendering format into a linear image in the display format. Although the **nngxTransferRenderImage** function provides equivalent functionality, this function has more general uses. Also, like **nngxTransferRenderImage**, this function only adds a transfer request command without adding a 3D split command.

The block image at the address specified by ***srcaddr*** is transferred as a linear image and stored at the address specified by ***dstaddr***. Both ***srcaddr*** and ***dstaddr*** must be 16-byte aligned.

The original image's width and height in pixels are given by ***srcwidth*** and ***srcheight***; the transferred image's width and height in pixels are given by ***dstwidth*** and ***dstheight***. These dimensions must all be multiples of the block size, which is either 8 or 32. However, if the transferred image uses 24 bits per pixel and a block size of 8, both the original and transferred images must have widths that are multiples of 16. This function exits without adding any commands if any of the image dimensions is 0. The width and height of the transferred image must be less than or equal to the width and height of the original image.

The original and transferred images have pixel formats specified by ***srcformat*** and ***dstformat*** using the following macros.

- `GL_RGBA8_OES`: 32-bit RGBA8
- `GL_RGB8_OES`: 24-bit RGB8
- `GL_RGBA4`: 16-bit RGBA4
- `GL_RGB5_A1`: 16-bit RGBA5551
- `GL_RGB565`: 16-bit RGB565

Conversions that increase the pixel size are not possible. For example, you cannot convert from a 24-bit format to a 32-bit format or from a 16-bit format to either a 24- or 32-bit format.

The antialiasing filter mode is specified by ***aamode*** using the following macros.

- `NN_GX_ANTIALIASE_NOT_USED`: No antialiasing
- `NN_GX_ANTIALIASE_2x1`: Transfer with 2x1 antialiasing
- `NN_GX_ANTIALIASE_2x2`: Transfer with 2x2 antialiasing

When antialiasing is enabled, the transferred image is shrunk in half in the filtering direction. Specifically, 2x2 antialiasing shrinks the image in half vertically and horizontally and 2x1 antialiasing shrinks the image in half horizontally.

The transferred image is flipped vertically when ***yflip*** is `GL_TRUE` and is not flipped when ***yflip*** is `GL_FALSE`. Nonzero values are considered to be `GL_TRUE`.

The original image is transferred using a block size of 8 or 32, specified by ***blocksize***.

This function generates the following errors.

- `GL_ERROR_807C_DMP` when 0 is bound to the current command list or the command request queue is full
- `GL_ERROR_807D_DMP` when ***srcaddr*** or ***dstaddr*** is not 16-byte aligned
- `GL_ERROR_807E_DMP` when ***blocksize*** is not 8 or 32
- `GL_ERROR_807F_DMP` when ***aamode*** is an invalid value
- `GL_ERROR_8080_DMP` when ***srcformat*** and ***dstformat*** are invalid values
- `GL_ERROR_8081_DMP` when ***dstformat*** has a larger pixel size than ***srcformat***
- `GL_ERROR_8082_DMP` when either ***srcwidth***, ***srcheight***, ***dstwidth***, or ***dstheight*** is invalid
- `GL_ERROR_8083_DMP` when the width or height of the transferred image is larger than the original image

3.3.24 Transferring a Linear Image That Is Converted into a Block Image

```
void nngxAddL2BTransferCommand(
        const GLvoid* srcaddr, GLvoid* dstaddr,
        GLsizei width, GLsizei height, GLenum format, GLsizei blocksize);
```

Adds commands to the command list to convert a linear image into a block image and then transfer it.

This function converts a linear image in the display format into a block image in the rendering format. Although the **`nngxTransferLinearImage`** function provides equivalent functionality, this function has more general uses. Also, like **`nngxTransferLinearImage`**, this function only adds a transfer request command without adding a 3D split command.

The linear image at the address specified by ***srcaddr*** is transferred as a block image and stored at the address specified by ***dstaddr***. Both ***srcaddr*** and ***dstaddr*** must be 16-byte aligned.

The width and height of both the original and transferred images (in pixels) are given by ***width*** and ***height***. Both images must have the same width and height and these dimensions must all be multiples of the block size, which is either 8 or 32. However, if the transferred image uses 24 bits per pixel and a block size of 8, the width must be a multiple of 32. This function exits without adding any commands if either ***width*** or ***height*** is 0.

The transferred image has a pixel format specified by ***format***. The original image must have the same format as the transferred image unless ***format*** is a 24-bit format, in which case the original image must use a 32-bit format. For each four-byte block of the original data that is transferred, the first byte is discarded (the hardware does not support transfers between 24-bit formats). Specify the pixel format using the following macros.

- `GL_RGBA8_OES`: 32-bit RGBA8
- `GL_RGB8_OES`: 24-bit RGB8
- `GL_RGBA4`: 16-bit RGBA4
- `GL_RGB5_A1`: 16-bit RGBA5551
- `GL_RGB565`: 16-bit RGB565

The transferred image has a block size of 8 or 32, specified by ***blocksize***.

This function generates the following errors.

- `GL_ERROR_806F_DMP` when 0 is bound to the current command list or the command request queue is full
- `GL_ERROR_8070_DMP` when ***srcaddr*** or ***dstaddr*** is not 16-byte aligned
- `GL_ERROR_8071_DMP` when ***blocksize*** is not 8 or 32
- `GL_ERROR_8072_DMP` when ***width*** or ***height*** is invalid
- `GL_ERROR_8073_DMP` when ***format*** is invalid

3.3.25 Transferring a Block Image

```
void nngxAddBlockImageCopyCommand(
        const GLvoid* srcaddr, GLsizei srcunit, GLsizei srcinterval,
        GLvoid* dstaddr, GLsizei dstunit, GLsizei dstinterval, GLsizei totalsize);
```

Adds a block image transfer command to the current command list.

This function can copy images between textures and rendered render buffers. This function's distinguishing feature is its ability to transfer a specified amount of data with a specified skip size, allowing you to cut a region out of the original image and fit an image into a partial region of the target image.

Data is transferred from the address specified by ***srcaddr*** into the address specified by ***dstaddr***. Both addresses must be 16-byte aligned.

The total number of bytes to transfer is specified by ***totalsize***, which must be a multiple of 16.

Data is transferred ***srcunit*** bytes at a time, with a skip size (in bytes) specified by ***srcinterval***. This process can be described as follows.

1. Read and transfer ***srcunit*** bytes of data.
2. Skip (do not transfer) the next ***srcinterval*** bytes of data.
3. Repeat until ***totalsize*** bytes have been transferred.

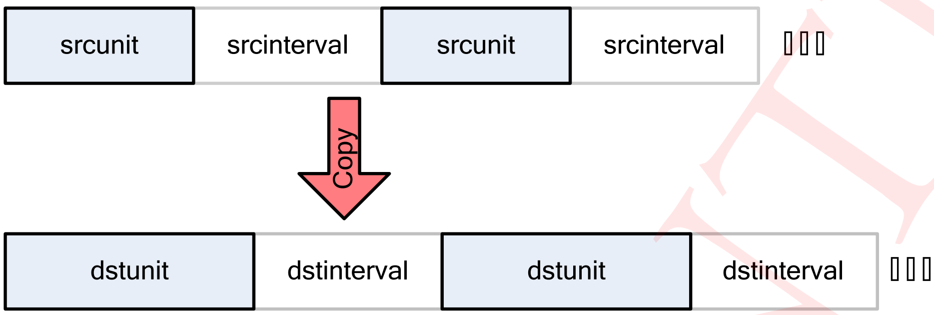
If ***srcinterval*** is 0, this function reads and transfers a continuous region of ***totalsize*** bytes. For any other ***srcinterval*** value, data is alternatively read and skipped; this allows you to transfer partial regions that are cut out of the original image.

The transferred data is written ***dstunit*** bytes at a time with a skip size of ***dstinterval*** bytes. This process can be described as follows.

1. Write ***dstunit*** bytes of transferred data.
2. Advance the write address by (skip) ***dstinterval*** bytes.
3. Repeat until ***totalsize*** bytes have been transferred.

If ***dstinterval*** is 0, this function writes a continuous region of ***totalsize*** bytes. For any other ***dstinterval*** value, data is alternatively written and skipped; this allows you to paste an image into a partial region of the target image.

Figure 3-6 Transferring Partial Image Regions



The colored regions in the figure are transferred.

The ***srcunit***, ***srcinterval***, ***dstunit***, and ***dstinterval*** arguments must all be non-negative multiples of 16 that are less than `0x100000`.

This function generates the following errors.

- `GL_ERROR_8074_DMP` when 0 is bound to the current command list or the command request queue is full
- `GL_ERROR_8075_DMP` when ***srcaddr*** or ***dstaddr*** is not 16-byte aligned
- `GL_ERROR_8076_DMP` when ***totalsize*** is not a multiple of 16
- `GL_ERROR_8077_DMP` when ***srcunit***, ***srcinterval***, ***dstunit***, or ***dstinterval*** is invalid

Note: When you set this function's arguments to transfer a block image that is the result of rendering (or some other process), remember that the image's starting address is at its upper-left corner and that, when it uses a block size of 8, its data is placed in 8x8 pixel blocks. For more details about block formats, see the section on the native PICA format in the *DMPGL 2.0 Specifications*.

3.3.26 Filling Memory

```
void nngxAddMemoryFillCommand(
        GLvoid* startaddr0, GLsizei size0, GLuint data0, GLsizei width0,
        GLvoid* startaddr1, GLsizei size1, GLuint data1, GLsizei width1);
```

Adds commands to the current command list to fill the specified regions with the specified data.

By filling memory with a specified data pattern, this function can be used to clear the color and depth (stencil) buffers. The **`glClear`** function provides equivalent functionality, but this function has more general uses. You can fill two regions using separate parameters for each. Channel 1 is configured by ***startaddr0***, ***size0***, ***data0***, and ***width0***. Channel 2 is configured by ***startaddr1***, ***size1***, ***data1***, and ***width1***.

Memory is filled starting at addresses ***startaddr0*** and ***startaddr1***. These addresses must be 16-byte aligned. If an address is 0, its corresponding channel is not used.

size0 and ***size1*** bytes of memory are filled. Both ***size0*** and ***size1*** must be multiples of 16.

Memory regions are filled by repeatedly storing the data specified by ***data0*** and ***data1***.

The number of bits in each fill pattern is specified by ***width0*** and ***width1***, which can be 16, 24, or 32.

- Given a value of 16, memory is filled 16 bits at a time using bits [15:0] of ***data0*** and ***data1***.
- Given a value of 24, memory is filled 24 bits at a time using bits [23:0] of ***data0*** and ***data1***.
- Given a value of 32, memory is filled 32 bits at a time using bits [31:0] of ***data0*** and ***data1***.

The following table shows which bits of ***data0*** and ***data1*** are used to clear various color buffer formats, as well as the corresponding brightness values for each component and the required values for ***width0*** and ***width1***. For example, a `GL_RGBA8_OES` color buffer's R, G, B, and A components are cleared using bits [31:24], [23:16], [15:8], and [7:0], respectively, of ***data0*** or ***data1***; each component's brightness is a value between 0 and 255; and the value of ***width0*** or ***width1*** must be 32.

Table 3-4 Color Buffer Formats and `nngxAddMemoryFillCommand` Parameters

Color Buffer Format	***data0 / data1*** Bits				Brightness Values				***width0 / width1*** Value
	R	G	B	A	R	G	B	A	
`GL_RGBA8_OES`	[31:24]	[23:16]	[15:8]	[7:0]	0–255	0–255	0–255	0–255	32
`GL_RGBA4`	[15:12]	[11:8]	[7:4]	[3:0]	0–15	0–15	0–15	0–15	16
`GL_RGB5_A1`	[15:11]	[10:6]	[5:1]	[0:0]	0–31	0–31	0–31	0 or 1	16
`GL_RGB565`	[15:11]	[10:5]	[4:0]	-	0–31	0–63	0–31	-	16

The following table shows which bits of ***data0*** and ***data1*** are used to clear various depth and stencil buffer formats, as well as the required values for ***width0*** and ***width1***. For example, a `GL_DEPTH24_STENCIL8_EXT` depth/stencil buffer's depth and stencil values are cleared using bits [23:0] and [31:24], respectively, of ***data0*** or ***data1***; the value of ***width0*** or ***width1*** must be 32.

Table 3-5 Depth/Stencil Buffer Formats and `nngxAddMemoryFillCommand` Parameters

Depth/Stencil Buffer Format	***data0 / data1*** Bits		***width0 / width1*** Value
	Depth	Stencil	
`GL_DEPTH24_STENCIL8_EXT`	[23:0]	[31:24]	32
`GL_DEPTH_COMPONENT24_OES`	[23:0]	-	24
`GL_DEPTH_COMPONENT16`	[15:0]	-	16

This function generates the following errors.

- `GL_ERROR_8078_DMP` when 0 is bound to the current command list or the command request queue is full

- `GL_ERROR_8079_DMP` when ***startaddr0*** or ***startaddr1*** is not 16-byte aligned
- `GL_ERROR_807A_DMP` when ***size0*** or ***size1*** is not a multiple of 16
- `GL_ERROR_807B_DMP` when ***width0*** or ***width1*** is invalid

If ***startaddr0*** is 0, ***size0***, ***data0***, and ***width0*** are not checked for errors. Likewise, if ***startaddr1*** is 0, ***size1***, ***data1***, and ***width1*** are not checked for errors.

Channel 0 and channel 1 are executed simultaneously. If they have overlapping regions, it is undefined which result will ultimately be applied.

4 Display Control API

This chapter describes the API for controlling the framebuffer (display buffer) in DMPGL 2.0.

This API allows you to perform the following types of operations.

- The applications can generate multiple new display buffers.
- The application can specify whether or not to transfer the rendered results to the display buffer. Rendering results from individual render passes can also be transferred to multiple display buffers.
- The application can then freely specify which display buffer to display to.

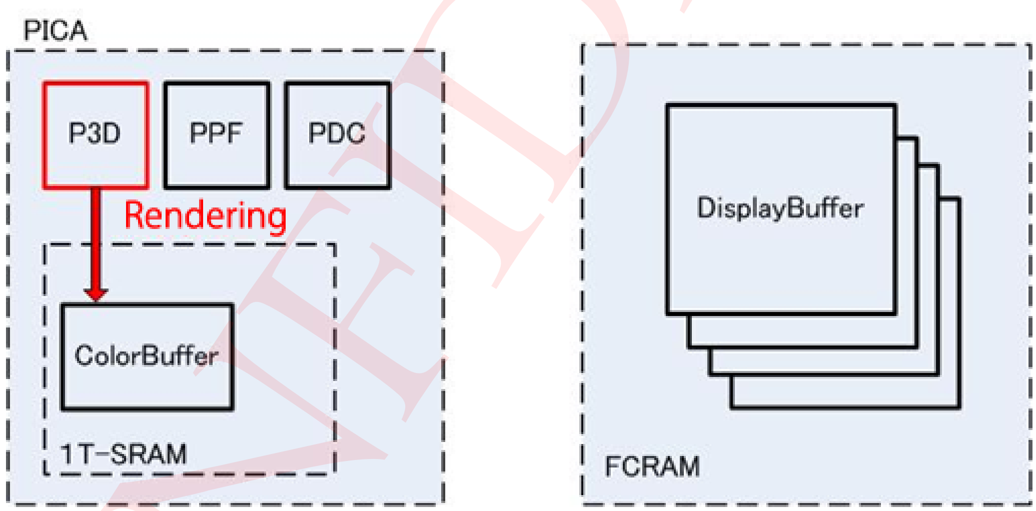
These features allow the CPU to create render commands several frames ahead of time without having to synchronize with the actual rendering. Furthermore, display buffers to which rendered results have been transferred can be displayed again any number of times.

4.1 Processing Flow from Rendering Through Display

4.1.1 Rendering

In this phase, multiple display buffers and a single color buffer are prepared, and data is rendered to the color buffer. (P3D is PICA's 3D rendering module.)

Figure 4-1 Rendering

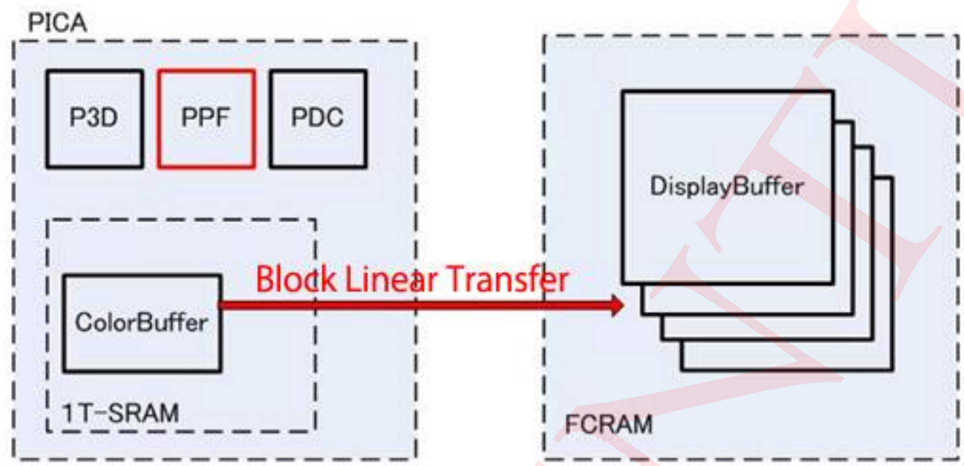


Note: The example shown in the figure assumes that the color buffer has been allocated in 1T-SRAM, and that the display buffers have been allocated in FCRAM.

4.1.2 Transferring Rendered Results

In this phase, rendered results are transferred to one or more display buffers by means of a block-linear transfer. Display buffers to which data has been transferred can then be displayed. (PPF stands for *PICA Post-Filter*, the module that performs post-filtering. This module converts rendered results from PICA's own native rendering format (block format) to the linear format used for display.)

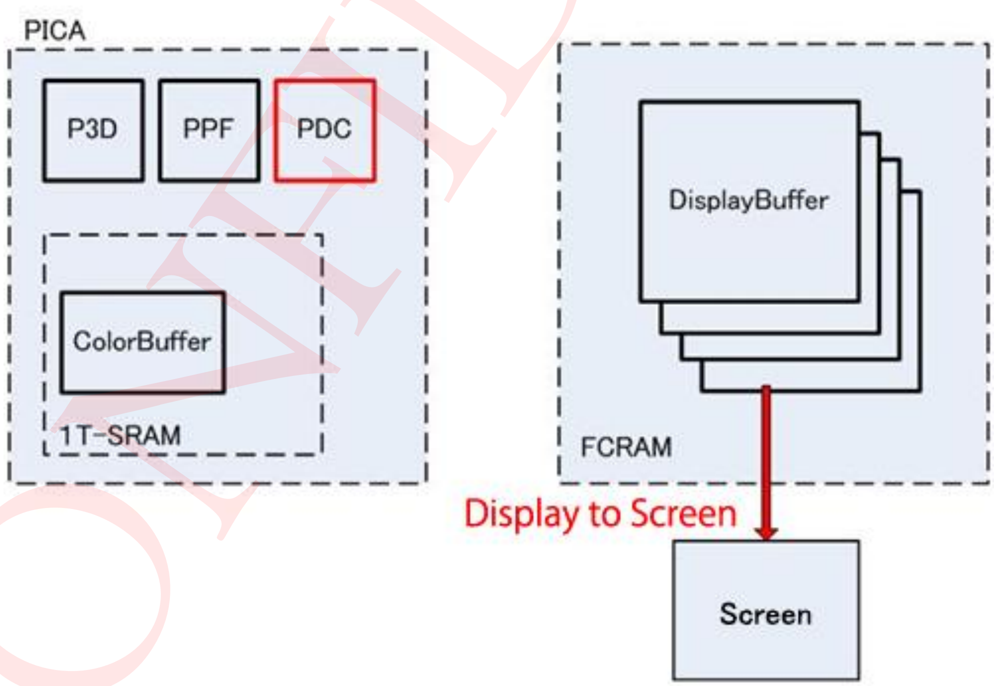
Figure 4-2 Transferring Rendered Results



4.1.3 Displaying

In this phase, the display buffer(s) to which the rendered results have been transferred are displayed. Switching between display buffers is done when V-Syncs occur. (PDC stands for *PICA LCD Controller*.)

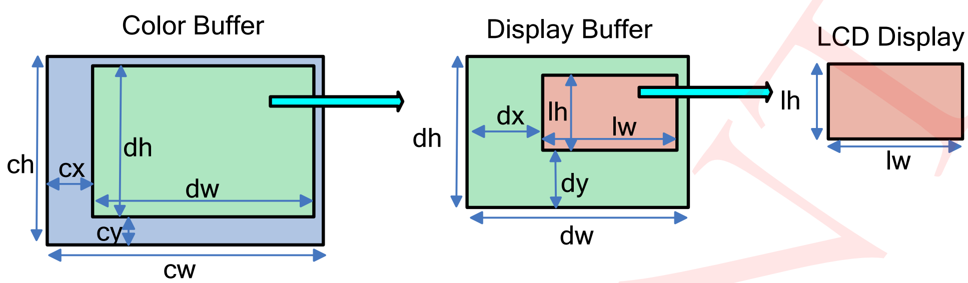
Figure 4-3 Displaying Images After Rendering



4.2 Specifying the Display Area

The figure below shows how the display area is specified during the transfer from the color buffer to the display buffer, and during the subsequent transfer from the display buffer to the LCD.

Figure 4-4 Specifying the Display Area



The blocks shown in the figure above are the color buffer, the display buffer, and the LCD display (from left to right). The dimensions **cw** and **ch** indicate the width and height of the color buffer, and these values are specified using the **`glRenderbufferStorage`** function. The area defined by the offsets **cx** and **cy** will be transferred to the display buffer. The offsets **cx** and **cy** are specified using the **`nngxTransferRenderImage`** function. The dimensions **dw** and **dh** indicate the width and height of the display buffer, and these values are specified using the **`nngxDisplaybufferStorage`** function. The area defined by the offsets **dx** and **dy** will be displayed to the LCD. The offsets **dx** and **dy** are specified using the **`nngxDisplayEnv`** function. The size of the display device is given by **lw** and **lh**.

4.3 API

This section describes the functions in the API.

4.3.1 Generating Display Buffer Objects

```
void nngxGenDisplaybuffers(GLsizei n, GLuint* buffers);
```

Generates display buffer objects. It generates ***n*** display buffer objects and stores the object names in ***buffers***. When a negative value is specified for ***n***, a `GL_ERROR_801C_DMP` error is generated. When memory failed to be allocated for the management region, a `GL_ERROR_801D_DMP` error is generated.

4.3.2 Deleting Display Buffer Objects

```
void nngxDeleteDisplaybuffers(GLsizei n, GLuint* buffers);
```

Deletes display buffer objects. Specifically, it will delete ***n*** display buffer objects whose names are stored in the ***buffers*** argument. If you attempt to delete the current display buffer object, a value of 0 is first bound to the current display buffer target. When a negative value is specified for ***n***, a `GL_ERROR_801E_DMP` error is generated.

4.3.3 Activating Display Targets

```
void nngxActiveDisplay(GLenum display);
```

Specify `NN_GX_DISPLAY0`, `NN_GX_DISPLAY1`, or `NN_GX_DISPLAY0_EXT` for the ***display*** argument. This will activate the specified display target and use the display buffer that is bound to the active display target for subsequent operations. When any other value is specified for ***display***, a `GL_ERROR_801F_DMP` error is generated.

4.3.4 Binding Display Buffers

```
void nngxBindDisplaybuffer(GLuint buffer);
```

Binds the display buffer object that is specified for the ***buffer*** argument. The binding target will be the display target that was activated using the **`nngxActiveDisplay`** function. It is used when allocating display buffer regions or when specifying which display buffer to display on the LCD. If a display buffer is bound using this function and the **`nngxSwapBuffers`** function is then called, the bound display buffer will be displayed. At that point, the display buffer that is bound to `NN_GX_DISPLAY0` will be displayed to screen 0, and the display buffer that is bound to `NN_GX_DISPLAY1` will be displayed on screen 1. A new display buffer object is generated when ***buffer*** refers to an unused object name. When memory fails to be allocated for the management region at this time, a `GL_ERROR_8020_DMP` error is generated.

4.3.5 Allocating Display Buffers

```
void nngxDisplaybufferStorage(
        GLenum format, GLsizei width, GLsizei height, GLenum area);
```

Allocates a memory region for the display buffer object that is bound to the currently active display target. Use the ***width*** and ***height*** arguments to specify the size of the display buffer. Use the ***format*** argument to specify one of the following display buffer formats:

- `GL_RGBA8_OES`: 32-bit R8G8B8A8
- `GL_RGB8_OES`: 24-bit R8G8B8
- `GL_RGBA4`: 16-bit R4G4B4A4
- `GL_RGB5_A1`: 16-bit R5G5B5A1
- `GL_RGB565`: 16-bit R5G6B5

Note that it is not possible to specify formats whose pixel sizes are larger than that of the color buffer. The values for the ***width*** and ***height*** arguments must be multiples of 8. However, an error occurs if the 32-block format is set and the **`nngxTransferRenderImage`** function is called with a display buffer that has a ***width*** and ***height*** that are not multiples of 32. If memory has already been allocated for the target display buffer object, that memory will be deallocated, and a new region will be allocated.

Use ***area*** to specify one of the following values as the location of the area being allocated.

- `NN_GX_MEM_FCRAM` Allocates the region from FCRAM
- `NN_GX_MEM_VRAMA` Allocates the region from the A channel in VRAM
- `NN_GX_MEM_VRAMB` Allocates the region from the B channel in VRAM

A GL_ERROR_8021_DMP error is generated when 0 is bound to the active display target. A GL_ERROR_8022_DMP error is generated when an invalid value is specified for ***width*** and ***height***. A GL_ERROR_8023_DMP error is generated when ***format*** is set equal to a value other than those listed in this section. A GL_ERROR_8024_DMP error is generated when ***area*** is set equal to a value other than those listed in this section. A GL_ERROR_8025_DMP error is generated when memory failed to be allocated for the display buffer.

4.3.6 Specifying the Display Area

```
void nngxDisplayEnv(GLint displayx, GLint displayy);
```

Specifies the area of the active display target's display buffer to display. The coordinates (***displayx***, ***displayy***) are used to specify the starting positions of the display area within the display buffer. (This will be the same size as the LCD's display area). The settings made using this function are not associated with display buffer objects and are set for each display screen (screen 0 and screen 1). When a negative value is set for either ***displayx*** or ***displayy***, a GL_ERROR_8026_DMP error is generated.

Values specified with this function affect the display buffer address set in the hardware, which must be aligned to a 16-byte address. When you set a value that conflicts with this restriction, an error is generated when the **nngxSwapBuffers** function is called. For details, see 4.3.8 Displaying Rendered Screens (Swapping).

4.3.7 Requesting Transfers of Rendered Results

```
void nngxTransferRenderImage(GLuint buffer, GLenum mode,
                             GLboolean yflip, GLint colorx, GLint colory);
```

Adds commands to the current command list that transfer rendering results from the current color buffer to the display buffer specified by ***buffer***. When the commands accumulated in the 3D command buffer have not been split, a split command is added before the transfer command.

A GL_ERROR_8027_DMP error is generated when 0 is bound to the current command list. A GL_ERROR_8028_DMP error is generated when the maximum number of accumulated command requests has been reached. A GL_ERROR_8029_DMP error is generated when a valid display buffer has not been bound. A GL_ERROR_802A_DMP error is generated when a valid color buffer has not been bound. A GL_ERROR_802F_DMP error is generated when the 3D command buffer is not large enough to add a split command.

The ***mode*** argument specifies the antialiasing mode using one of the following values.

- NN_GX_ANTIALIASE_NOT_USED   No antialiasing
- NN_GX_ANTIALIASE_2x1   Transfer using 2x1 antialiasing
- NN_GX_ANTIALIASE_2x2   Transfer using 2x2 antialiasing

When any other value is specified for ***mode***, a GL_ERROR_802B_DMP error is generated.

If ***yflip*** is GL_TRUE, the transferred image will be flipped in the y-direction. Any non-zero value specified for the ***yflip*** argument will be treated in the same way as if GL_TRUE had been specified.

An area the size of the display buffer is transferred from the color buffer to the display buffer. The starting positions of the data in the color buffer to transfer are specified using the coordinates (***colorx***, ***colory***). When the width[1] and the height[2] of the region of the color buffer to transfer are smaller than the width and height of the display buffer, a `GL_ERROR_802C_DMP` error is generated. When ***mode*** is set to `NN_GX_ANTIALIASE_2x1`, and the width of the region of the color buffer to transfer is less than twice the width of the display buffer, a `GL_ERROR_802C_DMP` error is generated. When ***mode*** is set to `NN_GX_ANTIALIASE_2x2`, and the width and height of the region of the color buffer to transfer are less than twice the width and height of the display buffer, a `GL_ERROR_802C_DMP` error is generated.

For the 8-block format, the arguments ***colorx*** and ***colory*** must both be positive integer multiples of eight. For the 32-block format, they must both be positive integer multiples of 32. Specifying any other value will cause a `GL_ERROR_802D_DMP` error.

A `GL_ERROR_802E_DMP` error is generated when the size of the display buffer (in pixels) where the data is being copied is greater than the size of the color buffer (in pixels) from which the data is being copied. A `GL_ERROR_8059_DMP` error is generated when the source color buffer or the destination display buffer has a width or height that is not a multiple of 32 while the 32-block format is set.

A `GL_ERROR_805A_DMP` error is generated when a color buffer is transferred to a display buffer that uses 24-bit pixels and the 8-block format when either buffer has a width or height that is not a multiple of 16.

When the current color buffer was rendered when the 32-block format was set, the 32-block format must be set when calling this function as well. The same applies to the 8-block format. When the block format setting when this function is called is not the same as the block format setting used when the color buffer was rendered, the rendered results will not come out correctly. The block format setting is configured using the **`glRenderBlockModeDMP`** function. For details, see the *DMPGL 2.0 Specification*.

4.3.8 Displaying Rendered Screens (Swapping)

```
void nngxSwapBuffers(GLenum display);
```

Displays the bound display buffer to the display target specified by ***display*** when the next V-Sync occurs. When `NN_GX_DISPLAY0` is specified for ***display***, only screen 0 (the first screen) will be processed. When `NN_GX_DISPLAY1` is specified, only screen 1 (the second screen) will be processed. When `NN_GX_DISPLAY_BOTH` is specified, both screens will be processed. When any other value is specified for ***display***, a `GL_ERROR_8030_DMP` error is generated. A `GL_ERROR_8031_DMP` error is generated when a valid display buffer has not been bound. A `GL_ERROR_8032_DMP` error is generated when the **`nngxDisplayEnv`** function sets a display region that falls outside of the display buffer that will be displayed.

[1] The width of the color buffer in pixels minus `colorx`

[2] The height of the color buffer in pixels minus `colory`

This function can be called at any time. Once this call has finished executing, it will display the display buffer that was bound at the time of the call once the first V-Sync occurs. If this function is called multiple times before the V-Sync occurs, only the most recent call will be applied.

This function sets a value in hardware indicating the address of the display buffer to show. The display buffer address that is ultimately set in hardware is calculated from the address allocated by the **nngxDisplaybufferStorage** function with consideration for the display buffer's resolution and pixel size, the LCD resolution, the offset values set by the **nngxDisplayEnv** function, and so on. The address set in the hardware must be 16-byte aligned. A GL_ERROR_8053_DMP error is generated for settings that conflict with this restriction. The address set in hardware is calculated by the following equation.

Equation 4-1 Display Buffer Address in Hardware

$$allocaddr + pixelsize \times (dbwidth \times (dbheight - lcdheight - displayy) + displayx)$$

In Equation 4-1 $allocaddr$ is the address allocated by the **nngxDisplaybufferStorage** function; $pixelsize$ is the number of bytes per pixel in the display buffer; $dbwidth$ and $dbheight$ are the width and height of the display buffer resolution; $lcdheight$ is the height of the LCD screen resolution; and $displayx$ and $displayy$ correspond to the ***displayx*** and ***displayy*** values in the **nngxDisplayEnv** function.

A GL_ERROR_9000_DMP error is generated when the display mode is NN_GX_DISPLAYMODE_STEREO and NN_GX_DISPLAY0_EXT is bound to either 0 or a display buffer without an allocated region. A GL_ERROR_9001_DMP error is generated when the display mode is NN_GX_DISPLAY_MODE_STEREO and the **nngxDisplayEnv** function specifies a display region outside of the display buffer. A GL_ERROR_9002_DMP error is generated when the display mode is NN_GX_DISPLAYMODE_STEREO and the display buffers bound to NN_GX_DISPLAY0 and NN_GX_DISPLAY0_EXT have a different resolution, format, or memory region.

4.3.9 Getting Parameters for Display Buffer Objects

```
void nngxGetDisplaybufferParameteri(GLenum pname, GLint* param);
```

Gets the parameters for the object bound to the active display target and stores them in ***param***. The settings are listed below. When values not listed in the table below are set for the ***pname*** parameter, a GL_ERROR_8033_DMP error is generated.

Table 4-1 List of Parameters for Display Buffer Objects

pname	Description
NN_GX_DISPLAYBUFFER_ADDRESS	Gets the address of the display buffer.
NN_GX_DISPLAYBUFFER_FORMAT	Gets the format of the display buffer.
NN_GX_DISPLAYBUFFER_WIDTH	Gets the width of the display buffer.
NN_GX_DISPLAYBUFFER_HEIGHT	Gets the height of the display buffer.

4.3.10 Display Mode Settings

```
void nngxSetDisplayMode(GLenum mode);
```

Sets the display mode. You can specify either `NN_GX_DISPLAYMODE_NORMAL` or `NN_GX_DISPLAYMODE_STEREO` for ***mode***. A `GL_ERROR_9003_DMP` error occurs when any other value is specified.

When the display mode is `NN_GX_DISPLAYMODE_NORMAL`, 400 lines of the display buffer are shown normally on screen 0. When the display mode is `NN_GX_DISPLAYMODE_STEREO`, the two display buffers are displayed stereoscopically to screen 0 for the left and right eyes. Screen 1 is unaffected.

The display target `NN_GX_DISPLAY0_EXT` is used when the display mode is `NN_GX_DISPLAYMODE_STEREO`. The display buffers bound to `NN_GX_DISPLAY0` and `NN_GX_DISPLAY0_EXT` are used for the left and right eyes, respectively. As with other display targets, use the **`nngxActiveDisplay`**, **`nngxBindDisplaybuffer`**, and **`nngxDisplayEnv`** functions to activate, bind a display buffer to, and specify a display region for `NN_GX_DISPLAY0_EXT`, respectively. The display buffers for the left and right eyes must have the same resolution and format and be placed in the same memory region. When any of these settings are different, an error occurs when the **`nngxSwapBuffers`** function is called.

The default display mode setting is `NN_GX_DISPLAYMODE_NORMAL`.

The display target macros `NN_GX_DISPLAY0_LEFT` and `NN_GX_DISPLAY0_RIGHT` are also prepared as aliases for `NN_GX_DISPLAY0` and `NN_GX_DISPLAY0_EXT`, respectively.

5 Command List Extended API

This chapter explains the extended API related to the command list objects handled in Chapter 3 Execution Control API. This API allows you to reuse executed commands that are generated by the DMPGL 2.0 API. Because you are reusing the commands themselves, you avoid the cost of calling DMPGL 2.0 functions that are normally required to generate them, which in turn reduces the CPU load. Hereafter, the mechanism for reusing command list objects is called the *command cache*.

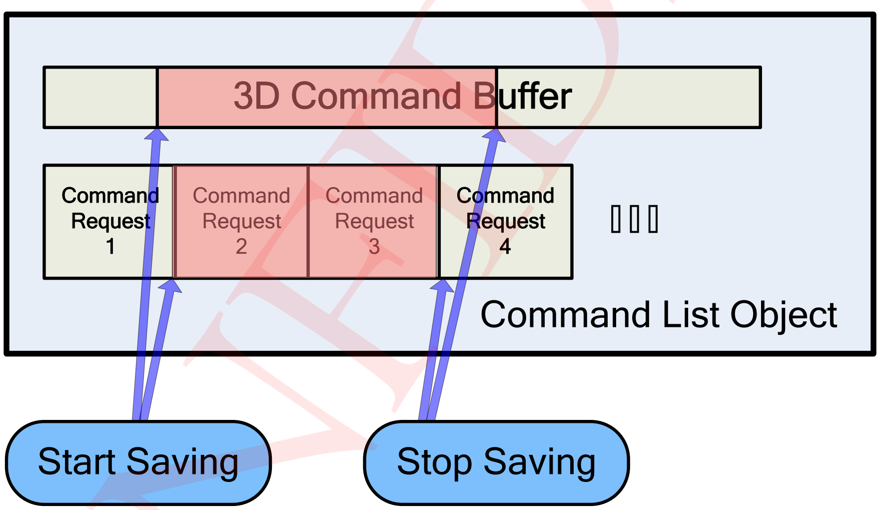
5.1 Saving and Reusing Command List Objects

The command cache API represents a process for accumulating commands in command list objects. You can specify when to start and stop saving commands, and then reuse the saved commands. It is actually the command requests and 3D command buffers maintained by command list objects that are saved and reused.

5.1.1 Saving Commands

To save commands, call the functions to start and stop saving commands as they accumulate in a command list. For more details, see sections 5.4.1 Start Saving Command Lists and 5.4.2 Stop Saving Command Lists.

Figure 5-1 Saving Command List Objects



The red area in the figure indicates the commands that have been saved. You get the following save information when you call the function to stop saving commands: the address at which you started saving the 3D command buffer, the save size, the ID at which you started saving command requests, and the number of command requests saved. To reuse saved commands, specify (as a set) the save information obtained when you stopped saving, along with the command list object from which content was saved.

When this feature "saves commands" it is actually recording accumulated command information. Command data itself is not saved outside of the region in which the command list object accumulates commands. You therefore cannot reuse a saved command list object that has been deleted or cleared.

You can start and stop saving a single command list object as many times as you like.

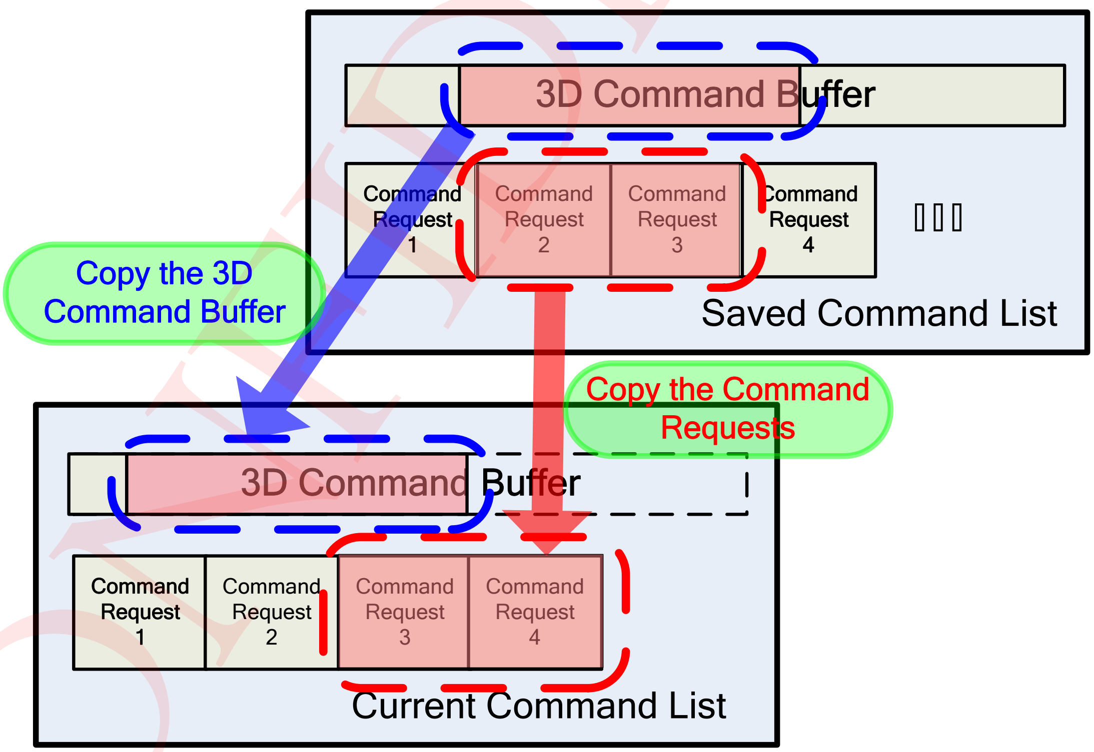
5.1.2 Using Saved Commands

You can call a function to use saved commands (see section 5.4.3 Using Saved Command Lists). When the function to use saved commands is called, saved commands are added to the command list object that is currently bound. Command requests are added to the current command list object object as copies. You can choose whether or not to copy the 3D command buffer to the current command list object.

5.1.2.1 The Method That Copies the 3D Command Buffer

With the method that copies the 3D command buffer, the saved 3D command buffer is copied by the CPU to the current 3D command buffer. We recommend that you use a small 3D command buffer to minimize the CPU load. If some of the copied command requests are 3D execution commands, their execution address information is converted from the original 3D command buffer to the copied 3D command buffer.

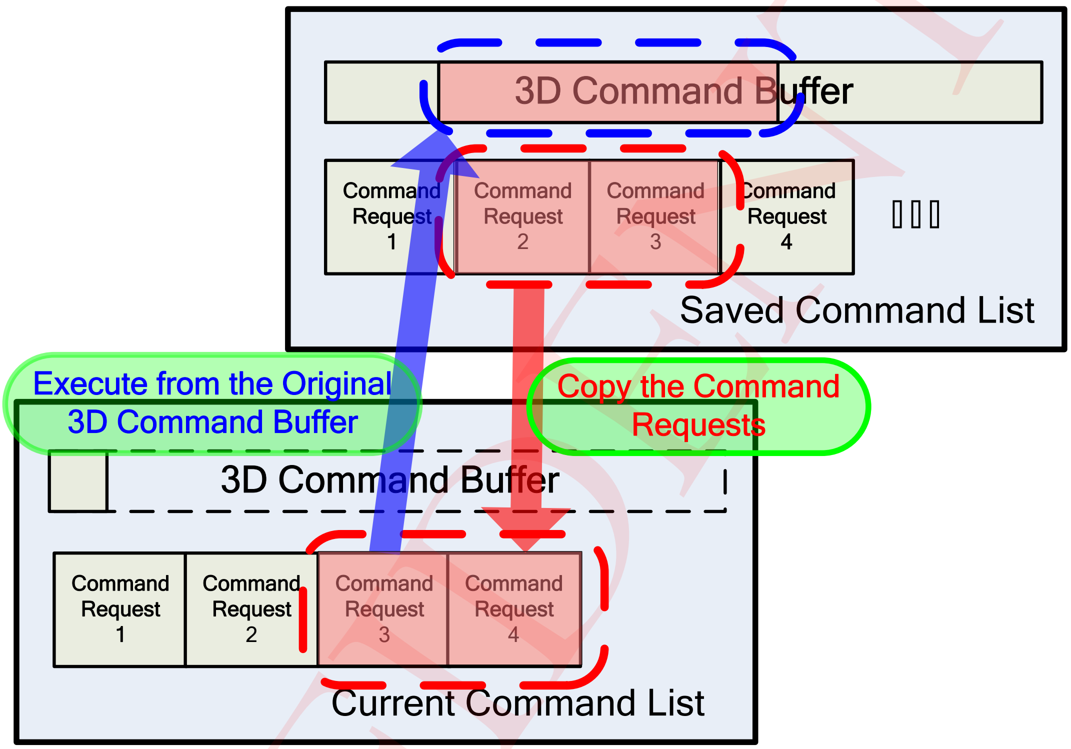
Figure 5-2 Using a Copy of a Saved 3D Command Buffer



5.1.2.2 The Method That Does Not Copy the 3D Command Buffer

With the method that does not copy the 3D command buffer, only command requests are copied and PICA directly accesses the 3D command buffer where it is saved. Because the CPU does not copy the 3D command buffer, we recommend that you use this method for a low CPU load and a large 3D command buffer. Copied 3D execution commands use their 3D command buffer execution address information unchanged. (However, the execution address of the first 3D execution command is converted to the address at which saving began.)

Figure 5-3 Using a Saved 3D Command Buffer Directly



The 3D command buffer execution address moves from the current 3D command buffer to the saved 3D command buffer when commands are executed. Once execution has finished in the saved 3D command buffer, it continues again from the current 3D command buffer. A split command must be inserted in the current 3D command buffer when its address returns to the current.

5.1.2.3 Copied Command Request Information

Though you can choose whether to copy the 3D command buffer, command requests are always copied. Command requests maintain information that is fixed for each command type and is all copied. This information does not change when a command is copied, even if the DMPGL state has changed since it was saved. However, information may change for the first 3D execution command to be copied.

- **DMA Transfer Commands**: The original and destination addresses, as well as the transfer size, are preserved for a DMA transfer.

- **3D Execution Commands**: The execution starting address and execution size in the 3D command buffer are preserved. When the address of the 3D command buffer at which saving began is not identical to the execution starting address, when the command request is copied, the execution starting address is replaced with the starting save address. The execution size is also changed to match.
- **Memory Fill Commands**: The starting address, size, and clear color are preserved for the color buffer to fill. The starting address, size, clear depth value, and clear stencil value are preserved for the depth stencil buffer.
- **Post-Transfer Commands**: The address, resolution, and format are preserved for both the source color buffer and the destination display buffer.
- **Render Texture Transfer Commands**: The address and resolution are preserved for both the source color buffer and the destination texture.

5.2 Editing Commands

You can edit a saved 3D command buffer directly to change commands. The 3D command buffer is a collection of commands that write to PICA registers. By replacing the data to write appropriately in accordance with register specifications, you can change settings that correspond to vertex shader uniforms, reserved fragment shader uniforms, and so on before execution. For details, see section 5.7 3D Command Buffer Specifications.

5.3 Other Features

The following features have been provided to make command list objects more convenient.

5.3.1 Importing and Exporting Command Lists

Commands accumulated in a command list object can be exported as binary data to a specified memory location. The exported data can be imported into any command list.

5.3.2 Copying Command List Objects

Commands accumulated in a command list object can be copied to another command list object.

5.3.3 3D Command Buffer Generation

A 3D command buffer is usually generated when a specific set of DMPGL 2.0 functions are called, but you can also generate the commands in 3D command buffers as complete sets of the commands relating to each feature.

Commands are normally generated only for states that have changed since commands were last generated (this is called *difference command generation*), but you can specify that all commands to be generated instead (this is called *complete command generation*).

With difference command generation, you also have the option to always generate commands related to the functions that have been called, regardless of whether the state has been changed.

5.3.4 Adding 3D Commands

You can copy any data to the current 3D command buffer to add commands. 3D execution commands can be added with the specified data region as the 3D command buffer's execution address.

5.4 API

This section describes each function in the API.

5.4.1 Start Saving Command Lists

```
void nngxStartCmdlistSave(void);
```

Starts saving the current command list object. You can get the information that is saved by using the **nngxStopCmdlistSave** function.

It is assumed that saved commands will be reused. As there is no way of knowing what the PICA register values will be when the 3D command buffer is reused, you must save all commands that need to be re-configured. If you call functions as usual when saving commands, only difference commands will be generated. Because difference commands are only generated for states whose settings have changed, some necessary commands may not be generated. To generate all of the necessary commands, either use complete commands or configure the command output mode.

Complete commands refer to commands that are entirely generated together for each state. Complete command generation is excessive because it generates all commands for each feature. For details, see section 5.4.9 Updating the DMPGL State.

You can configure the command output mode to always generate commands related to certain functions that are called, regardless of whether settings changed. For details, see section 5.4.10 Setting the Command Output Mode.

This combination of features allows you to generate and save the appropriate commands as necessary.

A `GL_ERROR_8034_DMP` error is generated when this function is called to save commands and then is called again before it finishes saving. A `GL_ERROR_8035_DMP` error is generated when 0 is bound to the current command list.

Calls to this function sometimes cause dummy commands to be generated in the 3D command buffer for padding.

5.4.2 Stop Saving Command Lists

```
void nngxStopCmdlistSave(
        GLuint* bufferoffset, GLsizei* buffersize,
        GLunit* requested, GLsizei* requestsize);
```

Stops saving the current command list object. When you stop saving commands, information is returned as follows: ***bufferoffset*** is the offset (in bytes) to the address the 3D command buffer's save start address; ***buffersize*** is the number of bytes saved in the 3D command buffer; ***requestid*** is the ID at which you started saving command requests; and ***requestsize*** is the number of command requests saved. Reuse command lists with this save information.

The offset (in bytes) to the save start address is returned in ***bufferoffset***, but this offset must be added to the starting address of the 3D command buffer to find the actual 3D command buffer save start address. To get the starting address of the 3D command buffer maintained by the command list that is currently bound, call the **`nngxGetCmdlistParameteri`** function with ***pname*** set to `NN_GX_CMDLIST_TOP_BUFADDR`.

Calling the **`nngxStopCmdlistSave`** function does not cause a split command to be generated in the 3D command buffer. Call the **`nngxSplitDrawCmdlist`** function explicitly if a split command is required. 3D execution command command requests may not be saved at all if the 3D command buffer has not been split. If the 3D command buffer does not have any split commands, you must use the copy method in order to reuse commands.

A `GL_ERROR_8036_DMP` error is generated when you have not started saving the command list.

Calls to this function sometimes cause dummy commands to be generated in the 3D command buffer for padding.

5.4.3 Using Saved Command Lists

```
void nngxUseSavedCmdlist(GLuint cmdlist,
        GLuint bufferoffset, GLsizei buffersize,
        GLunit requestid, GLsizei requestsize,
        GLbitfield statemask, GLboolean copycmd);
```

Adds saved commands to the current command list. Specify a saved command list for ***cmdlist***. Specify the save information obtained by the **`nngxStopCmdlistSave`** function for ***bufferoffset***, ***buffersize***, ***requestid***, and ***requestsize***. These are the offset (in bytes) from the starting address for saving the 3D command buffer, the number of bytes saved, the command request save start ID, and the number of command requests saved, respectively, but you should always specify the same set of values that you obtained from the **`nngxStopCmdlistSave`** function. The save information specified to this function is not checked for errors (whether it matches the value obtained when saving ended), so behavior is undefined if you specify invalid values.

Specify a bitwise OR of state flags for which to generate complete commands for ***statemask***. The DMPGL state and the actual PICA register settings will be in conflict after you call this function. To resolve this, you must generate all commands and re-set the PICA registers. It is sometimes redundant to generate all commands, however, so complete commands are generated only if they correspond to state flags specified by ***statemask***. For details on the state flags specified to ***statemask***, see section 5.5 State Flags.

When you specify `GL_TRUE` for ***copycmd***, the 3D command buffer copy method is used when commands are applied. When `GL_FALSE` is specified, the method that does not copy the 3D command buffer is used when commands are applied. For further details on behavior, see section 5.1.2 Using Saved Commands. If the method that does not copy 3D command buffer is used, only sections that have split commands properly configured are executed. Without a split command, execution would not otherwise return from the external 3D command buffer to the current command list. The 3D command buffer is ignored where it is not included in command request 3D execution commands. If you are using a command list with the method that does not copy the 3D command buffer, you must call the **nngxSplitDrawCmdlist** function to add a split command before you stop saving.

If the method that does not copy the 3D command buffer is used when commands are applied, the execution address will move from the 3D command buffer that is currently accumulating commands to an external 3D command buffer. The driver therefore calls the **nngxSplitDrawCmdlist** function to add a split command to the current 3D command buffer before it copies the command requests. The **nngxSplitDrawCmdlist** function is not called immediately after the current 3D command buffer is split.

When the copy method for the 3D command buffer is used, ***requestsize*** is nonzero, and the first command of the saved command requests is not a 3D execution command, the driver calls the **nngxSplitDrawCmdlist** function to add a split command to the current 3D command buffer before it copies the command list. The **nngxSplitDrawCmdlist** function is not called immediately after the current 3D command buffer is split.

A `GL_ERROR_8037_DMP` error is generated when 0 is bound to the current command list.
A `GL_ERROR_8038_DMP` error is generated when an invalid value is specified for ***cmdlist***.
A `GL_ERROR_8039_DMP` error is generated when ***cmdlist*** specifies the current command list.
A `GL_ERROR_803A_DMP` error is generated when this function has added saved commands past the maximum size of the current command list's 3D command buffer and command requests.

5.4.4 Exporting Command Lists

```
GLsizei nngxExportCmdlist(GLuint cmdlist,
       GLuint bufferoffset, GLsizei buffersize,
       GLuint requestid, GLsizei requestsize,
       GLsizei datasize, GLvoid* data);
```

Exports the command list specified by ***cmdlist*** into memory as binary data. A `GL_ERROR_803B_DMP` error is generated if an invalid value is specified for ***cmdlist***.

Specify the offset (in bytes) from the starting address of the 3D command buffer to the first address to export for ***bufferoffset***. Specify the number of bytes to export from the 3D command buffer for ***buffersize***. Specify the ID of the first command request to export for ***requestid***. Command request IDs start at 0 and increase sequentially in the order that commands are accumulated. Specify the number of command requests to export for ***requestsize***. To determine which values to specify for ***bufferoffset***, ***buffersize***, ***requestid***, and ***requestsize*** while commands are accumulating, call the **nngxGetCmdlistParameteri** function and get both the size of the accumulated 3D command buffer and the number of command requests. Set ***pname*** equal to `NN_GX_CMDLIST_USED_BUFSIZE` or `NN_GX_CMDLIST_USED_REQCOUNT` to get the size of the accumulated 3D command buffer or the number of accumulated command requests, respectively. The values specified for ***bufferoffset***,

buffersize, ***requestid***, and ***requestsize*** must not conflict with each other. To be safe, we recommend that you either export data based on the save information obtained by the **`nngxStopCmdlistSave`** function or use the values obtained by calling the **`nngxGetCmdlistParameteri`** function twice: once for ***bufferoffset*** and ***requested***, and once for ***buffersize*** and ***requestsize***.

Specify a pointer to a region used to store the exported data for ***data***. Specify the size (in bytes) of the ***data*** region for ***datasize***. Nothing is exported when the ***data*** argument is set equal to 0. The size (in bytes) of the exported data is returned.

You are expected to first call this function with ***data*** set equal to 0. Then, using the return value as the required data size (for exporting), allocate a data region and call this function again. A `GL_ERROR_803C_DMP` error is generated when the export data size is greater than ***datasize***.

You can call the **`nngxImportCmdlist`** function to import and use the exported data.

A `GL_ERROR_803D_DMP` error is generated when ***bufferoffset***, ***buffersize***, ***requestid***, and ***requestsize*** specify a region without any accumulated commands. A `GL_ERROR_803E_DMP` error is generated when ***bufferoffset*** or ***buffersize*** is not 8-byte aligned.

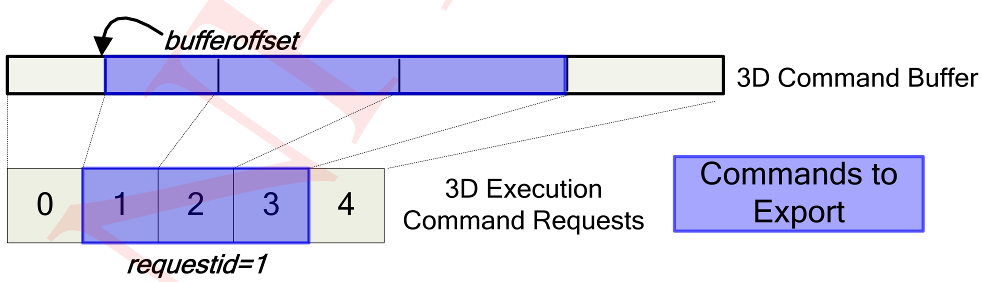
A `GL_ERROR_803F_DMP` error is generated when any of the command requests are 3D execution commands added using the **`nngxUseSavedCmdlist`** function without the copy method for the 3D command buffer.

A `GL_ERROR_8040_DMP` error is generated when ***bufferoffset*** or ***buffersize*** have not properly specified the 3D command buffer that is used to execute an exported 3D execution command.

The address of the 3D command buffer that begins the export must be specified within the region used to execute the first 3D execution command that is exported.

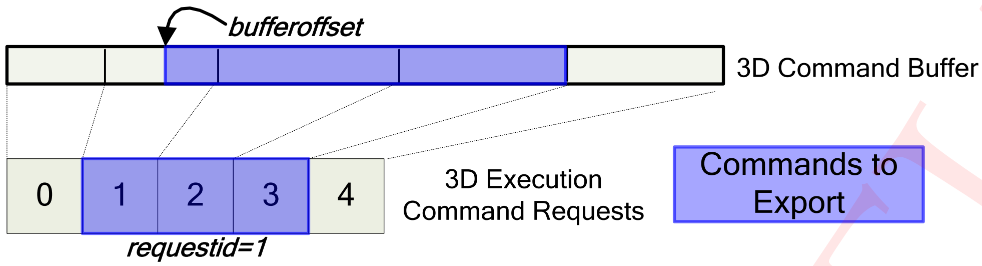
The following figure is an example of how to export correctly. This exports the entire 3D command buffer where the first 3D execution command (command 1) is executed.

Figure 5-4 First Example of Specifying an Export Correctly



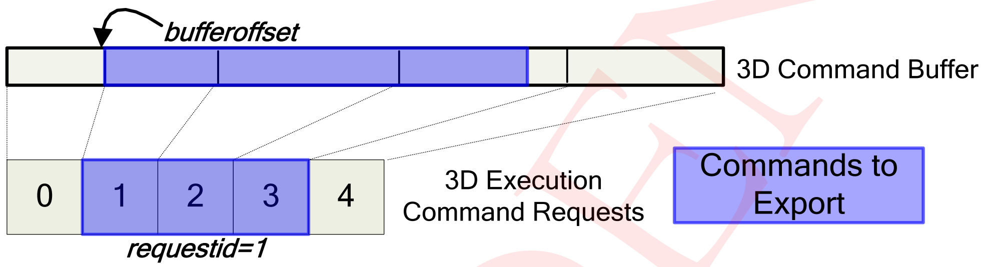
The following figure is also an example of how to export correctly. You can shift ***bufferoffset*** as long as it is within the region used to execute the first 3D execution command (command 1).

Figure 5-5 Second Example of Specifying an Export Correctly



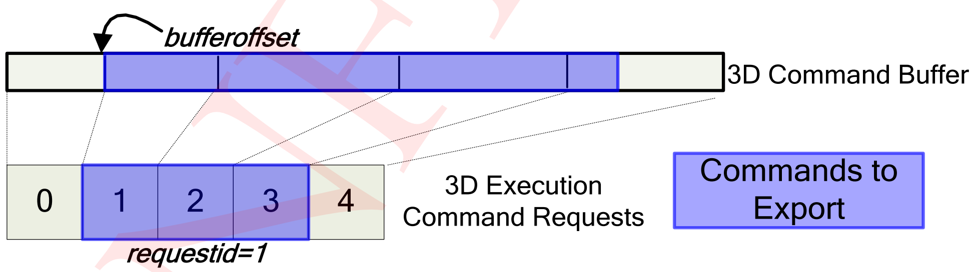
You must export all split commands run by 3D execution commands. The following figure is an example of how to export incorrectly. The split command for the 3D command buffer, executed by command 3, is not exported.

Figure 5-6 First Example of Specifying an Export Incorrectly



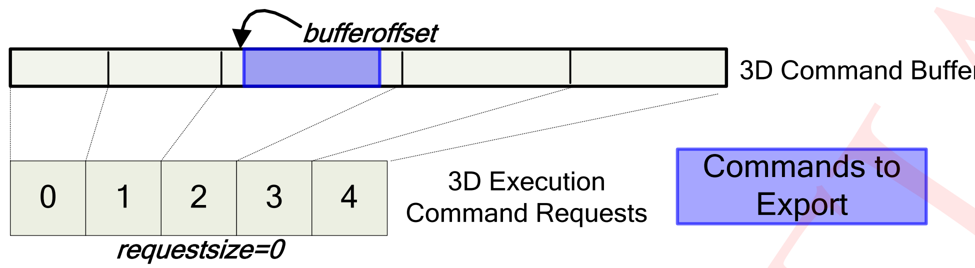
As long as it does not contain any split commands, you can export the 3D command buffer past the region where the last 3D execution command is executed. The following figure is an example of how to export correctly. You can export the 3D command buffer until just before the split command executed by command 4.

Figure 5-7 Third Example of Specifying an Export Correctly



If you do not export any 3D execution commands, the exported command buffer cannot contain any split commands. When you export data in a way that conflicts with this restriction, you will run into incorrect behavior when you import and use the data even though you will be unable to detect any errors. The following figure is an example of how to export correctly. If you do not export command requests, the exported 3D command buffer cannot contain any split commands.

Figure 5-8 Fourth Example of Specifying an Export Correctly



As mentioned before, this function exports a 3D command buffer whose content is not checked; consequently, the exported data may not behave correctly and errors may not be detected.

5.4.5 Importing Command Lists

```
void nngxImportCmdlist(GLuint cmdlist, GLvoid* data, GLsizei datasize);
```

Imports data exported by the **nngxExportCmdlist** function into a command list. Specify the command list object to import for ***cmdlist***. A `GL_ERROR_8041_DMP` error is generated when ***cmdlist*** is set to an invalid value.

Specify a pointer to the exported data for ***data***. Specify the size (in bytes) of the exported data for ***datasize***. A `GL_ERROR_8042` error is generated when ***data*** is a pointer to invalid data. A `GL_ERROR_8043` error is generated when ***datasize*** does not match the exported data size.

You can specify either the command list that is currently bound or an unbound command list for ***cmdlist***. The imported commands are added after any commands that have already been accumulated in ***cmdlist***. A `GL_ERROR_8044_DMP` error is generated when, by adding the imported commands, you have exceeded the maximum size for the 3D command buffer or command requests. If a 3D execution command is not the first command request that you import into a command list, bind that command list as the current one and then call the **nngxSplitDrawCmdlist** function to add a split command before calling this function. A `GL_ERROR_8045_DMP` error is generated when a 3D execution command is not the first command request imported into a (command list's) 3D command buffer that has not been split.

Calls to this function sometimes cause dummy commands to be generated for padding in the 3D command buffer of the command list into which you are importing data.

5.4.6 Getting Command List Information for Exported Data

```
void nngxGetExportedCmdlistInfo(GLvoid* data,
        GLsizei* buffersize, GLsizei* requestsize, GLuint* bufferoffset);
```

Gets the size of the 3D command buffer, the number of command requests, and the offset (in bytes) to the address at which the command buffer is stored in exported data. Specify a pointer to data exported by the **nngxExportCmdlist** function for ***data***. The ***buffersize*** argument gets the size (in bytes) of the 3D command buffer. The ***requestsize*** argument gets the number of command requests. The ***bufferoffset*** argument gets the offset (in bytes) to the region at which the 3D command buffer is stored in ***data***. A `GL_ERROR_8046_DMP` error is generated when ***data*** specifies invalid data.

5.4.7 Copying Command Lists

```
void nngxCopyCmdlist(GLuint scmdlist, GLuint dcmdlist);
```

Copies the commands accumulated in a command list. Specify the command list to copy for ***scmdlist*** and the destination command list for ***dcmdlist***. Commands copied into the command list overwrite any commands that have already been accumulated there.

A `GL_ERROR_8047_DMP` error is generated when ***dcmdlist*** is currently bound. A `GL_ERROR_8048_DMP` error is generated when ***scmdlist*** is an invalid value. A `GL_ERROR_8049_DMP` error is generated when ***dcmdlist*** is an invalid value. A `GL_ERROR_804A_DMP` error is generated when ***scmdlist*** and ***dcmdlist*** are the same value. A `GL_ERROR_804B_DMP` error is generated when ***dcmdlist*** is currently executing. An error is not generated if execution has finished or stopped. A `GL_ERROR_804C_DMP` error is generated when the size of the commands accumulated in ***scmdlist*** exceeds the maximum size of the 3D command buffer or the command requests in ***dcmdlist***.

5.4.8 Checking the DMPGL State and Generating Commands

```
void nngxValidateState (GLbitfield statemask, GLboolean drawelements);
```

Checks the DMPGL state and generates commands.

Commands are normally accumulated in the 3D command buffer when certain DMPGL 2.0 functions are called. Most of these commands are generated by the **glDrawElements** and **glDrawArrays** functions. DMPGL functions check the state and, if it is updated, generate the relevant commands. This is called *validation*. Nearly all states are validated at once by the **glDrawElements** and **glDrawArrays** functions, but you can validate particular groups of states with this function.

Specify a bitwise OR of the state flags to validate for ***statemask***. For more details on state flags, see section 5.5 State Flags. Specify `GL_TRUE` for ***drawelements*** when **glDrawElements** is called and `GL_FALSE` when **glDrawArrays** is called for actual rendering. To validate within states, it is sometimes necessary to know whether the **glDrawElements** or **glDrawArrays** function is used for rendering.

The **nngxValidateState** function generates commands when the specified states have been updated. You can use this function in combination with the **nngxUpdateState** function, which updates states, to generate complete commands related to states.

When you use this function to generate commands for individual states, the commands may not be generated in the same order as they originally would have been using the **glDrawElements** and **glDrawArrays** functions. Several state flags depend on others and must be specified accordingly. For details, see section 5.5.2 State Flag Dependencies.

A `GL_ERROR_8066_DMP` error is generated when there is an overflow in the 3D command buffer. A `GL_ERROR_806C_DMP` error is generated when verification causes various types of DMPGL errors.

The following conditions cause errors to occur during validation.

- *Texture memory has not been allocated for a valid texture.* You must call the **glTexImage2D**, **glCompressedTexImage2D**, or **glCopyTexImage2D** function to allocate texture memory. All six faces of a cube-map texture must be allocated.

- *A texture was bound with an invalid format.* Either a texture in the `GL_SHADOW_DMP` format was bound as Texture 1 or Texture 2, or a texture in the `GL_GAS_DMP` format was bound as a cube-map texture.
- *The six faces of a cube-map texture use different settings.* All six faces of a cube-map texture must have the same width, height, format, and number of mipmap levels.
- *The six faces of a cube-map texture have addresses that do not share a common value in the most-significant 7 bits.* The most-significant 7 bits of every face's address must be identical.
- *A lookup table object has not been bound correctly or a lookup table number has not been specified correctly.* A valid lookup table object must be bound to the appropriate lookup table number for fragment lighting, procedural textures, fog, and gas when they are configured to use lookup tables. The uniforms that specify the lookup table numbers must also be set correctly.
- *The region required for storing the value of the internal lookup table format failed to be allocated.*

5.4.9 Updating the DMPGL State

```
void nngxUpdateState (GLbitfield statemask);
```

Updates the DMPGL state. Complete commands are generated during validation when you use this function to update the state.

The **`glDrawElements`** and **`glDrawArrays`** functions check the DMPGL state and, if it is updated, generate the relevant commands. Commands are not usually generated when the state has not been updated. Once you call this function, the state is updated and complete commands are configured to be generated. This function does not itself generate commands. Commands are generated when a function such as **`glDrawElements`** or **`glDrawArrays`** is called after this one.

After you call this function, complete commands are generated until the first call to the **`glDrawElements`** or **`glDrawArrays`** functions. If you call the **`nngxValidateState`** function before the **`glDrawElements`** or **`glDrawArrays`** function, complete commands cease to be generated for each validated state flag.

Specify a bitwise OR of the state flags to update for ***statemask***. For more information on state flags, see section 5.5 State Flags.

You can use this function in combination with the **`nngxValidateState`** function to generate complete commands for individual state flags.

5.4.10 Setting the Command Output Mode

```
void nngxSetCommandGenerationMode(GLenum mode);
```

Sets the command output mode.

If you specify `NN_GX_CMDGEN_MODE_CONDITIONAL` for ***mode***, commands are generated only for states that have changed. If you specify `NN_GX_CMDGEN_MODE_UNCONDITIONAL` for ***mode***, commands are generated not only for states that have changed but also for functions that are called, regardless of whether the state changed.

The mode is set to `NN_GX_CMDGEN_MODE_CONDITIONAL` by default.

The following settings are affected by the `NN_GX_CMDGEN_MODE_UNCONDITIONAL` mode.

- Uniform settings for the reserved fragment shader.
- Integer uniform settings for the vertex shader.
- Settings for lookup table data. If you set reserved uniform values that specify various lookup table IDs, commands are generated during validation to load lookup tables. However, each lookup table must be enabled. For details, see section 5.5.3 Lookup Table Command Generation.
- Functions other than **glDrawArrays**, **glDrawElements**, and **nngxValidateState** that generate commands. For details, see section 5.6 DMPGL Functions That Generate Commands.

A `GL_ERROR_804D_DMP` error is generated if an invalid value is specified for ***mode***.

5.4.11 Getting the Command Output Mode

```
void nngxGetCommandGenerationMode(GLenum* mode);
```

Gets the currently set command output mode and returns it in the ***mode*** argument.

5.4.12 Adding 3D Commands

```
void nngxAdd3DCommand (
       const GLvoid* bufferaddr, GLsizei buffersize, GLboolean copycmd);
```

Adds data from the specified region to the current 3D command buffer or adds a 3D execution command that executes the specified region.

When ***copycmd*** is `GL_TRUE`, the data in the region specified by ***bufferaddr*** is copied to the current 3D command buffer. Specify the number of bytes to copy for ***buffersize***. Behavior is not guaranteed when a 3D command buffer with split commands is copied.

When ***copycmd*** is `GL_FALSE`, a 3D execution command is first generated with the region specified by ***bufferaddr*** as its execution address and then added to the current command requests. Specify the number of bytes in the 3D command buffer to execute for ***buffersize***. If unsplit 3D commands have accumulated in the current 3D command buffer, the **nngxSplitDrawCmdlist** function is called internally, and then a newly created 3D execution command is added. Behavior is not guaranteed if the last command in the specified region is not a split command.

You must specify a positive value for ***buffersize***. When ***copycmd*** is `GL_TRUE`, ***buffersize*** must be a multiple of 4. When ***copycmd*** is `GL_FALSE`, ***buffersize*** must be a multiple of 16.

The following errors will be generated under the conditions specified.

Error	Generated When
`GL_ERROR_804E_DMP`	A command list is not currently bound
`GL_ERROR_804F_DMP`	***buffersize*** is an invalid value
`GL_ERROR_8050_DMP`	***copycmd*** is `GL_TRUE` and the current 3D command buffer size is insufficient
`GL_ERROR_8051_DMP`	***copycmd*** is `GL_FALSE` and the current command request size is insufficient

Error	Generated When
`GL_ERROR_8052_DMP`	***copycmd*** is `GL_FALSE` and ***bufferaddr*** is not a multiple of 16

5.4.13 Adding a Copied Command List

```
void nngxAddCmdlist ( GLuint cmdlist );
```

Adds all commands accumulated in the command list specified by ***cmdlist*** to the current command list object. The commands are added after any commands that have already been accumulated in the current command list object.

If the current 3D command buffer has not just been split and a 3D execution command is *not* the first command request to add, this function calls the **`nngxSplitDrawCmdlist`** function internally to split the command buffer before adding commands.

If the current 3D command buffer has not just been split and a 3D execution command *is* the first command request to add, this function adds dummy commands to the current command buffer as necessary to adjust the alignment before adding commands.

The following errors will be generated under the conditions specified.

Error	Generated When
`GL_ERROR_8054_DMP`	An invalid value is specified for ***cmdlist***
`GL_ERROR_8055_DMP`	A command list is not currently bound
`GL_ERROR_8056_DMP`	The command list specified by ***cmdlist*** is the same as the current command list
`GL_ERROR_8057_DMP`	The current command list is currently being executed
`GL_ERROR_8058_DMP`	By adding a command buffer or command requests to the current command list, the maximum buffer size has been exceeded.

The maximum size is checked when this function calls the **`nngxSplitDrawCmdlist`** function internally, when dummy commands are added, and in other instances.

5.4.14 Getting the Updated DMPGL State

```
void nngxGetUpdatedState ( GLbitfield* statemask );
```

Gets the updated DMPGL state.

Each of the DMPGL states is updated when DMPGL functions and the **`nngxUpdateState`** function are called. When you call this function, any state flag that has currently been updated is set to 1 in ***statemask***. If you call this after the state has been validated by a function such as **`glDrawArrays`**, **`glDrawElemnts`**, or **`nngxValidateState`**, the validated state flags are not set in ***statemask***. This function sets the `NN_GX_STATE_OTHERS` state flag only when it has been updated by the **`nngxUpdateState`** function.

For more details on state flags, see section 5.5 State Flags.

5.4.15 Invalidating DMPGL State Updates

```
void nngxInvalidateState ( GLbitfield statemask );
```

Disables updates to the DMPGL state. For the ***statemask*** argument, specify the bitwise sum of the state flags for which you want to disable updates.

Calling this function will prevent generation of commands related to the state flags specified in the ***statemask*** argument, even if the state is updated.

For more details on state flags, see section 5.5 State Flags.

5.5 State Flags

State flags consolidate settings by feature for DMPGL function calls. A single state corresponds to one or more DMPGL functions or uniforms (and so on) and, when updated, causes relevant commands to be generated. You must specify a bitwise OR of state flags as an argument to the **nngxUseSavedCmdlist**, **nngxValidateState**, and **nngxUpdateState** functions.

5.5.1 State Flag Types

Each type of state flag is related to different commands and has different DMPGL 2.0 functions that cause it to be updated. Table 5-1 summarizes each of the state flag types.

Table 5-1 State Flag Types

State Flag Name	Summary
NN_GX_STATE_SHADERBINARY	The shader binary state. Commands are generated to load shader assembly code. This state is updated when the **glUseProgram** function switches the program and the original and new program objects are linked to shader objects that were loaded by separate calls to the **glShaderBinary** function.
NN_GX_STATE_SHADERPROGRAM	The shader program state. Commands are generated for settings that include the composition of vertex attributes. This state is updated when the **glUseProgram** function switches the program object. Commands are generated only for registers whose settings have changed. When this state is validated, its current settings are compared with its settings when it was last validated; commands are generated only for the settings that are different.
NN_GX_STATE_SHADERMODE	The shader mode state. Commands are generated to enable or disable the geometry shader. This state is updated when the **glUseProgram** function toggles the geometry shader between enabled and disabled.
NN_GX_STATE_SHADERFLOAT	The shader floating-point state. Commands are generated to set floating-point registers for which a `def` instruction has defined a value in shader assembly. This state is updated when the **glUseProgram** function switches to a program object with a different shader object attached.

State Flag Name	Summary
`NN_GX_STATE_VSUNIFORM`	The vertex shader uniform state. Commands are generated to set floating-point registers, Boolean registers, and integer registers defined as uniforms in shader assembly. This state is updated when the: • **`glUseProgram`** function switches to a different program object. • **`glUniform`** function sets the value of a vertex shader uniform. The state is updated even if settings have not changed for floating-point uniforms, but it is not updated if settings have not changed for integer uniforms.
`NN_GX_STATE_FSUNIFORM`	The reserved fragment shader uniform state. Commands are generated to set registers specific to reserved fragment shader uniforms. This state is updated when a uniform value is changed because the **`glUseProgram`** function switched the program object or the **`glUniform`** function set a fragment shader uniform.
`NN_GX_STATE_LUT`	The lookup table state. Commands are generated to set lookup tables. This state is updated when the: • content of a lookup table object bound by the **`glBindTexture`**, **`glTexImage1D`**, or **`glTexSubImage1D`** function changes. • **`glDeleteTextures`** function deletes a bound lookup table object. • **`glUseProgram`** or **`glUniform`** function changes the lookup table object specifying the uniforms used to set each lookup table ID.
`NN_GX_STATE_TEXTURE`	The texture state. Commands specific to texture units are generated. This does not include commands specific to procedural textures. This state is updated when the following functions are called. • **`glBindTexture`** • **`glTexImage2D`** • **`glCompressedTexImage2D`** • **`glCopyTexImage2D`** • **`glCopyTexSubImage2D`** • **`glTexParameter`** This state is also updated when the: • **`glDeleteTextures`** function deletes texture objects in use. • **`glUseProgram`** or **`glUniform`** function changes the reserved fragment uniform, `dmp_Texture[i].samplerType`.
`NN_GX_STATE_FRAMEBUFFER`	The framebuffer information state. Commands specific to the framebuffer format and buffer address are generated. This state is updated when the following functions are called. • **`glBindFramebuffer`** • **`glBindFramebufferRenderbuffer`** • **`glFramebufferTexture2D`** • **`glDeleteFramebuffers`** • **`glBindRenderbuffer`** • **`glRenderbufferStorage`** • **`glDeleteRenderbuffers`**

State Flag Name	Summary
NN_GX_STATE_VERTEX	The vertex attribute data state. Commands specific to vertex attribute data are generated. This state is updated when the following functions are called. • **glBindBuffer** • **glBufferData** • **glBufferSubData** • **glEnableVertexAttribArray** • **glDisableVertexAttribArray** • **glVertexAttribPointer** • **glVertexAttrib** • **glUseProgram** This state is also updated when the **glDeleteBuffers** function deletes the current vertex buffer.
NN_GX_STATE_TRIOFFSET	The polygon offset state. Commands specific to polygon offsets are generated. This state is updated when the: • **glEnable** or **glDisable** function changes the GL_POLYGON_OFFSET_FILL setting. • **glDepthRangef** or **glPolygonOffset** function changes settings. • **glUseProgram** function is called.
NN_GX_STATE_FBACCESS	The framebuffer access method state. Commands are generated for the framebuffer's R/W and other access methods. This state is updated when the: • **glEnable** or **glDisable** function changes the GL_COLOR_LOGIC_OP, GL_BLEND, GL_DEPTH_TEST, GL_EARLY_DEPTH_TEST_DMP, or GL_STENCIL_TEST setting. • **glDepthFunc**, **glEarlyDepthFuncDMP**, **glColorMask**, **glDepthMask**, or **glStencilMask** function changes settings. • **glUniform** function sets the reserved fragment uniform dmp_FragOperation.mode.
NN_GX_STATE_SCISSOR	A scissoring-related state. Commands specific to scissoring settings are generated. This state is updated when the: • **glEnable** or **glDisable** function changes the GL_SCISSOR_TEST setting. • **glScissor** function changes settings. • framebuffer size is changed with scissoring enabled.
NN_GX_STATE_OTHERS	This state flag represents a state related to commands generated by functions other than the **glDrawElements** and **glDrawArrays** functions. For details, see section 5.6 DMPGL Functions That Generate Commands.

Commands are generated by the first call to the **glDrawElements**, **glDrawArrays**, or **nngxValidateState** function after any state represented by a state flag is updated. Commands are also generated by a call to the **glReadPixels** or **glClear** function for the state flag NN_GX_STATE_FRAMEBUFFER.

5.5.2 State Flag Dependencies

Each state flag has commands related to it, some of which have dependencies on the order in which they are specified. When you call the **nngxValidateState** function, commands are generated in the order that they were specified by the application. This sometimes conflicts with dependency restrictions. The following table shows dependency restrictions that apply when the **nngxValidateState** function is called. Behavior is undefined when there is a conflict with these restrictions.

Table 5-2 State Flag Dependencies

First Value (Do Not Specify After the Second Value)	Second Value (Do Not Specify Before the First Value)
• NN_GX_STATE_FBACCESS • NN_GX_STATE_TRIOFFSET	• NN_GX_STATE_FSUNIFORM
• NN_GX_STATE_SHADERMODE	• NN_GX_STATE_SHADERBINARY • NN_GX_STATE_SHADERPROGRAM • NN_GX_STATE_SHADERFLOAT • NN_GX_STATE_VSUNIFORM
• NN_GX_STATE_FRAMEBUFFER • NN_GX_STATE_OTHERS	• NN_GX_STATE_FBACCESS
• NN_GX_STATE_FRAMEBUFFER	• NN_GX_STATE_SCISSOR

5.5.3 Lookup Table Command Generation

Commands that update lookup table data are only generated for enabled lookup tables. Commands are not generated for disabled lookup tables even if you call the **nngxUseSavedCmdlist** or **nngxUpdateState** function while complete commands are configured to be generated for the state flag NN_GX_STATE_LUT. The following table shows how to enable the various lookup tables.

Table 5-3 Conditions for Enabling Lookup Tables

Lookup Table	Conditions to Enable
Fragment Light: Distribution 0 (D0)	• dmp_FragmentLighting.enabled is GL_TRUE and • dmp_LightEnv.config is configured to use D0 and • dmp_LightEnv.lutEnabledD0 is GL_TRUE
Fragment Light: Distribution 1 (D1)	• dmp_FragmentLighting.enabled is GL_TRUE and • dmp_LightEnv.config is configured to use D1 and • dmp_LightEnv.lutEnabledD1 is GL_TRUE
Fragment Light: Spotlight Attenuation (SP)	• dmp_FragmentLighting.enabled is GL_TRUE and • dmp_LightEnv.config is configured to use SP and • dmp_FragmentLightSource[i].enabled is GL_TRUE and • dmp_FragmentLightSource[i].spotEnabled is GL_TRUE

Lookup Table	Conditions to Enable
Fragment Light: Fresnel Factor (FR)	• `dmp_FragmentLighting.enabled` is `GL_TRUE` and • `dmp_LightEnv.config` is configured to use FR and • `dmp_LightEnv.fresnelSelector` is not `GL_LIGHT_ENV_NO_FRESNEL_DMP`
Fragment Light: Reflection Red (RR)	• `dmp_FragmentLighting.enabled` is `GL_TRUE` and • `dmp_LightEnv.config` is configured to use RR and • `dmp_LightEnv.lutEnabledRefl` is `GL_TRUE`
Fragment Light: Reflection Green (RG)	• `dmp_FragmentLighting.enabled` is `GL_TRUE` and • `dmp_LightEnv.config` is configured to use RG and • `dmp_LightEnv.lutEnabledRefl` is `GL_TRUE`
Fragment Light: Reflection Blue (RB)	• `dmp_FragmentLighting.enabled` is `GL_TRUE` and • `dmp_LightEnv.config` is configured to use RB and • `dmp_LightEnv.lutEnabledRefl` is `GL_TRUE`
Fragment Light Distance Attenuation	• `dmp_FragmentLighting.enabled` is `GL_TRUE` and • `dmp_FragmentLightSource[i].enabled` is `GL_TRUE` and • `dmp_FragmentLightSource[i].distanceAttenuationEnabled` is `GL_TRUE`
Procedural Textures: RGB Mapping F Function	• `dmp_Texture[3].samplerType` is `GL_TEXTURE_PROCEDURAL_DMP`
Procedural Textures: Alpha Mapping F Function	• `dmp_Texture[3].samplerType` is `GL_TEXTURE_PROCEDURAL_DMP` and • `dmp_Texture[3].ptAlphaSeparate` is `GL_TRUE`
Procedural Textures: Noise Modulation Function	• `dmp_Texture[3].samplerType` is `GL_TEXTURE_PROCEDURAL_DMP` and • `dmp_Texture[3].ptNoiseEnable` is `GL_TRUE`
Procedural Texture Color	• `dmp_Texture[3].samplerType` is `GL_TEXTURE_PROCEDURAL_DMP`
Fog	• `dmp_Fog.mode` is not `GL_FALSE`
Gas Shading	• `dmp_Fog.mode` is `GL_GAS`

5.6 DMPGL Functions That Generate Commands

There are functions other than **`glDrawElements`**, **`glDrawArrays`**, and **`nngxValidateState`** that, when called, immediately generate commands for those functions' settings. Table 5-4 shows the functions that generate commands immediately.

Table 5-4 Function List

Function	Condition for Generating Commands
`glBlendColor`	A setting value has changed.

Function	Condition for Generating Commands
glBlendEquation	A setting value has changed.
glBlendEquationSeparate	A setting value has changed.
glBlendFunc	A setting value has changed.
glBlendFuncSeparate	A setting value has changed.
glClearEarlyDepthDMP	A setting value has changed.
glColorMask	A setting value has changed.
glCullFace	A setting value has changed.
glDepthFunc	A setting value has changed.
glDepthMask	A setting value has changed.
glDisable	One of the following settings values have changed. • GL_COLOR_LOGIC_OP • GL_BLEND • GL_DEPTH_TEST • GL_EARLY_DEPTH_TEST_DMP • GL_STENCIL_TEST • GL_CULL_FACE Commands are not generated for any other settings.
glEarlyDepthFuncDMP	A setting value has changed.
glEnable	One of the following settings values have changed. • GL_COLOR_LOGIC_OP • GL_BLEND • GL_DEPTH_TEST • GL_EARLY_DEPTH_TEST_DMP • GL_STENCIL_TEST • GL_CULL_FACE Commands are not generated for any other settings.
glFrontFace	A setting value has changed.
glLogicOp	A setting value has changed.
glRenderBlockModeDMP	A setting value has changed.
glStencilFunc	A setting value has changed.
glStencilMask	A setting value has changed.
glStencilOp	A setting value has changed.
glViewport	Always generated.

The functions in Table 5-4 generate commands according to the specified conditions.

All of these functions generate every relevant command together during validation (when the **glDrawElements** or **glDrawArrays** function is called, or when the **nngxValidateState** function is called with ***statemask*** set to `NN_GX_STATE_OTHERS`) if ***statemask*** is set to `NN_GX_STATE_OTHERS` in the **nngxUseSavedCmdlist** and **nngxUpdateState** functions.

Although these functions generate commands according to the given conditions, commands are always generated when the functions are called if the mode has been set to `NN_GX_CMDGEN_MODE_UNCONDITIONAL` by the **nngxSetCommandGenerationMode** function, regardless of the conditions in the table.

5.7 3D Command Buffer Specifications

This section describes 3D command buffer specifications. The 3D command buffer is a collection of commands that write to PICA registers. By replacing the 3D command buffers of saved and exported command lists according these specifications, you can change values that correspond to vertex shader and reserved fragment shader uniforms.

5.7.1 Basic Specifications

The 3D command buffer is a contiguous set of 64-bit commands, each of which comprises a 32-bit header and 32 bits of data. The amount of data varies with the content of the header.

The following table describes each of the 64 bits.

Table 5-5 Command Bit Structure

Bits	Name	Description
`[31:0]`	`DATA`	32 bits of data to write to a register.
`[47:32]`	`ADDR`	Address of the register to write to.
`[51:48]`	`BE`	Byte enable. Each bit corresponds to a byte of data, which is only written if that bit is set to 1. (Even if a command has a byte enable value of 0, the command itself is still sent to the module being set and can therefore be used as a dummy command to make internal timing adjustments, among other things. You must be careful, though, because the act of writing itself has meaning for some registers.)
`[58:52]`	`SIZE`	Data count - 1. If these bits have a value of 0, they indicate single access. If they have a value of 1 or greater, they indicate burst access.
`[63:63]`	`SEQ`	Indicates the burst access mode. If this bit is 0, data is written to a single register. If this bit is 1, data is written to consecutive registers.

Note: These bits use little-endian notation.

All commands are 64-bit aligned. The value of the `SIZE` bits indicates either single or burst access. In burst access mode, the `SEQ` bits determine whether data is written to a single register or to consecutive registers.

5.7.2 Single Access

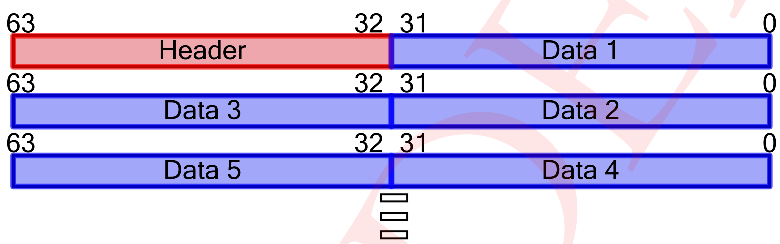
When the `SIZE` bits have a value of 0, commands are single-access. One unit of data (32 bits) is written once to a single register. The `ADDR` bits indicate the address of the register to write to. The `DATA` bits are the data to write. Each byte is only updated if its corresponding `BE` bit is set to 1. (A command does not write to a register where the `BE` bits are 0.) The `SEQ` bit is ignored. The 64 bits of data that follow a command represent the next command.

Consider the sample command, `0x000f011012345678`. `SIZE` is `0`, `BE` is `0xf`, `ADDR` is `0x110`, and `DATA` is `0x12345678`. This is single-access because the `SIZE` bits have a value of 0. This command writes `0x12345678` to register `0x110`.

5.7.3 Burst Access

When the `SIZE` bits have a value of 1 or greater, `SIZE+1` units of data are written to registers. The `DATA` bits are written first, followed by the data stored in the next 64 bits. The least-significant 32 bits are written before the most-significant 32 bits.

Figure 5-9 Command Structure for Burst Access



Because `SIZE+1` units of data are written, an even number of pieces are written when the `SIZE` bits have an odd value. In this case, the most-significant 32 bits of the last 64 bits of data are ignored. (The next command always starts at an address that is 64-bit aligned.) The `BE` bits' byte enable settings are applied to all writes uniformly.

If the `SEQ` bit is 0, data is written to a single register. If the `SEQ` bit is 1, data is written to consecutive registers.

5.7.3.1 Writing to a Single Register

Multiple units of data are written consecutively to a single register. The `ADDR` bits indicate the address of the register to write to. The `DATA` bits are written first, followed by the data stored in the next 64 bits. The least-significant 32 bits are written before the most-significant 32 bits. The number of data units written is one greater than the value of the `SIZE` bits.

Consider the following sample command.

- `0x004f008011111111`
- `0x3333333322222222`
- `0x5555555544444444`

In this command, `SIZE` is 4, `BE` is `0xf`, `ADDR` is `0x80`, `DATA` is `0x11111111`, `and SEQ` is 0. Because `SIZE` is 4 and `SEQ` is 0, five units of data are written consecutively to the same register. In other words, `0x11111111`, `0x22222222`, `0x33333333`, `0x44444444`, and `0x55555555` are written to the register at address `0x80`.

5.7.3.2 Writing to Consecutive Registers

Different values are written one at a time to multiple consecutive registers. The `ADDR` bits give the address of the first register to write. The `DATA` bits are written to the first register and then the following 64-bit data is written, starting with the least-significant 32 bits, to addresses that increment by one with each write. `SIZE+1` units of data are written to `SIZE+1` registers.

Consider the following sample command.

- `0x805f028011111111`
- `0x3333333322222222`
- `0x5555555544444444`
- `0x7777777766666666`

In this command, `SIZE` is 5, `BE` is `0xf`, `ADDR` is `0x28`, `DATA` is `0x11111111`, and `SEQ` is 1. Because `SIZE` is 5 and `SEQ` is 1, six units of data are written to consecutive registers. In other words, `0x11111111` is written to register `0x280`; `0x22222222` is written to register `0x281`; `0x33333333` is written to register `0x282`; `0x44444444` is written to register `0x283`; `0x55555555` is written to register `0x284`; and `0x66666666` is written to register `0x285`. The most-significant 32 bits (`0x77777777` in this example) of the last 64 bits of data are not used because the `SIZE` bit has an odd value. The next command is the 64 bits of data following `0x7777777766666666`.

5.8 PICA Register Information

This section describes PICA register information corresponding to specific features. The register information includes the register address, configuration method, value format, and so on. Using the information in this section, you can change the values set for a feature by searching for and replacing 3D command buffer locations that write to the corresponding registers.

5.8.1 Render Start Registers

If a value of 1 is written to register `0x22f` or `0x22e`, the **`glDrawElements`** or **`glDrawArrays`** function starts rendering using the vertex buffer, respectively. If a value of `0xf` is written to register `0x232`, the **`glDrawElements`** or **`glDrawArrays`** function starts rendering without using the vertex buffer.

5.8.2 Vertex Shader Floating-Point Registers

There are 96 floating-point registers for the vertex shader. Each one comprises four components: x, y, z, and w. These are written as `c0` through `c95` in shader assembly. You can use two methods to define either uniforms or constants with the `def` instruction. Internally, PICA uses 24-bit floating-point numbers with a 16-bit mantissa, 7-bit exponent, and 1-bit sign in order from the least-significant to the most-significant bit.

You can set either 24-bit or 32-bit PICA floating-point numbers. The DMPGL driver uses 24 bits to define a constant with `def` and 32 bits to define a uniform. The only difference between setting a 24-bit floating-point number and a 32-bit floating-point number is an increase in the number of commands to process. There is no processing overhead associated with a conversion between floating-point formats.

The 32-bit floating-point numbers mentioned here use the IEEE 754 single-precision format. When you set 32-bit values, PICA automatically converts them to 24 bits internally.

5.8.2.1 Address Information

Bits [7:0] of register `0x2c0` set the floating-point register index. (This is 0 for `c0` and `0x0a` for `c10`.) When 1 or 0 is simultaneously written to bit [31:31], floating-point numbers are input in 32-bit or 24-bit mode, respectively.

Data is written to the xyzw components of a floating-point register between `0x2c1` and `0x2c8`. Writing a value anywhere between `0x2c1` and `0x2c8` has the same effect. After you write an index to `0x2c0`, data is written from `0x2c1` to `0x2c8`.

5.8.2.2 How to Set the Input Mode for 32-Bit Floating-Point Numbers

When floating-point numbers are entered in 32-bit mode, 32 bits of data are written four times in wzyx order to any register between `0x2c1` and `0x2c8`. Once data is written four times, the index of the next floating-point register to write is automatically incremented by one.

Code 5-1 Sample 32-Bit Floating-Point Input

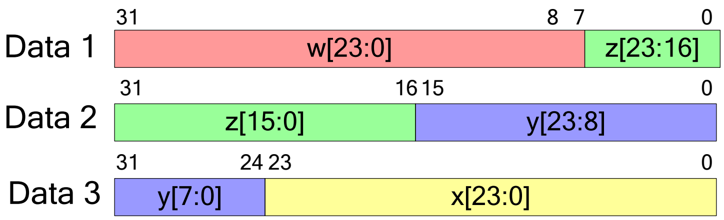
```
0x2c0 <= 0x80000023   // [31] = 1 for 32-bit input mode and [7:0] = 35
0x2c1 <= 0x40800000   // The value of c35.w
0x2c2 <= 0x40400000   // The value of c35.z
0x2c3 <= 0x40000000   // The value of c35.y
0x2c4 <= 0x3f800000   // The value of c35.x
0x2c1 <= 0x40800000   // The value of c36.w
0x2c2 <= 0x40400000   // The value of c36.z
0x2c3 <= 0x40000000   // The value of c36.y
0x2c4 <= 0x3f800000   // The value of c36.x
```

If you set register values as shown here, `c35.xyzw` and `36.xyzw` will be {1.f, 2.f, 3.f, 4.f}.

5.8.2.3 How to Set the Input Mode for 24-Bit Floating-Point Numbers

When floating-point numbers are entered in 24-bit mode, the w, z, y, and x components are converted into a 24-bit format and then packed into 32 bits of data, which is then written three times to any register between `0x2c1` and `0x2c8`. For details on how values are converted into 24-bit floating-point numbers, see section 5.9.1 Converting from float32 to float24. The following figure shows the 24-bit data layout.

Figure 5-10 How to Set 24-Bit Floating-Point Numbers



Data 1, **Data 2**, and **Data 3** in the figure are written in that order. Once data is written three times, the index of the next floating-point register to write is automatically incremented by one.

Code 5-2 Sample 24-Bit Floating-Point Input

```
0x2c0 <= 0x00000023   // [31] = 0 for 24-bit input mode and [7:0] = 35
0x2c1 <= 0x41000040   // [31:8]  = w[23:0] and [7:0]  = z[23:16]
0x2c2 <= 0x80004000   // [31:16] = z[15:0] and [15:0] = y[23:8]
0x2c3 <= 0x003f0000   // [31:24] = y[7:0]  and [23:0] = x[23:0]
```

When registers are set this way, the following values are set.

- `c35.x = 0x3f0000`
- `c35.y = 0x400000`
- `c35.z = 0x408000`
- `c35.w = 0x410000`

The value of `c35.xyzw` is therefore {1.f, 2.f, 3.f, 4.f}.

5.8.3 Vertex Shader Boolean Registers

There are 16 Boolean registers for the vertex shader. These are written as `b0`-`b15` in shader assembly. You can use these to define either uniforms or constants with the `defb` instruction.

Bits [15:0] of register `0x2b0` correspond to the vertex shader Boolean registers. Bits 0-15 correspond to `b0`-`b15`, respectively. A value of 1 is equivalent to `true`, 0 is equivalent to `false`.

5.8.4 Vertex Shader Integer Registers

There are 4 integer registers for the vertex shader. Each integer register comprises three components: x, y, and z. These are written as `i0`-`i3` in shader assembly. You can use these to define either uniforms or constants with the `defi` instruction.

The `0x2b1`, `0x2b2`, `0x2b3`, and `0x2b4` registers correspond to `i0`, `i1`, `i2`, and `i3`, respectively. For each register, bits [7:0] correspond to the x component, bits [15:8] correspond to the y component, and bits [23:16] correspond to the z component.

5.8.5 Vertex Shader Starting Address Setting Registers

Bits [15:0] of register `0x2ba` set the starting address of the vertex shader. This specifies the address of the `main` label defined in shader assembly.

5.8.6 Registers That Set the Number of Input Vertex Attributes

Bits [3:0] of registers `0x2b9` and `0x242` each set a value that is one less than the number of vertex attributes input to the vertex shader.

5.8.7 Registers That Set the Number of Output Registers Used by the Vertex Shader

These set the number of output registers written by the vertex shader. The specified value is the number of output registers defined by `#pragma output_map` in shader assembly. When `#pragma output_map` defines multiple attributes to be packed into a single output register, count those attributes as a single output register.

Bits [2:0] of register `0x4f` set the number of output registers to use. Bits [3:0] of registers `0x24a`, `0x25e`, and `0x251` each set a value that is one less than the number of output registers to use.

5.8.8 Registers That Set the Vertex Shader Output Mask

These use a bitmask to specify the output registers written by the vertex shader. Bits [15:0] of register `0x2bd` correspond to each of the 16 output registers (bit [0:0] corresponds to `o0`, bit [1:1] corresponds to `o1`, and bit [15:15] corresponds to `o15`).

A bit is set (1) if it corresponds to an output register defined by `#pragma output_map` in shader assembly. A bit is cleared (0) if it corresponds to an undefined output register.

5.8.9 Registers That Set Vertex Shader Output Attributes

These configure the vertex attributes output by the vertex shader. Data written to the output registers defined by `#pragma output_map` is output starting with the smallest index (so that `o0`, `o1`, `o2`, and `o3` are output in order and nothing is output for an output register that is not defined by `output_map`). Data attributes output by the vertex shader are specified one by one in registers, starting with data for the first register. The following table indicates register information.

Table 5-6 Registers That Set Output Attributes from the Vertex Shader

Setting Register	Description
`0x50`: bits [4:0]	Attribute for the x-component of the first set of output data. • `0x00`: Vertex coordinate x • `0x01`: Vertex coordinate y • `0x02`: Vertex coordinate z • `0x03`: Vertex coordinate w • `0x04`: Quaternion x • `0x05`: Quaternion y • `0x06`: Quaternion z • `0x07`: Quaternion w • `0x08`: Vertex color R • `0x09`: Vertex color G • `0x0a`: Vertex color B

Setting Register	Description
	• `0x0b`: Vertex color A • `0x0c`: Texture coordinate 0, u • `0x0d`: Texture coordinate 0, v • `0x0e`: Texture coordinate 1, u • `0x0f`: Texture coordinate 1, v • `0x10`: Texture coordinate 0, w • `0x12`: View vector x • `0x13`: View vector y • `0x14`: View vector z • `0x16`: Texture coordinate 2, u • `0x17`: Texture coordinate 2, v • `0x1f`: Invalid
`0x50`: bits [12:8]	The same settings as bits [4:0] of register `0x50` for the y-component attribute of the first set of output data.
`0x50`: bits [20:16]	The same settings as bits [4:0] of register `0x50` for the z-component attribute of the first set of output data.
`0x50`: bits [28:24]	The same settings as bits [4:0] of register `0x50` for the w-component attribute of the first set of output data.
`0x51`: bits [4:0], [12:8], [20:16], [28:24]	The same settings as register `0x50` for the second set of output data attributes.
`0x52`: bits [4:0], [12:8], [20:16], [28:24]	The same settings as register `0x50` for the third set of output data attributes.
`0x53`: bits [4:0], [12:8], [20:16], [28:24]	The same settings as register `0x50` for the fourth set of output data attributes.
`0x54`: bits [4:0], [12:8], [20:16], [28:24]	The same settings as register `0x50` for the fifth set of output data attributes.
`0x55`: bits [4:0], [12:8], [20:16], [28:24]	The same settings as register `0x50` for the sixth set of output data attributes.
`0x56`: bits [4:0], [12:8], [20:16], [28:24]	The same settings as register `0x50` for the seventh set of output data attributes.
`0x64`: bit [0:0]	Set equal to 1 when texture coordinates are output by the vertex shader and 0 when they are not.

For example, consider the following vertex shader definitions.

Code 5-3 Sample Vertex Shader Definitions

```
#pragma output_map(position, o0)
#pragma output_map(color, o1)
#pragma output_map(texture0, o2.xy)
#pragma output_map(texture0w, o2.z)
#pragma output_map(texture1, o3.xy)
```

The registers are set as follows.

- `0x50 <- 0x03020100`
- `0x51 <- 0x0b0a0908`
- `0x52 <- 0x1f100d0c` (`w` is invalid)
- `0x53 <- 0x1f1f0f0e` (`zw` are invalid)
- `0x54 <- 0x1f1f1f1f` (the fifth attribute is invalid)
- `0x55 <- 0x1f1f1f1f` (the sixth attribute is invalid)
- `0x56 <- 0x1f1f1f1f` (the seventh attribute is invalid)

5.8.10 Clock Control Setting Registers for Vertex Shader Output Attributes

The attributes output by the vertex shader cause the clock control register settings to change. When the bit that corresponds to an attribute is set equal to 0, the clock supply for the related module is stopped; this is effective in decreasing power consumption. The following table shows the registers that correspond to each attribute.

Table 5-7 Clock Control Setting Registers for Vertex Shader Output Attributes

Setting Register	Description
`0x6f`: bit [0:0]	• 1 when vertex coordinate z is output • 0 when vertex coordinate z is not output
`0x6f`: bit [1:1]	• 1 when the vertex color is output • 0 when the vertex color is not output
`0x6f`: bit [8:8]	• 1 when texture coordinate 0 is output • 0 when texture coordinate 0 is not output
`0x6f`: bit [9:9]	• 1 when texture coordinate 1 is output • 0 when texture coordinate 1 is not output
`0x6f`: bit [10:10]	• 1 when texture coordinate 2 is output • 0 when texture coordinate 2 is not output
`0x6f`: bit [16:16]	• 1 when the w component of texture coordinate 0 is output • 0 when the w component of texture coordinate 0 is not output
`0x6f`: bit [24:24]	• 1 when the view vector and quaternions are output • 0 when the view vector and quaternions are not output

5.8.11 Vertex Shader Program Code Setting Registers

The following table shows registers that set the program code executed by the vertex shader.

Table 5-8 Vertex Shader Program Code Setting Registers

Setting Register	Description
`0x2cb`: bits [11:0]	Sets the load address for program code.
`0x2cc`–`0x2d3`: bits [31:0]	Sets program code data.

When data is set in bits [31:0] of registers `0x2cc`–`0x2d3`, program data is loaded into the load address set by bits [11:0] of register `0x2cb`. Each time data is written to bits [31:0] of registers `0x2cc`–`0x2d3`, the load address is automatically incremented by 1. (The address advances by a single instruction in program code, or 32 bits.) Behavior is the same regardless of where the register between `0x2cc` and `0x2d3` is written.

After the program code is updated, some command must write a value of 1 to any bit in register `0x2bf` to send a notification that the program update is complete.

In addition to the program code just described, swizzle pattern data must be loaded. The following table shows the registers that set swizzle patterns.

Table 5-9 Vertex Shader Swizzle Pattern Setting Registers

Setting Register	Description
`0x2d5`: bits [11:0]	Sets the load address for swizzle patterns.
`0x2d6`–`0x2dd`: bits [31:0]	Sets swizzle pattern data.

When data is set in bits [31:0] of registers `0x2d6`–`0x2dd`, swizzle patterns are loaded into the load address set by bits [11:0] of register `0x2d5`. Each time data is written to bits [31:0] of registers `0x2d6`–`0x2dd`, the load address is automatically incremented by 1. (The address advances by one set of data, or 32 bits.) Behavior is the same regardless of where the register between `0x2d6` and `0x2dd` is written.

5.8.12 Registers That Map Vertex Attributes to Input Registers

These configure which input registers of the vertex shader are used to store input vertex attribute data and are shown in the following table.

Table 5-10 Registers That Map Vertex Attributes to Input Registers

Setting Register	Description
`0x2bb`: bits [3:0]	Index of the input register in which to store the 1st input vertex attributes.
`0x2bb`: bits [7:4]	Index of the input register in which to store the 2nd input vertex attributes.
`0x2bb`: bits [11:8]	Index of the input register in which to store the 3rd input vertex attributes.
`0x2bb`: bits [15:12]	Index of the input register in which to store the 4th input vertex attributes.
`0x2bb`: bits [19:16]	Index of the input register in which to store the 5th input vertex attributes.

Setting Register	Description
`0x2bb`: bits [23:20]	Index of the input register in which to store the 6th input vertex attributes.
`0x2bb`: bits [27:24]	Index of the input register in which to store the 7th input vertex attributes.
`0x2bb`: bits [31:28]	Index of the input register in which to store the 8th input vertex attributes.
`0x2bc`: bits [3:0]	Index of the input register in which to store the 9th input vertex attributes.
`0x2bc`: bits [7:4]	Index of the input register in which to store the 10th input vertex attributes.
`0x2bc`: bits [11:8]	Index of the input register in which to store the 11th input vertex attributes.
`0x2bc`: bits [15:12]	Index of the input register in which to store the 12th input vertex attributes.

The input register indices are set so that index 0 corresponds to `v0`, index 1 corresponds to `v1`, and so on up to index 15, which corresponds to `v15`. Vertex attributes are not input to the vertex shader in an order that corresponds to ***index*** in the **`glBindAttribLocation`** function. The input order instead corresponds to the internal vertex attribute numbers described in section 5.8.14 Registers for Vertex Attribute Array Settings. Please refer to that section together with this one.

5.8.13 Registers That Set Fixed Vertex Attribute Values

The fixed vertex attribute values set by the **`glVertexAttrib4f`** function and other functions are converted into 24-bit floating-point numbers and sent to the hardware. To do so, a value is first written to bits [3:0] of register `0x232` indicating the order in which vertex attributes are input to the vertex shader. Next, the fixed vertex attribute value is converted into three 24-bit floating-point numbers that are stored as 32-bit values and written to registers `0x233`, `0x234`, and `0x235`.

The values that are converted into 24-bit floating-point numbers and stored as 32-bit data follow the same data creation method as the one described in section 5.8.2.3 How to Set the Input Mode for 24-Bit Floating-Point Numbers. Vertex attributes are not input to the vertex shader in an order that corresponds to ***index*** in the **`glBindAttribLocation`** function. The input order instead corresponds to the internal vertex attribute numbers described in section 5.8.14 Registers for Vertex Attribute Array Settings. Please refer to that section together with this one.

Although these fixed vertex attribute settings are applied individually to each numbered internal vertex attribute, if an internal vertex attribute is switched to be used as a vertex array by the settings described in section 5.8.14 Registers for Vertex Attribute Array Settings, the fixed vertex attribute value configured for it by this setting is invalidated. Therefore, if a vertex array is changed back to a fixed vertex attribute, you must set its fixed vertex attribute value again

Hardware specifications do not allow all the vertex attributes to be used as fixed vertex attributes, with no vertex arrays used at all. At least one vertex array must be used.

5.8.14 Registers for Vertex Attribute Array Settings

This section describes registers that set the address, type, and other information for vertex attribute arrays when vertex buffers are in use. The register-setting commands explained in this section are generated by `NN_GX_STATE_VERTEX` validation. The registers are shown in the following table.

Table 5-11 Registers for Vertex Attribute Array Settings

Name	Register	Description
Base address	`0x200`, bits [28:1]	The common base address for all vertex arrays. This is specified as a 128-bit address.
Type of internal vertex attribute 0	`0x201`, bits [3:0]	Specifies the type of internal vertex attribute 0. The following list shows combinations of ***size*** and ***type*** to the **`glVertexAttribPointer`** function when it is called on a GL attribute number corresponding to internal vertex attribute 0. • `0x0: size = 1, type = GL_BYTE` • `0x1: size = 1, type = GL_UNSIGNED_BYTE` • `0x2: size = 1, type = GL_SHORT` • `0x3: size = 1, type = GL_FLOAT` • `0x4: size = 2, type = GL_BYTE` • `0x5: size = 2, type = GL_UNSIGNED_BYTE` • `0x6: size = 2, type = GL_SHORT` • `0x7: size = 2, type = GL_FLOAT` • `0x8: size = 3, type = GL_BYTE` • `0x9: size = 3, type = GL_UNSIGNED_BYTE` • `0xa: size = 3, type = GL_SHORT` • `0xb: size = 3, type = GL_FLOAT` • `0xc: size = 4, type = GL_BYTE` • `0xd: size = 4, type = GL_UNSIGNED_BYTE` • `0xe: size = 4, type = GL_SHORT` • `0xf: size = 4, type = GL_FLOAT`
Type of internal vertex attribute 1	`0x201`, bits [7:4]	Sets internal vertex attribute 1 in the same way as internal vertex attribute 0.
Type of internal vertex attribute 2	`0x201`, bits [11:8]	Sets internal vertex attribute 2 in the same way as internal vertex attribute 0.
Type of internal vertex attribute 3	`0x201`, bits [15:12]	Sets internal vertex attribute 3 in the same way as internal vertex attribute 0.
Type of internal vertex attribute 4	`0x201`, bits [19:16]	Sets internal vertex attribute 4 in the same way as internal vertex attribute 0.
Type of internal vertex attribute 5	`0x201`, bits [23:20]	Sets internal vertex attribute 5 in the same way as internal vertex attribute 0.
Type of internal vertex attribute 6	`0x201`, bits [27:24]	Sets internal vertex attribute 6 in the same way as internal vertex attribute 0.

Name	Register	Description
Type of internal vertex attribute 7	`0x201`, bits [31:28]	Sets internal vertex attribute 7 in the same way as internal vertex attribute 0.
Type of internal vertex attribute 8	`0x202`, bits [3:0]	Sets internal vertex attribute 8 in the same way as internal vertex attribute 0.
Type of internal vertex attribute 9	`0x202`, bits [7:4]	Sets internal vertex attribute 9 in the same way as internal vertex attribute 0.
Type of internal vertex attribute 10	`0x202`, bits [11:8]	Sets internal vertex attribute 10 in the same way as internal vertex attribute 0.
Type of internal vertex attribute 11	`0x202`, bits [15:12]	Sets internal vertex attribute 11 in the same way as internal vertex attribute 0.
Fixed vertex attribute mask	`0x202`, bits [27:16]	Sets the internal vertex attribute mask for fixed vertex attributes.
Vertex attribute count	`0x202`, bits [31:28]	Sets a number that is one less than the total vertex attribute count (this is one less than the total number of fixed vertex attributes and vertex attribute arrays).
Load array N address offset	`0x203+N×3`, bits [27:0]	The address of load array N. (N=0, 1, …, 11) Sets an offset (in bytes) from the base address.
1st component of load array N	`0x204+N×3`, bits [3:0]	Sets the first component of load array N. • `0x0`: Internal vertex attribute 0 • `0x1`: Internal vertex attribute 1 • `0x2`: Internal vertex attribute 2 • `0x3`: internal vertex attribute 3 • `0x4`: Internal vertex attribute 4 • `0x5`: Internal vertex attribute 5 • `0x6`: Internal vertex attribute 6 • `0x7`: internal vertex attribute 7 • `0x8`: Internal vertex attribute 8 • `0x9`: Internal vertex attribute 9 • `0xa`: Internal vertex attribute 10 • `0xb`: internal vertex attribute 11 • `0xc`: 4-byte padding • `0xd`: 8-byte padding • `0xe`: 12-byte padding • `0xf`: 16-byte padding
2nd component of load array N	`0x204+N×3`, bits [7:4]	Sets the 2nd component of load array N in the same way as the 1st component.
3rd component of load array N	`0x204+N×3`, bits [11:8]	Sets the 3rd component of load array N in the same way as the 1st component.
4th component of load array N	`0x204+N×3`, bits [15:12]	Sets the 4th component of load array N in the same way as the 1st component.

Name	Register	Description
5th component of load array N	`0x204+N×3`, bits [19:16]	Sets the 5th component of load array N in the same way as the 1st component.
6th component of load array N	`0x204+N×3`, bits [23:20]	Sets the 6th component of load array N in the same way as the 1st component.
7th component of load array N	`0x204+N×3`, bits [27:24]	Sets the 7th component of load array N in the same way as the 1st component.
8th component of load array N	`0x204+N×3`, bits [31:28]	Sets the 8th component of load array N in the same way as the 1st component.
9th component of load array N	`0x205+N×3`, bits [3:0]	Sets the 9th component of load array N in the same way as the 1st component.
10th component of load array N	`0x205+N×3`, bits [7:4]	Sets the 10th component of load array N in the same way as the 1st component.
11th component of load array N	`0x205+N×3`, bits [11:8]	Sets the 11th component of load array N in the same way as the 1st component.
12th component of load array N	`0x205+N×3`, bits [15:12]	Sets the 12th component of load array N in the same way as the 1st component.
Byte count for load array N	`0x205+N×3`, bits [23:16]	Number of bytes for a single vertex in load array N.
Load array N component count	`0x205+N×3`, bits [31:28]	The number of components in load array N.
Index array address offset	`0x227`, bits [27:0]	The address of the index array. This is an offset (in bytes) from the base address.

There are settings for the base address, vertex attribute types, a fixed vertex attribute mask, the total number of vertex attributes, the byte offset to each load array, information on load array components, the number of load array components, the load array byte count, and the index array offset.

5.8.14.1 Base Address

The addresses of all vertex arrays and the vertex index array are set as offsets from a 128-bit base address (the byte address divided by 16), which is itself specified in bits [28:1] of register `0x200`.

The base address is 16-byte aligned and is smaller than the addresses of all vertex arrays and of the index array. When the vertex arrays and index array use a range of addresses that has been fixed in advance, commands to this register do not need to be re-set for each vertex array combination.

5.8.14.2 Internal Vertex Attributes

Internal vertex attributes are vertex attribute numbers that are determined internally and used by PICA to load vertex arrays. Although they differ from *GL vertex attribute numbers*, which are the vertex attribute numbers specified as ***index*** to the **`glEnableVertexAttribArray`** function, internal vertex attribute numbers and GL vertex attribute numbers have a one-to-one correspondence.

Vertex arrays enabled by the **`glEnableVertexAttribArray`** function are assigned continuously in ascending order starting at internal vertex attribute 0. For example, when vertex arrays are enabled for the GL vertex attribute numbers 0 and 3, they are assigned to the internal vertex attributes 0 and 1. However, GL vertex attribute number 0 does not necessarily correspond to internal vertex attribute 0. The assignment of internal vertex attributes is driver implementation-dependent. The current implementation sorts vertex array addresses in ascending order and then assigns GL vertex attributes one by one starting with the first attribute, which is assigned to internal vertex attribute 0. (Because this is dependent on the driver implementation, it may change in the future.)

The vertex shader's input vertex attribute data is ordered according to the internal vertex attributes. See section 5.8.12 Registers That Map Vertex Attributes to Input Registers for more information.

Bits [31:0] of register `0x201` and bits [15:0] of register `0x202` specify the internal vertex attribute types; for each internal vertex attribute, a value is set for the combination of ***size*** and ***type*** given to the **`glVertexAttribPointer`** function for the corresponding GL vertex attribute.

5.8.14.3 Fixed Vertex Attribute Mask

As many vertex attributes are enabled as are defined by `#pragma bind_symbol` in the vertex shader assembly code, but if any of those enabled vertex attributes have a disabled vertex array (either because the **`glDisableVertexAttribArray`** function has been called on this vertex attribute or the **`glEnableVertexAttribArray`** function has not been called on it), a fixed vertex attribute is used in its place.

Fixed vertex attributes are assigned to internal vertex attributes in the same way as vertex arrays are assigned. Continuous internal numbers are assigned in ascending order following the numbers assigned to vertex arrays.

Bits [27:16] of register `0x202` set a mask for assigned internal vertex attributes. Bit [16+`i`:16+`i`] corresponds to internal vertex attribute `i` and is set to `1` if it is assigned to a fixed vertex attribute.

Hardware specifications do not allow all the vertex attributes to be used as fixed vertex attributes, with no vertex arrays used at all. At least one vertex array must be used.

If an internal vertex attribute has had its vertex array toggled between enabled and disabled or vice versa, configuring this register setting will disable the fixed vertex attribute value previously set for that vertex attribute, and the value must be reset. See section 5.8.13 Registers That Set Fixed Vertex Attribute Values for details.

5.8.14.4 Vertex Attribute Count

Bits [31:28] of register `0x202` set a value that is one less than the total number of fixed vertex attributes and vertex attributes that use vertex arrays.

5.8.14.5 Load Arrays

A *load array* is an internally managed data array unit that PICA uses to load vertex attributes. PICA loads data from 12 load arrays. (There is a register for each load array. The register address notation "`0x203+N×3`" indicates that there are 12 registers corresponding to values of N between 0 and 11.)

The 12 load arrays each comprise up to 12 components. A load array component is either vertex array data that makes up that load array or padding in 4-byte units. Basically, when vertex data is defined as an array of structures with multiple vertex attributes (called an *interleaved array*), a single interleaved array corresponds to a single load array. On the other hand, when vertex data is defined as a single vertex attribute array (called an *independent array*), that single vertex attribute corresponds to a single load array.

Because hardware performance improves as the number of used load arrays decreases, the DMPGL driver is configured to be able to load data with a small number of load arrays.

Bits [31:0] of registers `0x204+N×3` and bits [15:0] of register `0x205` specify the components that make up each load array in order from the first component. When bits [3:0] of register `0x204` specify 0, for example, the first component of load array 0 becomes internal vertex attribute 0 and the data placed at the start of load array 0 is placed according to internal vertex attribute 0's type, which is set by bits [3:0] of register `0x201`.

Bits [23:16] of registers `0x205+N×3` set the number of bytes in a single vertex for each load array. A load array with elements of more than one type may be automatically padded. The number of bytes per vertex must be set to the correct value that includes padding. Behavior is undefined if this setting does not match the total size of the load array elements.

Bits [31:28] of registers `0x205+N×3` set the number of components in each load array. A load array is not used when 0 is specified.

To find the actual address of a vertex attribute array, add the offset specified by ***ptr*** in the **`glVertexAttribPointer`** function to the address allocated by the **`glBufferData`** function. Bits [27:0] of registers `0x203+N×3` are set so that this actual address is equal to $(base\ address \times 16 + load\ array\ address\ offset)$.

Similarly, the vertex index array's address offset is set in bits [27:0] of register `0x227` as an offset from the base address. See section 5.8.38 Settings Registers Specific to the Rendering API for more information.

Consider the following example of an interleaved array.

Code 5-4 Sample Interleaved Array

```
struct vertex_t {
   float position[3];
   float color[4];
   float texcoord[2];
} vertex[NUM_VERTEX];
```

Vertex data created with this structure uses the following vertex array settings.

Code 5-5 Vertex Array Settings for an Interleaved Array

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(struct vertex t), 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(struct vertex t), 12);
```

```
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(struct vertex t), 28);
```

The three GL vertex attributes 0, 1, and 2 are components of a single load array. If a total of four vertex attributes are used—the three in Code 5-5 and one fixed vertex attribute—vertex attributes 0, 1, and 2 correspond to internal vertex attributes 0, 1, and 2 and the fixed vertex attribute corresponds to internal vertex attribute 3. Consequently, the relevant registers are set as follows.

- `0x201 <- 0x000007fb`

  Internal vertex attributes 0, 1, and 2 are of type `FLOAT_VEC3`, `FLOAT_VEC4`, and `FLOAT_VEC2`, respectively.

- `0x202 <- 0x30080000`

  There are a total of four vertex attributes; internal vertex attribute 3 is a fixed vertex attribute.

- `0x203 <- 0x00000000`

  Because we are only using one load array, the base address is set equal to the actual address.

- `0x204 <- 0x00000210`

  The components of load array 0 are internal vertex attributes 0, 1, and 2.

- `0x205 <- 0x30240000`

  Load array 0 uses 36 bytes (`float×9`) per vertex and has three components.

- `0x206-0x226 <- 0x00000000`

  Other load arrays are not used.

Now consider the following example of an independent array.

Code 5-6 Sample Independent Array

```
#define NUM_VERTEX (3)
struct attribute0_t {
   float position[3];
} attribute0[NUM_VERTEX];
struct attribute1_t {
   float color[4];
} attribute1[NUM_VERTEX];

struct attribute2_t {
   float tex[2];
} attribute2[NUM_VERTEX];
```

Vertex data created with this structure uses the following vertex array settings (a single vertex buffer object is shared and data is placed in order).

Code 5-7 Vertex Array Settings for an Independent Array

```
glBindBuffer(GL_ARRAY_BUFFER, 1);
glBufferData(GL_ARRAY_BUFFER,
   sizeof(attribute0)+sizeof(attribute1)+sizeof(attribute2), 0, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(attribute0), attribute0);
glBufferSubData(GL_ARRAY_BUFFER,
   sizeof(attribute0), sizeof(attribute1), attribute1);
glBufferSubData(GL_ARRAY_BUFFER,
   sizeof(attribute0)+sizeof(attribute1), sizeof(attribute2), attribute2);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
   (GLvoid*)(sizeof(attribute0)));
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
   (GLvoid*)(sizeof(attribute0)+sizeof(attribute1)));
```

GL vertex attributes 0, 1, and 2 are each separate load array components that correspond to internal vertex attributes 0, 1, and 2. The relevant registers settings are as follows.

- `0x201 <- 0x000007fb`

  Internal vertex attributes 0, 1, and 2 are of type `FLOAT_VEC3`, `FLOAT_VEC4`, and `FLOAT_VEC2`, respectively.

- `0x202 <- 0x20000000`

  There are a total of three vertex attributes and no fixed vertex attributes.

- `0x203 <- 0x00000000`

  Load array 0 is placed at the beginning.

- `0x204 <- 0x00000000`

  Load array 0 has a single component: internal vertex attribute 0.

- `0x205 <- 0x100c0000`

  Load array 0 uses 12 bytes (`float×3`) per vertex and has one component.

- `0x206 <- 0x00000024`

  The offset of load array 1 is `sizeof(attribute0)`.

- `0x207 <- 0x00000001`

  Load array 1 has a single component: internal vertex attribute 1.

- `0x208 <- 0x10100000`

  Load array 1 uses 16 bytes (`float×4`) per vertex and has one component.

- `0x209 <- 0x00000054`

  Load array 2 has an offset of `sizeof(attribute0)+sizeof(attribute1)`.

- `0x20a <- 0x00000002`

  Load array 2 has a single component: internal vertex attribute 2.

- `0x20b <- 0x10080000`

  Load array 2 uses 8 bytes (`float×2`) per vertex and has one component.

- `0x20c-0x226 <- 0x00000000`

  Other load arrays are not used.

5.8.14.6 Padding Components and Automatic Padding for the Load Array

Bits [31:0] of registers `0x204+N×3` and bits [15:0] of register `0x205` have four load array component values for padding: `0xc`, `0xd`, `0xe`, and `0xf`. These are used in load arrays with unused regions.

Consider vertex data created with the following structure.

Code 5-8 Sample Vertex Data Structure with Padding Components

```
struct vertex_t
{
    float position[3];
    float color[4];
    float texcoord[2];
} vertex[NUM_VERTEX];
```

Assume that `texcoord` is not used as a vertex attribute. Because the size of a single vertex is `float×9`, the last `float×2` bytes are unused. Internal vertex attributes are specified as the first and second components of the load array corresponding to this vertex data, but `0xd` (8-byte padding) is specified as the third component.

If the components of a single load array are vertex attributes with multiple different data types (`GL_FLOAT`, `GL_SHORT`, `GL_BYTE`, and `GL_UNSIGNED_BYTE`), less than four bytes of padding may automatically be inserted even if it is not specified in the load array components. Each component that makes up a load array is either a 4-byte type (corresponding to an internal vertex attribute type of `GL_FLOAT` or padding), a 2-byte type (corresponding to an internal vertex attribute type of `GL_SHORT`), or a 1-byte type (corresponding to an internal vertex attribute type of `GL_BYTE` or `GL_UNSIGNED_BYTE`). Each component in a load array is automatically padded to the alignment of the component type in that load array.

For example, consider vertex data with the following structure.

Code 5-9 Sample Vertex Data Structure with Automatic Padding

```
struct vertex_t
{
```

```
    GLfloat position[3];
    GLubyte color[3];
    GLfloat texcoord[2];
} vertex[NUM_VERTEX];
```

Assume that the load array's components are the three vertex attributes `position`, `color`, and `texcoord`. Although `color` uses 3 bytes, `texcoord` is 4-byte aligned because it is a `GLfloat`. In other words, a single byte of padding is automatically inserted immediately after `color`.

If a single load array's elements comprise vertex attributes of multiple data types (`GL_FLOAT`, `GL_SHORT`, `GL_BYTE`, and `GL_UNSIGNED_BYTE`), padding is automatically added at the end of each vertex's data to align it with the size of the load array element that has the largest data type.

For example, consider vertex data with the following structure.

Code 5-10 Another Sample Vertex Data Structure with Automatic Padding

```
struct vertex_t
{
  GLubyte color[3];
  GLfloat position[3];
  GLubyte param;
} vertex[NUM_VERTEX];
```

The load array can be thought to have three vertex attributes—`color`, `position`, and `param`—as elements. The largest of these three attributes is a `GLfloat`, which uses four bytes. Consequently, `vertex[0]`, `vertex[1]`, and so on through `vertex[NUM_VERTEX-1]` are all 4-byte aligned. In other words, three bytes of padding are automatically inserted immediately after `param`.

When padding is automatically inserted, the per-vertex size that includes this padding must be set in bits [23:16] of registers `0x205+N×3`.

5.8.15 Other Setting Registers Related to the Vertex Shader

See section 5.8.39 Settings Registers Specific to the Geometry Shader when you use the geometry shader. Even if you only use the vertex shader, section 5.8.39.13 Miscellaneous Registers mentions register settings that are required when the geometry shader is not in use.

5.8.16 Texture Address Setting Registers

This section describes registers that set texture data addresses. You must update the registers described in this section when a texture object is changed or placed at a different address.

Table 5-12 Texture Data Address Setting Registers

Texture Unit	Target	Registers
Texture 0	GL_TEXTURE_2D	0x85, bits [27:0]
Texture 0	GL_TEXTURE_CUBE_MAP_POSITIVE_X	0x85, bits [27:0]
Texture 0	GL_TEXTURE_CUBE_MAP_NEGATIVE_X	0x86, bits [21:0] 0x85, bits [27:22]
Texture 0	GL_TEXTURE_CUBE_MAP_POSITIVE_Y	0x87, bits [21:0] 0x85, bits [27:22]
Texture 0	GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	0x88, bits [21:0] 0x85, bits [27:22]
Texture 0	GL_TEXTURE_CUBE_MAP_POSITIVE_Z	0x89, bits [21:0] 0x85, bits [27:22]
Texture 0	GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	0x8a, bits [21:0] 0x85, bits [27:22]
Texture 1	GL_TEXTURE_2D	0x95, bits [27:0]
Texture 2	GL_TEXTURE_2D	0x9d, bits [27:0]

All texture memory addresses are set as 8-byte physical addresses. (This value is the physical address divided by 8.) The six cube map faces have 28-bit texture addresses. The most significant 6 bits of every address share bits [27:22] in register 0x85.

Using the information in this section, you can change texture data addresses to adjust texture data placement. The texture resolution, filter mode, number of mipmap levels, and other information are not expected to change.

5.8.17 Render Buffer Setting Registers

This section shows register settings related to the render buffer. The register setting commands described in this section are generated by NN_GX_STATE_FRAMEBUFFER validation.

Table 5-13 Block Format Setting Registers

Setting	Setting Register	Setting Value
Color buffer address	0x11d, bits [27:0]	Sets a value equal to the color buffer's byte address divided by 8.
Depth buffer address	0x11c, bits [27:0]	Sets a value equal to the depth buffer's byte address divided by 8.
Color buffer pixel size	0x117, bits [1:0]	• 0 when the color buffer format has a 16-bit pixel size • 2 when the color buffer format has a 32-bit pixel size

Setting	Setting Register	Setting Value
Color buffer format	`0x117`, bits [18:16]	• 0: `GL_RGBA8_OES` or `GL_GAS_DMP` • 2: `GL_RGB5_A1` • 3: `GL_RGB565` • 4: `GL_RGBA4`
Depth buffer format	`0x116`, bits [1:0]	• 0: `GL_DEPTH_COMPONENT16` • 2: `GL_DEPTH_COMPONENT24_OES` • 3: `GL_DEPTH24_STENCIL8_EXT`
Color and depth buffer width	`0x11e`, bits [10:0] `0x6e`, bits [10:0]	Sets the color and depth buffer width in pixels.
Color and depth buffer height	`0x11e`, bits [21:12] `0x6e`, bits [21:12]	Sets the color and depth buffer height in pixels. This value is one less than the actual height.

5.8.18 Texture Combiner Setting Registers

This section describes registers related to reserved fragment shader uniforms with `dmp_TexEnv[i]` in their names. The following table shows the texture combiner registers.

Table 5-14 Texture Combiner Setting Registers

Uniform	Register	Setting Value
`srcRgb`: 1st component	Starting register + 0 bits [3:0]	• `0x0 : GL_PRIMARY_COLOR` • `0x1 : GL_FRAGMENT_PRIMARY_COLOR_DMP` • `0x2 : GL_FRAGMENT_SECONDARY_COLOR_DMP` • `0x3 : GL_TEXTURE0` • `0x4 : GL_TEXTURE1` • `0x5 : GL_TEXTURE2` • `0x6 : GL_TEXTURE3` • `0xd : GL_PREVIOUS_BUFFER_DMP` • `0xe : GL_CONSTANT` • `0xf : GL_PREVIOUS`
`srcRgb`: 2nd component	Starting register + 0 bits [7:4]	Same as the 1st component of `srcRgb`.
`srcRgb`: 3rd component	Starting register + 0 bits [11:8]	Same as the 1st component of `srcRgb`.
`srcAlpha`: 1st component	Starting register + 0 bits [19:16]	Same as the 1st component of `srcRgb`.
`srcAlpha`: 2nd component	Starting register + 0 bits [23:20]	Same as the 1st component of `srcRgb`.
`srcAlpha`: 3rd component	Starting register + 0 bits [27:24]	Same as the 1st component of `srcRgb`.

Uniform	Register	Setting Value
operandRgb: 1st component	Starting register + 1 bits [3:0]	• 0x0 : GL_SRC_COLOR • 0x1 : GL_ONE_MINUS_SRC_COLOR • 0x2 : GL_SRC_ALPHA • 0x3 : GL_ONE_MINUS_SRC_ALPHA • 0x4 : GL_SRC_R_DMP • 0x5 : GL_ONE_MINUS_SRC_R_DMP • 0x8 : GL_SRC_G_DMP • 0x9 : GL_ONE_MINUS_SRC_G_DMP • 0xc : GL_SRC_B_DMP • 0xd : GL_ONE_MINUS_SRC_B_DMP
operandRgb: 2nd component	Starting register + 1 bits [7:4]	Same as the 1st component of operandRgb.
operandRgb: 3rd component	Starting register + 1 bits [11:8]	Same as the 1st component of operandRgb.
operandAlpha: 1st component	Starting register + 1 bits [14:12]	• 0x0 : GL_SRC_ALPHA • 0x1 : GL_ONE_MINUS_SRC_ALPHA • 0x2 : GL_SRC_R_DMP • 0x3 : GL_ONE_MINUS_SRC_R_DMP • 0x4 : GL_SRC_G_DMP • 0x5 : GL_ONE_MINUS_SRC_G_DMP • 0x6 : GL_SRC_B_DMP • 0x7 : GL_ONE_MINUS_SRC_B_DMP
operandAlpha: 2nd component	Starting register + 1 bits [18:16]	Same as the 1st component of operandAlpha.
operandAlpha: 3rd component	Starting register + 1 bits [22:20]	Same as the 1st component of operandAlpha.
combineRgb	Starting register + 2 bits [3:0]	• 0x0 : GL_REPLACE • 0x1 : GL_MODULATE • 0x2 : GL_ADD • 0x3 : GL_ADD_SIGNED • 0x4 : GL_INTERPOLATE • 0x5 : GL_SUBTRACT • 0x6 : GL_DOT3_RGB • 0x7 : GL_DOT3_RGBA • 0x8 : GL_MULT_ADD_DMP • 0x9 : GL_ADD_MULT_DMP

Uniform	Register	Setting Value
`combineAlpha`	Starting register + 2 bits [19:16]	• `0x0 : GL_REPLACE` • `0x1 : GL_MODULATE` • `0x2 : GL_ADD` • `0x3 : GL_ADD_SIGNED` • `0x4 : GL_INTERPOLATE` • `0x5 : GL_SUBTRACT` • `0x6 : GL_DOT3_RGB` • `0x7 : GL_DOT3_RGBA` • `0x8 : GL_MULT_ADD_DMP` • `0x9 : GL_ADD_MULT_DMP`
`constRgba`: 1st component	Starting register + 3 bits [7:0]	Floating-point number between 0 and 1 that was mapped to an integer value between 0 and 255. For details on how this value is converted, see section 5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer.
`constRgba`: 2nd component	Starting register + 3 bits [15:8]	Same as the 1st component of `constRgba`.
`constRgba`: 3rd component	Starting register + 3 bits [23:16]	Same as the 1st component of `constRgba`.
`constRgba`: 4th component	Starting register + 3 bits [31:24]	Same as the 1st component of `constRgba`.
`scaleRgb`	Starting register + 4 bits [1:0]	• `0x0 : 1.0` • `0x1 : 2.0` • `0x2 : 4.0`
`scaleAlpha`	Starting register + 4 bits [17:16]	Same as `scaleRgb`.
`bufferColor`: 1st component	`0x0fd` bits [7:0]	A floating-point number between 0 and 1 that was mapped to an integer between 0 and 255. For details on how this value is converted, see section 5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer.
`bufferColor`: 2nd component	`0x0fd` bits [15:8]	Same as the 1st component of `bufferColor`.
`bufferColor`: 3rd component	`0x0fd` bits [23:16]	Same as the 1st component of `bufferColor`.
`bufferColor`: 4th component	`0x0fd` bits [31:24]	Same as the 1st component of `bufferColor`.

Uniform	Register	Setting Value
bufferInput: 1st component	0x0e0 bit [7+i:7+i] (i corresponds to the i in dmp_TexEnv[i] and can have a value of 1, 2, 3, or 4)	• 0: GL_PREVIOUS_BUFFER_DMP • 1: GL_PREVIOUS
bufferInput: 2nd component	0x0e0 bit [11+i:11+i] (i corresponds to the i in dmp_TexEnv[i] and can have a value of 1, 2, 3, or 4)	• 0: GL_PREVIOUS_BUFFER_DMP • 1: GL_PREVIOUS

The names in the **Uniform** column of the table are preceded by "dmp_TexEnv[i].". The "starting register" in the **Register** column varies with the texture combiner number (this corresponds to the i in dmp_TexEnv[i], but there is only one register for bufferColor because it can only be set when i=0).

The following table shows the address of the starting register.

Table 5-15 Texture Combiner Numbers and Starting Registers

Combiner Number	Starting Register
0	0x0c0
1	0x0c8
2	0x0d0
3	0x0d8
4	0x0f0
5	0x0f8

5.8.19 Registers That Set Fragment Lighting

This section describes registers related to reserved fragment shader uniforms with dmp_FragmentLighting, dmp_FragmentMaterial, dmp_FragmentLightSource[i], or dmp_LightEnv in their names.

5.8.19.1 Enabling and Disabling Lighting

The following table shows register settings that enable and disable lighting.

Table 5-16 Registers That Enable or Disable Lighting

Uniform	Setting Register	Setting Value
dmp_FragmentLighting.enabled	0x1c6, bits [0:0]	• 0: GL_TRUE

Uniform	Setting Register	Setting Value
		• 1: GL_FALSE
	0x8f, bits [0:0]	• 0: GL_FALSE • 1: GL_TRUE
dmp_FragmentLightSource[i].enabled	0x1c2, bits [2:0]	This sets a value that is one less than the number of enabled light sources. This is set equal to 0 when all light sources are disabled.
	0x1d9, bits [2:0]	The ID of the 1st enabled light source
	0x1d9, bits [6:4]	The ID of the 2nd enabled light source
	0x1d9, bits [10:8]	The ID of the 3rd enabled light source
	0x1d9, bits [14:12]	The ID of the 4th enabled light source
	0x1d9, bits [18:16]	The ID of the 5th enabled light source
	0x1d9, bits [22:20]	The ID of the 6th enabled light source
	0x1d9, bits [26:24]	The ID of the 7th enabled light source
	0x1d9, bits [30:28]	The ID of the 8th enabled light source

The IDs of the enabled light sources are specified in `0x1d9` in ascending order (starting at light source 0). For example, when light sources 0, 1, 3, and 5 are enabled (`dmp_FragmentLightSource[0].enabled`, `dmp_FragmentLightSource[1].enabled`, `dmp_FragmentLightSource[3].enabled`, and `dmp_FragmentLightSource[5].enabled` are all `GL_TRUE`), `0x1d9` is set equal to `0x00005310`. When all light sources are enabled, a value of `0x76543210` is set. When all light sources are disabled, a value of 0 is set.

5.8.19.2 Global Ambient Settings

This section describes global ambient settings. Before each RGB component is set in a register, it is first calculated as `dmp_FragmentMaterial.emission` + `dmp_FragmentMaterial.ambient` × `dmp_FragmentLighting.ambient`, clamped to a value between 0 and 1, and then mapped to an unsigned, 8-bit integer between 0 and 255. Bits [29:20], [19:10], and [9:0] of register `0x1c0` set the R, G, and B components, respectively. For information on how to convert a floating-point number clamped between 0 and 1 into an 8-bit integer between 0 and 255, see section 5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer.

If light source 0 is not enabled, `0` will be applied as the global ambient term of the primary color, even if something is set for it in this settings register.

When lighting is enabled and all light sources are disabled (`dmp_FragmentLighting.enabled` is set to `GL_TRUE` and `dmp_FragmentLightSource[i].enabled` is set to `GL_FALSE` for all light sources), only the global ambient is applied to the primary color. Because enabling lighting also always enables one light source (due to the fact that bits [2:0] of register `0x1c2` set the number of light sources minus one), the DMPGL driver generates a command that sets `0x140`, `0x141`, `0x142`, and `0x143` to 0. This command sets all light source colors for light source 0 to black (`0.0, 0.0, 0.0, 0.0`).

The driver also generates two commands: one that enables light source 0 by taking the first enabled light source as light source 0 (setting bits [2:0] of register `0x1d9` to `0`), and one that improves performance by setting `dmp_LightEnv.config` to `GL_LIGHT_ENV_LAYER_CONFIG0_DMP` (setting bits [7:4] of register `0x1c3` to 0).

5.8.19.3 Per-Light Settings

This section describes how to configure individual light sources.

Register addresses and bits corresponding to per-light settings are calculated from light source numbers. A light source number corresponds to `i` in the uniform name, `dmp_FragmentLightSource[i].XXX`.

When `dmp_FragmentLightSource[0].enabled` and `dmp_FragmentLightSource[3].enabled` are set equal to `GL_TRUE`, for example, light sources 0 and 3 are enabled. The light source colors (explained later under **Light Source Color Settings**) `dmp_FragmentSource[0].specular0` and `dmp_FragmentSource[3].specular0` affect registers `0x140` and `0x170`, respectively.

Light Source Color Settings

There are ambient, diffuse, specular 0, and specular 1 settings for each enabled light source. The following table shows the registers that set each component.

Table 5-17 Registers That Set Each Color Component

Component	Setting Register	Setting Value (for each RGB component)
Specular 0	0x140 + (light source #) x 0x10	`dmp_FragmentMaterial.specular0 x dmp_FragmentLightSource[i].specular0`
Specular 1	0x140 + (light source #) x 0x10 + 1	When `dmp_LightEnv.lutEnabledRef1` is `GL_FALSE`: `dmp_FragmentMaterial.specular1 x dmp_FragmentLightSource[i].specular1` When `dmp_LightEnv.lutEnabledRef1` is `GL_TRUE`: `dmp_FragmentLightSource[i].specular1`
Diffuse	0x140 + (light source #) x 0x10 + 2	`dmp_FragmentMaterial.diffuse x dmp_FragmentLightSource[i].diffuse`
Ambient	0x140 + (light source #) x 0x10 + 3	`dmp_FragmentMaterial.ambient x dmp_FragmentLightSource[i].ambient`

The specular 0, specular 1, diffuse, and ambient RGB components are calculated as shown in Table 5-17 to produce floating-point numbers between 0 and 1, which are then mapped to integers between 0 and 255 and set in the corresponding registers. Bits [29:20], [19:10], and [9:0] are used for the R, G, and B components, respectively. For information on how to convert floating-point values into integer values, see section 5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer.

Setting Light Source Positions

The reserved uniform `dmp_FragmentLightSource[i].position` specifies the light source positions. The x, y, and z coordinates specified by the uniform are converted into 16-bit floating-point numbers before they are set as register values. For information on how to convert these numbers, see section 5.9.2 Converting from float32 to float16.

The following table shows the registers that set each component.

Table 5-18 Registers That Set Individual Components of Light Source Coordinates

Coordinate Component	Setting Register	Bits	Setting Value
X	0x140 + (light source #) × 0x10 + 4	[15:0]	16-bit floating-point number
Y	0x140 + (light source #) × 0x10 + 4	[31:16]	16-bit floating-point number
Z	0x140 + (light source #) × 0x10 + 5	[15:0]	16-bit floating-point number
W	0x140 + (light source #) × 0x10 + 9	[0:0]	1 when the fourth component of `dmp_FragmentLightSource[i].position` is 0 and 0 otherwise.

Setting the Spotlight Direction

The reserved uniform `dmp_FragmentLightSource[i].spotDirection` specifies the spotlight direction. The x, y, and z components specified by the register are first negated, then converted into 13-bit signed fixed-point numbers with 11 fractional bits, and finally set as register values. For each of these values, the most-significant bit indicates the sign and is followed by a single integer bit and 11 fractional bits, respectively. Negative values are represented in two's complement. For information on how to convert these numbers, see section 5.9.9 Converting a 32-Bit Floating-Point Number into a 13-Bit Signed Fixed-Point Number with 11 Fractional Bits.

The following table shows the registers that set each component.

Table 5-19 Registers That Set Individual Components of the Spotlight Direction

Component	Setting Register	Bits	Setting Value
X	0x140 + (light source #) x 0x10 + 6	[12:0]	13-bit fixed-point number
Y	0x140 + (light source #) x 0x10 + 6	[28:16]	13-bit fixed-point number
Z	0x140 + (light source #) x 0x10 + 7	[12:0]	13-bit fixed-point number

Bias and Scale Settings for Distance Attenuation

The reserved uniforms `dmp_FragmentLightSource[i].distanceAttenuationBias` and `dmp_FragmentLightSource[i].distanceAttenuationScale` specify the bias and scale values for distance attenuation, respectively. The values set for each of these registers are converted into 20-bit floating-point numbers and set in the registers. For more information on this conversion, see section 5.9.4 Converting from float32 to float20. The following table shows the registers to set.

Table 5-20 Setting Registers for the Bias and Scale with Distance Attenuation

Component	Setting Register	Bits	Setting Value
Bias	$0x140 + (light\ source\ \#) \times 0x10 + 0x0a$, bits [19:0]	[19:0]	20-bit floating-point number
Scale	$0x140 + (light\ source\ \#) \times 0x10 + 0x0b$, bits [19:0]	[19:0]	20-bit floating-point number

Miscellaneous Settings for Individual Lights

The following table shows registers used by other miscellaneous settings for individual light sources.

Table 5-21 Registers Used by Other Miscellaneous Settings for Individual Light Sources

Uniform	Setting Register	Setting Value
`dmp_FragmentLightSource[i].shadowed`	`0x1c4`, bit $[(light\ source\ \#)\ :(light\ source\ \#)\ ]$	• 0: `GL_TRUE` • 1: `GL_FALSE`
`dmp_FragmentLightSource[i].spotEnabled`	`0x1c4`, bit $[8 + (light\ source\ \#)\ :8 + (light\ source\ \#)\ ]$	• 0: `GL_TRUE` • 1: `GL_FALSE`
`dmp_FragmentLightSource[i].distanceAttenuationEnabled`	`0x1c4`, bit $[24 + (light\ source\ \#)\ :24 + (light\ source\ \#)\ ]$	• 0: `GL_TRUE` • 1: `GL_FALSE`
`dmp_FragmentLightSource[i].twoSideDiffuse`	$0x140 + (light\ source\ \#) \times 0x10 + 9$, bit [1:1]	• 0: `GL_FALSE` • 1: `GL_TRUE`
`dmp_FragmentLightSource[i].geomFactor0`	$0x140 + (light\ source\ \#) \times 0x10 +$ 90x140, bit [2:2]	• 0: `GL_FALSE` • 1: `GL_TRUE`
`dmp_FragmentLightSource[i].geomFactor1`	$0x140 + (light\ source\ \#) \times 0x10 +$ 90x140, bit [3:3]	• 0: `GL_FALSE` • 1: `GL_TRUE`

5.8.19.4 Lookup Table Settings

This section describes settings for the reserved uniforms `dmp_FragmentMaterial.sampler{RR,RG,RB,D0,D1,SP,FR}`, `dmp_FragmentLightSource[i].samplerSP`, and `dmpFragmentLightSource[i].samplerDA`. Each type of lookup table for fragment lighting has 256 data elements. The following table shows the registers used for each setting.

Table 5-22 Registers That Configure Lookup Tables for Fragment Lighting

Setting Register	Description
`0x1c5`, bits [7:0]	Specifies the index at which to set data in the lookup table.
`0x1c5`, bits [11:8]	Specifies the type of lookup table for which to set data. • 0: `dmp_FragmentMaterial.samplerD0` • 1: `dmp_FragmentMaterial.samplerD1` • 3: `dmp_FragmentMaterial.samplerFR` • 4: `dmp_FragmentMaterial.samplerRB` • 5: `dmp_FragmentMaterial.samplerRG` • 6: `dmp_FragmentMaterial.samplerRR` • 8+`i`: `dmp_FragmentLightSource[i].samplerSP` • 16+`i`: `dmp_FragmentLightSource[i].samplerDA`
`0x1c8`–`0x1cf`, bits [23:0]	Sets the lookup table data.

Use bits [11:8] of `0x1c5` to select the type of lookup table to configure. Before configuring more than one type of lookup table, you need to switch the table type with the same register. Use bits [7:0] of `0x1c5` to specify the index of the data to set. An index value of 0 indicates the start of the lookup table and 255 indicates the end.

Set lookup table data anywhere between `0x1c8` and `0x1cf`. When data is written, it updates the content of the lookup table at the specified index. The index is incremented by one for each data element that is written.

The `i`'th element and the `(i + 256)`'th element of the 512 data elements loaded by the **`glTexImage1D`** function are packed into a value that is written to index `i` of the lookup table object bound to the lookup table number specified by the uniform value. Convert the `i`'th data element into a 12-bit unsigned fixed-point number with 12 fractional bits and write it to bits [11:0] of any register between `0x1c8` and `0x1cf`. Convert the `(i+256)`'th data element into a 12-bit signed fixed-point number with 11 fractional bits and write it to bits [11:0] of any register between `0x1c8` and `0x1cf`. Results are the same regardless of where you write data between `0x1c8` and `0x1cf`.

For information on how to convert 12-bit unsigned fixed-point numbers with 12 fractional bits, see section 5.9.13 Converting a 32-Bit Floating-Point Number into a 12-Bit Unsigned Fixed-Point Number with 12 Fractional Bits.

For a 12-bit signed fixed-point number with 11 fractional bits, the most-significant bit indicates the sign and is followed by 11 fractional bits that specify an absolute value (negative values are not represented in two's complement). For more details on converting into this format, see section 5.9.6 Converting a 32-Bit Floating-Point Number into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits.

5.8.19.5 Setting the Range of Lookup Table Arguments

The following table shows register settings specific to the range of lookup table arguments.

Table 5-23 Registers That Set the Range of Lookup Table Arguments

Uniform	Setting Register	Setting Value
dmp_LightEnv.absLutInputD0	0x1d0, bit [1:1]	• 0: GL_TRUE • 1: GL_FALSE
dmp_LightEnv.absLutInputD1	0x1d0, bit [5:5]	Same as dmp_LightEnv.absLutInputD0.
dmp_LightEnv.absLutInputSP	0x1d0, bit [9:9]	Same as dmp_LightEnv.absLutInputD0.
dmp_LightEnv.absLutInputFR	0x1d0, bit [13:13]	Same as dmp_LightEnv.absLutInputD0.
dmp_LightEnv.absLutInputRB	0x1d0, bit [17:17]	Same as dmp_LightEnv.absLutInputD0.
dmp_LightEnv.absLutInputRG	0x1d0, bit [21:21]	Same as dmp_LightEnv.absLutInputD0.
dmp_LightEnv.absLutInputRR	0x1d0, bit [25:25]	Same as dmp_LightEnv.absLutInputD0.

5.8.19.6 Setting Lookup Table Input Values

The following table shows register settings specific to lookup table input values.

Table 5-24 Registers That Set Lookup Table Input Values

Uniform	Setting Register	Setting Value
dmp_LightEnv.lutInputD0	0x1d1, bits [2:0]	• 0: GL_LIGHT_ENV_NH_DMP • 1: GL_LIGHT_ENV_VH_DMP • 2: GL_LIGHT_ENV_NV_DMP • 3: GL_LIGHT_ENV_LN_DMP • 4: GL_LIGHT_ENV_SP_DMP • 5: GL_LIGHT_ENV_CP_DMP
dmp_LightEnv.lutInputD1	0x1d1, bits [6:4]	Same as dmp_LightEnv.lutInputD0.
dmp_LightEnv.lutInputSP	0x1d1, bits [10:8]	Same as dmp_LightEnv.lutInputD0.
dmp_LightEnv.lutInputFR	0x1d1, bits [14:12]	Same as dmp_LightEnv.lutInputD0.
dmp_LightEnv.lutInputRB	0x1d1, bits [18:16]	Same as dmp_LightEnv.lutInputD0.
dmp_LightEnv.lutInputRG	0x1d1, bits [22:20]	Same as dmp_LightEnv.lutInputD0.
dmp_LightEnv.lutInputRR	0x1d1, bits [26:24]	Same as dmp_LightEnv.lutInputD0.

5.8.19.7 Setting the Output Scaling for Lookup Tables

The following table shows the register settings specific to output scaling for lookup tables.

Table 5-25 Registers That Set the Output Scaling for Lookup Tables

Uniform	Setting Register	Setting Value
dmp_LightEnv.lutScaleD0	0x1d2, bits [2:0]	• 0: 1.0 • 1: 2.0 • 2: 4.0 • 3: 8.0 • 6: 0.25 • 7: 0.5
dmp_LightEnv.lutScaleD1	0x1d2, bits [6:4]	Same as dmp_LightEnv.lutScaleD0.
dmp_LightEnv.lutScaleSP	0x1d2, bits [10:8]	Same as dmp_LightEnv.lutScaleD0.
dmp_LightEnv.lutScaleFR	0x1d2, bits [14:12]	Same as dmp_LightEnv.lutScaleD0.
dmp_LightEnv.lutScaleRB	0x1d2, bits [18:16]	Same as dmp_LightEnv.lutScaleD0.
dmp_LightEnv.lutScaleRG	0x1d2, bits [22:20]	Same as dmp_LightEnv.lutScaleD0.
dmp_LightEnv.lutScaleRR	0x1d2, bits [26:24]	Same as dmp_LightEnv.lutScaleD0.

5.8.19.8 Shadow Attenuation Settings

The following table shows register settings specific to shadow attenuation.

Table 5-26 Registers for Shadow Attenuation Settings

Uniform	Setting Register	Setting Value
dmp_LightEnv.shadowSelector	0x1c3, bits [25:24]	• 0: GL_TEXTURE0 • 1: GL_TEXTURE1 • 2: GL_TEXTURE2 • 3: GL_TEXTURE3
dmp_LightEnv.shadowPrimary	0x1c3, bit [16:16]	• 0: GL_FALSE • 1: GL_TRUE
dmp_LightEnv.shadowSecondary	0x1c3, bit [17:17]	• 0: GL_FALSE • 1: GL_TRUE
dmp_LightEnv.invertShadow	0x1c3, bit [18:18]	• 0: GL_FALSE • 1: GL_TRUE
dmp_LightEnv.shadowAlpha	0x1c3, bit [19:19]	• 0: GL_FALSE • 1: GL_TRUE
Common	0x1c3, bit [0:0]	1 when any of the following are equal to GL_TRUE and 0 when all of the following are equal to GL_FALSE. • dmp_LightEnv.shadowPrimary • dmp_LightEnv.shadowSecondary • dmp_LightEnv.shadowAlpha

5.8.19.9 Miscellaneous Settings

The following table shows register settings specific to other miscellaneous fragment lighting.

Table 5-27 Registers for Other Miscellaneous Fragment Lighting Settings

Uniform	Setting Register	Setting Value
dmp_LightEnv.config	0x1c3, bits [7:4]	• 0: GL_LIGHT_ENV_LAYER_CONFIG0_DMP • 1: GL_LIGHT_ENV_LAYER_CONFIG1_DMP • 2: GL_LIGHT_ENV_LAYER_CONFIG2_DMP • 3: GL_LIGHT_ENV_LAYER_CONFIG3_DMP • 4: GL_LIGHT_ENV_LAYER_CONFIG4_DMP • 5: GL_LIGHT_ENV_LAYER_CONFIG5_DMP • 6: GL_LIGHT_ENV_LAYER_CONFIG6_DMP • 8: GL_LIGHT_ENV_LAYER_CONFIG7_DMP
dmp_LightEnv.fresnelSelector	0x1c3, bits [3:2]	• 0: GL_LIGHT_ENV_NO_FRESNEL_DMP • 1: GL_LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP • 2: GL_LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP • 3: GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP
	0x1c4, bit [19:19]	• 0: Not GL_LIGHT_ENV_NO_FRESNEL_DMP • 1: GL_LIGHT_ENV_NO_FRESNEL_DMP
dmp_LightEnv.bumpSelector	0x1c3, bits [23:22]	• 0: GL_TEXTURE0 • 1: GL_TEXTURE1 • 2: GL_TEXTURE2 • 3: GL_TEXTURE3
dmp_LightEnv.bumpMode	0x1c3, bits [29:28]	• 0: GL_LIGHT_ENV_BUMP_NOT_USED_DMP • 1: GL_LIGHT_ENV_BUMP_AS_BUMP_DMP • 2: GL_LIGHT_ENV_BUMP_AS_TANG_DMP
dmp_LightEnv.bumpRenorm	0x1c3, bit [30:30]	• 0 when dmp_LightEnv.bumpRenorm is GL_TRUE or dmp_LightEnv.bumpMode is GL_LIGHT_ENV_BUMP_NOT_USED • 1 otherwise
dmp_LightEnv.clampHighlights	0x1c3, bit [27:27]	• 0: GL_FALSE • 1: GL_TRUE
dmp_LightEnv.lutEnabledD0	0x1c4, bit [16:16]	• 0: GL_TRUE • 1: GL_FALSE
dmp_LightEnv.lutEnabledD1	0x1c4, bit [17:17]	• 0: GL_TRUE • 1: GL_FALSE
dmp_LightEnv.lutEnabledRef1	0x1c4, bits [22:20]	• 0: GL_TRUE • 7: GL_FALSE

Note: The dmp_LightEnv.config settings, specifically the values set in bits [7:4] of register 0x1c3, change the number of cycles used for per-pixel operations. This setting has an effect even when lighting is disabled. For performance reasons, if you disable lighting at any point, at that point arrange to configure `dmp_LightEnv.config` so that the number of cycles is 1. The DMPGL driver sets bits [7:4] of register `0x1c3` to 0 when lighting is disabled.

5.8.20 Texture Setting Registers

This section describes registers related to general texture parameters and reserved uniforms with `dmp_Texture[i]` in their names. Also see section 5.8.16 Texture Address Setting Registers. The register-setting commands for texture parameters described in this section are generated during `NN_GX_STATE_TEXTURE` validation.

5.8.20.1 Shadow Texture Settings

The following table shows register settings specific to reserved uniforms for shadow textures.

Table 5-28 Shadow Texture Setting Registers

Uniform	Setting Register	Setting Value
`dmp_Texture[0].perspectiveShadow`	`0x8b`, bit [0:0]	• 0: `GL_TRUE` • 1: `GL_FALSE`
`dmp_Texture[0].shadowZBias`	`0x8b`, bits [23:1]	The uniform value converted into a 23-bit, unsigned, fixed-point number.
`dmp_Texture[0].shadowZScale`	`0x8b`, bits [31:24]	An 8-bit integer that represents the index -127 from the 32-bit floating-point uniform value

The setting value of `dmp_Texture[0].shadowZBias` is converted into a 24-bit fixed-point number, of which the most-significant 23 bits are set in the register. For information on converting to a 24-bit fixed-point number, see section 5.9.14 Converting a 32-Bit Floating-Point Number into a 24-Bit Unsigned Fixed-Point Number with 24 Fractional Bits.

5.8.20.2 Setting the Texture Sampler Type

The following table shows register settings specific to reserved uniforms for the texture sampler type.

Table 5-29 Registers That Set the Texture Sampler Type

Uniform	Setting Register	Setting Value
`dmp_Texture[0].samplerType`	`0x80`, bit [0:0]	• 0: `GL_FALSE` • 1: A setting other than `GL_FALSE`
	`0x83`, bits [30:28]	• 0: `GL_TEXTURE_2D` • 1: `GL_TEXTURE_CUBE_MAP` • 2: `GL_TEXTURE_SHADOW_2D_MAP` • 3: `GL_TEXTURE_PROJECTION_DMP` • 4: `GL_TEXTURE_SHADOW_CUBE_MAP`
`dmp_Texture[1].samplerType`	`0x80`, bit [1:1]	• 0: `GL_FALSE` • 1: `GL_TEXTURE_2D`

dmp_Texture[2].samplerType	0x80, bit [2:2]	• 0: GL_FALSE • 1: GL_TEXTURE_2D
dmp_Texture[3].samplerType	0x80, bit [10:10]	• 0: GL_FALSE • 1: GL_TEXTURE_PROCEDURAL_DMP

Note: Note that setting registers for dmp_Texture[0].samplerType, dmp_Texture[1].samplerType, and dmp_Texture[2].samplerType are generated not when the state flag is NN_GX_STATE_FSUNIFORM but when the **glDrawElements** or **glDrawArrays** function is called..

5.8.20.3 Setting the Texture Coordinate Selection

The following table shows register settings specific to reserved uniforms for texture coordinate selection.

Table 5-30 Registers for Texture Coordinate Selection

Uniform	Setting Register	Setting Value
dmp_Texture[2].texcoord	0x80, bit [13:13]	• 0: GL_TEXTURE2 • 1: GL_TEXTURE1
dmp_Texture[3].texcoord	0x80, bits [9:8]	• 0: GL_TEXTURE0 • 1: GL_TEXTURE1 • 2: GL_TEXTURE2

5.8.20.4 Procedural Texture Settings

The following table shows register settings specific to reserved uniforms for procedural textures.

Table 5-31 Registers for Procedural Texture Settings

Uniform	Setting Register	Setting Value
dmp_Texture[3].ptRgbMap	0x0a8, bits [9:6]	• 0: GL_PROCTEX_U_DMP • 1: GL_PROCTEX_U2_DMP • 2: GL_PROCTEX_V_DMP • 3: GL_PROCTEX_V2_DMP • 4: GL_PROCTEX_ADD_DMP • 5: GL_PROCTEX_ADD2_DMP • 6: GL_PROCTEX_ADDSQRT2_DMP • 7: GL_PROCTEX_MIN_DMP • 8: GL_PROCTEX_MAX_DMP • 9: GL_PROCTEX_RMAX_DMP
dmp_Texture[3].ptAlphaMap	0x0a8, bits [13:10]	Same as dmp_Texture[3].ptRgbMap
dmp_Texture[3].ptAlphaSeparate	0x0a8, bit [14:14]	• 0: GL_FALSE • 1: GL_TRUE

Uniform	Setting Register	Setting Value
`dmp_Texture[3].ptClampU`	`0x0a8`, bits [2:0]	• 0: `GL_CLAMP_TO_ZERO_DMP` • 1: `GL_CLAMP_TO_EDGE` • 2: `GL_SYMMETRICAL_REPEAT_DMP` • 3: `GL_MIRRORED_REPEAT` • 4: `GL_PULSE_DMP`
`dmp_Texture[3].ptClampV`	`0x0a8`, bits [5:3]	Same as `dmp_Texture[3].ptClampU`
`dmp_Texture[3].ptShiftU`	`0x0a8`, bits [17:16]	• 0: `GL_NONE_DMP` • 1: `GL_ODD_DMP` • 2: `GL_EVEN_DMP`
`dmp_Texture[3].ptShiftV`	`0x0a8`, bits [19:18]	Same as `dmp_Texture[3].ptShiftU`
`dmp_Texture[3].ptMinFilter`	`0x0ac`, bits [2:0]	• 0: `GL_NEAREST` • 1: `GL_LINEAR` • 2: `GL_NEAREST_MIPMAP_NEAREST` • 3: `GL_LINEAR_MIPMAP_NEAREST` • 4: `GL_NEAREST_MIPMAP_LINEAR` • 5: `GL_LINEAR_MIPMAP_LINEAR`
`dmp_Texture[3].ptTexOffset`	`0x0ad`, bits [7:0]	Sets the uniform value
`dmp_Texture[3].ptTexWidth`	`0x0ac`, bits [18:11]	Sets the uniform value
`dmp_Texture[3].ptTexBias`	`0x0a8`, bits [27:20]	Sets the least-significant 8 bits of the uniform value after it is converted into a 16-bit floating-point number
	`0x0ac`, bits [26:19]	Sets the most-significant 8 bits of the uniform value after it is converted into a 16-bit floating-point number
`dmp_Texture[3].ptNoiseEnable`	`0x0a8`, bit [15:15]	• 0: `GL_FALSE` • 1: `GL_TRUE`
`dmp_Texture[3].ptNoiseU`	`0x0a9`, bits [31:0]	Bits [31:16] are set equal to a 16-bit floating-point number converted from the second component of the uniform. Bits [15:0] are set equal to a 16-bit fixed-point number, signed with 12 decimal bits, converted from the third component of the uniform.
	`0x0ab`, bits [15:0]	The first uniform component, converted into a 16-bit floating-point number.
`dmp_Texture[3].ptNoiseV`	`0x0aa`, bits [31:0]	Bits [31:16] are set with the second uniform component, converted into a 16-bit floating-point number. Bits [15:0] are set with the third uniform component, converted into a signed, 16-bit fixed-point number with 12 decimal bits.

Uniform	Setting Register	Setting Value
	`0x0ab`, bits [31:16]	The first uniform component, converted into a 16-bit floating-point number.

For details on converting the first and second uniform component of `dmp_Texture[3].ptTexBias`, `dmp_Texture[3].ptNoiseU`, and `dmp_Texture[3].ptNoiseV`, see section 5.9.2 Converting from float32 to float16.

The third uniform component of `dmp_Texture[3].ptNoiseU` and `dmp_Texture[3].ptNoiseV` is converted into a 16-bit fixed-point number in which the most-significant bit indicates the sign and is followed by three integer bits and 12 fractional bits, respectively. Negative values are represented in two's complement. For details on converting into this format, see section 5.9.10 Converting a 32-Bit Floating-Point Number into a 16-Bit Signed Fixed-Point Number with 12 Fractional Bits.

5.8.20.5 Lookup Table Settings for Procedural Textures

This section describes settings specific to the reserved uniforms `dmp_Texture[3].ptSampler{RgbMap,AlphaMap,NoiseMap,R,G,B,A}`. There are four types of lookup table data for procedural textures: RGB-mapping F functions, alpha-mapping F functions, noise-modulation functions, and color. Each table has a different number of elements. The following table shows the registers used for each setting.

Table 5-32 Registers That Configure Lookup Tables for Procedural Textures

Setting Register	Description
`0x0af`, bits [7:0]	Specifies the index at which to set data in the lookup table.
`0x0af`, bits [11:8]	Specifies the type of lookup table for which to set data. • 0: Noise-modulation functions • 2: RGB-mapping F functions • 3: Alpha-mapping F functions • 4: Color (color values) • 5: Color (difference values)
`0x0b0`–`0x0b7`, bits [31:0]	Sets the lookup table data.

Use bits [11:8] of `0x0af` to select the type of lookup table to configure. Before configuring more than one type of lookup table, you need to switch the table type with the same register. Use bits [7:0] of `0x0af` to specify the index of the data to set. An index value of 0 indicates the start of the lookup table and 1 indicates the second element.

Set lookup table data anywhere between `0x0b0` and `0x0b7`. When data is written, it updates the content of the lookup table at the specified index. The index is incremented by one for each data element that is written. Results are the same regardless of where you write data between `0x0b0` and `0x0b7`. A value of 1 must be written to bit [10:10] of register `0x80` (to enable procedural textures) when you set a value in registers `0x0b0`–`0x0b7`. If bit [10:10] of register `0x80` is 0, attempts to set registers `0x0b0`–`0x0b7` are ignored.

The format and size of data to write to the lookup table varies with the lookup table type.

Lookup Tables for RGB-Mapping F Functions, Alpha-Mapping F Functions, and Noise Modulation Functions

The lookup table for RGB-mapping F functions uses data loaded by the **glTexImage1D** function for the lookup table object bound to the lookup table number specified by `dmp_Texture[3].ptSamplerRgbMap`.

Similarly, the lookup table for alpha-mapping F functions uses data from the lookup table object specified by `dmp_Texture[3].ptSamplerAlphaMap`, and the lookup table for noise modulation functions uses data from the lookup table object specified by `dmp_Texture[3].ptSamplerNoiseMap`. There are 128 data items in the lookup table, and the index in 0x0af bits [7:0] can specify values from 0 to 127.

The data to write to index i of the lookup table is a value that packs the i'th and (i+128)'th element of the 256 data elements loaded by the **glTexImage1D** function. Convert the `i`'th data element into a 12-bit unsigned fixed-point number with 12 fractional bits and write it to bits [11:0]. Convert the `(i+128)`'th data element into a 12-bit signed fixed-point number with 12 fractional bits and write it to bits [23:12].

For details on converting 12-bit unsigned fixed-point numbers with 12 fractional bits, see section 5.9.13 Converting a 32-Bit Floating-Point Number into a 12-Bit Unsigned Fixed-Point Number with 12 Fractional Bits.

For a 12-bit signed fixed-point number with 11 fractional bits, the most-significant bit indicates the sign and is followed by 11 fractional bits. Negative values are represented in two's complement. For more details on converting into this format, see section 5.9.7 Converting a 32-Bit Floating-Point Number into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits (Alternate Method).

Color Lookup Tables

Color lookup tables use data loaded by the **glTexImage1D** function on the lookup table object bound to the lookup table number specified by `dmp_Texture[3].ptSampler{R,G,B,A}`. The color value and difference value lookup tables both contain 256 data items, and the index in 0x0af bits [7:0] can specify values from 0 to 255.

A packed value (the color value) is written to index `i` of a color lookup table using the `i`'th data element (of a maximum of 512) loaded by the **glTexImage1D** function for each RGBA color channel. The first 256 elements (of 512) are used. The `i`'th floating-point number between 0 and 1 is mapped to an integer between 0 and 255, and then data is written with the R, G, B, and A components in bits [7:0], [15:8], [23:16], and [31:24], respectively. For more details on this conversion, see section 5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer.

A packed value (the difference value) is written to index `i` of a color lookup table using the `(256+i)`'th data element (of a maximum of 512) loaded by the **glTexImage1D** function for each RGBA channel. The last 256 elements (of 512) are used. The `(i+256)`'th floating-point number is converted into an 8-bit signed fixed-point number with 7 fractional bits and then data is written with the R, G, B, and A components in bits [7:0], [15:8], [23:16], and [31:24], respectively. For each of these values, the most-significant bit indicates the sign and is followed by 7 fractional bits. Negative values are represented in two's complement. For details on this conversion, see section 5.9.5 Converting a 32-Bit Floating-Point Number into an 8-Bit Signed Fixed-Point Number with 7 Fractional Bits.

5.8.20.6 Texture Resolution

The following table shows registers that set the width and height of textures.

Table 5-33 Registers That Set the Texture Resolution

Setting Register	Description
`0x82`, bits [26:16]	Texture 0's width
`0x82`, bits [10:0]	Texture 0's height
`0x92`, bits [26:16]	Texture 1's width
`0x92`, bits [10:0]	Texture 1's height
`0x9a`, bits [26:16]	Texture 2's width
`0x9a`, bits [10:0]	Texture 2's height

5.8.20.7 Texture Formats

The following table shows registers that set the texture format

Table 5-34 Registers for Texture Format Settings

Setting Register	Description
`0x83`, bits [5:4]	Configures texture 0's format using the following values. • 0: Any value except `GL_ETC1_RGB8_NATIVE_DMP` • 2: `GL_ETC1_RGB8_NATIVE_DMP`
`0x93`, bits [5:4]	Configures texture 1's format using the same values as bits [5:4] of `0x83`.
`0x9b`, bits [5:4]	Configures texture 2's format using the same values as bits [5:4] of `0x83`.

Setting Register	Description
`0x8e`, bits [3:0]	Sets the following values corresponding to the ***format*** and ***type*** arguments to the **`glTexImage2D`** function and the ***internalformat*** argument to the **`glCompressedTexImage2D`** function for texture 0. • 0: `GL_RGBA` and `GL_UNSIGNED_BYTE`; `GL_SHADOW_DMP` and `GL_UNSIGNED_INT`; or `GL_GAS_DMP` and `GL_UNSIGNED_SHORT` • 1: `GL_RGB`, `GL_UNSIGNED_BYTE` • 2: `GL_RGBA`, `GL_UNSIGNED_SHORT_5_5_5_1` • 3: `GL_RGB`, `GL_UNSIGNED_SHORT_5_6_5` • 4: `GL_RGBA`, `GL_UNSIGNED_SHORT_4_4_4_4` • 5: `GL_LUMINANCE_ALPHA`, `GL_UNSIGNED_BYTE` • 6: `GL_HILO8_DMP`, `GL_UNSIGNED_BYTE` • 7: `GL_LUMINANCE`, `GL_UNSIGNED_BYTE` • 8: `GL_ALPHA`, `GL_UNSIGNED_BYTE` • 9: `GL_LUMINANCE_ALPHA`, `GL_UNSIGNED_BYTE_4_4_DMP` • 10: `GL_LUMINANCE`, `GL_UNSIGNED_4BITS_DMP` • 11: `GL_ALPHA`, `GL_UNSIGNED_4BITS_DMP` • 12: `GL_ETC1_RGB8_NATIVE_DMP` • 13: `GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP` (The native format uses the same setting values as the non-native format corresponding to the above.)
`0x96`, bits [3:0]	Configures texture 1's format using the same values as bits [3:0] of `0x8e`.
`0x9e`, bits [3:0]	Configures texture 2's format using the same values as bits [3:0] of `0x8e`.

5.8.20.8 Texture WRAP Modes

The following table shows registers that set texture WRAP modes.

Table 5-35 Registers for Texture WRAP Mode Settings

Setting Register	Description
`0x83`, bits [14:12]	Sets the following values for texture 0's `GL_TEXTURE_WRAP_S` parameter. • 0: `GL_CLAMP_TO_EDGE` • 1: `GL_CLAMP_TO_BORDER` • 2: `GL_REPEAT` • 3: `GL_MIRRORED_REPEAT`
`0x83`, bits [10:8]	Sets a value for texture 0's `GL_TEXTURE_WRAP_T` parameter using the same settings as bits [14:12] of `0x83`.
`0x93`, bits [14:12]	Sets a value for texture 1's `GL_TEXTURE_WRAP_S` parameter using the same settings as bits [14:12] of `0x83`.
`0x93`, bits [10:8]	Sets a value for texture 1's `GL_TEXTURE_WRAP_T` parameter using the same settings as bits [14:12] of `0x83`.
`0x9b`, bits [14:12]	Sets a value for texture 2's `GL_TEXTURE_WRAP_S` parameter using the same settings as bits [14:12] of `0x83`.

Setting Register	Description
0x9b, bits [10:8]	Sets a value for texture 2's GL_TEXTURE_WRAP_T parameter using the same settings as bits [14:12] of 0x83.

5.8.20.9 Texture Filter Modes

The following table shows registers that set texture filter modes.

Table 5-36 Registers for Texture Filter Mode Settings

Setting Register	Description
0x83, bit [1:1]	Sets the following values for texture 0's GL_TEXTURE_MAG_FILTER parameter. • 0: GL_NEAREST • 1: GL_LINEAR
0x83, bit [2:2]	Sets the following values for texture 0's GL_TEXTURE_MIN_FILTER parameter. • 0: GL_NEAREST, GL_NEAREST_MIPMAP_XXX • 1: GL_LINEAR, GL_LINEAR_MIPMAP_XXX
0x83, bit [24:24]	Sets the following values for texture 0's GL_TEXTURE_MIN_FILTER parameter. • 0: GL_NEAREST, GL_LINEAR, GL_XXX_MIPMAP_NEAREST • 1: GL_XXX_MIPMAP_LINEAR
0x93, bit [1:1]	Sets a value for texture 1's GL_TEXTURE_MAG_FILTER parameter using the same settings as bit [1:1] of 0x83.
0x93, bit [2:2]	Sets a value for texture 1's GL_TEXTURE_MIN_FILTER parameter using the same settings as bit [2:2] of 0x83.
0x93, bit [24:24]	Sets a value for texture 1's GL_TEXTURE_MIN_FILTER parameter using the same settings as bit [24:24] of 0x83.
0x9b, bit [1:1]	Sets a value for texture 2's GL_TEXTURE_MAG_FILTER parameter using the same settings as bit [1:1] of 0x83.
0x9b, bit [2:2]	Sets a value for texture 2's GL_TEXTURE_MIN_FILTER parameter using the same settings as bit [2:2] of 0x83.
0x9b, bit [24:24]	Sets a value for texture 2's GL_TEXTURE_MIN_FILTER parameter using the same settings as bit [24:24] of 0x83.

5.8.20.10 Texture Level of Detail

The following table shows registers that configure texture level of detail (LOD) settings.

Table 5-37 Registers for Texture LOD Settings

Setting Register	Description
0x84, bits [27:24]	Sets the minimum LOD for texture 0. This is 0 when the GL_TEXTURE_MIN_FILTER parameter is configured to not use LOD (GL_LINEAR or GL_NEAREST). This is the value of GL_TEXTURE_MIN_LOD (or 0 if GL_TEXTURE_MIN_LOD ≤ 0) when the GL_TEXTURE_MIN_FILTER parameter is configured to use LOD (GL_XXX_MIPMAP_XXX).

Setting Register	Description
0x84, bits [19:16]	Sets the maximum LOD for texture 0. This is 0 when the GL_TEXTURE_MIN_FILTER parameter is configured to not use LOD (GL_LINEAR or GL_NEAREST). This is one less than the number of mipmap levels loaded by the **glTexImage2D** or **glCompressedTexImage2D** function when the GL_TEXTURE_MIN_FILTER parameter is configured to use LOD (GL_XXX_MIPMAP_XXX).
0x94, bits [27:24]	This sets the minimum LOD for texture 1 in the same way as bits [27:24] of 0x84.
0x94, bits [19:16]	This sets the maximum LOD for texture 1 in the same way as bits [19:16] of 0x84.
0x9c, bits [27:24]	This sets the minimum LOD for texture 2 in the same way as bits [27:24] of 0x84.
0x9c, bits [19:16]	This sets the maximum LOD for texture 2 in the same way as bits [19:16] of 0x84.

5.8.20.11 Texture Border Color

The following table shows registers that set the texture border color.

Table 5-38 Registers for Texture Border Color Settings

Setting Register	Description
0x81, bits [31:0]	Sets the border color for texture 0. Each value set by the GL_TEXTURE_BORDER_COLOR parameter is first converted according to the method described in section 5.9.17 Alternate Conversion from a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer. The red, green, blue, and alpha components are then set in bits [7:0], [15:8], [23:16], and [31:24], respectively.
0x91, bits [31:0]	Sets the border color for texture 1 in the same way as bits [31:0] of 0x81.
0x99, bits [31:0]	Sets the border color for texture 2 in the same way as bits [31:0] of 0x81.

5.8.20.12 Registers for Texture LOD Bias Settings

The following table shows registers that set texture LOD biases.

Table 5-39 Registers for Texture LOD Bias Settings

Setting Register	Description
0x84, bits [12:0]	Sets the LOD bias for texture 0. This is converted from the value set for the GL_TEXTURE_LOD_BIAS parameter according to the method described in section 5.9.8 Converting a 32-Bit Floating-Point Number into a 13-Bit Signed Fixed-Point Number with 8 Fractional Bits.
0x94, bits [12:0]	Sets the LOD bias for texture 1 in the same way as bits [12:0] of 0x84.
0x9c, bits [12:0]	Sets the LOD bias for texture 2 in the same way as bits [12:0] of 0x84.

5.8.20.13 Shadow Texture Settings

When shadow-format textures are in use, GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T apply the GL_CLAMP_TO_BORDER setting for 2D textures and the GL_CLAMP_TO_EDGE setting for cube map textures, regardless of the values set by the **glTexParameter** function. GL_TEXTURE_MAG_FILTER and GL_TEXTURE_MIN_FILTER apply the GL_LINEAR setting for both 2D textures and cube map textures.

Bit [20:20] of register 0x83 is also set equal to 1 (the same bit is 0 for formats other than shadow textures).

5.8.20.14 Gas Texture Use Settings

When gas-format textures are in use, GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T apply the GL_CLAMP_TO_EDGE setting regardless of the values set by the **glTexParameter** function. GL_TEXTURE_MAG_FILTER and GL_TEXTURE_MIN_FILTER apply the GL_NEAREST setting.

5.8.20.15 Clearing the Texture Caches

All texture caches (both Level 1 and Level 2) are cleared when when 1 is written to bit [16:16] of register 0x80. The caches must be cleared when texture unit settings are changed but they do not need to be cleared when textures continue to be used with the same settings.

Each texture unit has a Level 1 (L1) texture cache. To clear it, the texture unit must be enabled. In other words, texture units must be enabled by bits [2:0] of register 0x80 before a value of 1 is written to bit [16:16] of register 0x80.

Even though register 0x80 holds the bits that are used to enable and disable texture units as well as the bit that is used to clear the texture caches, a single command cannot both enable texture units that are disabled and properly clear the texture caches. A separate command must be set to enable textures before the command that clears the texture caches.

If texture units are already enabled, however, a single command can disable those texture units and properly clear the texture caches.

5.8.21 Registers for Gas Settings

This section describes settings registers specific to gas features.

5.8.21.1 Gas-Related Reserved Uniform Settings

The following table shows register settings specific to gas reserved uniforms.

Table 5-40 Registers for Gas Settings

Uniform	Setting Register	Setting Value
dmp_Gas.lightXY	0x120, bits [23:0]	Each uniform component is converted into an 8-bit integer between 0 and 255. The first, second, and third components are written to bits [7:0], [15:8], and [23:16], respectively.
dmp_Gas.lightZ	0x121, bits [23:0]	Each uniform component is converted into an 8-bit integer between 0 and 255. The first, second, and third components are written to bits [7:0], [15:8], and [23:16], respectively.
	0x122, bits [7:0]	The fourth uniform component is converted into an 8-bit integer between 0 and 255 before it is set.

Uniform	Setting Register	Setting Value
dmp_Gas.deltaZ	0x126, bits [23:0]	The uniform value is converted into a 24-bit unsigned fixed-point number with 8 fractional bits before it is set.
dmp_Gas.accMax	0x0e5, bits [15:0]	The uniform value is converted into a 16-bit floating-point number before it is set.
dmp_Gas.attenuation	0x0e4, bits [15:0]	The uniform value is converted into a 16-bit floating-point number before it is set.
dmp_Gas.colorLutInput	0x122, bit [8:8]	• GL_GAS_DENSITY_DMP • GL_GAS_LIGHT_FACTOR_DMP
dmp_Gas.shadingDensitySrc	0x0e0, bit [3:3]	• GL_GAS_PLAIN_DENSITY_DMP • GL_GAS_DEPTH_DENSITY_DMP
dmp_Gas.autoAcc	0x125, bits [15:0]	This setting is cleared to 0 before density information is rendered and is updated by **nngxSetGasAutoAccumulationUpdate** after density information is rendered. For details, see the description of this uniform following the table.

The uniform values for dmp_Gas.lightXY and dmp_Gas.lightZ are floating-point numbers between 0 and 1; they are converted (mapped) into 8-bit integers between 0 and 255 before they are set. For more information on how to convert these numbers, see section 5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer.

The value of the dmp_Gas.deltaZ uniform is converted into a 24-bit unsigned fixed-point number with 8 fractional bits before it is set. For more information on how to convert these numbers, see section 5.9.15 Converting a 32-Bit Floating-Point Number into a 24-Bit Unsigned Fixed-Point Number with 8 Fractional Bits.

The uniform values for dmp_Gas.accMax and dmp_Gas.attenuation are converted into 16-bit floating-point numbers (with a 1-bit sign, 5-bit exponent, and 10-bit mantissa) before they are set. For more information on how to convert these numbers, see section 5.9.2 Converting from float32 to float16.

The value of dmp_Gas.autoAcc must be set differently from other formats. To implement dmp_Gas.autoAcc, set bits [15:0] of register 0x0e5 equal to the maximum value for the additive blend result $D1$, which is automatically calculated when gas density information is rendered. The maximum value of the additive blend result $D1$D1 is cleared to 0 before density information is rendered. You can clear this value by writing 0 to bits [15:0] of register 0x125 (the value is initialized by the contents of this register). After density information has been rendered, use the **nngxSetGasAutoAccumulationUpdate** function to apply the maximum value of the additive blend result $D1$, which is automatically calculated, to bits [15:0] of register 0x0e5. For more details, see section 3.3.22 Updating Additive Blend Results Rendered with Gas Density Information.

5.8.21.2 Shading Lookup Table Settings

This section describes shading lookup table settings. The shading lookup table has 16 data elements. Data loaded by the **glTexImage1D** function is set in the lookup table objects bound to the lookup table numbers specified by the reserved uniforms `dmp_Gas.sampler{TR,TG,TB}` for each RGB channel. The following table shows the registers used to set values.

Table 5-41 Registers That Set the Shading Lookup Table

Setting Register	Description
`0x123`, bits [15:0]	Specifies the lookup table index for which to set data.
`0x124`, bits [31:0]	Sets the lookup table data.

Bits [15:0] of register `0x123` specify the lookup table index. There are 16 data elements in the lookup table, so valid specifiable index values range from 0 to 15.

Lookup table data is set by `0x124`. When data is written, it updates the content of the lookup table at the specified index. The index is incremented by one for each data element that is written. The first and last eight elements of the lookup table are set differently.

For the first eight elements, a packed value is written to index `i` (`i` < 8) using the `(i+8)`'th of 16 data units loaded by the **glTexImage1D** function for each RGB channel. Data is converted into an 8-bit signed integer for each RGB component. The R, G, and B components are written to bits [7:0], [15:8], and [23:16], respectively. For more information on how to convert these numbers, see section 5.9.18 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Signed Integer.

For the last eight elements, a packed value is written to index `i` (`i` >= 8) using the `(i-8)`'th of 16 data units loaded by the **glTexImage1D** function for each RGB channel. The RGB components are multiplied by 255 and then converted into 8-bit unsigned fixed-point numbers with no fractional bits. The R, G, and B components are written to bits [7:0], [15:8], and [23:16], respectively. For more information on how the numbers are converted after they are multiplied by 255, see section 5.9.11 Converting a 32-Bit Floating-Point Number into an 8-Bit Unsigned Fixed-Point Number with No Fractional Bits.

Dummy commands are sometimes required before commands that set the gas shading lookup table. Specifically, 45 dummy commands are necessary before the gas shading lookup table can be set immediately following a command that sets registers `0x100`–`0x13f`, registers `0x0`–`0x35`, or any other register address not mentioned in this document. Any command that sets a register other than the ones just mentioned can be used as a dummy command. A single dummy command that uses a byte enable setting of 0 is also required for register `0x100` following a command that sets the shading lookup table.

A value of 7 must have been written to bits [2:0] of register `0x0e0` when you set register `0x124`. Attempts to set register `0x124` are ignored when bits [2:0] of register `0x0e0` have a value other than 7.

5.8.22 Fog Setting Registers

This section describes register settings specific to fog features.

5.8.22.1 Fog-related Reserved Uniform Settings

The table below shows the register settings specific to reserved uniforms for fog.

Table 5-42 Fog Setting Registers

Uniform	Setting Register	Setting Value
`dmp_Fog.mode`	`0x0e0`, bits [2:0]	• 0: `GL_FALSE` • 5: `GL_FOG` • 7: `GL_GAS_DMP`
`dmp_Fog.color`	`0x0e1`, bits [23:0]	Each element of the uniform is converted to an 8-bit integer value from 0 to 255, with the first element stored in bits [7:0], the second element in bits [15:8], and the third element in bits [23:16].
`dmp_Fog.zFlip`	`0x0e0`, bits [16:16]	• 0: `GL_FALSE` • 1: `GL_TRUE`

The `dmp_Fog.color` uniform value is set by mapping the floating point values in the [0, 1] range to 8-bit integers in the [0, 255] range. See section 5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer for details on the conversion method.

5.8.22.2 Fog Lookup Table Settings

This section describes the fog lookup table settings. The fog lookup table contains 128 pieces of data. The data loaded by the **`glTexImage1D`** function is set to the lookup table object bound to the lookup table number specified by the `dmp_Fog.sampler` reserved uniform. The table below shows the registers used for these settings.

Table 5-43 Fog Lookup Table Setting Registers

Setting Register	Description
`0x0e6`, bits [15:0]	Specifies the index of the lookup table to which data is set.
`0x0e8`–`0x0ef`, bits [23:0]	Sets lookup table data.

Set the lookup table index in register `0x0e6`, bits [15:0]. There are 128 data values in the lookup table, so valid specifiable index values range from 0 to 127.

Set the lookup table data anywhere in registers `0x0e8` through `0x0ef`. Writing the data updates the location within the lookup table pointed to by the index. The index is incremented by one every time a unit of data is written. Data may be written anywhere in registers `0x0e8` through `0x0ef`.

The data written to index `i` is the `i`th data unit of the 256 units of data loaded by the **`glTexImage1D`** function packed together with the (`i + 128`)th data unit. The `i`th data unit is converted to an 11-bit unsigned fixed-point with 11 fractional bits, which is then written to bits [23:13], while the (`i + 128`)th

data unit is converted to a 13-bit signed fixed-point with 11 fractional bits, which is then written to bits [12:0].

See section 5.9.12 Converting a 32-Bit Floating-Point Number into an 11-Bit Unsigned Fixed-Point Number with 11 Fractional Bits for details on conversion to an 11-bit unsigned fixed-point number with 11 fractional bits.

For a 13-bit signed fixed-point number with 11 fractional bits, the most-significant bit indicates the sign and is followed by one integer bit and 11 fractional bits, respectively. See section 5.9.9 Converting a 32-Bit Floating-Point Number into a 13-Bit Signed Fixed-Point Number with 11 Fractional Bits for details on conversion to a 13-bit signed fixed-point number with 11 fractional bits.

5.8.23 Fragment Operation Setting Registers

The table below shows the register settings specific to reserved uniforms for fragment operations.

Table 5-44 Fragment Operation Setting Registers

Uniform	Setting Register	Setting Value
`dmp_FragOperation.mode`	`0x100`, bits [1:0]	• 0: `GL_FRAGOP_MODE_GL_DMP` • 1: `GL_FRAGOP_MODE_GAS_ACC_DMP` • 3: `GL_FRAGOP_MODE_SHADOW_DMP`

The values described in section 5.8.28 Framebuffer Access Control Setting Registers must also be changed when register values are changed for this uniform.

5.8.24 Shadow Attenuation Factor Setting Registers

The table below shows the register settings for reserved uniforms specific to shadow attenuation factors.

Table 5-45 Fragment Operation Setting Registers

Uniform	Register	Settings
`dmp_FragOperation.penumbraScale`	`0x130`, bits [31:0]	The sign for `dmp_FragOperation.penumbraScale` is reversed, then that value is converted to a 16-bit floating-point value (with 1 bit as the sign, 5 bits as the exponent, and 10 bits as the significand), which is then written to bits [31:16].
`dmp_FragOperation.penumbraBias`		The sum of `dmp_FragOperation.penumbraScale` and `dmp_FragOperation.penumbraBias` is converted to a 16-bit floating-point value (with the same format as above), which is then written to bits [15:0].

See section 5.9.2 Converting from float32 to float16 for details on conversion to a 16-bit floating-point value.

5.8.25 w Buffer Setting Registers

The following table shows the register settings specific to reserved uniforms for the w buffer.

Table 5-46 Fragment Operation Setting Registers

Uniform	Register	Settings
`dmp_FragOperation.wScale`	`0x6d`, bits [0:0]	If uniform value is 0, value set to 1; if uniform value is not 0, value set to 0.
	`0x4d`, bits [23:0]	These bits set the scale value for the z clip coordinate; they are configured by the uniform value and the **`glDepthRangef`** setting. For more details, see the explanation following this table.
	`0x4e`, bits [23:0]	These bits set the bias value for the z clip coordinate; they are configured by the uniform value and the **`glDepthRangef`** and **`glPolygonOffset`** settings. For more details, see the explanation following this table.

The value set in bits [23:0] of register `0x4d` has its sign reversed when the `dmp_FragOperation.wScale` uniform value is nonzero. These bits are set equal to (***zNear*** - ***zFar***), using the ***zNear*** and ***zFar*** arguments to the **`glDepthRangef`** function, when the `dmp_FragOperation.wScale` uniform value is 0. The actual values set in the registers are first converted into 24-bit floating-point numbers (with a single sign bit, a 7-bit exponent, and a 16-bit mantissa).

Bits [23:0] of register `0x4e` are set equal to 0 when the `dmp_FragOperation.wScale` uniform value is nonzero. These bits are set equal to the ***zNear*** argument to the **`glDepthRangef`** function when the `dmpFragOperation.wScale` uniform value is 0. If polygon offset is enabled (**`glEnable`** is called with `GL_POLYGON_OFFSET_FILL` as an argument), the offset calculated from the ***units*** argument to the **`glPolygonOffset`** function is added to the value set in bits [23:0] of register `0x4e`. The value added by the polygon offset depends on the depth buffer format: it is $units/65535$ for a 16-bit depth buffer and $units/16777215$ for a 24-bit depth buffer. These values are converted into 24-bit floating-point numbers (with a single sign bit, 7-bit exponent, and 16-bit mantissa) before being set in the register.

See section 5.9.1 Converting from float32 to float24 for details on conversion to a 24-bit floating-point value.

5.8.26 User Clip Setting Registers

The table below shows the register settings specific to reserved uniforms for user clipping.

Table 5-47 User Clip Setting Registers

Uniform	Setting Register	Setting Value
`dmp_FragOperation.enableClippingPlane`	`0x47`, bits [0:0]	• 0: `GL_FALSE` • 1: `GL_TRUE`

Uniform	Setting Register	Setting Value
dmp_FragOperation.clippingPlane	0x48, bits [23:0]	Value is the first element of the uniform converted to a 24-bit floating-point value.
	0x49, bits [23:0]	Value is the second element of the uniform converted to a 24-bit floating-point value.
	0x4a, bits [23:0]	Value is the third element of the uniform converted to a 24-bit floating-point value.
	0x4b, bits [23:0]	Value is the fourth element of the uniform converted to a 24-bit floating-point value.

See section 5.9.1 Converting from float32 to float24 for details on conversion to a 24-bit floating-point value.

5.8.27 Alpha Test Setting Registers

The table below shows the register settings specific to reserved uniforms for alpha tests.

Table 5-48 Alpha Test Setting Registers

Uniform	Setting Register	Setting Value
dmp_FragOperation.enableAlphaTest	0x104, bits [0:0]	• 0: GL_FALSE • 1: GL_TRUE
dmp_FragOperation.alphaTestFunc	0x104, bits [6:4]	• 0: GL_NEVER • 1: GL_ALWAYS • 2: GL_EQUAL • 3: GL_NOTEQUAL • 4: GL_LESS • 5: GL_LEQUAL • 6: GL_GREATER • 7: GL_GEQUAL
dmp_FragOperation.alphaRefValue	0x104, bits [15:8]	Value is the uniform value mapped to an 8-bit integer in the [0, 255] range.

See section 5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer for details on the dmp_FragOperation.alphaRefValue conversion method.

5.8.28 Framebuffer Access Control Setting Registers

This section describes the registers for setting the framebuffer read-write access controls. These might need to be changed when changing other registers specific to certain functions and reserved uniforms.

Table 5-49 Framebuffer Access Control Setting Registers

Setting Register	Setting Value
`0x112`, bits [3:0]	Value set to 0x0f if color buffer reads are required, and set to 0 if reads are not required. Color buffer reads are required if any of the following conditions are met. • A value other than `GL_FRAGOP_MODE_GL_DMP` is set in the `dmp_FragOperation.mode` reserved uniform. • The **`glColorMask`** function defines one or more components as writable, and the **`glEnable`** function has enabled `GL_BLEND`. • The **`glColorMask`** function defines one or more components as writable, and one or more components as not writable. • The **`glColorMask`** function defines one or more components as writable, and the **`glEnable`** function has enabled `GL_COLOR_LOGIC_OP`.
`0x113`, bits [3:0]	Value set to 0x0f if color buffer writes are required, and set to 0 if writes are not required. Color buffer writes are required if any of the following conditions are met. • A value other than `GL_FRAGOP_MODE_GL_DMP` is set in the `dmp_FragOperation.mode` reserved uniform. • The **`glColorMask`** function defines one or more components as writable.
`0x114`, bits [1:0]	Bit [1:1] set to 1 if depth buffer reads are required, and bit [0:0] set to 1 if stencil buffer reads are required. Set to 0 if not required. Depth buffer reads are required if any of the following conditions are met. • `GL_FRAGOP_MODE_GAS_ACC_DMP` is set in the `dmp_FragOperation.mode` reserved uniform. • `GL_FRAGOP_MODE_GL_DMP` is set in the `dmp_FragOperation.mode` reserved uniform, the **`glEnable`** function has enabled `GL_DEPTH_TEST`, and `GL_TRUE` was set for the **`glDepthMask`** function. • `GL_FRAGOP_MODE_GL_DMP` is set in the `dmp_FragOperation.mode` reserved uniform, the **`glEnable`** function has enabled `GL_DEPTH_TEST`, and the **`glColorMask`** function defines one or more components as writable. Stencil buffer reads are required if any of the following conditions are met. • `GL_FRAGOP_MODE_GAS_ACC_DMP` is set in the `dmp_FragOperation.mode` reserved uniform. • `GL_FRAGOP_MODE_GL_DMP` is set in the `dmp_FragOperation.mode` reserved uniform, the **`glEnable`** function has enabled `GL_STENCIL_TEST`, and a value other than 0 was set for the **`glStencilMask`** function. • `GL_FRAGOP_MODE_GL_DMP` is set in the `dmp_FragOperation.mode` reserved uniform, the **`glEnable`** function has enabled `GL_STENCIL_TEST`, and the **`glColorMask`** function defines one or more components as writable.
`0x115`, bits [1:0]	Bit [1:1] set to 1 if depth buffer writes are required, and bit [0:0] set to 1 if stencil buffer writes are required. Set to 0 if not required. Depth buffer writes are required if all of the following conditions are met. • `GL_FRAGOP_MODE_GL_DMP` is set in the `dmp_FragOperation.mode` reserved uniform. • The **`glEnable`** function has enabled `GL_DEPTH_TEST`. • `GL_TRUE` was set for the **`glDepthMask`** function. Stencil buffer writes are required if all of the following conditions are met. • `GL_FRAGOP_MODE_GL_DMP` is set in the `dmp_FragOperation.mode` reserved uniform. • The **`glEnable`** function has enabled `GL_STENCIL_TEST`. • A value other than 0 is set for the **`glStencilMask`** function.

The hardware does not support certain combinations of read and write access to the color, depth, and stencil buffers. Behavior is undefined if any of these unsupported combinations are set. See the following table for more information on which combinations are supported.

Table 5-50 Combinations of Framebuffer Access Control Setting Registers

0x112, Bits [3:0]	0x113, Bits [3:0]	0x114, Bits [1:0]	0x115, Bits [1:0]	Supported?
0	0	0	0	No
Nonzero	0	0	0	No
0	Nonzero	0	0	Yes
Nonzero	Nonzero	0	0	Yes
0	0	Nonzero	0	No
Nonzero	0	Nonzero	0	No
0	Nonzero	Nonzero	0	Yes
Nonzero	Nonzero	Nonzero	0	Yes
0	0	0	Nonzero	No
Nonzero	0	0	Nonzero	No
0	Nonzero	0	Nonzero	No
Nonzero	Nonzero	0	Nonzero	No
0	0	Nonzero	Nonzero	Yes
Nonzero	0	Nonzero	Nonzero	No
0	Nonzero	Nonzero	Nonzero	Yes
Nonzero	Nonzero	Nonzero	Nonzero	Yes

5.8.29 Viewport Setting Registers

The following table shows register settings specific to the viewport.

Table 5-51 Viewport Setting Registers

Setting Function	Setting Register	Setting Value
`glViewport`	`0x41`, bits [23:0]	The result of dividing ***width*** by 2 as a floating-point number and then converting the quotient into a 24-bit floating-point number.
	`0x42`, bits [31:0]	The result of dividing 2 by ***width***, converting the quotient into a 31-bit floating-point number, and finally shifting the value left by 1 bit.
	`0x43`, bits [23:0]	The result of dividing ***height*** by 2 as a floating-point number and then converting the quotient into a 24-bit floating-point number.

	`0x44`, bits [31:0]	The result of dividing 2 by ***height***, converting the quotient into a 31-bit floating-point number, and finally shifting the value left by 1 bit.
	`0x68`, bits [9:0]	Sets ***x***.
	`0x68`, bits [25:16]	Sets ***y***.

For details on the conversion used for setting registers `0x41` and `0x43`, see section 5.9.1 Converting from float32 to float24. For details on the conversion used for setting registers `0x42` and `0x44`, see section 5.9.3 Converting from float32 to float31.

When changing these settings, you may also need to change Scissoring Setting Registers (see section 5.8.35) in the same way.

5.8.30 Depth Test Setting Registers

The following table shows register settings related to depth tests.

Table 5-52 Depth Test Setting Registers

Setting Function	Setting Register	Setting Value
`glEnable/glDisable (GL_DEPTH_TEST);`	`0x107`, bit [0:0]	• 0: Disable depth tests • 1: Enable depth tests
`glDepthFunc`	`0x107`, bits [6:4]	Corresponds to the ***func*** argument: • 0: `GL_NEVER` • 1: `GL_ALWAYS` • 2: `GL_EQUAL` • 3: `GL_NOTEQUAL` • 4: `GL_LESS` • 5: `GL_LEQUAL` • 6: `GL_GREATER` • 7: `GL_GEQUAL`
	`0x126`, bits [25:24]	Corresponds to the ***func*** argument: • 0: `GL_NEVER` • 1: `GL_ALWAYS` • 2: `GL_GREATER` or `GL_GEQUAL` • 3: Other
`glDepthMask`	`0x107`, bit [12:12]	Corresponds to the ***flag*** argument: • 0: `GL_FALSE` • 1: `GL_TRUE`

When changing these settings, you may also need to change the Framebuffer Access Control Setting Registers (see section 5.8.28) in the same way.

5.8.31 Logical Operation and Blend Setting Registers

Logical operations and blending share setting registers. The following table shows register settings specific to logical operations and blending.

Table 5-53 Logical Operation and Blend Setting Registers

Setting Function	Setting Register	Setting Value

`glEnable/glDisable (GL_COLOR_LOGIC_OP);`
 	
`glEnable/glDisable (GL_BLEND);`
	`0x100`, bit [8:8]	• 0: Enable logical operations
		• 1: Enable blending
		You cannot enable both logical operations and blending. Logical operations are given priority when both are enabled by the **`glEnable`** function. This is set equal to 1 when both are disabled.

`glBlendFunc`
 	
`glBlendFuncSeparate`
	`0x101`, bits [19:16]	When blending is disabled, this is set equal to 1. When blending is enabled, the following values are set by the ***sfactor*** or ***srcRGB*** argument.
		• 0: `GL_ZERO`
		• 1: `GL_ONE`
		• 2: `GL_SRC_COLOR`
		• 3: `GL_ONE_MINUS_SRC_COLOR`
		• 4: `GL_DST_COLOR`
		• 5: `GL_ONE_MINUS_DST_COLOR`
		• 6: `GL_SRC_ALPHA2`
		• 7: `GL_ONE_MINUS_SRC_ALPHA`
		• 8: `GL_DST_ALPHA`
		• 9: `GL_ONE_MINUS_DST_ALPHA`
		• 10: `GL_CONSTANT_COLOR`
		• 11: `GL_ONE_MINUS_CONSTANT_COLOR`
		• 12: `GL_CONSTANT_ALPHA`
		• 13: `GL_ONE_MINUS_CONSTANT_ALPHA`
		• 14: `GL_SRC_ALPHA_SATURATE`

	`0x101`, bits [23:20]	When blending is disabled, this is set equal to 0. When blending is enabled, the ***dfactor*** or ***dstRGB*** argument sets a value in the same way as bits [19:16] of `0x101`.
	`0x101`, bits [27:24]	When blending is disabled, this is set equal to 1. When blending is enabled, the ***sfactor*** or ***srcAlpha*** argument sets a value in the same way as bits [19:16] of `0x101`.
	`0x101`, bits [31:28]	When blending is disabled, this is set equal to 0. When blending is enabled, the ***dfactor*** or ***dstAlpha*** argument sets a value in the same way as bits [19:16] of `0x101`.

<table>
<tr><th>Setting Function</th><th>Setting Register</th><th>Setting Value</th></tr>
<tr><td rowspan="2"><code>glBlendEquation</code>

<code>glBlendEquationSeparate</code></td><td><code>0x101</code>, bits [2:0]</td><td>When blending is disabled, this is set equal to 0. When blending is enabled, the following values are set by the mode and modeRGB arguments.
• 0: <code>GL_FUNC_ADD</code>
• 1: <code>GL_FUNC_SUBTRACT</code>
• 2: <code>GL_FUNC_REVERSE_SUBTRACT</code>
• 3: <code>GL_MIN</code>
• 4: <code>GL_MAX</code></td></tr>
<tr><td><code>0x101</code>, bits [10:8]</td><td>When blending is disabled, this is set equal to 0. When blending is enabled, the mode or modeAlpha argument sets a value in the same way as bits [2:0] of <code>0x101</code>.</td></tr>
<tr><td rowspan="4"><code>glBlendColor</code></td><td><code>0x103</code>, bits [7:0]</td><td>The value set for the red argument is clamped between 0 and 1 and then the floating-point number is mapped to an integer between 0 and 255. For more details on this conversion, see section 5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer.</td></tr>
<tr><td><code>0x103</code>, bits [15:8]</td><td>The green argument sets a value in the same way as bits [7:0] of <code>0x103</code>.</td></tr>
<tr><td><code>0x103</code>, bits [23:16]</td><td>The blue argument sets a value in the same way as bits [7:0] of <code>0x103</code>.</td></tr>
<tr><td><code>0x103</code>, bits [31:24]</td><td>The alpha argument sets a value in the same way as bits [7:0] of <code>0x103</code>.</td></tr>
<tr><td><code>glLogicOp</code></td><td><code>0x102</code>, bits [3:0]</td><td>Corresponds to the opcode argument.
• 0: <code>GL_CLEAR</code>
• 1: <code>GL_AND</code>
• 2: <code>GL_AND_REVERSE</code>
• 3: <code>GL_COPY</code>
• 4: <code>GL_SET</code>
• 5: <code>GL_COPY_INVERTED</code>
• 6: <code>GL_NOOP</code>
• 7: <code>GL_INVERT</code>
• 8: <code>GL_NAND</code>
• 9: <code>GL_OR</code>
• 10: <code>GL_NOR</code>
• 11: <code>GL_XOR</code>
• 12: <code>GL_EQUIV</code>
• 13: <code>GL_AND_INVERTED</code>
• 14: <code>GL_OR_REVERSE</code>
• 15: <code>GL_OR_INVERTED</code></td></tr>
</table>

When changing these settings, you may also need to change the Framebuffer Access Control Setting Registers (see section 5.8.28) in the same way. Attempts to set register `0x101` are ignored when logical operations are enabled.

5.8.32 Early Depth Test Setting Registers

The following table shows register settings specific to early depth tests.

Table 5-54 Early Depth Test Setting Registers

Setting Function	Setting Register	Setting Value
`glEnable/glDisable (GL_EARLY_DEPTH_TEST_DMP);`	`0x62`, bit [0:0]	• 0: Disable early depth tests • 1: Enable early depth tests
	`0x118`, bit [0:0]	• 0: Disable early depth tests • 1: Enable early depth tests
`glEarlyDepthFuncDMP`	`0x61`, bits [1:0]	Corresponds to the ***func*** argument: • 0: `GL_GEQUAL` • 1: `GL_GREATER` • 2: `GL_LEQUAL` • 3: `GL_LESS`
`glClearEarlyDepthDMP`	`0x6a`, bits [23:0]	Sets the value of the ***depth*** argument unchanged.
`glClear`	`0x63`, bit [0:0]	Set when `GL_EARLY_DEPTH_BUFFER_BIT_DMP` is cleared.

When changing these settings, you may also need to change the Depth Test Setting Registers (see section 5.8.30) and Framebuffer Access Control Setting Registers (see section 5.8.28) in the same way.

5.8.33 Stencil Test Setting Registers

The following table shows register settings specific to stencil tests.

Table 5-55 Stencil Test Setting Registers

Setting Function	Setting Register	Setting Value
`glEnable/glDisable (GL_STENCIL_TEST);`	`0x105`, bit [0:0]	• 0: Disable stencil tests • 1: Enable stencil tests
`glStencilMask`	`0x105`, bits [15:8]	Sets the least-significant 8 bits of the ***mask*** argument.
`glStencilFunc`	`0x105`, bits [6:4]	Corresponds to the ***func*** argument: • 0: `GL_NEVER` • 1: `GL_ALWAYS` • 2: `GL_EQUAL` • 3: `GL_NOTEQUAL` • 4: `GL_LESS` • 5: `GL_LEQUAL` • 6: `GL_GREATER` • 7: `GL_GEQUAL`
	`0x105`, bits [23:16]	Sets the value of the ***ref*** argument unchanged.

Setting Function	Setting Register	Setting Value
	`0x105`, bits [31:24]	Sets the value of the ***mask*** argument unchanged.
`glStencilOp`	`0x106`, bits [2:0]	Corresponds to the ***fail*** argument: • 0: `GL_KEEP` • 1: `GL_ZERO` • 2: `GL_REPLACE` • 3: `GL_INCR` • 4: `GL_DECR` • 5: `GL_INVERT` • 6: `GL_INCR_WRAP` • 7: `GL_DECR_WRAP`
	`0x106`, bits [6:4]	The ***zfail*** argument sets a value in the same way as bits [2:0] of `0x106`.
	`0x106`, bits [10:8]	The ***zpass*** argument sets a value in the same way as bits [2:0] of `0x106`.

When changing these settings, you may also need to change the Framebuffer Access Control Setting Registers (see section 5.8.28) in the same way.

5.8.34 Culling Setting Registers

The following table shows register settings specific to culling.

Table 5-56 Culling Setting Registers

Setting Function	Setting Register	Setting Value

`glEnable/glDisable (GL_CULL_FACE);`
 	
`glCullFace`
 	
`glFrontFace`
	`0x40`, bits [1:0]	When culling is disabled, a value of 0 is set.
		When culling is enabled, a value of 2 is set in either of the following cases and a value of 1 is set otherwise.
		• The **`glCullFace`** function is `GL_FRONT` and the **`glFrontFace`** function is `GL_CW`
		• The **`glCullFace`** function is `GL_BACK` and the **`glFrontFace`** function is `GL_CCW`

5.8.35 Scissoring Setting Registers

The following table shows register settings specific to scissoring.

Table 5-57 Scissoring Setting Registers

Setting Function	Setting Register	Setting Value
`glEnable/glDisable (GL_SCISSOR_TEST);`	`0x65`, bits [1:0]	• 0: Disable scissoring • 3: Enable scissoring

Setting Function	Setting Register	Setting Value
`glScissor`	`0x66`, bits [9:0]	When scissoring is disabled, a value of 0 is set. When scissoring is enabled, the value of the ***x*** argument is set. When ***x*** is greater than or equal to the current color buffer width, however, a value that is one less than the color buffer width is set. When ***x*** is negative, a value of 0 is set.
	`0x66`, bits [25:16]	When scissoring is disabled, a value of 0 is set. When scissoring is enabled, the value of the ***y*** argument is set. When ***y*** is greater than or equal to the current color buffer height, however, a value that is one less than the color buffer height is set. When ***y*** is negative, a value of 0 is set.
	`0x67`, bits [9:0]	When scissoring is disabled, one less than the current color buffer width is set. When scissoring is enabled, $(\boldsymbol{x} + \boldsymbol{width} - 1)$ is set. When that value is greater than or equal to the current color buffer width, however, a value that is one less than the color buffer width is set. When $(\boldsymbol{x} + \boldsymbol{width} - 1)$ is negative, a value of 0 is set.
	`0x67`, bits [25:16]	When scissoring is disabled, one less than the current color buffer height is set. When scissoring is enabled, $(\boldsymbol{y} + \boldsymbol{height} - 1)$ is set. When that value is greater than or equal to the current color buffer height, however, a value that is one less than the color buffer height is set. When $(\boldsymbol{y} + \boldsymbol{height} - 1)$ is negative, a value of 0 is set.

5.8.36 Color Mask Setting Registers

The following table shows register settings specific to color masks.

Table 5-58 Color Mask Setting Registers

Function	Register	Values
`glColorMask`	`0x107`, bit [8:8]	Corresponds to the ***red*** argument: • 0: `GL_FALSE` • 1: `GL_TRUE`
	`0x107`, bit [9:9]	Corresponds to the ***green*** argument: • 0: `GL_FALSE` • 1: `GL_TRUE`
	`0x107`, bit [10:10]	Corresponds to the ***blue*** argument: • 0: `GL_FALSE` • 1: `GL_TRUE`
	`0x107`, bit [11:11]	Corresponds to the ***alpha*** argument: • 0: `GL_FALSE` • 1: `GL_TRUE`

When changing these settings, you may also need to change the Framebuffer Access Control Setting Registers (see section 5.8.28) in the same way.

5.8.37 Block Format Setting Registers

The following table shows register settings specific to the block format for rendering.

Table 5-59 Block Format Setting Registers

Setting Function	Setting Register	Setting Value
`glRenderBlockModeDMP`	`0x11b`, bit [0:0]	• 0: `GL_RENDER_BLOCK8_MODE_DMP` • 1: `GL_RENDER_BLOCK32_MODE_DMP`

5.8.38 Settings Registers Specific to the Rendering API

The rendering functions, **`glDrawElements`** and **`glDrawArrays`**, validate every state and thus generate register-setting commands related to each state. In addition to generating commands during validation, the rendering functions set registers required for rendering itself. The following sections explain these register settings.

5.8.38.1 With the Vertex Buffer in Use

This section describes the registers set by the rendering API when the vertex buffer is in use. All commands must be set before the rendering kick command unless you have some reason to set them in a different order.

Table 5-60 Register Settings Related to the Rendering API (if the Vertex Buffer Is in Use)

Setting	Setting Register	Setting Value
Rendering mode	`0x25e`, bits [9:8]	Set to 1 if the ***mode*** argument to the **`glDrawElements`** and/or **`glDrawArrays`** functions is `GL_TRIANGLE_STRIP`, to 2 if it is `GL_TRIANGLE_FAN`, and to 3 if it is `GL_GEOMETRY_PRIMITIVE_DMP`. Set to 0 if the ***mode*** argument to **`glDrawArrays`** is `GL_TRIANGLES`. Set to 3 if the ***mode*** argument to **`glDrawElements`** is `GL_TRIANGLES`. This does not need to be set per every rendering operation. It need only be reset when the setting has changed.
	`0x229`, bit [8:8]	Set to 1 when both the **`glDrawElements`** function is in use and the ***mode*** argument is `GL_TRIANGLES`. Cleared to 0 otherwise. This does not need to be set per every rendering operation. It need only be reset when the setting has changed.
	`0x253`, bit [8:8]	Set to 1 when both the **`glDrawElements`** function is in use and the ***mode*** argument is `GL_TRIANGLES`. Cleared to 0 otherwise. This does not need to be set per every rendering operation. It need only be reset when the setting has changed.

Setting	Setting Register	Setting Value
Rendering function indicator	`0x253`, bit [0:0]	Cleared to 0 when the **`glDrawElements`** function is the rendering function and set to 1 when the **`glDrawArrays`** function is the rendering function. This bit is cleared to 0 when **`nngxInitialize`** is called, so this is only set to 1 before a rendering kick when **`glDrawArrays`** is used, and immediately after the kick it is cleared to 0. When this bit is set to 1, register settings outside of register ranges `0x200` through `0x254` and `0x280` through `0x2df` are sometimes not properly executed. Since a command to register `0x111` is required for each rendering kick, this must be set to 1 and then reset to 0 per each **`glDrawArrays`** rendering kick.
Vertex index address	`0x227`, bits [27:0]	Specifies the address offset of the vertex index array. This is the offset from the common vertex array base address set by bits [28:1] of register `0x200`. This register's value is configured so that when it is added to the product of 16 and the value of bits [28:1] of register `0x200`, it is equal to the sum of the vertex buffer address allocated by the **`glBufferData`** function and the ***indices*** argument to the **`glDrawElements`** function. When **`glDrawArrays`** is in use, `0x10` is written here if either of the following conditions are met. If bits [31:0] of register `0x228` have a value larger than `0x10`, the condition that must be met is: ((bits [31:0] of `0x228 – 0x10)×2 +` (bits [28:1] of `0x200≪4)) &0xfff ≥ 0xff0` If bits [31:0] of register `0x228` have a value of `0x10` or smaller, the condition that must be met is: (bits [28:1] of `0x200≪4)&0xfff ≥ 0xff0` A value of 0 is written here in all other cases. This does not need to be set per every rendering operation. It need only be reset when the setting has changed.
Vertex index type	`0x227`, bit [31:31]	Set to 1 when the ***type*** argument to the **`glDrawElements`** function is `GL_UNSIGNED_SHORT` and 0 when the same argument is `GL_UNSIGNED_BYTE`. Set to 1 when the **`glDrawArrays`** is in use. This does not need to be set per every rendering operation. It need only be reset when the setting has changed.
Vertex count	`0x228`, bits [31:0]	Sets the number of vertices to render. This does not need to be set per every rendering operation. It need only be reset when the setting has changed.
Starting vertex offset	`0x22a`, bits [31:0]	Sets the value of the ***first*** argument for the **`glDrawArrays`** function. This does not need to be set per every rendering operation. It need only be reset when the setting has changed.

Setting	Setting Register	Setting Value
Vertex information reset	`0x25f`, bit [0:0]	Writing a value of 1 to this bit resets the information that indicates each vertex's index (0, 1, or 2) in the triangles that it forms. A reset is required for each rendering kick command when the **`glDrawElements`** function is called in `GL_TRIANGLES` mode. A reset is not required for consecutive calls–except for the first–to the **`glDrawArrays`** function in `GL_TRIANGLES` mode if and only if the **`glDrawElements`** function is not called and the number of rendered vertices is a multiple of 3. However, a reset is required after rendering in some other mode, after rendering with the **`glDrawElements`** function, and when the **`glDrawArrays`** function is called for the first time after the **`nngxInitialize`** function. In `GL_TRIANGLE_STRIP` or `GL_TRIANGLE_FAN` mode, a reset is required per each rendering kick command.
Rendering kick command	`0x22e`	Writes a value of 1 to an arbitrary bit when rendering starts with the **`glDrawArrays`** function.
	`0x22f`	Writes a value of 1 to an arbitrary bit when rendering starts with the **`glDrawElements`** function.
Post-vertex cache clear	`0x231`	Writes a value of 1 to an arbitrary bit immediately after a rendering kick command. Must be set per each rendering kick command.
Framebuffer cache flush	`0x111`, bit [0:0]	Writes a value of 1 immediately after a rendering kick command. Required per each rendering kick command.
Texture enabling	`0x80`, bits [2:0]	Set to 1 for the texture to enable immediately before a rendering kick command, then set to 0 immediately after the rendering kick command. Setting to 0 helps reduce power consumption, so this process ensures that the value is set to 0 at all times when not rendering. Leaving the value always set to 1 for an enabled texture does not cause any operation problems. See section 5.8.20.2 Setting the Texture Sampler Type for details on each bit.
Other registers	`0x245`, bit [0:0]	Set to 1 when the **`nngxInitialize`** function is called. Rendering is not performed properly when this is set to 1. When this is 0, settings commands to registers `0x2b0`–`0x2df` are not applied correctly. Both the **`glDrawElements`** and **`glDrawArrays`** functions generate commands to clear this bit to 0 immediately before the rendering kick command and then return it to 1 immediately afterward. If no settings commands to registers `0x2b0`–`0x2df` are used after the rendering kick command, it causes no problems to leave this bit cleared to 0.
	`0x25e`, bits [31:24]	These bits require two commands to clear them to 0 immediately after rendering kick commands. This is required per each rendering kick command. These commands are dummy commands and the exact value they set has no meaning.

Setting	Setting Register	Setting Value
	`0x2ba`, bits [31:16]	Write these bits to a value of `0x7fff` after a rendering kick command. Running this command just after rendering completes helps to reduce power consumption. Not setting these bits does not cause any operation problems. Set `0xc` with a byte enable, so as to have no effect on bits [15:0]. Bit [0:0] of register `0x245` must be set to 1 before this command.
	`0x28a`, bits [31:16]	Write these bits to a value of `0x7fff` after a rendering kick command. Running this command just after rendering completes helps to reduce power consumption. Not setting these bits does not cause any operation problems. Set `0xc` with a byte enable, so as to have no effect on bits [15:0]. When the pipeline is set not to use geometry shaders (bit [0:0] of register `0x244` is 0 and bits [1:0] of register `0x229` are 0), the setting of bits [31:16] of register `0x2ba` includes this command's setting, making this command unnecessary.

Note: Cautions About Command Dependencies:

Bits [31:16] of register `0x2ba` must be set only after bit [0:0] of register `0x245` is set. When bit [0:0] of register `0x253` has been set to 1, register settings outside of register ranges `0x200` through `0x254` and `0x280` through `0x2df` are sometimes not properly executed. Set the registers in these ranges only while bit [0:0] of register `0x253` has been set to 0. However, this restriction does not apply to the dummy commands for bits [31:24] of register `0x25e`.
There are several other commands that must always be set immediately after a rendering kick command, but these other commands have no ordering dependencies.

5.8.38.2 Without the Vertex Buffer in Use

When the vertex buffer is not used, the vertex data is itself input through registers. The following table shows how register settings change when the vertex buffer is not used. Vertex attribute data commands are handled the same way as rendering kick commands. All commands must be set before the vertex attribute data command unless you have some reason to use a different order.

Table 5-61 Register Settings Related to the Rendering API (when the Vertex Buffer Is Not in Use)

Setting	Setting Register	Setting Value
Rendering mode	`0x25e`, bits [9:8]	Set to 0 if the ***mode*** argument to the **`glDrawElements`** or **`glDrawArrays`** function is `GL_TRIANGLES`, to 1 if it is `GL_TRIANGLE_STRIP`, to 2 if it is `GL_TRIANGLE_FAN`, or to 3 if it is `GL_GEOMETRY_PRIMITIVE_DMP`. This does not need to be set per every rendering operation. It need only be reset when the setting has changed.

Setting	Setting Register	Setting Value
	0x229, bit [8:8]	Set to 0. This does not need to be set per every rendering operation.
	0x253, bit [8:8]	Set to 0. This does not need to be set per every rendering operation.
Rendering function indicator	0x253, bit [0:0]	Whether the function called was **glDrawElements** or **glDrawArrays**, this bit is set to 1 before a vertex attribute data command and then cleared to 0 after the command. This bit is cleared to 0 when **nngxInitialize** is called, so this is set to 1 before a vertex attribute data command, and immediately after the vertex attribute data command it is cleared to 0. When this bit is set to 1, register settings outside of register ranges 0x200 through 0x254 and 0x280 through 0x2df are sometimes not properly executed. Since a command to register 0x111 is required for each rendering command, this must be set to 1 and then reset to 0 per each rendering command.
Vertex index address	0x227, bits [27:0]	This setting is ignored.
Vertex index type	0x227, bit [31:31]	This setting is ignored.
Vertex count	0x228, bits [31:0]	This setting is ignored. The number of vertices to process is determined by the number of vertex attribute data items.
Starting vertex offset	0x22a, bits [31:0]	This setting is ignored.
Vertex data reset	0x25f, bit [0:0]	Same as when the vertex buffer is used.
Rendering kick command	0x22e	This setting is prohibited.
	0x22f	This setting is prohibited.
Enable slave input	0x232, bits [3:0]	Set to 0xf.
Vertex attribute data	0x233, 0x234, and 0x235,bits [31:0]	Sets vertex attribute data. This command is set after 0xf is written to bits [3:0] of 0x232. Data for each single vertex is stored in order one attribute at a time. All vertex attribute data is stored regardless of whether vertex arrays are used. A single attribute packs four 24-bit floating-point numbers into three 32-bit data units, which are stored in 0x233, 0x234, and 0x235, respectively. A single attribute is input by writing the data in 0x233, 0x234, and 0x235 one at a time (in that order). The 24-bit floating-point numbers are packed as described in section 5.8.2.3 How to Set the Input Mode for 24-Bit Floating-Point Numbers.
Post-vertex cache clear	0x231	Same as when the vertex buffer is used.
Framebuffer cache flush	0x111, bit [0:0]	Same as when the vertex buffer is used.

Setting	Setting Register	Setting Value
Texture enabling	`0x80`, bits [2:0]	Same as when the vertex buffer is used.
Other registers	`0x245`, bit [0:0]	Same as when the vertex buffer is used.
	`0x2ba`, bits [31:16]	Same as when the vertex buffer is used.
	`0x28a`, bits [31:16]	Same as when the vertex buffer is used.

When not using the vertex buffer, you do not need to set the registers described in section 5.8.14 Registers for Vertex Attribute Array Settings. Command-ordering dependencies are the same as when using the vertex buffer.

5.8.39 Settings Registers Specific to the Geometry Shader

This section describes settings registers when the geometry shader is in use.

5.8.39.1 Overview

There are multiple vertex shader processors installed on PICA for vertex processing. One of these vertex shader processors is used as the geometry shader processor when a geometry shader is in use. This is called a *shared processor*. When a geometry shader is not in use, the shared processor runs as a vertex shader processor and floating-point registers, Boolean registers, and other resources are set as vertex shader values. The vertex shader values must be changed to geometry shader settings when the geometry shader switches from being unused to used. Similarly, geometry shader values must be changed to vertex shader settings when the geometry shader switches from being used to unused.

Registers `0x2b0`–`0x2df` are the settings registers used for vertex shader processors. Setting one of these registers sets it for all of the vertex shader processors. These settings also apply to the shared processor except when bit [0:0] of register `0x244` is set equal to 1 (when the same bit is 0 and bits [1:0] of `0x229` are 0, settings for the vertex shader processors are also applied to the shared processor). Registers `0x280`–`0x2af` are used to apply the same settings as registers `0x2b0`–`0x2df` to the shared processor only.

When the geometry shader is in use, registers `0x280`–`0x2af` are configured to be geometry shader-specific. When the geometry shader is not in use, registers `0x280`–`0x2af` must have the same settings as registers `0x2b0`–`0x2af`. (You can also set bit [0:0] of register `0x244` equal to 0 and bits [1:0] of `0x229` equal to 0, applying vertex shader processor settings to the shared processor, before you re-set registers `0x2b0`–`0x2df`.)

To use the geometry shader, you need to set these register settings related to the shared processor as well as other register settings related to input, output, and so on.

5.8.39.2 Geometry Shader Floating-Point Registers

The geometry shader's floating-point registers first set bits [7:0] of register `0x290` equal to a floating-point register index and then write data to any registers between registers `0x291` and `0x298`. Depending on whether a value of 1 or 0 is written to bit [31:31] of `0x290`, the input mode is set to accept either 32-bit or 24-bit floating-point numbers, respectively. This is configured as described in section 5.8.2 Vertex Shader Floating-Point Registers.

5.8.39.3 Geometry Shader Boolean Registers

Bits [15:0] of register `0x280` correspond to the geometry shader's Boolean registers. These are set as described in section 5.8.3 Vertex Shader Boolean Registers.

5.8.39.4 Geometry Shader Integer Registers

Registers `0x281`, `0x282`, `0x283`, and `0x284` correspond to `i0`, `i1`, `i2`, and `i3`, respectively. These are set as described in section 5.8.4 Vertex Shader Integer Registers.

5.8.39.5 Geometry Shader Starting Address Setting Registers

Bits [15:0] of register `0x28a` set the geometry shader's starting address. These are set as described in section 5.8.5 Vertex Shader Starting Address Setting Registers.

5.8.39.6 Registers That Set the Number of Input Vertex Attributes

Bits [3:0] of register `0x289` set a value that is one less than the number of input vertex attributes to the geometry shader. The number of attributes input to the geometry shader is the same as the number of attributes output by the vertex shader (including `generic` attributes).

5.8.39.7 Registers That Set the Number of Output Registers Used by the Geometry Shader

The registers described in section 5.8.7 Registers That Set the Number of Output Registers Used by the Vertex Shader are set differently when the geometry shader is in use. Bits [2:0] of register `0x4f` set the number of output registers for the geometry shader. Bits [3:0] of register `0x25e` set a value that is one less than the number of output registers used by the geometry shader.

5.8.39.8 Register That Sets the Geometry Shader Output Register Mask

A bit mask is used to set the output registers written by the geometry shader. Bits [15:0] of register `0x28d` each correspond to one of the 16 output registers. These are set as described in section 5.8.8 Registers That Set the Vertex Shader Output Mask.

5.8.39.9 Registers That Set Geometry Shader Output Attributes

When a geometry shader is in use, the registers described in section 5.8.9 Registers That Set Vertex Shader Output Attributes—`0x50`, `0x51`, `0x52`, `0x53`, `0x54`, `0x55`, `0x56`, and `0x64`—set the attributes of vertices output by the geometry shader instead of the vertex shader.

The `#pragma output_map` settings defined in the geometry shader determine the geometry shader's output attributes. This information is generated in the map file that is created by the shader assembly linker (for details on the map file, see the *Vertex Shader Reference Manual*). Several geometry shaders define `generic` attributes as `output_map` attributes. The `#pragma output_map` settings that are only defined in the linked vertex shaders are applied to the attributes defined as `generic` attributes (excluding `generic` attributes defined by the vertex shader).

5.8.39.10 Clock Control Setting Registers for Geometry Shader Output Attributes

When a geometry shader is in use, register `0x6f` (described in section 5.8.10 Clock Control Setting Registers for Vertex Shader Output Attributes) sets the attributes of vertices output by the geometry shader instead of those output by the vertex shader.

5.8.39.11 Geometry Shader Program Code Setting Registers

The following table shows registers that are used to load swizzle pattern data and program code executed by the geometry shader.

Table 5-62 Geometry Shader Program Code and Swizzle Pattern Data Settings Registers

Setting Register	Description
`0x29b`, bits [11:0]	Sets the load address for program code.
`0x29c`–`0x2a3`, bits [31:0]	Sets program code data.
`0x28f`	Notification that a program update has completed.
`0x2a5`, bits [11:0]	Sets the load address for the swizzle pattern.
`0x2a6`–`0x2ad`, bits [31:0]	Sets swizzle pattern data.

Subtracting `0x30` from the addresses of the registers described in section 5.8.11 Vertex Shader Program Code Setting Registers gives the geometry shader registers, which are set the same way.

5.8.39.12 Registers That Map Vertex Attributes to Geometry Shader Input Registers

These set the input register map for vertex attributes input to the geometry shader as described in section 5.8.12 Registers That Map Vertex Attributes to Input Registers. Fixed values are set when a reserved geometry shader is used. Set register `0x28b` equal to `0x76543210` and register `0x28c` equal to `0xfedcba98`.

5.8.39.13 Miscellaneous Registers

The following registers must also be set when a geometry shader is in use.

Table 5-63 Miscellaneous Settings Registers When the Geometry Shader Is In Use

Setting Register	Description
`0x229`, bits [1:0]	Set to 2 when a geometry shader is in use and 0 when it is not. When you set this register, dummy commands are required both before and after the setting command. Use invalid commands whose byte enable bits are 0 as the dummy commands. A command that sets this register must be immediately preceded by 10 dummy commands that set register `0x251` and 30 dummy commands that set register `0x200`, and immediately followed by 30 dummy commands that set register `0x200`.
`0x229`, bit [31:31]	Set to 1 when reserved geometry shader subdivision (Loop or Catmull-Clark) is used. Set to 0 when any other geometry shader is used or when a geometry shader is not used.
`0x252`, bits [31:0]	Set to `0x00000001` when reserved geometry shader subdivision (Loop or Catmull-Clark) is used. Set to `0x01004302` when particle systems are used.

Setting Register	Description
	Set to 0x00000000 when any other geometry shader is used or a geometry shader is not used.
0x289, bits [31:24]	Set to 0x08 when a geometry shader is used and 0xa0 when a geometry shader is not used.
0x289, bits [15:8]	Set to 1 when reserved geometry shader subdivision (Loop or Catmull-Clark) is used. Set to 0 when any other geometry shader or no geometry shader is used.
0x254, bits [4:0]	Set to 3 when a reserved geometry shader is used for Catmull-Clark subdivision and 2 when a reserved geometry shader is used for Loop subdivision. Otherwise, this setting is ignored.

5.8.40 Settings Registers When Reserved Geometry Shaders Are Used

This section lists settings registers for the registers described in section 5.8.39 Settings Registers Specific to the Geometry Shader when each reserved geometry shader is used. It also shows which register is assigned to the uniform of each reserved geometry shader.

5.8.40.1 Point Shader

The following table shows the register values that should be set when the point shader is used.

Table 5-64 Register Setting Values When the Point Shader Is Used

Setting Register	Description
0x4f, bits [2:0]	Set equal to the number of output registers defined by #pragma output_map for the linked vertex shader, not including generic attributes.
0x50–0x56	Starting at 0x50, which must be set equal to 0x03020100, these registers are filled with the attributes defined by #pragma output_map for the linked vertex shader. The point size is output as a generic attribute but it does not affect this register. Starting at 0x51, registers are filled with defined attributes in ascending order of output register indices. For example, because a point sprite's vertex coordinates should be followed by texture coordinates, register 0x51 would be set equal to 0x1f1f0d0c for a definition of #pragma output_map(texture0, o2.xy). Each byte of unused attributes is filled in using 0x1f.
0x64	Set in accordance with the attributes defined by #pragma output_map for the linked vertex shader.
0x6f	Set in accordance with the attributes defined by #pragma output_map for the linked vertex shader.
0x229, bit [31:31]	Set equal to 0.
0x242, bits [3:0]	Set equal to one less than the number of input vertex attributes to the linked vertex shader.
0x24a, bits [3:0]	Set equal to one less than the number of output registers defined by #pragma output_map for the linked vertex shader. This also includes generic attributes.
0x251, bits [3:0]	Set equal to one less than the number of output registers defined by #pragma output_map for the linked vertex shader. This also includes generic attributes.

Setting Register	Description
`0x252`	Set equal to 0.
`0x254`, bits [4:0]	No required settings.
`0x25e`, bits [3:0]	Set equal to one less than the number of output registers defined by `#pragma output_map` for the linked vertex shader. This does not include `generic` attributes.
`0x280`, bits [15:0]	Set equal to 0.
`0x281`, bits [23:0]	Set equal to 0.
`0x282`, bits [23:0]	Set equal to 0.
`0x283`, bits [23:0]	Set equal to 0.
`0x284`, bits [23:0]	Set equal to 0.
`0x289`, bits [3:0]	Set equal to one less than the number of output registers defined by `#pragma output_map` for the linked vertex shader. This also includes `generic` attributes.
`0x289`, bits [15:8]	Set equal to 0.
`0x289`, bits [31:24]	Set equal to 8.
`0x28d`, bits [15:0]	Set equal to `((1<<N)-1)`, where `N` is the number of output registers defined by `#pragma output_map` for the linked vertex shader. This does not include `generic` attributes.
`0x290`–`0x923`	Write the values in each of the following combinations to registers `0x290`, `0x291`, `0x292`, and `0x293`, respectively; these are used to set floating-point constants. • {`0x0000004c`, `0x40800040`, `0x00003f00`, `0x00000000`} • {`0x0000004d`, `0x3d00003e`, `0x00000000`, `0x003c0000`}

The registers assigned to each uniform are shown in the table below.

Table 5-65: Point Shader Uniforms and Their Corresponding Registers

Uniform	Bound Register
`dmp_Point.viewport`	`c67.xy`
`dmp_Point.distanceAttenuation`	`b0`

5.8.40.2 Line Shader

The following table shows the register values that should be set when the line shader is used.

Table 5-66 Register Setting Values When Line Shading Is Used

Setting Register	Description
`0x4f`, bits [2:0]	Set equal to the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x50`–`0x56`	Starting at `0x50`, which must be set equal to `0x03020100`, these registers are filled with

Setting Register	Description
	the attributes defined by `#pragma output_map` for the linked vertex shader. Starting at `0x51`, registers are filled with defined attributes in ascending order of output register indices. Each byte of unused attributes is filled in using `0x1f`.
`0x64`	Set in accordance with the attributes defined by `#pragma output_map` for the linked vertex shader.
`0x6f`	Set in accordance with the attributes defined by `#pragma output_map` for the linked vertex shader.
`0x229`, bit [31:31]	Set equal to 0.
`0x242`, bits [3:0]	Set equal to one less than the number of input vertex attributes to the linked vertex shader.
`0x24a`, bits [3:0]	Set equal to one less than the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x251`, bits [3:0]	Set equal to one less than the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x252`	Set equal to 0.
`0x254`, bits [4:0]	No required settings.
`0x25e`, bits [3:0]	Set equal to one less than the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x280`, bits [15:0]	Set equal to `0x0000`. Bit [15:15] must be set for each draw operation.
`0x281`, bits [23:0]	Set equal to 0.
`0x282`, bits [23:0]	Set equal to 0.
`0x283`, bits [23:0]	Set equal to 0.
`0x284`, bits [23:0]	Set equal to 0.
`0x289`, bits [3:0]	Set equal to one less than the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x289`, bits [15:8]	Set equal to 0.
`0x289`, bits [31:24]	Set equal to 8.
`0x28d`, bits [15:0]	Set equal to `((1<<N)-1)`, where `N` is the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x290-0x293`	Write the values in each of the following combinations to registers `0x290`, `0x291`, `0x292`, and `0x293`, respectively; these are used to set floating-point constants. • {`0x0000004c`, `0x40800040`, `0x00003f00`, `0x00000000`} • {`0x0000004d`, `0x3d00003e`, `0x00000000`, `0x003c0000`} • {`0x0000004e`, `0x4300003d`, `0x00003e80`, `0x00420000`} • {`0x0000004f`, `0x3c60003c`, `0xc8003780`, `0x00390000`} • {`0x00000050`, `0x3d0c0039`, `0x80003700`, `0x003b8000`} • {`0x00000051`, `0x3cc0003c`, `0x70003a60`, `0x003c2800`}

Setting Register	Description
	• {0x00000052, 0x3d16003b, 0x0c003500, 0x003d8000} • {0x00000053, 0x3daaaa39, 0xc71c3c55, 0x55be2aaa} • {0x00000054, 0x3d871c3a, 0x425e3c55, 0x55be3c71} • {0x00000055, 0x3e200039, 0x00003b80, 0x00bdc000} • {0x00000056, 0x3d940039, 0x8fff3c04, 0x00be3600} • {0x00000057, 0x0000003f, 0x00004180, 0x00c0c000} • {0x00000058, 0x00000040, 0x00004230, 0x00c17000} • {0x00000059, 0x000000c0, 0xc000c350, 0x00428800} • {0x0000005a, 0x4effff41, 0x80004140, 0x00410000} • {0x0000005b, 0x3c80003b, 0x00003c80, 0x003e2000} • {0x0000005c, 0x00000000, 0x00003c00, 0x003d8000}

The registers assigned to each uniform are shown in the table below.

Table 5-67: Line Shader Uniforms and Their Corresponding Registers

Uniform	Bound Register
`dmp_Line.width`	`c67.xyzw`

5.8.40.3 Silhouette Shader

The following table shows the register values that should be set when the silhouette shader is used.

Table 5-68 Register Setting Values When the Silhouette Shader Is Used

Setting Register	Description
`0x4f`, bits [2:0]	Set equal to 2.
`0x50`–`0x56`	• Set register `0x50` equal to `0x03020100` • Set register `0x51` equal to `0x0b0a0908` • Set registers `0x52`–`0x56` equal to `0x1f1f1f1f`
`0x64`	Set equal to 3.
`0x6f`	Set equal to 0.
`0x229`, bit [31:31]	Set equal to 0.
`0x242`, bits [3:0]	Set equal to one less than the number of input vertex attributes to the linked vertex shader.
`0x24a`, bits [3:0]	Set equal to 2.
`0x251`, bits [3:0]	Set equal to 2.
`0x252`	Set equal to 0.
`0x254`, bits [4:0]	No required settings.

Setting Register	Description
0x25e, bits [3:0]	Set equal to 1.
0x280, bits [15:0]	Set equal to 0x0000. Bit [15:15] must be set for each draw operation.
0x281, bits [23:0]	Set equal to 0.
0x282, bits [23:0]	Set equal to 0.
0x283, bits [23:0]	Set equal to 0.
0x284, bits [23:0]	Set equal to 0.
0x289, bits [3:0]	Set equal to 2 because there are three output attributes for the vertex shader: vertex coordinates, color, and normals.
0x289, bits [15:8]	Set equal to 0.
0x289, bits [31:24]	Set equal to 8.
0x28d, bits [15:0]	Set equal to 3.
0x290-0x293	Write the values in each of the following combinations to registers 0x290, 0x291, 0x292, and 0x293, respectively; these are used to set floating-point constants. • {0x0000004c, 0x40800040, 0x00003f00, 0x00000000} • {0x0000004d, 0x3d00003e, 0x00000000, 0x003c0000} • {0x0000004e, 0x4effff41, 0x80004140, 0x00410000}

The registers assigned to each uniform are shown in the table below.

Table 5-69: Silhouette Shader Uniforms and Their Corresponding Registers

Uniform	Bound Register
dmp_Silhouette.width	c71.xy
dmp_Silhouette.openEdgeDepthBias	c71.z
dmp_Silhouette.color	c72.xyzw
dmp_Silhouette.openEdgeColor	c73.xyzw
dmp_Silhouette.openEdgeWidth	c74.xyzw
dmp_Silhouette.acceptEmptyTriangles	b0
dmp_Silhouette.scaleByW	b1
dmp_Silhouette.frontFaceCCW	b2
dmp_Silhouette.openEdgeWidthScaleByW	b3
dmp_Silhouette.openEdgeDepthBiasScaleByW	b4

5.8.40.4 Catmull-Clark Subdivision

The following table shows the register values that should be set when Catmull-Clark subdivision is used.

Table 5-70 Register Setting Values When Catmull-Clark Subdivision Is Used

Setting Register	Description
`0x4f`, bits [2:0]	Set equal to the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x50`–`0x56`	Starting at `0x50`, which must be set equal to `0x03020100`, these registers are filled with the attributes defined by `#pragma output_map` for the linked vertex shader. Starting at `0x51`, registers are filled with defined attributes in ascending order of output register indices. Each byte of unused attributes is filled in using `0x1f`.
`0x64`	Set in accordance with the attributes defined by `#pragma output_map` for the linked vertex shader.
`0x6f`	Set in accordance with the attributes defined by `#pragma output_map` for the linked vertex shader.
`0x229`, bit [31:31]	Set equal to 1.
`0x242`, bits [3:0]	Set equal to one less than the number of input vertex attributes to the linked vertex shader.
`0x24a`, bits [3:0]	Set equal to one less than the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x251`, bits [3:0]	Set equal to one less than the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x252`	Set equal to 1.
`0x254`, bits [4:0]	Set equal to 3.
`0x25e`, bits [3:0]	Set equal to one less than the number of output registers defined by `#pragma output_map` for the linked vertex shader.
`0x280`, bits [15:0]	Set equal to `0x0000`. Bit [15:15] must be set for each draw operation.
`0x281`, bits [23:0]	Set equal to 0.
`0x282`, bits [23:0]	• `0x020eff` for `DMP_subdivision0.obj` • `0x0212ff` for `DMP_subdivision1.obj` • `0x0216ff` for `DMP_subdivision2.obj` • `0x021aff` for `DMP_subdivision3.obj` • `0x021eff` for `DMP_subdivision4.obj` • `0x0222ff` for `DMP_subdivision5.obj` • `0x0226ff` for `DMP_subdivision6.obj`
`0x283`, bits [23:0]	• `0x0202ff` for `DMP_subdivision0.obj` • `0x0206ff` for `DMP_subdivision1.obj` • `0x020aff` for `DMP_subdivision2.obj` • `0x020eff` for `DMP_subdivision3.obj`

Setting Register	Description
	• 0x0212ff for DMP_subdivision4.obj • 0x0216ff for DMP_subdivision5.obj • 0x021aff for DMP_subdivision6.obj
0x284, bits [23:0]	Set equal to 0.
0x289, bits [3:0]	Set equal to one less than the number of output registers defined by #pragma output_map for the linked vertex shader.
0x289, bits [15:8]	Set equal to 1.
0x289, bits [31:24]	Set equal to 8.
0x28d, bits [15:0]	Set equal to ((1<<N)-1), where N is the number of output registers defined by #pragma output_map for the linked vertex shader.
0x290–0x293	Write the values in each of the following combinations to registers 0x290, 0x291, 0x292, and 0x293, respectively; these are used to set floating-point constants. • {0x0000004c, 0x40800040, 0x00003f00, 0x00000000} • {0x0000004d, 0x3d00003e, 0x00000000, 0x003c0000} • {0x0000004e, 0x4300003d, 0x00003e80, 0x00420000} • {0x0000004f, 0x3c60003c, 0xc8003780, 0x00390000} • {0x00000050, 0x3d0c0039, 0x80003700, 0x003b8000} • {0x00000051, 0x3cc0003c, 0x70003a60, 0x003c2800} • {0x00000052, 0x3d16003b, 0x0c003500, 0x003d8000} • {0x00000053, 0x3daaaa39, 0xc71c3c55, 0x55be2aaa} • {0x00000054, 0x3d871c3a, 0x425e3c55, 0x55be3c71} • {0x00000055, 0x3e200039, 0x00003b80, 0x00bdc000} • {0x00000056, 0x3d940039, 0x8fff3c04, 0x00be3600} • {0x00000057, 0x0000003f, 0x00004180, 0x00c0c000} • {0x00000058, 0x00000040, 0x00004230, 0x00c17000} • {0x00000059, 0x000000c0, 0xc000c350, 0x00428800} • {0x0000005a, 0x4effff41, 0x80004140, 0x00410000} • {0x0000005b, 0x3c80003b, 0x00003c80, 0x003e2000} • {0x0000005c, 0x00000000, 0x00003c00, 0x003d8000} Furthermore, • Set {0x0000004b, 0x41000040, 0x80004000, 0x003f0000} only for DMP_subdivision0.obj • Set {0x0000004b, 0x42000041, 0x80004100, 0x00400000} only for DMP_subdivision1.obj • Set {0x0000004b, 0x42800042, 0x20004180, 0x00408000} only for DMP_subdivision2.obj • Set {0x0000004b, 0x43000042, 0x80004200, 0x00410000} only for DMP_subdivision3.obj • Set {0x0000004b, 0x43400042, 0xe0004240, 0x00414000} only for DMP_subdivision4.obj • Set {0x0000004b, 0x43800043, 0x20004280, 0x00418000} only for DMP_subdivision5.obj • Set {0x0000004b, 0x43c00043, 0x500042c0, 0x0041c000} only for DMP_subdivision6.obj

Table 5-71: Catmull-Clark Subdivision Shader Uniforms and Their Corresponding Registers

Uniform	Bound Register
dmp_Subdivision.level	c74.x
dmp_Subdivision.fragmentLightingEnabled	b2

5.8.40.5 Loop Subdivision

The following table shows the register values that should be set when Loop subdivision is used.

Table 5-72 Register Setting Values When Loop Subdivision Is Used

Setting Register	Description
0x4f, bits [2:0]	Set equal to the number of output registers defined by #pragma output_map for the linked vertex shader, not including generic attributes.
0x50–0x56	Starting at 0x50, which must be set equal to 0x03020100, these registers are filled with the attributes defined by #pragma output_map for the linked vertex shader. Starting at 0x51, registers are filled with defined attributes in ascending order of output register indices. All generic attributes are ignored, and each byte of unused attributes is filled in using 0x1f.
0x64	Set in accordance with the attributes defined by #pragma output_map for the linked vertex shader.
0x6f	Set in accordance with the attributes defined by #pragma output_map for the linked vertex shader.
0x229, bit [31:31]	Set equal to 1.
0x242, bits [3:0]	Set equal to one less than the number of input vertex attributes to the linked vertex shader.
0x24a, bits [3:0]	Set equal to one less than the number of output registers defined by #pragma output_map for the linked vertex shader. This also includes generic attributes.
0x251, bits [3:0]	Set equal to one less than the number of output registers defined by #pragma output_map for the linked vertex shader. This also includes generic attributes.
0x252	Set equal to 1.
0x254, bits [4:0]	Set equal to 2.
0x25e, bits [3:0]	Set equal to one less than the number of output registers defined by #pragma output_map for the linked vertex shader. This does not include generic attributes.
0x280, bits [15:0]	Set equal to 0x0000. Bit [15:15] must be set for each draw operation.
0x281, bits [23:0]	Set equal to 0.
0x282, bits [23:0]	Set equal to 0.
0x283, bits [23:0]	Set equal to 0.

Setting Register	Description
0x284, bits [23:0]	Set equal to 0.
0x289, bits [3:0]	Set equal to one less than the number of output registers defined by #pragma output_map for the linked vertex shader. This also includes generic attributes.
0x289, bits [15:8]	Set equal to 1.
0x289, bits [31:24]	Set equal to 8.
0x28d, bits [15:0]	Set equal to ((1<<N)-1), where N is the number of output registers defined by #pragma output_map for the linked vertex shader. This does not include generic attributes.
0x290–0x293	Write the values in each of the following combinations to registers 0x290, 0x291, 0x292, and 0x293, respectively; these are used to set floating-point constants. • {0x00000057, 0x40800040, 0x00003f00, 0x00000000} • {0x00000058, 0x3d00003e, 0x00000000, 0x003c0000} • {0x00000059, 0x4300003d, 0x00003e80, 0x00420000} • {0x0000005a, 0x0000003b, 0x00003d80, 0x00390000} • {0x0000005b, 0x3c200038, 0x00003980, 0x003d3000} • {0x0000005c, 0x3c60003a, 0x80003b80, 0x003c8000} • {0x0000005d, 0x42200056, 0xffff3c20, 0x003ce000} • {0x0000005e, 0x3c98003d, 0x9c003c80, 0x003dc000} • {0x0000005f, 0x3de0003e, 0x10003d80, 0x003e4000}

Table 5-73: Loop Subdivision Shader Uniforms and Their Corresponding Registers

Uniform	Bound Register
dmp_Subdivision.level	c86.x
dmp_Subdivision.fragmentLightingEnabled	b0

5.8.40.6 Particle System

The following table shows the register values that should be set when the particle system shader is used.

Table 5-74 Register Setting Values When the Particle System Shader Is Used

Setting Register	Description
0x4f, bits [2:0]	Set equal to 3.
0x50–0x56	• Set register 0x50 equal to 0x03020100 • Set register 0x51 equal to 0x0b0a0908 • Set register 0x52 equal to 0x17160d0c when texture coordinate 2 is used or 0x1f1f0d0c otherwise • Set registers 0x53–0x56 equal to 0x1f1f1f1f
0x64	Set equal to 0x00000503 when texture coordinate 2 is used or 0x00000103 otherwise.

Setting Register	Description
0x6f	Set equal to 1.
0x229, bit [31:31]	Set equal to 0.
0x242, bits [3:0]	Set equal to one less than the number of input vertex attributes to the linked vertex shader.
0x24a, bits [3:0]	Set equal to 4.
0x251, bits [3:0]	Set equal to 4.
0x252	Set equal to 0x01004302.
0x254, bits [4:0]	No required settings.
0x25e, bits [3:0]	Set equal to 2.
0x280, bits [15:0]	Set equal to 0.
0x281, bits [23:0]	Set equal to 0.
0x282, bits [23:0]	Set equal to 0.
0x283, bits [23:0]	Set equal to 0.
0x284, bits [23:0]	Set equal to 0x0100fe.
0x289, bits [3:0]	Set equal to 4 because there are a total of five output attributes for the vertex shader: the vertex coordinates and the four bounding-box sizes for the control points.
0x289, bits [15:8]	Set equal to 1.
0x289, bits [31:24]	Set equal to 8.
0x28d, bits [15:0]	Set equal to 0x0007.
0x290–0x293	Write the values in each of the following combinations to registers 0x290, 0x291, 0x292, and 0x293, respectively; these are used to set floating-point constants. • {0x0000004c, 0x40800040, 0x00003f00, 0x00000000} • {0x0000004d, 0x3d00003e, 0x00000000, 0x003c0000} • {0x0000004e, 0x00000000, 0x00003f00, 0x00bf0000} • {0x0000004f, 0x40921f3c, 0x45f34192, 0x1f418000} • {0x0000005d, 0x3f00003f, 0x0000bc55, 0x55be0000} • {0x0000005e, 0x3811113a, 0x5555b2a0, 0x1ab56c16} • {0x0000005f, 0x2c71de2f, 0xa01aa5ae, 0x64a927e4}

Table 5-75: Particle System Shader Uniforms and Their Corresponding Registers

Uniform	Bound Register
dmp_PartSys.color	c26.xyzw - –c29.xyzw
dmp_PartSys.viewport	c30.xy
dmp_PartSys.pointSize	c31.xy

Uniform	Bound Register
dmp_PartSys.time	c31.z
dmp_PartSys.speed	c31.w
dmp_PartSys.distanceAttenuation	c32.xyz
dmp_PartSys.countMax	c32.w
dmp_PartSys.randSeed	c33.xyzw
dmp_PartSys.aspect	c34.xyzw - —c37.xyzw
dmp_PartSys.randomCore	c38.xyzw

5.8.41 Clearing the Framebuffer Cache

Cached data is flushed for both the color buffer and depth buffer if a value of 1 is written to bit [0:0] of register `0x111`. The cache tag is cleared for both the color buffer and depth buffer if a value of 1 is written to bit [0:0] of register `0x110`. A `0x110` command must always be accompanied by a `0x111` command, with the `0x111` command first.

These commands are inserted immediately before commands that generate interrupts. Commands that generate interrupts occur when the **glFlush**, **glFinish**, or **glClear** function is called, when `NN_GX_STATE_FRAMEBUFFER` is validated after the color buffer or depth buffer address has changed, when `NN_GX_STATE_FBACCESS` is validated, and when the 3D command buffer is split by **nngxSplitDrawCmdlist** or a similar function. In addition to the situations just listed, standalone `0x111` commands are generated by the **glDrawArrays** and **glDrawElements** functions immediately after a rendering kick command.

In general, a clear operation performed by a `0x111` and `0x110` command pair is required when the color buffer or depth buffer are cleared, when the color buffer or depth buffer settings (size, address or format) are changed, and when the read-write access pattern is changed. Separately from these situations, a standalone `0x111` command must always be set immediately after a rendering command.

5.8.42 Commands That Generate Interrupts (Split Commands)

Writing a value of `0x12345678` to register `0x10` causes a P3D (PICA 3D Module) interrupt to occur. Set this command when splitting the 3D command buffer.

5.8.43 Settings Information for Otherwise Undocumented Bits

If information is not given for the bit settings of a register described in these specifications, it might not be accessed via a byte enable setting of 0 or it might be set to a fixed value. This information is shown in the following table. Any bits not mentioned in the following table are assumed to be arbitrary settings that do not affect the hardware, but we recommend setting them equal to 0. (Do not set any registers whose existence itself is not mentioned in this document.)

Table 5-76 Otherwise Undocumented Bit Setting Information

Setting Register	Description
`0x47`, bits [31:8]	Do not set a byte enable value of 0 to access this.
`0x61`, bits [31:8]	Do not set a byte enable value of 0 to access this.
`0x62`, bits [31:8]	Do not set a byte enable value of 0 to access this.
`0x6a`, bits [31:24]	Do not set a byte enable value of 0 to access this.
`0x6e`, bit [24:24]	Set equal to 1.
`0x80`, bit [3:3] and bits [31:24]	Set equal to 0.
`0x80`, bits [23:17]	Set equal to 0 when writing to bit [16:16] of the same register to clear the texture cache. Otherwise, do not set a byte enable value of 0 to access this.
`0x80`, bit [12:12]	Set equal to 1.
`0x83`, bits [17:16]	Set equal to 0.
`0x93`, bits [17:16]	Set equal to 0.
`0x9b`, bits [17:16]	Set equal to 0.
`0x0ac`, bits [10:3]	Set equal to `0x60`.
`0x0ad`, bits [31:8]	Set equal to `0xe0c080`.
`0x0e0`, bits [25:24]	Set equal to 0.
`0x100`, bits [25:16]	Set equal to `0x0e4`.
`0x110`, bits [31:1]	Set equal to 0.
`0x111`, bits [31:1]	Set equal to 0.
`0x11e`, bit [24:24]	Set equal to 1.
`0x1c3`, bit [31:31]	Set equal to 1.
`0x1c3`, bits [11:8]	Set equal to 4.
`0x1c4`, bit [18:18]	Set equal to 1.
`0x229`, bit [9:9]	Set equal to 0.
`0x229`, bits [23:16]	Do not set a byte- enable value of 0 to access this.
`0x244`, bits [31:8]	Do not set a byte- enable value of 0 to access this.
`0x245`, bits [7:1]	Set equal to 0.
`0x245`, bits [31:8]	Do not set a byte- enable value of 0 to access this.
`0x253`, bits [31:16]	Do not set a byte- enable value of 0 to access this.
`0x25e`, bit [16:16]	Do not set a byte- enable value of 0 to access this.

Setting Register	Description
0x25f, bits [31:1]	Set equal to 0.
0x280, bits [31:16]	Set equal to 0x7fff.
0x289, bits [23:16]	Do not set a byte- enable value of 0 to access this.
0x28a, bits [31:16]	Set equal to 0x7fff.
0x28d, bits [31:16]	Set equal to 0.
0x2b0, bits [31:16]	Set equal to 0x7fff.
0x2b9, bits [15:8]	Set equal to 0.
0x2b9, bits [23:16]	Do not set a byte- enable value of 0 to access this.
0x2b9, bits [31:24]	Set equal to 0xa0.
0x2ba, bits [31:16]	Set equal to 0x7fff.
0x2bd, bits [31:16]	Set equal to 0.

5.9 Code to Convert Formats for PICA Register Settings

When an application sets a value using the DMPGL 2.0 API, the DMPGL 2.0 driver may convert it into a different format before writing it to a PICA register. This section shows code used by the DMPGL 2.0 driver to convert formats.

5.9.1 Converting from float32 to float24

The following code converts a 32-bit floating-point number into a 24-bit floating-point number (with a 1-bit sign, 7-bit exponent, and 16-bit mantissa). If you pass a 32-bit floating-point number to _inarg, a 24-bit floating-point number is stored as an unsigned int variable in _outarg.

Code 5-11 Conversion into a 24-Bit Floating-Point Number

```
#define UTL_F2F_16M7E(_inarg, _outarg) ¥
   { ¥
      unsigned uval_, m_; ¥
      int e_; ¥
      float f_; ¥
      static const int bias_ = 128 - (1 << (7 - 1)); ¥
      f_ = (_inarg); ¥
      uval_ = *(unsigned*)&f_; ¥
      e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; ¥
      m_ = (uval_ & 0x7fffff) >> (23 - 16); ¥
      if (e_ >= 0) ¥
         outarg = m_ | (e_ << 16) | ((uval_ >> 31) << (16 + 7)); ¥
```

```
        else ¥
            outarg = ((uval_ >> 31) << (16 + 7)); ¥
    }
```

5.9.2 Converting from float32 to float16

The following code converts a 32-bit floating-point number into a 16-bit floating-point number (with a 1-bit sign, 5-bit exponent, and 10-bit mantissa). If you pass a 32-bit floating-point number to `_inarg`, a 16-bit floating-point number is stored as an `unsigned int` variable in `_outarg`.

Code 5-12 Conversion into a 16-Bit Floating-Point Number

```
#define UTL_F2F_10M5E(_inarg, _outarg) ¥
    { ¥
        unsigned uval_, m_; ¥
        int e_; ¥
        float f_; ¥
        static const int bias_ = 128 - (1 << (5 - 1)); ¥
        f_ = (_inarg); ¥
        uval_ = *(unsigned*)&f_; ¥
        e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; ¥
        m_ = (uval_ & 0x7fffff) >> (23 - 10); ¥
        if (e_ >= 0) ¥
            outarg = m_ | (e_ << 10) | ((uval_ >> 31) << (10 + 5)); ¥
        else ¥
            outarg = ((uval_ >> 31) << (10 + 5)); ¥
    }
```

5.9.3 Converting from float32 to float31

The following code converts a 32-bit floating-point number into a 31-bit floating-point number (with a 1-bit sign, 7-bit exponent, and 23-bit mantissa). When you pass a 32-bit floating-point number into `_inarg`, a 31-bit floating-point number is stored as an `unsigned int` variable in `_outarg`.

Code 5-13 Conversion into a 31-Bit Floating-Point Number

```
#define UTL_F2F_23M7E(_inarg, _outarg) ¥
    { ¥
        unsigned uval_, m_; ¥
        int e_; ¥
        float f_; ¥
        static const int bias_ = 128 - (1 << (7 - 1)); ¥
        f_ = (_inarg); ¥
        uval_ = *(unsigned*)&f_; ¥
        e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; ¥
        m_ = (uval_ & 0x7fffff) >> (23 - 23); ¥
```

```
        if (e_ >= 0) ¥
            outarg = m_ | (e_ << 23) | ((uval_ >> 31) << (23 + 7)); ¥
        else ¥
            outarg = ((uval_ >> 31) << (23 + 7)); ¥
    }
```

5.9.4 Converting from float32 to float20

The following code converts a 32-bit floating-point number into a 20-bit floating-point number (with a 1-bit sign, 7-bit exponent, and 12-bit mantissa). When you pass a 32-bit floating-point number into `_inarg`, a 20-bit floating-point number is stored as an `unsigned int` variable in `_outarg`.

Code 5-14 Conversion into a 20-Bit Floating-Point Number

```
#define UTL_F2F_12M_7E(_inarg, _outarg) ¥
    { ¥
        unsigned uval_, m_; ¥
        int e_; ¥
        float f_; ¥
        static const int bias_ = 128 - (1 << (7 - 1)); ¥
        f_ = (_inarg); ¥
        uval_ = *(unsigned*)&f_; ¥
        e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; ¥
        m_ = (uval_ & 0x7fffff) >> (23 - 12); ¥
        if (e_ >= 0) ¥
            _outarg = m_ | (e_ << 12) | ((uval_ >> 31) << (12 + 7)); ¥
        else ¥
            _outarg = ((uval_ >> 31) << (12 + 7)); ¥
    }
```

5.9.5 Converting a 32-Bit Floating-Point Number into an 8-Bit Signed Fixed-Point Number with 7 Fractional Bits

The following code converts a 32-bit floating-point number into an 8-bit signed fixed-point number with 7 decimal bits. The most-significant bit indicates the sign and is followed by seven fractional bits. Negative values are represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, an 8-bit fixed-point number is stored in `_outarg`.

Code 5-15 Conversion into an 8-Bit Signed Fixed-Point Number with 7 Fractional Bits

```
#define UTL_F2FX_8W_1I_T(_inarg, _outarg) ¥
    { ¥
        float f_; ¥
        unsigned v_; ¥
        f_ = (_inarg); ¥
        v_ = *(unsigned*)&f_; ¥
```

```
        if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) ¥
            outarg = 0; ¥
        else ¥
        { ¥
            f_ += 0.5f * (1 << 1); ¥
            f_ *= 1 << (8 - 1); ¥
            if (f_ < 0) ¥
                f_ = 0; ¥
            else if (f_ >= (1 << 8)) ¥
                f_ = (1 << 8) - 1; ¥
            if (f_ >= (1 << (8 - 1))) ¥
                outarg = (unsigned)(f_ - (1 << (8 - 1))); ¥
            else ¥
                outarg = (unsigned)(f_ + (1 << (8 - 1))); ¥
        } ¥
    }
```

5.9.6 Converting a 32-Bit Floating-Point Number into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits

The following code converts a 32-bit floating-point number into a 12-bit signed fixed-point number with 11 fractional bits. The most-significant bit indicates the sign and is followed by 11 fractional bits that set an absolute value (negative values are not represented in two's complement). If you pass a 32-bit floating-point number to `_inarg`, a 12-bit fixed-point number is stored in `_outarg`.

Code 5-16 Conversion into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits

```
#define UTL_F2FX_12W_1I_F(_inarg, _outarg) ¥
    { ¥
        float f_; ¥
        unsigned v_; ¥
        f_ = (_inarg); ¥
        v_ = *(unsigned*)&f_; ¥
        if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) ¥
            outarg = 0; ¥
        else ¥
        { ¥
            f_ *= (1 << (12 - 1)); ¥
            if (f_ < 0) ¥
            { ¥
                outarg = 1 << (12 - 1); ¥
                f_ = -f_; ¥
            } ¥
            else ¥
```

```
            outarg = 0; ¥
        if (f_ >= (1 << (12 - 1))) f_ = (1 << (12 - 1)) - 1; ¥
            outarg |= (unsigned)(f_); ¥
    } ¥
}
```

5.9.7 Converting a 32-Bit Floating-Point Number into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits (Alternate Method)

The following code converts a 32-bit floating-point number into a 12-bit signed fixed-point number with 11 fractional bits. The most-significant bit indicates the sign and is followed by 11 fractional bits. Negative values are represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, a 12-bit fixed-point number is stored in `_outarg`.

Code 5-17 Alternate Conversion into a 12-Bit Signed Fixed-Point Number with 11 Fractional Bits

```
#define UTL_F2FX_12W_1I_T(_inarg, _outarg) ¥
    { ¥
        float f_; ¥
        unsigned v_; ¥
        f_ = (_inarg); ¥
        v_ = *(unsigned*)&f_; ¥
        if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) ¥
            outarg = 0; ¥
        else ¥
        { ¥
            f_ += 0.5f * (1 << 1); ¥
            f_ *= 1 << (12 - 1); ¥
            if (f_ < 0) ¥
                f_ = 0; ¥
            else if (f_ >= (1 << 12)) ¥
                f_ = (1 << 12) - 1; ¥
            if (f_ >= (1 << (12 - 1))) ¥
                outarg = (unsigned)(f_ - (1 << (12 - 1))); ¥
            else ¥
                outarg = (unsigned)(f_ + (1 << (12 - 1))); ¥
        } ¥
    }
```

5.9.8 Converting a 32-Bit Floating-Point Number into a 13-Bit Signed Fixed-Point Number with 8 Fractional Bits

The following code converts a 32-bit floating-point number into a 13-bit signed fixed-point number with 8 fractional bits. The most-significant bit indicates the sign and is followed by four integer bits and eight fractional bits, respectively. Negative values are represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, a 13-bit fixed-point number is stored in `_outarg`.

Code 5-18 Conversion into a 13-Bit Signed Fixed-Point Number with 8 Fractional Bits

```
#define UTL_F2FX_13W_5I_T(_inarg, _outarg) ¥
{ ¥
    float f_; ¥
    unsigned v_; ¥
    f_ = (_inarg); ¥
    v_ = *(unsigned*)&f_; ¥
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) ¥
        outarg = 0; ¥
    else ¥
    { ¥
        f_ += 0.5f * (1 << 5); ¥
        f_ *= 1 << (13 - 5); ¥
        if (f_ < 0) ¥
            f_ = 0; ¥
        else if (f_ >= (1 << 13)) ¥
            f_ = (1 << 13) - 1; ¥
        if (f_ >= (1 << (13 - 1))) ¥
            outarg = (unsigned)(f_ - (1 << (13 - 1))); ¥
        else ¥
            outarg = (unsigned)(f_ + (1 << (13 - 1))); ¥
    } ¥
}
```

5.9.9 Converting a 32-Bit Floating-Point Number into a 13-Bit Signed Fixed-Point Number with 11 Fractional Bits

The following code converts a 32-bit floating-point number into a 13-bit signed fixed-point number with 11 fractional bits. The most-significant bit indicates the sign and is followed by one integer bit and 11 fractional bits, respectively. Negative values are represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, a 13-bit fixed-point number is stored in `_outarg`.

Code 5-19 Conversion into a 13-Bit Signed Fixed-Point Number with 11 Fractional Bits

```
#define UTL_F2FX_13W_2I_T(_inarg, _outarg) ¥
    { ¥
        float f_; ¥
        unsigned v_; ¥
        f_ = (_inarg); ¥
        v_ = *(unsigned*)&f_; ¥
        if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) ¥
```

```
            outarg = 0; ¥
        else ¥
        { ¥
            f_ += 0.5f * (1 << 2); ¥
            f_ *= 1 << (13 - 2); ¥
            if (f_ < 0) ¥
                f_ = 0; ¥
            else if (f_ >= (1 << 13)) ¥
                f_ = (1 << 13) - 1; ¥
            if (f_ >= (1 << (13 - 1))) ¥
                outarg = (unsigned)(f_ - (1 << (13 - 1))); ¥
            else ¥
                outarg = (unsigned)(f_ + (1 << (13 - 1))); ¥
        } ¥
    }
```

5.9.10 Converting a 32-Bit Floating-Point Number into a 16-Bit Signed Fixed-Point Number with 12 Fractional Bits

The following code converts a 32-bit floating-point number into a 16-bit signed fixed-point number with 12 fractional bits. The most-significant bit indicates the sign and is followed by three integer bits and 12 fractional bits, respectively. Negative values are represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, a 16-bit fixed-point number is stored in `_outarg`.

Code 5-20 Conversion into a 16-Bit Fixed-Point Number

```
#define UTL_F2FX_16W_4I_T(_inarg, _outarg) ¥
    { ¥
        float f_; ¥
        unsigned v_; ¥
        f_ = (_inarg); ¥
        v_ = *(unsigned*)&f_; ¥
        if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) ¥
            outarg = 0; ¥
        else ¥
        { ¥
            f_ += 0.5f * (1 << 4); ¥
            f_ *= 1 << (16 - 4); ¥
            if (f_ < 0) ¥
                f_ = 0; ¥
            else if (f_ >= (1 << 16)) ¥
                f_ = (1 << 16) - 1; ¥
            if (f_ >= (1 << (16 - 1))) ¥
                outarg = (unsigned)(f_ - (1 << (16 - 1))); ¥
```

```
        else ¥
            outarg = (unsigned)(f_ + (1 << (16 - 1))); ¥
    } ¥
}
```

5.9.11 Converting a 32-Bit Floating-Point Number into an 8-Bit Unsigned Fixed-Point Number with No Fractional Bits

The following code converts a 32-bit floating-point number into an 8-bit unsigned fixed-point number with no fractional bits. If you pass a 32-bit floating-point number to `_inarg`, an 8-bit fixed-point number is stored in `_outarg`.

Code 5-21 Conversion into an 8-Bit Unsigned Fixed-Point Number with No Fractional Bits

```
#define UTL_F2UFX_8W_8I(_inarg, _outarg) ¥
    { ¥
        float f_ = (_inarg); ¥
        unsigned val_; ¥
        unsigned v_ = *(unsigned*)&f_; ¥
        if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) ¥
            val_ = 0; ¥
        else ¥
        { ¥
            f_ *= 1 << (8 - 8); ¥
            if (f_ >= (1 << 8)) ¥
                val_ = (1 << 8) - 1; ¥
            else ¥
                val_ = (unsigned)(f_); ¥
        } ¥
        (_outarg) = val_; ¥
    }
```

5.9.12 Converting a 32-Bit Floating-Point Number into an 11-Bit Unsigned Fixed-Point Number with 11 Fractional Bits

The following code converts a 32-bit floating-point number into an 11-bit unsigned fixed-point number with 11 fractional bits. If you pass a 32-bit floating-point number to `_inarg`, an 11-bit fixed-point number is stored in `_outarg`.

Code 5-22 Conversion into an 11-Bit Unsigned Fixed-Point Number with 11 Fractional Bits

```
#define UTL_F2UFX_11W_0I(_inarg, _outarg) ¥
    { ¥
        float f_ = (_inarg); ¥
        unsigned val_; ¥
        unsigned v_ = *(unsigned*)&f_; ¥
```

```
      if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) ¥
         val_ = 0; ¥
      else ¥
      { ¥
         f_ *= 1 << (11 - 0); ¥
         if (f_ >= (1 << 11)) ¥
            val_ = (1 << 11) - 1; ¥
         else ¥
            val_ = (unsigned)(f_); ¥
      } ¥
      (_outarg) = val_; ¥
   }
```

5.9.13 Converting a 32-Bit Floating-Point Number into a 12-Bit Unsigned Fixed-Point Number with 12 Fractional Bits

The following code converts a 32-bit floating-point number into a 12-bit unsigned fixed-point number with 12 fractional bits. If you pass a 32-bit floating-point number to `_inarg`, a 12-bit fixed-point number is stored in `_outarg`.

Code 5-23 Conversion into a 12-Bit Unsigned Fixed-Point Number with 12 Fractional Bits

```
#define UTL_F2UFX_12W_0I(_inarg, _outarg) ¥
   { ¥
      float f_ = (_inarg); ¥
      unsigned val_; ¥
      unsigned v_ = *(unsigned*)&f_; ¥
      if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) ¥
         val_ = 0; ¥
      else ¥
      { ¥
         f_ *= 1 << (12 - 0); ¥
         if (f_ >= (1 << 12)) ¥
            val_ = (1 << 12) - 1; ¥
         else ¥
            val_ = (unsigned)(f_); ¥
      } ¥
      (_outarg) = val_; ¥
   }
```

5.9.14 Converting a 32-Bit Floating-Point Number into a 24-Bit Unsigned Fixed-Point Number with 24 Fractional Bits

The following code converts a 32-bit floating-point number into a 24-bit unsigned fixed-point number with 24 fractional bits. If you pass a 32-bit floating-point number to `_inarg`, a 24-bit fixed-point number is stored in `_outarg`.

Code 5-24 Conversion into a 24-Bit Fixed-Point Number with 24 Fractional Bits

```
#define UTL_F2UFX_24W_0I(_inarg, _outarg) ¥
    { ¥
        float f_ = (_inarg); ¥
        unsigned val_; ¥
        unsigned v_ = *(unsigned*)&f_; ¥
        if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) ¥
            val_ = 0; ¥
        else ¥
        { ¥
            f_ *= 1 << (24 - 0); ¥
            if (f_ >= (1 << 24)) ¥
                val_ = (1 << 24) - 1; ¥
            else ¥
                val_ = (unsigned)(f_); ¥
        } ¥
        (_outarg) = val_; ¥
    }
```

5.9.15 Converting a 32-Bit Floating-Point Number into a 24-Bit Unsigned Fixed-Point Number with 8 Fractional Bits

The following code converts a 32-bit floating-point number into a 24-bit unsigned fixed-point number with 8 fractional bits. If you pass a 32-bit floating-point number to `_inarg`, a 24-bit fixed-point number is stored in `_outarg`.

Code 5-25 Conversion into a 24-Bit Fixed-Point Number with 8 Fractional Bits

```
#define UTL_F2UFX_24W_16I(_inarg, _outarg) ¥
    { ¥
        float f_ = (_inarg); ¥
        unsigned val_; ¥
        unsigned v_ = *(unsigned*)&f_; ¥
        if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) ¥
            val_ = 0; ¥
        else ¥
        { ¥
            f_ *= 1 << (24 - 16); ¥
```

```
        if (f_ >= (1 << 24)) ¥
            val_ = (1 << 24) - 1; ¥
        else ¥
            val_ = (unsigned)(f_); ¥
    } ¥
    (_outarg) = val_; ¥
}
```

5.9.16 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer

The following code converts a 32-bit floating-point number between 0 and 1 into an 8-bit unsigned integer. If you pass a 32-bit floating-point number into `f`, an 8-bit unsigned integer is returned.

Code 5-26 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer

```
((unsigned)(0.5f + (f) * (float)((1 << 8) - 1)))
```

5.9.17 Alternate Conversion from a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer

The following code converts a 32-bit floating-point number between 0 and 1 into an 8-bit unsigned integer. If you pass a 32-bit floating-point number into `f`, an 8-bit unsigned integer is returned.

Code 5-27 Alternate Conversion of a 32-Bit Floating-Point Number (0–1) into an 8-Bit Unsigned Integer

```
((unsigned)((f) * (float)((1 << 8) - 1)))
```

5.9.18 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Signed Integer

The following code converts a 32-bit floating-point number between 0 and 1 into an 8-bit signed integer. If you pass a 32-bit floating-point number into `f`, an 8-bit signed integer is returned.

Code 5-28 Converting a 32-Bit Floating-Point Number (0–1) into an 8-Bit Signed Integer

```
(((unsigned int)(fabs(127.f * (f))) & 0x7f)|(f < 0 ? 0x80 : 0))
```

5.9.19 Converting a 16-Bit Floating-Point Value into a 32-Bit Floating-Point Value

The following code converts a 16-bit floating-point number (with one sign bit, a 5-bit exponent, and a 10-bit mantissa) into a 32-bit floating-point number. If you pass a 16-bit floating-point number stored as an `unsigned int` to `_inarg`, a 32-bit floating-point number is stored in the `float` type variable specified by `_outarg`.

Code 5-29 Converting a 16-Bit Floating-Point Value into a 32-Bit Floating-Point Value

```
#define UTL_U2F_10M5E(_inarg, _outarg) ¥
{ ¥
    int e_; ¥
```

```
    unsigned m_; ¥
    unsigned u_ = (_inarg); ¥
    const int width_ = 10 + 5 + 1; ¥
    const int bias_ = 128 - (1 << (5 - 1)); ¥
    e_ = (u_ >> 10) & ((1 << 5) - 1); ¥
    m_ = u_ & ((1 << 10) - 1); ¥
    if (u_ & ((1 << (width_ - 1)) - 1)) ¥
        u_ = ((u_ >> (5 + 10)) << 31) | (m_ << (23 - 10)) | ((e_ + bias_) << 23); ¥
    else ¥
        u_ = ((u_ >> (5 + 10)) << 31); ¥
    (_outarg) = *(float*)&u_; ¥
}
```

5.10 Command Cache Restrictions and Precautions

The following restrictions and precautions apply when you use the command cache.

- Even after the **nngxValidateState** function has validated the state of the reserved fragment shader uniforms, lighting-related commands are generated again when rendering functions are called when fragment lighting is enabled (`dmp_FragmentLighting.enabled` is `GL_TRUE`) and all light sources are disabled (`dmp_FragmentLightSource[i].enabled` is `GL_FALSE` for every light source).
- Even after the **nngxValidateState** function has validated the state of the reserved fragment shader uniforms, commands related to the `dmp_Gas.accMax` reserved uniform are generated again when rendering functions are called.
- If the `dmp_Gas.autoAcc` reserved fragment uniform is `GL_TRUE` and you start or stop saving a command list at the same time as the value of `dmp_FragOperation.mode` changes to or from `GL_FRAGOP_MODE_GAS_ACC_DMP`, commands related to `dmp_Gas.autoAcc` in that command list may not be applied correctly.
- The size of the 3D command buffer to execute must be a multiple of 16. Use the **nngxAdd3DCommand** function to add `0x0000000000000000` as dummy data and thereby adjust the size.
- When the **glUseProgram** function specifies `0` and has then been called, no commands will be generated even if the states related to the program or shader are validated.

5.11 PICA Register List

The following table lists the functions, state flags, uniforms, and other items related to each of the PICA registers. Related functions do not necessarily generate commands when called. Some of the functions mentioned here have parameters that affect settings. If a setting depends on the shader assembly implementation, it is noted as the **glUseProgram** function.

Table 5-77 PICA Register List

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x10	[31:0]	• **nngxSplitDrawCmdlist** • **nngxTransferRenderImage**	-
0x40	[1:0]	• **glCullFace** • **glDisable**(***GL_CULL_FACE***) • **glEnable**(***GL_CULL_FACE***) • **glFrontFace**	• NN_GX_STATE_OTHERS
0x41	[23:0]	• ***width*** in **glViewport**	• NN_GX_STATE_OTHERS
0x42	[31:0]		
0x43	[23:0]	• ***height*** in **glViewport**	
0x44	[31:0]		
0x47	[0:0]	• dmp_FragOperation.enableClippingPlane	• NN_GX_STATE_FSUNIFORM
0x48	[23:0]	• dmp_FragOperation.clippingPlane	• NN_GX_STATE_FSUNIFORM
0x49	[23:0]		
0x4a	[23:0]		
0x4b	[23:0]		
0x4d	[23:0]	• dmp_FragOperation.wScale • **glDepthRangef**	• NN_GX_STATE_FSUNIFORM • NN_GX_STATE_TRIOFFSET
0x4e	[23:0]	• dmp_FragOperation.wScale • **glDepthRangef** • **glDisable**(***GL_POLYGON_OFFSET_FILL***) • **glEnable**(***GL_POLYGON_OFFSET_FILL***) • ***units*** in **glPolygonOffset**	
0x4f	[2:0]	• **glUseProgram**	• NN_GX_STATE_SHADERPROGRAM
0x50	[31:0]		
0x51	[31:0]		
0x52	[31:0]		
0x53	[31:0]		
0x54	[31:0]		
0x55	[31:0]		
0x56	[31:0]		
0x61	[1:0]	• **glEarlyDepthFuncDMP**	• NN_GX_STATE_OTHERS
0x62	[0:0]	• **glDisable**(***GL_EARLY_DEPTH_TEST_DMP***)	• NN_GX_STATE_OTHERS

Register Setting	Bits	Related Functions and Uniforms	State Flags
		• **glEnable**(***GL_EARLY_DEPTH_TEST_DMP***)	
0x63	[0:0]	• **glClear**(***GL_EARLY_DEPTH_BUFFER_BIT_DMP***)	-
0x64	[0:0]	• **glUseProgram**	• NN_GX_STATE_SHADERPROGRAM
0x65	[1:0]	• **glDisable**(***GL_SCISSOR_TEST***) • **glEnable**(***GL_SCISSOR_TEST***) • **glScissor**	• NN_GX_STATE_SCISSOR
0x66	[9:0]		
	[25:16]		
0x67	[9:0]		
	[25:16]		
0x68	[9:0]	• ***x*** and ***y*** in **glViewport**	• NN_GX_STATE_OTHERS
	[25:16]		
0x6a	[23:0]	• **glClearEarlyDepthDMP**	• NN_GX_STATE_OTHERS
0x6d	[0:0]	• dmp_FragOperation.wScale	• NN_GX_STATE_FSUNIFORM
0x6e	[10:0]	Target rendering object: • ***width*** in **glRenderbufferStorage** • ***width*** in **glTexture2Dimage2D**	• NN_GX_STATE_FRAMEBUFFER
0x6e	[21:12]	Target rendering object: • ***height*** in **glRenderbufferStorage** • ***height*** in **glTexture2Dimage2D**	• NN_GX_STATE_FRAMEBUFFER
0x6f	[1:0]	• **glUseProgram**	• NN_GX_STATE_SHADERPROGRAM
	[10:8]		
	[16:16]		
	[24:24]		
0x80	[2:0]	• dmp_Texture[i].samplerType(i=0,1,2) • **glDrawArrays** • **glDrawElements**	-
0x80	[9:8]	• dmp_Texture[3].texcoord	• NN_GX_STATE_FSUNIFORM
0x80	[10:10]	• dmp_Texture[3].samplerType	• NN_GX_STATE_FSUNIFORM
0x80	[13:13]	• dmp_Texture[2].texcoord	• NN_GX_STATE_FSUNIFORM
0x80	[16:16]	• dmp_Texture[i].samplerType(i=0,1,2) • General texture settings made by **glTexParameter**	• NN_GX_STATE_TEXTURE
0x81	[31:0]	• **glTexParameter**(***pname=GL_TEXTURE_BORDER_COLOR***)	• NN_GX_STATE_TEXTURE

Register Setting	Bits	Related Functions and Uniforms	State Flags
		This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	
0x82	[10:0]	• ***height*** in **glTexImage2D** • ***height*** in **glCompressedTexImage2D** • ***height*** in **glCopyTexImage2D** This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	• NN_GX_STATE_TEXTURE
0x82	[26:16]	• ***width*** in **glTexImage2D** • ***width*** in **glCompressedTexImage2D** • ***width*** in **glCopyTexImage2D** This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	• NN_GX_STATE_TEXTURE
0x83	[1:1]	• **glTexParameter**(***pname=GL_TEXTURE_MAG_FILTER***) This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	• NN_GX_STATE_TEXTURE
0x83	[2:2]	• **glTexParameter**(***pname=GL_TEXTURE_MIN_FILTER***) This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	• NN_GX_STATE_TEXTURE
0x83	[5:4]	• ***internalformat*** in **glTexImage2D** • ***internalformat*** in **glCompressedTexImage2D** • ***internalformat*** in **glCopyTexImage2D** • This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	• NN_GX_STATE_TEXTURE
0x83	[10:8]	• **glTexParameter**(***pname=GL_TEXTURE_WRAP_T***) This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	• NN_GX_STATE_TEXTURE
0x83	[14:12]	• **glTexParameter**(***pname=GL_TEXTURE_WRAP_S***) This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	• NN_GX_STATE_TEXTURE
0x83	[20:20]	• ***internalformat*** in **glTexImage2D** This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	• NN_GX_STATE_TEXTURE
0x83	[24:24]	• **glTexParameter**(***pname=GL_TEXTURE_MIN_FILTER***) This depends on settings for the texture object bound to GL_TEXTURE0 when rendering.	• NN_GX_STATE_TEXTURE
0x83	[30:28]	• dmp_Texture[0].samplerType	• NN_GX_STATE_TEXTURE
0x84	[12:0]	• **glTexParameter**(***pname=GL_TEXTURE_LOD_BIAS***)	• NN_GX_STATE_TEXTURE

Register Setting	Bits	Related Functions and Uniforms	State Flags
		This depends on settings for the texture object bound to `GL_TEXTURE0` when rendering.	
`0x84`	[19:16]	• *level* in **`glTexImage2D`** • *level* in **`glCompressedTexImage2D`** • *level* in **`glCopyTexImage2D`** • **`glTexParameter`**(***pname=GL_TEXTURE_MIN_FILTER***) This depends on settings for the texture object bound to `GL_TEXTURE0` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x84`	[27:24]	• **`glTexParameter`**(***pname=GL_TEXTURE_MIN_FILTER***) • **`glTexParameter`**(***pname=GL_TEXTURE_MIN_LOD***) This depends on settings for the texture object bound to `GL_TEXTURE0` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x85`	[27:0]	• Texture address allocated by **`glTexImage2D`**, **`glCompressedTexImage2D`**, or **`glCopyTexImage2D`** This depends on settings for the texture object bound to `GL_TEXTURE0` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x86`	[21:0]		
`0x87`	[21:0]		
`0x88`	[21:0]		
`0x89`	[21:0]		
`0x8a`	[21:0]		
`0x8b`	[0:0]	• `dmpTexture[0].perspectiveShadow`	• `NN_GX_STATE_FSUNIFORM`
`0x8b`	[23:1]	• `dmpTexture[0].shadowZBias`	• `NN_GX_STATE_FSUNIFORM`
`0x8b`	[31:24]	• `dmpTexture[0].shadowZScale`	• `NN_GX_STATE_FSUNIFORM`
`0x8e`	[3:0]	• ***internalformat*** in **`glTexImage2D`** • ***internalformat*** in **`glCompressedTexImage2D`** • ***internalformat*** in **`glCopyTexImage2D`** This depends on settings for the texture object bound to `GL_TEXTURE0` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x8f`	[0:0]	• `dmp_FragmentLighting.enabled`	• `NN_GX_STATE_FSUNIFORM`
`0x91`	[31:0]	• **`glTexParameter`**(***pname=GL_TEXTURE_BORDER_COLOR***) This depends on settings for the texture object bound to `GL_TEXTURE1` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x92`	[10:0]	• *height* in **`glTexImage2D`** • *height* in **`glCompressedTexImage2D`** • *height* in **`glCopyTexImage2D`** This depends on settings for the texture object bound to `GL_TEXTURE1` when rendering.	• `NN_GX_STATE_TEXTURE`

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x92	[26:16]	• ***width*** in **glTexImage2D** • ***width*** in **glCompressedTexImage2D** • ***width*** in **glCopyTexImage2D** This depends on settings for the texture object bound to GL_TEXTURE1 when rendering.	• NN_GX_STATE_TEXTURE
0x93	[1:1]	• **glTexParameter**(***pname=GL_TEXTURE_MAG_FILTER***) This depends on settings for the texture object bound to GL_TEXTURE1 when rendering.	• NN_GX_STATE_TEXTURE
0x93	[2:2]	• **glTexParameter**(***pname=GL_TEXTURE_MIN_FILTER***) This depends on settings for the texture object bound to GL_TEXTURE1 when rendering.	• NN_GX_STATE_TEXTURE
0x93	[5:4]	• ***internalformat*** in **glTexImage2D** • ***internalformat*** in **glCompressedTexImage2D** • ***internalformat*** in **glCopyTexImage2D** This depends on settings for the texture object bound to GL_TEXTURE1 when rendering.	• NN_GX_STATE_TEXTURE
0x93	[10:8]	• **glTexParameter**(***pname=GL_TEXTURE_WRAP_T***) This depends on settings for the texture object bound to GL_TEXTURE1 when rendering.	• NN_GX_STATE_TEXTURE
0x93	[14:12]	• **glTexParameter**(***pname=GL_TEXTURE_WRAP_S***) This depends on settings for the texture object bound to GL_TEXTURE1 when rendering.	• NN_GX_STATE_TEXTURE
0x93	[24:24]	• **glTexParameter**(***pname=GL_TEXTURE_MIN_FILTER***) This depends on settings for the texture object bound to GL_TEXTURE1 when rendering.	• NN_GX_STATE_TEXTURE
0x94	[12:0]	• **glTexParameter**(***pname=GL_TEXTURE_LOD_BIAS***) • This depends on settings for the texture object bound to GL_TEXTURE1 when rendering.	• NN_GX_STATE_TEXTURE
0x94	[19:16]	• ***level*** in **glTexImage2D** • ***level*** in **glCompressedTexImage2D** • ***level*** in **glCopyTexImage2D** • **glTexParameter**(***pname=GL_TEXTURE_MIN_FILTER***) This depends on settings for the texture object bound to GL_TEXTURE1 when rendering.	• NN_GX_STATE_TEXTURE
0x94	[27:24]	• **glTexParameter**(***pname=GL_TEXTURE_MIN_FILTER***) • **glTexParameter**(	• NN_GX_STATE_TEXTURE

Register Setting	Bits	Related Functions and Uniforms	State Flags
		pname=GL_TEXTURE_MIN_LOD) This depends on settings for the texture object bound to `GL_TEXTURE1` when rendering.	
`0x95`	[27:0]	• Texture address allocated by **`glTexImage2D`**, **`glCompressedTexImage2D`**, or **`glCopyTexImage2D`** This depends on settings for the texture object bound to `GL_TEXTURE1` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x96`	[3:0]	• ***internalformat*** in **`glTexImage2D`** • ***internalformat*** in **`glCompressedTexImage2D`** • ***internalformat*** in **`glCopyTexImage2D`** This depends on settings for the texture object bound to `GL_TEXTURE1` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x99`	[31:0]	• **`glTexParameter`**(***pname=GL_TEXTURE_BORDER_COLOR***) This depends on settings for the texture object bound to `GL_TEXTURE2` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x9a`	[10:0]	• ***height*** in **`glTexImage2D`** • ***height*** in **`glCompressedTexImage2D`** • ***height*** in **`glCopyTexImage2D`** This depends on settings for the texture object bound to `GL_TEXTURE2` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x9a`	[26:16]	• ***width*** in **`glTexImage2D`** • ***width*** in **`glCompressedTexImage2D`** • ***width*** in **`glCopyTexImage2D`** This depends on settings for the texture object bound to `GL_TEXTURE2` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x9b`	[1:1]	• **`glTexParameter`**(***pname=GL_TEXTURE_MAG_FILTER***) This depends on settings for the texture object bound to `GL_TEXTURE2` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x9b`	[2:2]	• **`glTexParameter`**(***pname=GL_TEXTURE_MIN_FILTER***) This depends on settings for the texture object bound to `GL_TEXTURE2` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x9b`	[5:4]	• ***internalformat*** in **`glTexImage2D`** • ***internalformat*** in **`glCompressedTexImage2D`** • ***internalformat*** in **`glCopyTexImage2D`** This depends on settings for the texture object bound to `GL_TEXTURE2` when rendering.	• `NN_GX_STATE_TEXTURE`
`0x9b`	[10:8]	• **`glTexParameter`**(***pname=GL_TEXTURE_WRAP_T***) This depends on settings for the texture object bound to `GL_TEXTURE2` when rendering.	• `NN_GX_STATE_TEXTURE`

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x9b	[14:12]	• **glTexParameter**(*pname=GL_TEXTURE_WRAP_S*) This depends on settings for the texture object bound to GL_TEXTURE2 when rendering.	• NN_GX_STATE_TEXTURE
0x9b	[24:24]	• **glTexParameter**(*pname=GL_TEXTURE_MIN_FILTER*) This depends on settings for the texture object bound to GL_TEXTURE2 when rendering.	• NN_GX_STATE_TEXTURE
0x9c	[12:0]	• **glTexParameter**(*pname=GL_TEXTURE_LOD_BIAS*) This depends on settings for the texture object bound to GL_TEXTURE2 when rendering.	• NN_GX_STATE_TEXTURE
0x9c	[19:16]	• *level* in **glTexImage2D** • *level* in **glCompressedTexImage2D** • *level* in **glCopyTexImage2D** • **glTexParameter**(*pname=GL_TEXTURE_MIN_FILTER*) This depends on settings for the texture object bound to GL_TEXTURE2 when rendering.	• NN_GX_STATE_TEXTURE
0x9c	[27:24]	• **glTexParameter**(*pname=GL_TEXTURE_MIN_FILTER*) • **glTexParameter**(*pname=GL_TEXTURE_MIN_LOD*) This depends on settings for the texture object bound to GL_TEXTURE2 when rendering.	• NN_GX_STATE_TEXTURE
0x9d	[27:0]	• Texture address allocated by **glTexImage2D**, **glCompressedTexImage2D**, or **glCopyTexImage2D** This depends on settings for the texture object bound to GL_TEXTURE2 when rendering.	• NN_GX_STATE_TEXTURE
0x9e	[3:0]	• *internalformat* in **glTexImage2D** • *internalformat* in **glCompressedTexImage2D** • *internalformat* in **glCopyTexImage2D** • This depends on settings for the texture object bound to GL_TEXTURE2 when rendering.	• NN_GX_STATE_TEXTURE
0x0a8	[2:0]	• dmp_Texture[3].ptClampU	• NN_GX_STATE_FSUNIFORM
0x0a8	[5:3]	• dmp_Texture[3].ptClampV	• NN_GX_STATE_FSUNIFORM
0x0a8	[9:6]	• dmp_Texture[3].ptRgbMap	• NN_GX_STATE_FSUNIFORM
0x0a8	[13:10]	• dmp_Texture[3].ptAlphaMap	• NN_GX_STATE_FSUNIFORM
0x0a8	[14:14]	• dmp_Texture[3].ptAlphaSeparate	• NN_GX_STATE_FSUNIFORM
0x0a8	[15:15]	• dmp_Texture[3].ptNoiseEnable	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x0a8	[17:16]	• dmp_Texture[3].ptShiftU	• NN_GX_STATE_FSUNIFORM
0x0a8	[19:18]	• dmp_Texture[3].ptShiftV	• NN_GX_STATE_FSUNIFORM
0x0a8	[27:20]	• dmp_Texture[3].ptTexBias	• NN_GX_STATE_FSUNIFORM
0x0a9	[15:0]	• dmp_Texture[3].ptNoiseU (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0a9	[31:16]	• dmp_Texture[3].ptNoiseU (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0aa	[15:0]	• dmp_Texture[3].ptNoiseV (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0aa	[31:16]	• dmp_Texture[3].ptNoiseV (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0ab	[15:0]	• dmp_Texture[3].ptNoiseU (1st component)	• NN_GX_STATE_FSUNIFORM
0x0ab	[31:16]	• dmp_Texture[3].ptNoiseV (1st component)	• NN_GX_STATE_FSUNIFORM
0x0ac	[2:0]	• dmp_Texture[3].ptMinFilter	• NN_GX_STATE_FSUNIFORM
0x0ac	[18:11]	• dmp_Texture[3].ptTexWidth	• NN_GX_STATE_FSUNIFORM
0x0ad	[26:19]	• dmp_Texture[3].ptTexBias	• NN_GX_STATE_FSUNIFORM
0x0ad	[7:0]	• dmp_Texture[3].ptTexOffset	• NN_GX_STATE_FSUNIFORM
0x0af	[11:8]	• dmp_Texture[3].ptSampler {RgbMap,AlphaMap,NoiseMap,R,G,B,A} • LUT object data created by **glTexImage1D**	• NN_GX_STATE_LUT
0x0b0–0x0b7	[31:0]		
0x0c0	[3:0]	• dmp_TexEnv[0].srcRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0c0	[7:4]	• dmp_TexEnv[0].srcRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0c0	[11:8]	• dmp_TexEnv[0].srcRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0c0	[19:16]	• dmp_TexEnv[0].srcAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0c0	[23:20]	• dmp_TexEnv[0].srcAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0c0	[27:24]	• dmp_TexEnv[0].srcAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0xc1	[3:0]	• dmp_TexEnv[0].operandRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0xc1	[7:4]	• dmp_TexEnv[0].operandRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0xc1	[11:8]	• dmp_TexEnv[0].operandRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0xc1	[14:12]	• dmp_TexEnv[0].operandAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0xc1	[18:16]	• dmp_TexEnv[0].operandAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0xc1	[22:20]	• dmp_TexEnv[0].operandAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0c2	[3:0]	• dmp_TexEnv[0].combineRgb	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x0c2	[19:16]	• dmp_TexEnv[0].combineAlpha	• NN_GX_STATE_FSUNIFORM
0x0c3	[7:0]	• dmp_TexEnv[0].constRgba (1st component)	• NN_GX_STATE_FSUNIFORM
0x0c3	[15:8]	• dmp_TexEnv[0].constRgba (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0c3	[23:16]	• dmp_TexEnv[0].constRgba (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0c3	[31:24]	• dmp_TexEnv[0].constRgba (4th component)	• NN_GX_STATE_FSUNIFORM
0x0c4	[1:0]	• dmp_TexEnv[0].scaleRgb	• NN_GX_STATE_FSUNIFORM
0x0c4	[17:16]	• dmp_TexEnv[0].scaleAlpha	• NN_GX_STATE_FSUNIFORM
0x0c8	[3:0]	• dmp_TexEnv[1].srcRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0c8	[7:4]	• dmp_TexEnv[1].srcRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0c8	[11:8]	• dmp_TexEnv[1].srcRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0c8	[19:16]	• dmp_TexEnv[1].srcAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0c8	[23:20]	• dmp_TexEnv[1].srcAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0c8	[27:24]	• dmp_TexEnv[1].srcAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0c9	[3:0]	• dmp_TexEnv[1].operandRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0c9	[7:4]	• dmp_TexEnv[1].operandRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0c9	[11:8]	• dmp_TexEnv[1].operandRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0c9	[14:12]	• dmp_TexEnv[1].operandAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0c9	[18:16]	• dmp_TexEnv[1].operandAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0c9	[22:20]	• dmp_TexEnv[1].operandAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0ca	[3:0]	• dmp_TexEnv[1].combineRgb	• NN_GX_STATE_FSUNIFORM
0x0ca	[19:16]	• dmp_TexEnv[1].combineAlpha	• NN_GX_STATE_FSUNIFORM
0x0cb	[7:0]	• dmp_TexEnv[1].constRgba (1st component)	• NN_GX_STATE_FSUNIFORM
0x0cb	[15:8]	• dmp_TexEnv[1].constRgba (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0cb	[23:16]	• dmp_TexEnv[1].constRgba (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0cb	[31:24]	• dmp_TexEnv[1].constRgba (4th component)	• NN_GX_STATE_FSUNIFORM
0x0cc	[1:0]	• dmp_TexEnv[1].scaleRgb	• NN_GX_STATE_FSUNIFORM
0x0cc	[17:16]	• dmp_TexEnv[1].scaleAlpha	• NN_GX_STATE_FSUNIFORM
0x0d0	[3:0]	• dmp_TexEnv[2].srcRgb (1st component)	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x0d0	[7:4]	• dmp_TexEnv[2].srcRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0d0	[11:8]	• dmp_TexEnv[2].srcRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0d0	[19:16]	• dmp_TexEnv[2].srcAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0d0	[23:20]	• dmp_TexEnv[2].srcAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0d0	[27:24]	• dmp_TexEnv[2].srcAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0d1	[3:0]	• dmp_TexEnv[2].operandRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0d1	[7:4]	• dmp_TexEnv[2].operandRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0d1	[11:8]	• dmp_TexEnv[2].operandRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0d1	[14:12]	• dmp_TexEnv[2].operandAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0d1	[18:16]	• dmp_TexEnv[2].operandAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0d1	[22:20]	• dmp_TexEnv[2].operandAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0d2	[3:0]	• dmp_TexEnv[2].combineRgb	• NN_GX_STATE_FSUNIFORM
0x0d2	[19:16]	• dmp_TexEnv[2].combineAlpha	• NN_GX_STATE_FSUNIFORM
0x0d3	[7:0]	• dmp_TexEnv[2].constRgba (1st component)	• NN_GX_STATE_FSUNIFORM
0x0d3	[15:8]	• dmp_TexEnv[2].constRgba (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0d3	[23:16]	• dmp_TexEnv[2].constRgba (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0d3	[31:24]	• dmp_TexEnv[2].constRgba (4th component)	• NN_GX_STATE_FSUNIFORM
0x0d4	[1:0]	• dmp_TexEnv[2].scaleRgb	• NN_GX_STATE_FSUNIFORM
0x0d4	[17:16]	• dmp_TexEnv[2].scaleAlpha	• NN_GX_STATE_FSUNIFORM
0x0d8	[3:0]	• dmp_TexEnv[3].srcRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0d8	[7:4]	• dmp_TexEnv[3].srcRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0d8	[11:8]	• dmp_TexEnv[3].srcRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0d8	[19:16]	• dmp_TexEnv[3].srcAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0d8	[23:20]	• dmp_TexEnv[3].srcAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0d8	[27:24]	• dmp_TexEnv[3].srcAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0d9	[3:0]	• dmp_TexEnv[3].operandRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0d9	[7:4]	• dmp_TexEnv[3].operandRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0d9	[11:8]	• dmp_TexEnv[3].operandRgb (3rd component)	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x0d9	[14:12]	• dmp_TexEnv[3].operandAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0d9	[18:16]	• dmp_TexEnv[3].operandAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0d9	[22:20]	• dmp_TexEnv[3].operandAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0da	[3:0]	• dmp_TexEnv[3].combineRgb	• NN_GX_STATE_FSUNIFORM
0x0da	[19:16]	• dmp_TexEnv[3].combineAlpha	• NN_GX_STATE_FSUNIFORM
0x0db	[7:0]	• dmp_TexEnv[3].constRgba (1st component)	• NN_GX_STATE_FSUNIFORM
0x0db	[15:8]	• dmp_TexEnv[3].constRgba (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0db	[23:16]	• dmp_TexEnv[3].constRgba (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0db	[31:24]	• dmp_TexEnv[3].constRgba (4th component)	• NN_GX_STATE_FSUNIFORM
0x0dc	[1:0]	• dmp_TexEnv[3].scaleRgb	• NN_GX_STATE_FSUNIFORM
0x0dc	[17:16]	• dmp_TexEnv[3].scaleAlpha	• NN_GX_STATE_FSUNIFORM
0x0e0	[2:0]	• dmp_Fog.mode	• NN_GX_STATE_FSUNIFORM
0x0e0	[3:3]	• dmp_Gas.shadingDensitySrc	• NN_GX_STATE_FSUNIFORM
0x0e0	[8:8]	• dmp_TexEnv[1].bufferInput (1st component)	• NN_GX_STATE_FSUNIFORM
0x0e0	[9:9]	• dmp_TexEnv[2].bufferInput (1st component)	• NN_GX_STATE_FSUNIFORM
0x0e0	[10:10]	• dmp_TexEnv[3].bufferInput (1st component)	• NN_GX_STATE_FSUNIFORM
0x0e0	[11:11]	• dmp_TexEnv[4].bufferInput (1st component)	• NN_GX_STATE_FSUNIFORM
0x0e0	[12:12]	• dmp_TexEnv[1].bufferInput (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0e0	[13:13]	• dmp_TexEnv[2].bufferInput (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0e0	[14:14]	• dmp_TexEnv[3].bufferInput (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0e0	[15:15]	• dmp_TexEnv[4].bufferInput (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0e0	[16:16]	• dmp_Fog.zFlip	• NN_GX_STATE_FSUNIFORM
0x0e1	[7:0]	• dmp_Fog.color (1st component)	• NN_GX_STATE_FSUNIFORM
0x0e1	[15:8]	• dmp_Fog.color (2nd component)	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x0e1	[23:16]	• dmp_Fog.color (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0e4	[15:0]	• dmp_Gas.attenuation	• NN_GX_STATE_FSUNIFORM
0xe05	[15:0]	• dmp_Gas.accMax	• NN_GX_STATE_FSUNIFORM
0x0e6	[15:0]	• dmp_Fog.sampler • LUT object data created by **glTexImage1D**	• NN_GX_STATE_LUT
0x0e8–0x0ef	[23:0]		
0x0f0	[3:0]	• dmp_TexEnv[4].srcRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0f0	[7:4]	• dmp_TexEnv[4].srcRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0f0	[11:8]	• dmp_TexEnv[4].srcRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0f0	[19:16]	• dmp_TexEnv[4].srcAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0f0	[23:20]	• dmp_TexEnv[4].srcAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0f0	[27:24]	• dmp_TexEnv[4].srcAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0f1	[3:0]	• dmp_TexEnv[4].operandRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0f1	[7:4]	• dmp_TexEnv[4].operandRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0f1	[11:8]	• dmp_TexEnv[4].operandRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0f1	[14:12]	• dmp_TexEnv[4].operandAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0f1	[18:16]	• dmp_TexEnv[4].operandAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0f1	[22:20]	• dmp_TexEnv[4].operandAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0f2	[3:0]	• dmp_TexEnv[4].combineRgb	• NN_GX_STATE_FSUNIFORM
0x0f2	[19:16]	• dmp_TexEnv[4].combineAlpha	• NN_GX_STATE_FSUNIFORM
0x0f3	[7:0]	• dmp_TexEnv[4].constRgba (1st component)	• NN_GX_STATE_FSUNIFORM
0x0f3	[15:8]	• dmp_TexEnv[4].constRgba (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0f3	[23:16]	• dmp_TexEnv[4].constRgba (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0f3	[31:24]	• dmp_TexEnv[4].constRgba (4th component)	• NN_GX_STATE_FSUNIFORM
0x0f4	[1:0]	• dmp_TexEnv[4].scaleRgb	• NN_GX_STATE_FSUNIFORM
0x0f4	[17:16]	• dmp_TexEnv[4].scaleAlpha	• NN_GX_STATE_FSUNIFORM
0x0f8	[3:0]	• dmp_TexEnv[5].srcRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0f8	[7:4]	• dmp_TexEnv[5].srcRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0f8	[11:8]	• dmp_TexEnv[5].srcRgb (3rd component)	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x0f8	[19:16]	• dmp_TexEnv[5].srcAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0f8	[23:20]	• dmp_TexEnv[5].srcAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0f8	[27:24]	• dmp_TexEnv[5].srcAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0f9	[3:0]	• dmp_TexEnv[5].operandRgb (1st component)	• NN_GX_STATE_FSUNIFORM
0x0f9	[7:4]	• dmp_TexEnv[5].operandRgb (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0f9	[11:8]	• dmp_TexEnv[5].operandRgb (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0f9	[14:12]	• dmp_TexEnv[5].operandAlpha (1st component)	• NN_GX_STATE_FSUNIFORM
0x0f9	[18:16]	• dmp_TexEnv[5].operandAlpha (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0f9	[22:20]	• dmp_TexEnv[5].operandAlpha (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0fa	[3:0]	• dmp_TexEnv[5].combineRgb	• NN_GX_STATE_FSUNIFORM
0x0fa	[19:16]	• dmp_TexEnv[5].combineAlpha	• NN_GX_STATE_FSUNIFORM
0x0fb	[7:0]	• dmp_TexEnv[5].constRgba (1st component)	• NN_GX_STATE_FSUNIFORM
0x0fb	[15:8]	• dmp_TexEnv[5].constRgba (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0fb	[23:16]	• dmp_TexEnv[5].constRgba (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0fb	[31:24]	• dmp_TexEnv[5].constRgba (4th component)	• NN_GX_STATE_FSUNIFORM
0x0fc	[1:0]	• dmp_TexEnv[5].scaleRgb	• NN_GX_STATE_FSUNIFORM
0x0fc	[17:16]	• dmp_TexEnv[5].scaleAlpha	• NN_GX_STATE_FSUNIFORM
0x0fd	[7:0]	• dmp_TexEnv[0].bufferColor (1st component)	• NN_GX_STATE_FSUNIFORM
0x0fd	[15:8]	• dmp_TexEnv[0].bufferColor (2nd component)	• NN_GX_STATE_FSUNIFORM
0x0fd	[23:16]	• dmp_TexEnv[0].bufferColor (3rd component)	• NN_GX_STATE_FSUNIFORM
0x0fd	[31:24]	• dmp_TexEnv[0].bufferColor (4th component)	• NN_GX_STATE_FSUNIFORM
0x100	[1:0]	• dmp_FragOperation.mode	• NN_GX_STATE_FSUNIFORM
0x100	[8:8]	• **glDisable**(***GL_BLEND***) • **glDisable**(***GL_COLOR_LOGIC_OP***) • **glEnable**(***GL_BLEND***) • **glEnable**(***GL_COLOR_LOGIC_OP***)	• NN_GX_STATE_OTHERS
0x101	[2:0]	• ***mode*** in **glBlendEquation** • ***modeRGB*** in **glBlendEquationSeparate**	• NN_GX_STATE_OTHERS

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x101	[10:8]	• *mode* in **glBlendEquation** • *modeAlpha* in **glBlendEquationSeparate**	• NN_GX_STATE_OTHERS
0x101	[19:16]	• *sfactor* in **glBlendFunc** • *srcRGB* in **glBlendFuncSeparate**	• NN_GX_STATE_OTHERS
0x101	[23:20]	• *dfactor* in **glBlendFunc** • *dstRGB* in **glBlendFuncSeparate**	• NN_GX_STATE_OTHERS
0x101	[27:24]	• *sfactor* in **glBlendFunc** • *srcAlpha* in **glBlendFuncSeparate**	• NN_GX_STATE_OTHERS
0x101	[31:28]	• *dfactor* in **glBlendFunc** • *dstAlpha* in **glBlendFuncSeparate**	• NN_GX_STATE_OTHERS
0x102	[3:0]	• **glLogicOp**	• NN_GX_STATE_OTHERS
0x103	[7:0]	• *red* in **glBlendColor**	• NN_GX_STATE_OTHERS
0x103	[15:8]	• *green* in **glBlendColor**	• NN_GX_STATE_OTHERS
0x103	[23:16]	• *blue* in **glBlendColor**	• NN_GX_STATE_OTHERS
0x103	[31:24]	• *alpha* in **glBlendColor**	• NN_GX_STATE_OTHERS
0x104	[0:0]	• dmp_FragOperation.enableAlphaTest	• NN_GX_STATE_OTHERS
0x104	[6:4]	• dmp_FragOperation.alphaTestFunc	• NN_GX_STATE_OTHERS
0x104	[15:8]	• dmp_FragOperation.alphaRefValue	• NN_GX_STATE_OTHERS
0x105	[0:0]	• **glDisable**(*GL_STENCIL_TEST*) • **glEnable**(*GL_STENCIL_TEST*)	• NN_GX_STATE_OTHERS
0x105	[6:4]	• *func* in **glStencilFunc**	• NN_GX_STATE_OTHERS
0x105	[15:8]	• **glStencilMask**	• NN_GX_STATE_OTHERS
0x105	[23:16]	• *ref* in **glStencilFunc**	• NN_GX_STATE_OTHERS
0x105	[31:24]	• *mask* in **glStencilFunc**	• NN_GX_STATE_OTHERS
0x106	[2:0]	• *fail* in **glStencilOp**	• NN_GX_STATE_OTHERS
0x106	[6:4]	• *zfail* in **glStencilOp**	• NN_GX_STATE_OTHERS
0x106	[10:8]	• *zpass* in **glStencilOp**	• NN_GX_STATE_OTHERS
0x107	[0:0]	• **glDisable**(*GL_DEPTH_TEST*) • **glEnable**(*GL_DEPTH_TEST*)	• NN_GX_STATE_OTHERS
0x107	[6:4]	• **glDepthFunc**	• NN_GX_STATE_OTHERS
0x107	[8:8]	• *red* in **glColorMask**	• NN_GX_STATE_OTHERS
0x107	[9:9]	• *green* in **glColorMask**	• NN_GX_STATE_OTHERS

Register Setting	Bits	Related Functions and Uniforms	State Flags
`0x107`	[10:10]	• ***blue*** in **`glColorMask`**	• `NN_GX_STATE_OTHERS`
`0x107`	[11:11]	• ***alpha*** in **`glColorMask`**	• `NN_GX_STATE_OTHERS`
`0x107`	[12:12]	• **`glDepthMask`**	• `NN_GX_STATE_OTHERS`
`0x110`	[0:0]	• **`glFinish`** • **`glFlush`** • **`nngxSplitDrawCmdlist`** • **`nngxTransferRenderImage`**	• `NN_GX_STATE_FRAMEBUFFER` • `NN_GX_STATE_FBACCESS`
`0x111`	[0:0]	• **`glFinish`** • **`glFlush`** • **`glDrawArrays`** • **`glDrawElements`** • **`nngxSplitDrawCmdlist`** • **`nngxTransferRenderImage`**	• `NN_GX_STATE_FRAMEBUFFER` • `NN_GX_STATE_FBACCESS`
`0x112`	[3:0]	• `dmp_FragOperation.mode` • **`glDisable`**(***GL_BLEND***) • **`glDisable`**(***GL_COLOR_LOGIC_OP***) • **`glEnable`**(***GL_BLEND***) • **`glEnable`**(***GL_COLOR_LOGIC_OP***) • **`glColorMask`**	• `NN_GX_STATE_FBACCESS`
`0x113`	[3:0]	• `dmp_FragOperation.mode` • **`glColorMask`**	• `NN_GX_STATE_FBACCESS`
`0x114`	[1:0]	• `dmp_FragOperation.mode` • **`glDisable`**(***GL_DEPTH_TEST***) • **`glDisable`**(***GL_STENCIL_TEST***) • **`glEnable`**(***GL_DEPTH_TEST***) • **`glEnable`**(***GL_STENCIL_TEST***)	• `NN_GX_STATE_FBACCESS`
`0x115`	[1:0]	• `dmp_FragOperation.mode` • **`glDisable`**(***GL_DEPTH_TEST***) • **`glDisable`**(***GL_STENCIL_TEST***) • **`glEnable`**(***GL_DEPTH_TEST***) • **`glEnable`**(***GL_STENCIL_TEST***) • **`glDepthMask`** • **`glStencilMask`**	• `NN_GX_STATE_FBACCESS`
`0x116`	[1:0]	• ***internalformat*** in **`glRenderbufferStorage`** for the depth buffer that is the rendering target	• `NN_GX_STATE_FRAMEBUFFER`
`0x117`	[1:0]	• ***internalformat*** in **`glRenderbufferStorage`** for the color buffer that is the rendering target	• `NN_GX_STATE_FRAMEBUFFER`
	[18:16]	• ***internalformat*** in **`glTexture2DImage2D`** for the color buffer that is the rendering target	

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x118	[0:0]	• **glDisable**(***GL_EARLY_DEPTH_TEST_DMP***) • **glEnable**(***GL_EARLY_DEPTH_TEST_DMP***)	• NN_GX_STATE_OTHERS
0x11b	[0:0]	• **glRenderBlockModeDMP**	• NN_GX_STATE_OTHERS
0x11c	[27:0]	• Render buffer address allocated by **glRenderbufferStorage** for the depth buffer that is the rendering target	• NN_GX_STATE_FRAMEBUFFER
0x11d	[27:0]	• Render buffer address allocated by **glRenderbufferStorage** for the color buffer that is the rendering target • Texture address allocated by **glTexImage2D**	• NN_GX_STATE_FRAMEBUFFER
0x11e	[10:0]	• ***width*** in **glRenderbufferStorage** for the color buffer that is the rendering target • ***width*** in **glTexture2DImage2D** for the color buffer that is the rendering target	• NN_GX_STATE_FRAMEBUFFER
0x11e	[21:12]	• ***height*** in **glRenderbufferStorage** for the color buffer that is the rendering target • ***height*** in **glTexture2DImage2D** for the color buffer that is the rendering target	• NN_GX_STATE_FRAMEBUFFER
0x120	[7:0]	• dmp_Gas.lightXY (1st component)	• NN_GX_STATE_FSUNIFORM
0x120	[15:8]	• dmp_Gas.lightXY (2nd component)	• NN_GX_STATE_FSUNIFORM
0x120	[23:16]	• dmp_Gas.lightXY (3rd component)	• NN_GX_STATE_FSUNIFORM
0x121	[7:0]	• dmp_Gas.lightZ (1st component)	• NN_GX_STATE_FSUNIFORM
0x121	[15:8]	• dmp_Gas.lightZ (2nd component)	• NN_GX_STATE_FSUNIFORM
0x121	[23:16]	• dmp_Gas.lightZ (3rd component)	• NN_GX_STATE_FSUNIFORM
0x122	[7:0]	• dmp_Gas.lightZ (4th component)	• NN_GX_STATE_FSUNIFORM
0x123	[15:0]	• dmp_Gas.sampler{TR,TG,TB} • LUT object data created by **glTexImage1D**	• NN_GX_STATE_LUT
0x124	[31:0]		
0x125	[31:0]	• dmp_Gas.autoAcc	-
0x126	[23:0]	• dmp_Gas.deltaZ	• NN_GX_STATE_FSUNIFORM
0x126	[25:24]	• **glDepthFunc**	• NN_GX_STATE_OTHERS
0x130	[15:0]	• dmp_FragOperation.penumbraScale • dmp_FragOperation.penumbraBias	• NN_GX_STATE_FSUNIFORM
0x130	[31:16]	• dmp_FragOperation.penumbraScale	• NN_GX_STATE_FSUNIFORM
0x140	[29:0]	• dmp_FragmentMaterial.specular0 • dmp_FragmentLightSource[0].specular0	• NN_GX_STATE_FSUNIFORM
0x141	[29:0]	• dmp_LightEnv.lutEnabledRefl	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
		• dmp_FragmentMaterial.specular1 • dmp_FragmentLightSource[0].specular1	
0x142	[29:0]	• dmp_FragmentMaterial.diffuse • dmp_FragmentLightSource[0].diffuse	• NN_GX_STATE_FSUNIFORM
0x143	[29:0]	• dmp_FragmentMaterial.ambient • dmp_FragmentLightSource[0].ambient	• NN_GX_STATE_FSUNIFORM
0x144	[31:0]	• dmp_FragmentLightSource[0].position	• NN_GX_STATE_FSUNIFORM
0x145	[15:0]		
0x146	[12:0]	• dmp_FragmentLightSource[0]. spotDirection	• NN_GX_STATE_FSUNIFORM
	[28:16]		
0x147	[12:0]		
0x149	[0:0]	• dmp_FragmentLightSource[0].position	• NN_GX_STATE_FSUNIFORM
0x149	[1:1]	• dmp_FragmentLightSource[0]. twoSideDiffuse	• NN_GX_STATE_FSUNIFORM
0x149	[2:2]	• dmp_FragmentLightSource[0]. geomFactor0	• NN_GX_STATE_FSUNIFORM
0x149	[3:3]	• dmp_FragmentLightSource[0]. geomFactor1	• NN_GX_STATE_FSUNIFORM
0x14a	[19:0]	• dmp_FragmentLightSource[0]. distanceAttenuationBias	• NN_GX_STATE_FSUNIFORM
0x14b	[19:0]	• dmp_FragmentLightSource[0]. distanceAttenuationScale	• NN_GX_STATE_FSUNIFORM
0x150	[29:0]	• dmp_FragmentMaterial.specular0 • dmp_FragmentLightSource[1].specular0	• NN_GX_STATE_FSUNIFORM
0x151	[29:0]	• dmp_LightEnv.lutEnabledRefl • dmp_FragmentMaterial.specular1 • dmp_FragmentLightSource[1].specular1	• NN_GX_STATE_FSUNIFORM
0x152	[29:0]	• dmp_FragmentMaterial.diffuse • dmp_FragmentLightSource[1].diffuse	• NN_GX_STATE_FSUNIFORM
0x153	[29:0]	• dmp_FragmentMaterial.ambient • dmp_FragmentLightSource[1].ambient	• NN_GX_STATE_FSUNIFORM
0x154	[31:0]	• dmp_FragmentLightSource[1].position	• NN_GX_STATE_FSUNIFORM
0x155	[15:0]		

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x156	[12:0]	• dmp_FragmentLightSource[1]. spotDirection	• NN_GX_STATE_FSUNIFORM
	[28:16]		
0x157	[12:0]		
0x159	[0:0]	• dmp_FragmentLightSource[1].position	• NN_GX_STATE_FSUNIFORM
0x159	[1:1]	• dmp_FragmentLightSource[1]. twoSideDiffuse	• NN_GX_STATE_FSUNIFORM
0x159	[2:2]	• dmp_FragmentLightSource[1]. geomFactor0	• NN_GX_STATE_FSUNIFORM
0x159	[3:3]	• dmp_FragmentLightSource[1]. geomFactor1	• NN_GX_STATE_FSUNIFORM
0x15a	[19:0]	• dmp_FragmentLightSource[1]. distanceAttenuationBias	• NN_GX_STATE_FSUNIFORM
0x15b	[19:0]	• dmp_FragmentLightSource[1]. distanceAttenuationScale	• NN_GX_STATE_FSUNIFORM
0x160	[29:0]	• dmp_FragmentMaterial.specular0 • dmp_FragmentLightSource[2].specular0	• NN_GX_STATE_FSUNIFORM
0x161	[29:0]	• dmp_LightEnv.lutEnabledRefl • dmp_FragmentMaterial.specular1 • dmp_FragmentLightSource[2].specular1	• NN_GX_STATE_FSUNIFORM
0x162	[29:0]	• dmp_FragmentMaterial.diffuse • dmp_FragmentLightSource[2].diffuse	• NN_GX_STATE_FSUNIFORM
0x163	[29:0]	• dmp_FragmentMaterial.ambient • dmp_FragmentLightSource[2].ambient	• NN_GX_STATE_FSUNIFORM
0x164	[31:0]	• dmp_FragmentLightSource[2].position	• NN_GX_STATE_FSUNIFORM
0x165	[15:0]		
0x166	[12:0]	• dmp_FragmentLightSource[2]. spotDirection	• NN_GX_STATE_FSUNIFORM
	[28:16]		
0x167	[12:0]		
0x169	[0:0]	• dmp_FragmentLightSource[2].position	• NN_GX_STATE_FSUNIFORM
0x169	[1:1]	• dmp_FragmentLightSource[2]. twoSideDiffuse	• NN_GX_STATE_FSUNIFORM
0x169	[2:2]	• dmp_FragmentLightSource[2]. geomFactor0	• NN_GX_STATE_FSUNIFORM
0x169	[3:3]	• dmp_FragmentLightSource[2]. geomFactor1	• NN_GX_STATE_FSUNIFORM
0x16a	[19:0]	• dmp_FragmentLightSource[2].	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
		distanceAttenuationBias	
0x16b	[19:0]	• dmp_FragmentLightSource[2].distanceAttenuationScale	• NN_GX_STATE_FSUNIFORM
0x170	[29:0]	• dmp_FragmentMaterial.specular0 • dmp_FragmentLightSource[3].specular0	• NN_GX_STATE_FSUNIFORM
0x171	[29:0]	• dmp_LightEnv.lutEnabledRefl • dmp_FragmentMaterial.specular1 • dmp_FragmentLightSource[3].specular1	• NN_GX_STATE_FSUNIFORM
0x172	[29:0]	• dmp_FragmentMaterial.diffuse • dmp_FragmentLightSource[3].diffuse	• NN_GX_STATE_FSUNIFORM
0x173	[29:0]	• dmp_FragmentMaterial.ambient • dmp_FragmentLightSource[3].ambient	• NN_GX_STATE_FSUNIFORM
0x174	[31:0]	• dmp_FragmentLightSource[3].position	• NN_GX_STATE_FSUNIFORM
0x175	[15:0]		
0x176	[12:0]		
	[28:16]	• dmp_FragmentLightSource[3].spotDirection	• NN_GX_STATE_FSUNIFORM
0x177	[12:0]		
0x179	[0:0]	• dmp_FragmentLightSource[3].position	• NN_GX_STATE_FSUNIFORM
0x179	[1:1]	• dmp_FragmentLightSource[3].twoSideDiffuse	• NN_GX_STATE_FSUNIFORM
0x179	[2:2]	• dmp_FragmentLightSource[3].geomFactor0	• NN_GX_STATE_FSUNIFORM
0x179	[3:3]	• dmp_FragmentLightSource[3].geomFactor1	• NN_GX_STATE_FSUNIFORM
0x17a	[19:0]	• dmp_FragmentLightSource[3].distanceAttenuationBias	• NN_GX_STATE_FSUNIFORM
0x17b	[19:0]	• dmp_FragmentLightSource[3].distanceAttenuationScale	• NN_GX_STATE_FSUNIFORM
0x180	[29:0]	• dmp_FragmentMaterial.specular0 • dmp_FragmentLightSource[4].specular0	• NN_GX_STATE_FSUNIFORM
0x181	[29:0]	• dmp_LightEnv.lutEnabledRefl • dmp_FragmentMaterial.specular1 • dmp_FragmentLightSource[4].specular1	• NN_GX_STATE_FSUNIFORM
0x182	[29:0]	• dmp_FragmentMaterial.diffuse • dmp_FragmentLightSource[4].diffuse	• NN_GX_STATE_FSUNIFORM
0x183	[29:0]	• dmp_FragmentMaterial.ambient	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
		• dmp_FragmentLightSource[4].ambient	
0x184	[31:0]	• dmp_FragmentLightSource[4].position	• NN_GX_STATE_FSUNIFORM
0x185	[15:0]		
0x186	[12:0]	• dmp_FragmentLightSource[4]. spotDirection	• NN_GX_STATE_FSUNIFORM
	[28:16]		
0x187	[12:0]		
0x189	[0:0]	• dmp_FragmentLightSource[4].position	• NN_GX_STATE_FSUNIFORM
0x189	[1:1]	• dmp_FragmentLightSource[4]. twoSideDiffuse	• NN_GX_STATE_FSUNIFORM
0x189	[2:2]	• dmp_FragmentLightSource[4]. geomFactor0	• NN_GX_STATE_FSUNIFORM
0x189	[3:3]	• dmp_FragmentLightSource[4]. geomFactor1	• NN_GX_STATE_FSUNIFORM
0x18a	[19:0]	• dmp_FragmentLightSource[4]. distanceAttenuationBias	• NN_GX_STATE_FSUNIFORM
0x18b	[19:0]	• dmp_FragmentLightSource[4]. distanceAttenuationScale	• NN_GX_STATE_FSUNIFORM
0x190	[29:0]	• dmp_FragmentMaterial.specular0 • dmp_FragmentLightSource[5].specular0	• NN_GX_STATE_FSUNIFORM
0x191	[29:0]	• dmp_LightEnv.lutEnabledRefl • dmp_FragmentMaterial.specular1 • dmp_FragmentLightSource[5].specular1	• NN_GX_STATE_FSUNIFORM
0x192	[29:0]	• dmp_FragmentMaterial.diffuse • dmp_FragmentLightSource[5].diffuse	• NN_GX_STATE_FSUNIFORM
0x193	[29:0]	• dmp_FragmentMaterial.ambient • dmp_FragmentLightSource[5].ambient	• NN_GX_STATE_FSUNIFORM
0x194	[31:0]	• dmp_FragmentLightSource[5].position	• NN_GX_STATE_FSUNIFORM
0x195	[15:0]		
0x196	[12:0]	• dmp_FragmentLightSource[5]. spotDirection	• NN_GX_STATE_FSUNIFORM
	[28:16]		
0x197	[12:0]		
0x199	[0:0]	• dmp_FragmentLightSource[5].position	• NN_GX_STATE_FSUNIFORM
0x199	[1:1]	• dmp_FragmentLightSource[5]. twoSideDiffuse	• NN_GX_STATE_FSUNIFORM
0x199	[2:2]	• dmp_FragmentLightSource[5].	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
		geomFactor0	
0x199	[3:3]	• dmp_FragmentLightSource[5].geomFactor1	• NN_GX_STATE_FSUNIFORM
0x19a	[19:0]	• dmp_FragmentLightSource[5].distanceAttenuationBias	• NN_GX_STATE_FSUNIFORM
0x19b	[19:0]	• dmp_FragmentLightSource[5].distanceAttenuationScale	• NN_GX_STATE_FSUNIFORM
0x1a0	[29:0]	• dmp_FragmentMaterial.specular0 • dmp_FragmentLightSource[6].specular0	• NN_GX_STATE_FSUNIFORM
0x1a1	[29:0]	• dmp_LightEnv.lutEnabledRefl • dmp_FragmentMaterial.specular1 • dmp_FragmentLightSource[6].specular1	• NN_GX_STATE_FSUNIFORM
0x1a2	[29:0]	• dmp_FragmentMaterial.diffuse • dmp_FragmentLightSource[6].diffuse	• NN_GX_STATE_FSUNIFORM
0x1a3	[29:0]	• dmp_FragmentMaterial.ambient • dmp_FragmentLightSource[6].ambient	• NN_GX_STATE_FSUNIFORM
0x1a4	[31:0]	• dmp_FragmentLightSource[6].position	• NN_GX_STATE_FSUNIFORM
0x1a5	[15:0]		
0x1a6	[12:0]	• dmp_FragmentLightSource[6].spotDirection	• NN_GX_STATE_FSUNIFORM
	[28:16]		
0x1a7	[12:0]		
0x1a9	[0:0]	• dmp_FragmentLightSource[6].position	• NN_GX_STATE_FSUNIFORM
0x1a9	[1:1]	• dmp_FragmentLightSource[6].twoSideDiffuse	• NN_GX_STATE_FSUNIFORM
0x1a9	[2:2]	• dmp_FragmentLightSource[6].geomFactor0	• NN_GX_STATE_FSUNIFORM
0x1a9	[3:3]	• dmp_FragmentLightSource[6].geomFactor1	• NN_GX_STATE_FSUNIFORM
0x1aa	[19:0]	• dmp_FragmentLightSource[6].distanceAttenuationBias	• NN_GX_STATE_FSUNIFORM
0x1ab	[19:0]	• dmp_FragmentLightSource[6].distanceAttenuationScale	• NN_GX_STATE_FSUNIFORM
0x1b0	[29:0]	• dmp_FragmentMaterial.specular0 • dmp_FragmentLightSource[7].specular0	• NN_GX_STATE_FSUNIFORM
0x1b1	[29:0]	• dmp_LightEnv.lutEnabledRefl • dmp_FragmentMaterial.specular1 • dmp_FragmentLightSource[7].specular1	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x1b2	[29:0]	• dmp_FragmentMaterial.diffuse • dmp_FragmentLightSource[7].diffuse	• NN_GX_STATE_FSUNIFORM
0x1b3	[29:0]	• dmp_FragmentMaterial.ambient • dmp_FragmentLightSource[7].ambient	• NN_GX_STATE_FSUNIFORM
0x1b4	[31:0]	• dmp_FragmentLightSource[7].position	• NN_GX_STATE_FSUNIFORM
0x1b5	[15:0]		
0x1b6	[12:0]	• dmp_FragmentLightSource[7]. spotDirection	• NN_GX_STATE_FSUNIFORM
	[28:16]		
0x1b7	[12:0]		
0x1b9	[0:0]	• dmp_FragmentLightSource[7].position	• NN_GX_STATE_FSUNIFORM
0x1b9	[1:1]	• dmp_FragmentLightSource[7]. twoSideDiffuse	• NN_GX_STATE_FSUNIFORM
0x1b9	[2:2]	• dmp_FragmentLightSource[7]. geomFactor0	• NN_GX_STATE_FSUNIFORM
0x1b9	[3:3]	• dmp_FragmentLightSource[7]. geomFactor1	• NN_GX_STATE_FSUNIFORM
0x1ba	[19:0]	• dmp_FragmentLightSource[7]. distanceAttenuationBias	• NN_GX_STATE_FSUNIFORM
0x1bb	[19:0]	• dmp_FragmentLightSource[7]. distanceAttenuationScale	• NN_GX_STATE_FSUNIFORM
0x1c0	[29:0]	• dmp_FragmentLighting.ambient • dmp_FragmentMaterial.ambient • dmp_FragmentMaterial.emission	• NN_GX_STATE_FSUNIFORM
0x1c2	[2:0]	• dmp_FragmentLightSource[i].enabled	• NN_GX_STATE_FSUNIFORM
0x1c3	[0:0]	• dmp_LightEnv.shadowPrimary • dmp_LightEnv.shadowSecondary • dmp_LightEnv.shadowAlpha	• NN_GX_STATE_FSUNIFORM
0x1c3	[3:2]	• dmp_LightEnv.fresnelSelector	• NN_GX_STATE_FSUNIFORM
0x1c3	[7:4]	• dmp_LightEnv.config	• NN_GX_STATE_FSUNIFORM
0x1c3	[16:16]	• dmp_LightEnv.shadowPrimary	• NN_GX_STATE_FSUNIFORM
0x1c3	[17:17]	• dmp_LightEnv.shadowSecondary	• NN_GX_STATE_FSUNIFORM
0x1c3	[18:18]	• dmp_LightEnv.invertShadow	• NN_GX_STATE_FSUNIFORM
0x1c3	[19:19]	• dmp_LightEnv.shadowAlpha	• NN_GX_STATE_FSUNIFORM
0x1c3	[23:22]	• dmp_LightEnv.bumpSelector	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x1c3	[25:24]	• dmp_LightEnv.shadowSelector	• NN_GX_STATE_FSUNIFORM
0x1c3	[27:27]	• dmp_LightEnv.clampHighlights	• NN_GX_STATE_FSUNIFORM
0x1c3	[29:28]	• dmp_LightEnv.bumpMode	• NN_GX_STATE_FSUNIFORM
0x1c3	[30:30]	• dmp_LightEnv.bumpMode • dmp_LightEnv.bumpRenorm	• NN_GX_STATE_FSUNIFORM
0x1c4	[0:0]	• dmp_FragmentLightSource[0].shadowed	• NN_GX_STATE_FSUNIFORM
0x1c4	[1:1]	• dmp_FragmentLightSource[1].shadowed	• NN_GX_STATE_FSUNIFORM
0x1c4	[2:2]	• dmp_FragmentLightSource[2].shadowed	• NN_GX_STATE_FSUNIFORM
0x1c4	[3:3]	• dmp_FragmentLightSource[3].shadowed	• NN_GX_STATE_FSUNIFORM
0x1c4	[4:4]	• dmp_FragmentLightSource[4].shadowed	• NN_GX_STATE_FSUNIFORM
0x1c4	[5:5]	• dmp_FragmentLightSource[5].shadowed	• NN_GX_STATE_FSUNIFORM
0x1c4	[6:6]	• dmp_FragmentLightSource[6].shadowed	• NN_GX_STATE_FSUNIFORM
0x1c4	[7:7]	• dmp_FragmentLightSource[7].shadowed	• NN_GX_STATE_FSUNIFORM
0x1c4	[8:8]	• dmp_FragmentLightSource[0].spotEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[9:9]	• dmp_FragmentLightSource[1].spotEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[10:10]	• dmp_FragmentLightSource[2].spotEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[11:11]	• dmp_FragmentLightSource[3].spotEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[12:12]	• dmp_FragmentLightSource[4].spotEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[13:13]	• dmp_FragmentLightSource[5].spotEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[14:14]	• dmp_FragmentLightSource[6].spotEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[15:15]	• dmp_FragmentLightSource[7].spotEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[16:16]	• dmp_LightEnv.lutEnabledD0	• NN_GX_STATE_FSUNIFORM
0x1c4	[17:17]	• dmp_LightEnv.lutEnabledD1	• NN_GX_STATE_FSUNIFORM
0x1c4	[19:19]	• dmp_LightEnv.fresnelSelector	• NN_GX_STATE_FSUNIFORM
0x1c4	[22:20]	• dmp_LightEnv.lutEnabledRefl	• NN_GX_STATE_FSUNIFORM
0x1c4	[24:24]	• dmp_FragmentLightSource[0].distanceAttenuationEnabled	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x1c4	[25:25]	• dmp_FragmentLightSource[1].distanceAttenuationEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[26:26]	• dmp_FragmentLightSource[2].distanceAttenuationEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[27:27]	• dmp_FragmentLightSource[3].distanceAttenuationEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[28:28]	• dmp_FragmentLightSource[4].distanceAttenuationEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[29:29]	• dmp_FragmentLightSource[5].distanceAttenuationEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[30:30]	• dmp_FragmentLightSource[6].distanceAttenuationEnabled	• NN_GX_STATE_FSUNIFORM
0x1c4	[31:31]	• dmp_FragmentLightSource[7].distanceAttenuationEnabled	• NN_GX_STATE_FSUNIFORM
0x1c5	[7:0] [11:8]	• dmp_FragmentMaterial.sampler{D0,D1,FR,RB,RG,RR} • dmp_FragmentLightSource[i].sampler{SP,DA} • LUT object data created by **glTexImage1D**	• NN_GX_STATE_LUT
0x1c6	[0:0]	• dmp_FragmentLighting.enabled	• FS_STATE_FSUNIFORM
0x1c8–0x1cf	[23:0]	• dmp_FragmentMaterial.sampler{D0,D1,FR,RB,RG,RR} • dmp_FragmentLightSource[i].sampler{SP,DA} • LUT object data created by **glTexImage1D**	• NN_GX_STATE_LUT
0x1d0	[1:1]	• dmp_LightEnv.absLutInputD0	• NN_GX_STATE_FSUNIFORM
0x1d0	[5:5]	• dmp_LightEnv.absLutInputD1	• NN_GX_STATE_FSUNIFORM
0x1d0	[9:9]	• dmp_LightEnv.absLutInputSP	• NN_GX_STATE_FSUNIFORM
0x1d0	[13:13]	• dmp_LightEnv.absLutInputFR	• NN_GX_STATE_FSUNIFORM
0x1d0	[17:17]	• dmp_LightEnv.absLutInputRB	• NN_GX_STATE_FSUNIFORM
0x1d0	[21:21]	• dmp_LightEnv.absLutInputRG	• NN_GX_STATE_FSUNIFORM
0x1d0	[25:25]	• dmp_LightEnv.absLutInputRR	• NN_GX_STATE_FSUNIFORM
0x1d1	[2:0]	• dmp_LightEnv.lutInputD0	• NN_GX_STATE_FSUNIFORM
0x1d1	[6:4]	• dmp_LightEnv.lutInputD1	• NN_GX_STATE_FSUNIFORM
0x1d1	[10:8]	• dmp_LightEnv.lutInputSP	• NN_GX_STATE_FSUNIFORM
0x1d1	[14:12]	• dmp_LightEnv.lutInputFR	• NN_GX_STATE_FSUNIFORM
0x1d1	[18:16]	• dmp_LightEnv.lutInputRB	• NN_GX_STATE_FSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
`0x1d1`	[22:20]	• `dmp_LightEnv.lutInputRG`	• `NN_GX_STATE_FSUNIFORM`
`0x1d1`	[26:24]	• `dmp_LightEnv.lutInputRR`	• `NN_GX_STATE_FSUNIFORM`
`0x1d2`	[2:0]	• `dmp_LightEnv.lutScaleD0`	• `NN_GX_STATE_FSUNIFORM`
`0x1d2`	[6:4]	• `dmp_LightEnv.lutScaleD1`	• `NN_GX_STATE_FSUNIFORM`
`0x1d2`	[10:8]	• `dmp_LightEnv.lutScaleSP`	• `NN_GX_STATE_FSUNIFORM`
`0x1d2`	[14:12]	• `dmp_LightEnv.lutScaleFR`	• `NN_GX_STATE_FSUNIFORM`
`0x1d2`	[18:16]	• `dmp_LightEnv.lutScaleRB`	• `NN_GX_STATE_FSUNIFORM`
`0x1d2`	[22:20]	• `dmp_LightEnv.lutScaleRG`	• `NN_GX_STATE_FSUNIFORM`
`0x1d2`	[26:24]	• `dmp_LightEnv.lutScaleRR`	• `NN_GX_STATE_FSUNIFORM`
`0x1d9`	[2:0] [6:4] [10:8] [14:12] [18:16] [22:20] [26:24] [30:28]	• `dmp_FragmentLightSource[i].enabled`	• `NN_GX_STATE_FSUNIFORM`
`0x200`	[28:1]	• Vertex buffer address allocated by **`glBufferData`**	• `NN_GX_STATE_VERTEX`
`0x201`	[31:0]	• ***size*** and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
`0x202`	[15:0]		
`0x202`	[27:16]	• **`glEnableVertexAttribArray`** • **`glDIsableVertexAttribArray`** • **`glUseProgram`**	• `NN_GX_STATE_VERTEX`
`0x202`	[31:28]	• **`glUseProgram`**	• `NN_GX_STATE_VERTEX`
`0x203`	[27:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
`0x204`	[31:0]		
`0x205`	[15:0] [23:16] [31:28]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`

Register Setting	Bits	Related Functions and Uniforms	State Flags
`0x206`	[27:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
`0x207`	[31:0]		
`0x208`	[15:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
	[23:16]		
	[31:28]		
`0x209`	[27:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
`0x20a`	[31:0]		
`0x20b`	[15:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
	[23:16]		
	[31:28]		
`0x20c`	[27:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
`0x20d`	[31:0]		
`0x20e`	[15:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
	[23:16]		
	[31:28]		
`0x20f`	[27:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
`0x210`	[31:0]		
`0x211`	[15:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
	[23:16]		
	[31:28]		
`0x212`	[27:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
`0x213`	[31:0]		
`0x214`	[15:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in **`glVertexAttribPointer`**	• `NN_GX_STATE_VERTEX`
	[23:16]		
	[31:28]		
`0x215`	[27:0]	• Vertex buffer address allocated by **`glBufferData`** • ***ptr***, ***stride***, ***size***, and ***type*** in	• `NN_GX_STATE_VERTEX`
`0x216`	[31:0]		

Register Setting	Bits	Related Functions and Uniforms	State Flags
		`glVertexAttribPointer`	
0x217	[15:0] [23:16] [31:28]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x218	[27:0]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x219	[31:0]		
0x21a	[15:0] [23:16] [31:28]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x21b	[27:0]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x21c	[31:0]		
0x21d	[15:0] [23:16] [31:28]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x21e	[27:0]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x21f	[31:0]		
0x220	[15:0] [23:16] [31:28]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x221	[27:0]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x222	[31:0]		
0x223	[15:0] [23:16] [31:28]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x224	[27:0]	• Vertex buffer address allocated by `glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	• NN_GX_STATE_VERTEX
0x225	[31:0]		
0x226	[15:0]	• Vertex buffer address allocated by	• NN_GX_STATE_VERTEX

Register Setting	Bits	Related Functions and Uniforms	State Flags
	[23:16] [31:28]	`glBufferData` • ***ptr***, ***stride***, ***size***, and ***type*** in `glVertexAttribPointer`	
0x227	[27:0]	• Vertex buffer address allocated by `glBufferData` • ***indices*** in `glDrawElements`	-
0x227	[31:31]	• ***type*** in `glDrawElements`	-
0x228	[31:0]	• ***count*** in `glDrawElements` • ***count*** in `glDrawArrays`	-
0x229	[1:0]	• `glUseProgram`	• NN_GX_STATE_SHADERMODE
0x229	[8:8]	• ***mode*** in `glDrawElements`	-
0x229	[31:31]	• `glUseProgram`	• NN_GX_STATE_SHADERPROGRAM
0x22a	[31:0]	• ***first*** in `glDrawArrays`	-
0x22e		• `glDrawArrays`	-
0x22f		• `glDrawElements`	-
0x231		• `glDrawElements` • `glDrawArrays`	-
0x232 0x233 0x234 0x235	[3:0] [31:0] [31:0] [31:0]	• ***ptr*** in `glVertexAttribPointer` • Vertex attribute data content created by `glVertexAttrib{1234}fv` or `glVertexAttrib{1234}f`	• NN_GX_STATE_VERTEX for fixed vertex attribute values when the vertex buffer is used
0x242	[3:0]	• `glUseProgram`	• NN_GX_STATE_SHADERPROGRAM
0x244	[0:0]	• `glUseProgram`	• NN_GX_STATE_SHADERMODE
0x245	[0:0]	• `glDrawElements` • `glDrawArrays`	-
0x24a	[3:0]	• `glUseProgram`	• NN_GX_STATE_SHADERPROGRAM
0x251	[3:0]	• `glUseProgram`	• NN_GX_STATE_SHADERPROGRAM
0x252	[31:0]	• `glUseProgram`	• NN_GX_STATE_SHADERPROGRAM
0x253	[0:0] [8:8]	• `glDrawElements` • `glDrawArrays`	-
0x254	[4:0]	• `glUseProgram`	• NN_GX_STATE_SHADERPROGRAM
0x25e	[3:0]	• `glUseProgram`	• NN_GX_STATE_SHADERPROGRAM

Register Setting	Bits	Related Functions and Uniforms	State Flags
0x25e	[9:8]	• **glDrawElements** • **glDrawArrays**	-
0x25f	[0:0]	• **glDrawElements** • **glDrawArrays**	-
0x280	[15:0]	• **glUseProgram** • **glUniformi**	• NN_GX_STATE_VSUNIFORM • NN_GX_STATE_SHADERMODE
0x281	[23:0]	• **glUseProgram** • **glUniformi**	• NN_GX_STATE_VSUNIFORM • NN_GX_STATE_SHADERMODE
0x282	[23:0]	• **glUseProgram** • **glUniformi**	• NN_GX_STATE_VSUNIFORM • NN_GX_STATE_SHADERMODE
0x283	[23:0]	• **glUseProgram** • **glUniformi**	• NN_GX_STATE_VSUNIFORM • NN_GX_STATE_SHADERMODE
0x284	[23:0]	• **glUseProgram** • **glUniformi**	• NN_GX_STATE_VSUNIFORM • NN_GX_STATE_SHADERMODE
0x289	[3:0] [15:8] [31:24]	• **glUseProgram**	• NN_GX_STATE_SHADERPROGRAM • NN_GX_STATE_SHADERMODE
0x28a	[15:0]	• **glUseProgram**	• NN_GX_STATE_SHADERPROGRAM • NN_GX_STATE_SHADERMODE
0x28b	[31:0]	• **glUseProgram**	• NN_GX_STATE_VERTEX
0x28c	[31:0]	• **glUseProgram**	• NN_GX_STATE_VERTEX
0x28d	[15:0]	• **glUseProgram**	• NN_GX_STATE_SHADERPROGRAM • NN_GX_STATE_SHADERMODE
0x28f		• **glUseProgram**	• NN_GX_STATE_SHADERBINARY
0x290	[7:0] [31:31]	• **glUseProgram** • **glUniformf**	• NN_GX_STATE_SHADERFLOAT • NN_GX_STATE_VSUNIFORM
0x291–0x298	[31:0]	• **glUseProgram** • **glUniformf**	• NN_GX_STATE_SHADERFLOAT • NN_GX_STATE_VSUNIFORM
0x29b	[11:0]	• **glUseProgram**	• NN_GX_STATE_SHADERBINARY
0x29c–0x2a3	[31:0]	• **glUseProgram**	• NN_GX_STATE_SHADERBINARY
0x2a5	[11:0]	• **glUseProgram**	• NN_GX_STATE_SHADERBINARY
0x2a6–0x2ad	[31:0]	• **glUseProgram**	• NN_GX_STATE_SHADERBINARY
0x2b0	[15:0]	• **glUseProgram**	• NN_GX_STATE_VSUNIFORM

Register Setting	Bits	Related Functions and Uniforms	State Flags
		• **glUniformi**	• NN_GX_STATE_SHADERMODE
0x2b1	[23:0]	• **glUseProgram** • **glUniformi**	• NN_GX_STATE_VSUNIFORM • NN_GX_STATE_SHADERMODE
0x2b2	[23:0]	• **glUseProgram** • **glUniformi**	• NN_GX_STATE_VSUNIFORM • NN_GX_STATE_SHADERMODE
0x2b3	[23:0]	• **glUseProgram** • **glUniformi**	• NN_GX_STATE_VSUNIFORM • NN_GX_STATE_SHADERMODE
0x2b4	[23:0]	• **glUseProgram** • **glUniformi**	• NN_GX_STATE_VSUNIFORM • NN_GX_STATE_SHADERMODE
0x2b9	[3:0] [15:8] [31:24]	• **glUseProgram**	• NN_GX_STATE_SHADERPROGRAM • NN_GX_STATE_SHADERMODE
0x2ba	[15:0]	• **glUseProgram**	• NN_GX_STATE_SHADERPROGRAM • NN_GX_STATE_SHADERMODE
0x2bb	[31:0]	• **glUseProgram**	• NN_GX_STATE_VERTEX
0x2bc	[31:0]	• **glUseProgram**	• NN_GX_STATE_VERTEX
0x2bd	[15:0]	• **glUseProgram**	• NN_GX_STATE_SHADERPROGRAM • NN_GX_STATE_SHADERMODE
0x2bf		• **glUseProgram**	• NN_GX_STATE_SHADERBINARY
0x2c0	[7:0] [31:31]	• **glUseProgram** • **glUniformf**	• NN_GX_STATE_SHADERFLOAT • NN_GX_STATE_VSUNIFORM
0x2c1–0x2c8	[31:0]	• **glUseProgram** • **glUniformf**	• NN_GX_STATE_SHADERFLOAT • NN_GX_STATE_VSUNIFORM
0x2cb	[11:0]	• **glUseProgram**	• NN_GX_STATE_SHADERBINARY
0x2cc–0x2d3	[31:0]	• **glUseProgram**	• NN_GX_STATE_SHADERBINARY
0x2d5	[11:0]	• **glUseProgram**	• NN_GX_STATE_SHADERBINARY
0x2d6–0x2dd	[31:0]	• **glUseProgram**	• NN_GX_STATE_SHADERBINARY

6 Error Codes

This chapter lists error codes that may be generated when system functions are called. Use the **`glGetError`** function to get error codes.

Table 6-1 Error Code List

Error Code	Error Generating Function	Error Cause
`GL_ERROR_8000_DMP`	**`nngxGenCmdlists`**	Negative value specified for ***n***.
`GL_ERROR_8001_DMP`	**`nngxGenCmdlists`**	Failed to allocate memory in the management region.
`GL_ERROR_8002_DMP`	**`nngxDeleteCmdlists`**	Negative value specified for ***n***.
`GL_ERROR_8003_DMP`	**`nngxDeleteCmdlists`**	Command list object deleted during execution.
`GL_ERROR_8004_DMP`	**`nngxBindCmdlist`**	Failed to allocate memory in the management region.
`GL_ERROR_8005_DMP`	**`nngxBindCmdlist`**	This API function was called while saving the command list.
`GL_ERROR_8006_DMP`	**`nngxCmdlistStorage`**	Failed to allocate memory for command buffer or command request.
`GL_ERROR_8007_DMP`	**`nngxCmdlistStorage`**	This function was called against the executing command list.
`GL_ERROR_8008_DMP`	**`nngxCmdlistStorage`**	Negative value specified for ***bufsize*** or ***requestcount***.
`GL_ERROR_8009_DMP`	**`nngxRunCmdlist`**	Command buffer and command request memory not allocated for bound command list.
`GL_ERROR_800A_DMP`	**`nngxReserveStopCmdlist`**	This function was called against the executing command list.
`GL_ERROR_800B_DMP`	**`nngxReserveStopCmdlist`**	0, a negative value, or a value greater than the maximum number of command requests specified for ***id***.
`GL_ERROR_800C_DMP`	**`nngxSplitDrawCmdlist`**	0 bound to current command list.
`GL_ERROR_800D_DMP`	**`nngxSplitDrawCmdlist`**	Maximum number of accumulated command requests has been reached.
`GL_ERROR_800E_DMP`	**`nngxSplitDrawCmdlist`**	A command to stop reading 3D commands was added to a 3D command buffer that has finished accumulating, exceeding the maximum command buffer size.
`GL_ERROR_800F_DMP`	**`nngxClearCmdlist`**	This function was called against the executing command list.

Error Code	Error Generating Function	Error Cause
GL_ERROR_8010_DMP	**nngxSetCmdlistCallback**	This function was called against the executing command list.
GL_ERROR_8012_DMP	**nngxEnableCmdlistCallback**	0, a negative value other than -1, or a value greater than the maximum number of command requests specified for ***id***.
GL_ERROR_8014_DMP	**nngxDisableCmdlistCallback**	0, a negative value other than -1, or a value greater than the maximum number of command requests specified for ***id***.
GL_ERROR_8015_DMP	**nngxSetCmdlistParameteri**	This function was called against the executing command list.
GL_ERROR_8016_DMP	**nngxSetCmdlistParameteri**	Invalid values specified for ***pname*** and ***param***.
GL_ERROR_8017_DMP	**nngxGetCmdlistParameteri**	Invalid value specified for ***pname***.
GL_ERROR_8018_DMP	**nngxGetCmdlistParameteri**	The bound command list is 0, and a value other than NX_GX_CMDLIST_BINDING is specified for ***pname***.
GL_ERROR_8019_DMP	**nngxCheckVSync**	Invalid value specified for ***display***.
GL_ERROR_801A_DMP	**nngxWaitVSync**	Invalid value specified for ***display***.
GL_ERROR_801B_DMP	**nngxSetVSyncCallback**	Invalid value specified for ***display***.
GL_ERROR_801C_DMP	**nngxGenDisplaybuffers**	Negative value specified for ***n***.
GL_ERROR_801D_DMP	**nngxGenDisplaybuffers**	Failed to allocate memory in the management region.
GL_ERROR_801E_DMP	**nngxDeleteDisplaybuffers**	Negative value specified for ***n***.
GL_ERROR_801F_DMP	**nngxActiveDisplay**	Invalid value specified for ***display***.
GL_ERROR_8020_DMP	**nngxBindDisplaybuffer**	Failed to allocate memory in the management region.
GL_ERROR_8021_DMP	**nngxDisplaybufferStorage**	0 is bound to the display target.
GL_ERROR_8022_DMP	**nngxDisplaybufferStorage**	Invalid value specified for ***width*** and ***height***.
GL_ERROR_8023_DMP	**nngxDisplaybufferStorage**	Invalid value specified for ***format***.
GL_ERROR_8024_DMP	**nngxDisplaybufferStorage**	Invalid value specified for ***area***.
GL_ERROR_8025_DMP	**nngxDisplaybufferStorage**	Failed to allocate memory for display buffer.
GL_ERROR_8026_DMP	**nngxDisplayEnv**	Negative value specified for ***displayx*** or ***displayy***.
GL_ERROR_8027_DMP	**nngxTransferRenderImage**	0 bound to current command list.

Error Code	Error Generating Function	Error Cause
GL_ERROR_8028_DMP	**nngxTransferRenderImage**	The current command list has already accumulated the maximum number of command requests.
GL_ERROR_8029_DMP	**nngxTransferRenderImage**	Invalid value specified for ***buffer***. Invalid object name, or display buffer memory not allocated.
GL_ERROR_802A_DMP	**nngxTransferRenderImage**	Current color buffer invalid. Render buffer not attached, or render buffer memory not allocated.
GL_ERROR_802B_DMP	**nngxTransferRenderImage**	Invalid value specified for ***mode***.
GL_ERROR_802C_DMP	**nngxTransferRenderImage**	Color buffer resolution lower than the resolution of the transfer destination display buffer.
GL_ERROR_802D_DMP	**nngxTransferRenderImage**	Invalid value specified for ***colorx*** or ***colory***.
GL_ERROR_802E_DMP	**nngxTransferRenderImage**	Pixel size of the transfer destination display buffer is larger than the pixel size of the transfer origin color buffer.
GL_ERROR_802F_DMP	**nngxTransferRenderImage**	No space available in the command buffer, so could not add split command.
GL_ERROR_8030_DMP	**nngxSwapBuffers**	Invalid value specified for ***display***.
GL_ERROR_8031_DMP	**nngxSwapBuffers**	0 bound to current display buffer, or display buffer memory not allocated.
GL_ERROR_8032_DMP	**nngxSwapBuffers**	The display region specified by the **nngxDisplayEnv** function lies outside of the display buffer.
GL_ERROR_8033_DMP	**nngxGetDisplaybufferParameteri**	Invalid value specified for ***pname***.
GL_ERROR_8034_DMP	**nngxStartCmdlistSave**	This function was called again before the previous call to this function finished saving the command.
GL_ERROR_8035_DMP	**nngxStartCmdlistSave**	0 bound to current command list.
GL_ERROR_8036_DMP	**nngxStopCmdlistSave**	Command list save not started.
GL_ERROR_8037_DMP	**nngxUseSavedCmdlist**	0 bound to current command list.
GL_ERROR_8038_DMP	**nngxUseSavedCmdlist**	Invalid object name specified for ***cmdlist***.
GL_ERROR_8039_DMP	**nngxUseSavedCmdlist**	Current command list specified for ***cmdlist***.
GL_ERROR_803A_DMP	**nngxUseSavedCmdlist**	Command was added, exceeding the maximum size of the 3D command buffer or of the command request list.
GL_ERROR_803B_DMP	**nngxExportCmdlist**	Invalid value specified for ***cmdlist***.

Error Code	Error Generating Function	Error Cause
GL_ERROR_803C_DMP	**nngxExportCmdlist**	Value specified for ***datasize*** is smaller than the size of the exported data.
GL_ERROR_803D_DMP	**nngxExportCmdlist**	***bufferoffset***, ***buffersize***, ***requestid***, and ***requestsize*** specify regions for which commands have not been accumulated.
GL_ERROR_803E_DMP	**nngxExportCmdlist**	Values specified for ***bufferoffset*** and ***buffersize*** are not 8-byte aligned.
GL_ERROR_803F_DMP	**nngxExportCmdlist**	Attempted to export 3D execution command that was added with the method that does not copy the 3D command buffer, using the **nngxUseSavedCmdlist** function.
GL_ERROR_8040_DMP	**nngxExportCmdlist**	Values specified for ***bufferoffset*** and ***buffersize*** do not properly specify the 3D command buffer to be executed by the exported 3D execution command.
GL_ERROR_8041_DMP	**nngxImportCmdlist**	Invalid value specified for ***cmdlist***.
GL_ERROR_8042_DMP	**nngxImportCmdlist**	Pointer to invalid data specified for ***data***.
GL_ERROR_8043_DMP	**nngxImportCmdlist**	Value specified for ***datasize*** does not match size of exported data.
GL_ERROR_8044_DMP	**nngxImportCmdlist**	Command was imported, exceeding the maximum size of the 3D command buffer or of the command request list.
GL_ERROR_8045_DMP	**nngxImportCmdlist**	3D execution command was not the first command request imported into a command list's 3D command buffer that has not been split.
GL_ERROR_8046_DMP	**nngxGetExportedCmdlistInfo**	Pointer to invalid data specified for ***data***.
GL_ERROR_8047_DMP	**nngxCopyCmdlist**	Current command list specified for ***dcmdlist***.
GL_ERROR_8048_DMP	**nngxCopyCmdlist**	Invalid value specified for ***scmdlist***.
GL_ERROR_8049_DMP	**nngxCopyCmdlist**	Invalid value specified for ***dcmdlist***.
GL_ERROR_804A_DMP	**nngxCopyCmdlist**	Same value specified for both ***scmdlist*** and ***dcmdlist***.
GL_ERROR_804B_DMP	**nngxCopyCmdlist**	Command list specified for ***dcmdlist*** is currently being executed.
GL_ERROR_804C_DMP	**nngxCopyCmdlist**	Size of the commands accumulated in ***scmdlist*** exceeds the maximum size of the 3D command buffer or of the command request list specified by ***dcmdlist***.
GL_ERROR_804D_DMP	**nngxSetCommandGenerationMode**	Invalid value specified for ***mode***.

Error Code	Error Generating Function	Error Cause
GL_ERROR_804E_DMP	**nngxAdd3DCommand**	0 bound to current command list.
GL_ERROR_804F_DMP	**nngxAdd3DCommand**	Value specified for ***buffersize*** is either 0, or not a multiple of 8.
GL_ERROR_8050_DMP	**nngxAdd3DCommand**	***copycmd*** specifies GL_TRUE, and the size of the 3D command buffer exceeds the maximum.
GL_ERROR_8051_DMP	**nngxAdd3DCommand**	***copycmd*** specifies GL_FALSE, and the size of the 3D command request exceeds the maximum.
GL_ERROR_8052_DMP	**nngxAdd3DCommand**	Value specified for ***bufferaddr*** not a multiple of 16.
GL_ERROR_8053_DMP	**nngxSwapBuffers**	The display buffer address is not 16-byte aligned.
GL_ERROR_8054_DMP	**nngxAddCmdlist**	An invalid value is specified for ***cmdlist***.
GL_ERROR_8055_DMP	**nngxAddCmdlist**	No command list is currently bound.
GL_ERROR_8056_DMP	**nngxAddCmdlist**	***cmdlist*** specifies the current command list.
GL_ERROR_8057_DMP	**nngxAddCmdlist**	The current command list is in the middle of execution.
GL_ERROR_8058_DMP	**nngxAddCmdlist**	There is not enough memory for command buffers or command requests.
GL_ERROR_8059_DMP	**nngxTransferRenderImage**	The 32-block format is set and the transfer's source color buffer or destination display buffer has a width or height that is not a multiple of 32.
GL_ERROR_805A_DMP	**nngxTransferRenderImage**	A color buffer was transferred to a display buffer that uses 24-bit pixels and the 8-block format when either the color buffer or display buffer had a width or height that was not a multiple of 16.
GL_ERROR_805B_DMP	nngxTransferLinearImage	The current command list is bound to 0.
GL_ERROR_805C_DMP	nngxTransferLinearImage	The current command list is has already accumulated the maximum number of command requests.
GL_ERROR_805D_DMP	nngxTransferLinearImage	The current 3D command buffer is of insufficient size.
GL_ERROR_805E_DMP	nngxTransferLinearImage	Either the object specified to the ***dstid*** argument does not exist, or the address of the data has not yet been allocated.
GL_ERROR_805F_DMP	nngxTransferLinearImage	Either the width or the height of the destination render buffer is invalid.

Error Code	Error Generating Function	Error Cause
GL_ERROR_8060_DMP	nngxTransferLinearImage	The ***target*** argument is invalid.
GL_ERROR_8062_DMP	**nngxAddVramDmaCommand**	Either a valid command list object is not currently bound, or the current command request queue is too small.
GL_ERROR_8064_DMP	**nngxAddVramDmaCommand**	A negative value was specified for ***size***.
GL_ERROR_8065_DMP	**nngxClearFillCmdlist**	This function was called on a command list that was still being executed.
GL_ERROR_8066_DMP	**nngxValidateState**	There was an overflow in the 3D command buffer.
GL_ERROR_8067_DMP	**nngxTransferLinearImage**	Either the destination render buffer or the texture's pixels are of invalid size.
GL_ERROR_8068_DMP	**nngxFilterBlockImage**	Either a valid command list object is not currently bound, or the current command request queue is too small.
GL_ERROR_8069_DMP	**nngxFilterBlockImage**	Either ***srcaddr*** or ***dstaddr*** is not 8-byte aligned.
GL_ERROR_806A_DMP	**nngxFilterBlockImage**	An invalid value was specified for ***width*** or ***height***.
GL_ERROR_806B_DMP	**nngxFilterBlockImage**	An invalid value was specified for ***format***.
GL_ERROR_806C_DMP	**nngxValidateState**	An error was generated during validation.
GL_ERROR_806D_DMP	**nngxSetGasAutoAccumulationUpdate**	0 is bound to the current command list.
GL_ERROR_806E_DMP	**nngxSetGasAutoAccumulationUpdate**	An invalid value was specified for ***id***.
GL_ERROR_806F_DMP	**nngxAddL2BTransferCommand**	A valid command list object is not currently bound or the current command request queue is too small.
GL_ERROR_8070_DMP	**nngxAddL2BTransferCommand**	Either ***srcaddr*** or ***dstaddr*** uses an invalid alignment.
GL_ERROR_8071_DMP	**nngxAddL2BTransferCommand**	***blocksize*** is invalid.
GL_ERROR_8072_DMP	**nngxAddL2BTransferCommand**	Either ***width*** or ***height*** is invalid.
GL_ERROR_8073_DMP	**nngxAddL2BTransferCommand**	***format*** is invalid.
GL_ERROR_8074_DMP	**nngxAddBlockImageCopyCommand**	A valid command list object is not currently bound or the current command request queue is too small.
GL_ERROR_8075_DMP	**nngxAddBlockImageCopyCommand**	Either ***srcaddr*** or ***dstaddr*** uses an invalid alignement.
GL_ERROR_8076_DMP	**nngxAddBlockImageCopyCommand**	***totalsize*** is invalid.
GL_ERROR_8077_DMP	**nngxAddBlockImageCopyCommand**	Either ***srcunit***, ***srcinterval***, ***dstunit***, or

Error Code	Error Generating Function	Error Cause
		dstinterval is invalid.
GL_ERROR_8078_DMP	**nngxAddMemoryFillCommand**	A valid command list object is not currently bound or the current command request queue is too small.
GL_ERROR_8079_DMP	**nngxAddMemoryFillCommand**	Either ***startaddr0*** or ***startaddr1*** uses an invalid alignment.
GL_ERROR_807A_DMP	**nngxAddMemoryFillCommand**	Either ***size0*** or ***size1*** is invalid.
GL_ERROR_807B_DMP	**nngxAddMemoryFillCommand**	Either ***width0*** or ***width1*** is invalid.
GL_ERROR_807C_DMP	**nngxAddB2LTransferCommand**	A valid command list object is not currently bound or the current command request queue is too small.
GL_ERROR_807D_DMP	**nngxAddB2LTransferCommand**	Either ***srcaddr*** or ***dstaddr*** uses an invalid alignment.
GL_ERROR_807E_DMP	**nngxAddB2LTransferCommand**	***blocksize*** is invalid.
GL_ERROR_807F_DMP	**nngxAddB2LTransferCommand**	***aamode*** is invalid.
GL_ERROR_8080_DMP	**nngxAddB2LTransferCommand**	Either ***srcformat*** or ***dstformat*** is invalid.
GL_ERROR_8081_DMP	**nngxAddB2LTransferCommand**	The target image has a larger pixel size than the original image.
GL_ERROR_8082_DMP	**nngxAddB2LTransferCommand**	Either ***srcwidth***, ***srcheight***, ***dstwidth***, or ***dstheight*** is invalid.
GL_ERROR_8083_DMP	**nngxAddB2LTransferCommand**	The target image is wider or taller than the original image in pixels.
GL_ERROR_9000_DMP	**nngxSwapBuffers**	The display mode is NN_GX_DISPLAYMODE_STEREO and either 0 is bound to NN_GX_DISPLAY0_EXT or the display buffer region has not been allocated.
GL_ERROR_9001_DMP	**nngxSwapBuffers**	The display mode is NN_GX_DISPLAYMODE_STEREO and the display region specified by the **nngxDisplayEnv** function is outside of the display buffer.
GL_ERROR_9002_DMP	**nngxSwapBuffers**	The display mode is NN_GX_DISPLAYMODE_STEREO and the resolution, format, or memory region differs between the display buffers bound to NN_GX_DISPLAY0 and NN_GX_DISPLAY0_EXT.
GL_ERROR_9003_DMP	**nngxSetDisplayMode**	An invalid value is specified for ***mode***.