Revolution SDK

Graphics Library (GX)

Version 1.01

Graphics Library (GX)

Version 1.01

Contents

Revision History xi
1 Introduction 1
1.1 Document Organization 1
1.2 Syntax Notes 1
1.3 A Note on Pointers 2
1.4 Useful Books 2
2 Code Example: smp-onetri.c 3
3 Initialization 9
3.1 Video Initialization 9
3.2 Graphics Initialization 9
3.3 Graphics Processor (GP) 10
4 Vertex and Primitive Data 11
4.1 Describing the Vertex Data 12
4.2 Describing Arrays 14
4.3 Describing Attribute Data Formats 15
4.4 Drawing Graphics Primitives 17
4.4.1 Primitive Types 17
4.4.2 Points and Lines 18
4.4.3 Rasterization Rules 20
4.4.4 Using Vertex Functions 20
4.5 Vertex Data Organization 22
4.5.1 Indexed Vertex Data 25
4.5.2 Direct Vertex Data 26
4.5.3 Mixture of Direct and Indexed Data 27
4.6 Display Lists 27
4.6.1 Creating Display Lists 28
4.6.2 Drawing Primitives Using Display Lists 30
4.6.3 Effect on Machine State 30
4.7 GXDraw Functions 31
5 Viewing 33
5.1 Loading a Modelview Matrix 34
5.2 Setting a Projection Matrix 35
5.3 Culling, Clipping, and Scissoring 36
5.4 Viewport and Scissoring 37
5.5 Coordinate Systems 38
5.6 How to Override the Default Matrix Memory Configuration 39
6 Vertex Lighting 41
6.1 Lighting Pipeline 41
6.1.1 Diffuse Lights, Diffuse Attenuation and Vertex Normals 41
6.1.2 Local Lights and Range Attenuation 41
6.1.3 Spotlights, Directional Lights and Angle Attenuation 41
6.2 Diffuse Lighting Equations 43
6.3 Matrix Memory 44
6.4 Light Parameters 44
6.4.1 Angle Attenuation 44
6.4.2 Distance Attenuation 46

6.5 Channel Parameters 47
6.5.1 Channel Colors 47
6.5.2 Channel Control 47
6.5.3 Pre-lighting 48
6.6 Specular Lighting 49
6.7 Vertex performance 50
6.8 Lighting Performance 51
7 Texture Coordinate Generation 53
7.1 Specifying Texgens 53
7.2 Renormalization and “post-transform” Matrices Added for Texgens 55
7.3 Other Texture Coordinate Generation Issues 56
7.4 Texture Coordinate Generation Performance 56
8 Texture Mapping 57
8.1 Example: Drawing a Textured Triangle 57
8.2 Loading a Texture into Main Memory 60
8.3 Describing a Texture Object 60
8.3.1 Texel Formats 62
8.3.2 Texture Lookup Table (TLUT) Formats 63
8.3.3 Texture Image Formats 63
8.3.4 Texture Coordinate Space 64
8.3.5 Filter Modes and LOD Controls 66
8.4 Loading Texture Objects 72
8.5 Loading Texture Lookup Tables (TLUTs) 73
8.6 How to Override the Default Texture Configuration 73
8.6.1 Texture Regions 74
8.6.2 Cached Regions 75
8.6.3 TLUT Regions 76
8.6.4 Preloaded Regions 76
8.6.5 Texture Cache Allocation 82
8.6.6 TLUT Allocation 82
8.7 Invalidating Texture Cache 83
8.8 Changing the Usage of TMEM Regions 83
8.9 Creating Textures by Copying the Embedded Frame Buffer 84
8.10 Z Textures 87
8.11 Texture Performance 89
9 Texture Environment (TEV) 91
9.1 Description 91
9.2 Default Texture Pipeline Configuration 92
9.3 Number of Active TEV Stages 92
9.4 GXSetTevOp 92
9.5 Color/alpha Combine Operations 93
9.5.1 Compare Mode 95
9.6 Color Inputs 97
9.7 Alpha Inputs 99
9.8 Color Component Swap Mode 99
9.9 TEV “constant” Color Registers 100
9.10 Example Settings 101
9.11 Alpha Compare Function 103
9.12 Z Textures 104
9.13 Texture Pipeline Configuration 104
10 Indirect Texture Mapping 107
10.1 Setting Up Indirect Texture Stages 109
10.2 Basic Indirect Texture Processing 111

10.3 Basic Indirect Texture Functions 112
10.3.1 Texture Warping 112
10.3.2 Environment-mapped Bump-mapping (EMBM) (*dX*, *dY*, *dZ*) 113
10.4 Advanced Indirect Texture Processing 114
10.4.1 Selecting "bump alpha" 114
10.4.2 Dynamic Matrices 114
10.4.3 Selecting Texture Coordinates for Texture LOD 115
10.4.4 Adding Texture Coordinates from Previous TEV Stages 115
10.5 Advanced Indirect Functions 115
10.5.1 Texture Tiling and Pseudo-3D Texturing 115
10.5.2 Environment-mapped Bump-mapping (EMBM) (*dS*, *dT*) 116
10.5.3 General Indirect Texturing 117
11 Fog, Z-compare, Blending, and Dithering 119
11.1 Fog 119
11.1.1 Fog Curves 120
11.1.2 Fog Parameters 122
11.2 Z-compare 122
11.2.1 Z Buffer Format 123
11.3 Blending 124
11.3.1 Blend Equation 124
11.3.2 Blending Parameters 125
11.3.3 Logic Operations 126
11.4 Dithering 127
12 Video Output 129
12.1 The Copy Pipeline 129
12.1.1 Copy Source 129
12.1.2 Antialiasing and Deflickering 129
12.1.3 Gamma Correction 130
12.1.4 RGB to YUV 131
12.1.5 Y Scale 131
12.1.6 Copy Destination 131
12.1.7 Clear Color and Z for Next Frame 131
12.2 Predefined Render Modes 132
12.2.1 Double-strike, Non-antialiased Mode 133
12.2.2 Double-strike, Antialiased Mode 133
12.2.3 Interlaced, Non-antialiased, Field-rendering Mode 134
12.2.4 Interlaced, Antialiased, Field-rendering Mode 134
12.2.5 Interlaced, Non-antialiased, Frame-rendering, Deflicker Mode 134
12.2.6 Interlaced, Non-antialiased, Frame-rendering, Non-deflicker Mode 135
12.2.7 Interlaced, Antialiased, Frame-rendering, Deflicker Mode 135
12.2.8 Progressive Mode 136
12.2.9 Progressive, "Soft" Mode 136
12.2.10 Progressive Antialised Mode 136
12.3 GX API Default Render Mode 137
12.4 Embedded Frame Buffer Formats 137
12.4.1 48-bit Format – Non-antialiasing 137
12.4.2 96-bit Super-sampling Format – Antialiasing 138
12.5 External Frame Buffer Format 138
13 Graphics FIFO 139
13.1 Description 139
13.2 Creating a FIFO 141
13.3 Attaching and Saving FIFOs 142
13.4 FIFO Status 143
13.5 FIFO Flow Control 144

13.6 Draw Synchronization Functions 144
13.6.1 GXDrawDone 145
13.6.2 GXDrawSync 145
13.6.3 FIFO Breakpoint 146
13.6.4 Abort Frame 147
13.6.5 VI Synchronization 147
13.7 Draw Synchronization Methods 147
13.7.1 Double-Buffering 147
13.7.2 Triple-Buffering 148
13.8 Graphics FIFO vs. Display List 148
13.9 Notes About the Write-Gather Pipe 148
13.10 GX Verify 149
14 Performance Metrics 151
14.1 Types of Metrics 151
14.2 GP Front-End and Texture-Related Metrics 151
14.2.1 GP Counter 0 Details 151
14.2.2 Counter 1 Details 153
14.3 Using Performance Counters 155
14.4 Vertex Cache Metrics 155
14.5 Pixel Metrics 156
14.6 Memory Metrics 156
15 Limitations 159
15.1 Antialiasing 159
15.2 CPU Access to the Frame Buffer 159
15.3 Display Lists 159
15.4 Vertex Performance 159
15.5 Matrix Memory 159
15.6 Texture 159
15.7 Blending and Logic Operations 160
15.8 Sharing Main Memory Resources 160
Appendix A.GX API Functions 161
Appendix B.GXInit Defaults 163
Appendix C.Display List Format 167
C.1 Display List Opcodes 167
C.2 Attribute Order Requirements 168
C.3 Example Display List (primitives only) 169
C.4 State Commands 170
C.4.1 Set_TextureMode0 172
C.4.2 Set_TextureMode1 172
C.4.3 Set_TextureImage0 172
C.4.4 Set_TextureImage1 173
C.4.5 Set_TextureImage2 173
C.4.6 Set_TextureImage3 173
C.4.7 Set_TextureTLUT 174
C.4.8 SU_TS0 174
C.4.9 SU_TS1 174
Appendix D.Wii Texture Formats 175
D.1 Texel Formats 176
D.2 Texture Tile Formats 177
D.3 Texture Image Formats 178
Appendix E.Memory Issues 181
E.1 Rules of Alignment 181
E.2 Alignment Assistance Functions 182

E.3 Data Coherency 182
E.3.1 Loading Graphics Data with the Optical Disc Drive Library 184
E.3.2 CPU Generating or Modifying Graphics Data 184

Code Examples

Code 2-1 onetri.c 3
Code 4-1 Vertex Descriptor 12
Code 4-2 GXSetVtxAttrFmt 16
Code 4-3 GXSetPointSize 18
Code 4-4 GXSetLineSize 18
Code 4-5 Vertex Functions 20
Code 4-6 Drawing Primitives Using Vertex Functions 21
Code 4-7 Using Vertex Functions 21
Code 4-8 Indexed vs. Direct Compression Example 23
Code 4-9 GXSetArray 25
Code 4-10 Arrays of Vertex Structures 26
Code 4-11 Direct Vertex Data 27
Code 4-12 Mixture of Direct and Indexed Data 27
Code 4-13 GXBeginDisplayList 28
Code 4-14 Sample Array Containing Display List 29
Code 4-15 GXCallDisplayList 30
Code 4-16 GX Draw Functions 31
Code 5-1 GXLoadPosMtxImm 34
Code 5-2 GXSetProjection 35
Code 5-3 GXSetViewport 38
Code 5-4 GXSetScissor 38
Code 5-5 GXProject 39
Code 6-1 GXSetNumChans 41
Code 6-2 GXInitLightAttn 44
Code 6-3 GXInitLightSpot 45
Code 6-4 GXInitLightDistAttn 46
Code 6-5 GXSetChanAmbColor 47
Code 6-6 GXSetChanCtrl 47
Code 6-7 Pre-lighting API 48
Code 6-8 GXInitLightShininess() 49
Code 7-1 GXSetTexCoordGen 53
Code 7-2 GXSetNumTexGens 54
Code 7-3 GXSetTexCoordGen2 56
Code 7-4 GXSetTexCoordScaleManually 56
Code 7-5 GXSetTexCoordCylWrap 56
Code 8-1 Simple Texture Example 57
Code 8-2 Initializing or Changing a Texture Object 61
Code 8-3 Texture Component Promotion to 8 bits 62
Code 8-4 GXInitTexObjLOD 66
Code 8-5 GXLoadTexObj 72
Code 8-6 Loading TLUTs 73
Code 8-7 GXInitTexCacheRegion 75
Code 8-8 GXInitTlutRegion 76
Code 8-9 GXInitTexPreLoadRegion() 76
Code 8-10 GXPreLoadEntireTexture() 81
Code 8-11 GXLoadTexObjPreLoaded() 82
Code 8-12 GXSetTexRegionCallback 82
Code 8-13 GXSetTlutRegionCallback 82

Code 8-14 Invalidating Texture Memory 83
Code 8-15 Texture Copy Functions 86
Code 8-16 GXSetZTexture 88
Code 9-1 GXSetNumTevStages 92
Code 9-2 GXSetTevOp 93
Code 9-3 GXSetTevColorOp, GXSetTevAlphaOp 94
Code 9-4 GXSetTevColorIn 97
Code 9-5 Setting Constant Color 98
Code 9-6 GXSetTevAlphaIn 99
Code 9-7 GXSetTevSwapMode, GXSetTevSwapModeTable 100
Code 9-8 Pass Texture Color 101
Code 9-9 Modulate 102
Code 9-10 Modulate 2X 102
Code 9-11 Add 102
Code 9-12 Subtract 102
Code 9-13 Blend 103
Code 9-14 GXSetAlphaCompare 103
Code 9-15 GXSetTevOrder 104
Code 10-1 GXSetIndTexOrder 110
Code 10-2 GXSetIndTexCoordScale 110
Code 10-3 GXSetIndTexMtx 112
Code 10-4 GXSetTevIndWarp 112
Code 10-5 GXSetTevIndBumpXYZ 113
Code 10-6 GXSetTevIndTile 115
Code 10-7 GXSetTexCoordScaleManually 115
Code 10-8 GXSetTevIndBumpST 116
Code 10-9 GXSetTevIndRepeat 117
Code 10-10 GXSetTevIndirect 117
Code 10-11 GXSetTevDirect 117
Code 11-1 GXSetFog 122
Code 11-2 Fog Range Adjustment Functions 122
Code 11-3 GXSetZMode 123
Code 11-4 GXSetBlendMode 124
Code 11-5 GXSetDither 127
Code 12-1 GXSetDispCopySrc 129
Code 12-2 GXSetCopyFilter 130
Code 12-3 GXSetCopyClamp 130
Code 12-4 GXSetDispCopyGamma 130
Code 12-5 GXSetDispCopyYScale 131
Code 12-6 GXCopyDisp 131
Code 12-7 GXSetCopyClear 131
Code 12-8 GXSetScissorBoxOffset 136
Code 12-9 GXSetPixelFmt 137
Code 13-1 GXFifoObj 141
Code 13-2 FIFO Initialization Functions 142
Code 13-3 FIFO Basic Inquiry Functions 142
Code 13-4 FIFO Attachment Functions 142
Code 13-5 FIFO Attachment Inquiry Functions 142
Code 13-6 GXGetCPUFifo 143
Code 13-7 FIFO Status Functions 143
Code 13-8 APIs to Get and Set the Current GX Thread 144
Code 13-9 GXDrawDone Synchronization Commands 145
Code 13-10 GXDrawSync Synchronization Commands 146
Code 13-11 GXEnableBreakPt 146

Code 13-12 GXDisableBreakPt 146
Code 13-13 GXSetBreakPtCallback 146
Code 13-14 GXAbortFrame 147
Code 13-15 VI Synchronization Commands 147
Code 13-16 Double-Buffer Copy Synchronization 147
Code 13-17 Single-Buffer Copy Synchronization 148
Code 13-18 APIs to Control the Write-Gather Pipe 149
Code 13-19 APIs to Control Verification 149
Code 14-1 GP Metric Functions 151
Code 14-2 Counting a Metric 155
Code 14-3 Vertex Cache Metric Functions 155
Code 14-4 Pixel Metric Functions 156
Code 14-5 Memory Metric Functions 156
Code B-1 GXInit Defaults 163
Code C-2 Code Necessary to Utilize Example Display List 170
Code C-3 Set_TextureMode0 172
Code C-4 Set_TextureMode1 172
Code C-5 Set_TextureImage0 172
Code C-6 Set_TextureImage1 173
Code C-7 Set_TextureImage2 173
Code C-8 Set_TextureImage3 173
Code C-9 Set_TextureTLUT 174
Code C-10 SU_TS0 174
Code C-11 SU_TS1 174
Code E-12 DVDSetAutoInvalidation 184
Code E-13 Commands to Flush the CPU Data Cache 184

Equations

Equation 4-1 Attribute Address 14
Equation 5-1 Vertex position transform 34
Equation 5-2 Vertex Normal Transform 34
Equation 5-3 Perspective Projection 35
Equation 5-4 Orthographic Projection 35
Equation 5-5 Clip Space to Screen Space Conversion 37
Equation 5-6 Normal Matrix Index 40
Equation 6-1 Light Parameters 43
Equation 6-2 Rasterized Color 43
Equation 6-3 Color Channel 43
Equation 6-4 Material Source 43
Equation 6-5 Channel Enable 43
Equation 6-6 Sum of Lights in a Channel 43
Equation 6-7 Ambient Source 43
Equation 6-8 Diffuse Attenuation 43
Equation 6-9 Diffuse Angle and Distance Attenuation 44
Equation 6-10 Pre-lighting 48
Equation 6-11 Specular Attenuation 49
Equation 7-1 Texture Coordinate Generation 53
Equation 7-2 Transforming *src_param* by 2x4 and 3x4 Matrices 53
Equation 7-3 Input Coordinates 53
Equation 9-1 Regular TEV output 95
Equation 9-2 Compare TEV output 95
Equation 9-3 Pass Texture Color 101
Equation 9-4 Modulate 102

Equation 9-5 Modulate 2X 102
Equation 9-6 Add 102
Equation 9-7 Subtract 102
Equation 9-8 Blend 103
Equation 9-9 Alpha Compare 103
Equation 9-10 Sample Alpha Compare 103
Equation 10-1 Dynamic Indirect Matrices 114
Equation 11-1 Blending 124
Equation 11-2 Subtractive Blend Operation 124
Equation 11-3 Bayer Matrix 127
Equation 11-4 5-bit Dithering (ideal) 127
Equation 11-5 5-bit Dithering (approximation actually used) 127
Equation 11-6 6-bit Dithering (ideal) 127
Equation 11-7 6-bit Dithering (approximation actually used) 127
Equation 12-1 RGB to YUV conversion 138
Equation 14-1 Miss-Rate Calculation 153

Figures

Figure 3-1 Schematic of the GP 10
Figure 4-1 Vertex and Attribute Description 12
Figure 4-2 Vertex Attribute Format Table (VAT) 15
Figure 4-3 Graphics Primitives 17
Figure 4-4 Point Definition 18
Figure 4-5 Line Definition 19
Figure 4-6 Polygon Rasterization Rules 20
Figure 4-7 Flow of Indexed Vertex Data 22
Figure 4-8 Flow of Direct Vertex Data 24
Figure 4-9 Indexed Vertex Data 25
Figure 4-10 Display List Flow 28
Figure 5-1 Modelview and Projection Data Path 33
Figure 5-2 Clipping and Culling Data Path 36
Figure 5-3 Clip Coordinates 37
Figure 5-4 Coordinate System Transformations 38
Figure 5-5 Matrix Memory 39
Figure 6-1 Associating Lights with Color Channels 42
Figure 6-2 Lighting Vectors 43
Figure 6-3 Spotlight Functions 45
Figure 6-4 Distance Attenuation Functions 46
Figure 6-5 Specular Lighting Vectors 49
Figure 6-6 GXInitLightShininess Values 50
Figure 7-1 Texgen Computation Path 55
Figure 8-1 Map-relative Texture Coordinates 64
Figure 8-2 Linear Filter—Clamp, Repeat, Mirror 65
Figure 8-3 Nearest Filter—Clamp, Repeat, Mirror 65
Figure 8-4 Pixel Projected in Texture Space Example 66
Figure 8-5 LOD Calculation 67
Figure 8-6 LOD Bias 68
Figure 8-7 Anisotropic Filtering 69
Figure 8-8 Mipmap Pyramid for the Largest Texture Size 70
Figure 8-9 GX_NEAR 71
Figure 8-10 GX_LINEAR 71
Figure 8-11 Default TMEM Configuration 74
Figure 8-12 Mipmap in TMEM 77

Figure 8-13 Planar Texture in TMEM 78
Figure 8-14 32-bit Planar Texture in TMEM 79
Figure 8-15 Color Index Mipmap in TMEM 80
Figure 8-16 32-bit Mipmap in TMEM 81
Figure 8-17 Texture Copy Data Path 85
Figure 8-18 Copying Small Textures into a Larger Texture in Main Memory 86
Figure 8-19 Z Texture Block Diagram 88
Figure 9-1 TEV Block Diagram 91
Figure 9-2 Default Texture Pipeline 92
Figure 9-3 TEV Operations 94
Figure 9-4 TEV Stage Color Inputs 97
Figure 9-5 TEV Stage Alpha Inputs 99
Figure 9-6 Texture Pipeline Control 105
Figure 10-1 Indirect Texture Operation 107
Figure 10-2 Tiled Texture Mapping 107
Figure 10-3 Pseudo-3D Textures 108
Figure 10-4 Regular Texture Functional Diagram 109
Figure 10-5 Regular and Indirect Texture Functional Diagram 110
Figure 10-6 Texture Coordinate Sharing Example 111
Figure 10-7 Indirect Texture Processing, Part 1 111
Figure 10-8 Indirect Texture Processing, Part 2 114
Figure 11-1 Fog Range Adjustment 119
Figure 11-2 Linear Fog Curve 120
Figure 11-3 Exponential Fog Curve 120
Figure 11-4 Exponential Squared Fog Curve 121
Figure 11-5 Reverse Exponential Fog Curve 121
Figure 11-6 Reverse Exponential Squared Fog Curve 121
Figure 12-1 EFB-to-XFB Copy Pipeline 129
Figure 12-2 Render Mode Structure, Related Calls and Hardware Modules 133
Figure 12-3 Double-strike, Non-antialiased Mode 133
Figure 12-4 Interlaced, Non-antialiased, Field-rendering Mode 134
Figure 12-5 Interlaced, Non-antialiased, Frame-rendering, Deflicker Mode 135
Figure 12-6 Interlaced, Antialiased, Frame-rendering, Deflicker Mode 135
Figure 12-7 Overlapping Copy 136
Figure 12-8 XFB Format in Main Memory 138
Figure 13-1 GXFifoObj 139
Figure 13-2 Immediate Mode 140
Figure 13-3 Multi-Buffer Mode 141
Figure D-1 Texel Formats 176
Figure D-2 Texture Tile Formats 177
Figure D-3 Texture Image Formats 179
Figure E-4 Data Coherency 183

Tables

Table 4-1 Vertex Attribute Order Requirements 13
Table 1 - Vertex Performance 51
Table 7-1 Texture Coordinate Generation Order 54
Table 8-1 Texel Formats 62
Table 8-2 TLUT Formats 63
Table 8-3 Mipmap Minimum Filter Modes 72
Table 8-4 Texture Copy Formats and Conversion Notes 84
Table 8-5 Texture Performance 89
Table 9-1 GXTevMode Types 93

Table 9-2 Correspondence Between TEV Input and Output Register Names 95
Table 9-3 Color or Alpha Compare Operations 96
Table 9-4 Color-only Compare Operations 96
Table 9-5 Alpha-only Compare Operations 96
Table 9-6 Color and Alpha Constant Register Values 100
Table 11-1 16-bit Z Buffer Formats 123
Table 11-2 Blending Parameters 125
Table 11-3 Logic Operations 126
Table 14-1 Memory Metrics 157
Table C-1 Display List Opcodes 167
Table C-2 Vertex Index Stream Order Requirements 168
Table C-3 Example Display List 169
Table E-4 Memory Alignment Rules 181
Table E-5 Alignment Assistance Functions 182

Revision History

Version	Date Revised	Item	Description
1.01	2008/07/10	6.7, 6.8, 8.11	Updated the performance explanations, which formerly used old Nintendo GameCube data.
		12.2	Added information on progressive mode.
		13	Updated all content of Chapter 13 relating to FIFO and the display list.
		13.6.1	Added information on XF stalls.
		15.1	Removed the sentence “Dithering doesn’t work with antialiasing.”
		4.6.1.2	Deleted the former section 4.6.1.2, “Loading GPL Files”.
		16	Deleted the former Chapter 16, “OpenGL Comparison”.
		Appendix A	Deleted the list of GX API functions.
		-	Fixed typos and made other small corrections.
1.00	2007/12/01	-	Applied a standardized format.
	2006/03/01	-	First release by Nintendo of America Inc.

1 Introduction

The Revolution SDK Graphics Library (GX Library) is a programmer's interface to the Wii Graphics Processor (GP). The GX Library is intended to be as thin as possible in order to achieve high performance, but it must also provide a logical and straightforward view of the hardware. Our design goal for the GX Library is to provide a default configuration of the hardware so that initially the programmer can concentrate on the basics without being overwhelmed by unnecessary details. Later, as the hardware becomes more familiar, programmers can easily override the default configuration to expose more flexibility and features.

1.1 Document Organization

This document serves as a starting point for graphics programmers to learn about the Wii console's graphics capabilities. Chapters are organized as follows:

- Chapter 2 presents a simple code example.
- Chapter 3 discusses system initialization and presents a Graphics Processor block diagram.
- Chapters 4-12 document the GX functions to control the graphics pipeline (roughly in pipeline order).
- Chapter 13 explains the CPU-to-graphics interface in more detail.
- Chapter 14 discusses how to gather performance statistics, the CPU-to-EFB interface, and the GX verify system.
- Chapter 15 lists feature limitations and notes.
- Appendix A: Refer to the GX pages in the online *Revolution Function Reference Manual* for a list of functions in the GX API.
- Appendix B lists the default state set by `GXInit`.
- Appendix C provides more details on the display list format.
- Appendix D outlines the texture format used by the Wii console.
- Appendix E discusses in-memory data alignment and coherence issues.

1.2 Syntax Notes

All of the Graphics library functions are prefixed by `GX`, Video Interface library functions by `VI`, and Matrix-Vector library functions by `MTX`. For details on these other libraries, refer to the corresponding sections in this guide. Functions prefixed with `OS` and `PAD` are described in "Operating System Library (`OS.pdf`)" and "Nintendo GameCube Controller Library (`PAD.pdf`)," respectively, in the *"Programmer's Guide"* directory of the manuals.

While some VI and MTX functions are required to write a basic application, this overview is concerned primarily with the GX library. GX functions follow the naming conventions listed below.

Note: Here, and in other places throughout the document, an asterisk (*) is used to indicate a wildcard.

- `GXSet*` functions immediately set state in the graphics hardware. More accurately, they send state commands through a command FIFO which is then read by the Graphics Processor and routed to the proper register(s) (see "13 Graphics FIFO" on page 139).
- `GXInit*` functions store initial settings for state registers in various types of object structures.
- `GXLoad*` functions set indexed state, usually from a precompiled object (structure). Indexed state includes light parameters, matrices, texture state, etc.
- `GXWrite*` functions write data directly to CPU-accessible registers and thus set state asynchronously with the graphics command pipeline.
- `GXGet*` functions read state back from a shadow copy of the state kept by the GX API.
- `GXRead*` functions read data directly from CPU-accessible registers.

The GX API uses many enumerated types and several structure types. A structure type will be suffixed by `Obj`, `Region`, or `List`. `GXColor` is also a structure type. Any other type without these suffixes is an enumerated type.

1.3 A Note on Pointers

The GX API sometimes expects pointers as arguments to its functions. The Wii operating system (see “Operating System” in the *“Programmer’s Guide*” directory) sets up the CPU to treat virtual addresses in a manner similar to a MIPS CPU. That is, the most significant bits (MSBs) of a virtual address indicate whether the target data is mapped to cached or uncached memory. The rest of the bits are the physical address of the data. The Wii Graphics Processor (GP) ignores these MSBs and therefore is only concerned with physical addresses. The application is *not* required to convert virtual addresses to physical addresses on behalf of either the Graphics Processor or the GX API.

In general, the application will be working with cache-mapped data. If the application is accessing the same data as the Graphics Processor, the application must be careful to flush the data from the CPU cache before the Graphics Processor uses it. The Graphics Processor has no visibility to data in the CPU cache. For examples, see Appendix E.

1.4 Useful Books

This document assumes that you know how to program in the C language, and that are you are knowledgeable about matrix and vector mathematics. If this is not the case, you might want to do some preparatory reading. Here are some useful sources (check your local bookstore or library for the latest editions):

Foley, James D., et al., *Computer Graphics: Principles and Practice, 2nd Ed., Ad*dison-Wesley, Reading, MA, 1990.

Kempf, Renate and Chris Frazier (eds.), *OpenGL Reference Manual, 2nd Ed., Addiso*n-Wesley, Reading, MA, 1997.

Woo, Mason, et al., *OpenGL Programming Guide, 2nd Ed., Addiso*n-Wesley, Reading, MA, 1997.

2 Code Example: smp-onetri.c

`Onetri.c` is a fairly simple example that shows the basics of initializing graphics, making vertex formats, and drawing flat-shaded primitives. All the data and code are included in a single file. This demo is also available in the source tree at `/rvl_sdk/build/demos/gxdemo/src/Simple/smp-onetri.c`.

Note: Despite the name, this demo draws more than one triangle.

Code 2-1 onetri.c

```
#include <demo.h>

/*---------------------------------------------------------------------------*
  Model Data
 *---------------------------------------------------------------------------*/
#define STRUT_LN        130     // long side of strut
#define STRUT_SD        4       // short side of strut
#define JOINT_SD        10      // joint is a cube

/*---------------------------------------------------------------------------*
   The macro ATTRIBUTE_ALIGN provides a convenient way to align initialized
   arrays.  Alignment of vertex arrays to 32B IS NOT required, but may result
   in a slight performance improvement.
 *---------------------------------------------------------------------------*/
s16 Verts_s16[] ATTRIBUTE_ALIGN(32) =
{
//      x             y             z
    -STRUT_SD,     STRUT_SD,    -STRUT_SD,   // 0
     STRUT_SD,     STRUT_SD,    -STRUT_SD,   // 1
     STRUT_SD,     STRUT_SD,     STRUT_SD,   // 2
    -STRUT_SD,     STRUT_SD,     STRUT_SD,   // 3
     STRUT_SD,    -STRUT_SD,    -STRUT_SD,   // 4
     STRUT_SD,    -STRUT_SD,     STRUT_SD,   // 5
     STRUT_SD,     STRUT_LN,    -STRUT_SD,   // 6
     STRUT_SD,     STRUT_LN,     STRUT_SD,   // 7
    -STRUT_SD,     STRUT_LN,     STRUT_SD,   // 8
    -STRUT_SD,     STRUT_SD,    -STRUT_LN,   // 9
     STRUT_SD,     STRUT_SD,    -STRUT_LN,   // 10
     STRUT_SD,    -STRUT_SD,    -STRUT_LN,   // 11
     STRUT_LN,     STRUT_SD,    -STRUT_SD,   // 12
     STRUT_LN,     STRUT_SD,     STRUT_SD,   // 13
     STRUT_LN,    -STRUT_SD,     STRUT_SD,   // 14
    -JOINT_SD,     JOINT_SD,    -JOINT_SD,   // 15
     JOINT_SD,     JOINT_SD,    -JOINT_SD,   // 16
     JOINT_SD,     JOINT_SD,     JOINT_SD,   // 17
    -JOINT_SD,     JOINT_SD,     JOINT_SD,   // 18
     JOINT_SD,    -JOINT_SD,    -JOINT_SD,   // 19
     JOINT_SD,    -JOINT_SD,     JOINT_SD,   // 20
    -JOINT_SD,    -JOINT_SD,     JOINT_SD    // 21
};

u8 Colors_rgba8[] ATTRIBUTE_ALIGN(32) =
{
//    r,   g,   b,   a
     42,  42,  50, 255,   // 0
     80,  80,  80, 255,   // 1
    114, 114, 110, 255    // 2
};

/*---------------------------------------------------------------------------*
   Forward references
 *---------------------------------------------------------------------------*/
```

```
void        main            ( void );
static void CameraInit      ( Mtx v );
static void DrawInit        ( void );
static void DrawTick        ( Mtx v );
static void AnimTick        ( Mtx v );
static void PrintIntro      ( void );

/*---------------------------------------------------------------------------*
   Application main loop
 *---------------------------------------------------------------------------*/
void main ( void )
{
    Mtx         v;  // view matrix
    PADStatus   pad[PAD_MAX_CONTROLLERS]; // Controller state

    pad[0].button = 0;

    DEMOInit(NULL); // Init os, pad, gx, vi

    CameraInit(v);  // Initialize the camera.
    DrawInit();     // Define my vertex formats and set array pointers.

    PrintIntro(); // Print demo directions

    while(!(pad[0].button & PAD_BUTTON_MENU))
    {
        DEMOBeforeRender();
        DrawTick(v);        // Draw the model.
        DEMODoneRender();
        AnimTick(v);        // Update animation.
        PADRead(pad);
    }

    OSHalt("End of demo");
}

/*---------------------------------------------------------------------------*
   Functions
 *---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*
    Name:           CameraInit

    Description:    Initialize the projection matrix and load into hardware.
                    Initialize the view matrix.

    Arguments:      v      view matrix

    Returns:        none
 *---------------------------------------------------------------------------*/
static void CameraInit ( Mtx v )
{
    Mtx44   p;      // projection matrix
    Vec     up      = {0.20F, 0.97F, 0.0F};
    Vec     camLoc  = {90.0F, 110.0F, 13.0F};
    Vec     objPt   = {-110.0F, -70.0F, -190.0F};
    f32     left    = 24.0F;
    f32     top     = 32.0F;
    f32     znear    = 50.0F;
    f32     zfar     = 2000.0F;

    MTXFrustum(p, left, -left, -top, top, znear, zfar);
    GXSetProjection(p, GX_PERSPECTIVE);

    MTXLookAt(v, &camLoc, &up, &objPt);
}
```

```
/*---------------------------------------------------------------------------*
    Name:           DrawInit

    Description:    Initializes the vertex attribute format 0, and sets
                    the array pointers and strides for the indexed data.

    Arguments:      none

    Returns:        none
 *---------------------------------------------------------------------------*/
static void DrawInit( void )
{
    GXColor black = {0, 0, 0, 0};

    GXSetCopyClear(black, GX_MAX_Z24);

    // Set current vertex descriptor to enable position and color0.
    // Both use 8b index to access their data arrays.
    GXClearVtxDesc();
    GXSetVtxDesc(GX_VA_POS, GX_INDEX8);
    GXSetVtxDesc(GX_VA_CLR0, GX_INDEX8);

    // Position has 3 elements (x,y,z), each of type s16,
    // no fractional bits (integers)
    GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_POS, GX_POS_XYZ, GX_S16, 0);

    // Color 0 has 4 components (r, g, b, a), each component is 8b.
    GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_CLR0, GX_CLR_RGBA, GX_RGBA8, 0);

    // stride = 3 elements (x,y,z) each of type s16
    GXSetArray(GX_VA_POS, Verts_s16, 3*sizeof(s16));
    // stride = 4 elements (r,g,b,a) each of type u8
    GXSetArray(GX_VA_CLR0, Colors_rgba8, 4*sizeof(u8));

    // Initialize lighting, texgen, and tev parameters
    GXSetNumChans(1);   // default, color = vertex color
    GXSetNumTexGens(0); // no texture in this demo
    GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR0A0);
    GXSetTevOp(GX_TEVSTAGE0, GX_PASSCLR);
}

/*---------------------------------------------------------------------------*
    Name:           Vertex

    Description:    Create my vertex format

    Arguments:      v        8-bit position index
                    c        8-bit color index

    Returns:        none
 *---------------------------------------------------------------------------*/
static inline void Vertex( u8 v, u8 c )
{
    GXPosition1x8(v);
    GXColor1x8(c);
}

/*---------------------------------------------------------------------------*
    Name:           DrawFsQuad

    Description:    Draw a flat-shaded quad.

    Arguments:      v0    8-bit position index 0
                    v1    8-bit position index 1
```

```
                    v2    8-bit position index 2
                    v3    8-bit position index 3
                    c     8-bit color index

    Returns:        none
 *---------------------------------------------------------------------------*/
static inline void DrawFsQuad(
    u8 v0,
    u8 v1,
    u8 v2,
    u8 v3,
    u8 c )
{
    Vertex(v0, c);
    Vertex(v1, c);
    Vertex(v2, c);
    Vertex(v3, c);
}

/*---------------------------------------------------------------------------*
    Name:           DrawTick

    Description:    Draw the model once.  Replicates a simple strut model
                    many times in the x, y, z directions to create a dense
                    3D grid.  GXInit makes GX_PNMTX0 the default matrix.

    Arguments:      v        view matrix

    Returns:        none
 *---------------------------------------------------------------------------*/
static void DrawTick( Mtx v )
{
    f32  x;  // Translation in x.
    f32  y;  // Translation in y.
    f32  z;  // Translation in z.
    Mtx  m;  // Model matrix.
    Mtx  mv; // Modelview matrix.

    GXSetNumTexGens( 0 );
    GXSetNumTevStages( 1 );
    GXSetTevOp( GX_TEVSTAGE0, GX_PASSCLR );

    MTXIdentity(m);

    for(x = -10*STRUT_LN; x < 2*STRUT_LN; x += STRUT_LN)
    {
        for(y = -10*STRUT_LN; y < STRUT_LN; y += STRUT_LN)
        {
            for(z = STRUT_LN; z > -10*STRUT_LN; z -= STRUT_LN)
            {
                MTXRowCol(m, 0, 3) = x;
                MTXRowCol(m, 1, 3) = y;
                MTXRowCol(m, 2, 3) = z;
                MTXConcat(v, m, mv);
                GXLoadPosMtxImm(mv, GX_PNMTX0);

                GXBegin(GX_QUADS, GX_VTXFMT0, 36); //4 vtx/qd x 9 qd = 36 vtx
                    DrawFsQuad(8, 7, 2, 3, 0);
                    DrawFsQuad(1, 2, 7, 6, 1);
                    DrawFsQuad(1, 0, 9, 10, 2);
                    DrawFsQuad(4, 1, 10, 11, 1);
                    DrawFsQuad(1, 12, 13, 2, 2);
                    DrawFsQuad(2, 13, 14, 5, 0);
                    DrawFsQuad(18, 15, 16, 17, 2);
                    DrawFsQuad(20, 17, 16, 19, 1);
```

```
                    DrawFsQuad(20, 21, 18, 17, 0);
                GXEnd();
            }
        }
    }
}

/*---------------------------------------------------------------------------*
    Name:           AnimTick

    Description:    Moves viewpoint through the grid.  Loops animation so
                    that it appears viewpoint is continuously moving forward.

    Arguments:      v        view matrix

    Returns:        none
 *---------------------------------------------------------------------------*/
static void AnimTick( Mtx v )
{
    static u32  ticks = 0; // Counter.
    Mtx         fwd;       // Forward stepping translation matrix.
    Mtx         back;      // Loop back translation matrix.

    u32    animSteps    = 100;
    f32    animLoopBack = (f32)STRUT_LN;
    f32    animStepFwd  = animLoopBack / animSteps;

    MTXTrans(fwd, 0, 0, animStepFwd);
    MTXTrans(back, 0, 0, -animLoopBack);

    MTXConcat(v, fwd, v);
    if((ticks % animSteps) == 0)
        MTXConcat(v, back, v);

    ticks++;
}

/*---------------------------------------------------------------------------*
    Name:           PrintIntro

    Description:    Prints the directions on how to use this demo.

    Arguments:      none

    Returns:        none
 *---------------------------------------------------------------------------*/
static void PrintIntro( void )
{
    OSReport("\n\n****************************************\n");
    OSReport("to quit:\n");
    OSReport("    hit the start button\n");
    OSReport("****************************************\n");
}
```

The following library functions are explained in the listed sections:

- DEMOInit ("3.1 Video Initialization" on page 9).
- GXInit ("3.2 Graphics Initialization" on page 9).
- GXSetVtxDesc ("4.1 Describing the Vertex Data" on page 12).
- GXSetArray ("4.2 Describing Arrays" on page 14 and "4.5.1 Indexed Vertex Data" on page 25).
- GXSetVtxAttrFmt ("4.3 Describing Attribute Data Formats" on page 15).
- GXBegin/GXEnd ("4.4.4 Using Vertex Functions" on page 20).
- GXPosition, GXColor ("4.4.4 Using Vertex Functions" on page 20).
- GXLoadPosMatrixImm ("5.1 Loading a Modelview Matrix" on page 34).
- GXSetProjection ("5.2 Setting a Projection Matrix" on page 35).
- GXSetNumChans ("6.1.3 Spotlights, Directional Lights and Angle Attenuation" on page 41).
- GXSetNumTexGens ("7.1 Specifying Texgens" on page 53).
- GXSetTevOp ("9.4 GXSetTevOp" on page 92)
- GXSetTevOrder ("9.13 Texture Pipeline Configuration" on page 104).
- GXSetCopyClear ("12.1.7 Clear Color and Z for Next Frame" on page 131).

The GX Library can be accessed using <revolution.h>. This header file is included here by way of the <demo.h> header file.

3 Initialization

3.1 Video Initialization

In the preceding code segment, the `DEMOInit` function initializes the operating system, Controller, graphics, and video. The `DEMOInit` function takes a parameter that is a pointer to the render mode to be used. If you pass in a `NULL` pointer, then a default mode is chosen based upon the video standard.

For NTSC, the default render mode is 640x480, non-antialiased, double-buffered, and deflickered. On the 640x480 screen, some pixels are trimmed off to account for overscan. Currently, `DEMOInit` trims 16 pixels from the top and bottom of the screen, resulting in an actual size of 640x448. For more information about render modes, see "12 Video Output" on page 129 or refer to the “Graphics (GX)” and “Render Modes” pages in the *Revolution Function Reference Manual* (HTML).

The `DEMO` library encapsulates common functionality for the Wii demo programs.

3.2 Graphics Initialization

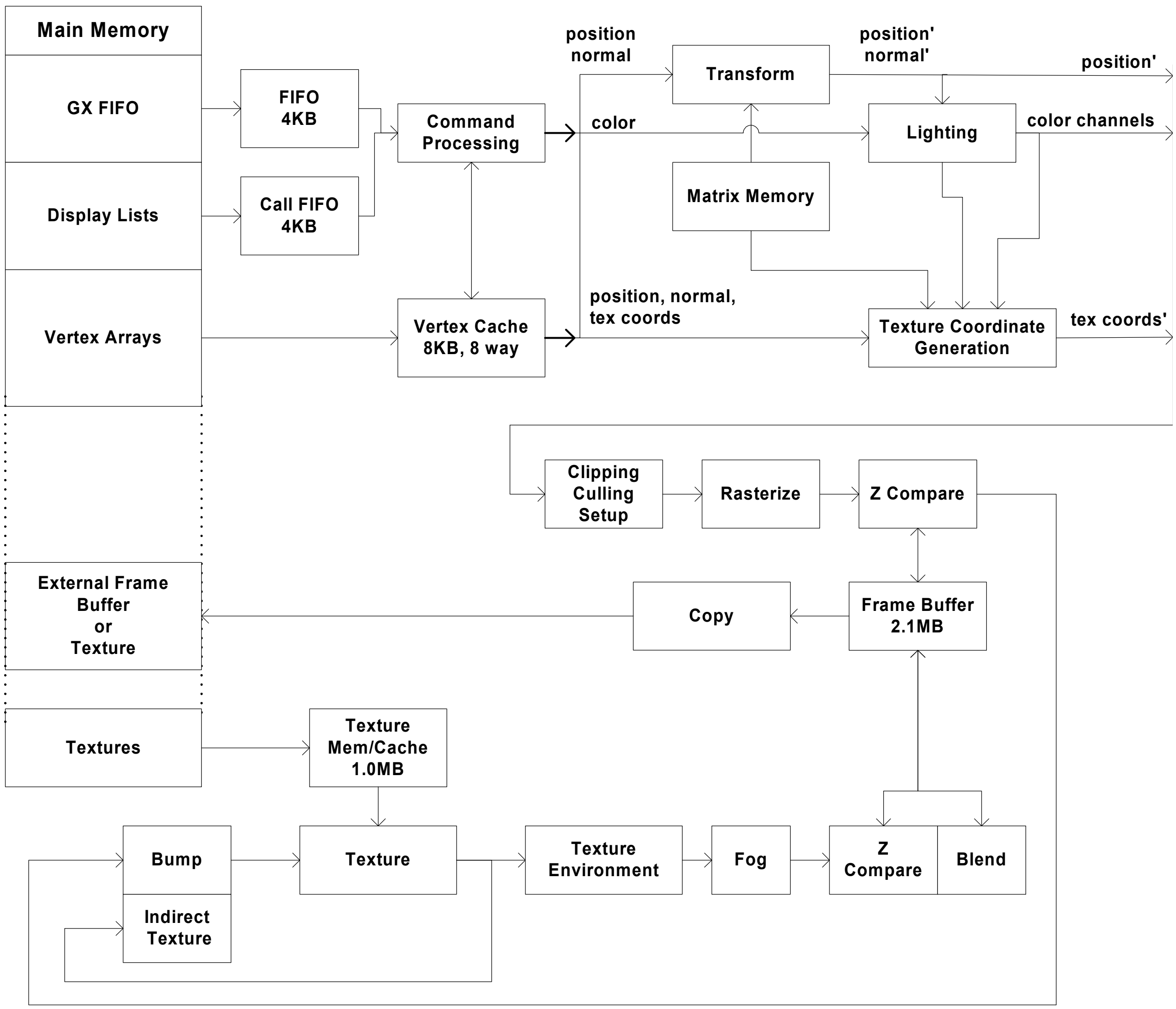
The Graphics Processor (GP) comes out of reset with unknown register values. The `GXInit` function is used to set all the registers in the GP to default values. In the code example (in Chapter 2), `GXInit` is called by way of `DEMOInit`.

`GXInit` also initializes a FIFO in main memory that is used to send graphics commands and data from the CPU to the GP (for more information on FIFO, see "13 Graphics FIFO" on page 139). The FIFO write port is attached to the CPU, and the FIFO read port is attached to the GP. This configuration is known as *immediate mode*, because graphics commands are sent immediately from the CPU to the GP as they are executed. In immediate mode, the GP will interrupt the CPU when the FIFO is nearly filled and temporarily stop any threads calling GX functions. As a result, multi-threaded applications must call GX commands from a single thread only. In this way, even if the FIFO is nearly full, only the thread generating graphics commands will be stopped. It is also possible to set the FIFO(s) up in a *multi-buffered* mode, in which the CPU writes commands to one FIFO while the GP reads commands from a different FIFO. See "13 Graphics FIFO" on page 139 for more information on the graphics FIFO.

3.3 Graphics Processor (GP)

The following figure is a logical block diagram of the GP. The subsequent chapters in this book roughly follow the order of the pipeline's processing blocks.

Figure 3-1 Schematic of the GP



4 Vertex and Primitive Data

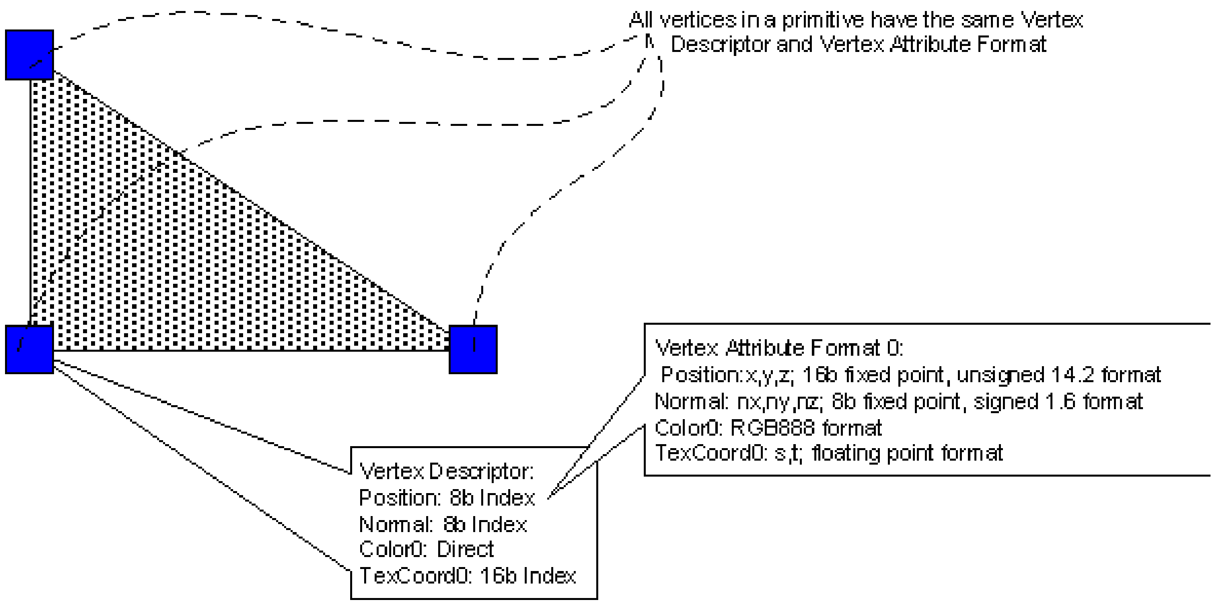
The GX API supports indexed and direct vertex data with flexible sizes and types. Here is a list of the terms that we will use in the following discussion:

- ***Attribute***: A component of a vertex; for example, position, normal, color, texture coordinate, or matrix index. Each attribute consists of one or more *elementary* types; for example, a position may consist of three floats (*x, y, z*).
- ***Vertex***: A group of attributes attached to a point in space; therefore, every vertex must have, at minimum, a position attribute.
- ***Primitive***: A geometric object described by a group of vertices with the *same* format.
- ***Vertex Descriptor***: Describes which attributes are present in a particular vertex format and how they are transmitted from the CPU to the GP (either *direct* or *indexed*).
- ***Vertex Attribute Format***: Describes the format (type, size, format, fixed point scale, etc.) of each attribute in a particular vertex format.
- ***Vertex Format***: A Vertex Attribute Format together with the Vertex Descriptor.
- ***Direct***: When an attribute is `GX_DIRECT`, the data representing that attribute is sent to the GP via the CPU/Graphics FIFO.
- ***Indexed***: When an attribute is `GX_INDEX*`, an index to the attribute data is sent to the GP via the Graphics FIFO. The GP then fetches the actual attribute data automatically by using the index and an array pointer.

To draw a primitive, you should follow these steps:

1. Describe which attributes are present in the vertex format and describe whether the attributes are indexed or referenced directly.
2. For indexed data, set the array pointers and strides.
3. Describe the number of elements in each attribute and their types.
4. Describe the primitive type.
5. Draw the primitive by sending the GP a stream of vertices that match the vertex description and attribute format.

Figure 4-1 Vertex and Attribute Description



4.1 Describing the Vertex Data

Code 4-1 Vertex Descriptor

```
GXClearVtxDesc();
GXSetVtxDesc(GX_VA_POS,  GX_INDEX8);
GXSetVtxDesc(GX_VA_NRM,  GX_INDEX8);
GXSetVtxDesc(GX_VA_CLR0, GX_DIRECT);
GXSetVtxDesc(GX_VA_TEX0, GX_INDEX16);
```

The `GXSetVtxDesc` function is used to indicate which attributes are present in the vertex data, and whether they are indexed or direct. There is only one active vertex descriptor, known as the *current vertex descriptor*. The `GXClearVtxDesc` command is used to set the value `GX_NONE` for all the attributes in the current vertex descriptor. `GX_NONE` indicates that no data for this attribute will be present in the vertex. Once cleared, you only need to describe attributes that you intend to provide. The possible attributes are:

- Position, `GX_VA_POS` (this attribute is required for every vertex descriptor).
- Normal, `GX_VA_NRM`, or normal/binormal/tangent, `GX_VA_NBT`.
- Color_0, `GX_VA_CLR0`.
- Color_1, `GX_VA_CLR1`.
- Up to 8 texture coordinates, `GX_VA_TEX0-7`.
- A position/normal matrix index, `GX_VA_PNMTXIDX`.
- A texture matrix index, `GX_VA_TEX0MTXIDX` - `GX_VA_TEX7MTXIDX`.

These last two attributes are 8-bit indices which can reference a transformation matrix in the on-chip matrix memory. This supports simple skinning of a character (for more on skinning, see the *Graphics Library (Advanced Rendering)* manual in this guide). These indices are different from the other attributes in that they may be sent only as direct data.

The GP assumes that you will send any specified attribute data in the ascending order shown in the table below:

Table 4-1 Vertex Attribute Order Requirements

Order	Attribute
0	GX_VA_PNMTXIDX
1	GX_VA_TEX0MTXIDX
2	GX_VA_TEX1MTXIDX
3	GX_VA_TEX2MTXIDX
4	GX_VA_TEX3MTXIDX
5	GX_VA_TEX4MTXIDX
6	GX_VA_TEX5MTXIDX
7	GX_VA_TEX6MTXIDX
8	GX_VA_TEX7MTXIDX
9	GX_VA_POS
10	GX_VA_NRM or GX_VA_NBT
11	GX_VA_CLR0
12	GX_VA_CLR1
13	GX_VA_TEX0
14	GX_VA_TEX1
15	GX_VA_TEX2
16	GX_VA_TEX3
17	GX_VA_TEX4
18	GX_VA_TEX5
19	GX_VA_TEX6
20	GX_VA_TEX7

Note: Texture coordinates must be enabled sequentially, starting at `GX_VA_TEX0`.

4.2 Describing Arrays

The attributes of a vertex may be indexed or direct (with the exception of `GX_VA_PNMTXIDX`, and `GX_VA_TEX0MTXIDX-GX_VA_TEX7MTXIDX`, which are always direct). For an indexed attribute (`GX_INDEX8` or `GX_INDEX16`), you need only to send an index to the attribute data. The GP will use the following equation to compute the address of the data:

Equation 4-1 Attribute Address

$$AttrAddress = AttrBase + Index * AttrStride$$

The `GX_INDEX8` index type allows a maximum array size of 255 elements (0-254). *The index 255 is reserved and indicates that this vertex should be skipped in the command stream.* See “Multi-Resolution Geometry” in *Advanced Rendering* for applications of this feature.

The `GX_INDEX16` index type allows a maximum array size of 65,535 elements (0-65,534). *The index 0xffff (65,535) is used to indicate that this vertex should be skipped.*

The attribute base pointer (byte-aligned) and stride (in bytes) are set using the `GXSetArray` function (described further in "4.5.1 Indexed Vertex Data" on page 25). The hardware will read the data described by the vertex attribute format (see "4.3 Describing Attribute Data Formats" on page 15) from the array. This avoids the need to read the data into the CPU, only to copy it back into the graphics FIFO. *However, indexing vertex data has cache coherency issues; see Appendix E.*

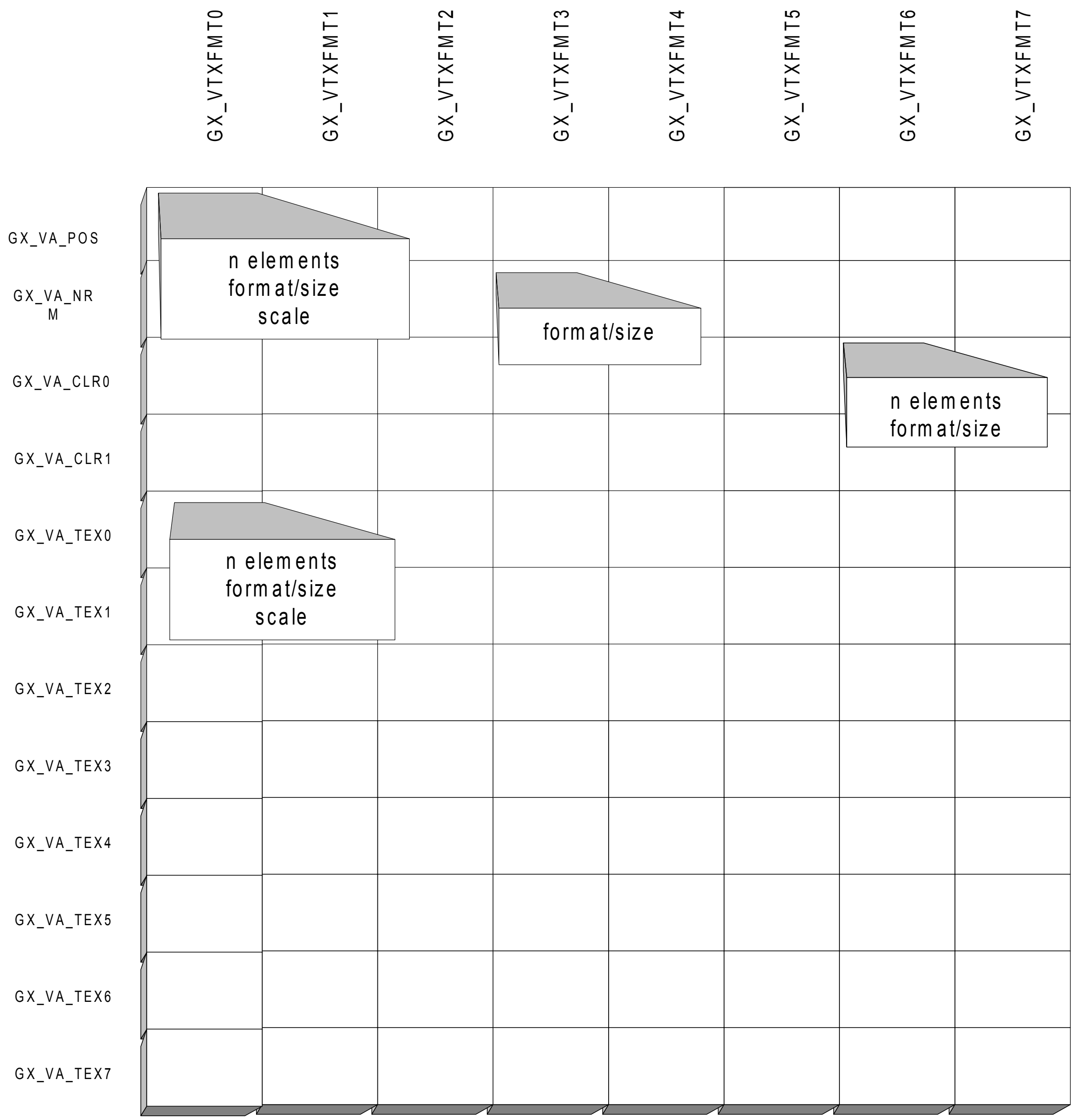
The Graphics Processor has its own vertex data cache in order to make the fetching of indexed data more efficient. The vertex cache is an 8-KB, 8-way set-associative cache. Notice that each attribute can be stored as a separate array. There is no need to pack a vertex structure in memory, because the current vertex descriptor and vertex attribute format allow the assembly of vertex data from the various arrays at run time. You can invalidate the vertex cache using `GXInvalidateVtxCache`. This will force the cache to reload vertex data.

A directly referenced attribute (`GX_DIRECT`) will have its data copied directly into the Graphics FIFO. Direct data can be used when the data is already available in the CPU cache or when you are generating data algorithmically on the fly.

4.3 Describing Attribute Data Formats

The Vertex Attribute Format Table (VAT) allows you to specify the format of each attribute for up to eight different vertex formats. The VAT is organized as shown:

Figure 4-2 Vertex Attribute Format Table (VAT)



You can store eight predefined vertex formats in the table. For each attribute in a vertex, you can specify the following:

- The number of elements for the attribute.
- The format and size information.
- The number of fractional bits for fixed-point formats using the scale parameter. (The scale parameter is not relevant to color or floating-point data.)

Code 4-2 GXSetVtxAttrFmt

```
//              format index  attribute   n elements   format   n frac bits
GXSetVtxAttrFmt(GX_VTXFMT0,  GX_VA_POS,  GX_POS_XYZ,  GX_S8,    0);
GXSetVtxAttrFmt(GX_VTXFMT0,  GX_VA_CLR0, GX_CLR_RGBA, GX_RGBA8, 0);
```

The code above defines vertex attribute format zero. `GX_VTXFMT0` indicates that position is a 3-element coordinate (*x, y, z*) where each element is an 8-bit 2's complement signed number. The scale value, labeled `n frac bits` in the code above, indicates the number of fractional bits for a fixed-point number, so zero indicates that the data has no fractional bits. The `GX_VA_CLR0` attribute has four elements (*r, g, b, a*) where each element is 8 bits.

Notes:

- The matrix index format is not specified in the table, because it is always an unsigned 8-bit value.
- The scale value is implied for normals (scale = 6 or scale = 14) and not needed for colors. Also, normals are assumed to have three elements (*Nx*, *Ny*, *Nz*) for `GX_VA_NRM`, and nine elements (*Nx*, *Ny*, *Nz*, *Bx*, *By*, *Bz*, *Tx*, *Ty*, *Tz*) for `GX_VA_NBT`. Normals are always signed values.
- The normal format (`GX_VA_NRM`) is also used for binormals/tangents (`GX_VA_NBT`) when they are enabled in the current vertex descriptor.

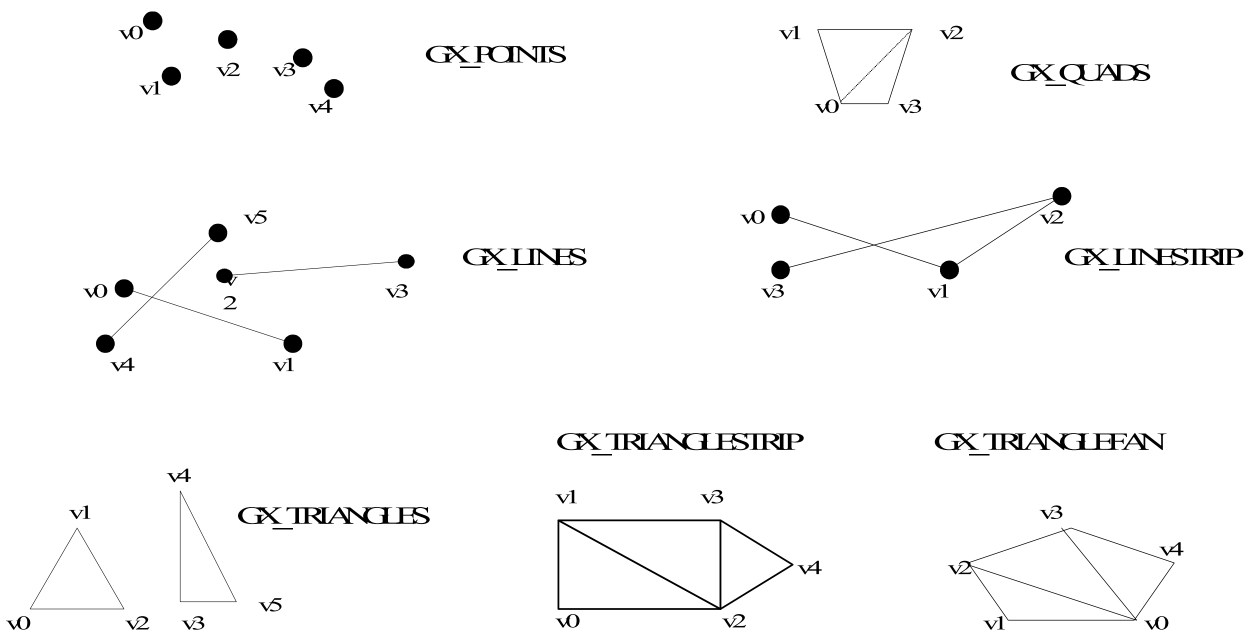
The VAT in the Graphics Processor has room for eight vertex formats. The idea is to describe most of your attribute quantization formats early in the application, loading this table as required. Then you provide an index into this table (which specifies the vertex attribute data format) when you start drawing a group of primitives using `GXBegin`. If you require more than eight vertex formats, you must manage the VAT table in your application, reloading new vertex formats as needed.

4.4 Drawing Graphics Primitives

4.4.1 Primitive Types

The following figure illustrates the types of primitives supported:

Figure 4-3 Graphics Primitives



`GX_POINTS` draws a point at each of the *n* vertices. Points are described further in "4.4.2 Points and Lines" on page 18.

`GX_LINES` draws a series of unconnected lines. Segments are drawn between *v0* and *v1*, *v2* and *v3*, and so forth. The number of vertices drawn should be a multiple of 2. Lines are described further in "4.4.2 Points and Lines" on page 18.

`GX_LINESTRIP` draws a series of connected lines, from *v0* to *v1*, then from *v1* to *v2*, and so on. If *n* vertices are drawn, *n-1* lines are drawn.

`GX_TRIANGLES` draws a series of triangles (three-sided polygons) using vertices *v0, v1, v2*, then *v3, v4, v5*, and so on. The number of vertices drawn should be a multiple of 3, and the minimum number is 3.

`GX_TRIANGLSTRIP` draws a series of triangles (three-sided polygons) using vertices *v0, v1, v2*, then *v1, v3, v2* (note the order), then *v2, v3, v4*, and so on. The number of vertices must be at least 3.

`GX_TRIANGLEFAN` draws a series of triangles (three-sided polygons) using vertices *v0, v1, v2*, then *v0, v2, v3,* and so on. The number of vertices must be at least 3.

`GX_QUADS` draws a series of non-planar quadrilaterals (4-sided polygons) beginning with *v0, v1, v2, v3*, then *v4, v5, v6, v7*, and so on. The quad is actually drawn using two triangles, so the four vertices are not required to be coplanar. The diagonal common edge between the two triangles of a quad is oriented as shown in "Figure 4-3 Graphics Primitives" on page 17. The minimum number of vertices is 4.

4.4.2 Points and Lines

Points are described by a single vertex, either 2D or 3D, and may be textured or not. You define a point's size using:

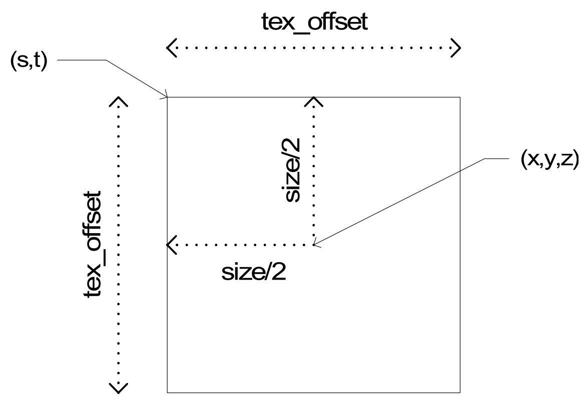
Code 4-3 GXSetPointSize

```
GXSetPointSize(u8 size, GXTexOffset tex_offset);
```

Points are drawn as a square in screen space, centered about the location of the vertex. The *size* of a point may be specified in 1/6 pixel units, and the maximum size is 42.5 pixels. If the point is textured, a texture coordinate should be generated or supplied per point. This texture coordinate is attached to the top-left corner of the point. The other texture coordinates for the other corners of the point are generated using *tex_offset*.

Note: *tex_offset* is specified in normalized texture coordinates.

Figure 4-4 Point Definition



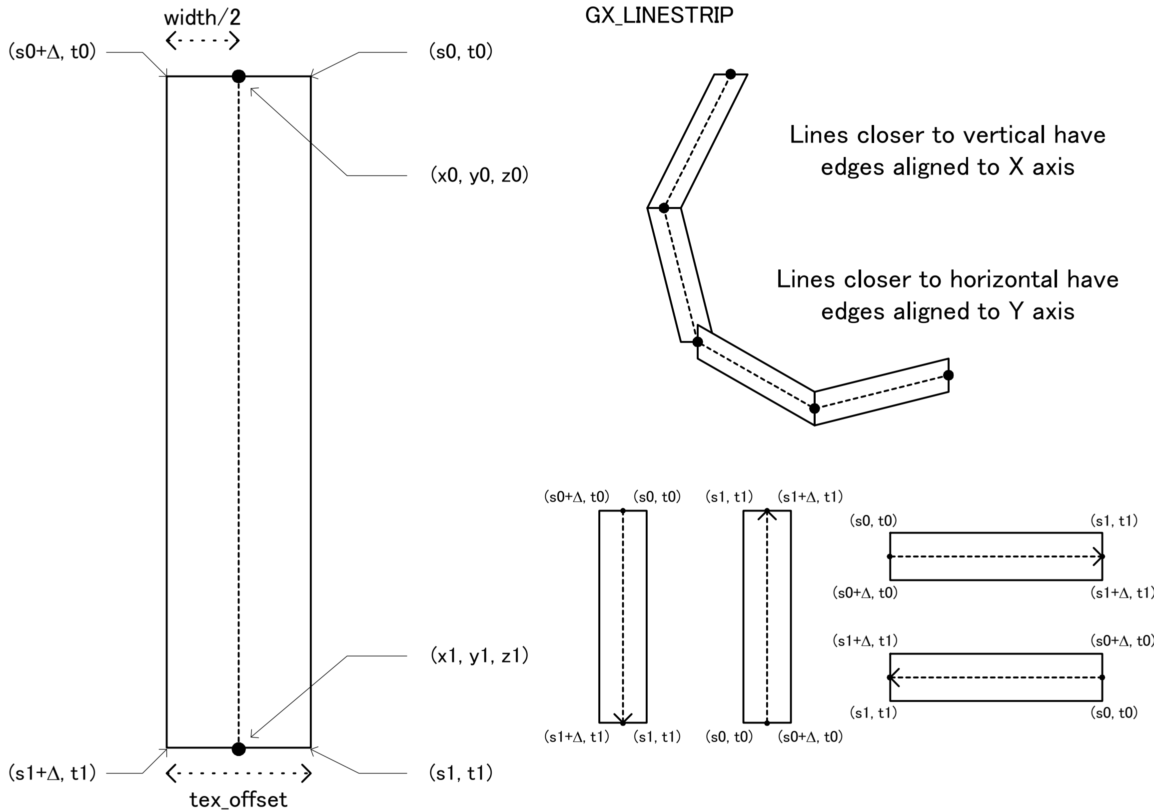
Lines are described by two vertices, either 2D or 3D, and may be textured or not. You define a line's width using:

Code 4-4 GXSetLineSize

```
GXSetLineSize(u8 width, GXTexOffset tex_offset);
```

Lines are centered about the location of the vertices. The small edges of a line will be drawn horizontally or vertically, depending upon the slope of the line (see "Figure 4-5 Line Definition" on page 19). The *width* of a line is specified in units of 1/6 pixel, and the maximum width is 42.5 pixels. If you are looking down the line from the start point to the end point, the starting texture coordinate is attached to the near left-hand corner of the line, while the ending texture coordinate is attached to the far left-hand corner. The texture coordinates for the right-hand corners are produced by adding the *tex_offset* value to the corresponding left-hand corner texture coordinates. The *tex_offset* is only added to the *s* component. See Figure 4-5 for details.

Figure 4-5 Line Definition



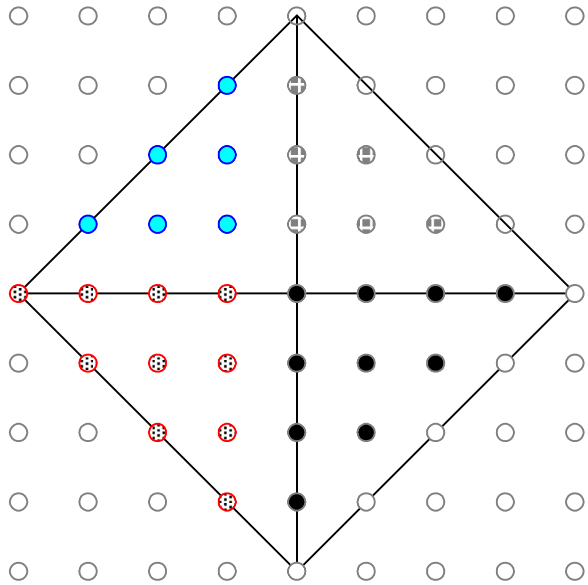
Wide line strips created using `GX_LINESTRIP` may have overlapped joints that may show gaps and cracks.

4.4.3 Rasterization Rules

As shown in "Figure 4-3 Graphics Primitives" on page 17, polygons whose vertices appear in clockwise order are defined to be frontfacing.

Polygon edges that fall exactly across a sample center will include the sample if the sample lies on a left edge of a polygon; samples that fall exactly on right edges will not be included. When an edge is horizontal, samples that fall exactly on upper edges are included, while samples that fall exactly on lower edges are not. A sample that occurs at the intersection of two edges will be included only if each edge follows the aforementioned rules at that intersection. These rules are illustrated in the following diagram.

Figure 4-6 Polygon Rasterization Rules



Samples are indicated by circles.

Each triangle includes only the similarly-shaded samples that it overlaps.

Points are converted into a quad (two triangles) by extending them by half the point size horizontally and vertically about the center. After this is completed, the rules for polygon edges are applied to determine which samples the point includes. Lines are converted into quads in a similar way, and again, the rules for polygon edges are applied to determine which samples the line will include.

4.4.4 Using Vertex Functions

The following functions can specify vertex data and indices to vertex data:

Code 4-5 Vertex Functions

```
GXPosition[n][t]
        n: {1, 2, 3}, t: {s8, u8, s16, u16, f32, x8, x16}
GXNormal[n][t]
        n: {1, 3}, t: {s8, s16, f32, x8, x16}
GXColor[n][t]
        n: {1, 3, 4}, t: {u8, u16, u32, x8, x16}
GXTexCoord[n][t]
        n: {1, 2}, t: {s8, u8, s16, u16, f32, x8, x16}
GXMatrixIndex1u8
```

You draw primitives by calling *vertex functions* (`GXPosition`, `GXColor`, etc.) between `GXBegin/GXEnd` pairs. You must call a vertex function for each attribute you enable using `GXSetVtxDesc()`, and for each vertex in the order specified in "Table 4-1 Vertex Attribute Order Requirements" on page 13. Each vertex function has a suffix of the form `GX[data][n][t]`, where *data* indicates an attribute (such as position or color), *n* indicates the number of attributes, and *t* indicates the types of each of the elements passed to the vertex function. See the GX pages in the online *Revolution Function Reference Manual* for details on each particular vertex function.

Code 4-6 Drawing Primitives Using Vertex Functions

```
        GXBegin(GX_TRIANGLES, GX_VTXFMT0, 3);

                GXPosition1x8(0); // index to position
                GXColor1x16(0);   // index to color

                GXPosition1x8(1);
                GXColor1x16(1);

                GXPosition1x8(2);
                GXColor1x16(2);

        GXEnd();
```

`GXBegin` specifies the type of primitive, an index into the VAT, and the number of vertices between the `GXBegin/GXEnd` pair. This information, along with the latest call to `GXSetVtxDesc()`, fully describes the primitive, vertex, and attribute format. `GXEnd()` is a null macro in the non-debug version of the library. In the debug version, it makes sure that `GXBegin` and `GXEnd` are paired properly. You may call vertex functions between `GXBegin` and `GXEnd` only.

The data type for each attribute should correspond to the vertex attribute format selected. In the example below, the data is indexed, so you call vertex functions that describe the format of the index (the '`x8`' or '`x16`' type is used to indicate indices):

Code 4-7 Using Vertex Functions

```
GXClearVtxDesc ();
GXSetVtxDesc(GX_VA_CLR0, GX_INDEX8);
GXSetVtxDesc(GX_VA_POS, GX_INDEX16);

GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_POS,  GX_POS_XYZ,  GX_U8,  0);
GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_CLR0, GX_CLR_RGBA, GX_RGBA8, 0);

// …

GXBegin(GX_TRIANGLES, GX_VTXFMT0, 3);

        GXPosition1x8(0);   // index to position
        GXColor1x16(0x3e4); // index to color

        GXPosition1x8(1);
        GXColor1x16(0x123e);

        GXPosition1x8(2);
        GXColor1x16(17);

GXEnd();
```

The function `GXPosition1x8` indicates that this function takes one unsigned char (8-bit) *index* as a parameter. The function `GXColor1x16` indicates that this function takes one unsigned short (16-bit) *index* as a parameter.

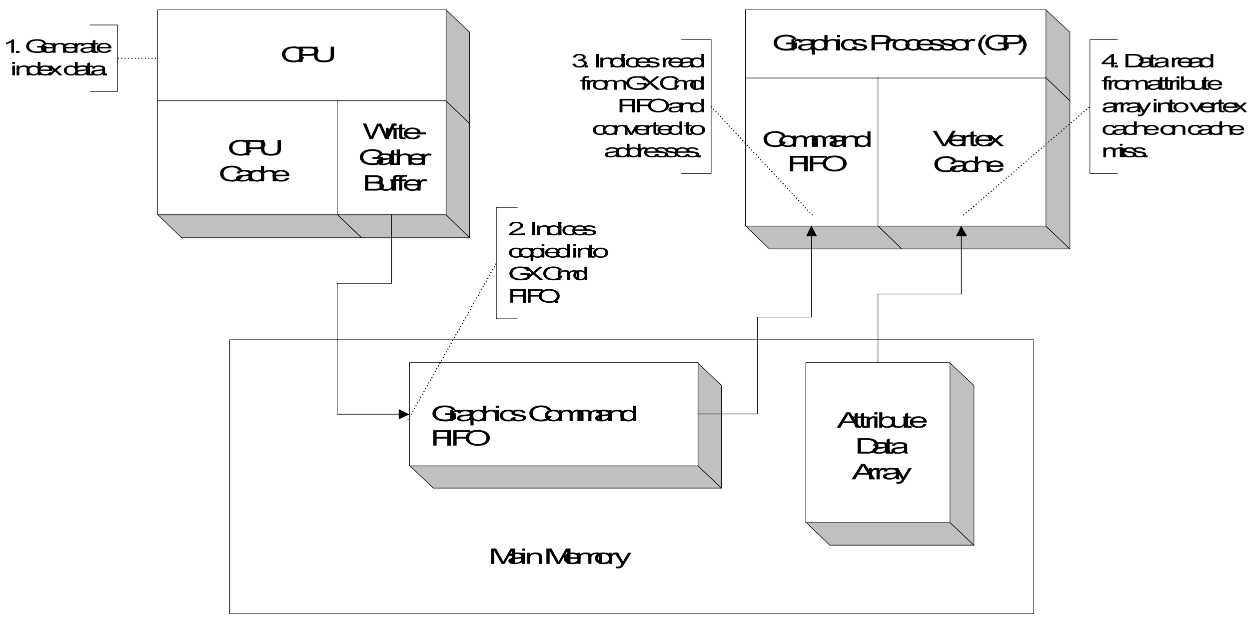
4.5 Vertex Data Organization

Indexed attribute data supports the Wii system's goal of flexible data organization. Programmers can organize data used by animation, collision, and graphics with a minimum of reformatting.

When using *indexed attribute data*, the attribute values themselves are stored in main memory in an array. The programmer describes graphics primitives using indices to reference into one or more of these arrays. The graphics hardware computes the physical addresses from the indices and fetches the data.

A *vertex cache* is used to cache parts of the arrays as they are accessed, taking advantage of the natural locality in geometric data. The data is stored in the vertex cache in quantized format, which improves effective memory bandwidth.

Figure 4-7 Flow of Indexed Vertex Data



The following example of a simple textured box shows how indexed data can be more compressed than direct data.This example only computes memory size. Indexing can also be beneficial in terms of bandwidth, because data already in the vertex cache will not need to be read from main memory.

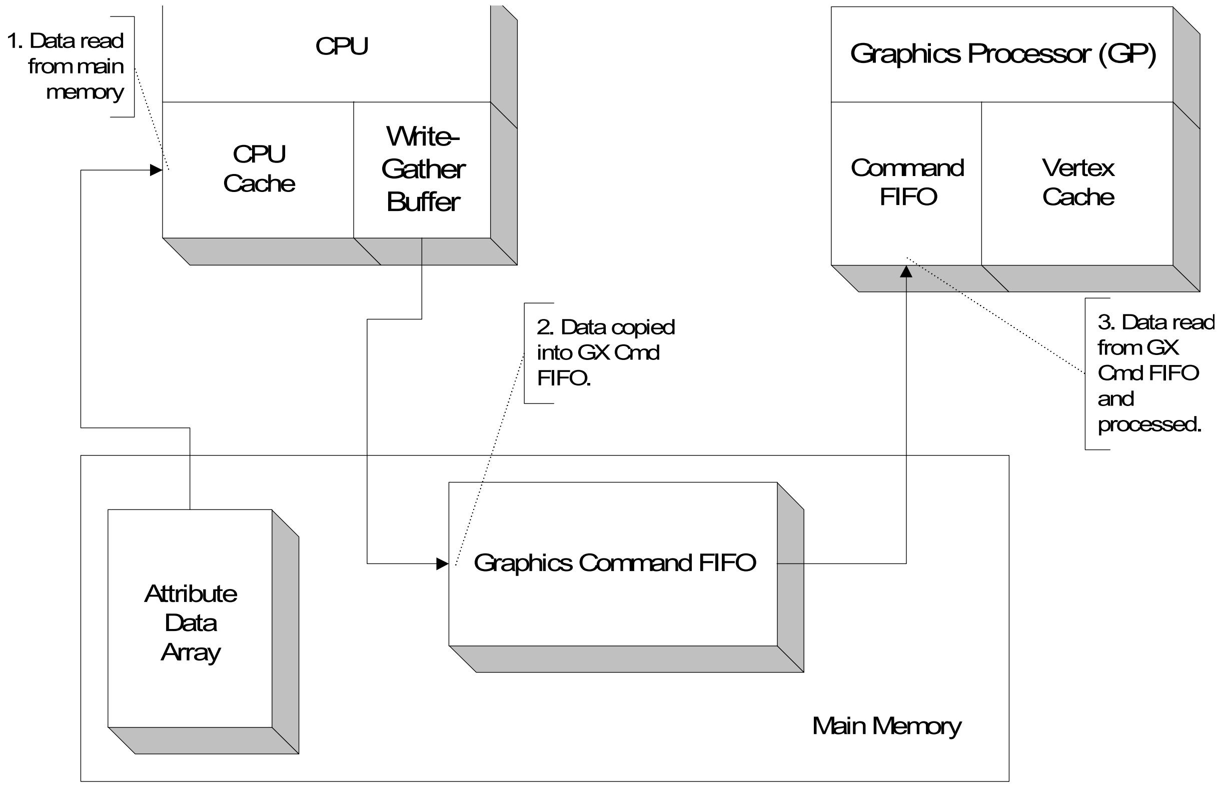
Code 4-8 Indexed vs. Direct Compression Example

```
//
//  Indexed version of textured cube
//
// position data
f32 MyPos[] =
{
     100.0,  100.0,  100.0,
     100.0,  100.0, -100.0,
     100.0, -100.0,  100.0,
     100.0, -100.0, -100.0,
    -100.0,  100.0,  100.0,
    -100.0,  100.0, -100.0,
    -100.0, -100.0,  100.0,
    -100.0, -100.0, -100.0
};
// texture data
u16 MyTex[] =
{
    0x0000, 0x0000,
    0x0000, 0x0f00,
    0x0f00, 0x0000,
    0x0f00, 0x0f00
};
//
// draw 6 sides of cube, 2 x 8-bit index per vert
// 6 sides x 4 verts x 1B/indx x 2 indx/vert = 48B
// 8 pos x 3 f32 x 4B/f32 = 96B
// 4 texcoord x 2 u16/texcoord x 2B/u16 = 16B
// ---------------------------------
//  total = 48B + 96B + 16B = 160B


//
//  Direct version of textured cube
//
// draw 6 sides of cube, 4 vertex each
// 1 pos x 3 f32/pos x 4B/f32 + 1 texcoord x 2 u16/texcoord x 2B/u16 = 16B/vtx
// ---------------------------------
// total = 24 vtx x 16B/vtx = 384B
```

The indexed attribute arrays can be compressed, removing duplicate attribute data. Also, you can order the data for animation or collision processing so that it will be loaded efficiently into the CPU cache. Keep in mind that the graphics chip is not cache-coherent with the CPU. In other words, before the GP accesses the data, you must explicitly flush the CPU cache of any attribute array data it has accessed previously (see OS function `DCStoreRange`). Also, you must invalidate the vertex cache using `GXInvalidateVtxCache` if you relocate or modify an array of vertex data that is read by, or may be cached by, the vertex cache.

The Wii system also supports *direct* data, which is copied directly into the graphics FIFO (see "13 Graphics FIFO" on page 139). The hardware gets the data from the FIFO and sends it down the pipeline; it does not go through the vertex cache.

Figure 4-8 Flow of Direct Vertex Data



In addition, it is possible to use indexed and direct access together for a single vertex.

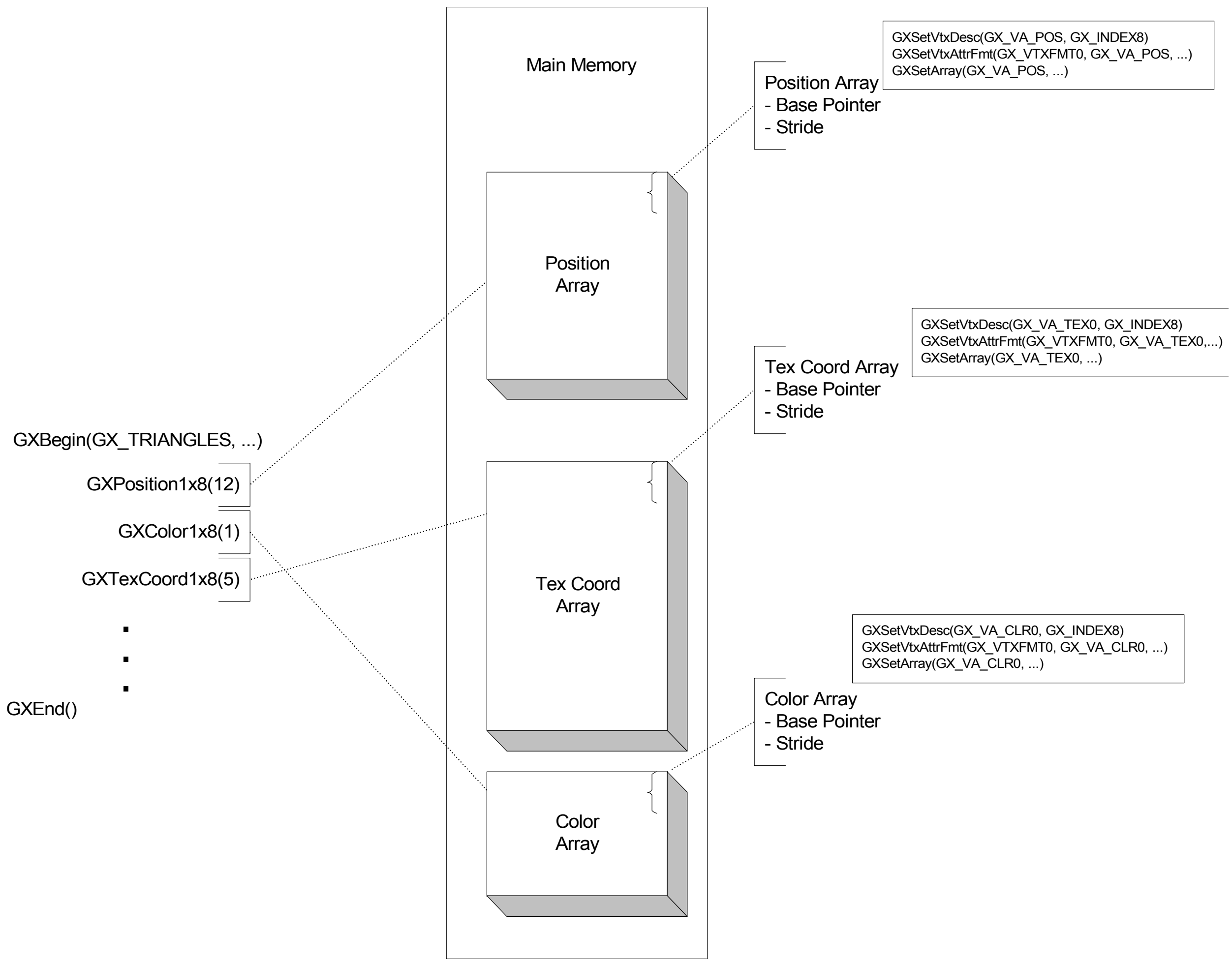
Finally, the Wii system can generate new vertex data based on the existing vertex data in hardware. For example, you can generate texture coordinates from position. This can be considered another form of data compression. See "7 Texture Coordinate Generation" on page 53 for more information.

4.5.1 Indexed Vertex Data

In addition to specifying the vertex attribute descriptor (using `GXSetVtxDesc`) to indicate that an attribute is indexed, you must also specify a pointer to the array of attribute data and the stride in bytes between successive elements in the array.

There is a unique base pointer and stride for every attribute type. The pointer and stride are set using the `GXSetArray` function. You can also use `GXSetArray` for setting pointers and strides for indexed light arrays and matrix arrays.

Figure 4-9 Indexed Vertex Data



Code 4-9 GXSetArray

```
s8 Verts8[9] = {       -100, 100, 0,
                        100, 100, 0,
                       -100, -100, 0 };

u32 Colors[3] = {
0xFF000000,
0x00FF0000,
0x0000FF00 };

GXSetArray(GX_VA_POS, (u32)Verts8, 3);
GXSetArray(GX_VA_CLR0, (u32)Colors, sizeof(u32));
// …
```

The stride parameter also allows you to handle structures such as the following.

Code 4-10 Arrays of Vertex Structures

```
typedef struct {
  u8  x,y,z;
  u16 s,t;
  u32 rgba;
  u32 private_data;
} MyVert;

MyVert myverts[100];

// …
GXSetArray(GX_VA_POS, &myverts[0].x, sizeof(MyVert));
GXSetArray(GX_VA_CLR0, &myverts[0].rgba, sizeof(MyVert));
GXSetArray(GX_VA_TEX0, &myverts[0].s, sizeof(MyVert));

GXSetVtxDesc(GX_VA_POS, GX_INDEX8);
GXSetVtxDesc(GX_VA_CLR0, GX_INDEX8);
GXSetVtxDesc(GX_VA_TEX0, GX_INDEX8);

GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_POS, GX_POS_XYZ, GX_U8, 2);
GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_CLR0, GX_CLR_RGBA, GX_RGBA8, 0);
GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_TEX0, GX_TEX_ST, GX_S16, 4);
```

4.5.2 Direct Vertex Data

When the vertex descriptor for an attribute is set to GX_DIRECT, the vertex function will copy the vertex data into the graphics FIFO (see "13 Graphics FIFO" on page 139). This is different from indexed primitives, in which the vertex function copies an index to the data into the graphics FIFO, and the data is read directly from main memory by the Graphics Processor.

Direct data is coherent with the CPU cache, since it is copied through it.

Direct data is also useful in cases where the data you want to send is already in the cache. For example, when a vertex's position coordinates are calculated and generated by the CPU, it may be more efficient to write them to the graphics FIFO directly, rather than index them from an array. On the other hand, direct data uses more bandwidth because each element must be read into the CPU cache, written to the graphics FIFO, and then read out of the FIFO again into the Graphics Processor. With indexed data, you write the index into the FIFO and only read the data if there is a miss in the vertex cache.

Code 4-11 Direct Vertex Data

```
GXClearVtxDesc();
GXSetVtxDesc(GX_VA_POS, GX_DIRECT);
GXSetVtxDesc(GX_VA_CLR0, GX_DIRECT);

GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_CLR0, GX_CLR_RGB, GX_RGB8,  0);
GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_POS,  GX_POS_XYZ, GX_F32, 0);

GXBegin(GX_TRIANGLES, GX_VTXFMT0, 3);
  // vert 0
  GXPosition3f32( 100.0, 100.0, 0.0 );
  GXColor3u8( 0xff, 0x00, 0x00 );
  // vert 1
  GXPosition3f32(0.0, 0.0, 0.0);
  GXColor3u8( 0x00, 0xff, 0x00 );
  // vert 2
  GXPosition3f32(100.0, 0.0, 0.0);
  GXColor3u8( 0x00, 0x00, 0xff);
GXEnd();
```

4.5.3 Mixture of Direct and Indexed Data

Indexed data may be mixed with direct data in a vertex format. Sometimes, the data size may be small and since each vertex is unique, it is not worth indexing. This data can be sent directly while using indexing for other attributes:

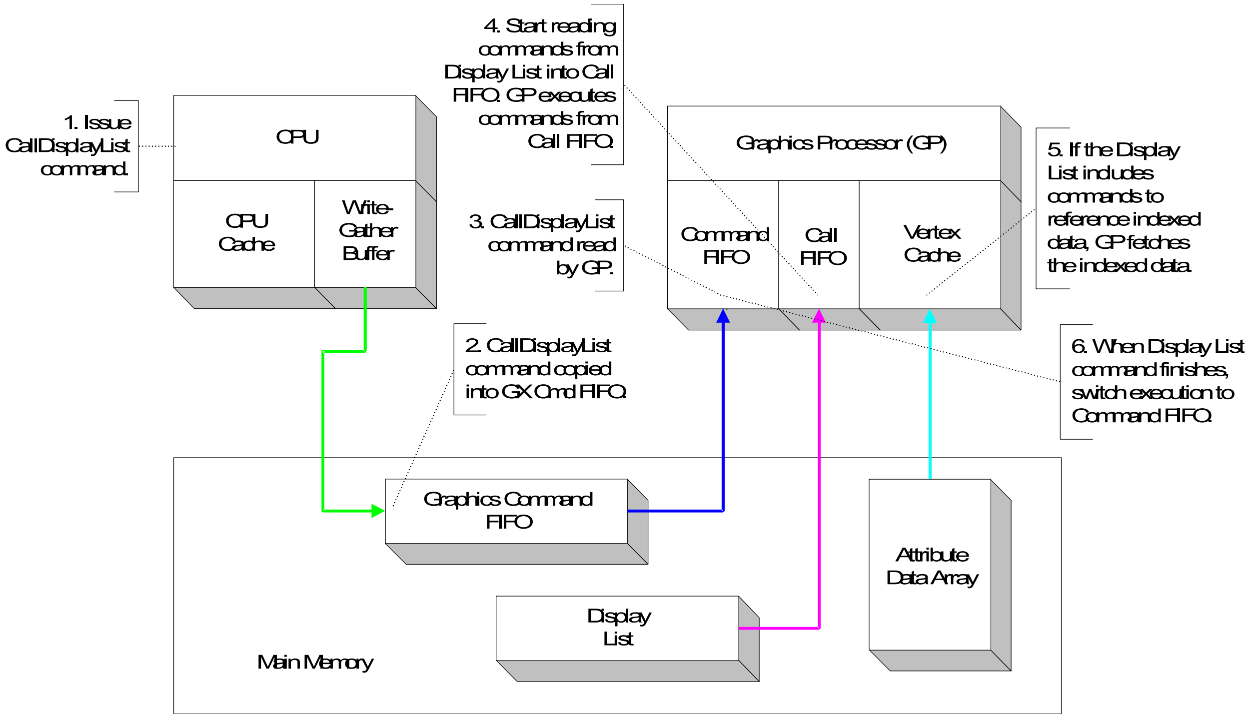
Code 4-12 Mixture of Direct and Indexed Data

```
GXClearVtxDesc();
GXSetVtxDesc(GX_VA_POS, GX_INDEX8);
GXSetVtxDesc(GX_VA_CLR0, GX_DIRECT);

GXSetArray(GX_VA_POS, &MyPos[0], sizeof(f32)*3);

GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_CLR0, GX_CLR_RGB, GX_RGB565, 0);
GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_POS,  GX_POS_XYZ, GX_F32,  0);

GXBegin(GX_TRIANGLES, GX_VTXFMT0, 3);
  // vert 0
  GXPosition1x8( 1 );     // this is an index, use the 'x' type
  GXColor1u16( 0xf551 );  // this is color data, not an index
  // vert 1
  GXPosition1x8( 0 );
  GXColor1u16 ( 0x3243 );
  // vert 2
  GXPosition1x8( 2 );
  GXColor1u16 ( 0x1897 );
GXEnd();
```

4.6 Display Lists

A display list is a pre-compiled list of primitive-rendering or state-setting commands. Once the list has been created, the GP can access it directly and process the commands as many times as needed. This provides tremendous savings in memory bandwidth, compared to having the CPU re-create and send primitives every time. To get optimum performance from the system, therefore, requires the use of display lists.

Figure 4-10 Display List Flow



4.6.1 Creating Display Lists

Display lists may be created in various ways:

- By calling `GXBeginDisplayList`, GX primitive commands, and `GXEndDisplayList`, in that order.
- By creating arrays containing display list command tokens and data.

4.6.1.1 Using GXBeginDisplayList and GXEndDisplayList

Code 4-13 GXBeginDisplayList

```
void GXBeginDisplayList(
    void      *list,
    u32       size);
```

The `GXBeginDisplayList` function prevents display lists from starting and disables writes to the FIFO currently attached to the CPU. After this function has been called, the functions in the GX API that usually send commands and data to the CPU FIFO will send commands and data to a display list buffer instead, until the `GXEndDisplayList` function is called. Running the `GXEndDisplayList` function will re-enable writes to the CPU FIFO. The *list* argument is the starting address of the display list buffer. The *size* argument indicates the number of bytes available in the allocated space for writing display list commands; it allows the system to check for overflow.

The application must allocate the memory for the display list buffer. If the display list exceeds the maximum size of the buffer, *size*, the `GXEndDisplayList` function will return an error. The buffer address is 32-byte aligned. Use the memory allocation functions provided by the OS library and MEM library to allocate a memory region for the display list buffer that is guaranteed to be 32-byte aligned.

The CPU write-gather pipe is used to write graphics commands to a display list. Consequently, information in the CPU cache must be flushed before beginning a write. This is done because the CPU write-gather buffer (the buffer used to write GP commands into memory) is not cache coherent. You can call the `DCInvalidateRange` function, provided by the OS library, to ensure that the cache data in this memory region is deleted. Additionally, the mechanism that flushes the write-gather pipe requires that the display list buffer be at least 32 bytes larger than the total maximum data value expected to be stored. In order to make the pointer to the write-gather pipe 32-byte aligned and always push graphics commands into the display list buffer from its starting address, the `GXBeginDisplayList` function will call the `GXResetWriteGatherPipe` function internally and reset the write-gather pipe.

Display lists cannot be nested. This means that the `GXBeginDisplayList`, `GXCallDisplayList`, and `GXEndDisplayList` functions cannot be invoked between a call to the `GXBeginDisplayList` and `GXEndDisplayList` functions. Also, the `GXBeginDisplayList` function calls the `GXFlush` function, so the `GXFlush` function does not need to be called explicitly after running the `GXBeginDisplayList` function.

For the most part, GX API commands can be sent to a display list. However, it is possible to bypass the consistency of the states controlled by the GX API when display lists are used at runtime. This sometimes causes unexpected behavior, stops the graphics pipeline, or creates some other problem, resulting in a state collision or conflict. The most recommended safe method of sending GX API commands is to send only primitives which will not cause state collisions, within the range which can be bracketed by the `GXBegin` and `GXEnd` functions.

4.6.1.2 Creating Arrays Containing Display List Commands

Appendix C describes some of the command tokens that go into a display list. You can put such tokens and the associated data directly into an array. When creating an array whose elements are defined at compile time, use the `ATTRIBUTE_ALIGN(32)` macro to guarantee proper alignment. You must also be sure to pad the length of the array to a multiple of 32 bytes using `GX_NOP` commands, as shown in Code 4-14.

Code 4-14 Sample Array Containing Display List

```
u8 OneTriDL[] ATTRIBUTE_ALIGN(32) =
{
    (GX_DRAW_QUADS | GX_VTXFMT0),      // command, primitive type | vat idx
    0, 36,                             // number of verts, 16b
    8,  0,  7, 0,  2, 0,  3, 0,        // quad 0
    1,  1,  2, 1,  7, 1,  6, 1,        // quad 1
    1,  2,  0, 2,  9, 2, 10, 2,        // quad 2
    4,  1, 1,  1, 10, 1, 11, 1,        // quad 3
    1,  2, 12, 2, 13, 2,  2, 2,        // quad 4
    2,  0, 13, 0, 14, 0,  5, 0,        // quad 5
    18, 2, 15, 2, 16, 2, 17, 2,        // quad 6
    20, 1, 17, 1, 16, 1, 19, 1,        // quad 7
    20, 0, 21, 0, 18, 0, 17, 0,        // quad 8
    GX_NOP, GX_NOP, GX_NOP, GX_NOP, GX_NOP, GX_NOP, GX_NOP, // pad
    GX_NOP, GX_NOP, GX_NOP, GX_NOP, GX_NOP, GX_NOP, GX_NOP, // pad
    GX_NOP, GX_NOP, GX_NOP, GX_NOP, GX_NOP, GX_NOP, GX_NOP  // pad to 32B
};
```

Arrays created in this manner are loaded into memory via the disc system, and thus should be guaranteed to be resident in memory and not in any CPU cache, so no special cache flushing is required.

In addition, display list arrays can be created dynamically. Use the memory allocation functions provided with the OS and MEM libraries to allocate a memory region guaranteed to be 32-byte aligned for the display list array. Next, fill the array with the desired command tokens and data (remember, of course, to pad the end of the command stream to a 32-byte boundary using `GX_NOP` commands). Finally, the array must be flushed from the CPU's data cache before the Graphics Processor can call it. You can use the `DCStoreRange` (or `DCFlushRange`) function, provided by the OS library, to do this.

4.6.2 Drawing Primitives Using Display Lists

Once a display list has been created, you can call it using `GXCallDisplayList`, as shown in Code 4-15.

Code 4-15 GXCallDisplayList

```
void GXCallDisplayList(
    void      *list,
    u32       size);
```

This call takes the size of the display list to increase the efficiency of display list processing and obviate the need for an explicit return command.

As shown in "Figure 4-10 Display List Flow" on page 28, the Graphics Processor has its own logic to handle `GXCallDisplayList` commands. CPU involvement is not necessary to change over the FIFO source, since the GP handles this (and in fact, has separate internal FIFOs for mainstream graphics commands vs. display list commands). This allows the GP to handle display lists very efficiently.

4.6.3 Effect on Machine State

A call to `GXCallDisplayList` does not perform any state pushing and popping. The only effect is to temporarily change the source of graphics commands from the original source to the display list. Therefore, any state that is changed during a display list call remains changed after the display list has been processed.

Display lists that include state-changing commands have further complications. Certain state registers in the GP include more than one piece of state. However, when writing to such a register, all of the included state is affected. You cannot write to only certain bits of a register without writing to all the other bits.

The GX API maintains shadow copies of the registers that are updated as the CPU processes GX commands. However, when the GP processes display lists, any state-changing commands in the display list will update the actual registers and not update the shadow copies of the registers. Consequently, the state-changing commands that occur in a display list (or after a display list) may unexpectedly affect other pieces of state as well. Also, calls to inquire about current state (which read the shadow registers) may return incorrect results.

As a result, one should be very careful about placing state-changing commands within display lists. You may want to separate state from geometry within a display list, or else limit the state-changing commands to ones that relate to the contained geometry. Geometry-only display lists can thus be used without worrying about side effects, and the user need only pay special attention when using state-changing display lists.

There is one further complication. Not all of the GX commands act immediately ("act" means inserting commands into the current FIFO or command buffer). Instead, these GX commands will only set internal variables maintained by the GX library; the actual GP commands will not be sent out until a GXBegin command is called. This is known as "lazy evaluation," and the purpose is to avoid sending unnecessary commands that could slow the system down. This "lazy state" is also flushed by `GXBeginDisplayList` before the display list is started, by `GXEndDisplayList` before the display list is finished, and by `GXCallDisplayList` before the display list is called. Currently, the lazy state includes the VCD/VAT registers, the texture-coordinate scale registers, and the `GEN_MODE` register. The latter contains bits indicating the number of active TEV and indirect stages, the number of active textures, the number of rasterized colors, back/front culling mode, and antialiasing mode (these features are described in later chapters).

Appendix C includes more information on display-list format, and it details which pieces of state are tied together within the hardware registers.

4.7 GXDraw Functions

A number of basic 3D objects can be drawn using functions provided by GX. The following are provided:

- Cylinder
- Torus
- Sphere (iterated)
- Sphere (recursive)
- Cube
- Dodecahedron
- Octahedron
- Icosahedron
- Normal table

The following features are common:

- All shapes fit tightly within X = +/- 1, Y = +/- 1, Z = +/- 1.
- The shapes are typically symmetric around the Z axis.
- All functions save and restore the VCD and VAT/VTXFMT3.
- Positions and normals are provided for all shapes.

Texture coordinates may be provided for the torus, iterated sphere, and cube. They will be sent if the VCD had texture coordinates enabled prior to the function being called. Similarly, NBT normals may be provided for the cube.

These are the functions themselves:

Code 4-16 GX Draw Functions

```
void GXDrawCylinder(u8 numEdges);
void GXDrawTorus(f32 rc, u8 numc, u8 numt);
void GXDrawSphere(u8 numMajor, u8 numMinor);
void GXDrawCube(void);
void GXDrawDodeca(void);
void GXDrawOctahedron( void );
void GXDrawIcosahedron( void );
void GXDrawSphere1( u8 depth );
u32  GXGenNormalTable( u8 depth, f32* table );
```

`GXDrawTorus` takes arguments specifying the radius of the cross-section (such as the “fatness,” with 0 < ***rc*** < 1), the number of subdivisions around the cross-section, and the number of subdivisions around the overall torus. `GXDrawSphere` is the iterated sphere, and it takes arguments specifying the number of lateral subdivisions and the number of longitudinal subdivisions. `GXDrawSphere1` is the recursive sphere;

it is generated by recursively subdividing an icosahedron to the specified depth. `GXGenNormalTable` allows a normal table to be generated by recursive subdivision of an icosahedron. You specify the recursion depth and a pointer to memory to store the table. The function returns the total number of normals generated.

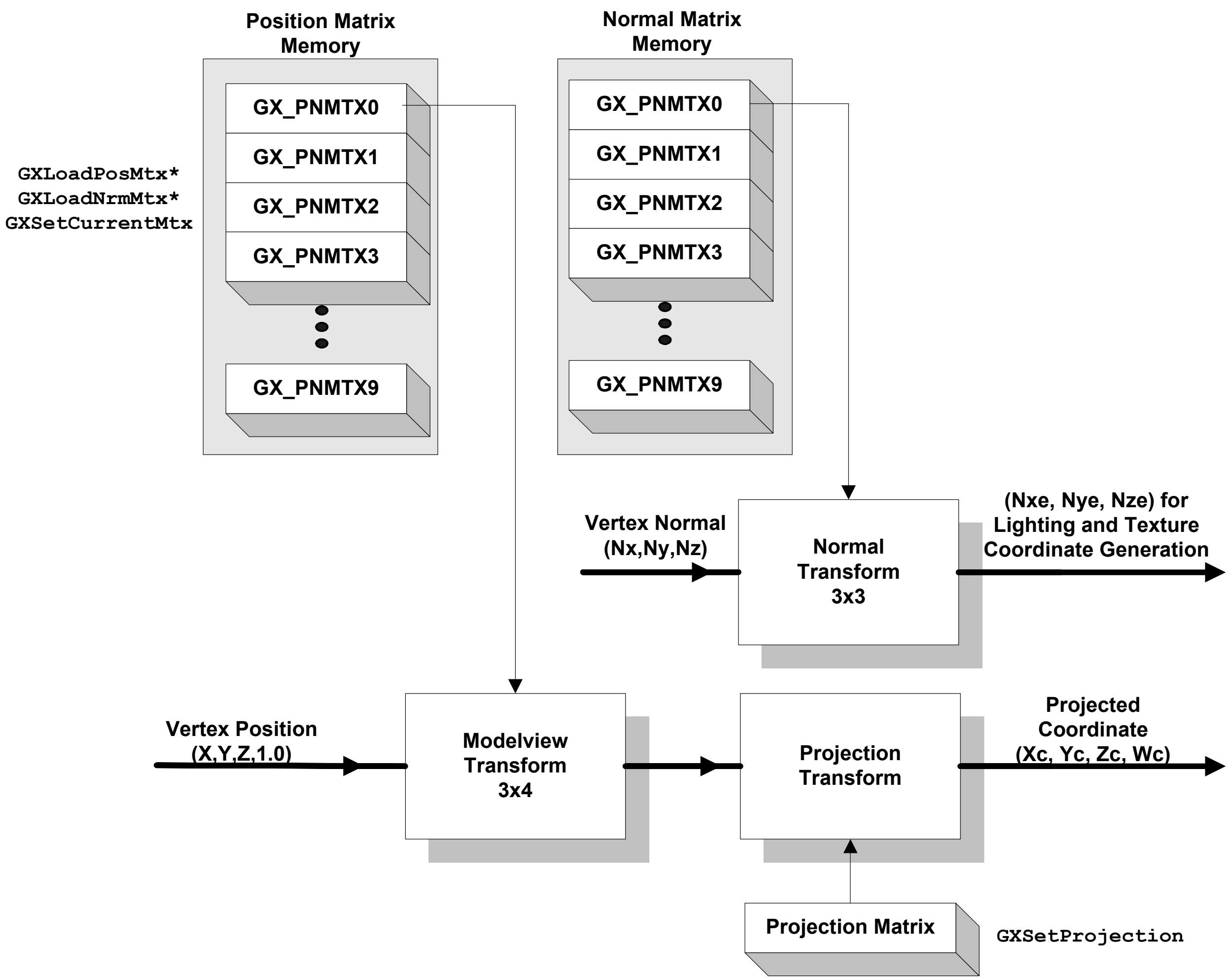
5 Viewing

This chapter describes the transformation section of the Graphics Processor.

The GP has an internal matrix memory. Programmers can load several matrices into the memory and specify one of them as the default matrix using `GXSetCurrentMtx`. Another way to specify a matrix is to provide a per-vertex matrix index. Vertices that do not specify a modelview matrix index will use the current matrix. When a matrix index is specified, the index used becomes the current matrix index and overwrites the index set by `GXSetCurrentMtx`.

The same matrix memory *index* that specifies the modelview matrix that transforms the vertex position also specifies the normal matrix when lighting is enabled. The picture below is somewhat simplified. Refer to "5.6 How to Override the Default Matrix Memory Configuration" on page 39 for more details on the matrix configuration.

Figure 5-1 Modelview and Projection Data Path



5.1 Loading a Modelview Matrix

The functions shown below are used to load a position matrix and to specify which matrix should be used:

Code 5-1 GXLoadPosMtxImm

```
GXLoadPosMtxImm(&v, GX_PNMTX0);
GXSetCurrentMtx(GX_PNMTX0);
```

Assuming an implicit *W* of 1.0, the basic vertex position transform from object or model space to homogeneous eye space is as shown in Equasion 5-1.

Equation 5-1 Vertex position transform

$$\begin{bmatrix} X_e \\ Y_e \\ Z_e \end{bmatrix} = \left[TransformMatrix(n)(3x4) \right] \times \begin{bmatrix} X \\ Y \\ Z \\ 1.0 \end{bmatrix}$$

The matrix memory is configured by default to contain:

- 10 position and normal matrix pairs (`GX_PNMTX0-9`), described by the enumeration `GXPosNrmMtx`.
- 10 texture matrices (`GX_TEXMTX0-9`), described by the enumeration `GXTexMtx`.
- An identity matrix (`GX_IDENTITY`), also described by the enumeration `GXTexMtx`.

All matrices use floating point data.

The normal matrices are used for vertex lighting (see "6 Vertex Lighting" on page 41). The texture matrices are used for various texture coordinate operations ("7 Texture Coordinate Generation" on page 53). In the example `onetri.c`, `GXInit` sets the default matrix to `GX_PNMTX0`.

The normal transform is similar to the position transform; however, translation is neither required nor useful. Consequently, the matrix does not need to convert a homogeneous normal. The normal is not transformed by the projection and screen space conversions. The `GXLoadNrmMtxImm` function converts a 3x4 matrix into a 3x3 matrix and loads it into normal matrix memory. This assumes that the normal matrix is the inverse transpose of the modelview matrix, which is usually a 3x4 matrix. The functions `GXLoadNrmMtxImm3x3` and `GXLoadNrmMtxIndx3x3` may be used to load a normal matrix directly from a 3x3 matrix in main memory. There is no function to do an indexed load of a 3x3 matrix from an array of 3x4 matrices.

The transformed normal is re-normalized before being used in the lighting calculations.

Equation 5-2 Vertex Normal Transform

$$\begin{bmatrix} N_{xe} \\ N_{ye} \\ N_{ze} \end{bmatrix} = \left[NormalMatrix(n)(3x3) \right] \times \begin{bmatrix} N_x \\ N_y \\ N_z \end{bmatrix}$$

A matrix can be loaded either by copying it directly into the Graphics FIFO (`GXLoadPosMtxImm`) or by indexing a matrix array in main memory (`GXLoadPosMtxIndx`). Indexed matrices are loaded directly by the graphics hardware. The matrix data is never sent through the Graphics FIFO.

Note: Indexed matrix loads have cache coherency issues; see Appendix E.

When using index matrices, you must first set up a *base pointer* and *stride* for the array you wish to access using the `GXSetArray` function. The base pointer is the address of the first element in the array (index = 0). The stride is the number of bytes between successive elements of interest in an array. For example, an array of 3x4 floating point matrices would have a stride of 48 bytes = 3 rows x 4 columns x 4 bytes/float. The stride also allows the programmer to index arrays of structures that have additional data in the same manner as indexing a normal array of matrices. The maximum stride accepted by `GXSetArray` is 255 bytes. Use the attributes `GX_POS_MTX_ARRAY`, `GX_NRM_MTX_ARRAY`, and `GX_TEX_MTX_ARRAY` to specify a matrix array.

Matrix memory is not a matrix *stack*. It is assumed that the application will manage matrix stacks in main memory and concatenate matrices using the CPU. The loaded matrix should transform the object to be drawn from local model space to view space. The `MTX` library supports creating and manipulating matrix stacks.

It is the application's responsibility to manage the matrix memory; the GX API simply provides a mechanism for loading and using matrices.

A utility library is provided for matrix and vector math functions; see *Matrix-Vector Library (MTX)*.

5.2 Setting a Projection Matrix

The GP performs standard projection transforms for each vertex position. The projection is done separately from, and after, the modelview transform in the GP. The `GXSetProjection` function will load a single projection matrix.

Code 5-2 GXSetProjection

```
void GXSetProjection(f32 mtx[4][4], GXProjectionType type);
```

The first argument is a pointer to a 4x4 projection matrix (see *Matrix-Vector Library (MTX)* for more information on matrix format and construction). The second argument specifies whether the matrix is perspective (`GX_PERSPECTIVE`) or orthographic (`GX_ORTHOGRAPHIC`). The projection transform hardware assumes the forms shown in Equation 5-3 and Equation 5-4 for the projection matrix:

Equation 5-3 Perspective Projection

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ W_c \end{bmatrix} = \begin{bmatrix} p0 & 0 & p1 & 0 \\ 0 & p2 & p3 & 0 \\ 0 & 0 & p4 & p5 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} X_e \\ Y_e \\ Z_e \\ 1 \end{bmatrix}$$

Equation 5-4 Orthographic Projection

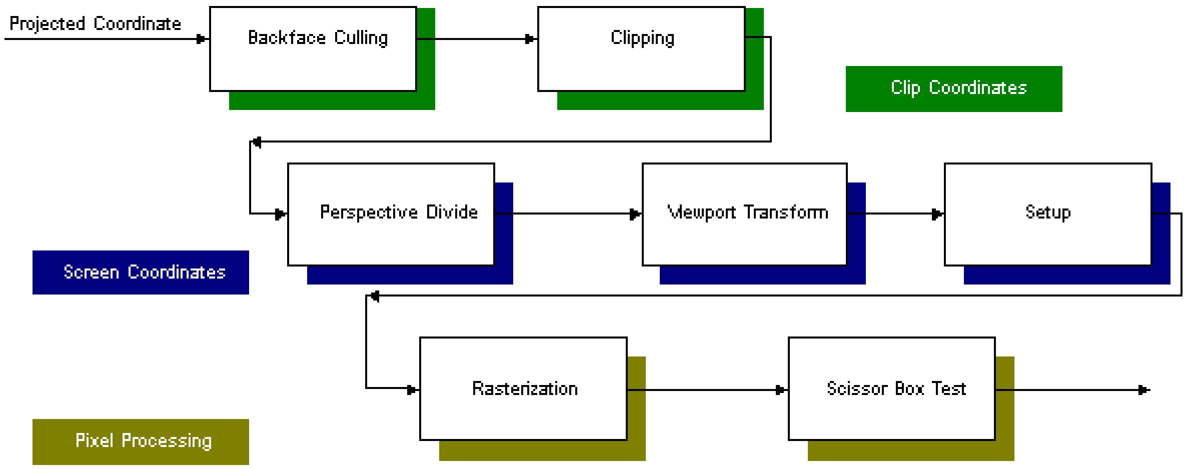
$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ W_c \end{bmatrix} = \begin{bmatrix} p0 & 0 & 0 & p1 \\ 0 & p2 & 0 & p3 \\ 0 & 0 & p4 & p5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_e \\ Y_e \\ Z_e \\ 1 \end{bmatrix}$$

The Matrix-Vector library contains functions to help set up perspective and orthographic projection matrices conveniently. They include `MTXFrustum`, `MTXPerspective`, and `MTXOrtho`.

The resultant homogeneous coordinates are in clip space. Once the clip space coordinates are obtained, *1.0/Wc* is computed for converting to non-homogenous coordinates. Once this is done, the (*x, y, z*) coordinates will lie within the normalized space of (*[-1…+1]*, *[-1…+1]*, *[-1…0]*).

5.3 Culling, Clipping, and Scissoring

Figure 5-2 Clipping and Culling Data Path



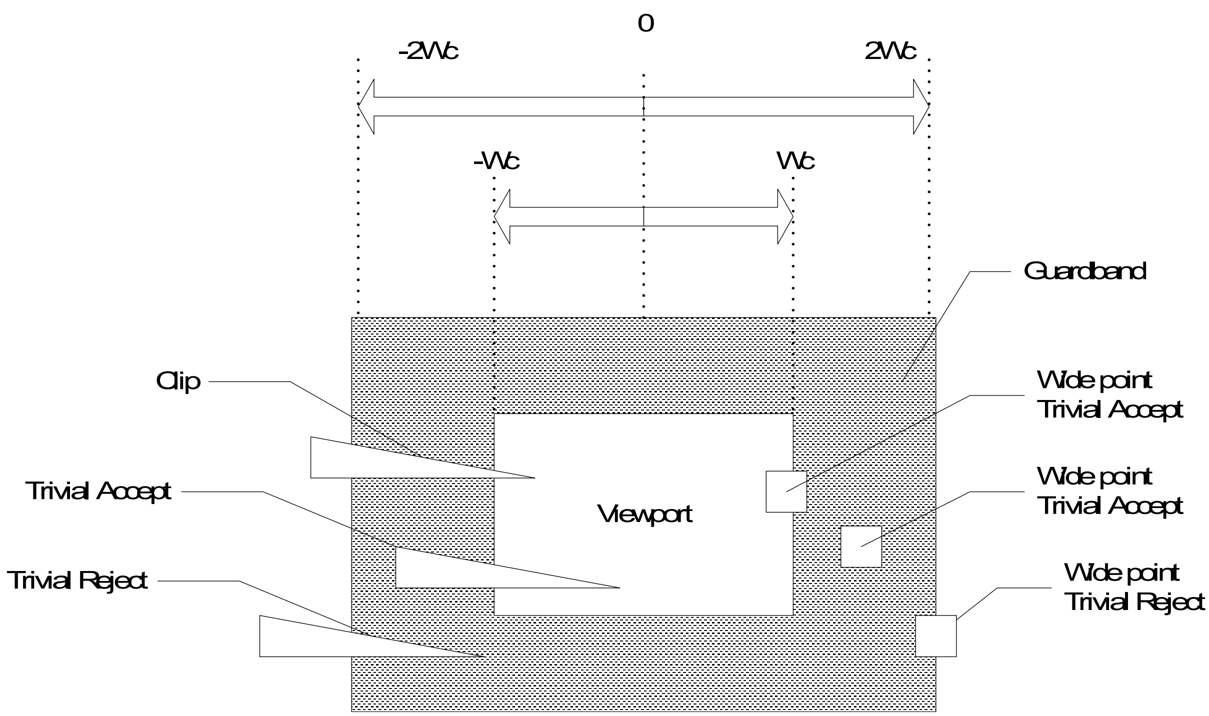
The coordinates resulting from the projection transform are in clip space [*Xc, Yc, Zc, Wc*]. The hardware will conditionally reject triangles that are frontfacing, backfacing, or both front- and backfacing as set by the `GXSetCullMode` function. As stated earlier, frontfacing triangles are those whose vertices appear onscreen in clockwise order.

Guardband clipping to *+/-2Wc* reduces the amount of clipping in the transform unit. The rasterization process uses evaluation to compute only pixels that are within the visible screen, so there is no fill rate penalty for this type of clipping.

Triangles are clipped when they are both outside the guardband region and inside the viewport. Other triangles will be either totally out of the viewport (trivially rejected), totally inside the viewport (trivially accepted), or partially inside the viewport (trivially accepted but scissored).

Points are trivially accepted when they are within +/-*2Wc* and are trivially rejected when they are outside +/ -*2Wc*. This prevents wide points from being trivially rejected when the vertex is outside of *+/-Wc.* In this case, due to the point's width, some of it may be visible. Lines are also only trivially rejected when they are outside +/-*2Wc*.

Figure 5-3 Clip Coordinates



After clipping, the vertex (*x, y, z*) is perspective-divided. The viewport transform converts the coordinates into screen space.

Actual clipping is a very slow procedure that creates stalls in the transform engine, thus it should be avoided whenever possible. Clipping can be disabled by the function `GXSetClipMode`. There is a guard-band at the far clipping plane that makes it acceptable to turn off clipping for most far-clipped objects. However, disabling clipping for near-clipped objects results in incorrectly drawn polygons.

`GXInit` enables backface culling and sets the viewport and scissor box to full screen size.

5.4 Viewport and Scissoring

The following equation performs the conversion from clip space to screen space and perspective scaling:

Equation 5-5 Clip Space to Screen Space Conversion

$$\begin{bmatrix} X_s \\ Y_s \\ Z_s \end{bmatrix} = \frac{1.0}{W_c} * \begin{bmatrix} X_c * X_{scale} \\ Y_c * Y_{scale} \\ Z_c * Z_{scale} \end{bmatrix} + \begin{bmatrix} X_{offset} \\ Y_{offset} \\ Z_{offset} \end{bmatrix}$$

The resultant screen space coordinate is then sent to the setup unit for rasterization. The *1/Wc* value is also sent to the setup unit for texture space computations.

The viewport is set using the following function:

Code 5-3 GXSetViewport

```
GXSetViewport(
    f32 xOrig,
    f32 yOrig,
    f32 width,
    f32 height,
    f32 nearZ,
    f32 farZ );
```

The screen space origin (0, 0) is at the top-left corner of the display. The screen space scale and offset are computed using floating-point values. This is used to advantage in the function `GXSetViewportJitter` to jitter the viewport by half a line in field rendering modes.

`GXSetScissor` sets the scissor box, typically to the same size as the viewport.

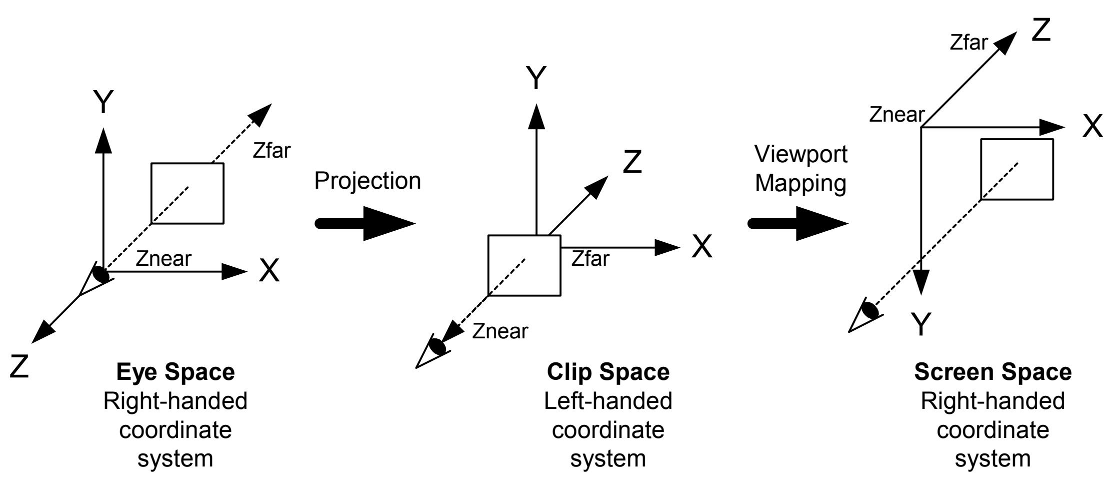
Code 5-4 GXSetScissor

```
GXSetScissor(
    u32 left,
    u32 top,
    u32 width,
    u32 height);
```

5.5 Coordinate Systems

The GX API assumes a right-handed coordinate system for model and eye space. The eye in eye space is assumed to be looking down the negative Z axis. In order to map eye space into the viewport, the coordinates undergo two changes of coordinate systems. The first occurs during projection into clip space (assuming the MTX library projection routines are used). As a result of this transformation, the Z axis is flipped, with *Znear* mapped to *-W* and *Zfar* mapped to zero. The second change occurs during the viewport mapping into screen space. In this mapping, the Y axis is flipped, and the Z values are offset such that *Znear* maps to the viewport's Z value that is close to the viewer, and *Zfar* maps to the viewport's Z value that is far from the viewer. The diagram below illustrates the process:

Figure 5-4 Coordinate System Transformations



The function GXProject is provided to transform a single point from object space to screen space. You pass it the object coordinate, the model-view matrix, the projection matrix, and the viewport. It returns the transformed point:

Code 5-5 GXProject

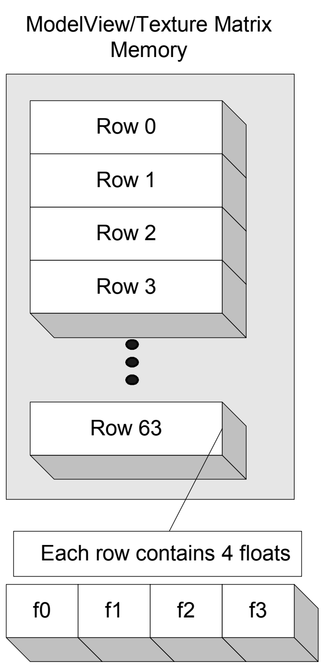
```
void GXProject (
    f32  x,           // object coordinates
    f32  y,
    f32  z,
    f32  mtx[3][4],   // model-view matrix
    f32* pm,          // projection matrix, as returned by GXGetProjectionv
    f32* vp,          // viewport, as returned by GXGetViewportv
    f32* sx,          // returned screen coordinates
    f32* sy,
    f32* sz );
```

5.6 How to Override the Default Matrix Memory Configuration

The GX API configures the matrix memory in a way that is useful for the majority of general applications. In some specific cases, you may want to override this default configuration. Here we describe the physical layout of matrix memory and the rules an application must follow to allocate the matrix memory successfully.

The matrix memory consists physically of three separate memories. The first is for modelview and texture matrices, the second is for normal matrices, and the third is matrix region for post-transform. For more on the matrix region for post-transform, see Chapter 7.

Figure 5-5 Matrix Memory



The modelview/texture matrix memory consists of 64 rows, each row consisting of four floats. A matrix is loaded as a set of contiguous rows in matrix memory. The index used by GXLoadPosMtx* and GXLoadTexMtx* to specify the matrix in matrix memory (GX_PNMTX0 or GX_TEXMTX5, for example) is actually the row address where the first row of the matrix is loaded. Modelview matrices loaded using GXLoadPosMtx* are assumed to have three rows (3x4 matrices). Texture matrices can have either two or three rows, specified in GXSetTexCoordGen. The matrices may be loaded starting at any row address.

The normal matrix memory consists of 32 rows, but each row contains only three floats. Since normals are not required to be homogeneous, only 3x3 matrices are needed. Normal matrices can be loaded from either a 3x4 matrix (`GXLoadNrmMtxImm`) or from a 3x3 matrix (`GXLoadNrmMtxImm3x3`) in main memory. Like modelview and texture matrices, normal matrices are indexed using the starting row address in normal matrix memory. However, normal matrices use the same index as modelview matrix, so the two must be allocated as a pair. Normally, the matrix used to transform the normal is the inverse transpose of the modelview matrix, so they are naturally pairs.

Since the modelview matrix index can address 64 rows, but the normal matrix index can address only 32 rows, the hardware computes the normal index as:

Equation 5-6 Normal Matrix Index

norm_mtx_indx = pos_mtx_indx % 32

For example, a modelview matrix row address of 33 or 1 will address the same normal matrix (at row address 1). If you wish to use modelview matrices beyond the first ten matrices (thirty rows), you need to leave a gap at row 30 and row 31. The eleventh modelview matrix must start at row 32 in order to line up with a normal matrix.

The default matrix configuration organizes the modelview/texture matrix memory as follows:

- Ten modelview matrices (3x4).
- Ten texture matrices (3x4).
- One identity matrix (3x4).

Sixty-three rows are used in this configuration.

6 Vertex Lighting

6.1 Lighting Pipeline

The Wii system supports lighting in hardware as a per-vertex calculation. This means that a color (RGB) value can be computed for every lit vertex, and that these colors are then linearly interpolated over the surface of each lit triangle (known as Gouraud shading).

Wii has full support for *diffuse local spotlights*. There is also some support for *infinite specular lighting*. This chapter focuses mainly on diffuse lighting. Specular lighting is covered in "6.6 Specular Lighting" on page 49.

6.1.1 Diffuse Lights, Diffuse Attenuation and Vertex Normals

The hardware supports *diffuse* attenuation. This means that the front of the object can be brighter than the sides, and the back darkest. Diffuse attenuation is the primary reason vertex normals are supported. For each vertex, the vertex normal (N) is compared against the vector between the vertex and light position.

This encompasses two important physical effects. First, surfaces on “the back” of an object receive no light. This can be seen as a simple self-shadowing technique—one that only works for convex objects. Second, surfaces facing the light are lit more or less, depending on the incident angle of the incoming light.

6.1.2 Local Lights and Range Attenuation

The hardware supports *local* lights. Local lights have a position within the world and possibly, a direction. In fact, each light must have a position. Using the position of each vertex and the position of the light, the hardware can perform per-vertex distance attenuation. This means that you can make the brightness of the light shining on an object decrease as the object moves away from the light.

6.1.3 Spotlights, Directional Lights and Angle Attenuation

The hardware supports directional lights, ranging from non-directional lights, to subtle directional effects, to highly directional *spotlights*. These effects are supported by angle attenuation. This means that vertices directly “in the beam” of the light can be made brighter than vertices outside the beam or behind the light.

Local diffuse lights can be both distance- and angle-attenuated (spotlights). By programming the proper lighting equation, you can obtain the attenuation value as an output color or alpha. This color or alpha can then be used in the Texture Environment (TEV) unit to attenuate projected texture lights.

The hardware supports eight *physical lights*. The programmer can describe the attenuation parameters, position, direction, and color of each light. The programmer can control up to four *physical color channels* that accumulate the result of the lighting equation. By associating lights with channels, the programmer can choose to sum the effect of multiple lights per vertex or combine them later in the TEV. The number of channels available to the TEV is set by `GXSetNumChans`. In some cases (for example, when using a color channel to generate texture coordinates), a light channel is computed but not output. If only one channel is available to the TEV, it is `GX_COLOR0A0`. The light channel `GX_COLOR1A1` is only available if two channels are output to the TEV.

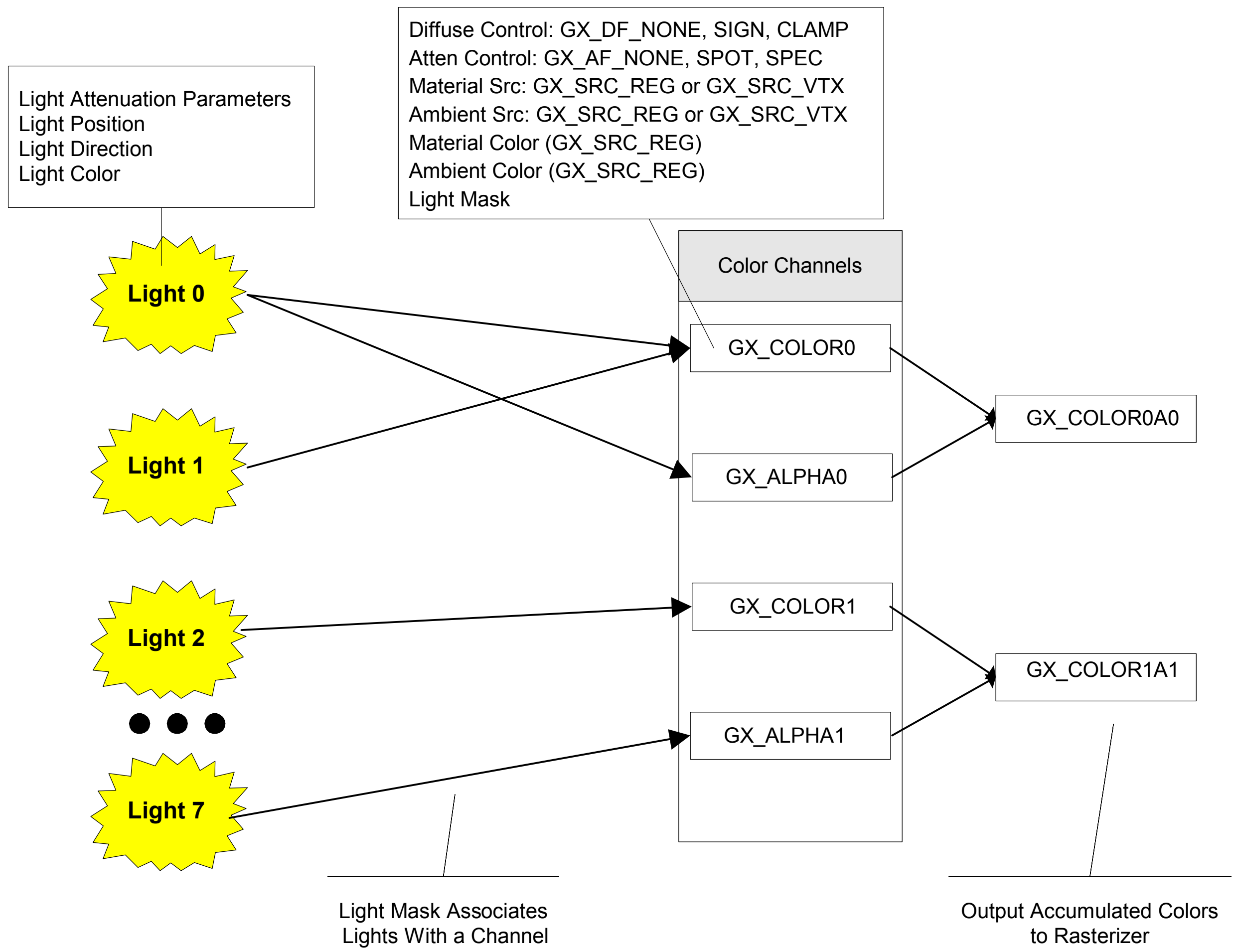
Code 6-1 GXSetNumChans

```
GXSetNumChans( u8 nChans );
```

Each color channel allows the enabling of attenuation and the selection of the color source. A light mask associates up to eight lights with the channel.

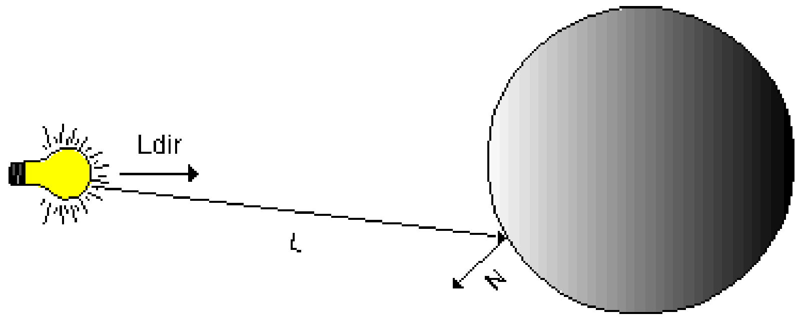
One light is supported at the peak vertex rate of 48.6 million vertices/second. With two texture coordinates per vertex and two lights, the peak vertex rate is 27 million vertices/second. With two texture coordinates per vertex and four lights, the peak vertex rate is 14.3 million vertices/second.

Figure 6-1 Associating Lights with Color Channels



6.2 Diffuse Lighting Equations

Figure 6-2 Lighting Vectors



Equation 6-1 Light Parameters

$$i = \{0_{RGB}, 0_A, 1_{RGB}, 1_A\}, ColorChannel$$
$$j = \{0,1,2,3,4,5,6,7\}, Light$$
$$l = LightParameter$$

Equation 6-2 Rasterized Color

$$RasterizedColor_{RGBA} = \begin{cases} ChannelColor_{i_{RGB}} \\ ChannelColor_{i_A} \end{cases}$$

Equation 6-3 Color Channel

$$ChannelColor_i = Material_i \times LightFunc_i$$

Equation 6-4 Material Source

$$Material_i = (MaterialSrc_i = GX\_SRC\_REG \; ? \; MaterialReg_i : VertexColor_i)$$

Equation 6-5 Channel Enable

$$LightFunc_i = \begin{cases} 1.0, & \text{if } LightFuncEnable_i = \text{FALSE} \\ Illum_i, & \text{if } LightFuncEnable_i = \text{TRUE} \end{cases}$$

Equation 6-6 Sum of Lights in a Channel

$$Illum_i = Clamp\left(Amb_i + SignedInt\left(\sum_{j=0}^{7} LightMask_i(j)Atten_i(j)DiffuseAtten_i(j)Color_j\right)\right)$$

Equation 6-7 Ambient Source

$$Amb_i = (AmbSrc_i = GX\_SRC\_REG \; ? \; AmbientReg_i : VertexColor_i)$$

Equation 6-8 Diffuse Attenuation

$$DiffuseAtten_i(j) = \begin{cases} 1.0, & if\ DiffAttenSelect_i = GX\_DF\_NONE \\ \hat{N} \cdot \vec{L}_j, & if\ DiffAttenSelect_i = GX\_DF\_SIGN \\ Clamp0(\hat{N} \cdot \vec{L}_j), & if\ DiffAttenSelect_i = GX\_DF\_CLAMP \end{cases}$$

Equation 6-9 Diffuse Angle and Distance Attenuation

$$Atten_l(j) = GX\_AF\_NONE\ ?\ 1 : \frac{Clamp0\left(a_{2_j} AAtt_l(j)^2 + a_{1_j} AAtt_l(j) + a_{0_j}\right)}{k_{2_j} d_l(j)^2 + k_{1_j} d_l(j) + k_{0_j}}$$

$$AAtt_l(j) = Clamp0(\vec{L}_j \cdot \vec{L}_{dir})\ //\ GX\_AF\_SPOT$$

$$d_l(j) = \sqrt{\vec{L}_j \cdot \vec{L}_j}\ //\ GX\_AF\_SPOT$$

6.3 Matrix Memory

To use per-vertex lighting, you must provide a normal with each vertex. In order for the hardware to transform the normal, you must provide a normal matrix. The normal matrix must be the inverse transpose of the modelview matrix. The position modelview and normal modelview matrix are indexed by a single index. In other words, they are considered to be a pair. The transformed normal is re-normalized before the lighting computations.

For directional lights, you must provide a light normal for each active light. It is the application's responsibility to transform the light normal and light position into view space when the viewpoint changes. The world-to-view matrix should be used to transform the light's position. The light's direction should be transformed by the inverse transpose of the world-to-view matrix. The light's direction is *not* normalized by the hardware; therefore, the application must ensure it is properly normalized.

6.4 Light Parameters

You may define the position, direction, attenuation factors, and color for each physical light. There are eight sets of physical light parameters. Light state is stored in a `GXLightObj` structure. The application is responsible for allocating the memory for a `GXLightObj`. The `GXInitLight*` functions can be used to initialize or modify the `GXLightObj` structure. The `GXLoadLightObjImm` or `GXLoadLightObjIndx` function is used to load the `GXLightObj` parameters into a physical light. `GXLoadLightObjIndx` has cache-coherence issues; see "E.3 Data Coherency" on page 182 for more details.

6.4.1 Angle Attenuation

The function `GXInitLightAttn` is used to initialize parameters used to compute angle and distance attenuation, as shown in "Equation 6-2 Rasterized Color" on page 43.

Code 6-2 GXInitLightAttn

```
void GXInitLightAttn(
        GXLightObj *lt_obj,
        f32         a0,
        f32         a1,
        f32         a2,
        f32         k0,
        f32         k1,
        f32         k2 );
```

The angle attenuation (`a0,a1,a2`) is a quadratic function of the cosine of the angle between the light direction and the light to vertex direction. By controlling the quadratic function's coefficients, you control the effective angle of the light.

A more convenient way of controlling the angle attenuation is provided by:

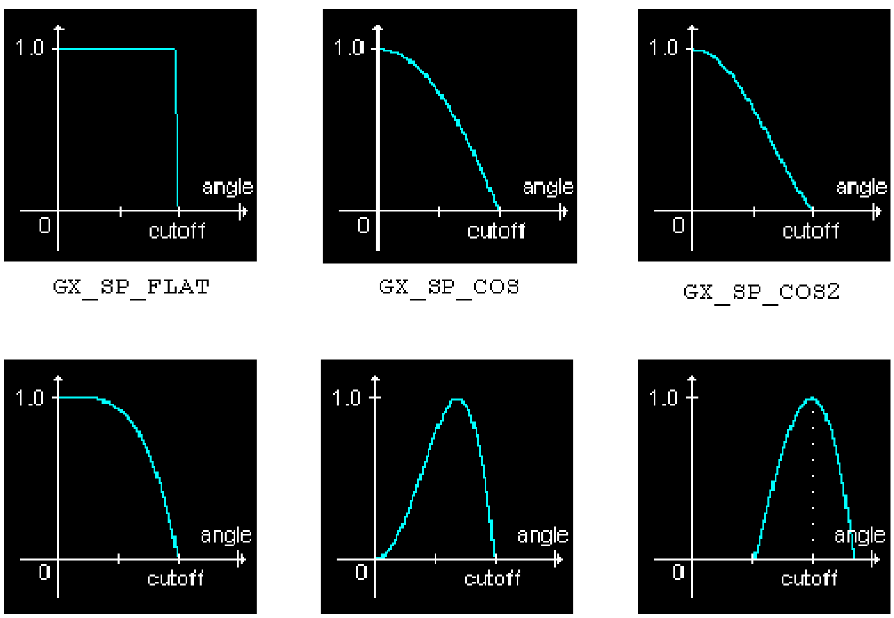
Code 6-3 GXInitLightSpot

```
GXInitLightSpot(
        GXLightObj*   lt_obj,
        f32           cutoff,
        GXSpotFn      spot_fn );
```

This function defines two easy-to-control parameters, rather than *a0, a1, a2* used by `GXInitLightAttn`. The parameter *cutoff* specifies cutoff angle of the spotlight in degrees. The spotlight works while the angle between the ray for a vertex and the light direction given by `GXInitLightDir` is smaller than this cutoff angle. The value for *cutoff* should be within *(0.0 < cutoff <= 90.0)*, otherwise given light object doesn't become a spotlight.

The parameter *spot_fn* defines type of the illumination distribution within cutoff angle. The following graphs show the curve shape of the distribution functions given by acceptable values for *spot_fn*. The value `GX_SP_OFF` turns the spotlight feature off even if the color channel setting is using `GX_AF_SPOT` (see `GXSetChanCtrl`).

Figure 6-3 Spotlight Functions



GX_SP_FLAT GX_SP_COS GX_SP_COS2

GX_SP_SHARP GX_SP_RING1 GX_SP_RING2

This function sets parameters only for angular attenuation. Parameters for distance attenuation should be set using `GXInitLightDistAttn`. You can also use `GXInitLightAttn`, but you have to care about the order for calling these functions because `GXInitLightAttn` overwrites parameters for both angle and distance attenuation.

6.4.2 Distance Attenuation

`GxInitLightAttn` can also be used to control the light's distance attenuation characteristics. As shown in "Equation 6-2 Rasterized Color" on page 43, the distance attenuation is an inverse quadratic function of distance from the light to the vertex in world coordinates. By controlling the coefficients *k0, k1*, and *k2* the attenuation function can be controlled.

A more convenient way to control the distance attenuation is provided by GXInitLightDistAttn, as shown by Code 6-4.

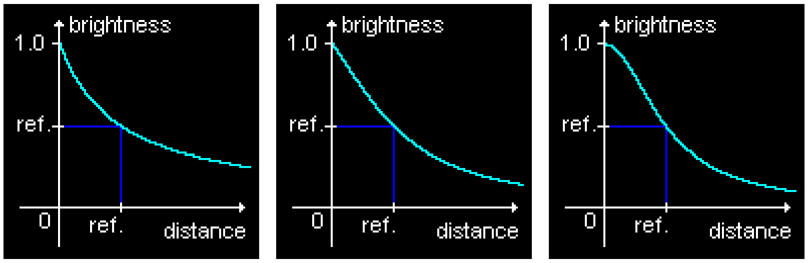
Code 6-4 GXInitLightDistAttn

```
GXInitLightDistAttn (
        GXLightObj  *lt_obj,
        f32          ref_distance,
        f32          ref_brightness,
        GXDistAttnFn dist_func );
```

In this function, you can specify the brightness on a defined reference point. The parameter *ref_distance* is distance between the light and the reference point. The parameter *ref_brightness* specifies the ratio of the brightness at the reference point. The value for *ref_distance* should be greater than 0 and *ref_brightness* should be within *0<ref_brightness<1*, otherwise the distance attenuation feature is turned off.

The parameter *dist_func* defines how brightness decreases as a function of distance. The following graphs show the curve shapes given by acceptable values for *dist_func*. The value `GX_DA_OFF` turns the distance attenuation feature off.

Figure 6-4 Distance Attenuation Functions



GX_DA_GENTLE GX_DA_MEDIUM GX_DA_STEEP

This function sets parameters only for distance attenuation. The parameters for angle attenuation should be set using `GXInitLightSpot`. You can also use `GXInitLightAttn`, but you have to be careful about order when calling these functions because `GXInitLightAttn` overwrites parameters for both angle and distance attenuation.

6.5 Channel Parameters

6.5.1 Channel Colors

A vertex may include up to two colors, each having up to two channels: Color (*R, G, B*) and Alpha (*A*). Taken together there are a total of four channels: *color0*, *color1*, *alpha0*, and *alpha1*.

Each channel has an associated ambient color or alpha and a material color or alpha. These colors can come from vertex colors or from special ambient and material registers. The register colors are set using the functions shown in Code 6-5.

Code 6-5 GXSetChanAmbColor

```
void GXSetChanAmbColor(
        GXChannelID   chan,
        GXColor       amb_color );
void GXSetChanMatColor(
        GXChannelID   chan,
        GXColor       mat_color );
```

Only the components of `GXColor` needed by the channel are set. Also, you may set both the color and alpha of a channel at the same time if this is more convenient.

6.5.2 Channel Control

Each channel is controlled using the function:

Code 6-6 GXSetChanCtrl

```
GXSetChanCtrl(
        GXChannelID   chan,
        GXBool        enable,
        GXColorSrc    amb_src,
        GXColorSrc    mat_src,
        GXLightID     light_mask,
        GXDiffuseFn   diff_fn,
        GXAttnFn      attn_fn );
```

If a lighting channel is disabled, *enable* `= GX_DISABLE`, the material color for that channel will be passed through unmodified to be rasterized. The *mat_src* parameter determines whether the material color comes from the vertex color or from the material register.

When a channel is enabled, the lighting equation is computed for each light enabled in the *light_mask*. The spot light enable, *attn_fn*, is defined as part of the channel, even though the angle attenuation parameters are part of the light description. Diffuse attenuation can be enabled using *diff_fn*. Normally, when *diff_fn* is enabled, the `GX_DF_CLAMP` value is used. Sometimes it is useful to disable diffuse attenuation in order to pass distance attenuation directly to the TEV.

6.5.3 Pre-lighting

Often, it is useful to pre-light an object in a modeling tool, such as 3D Studio MAX. The result of pre-lighting is usually captured in the vertex colors of the object. When the lighting channel is configured properly, you can combine hardware-computed local diffuse lighting with pre-lighting. One example, shown below, assumes the pre-lit color is `GX_VA_CLR0` per-vertex. The equation we want to implement is:

Equation 6-10 Pre-lighting

$$lit\_clr = pre\_lit\_clr * (amb\_scale + diff\_scale * other\_attn * diff\_lit\_clr)$$

$$amb\_scale + diff\_scale = 1.0$$

When no local diffuse light is shining on an object, the color is equal to the ambient pre-lit color which is (*pre_lit_clr*amb_scale*). When a light is shining on the object, the percentage of pre-lit color is increased until, where the light is the brightest, the full value of pre-lit color is used.

The following example sets up `GX_COLOR0` channel for pre-lighting. The vertex color0 will be the pre-lit color at full intensity. The ambient color is set to white and scaled so that with no lighting the vertex color will be 25% of the pre-lit color. The diffuse light color is scaled so that when fully lit, the vertex color will equal 100% of the pre-lit color.

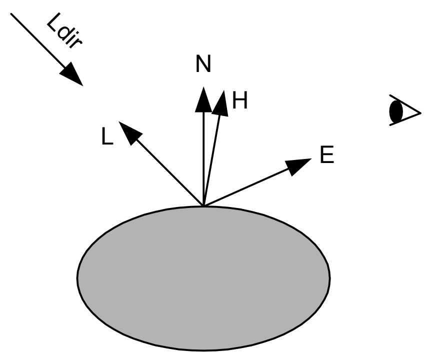
Code 6-7 Pre-lighting API

```
// init light position and direction and dist atten
GXInitLightColor(
 Lt_obj,
 ScaleColor(myLitColor, 0.75) ); // diffuse scale
GXInitLightSpot(
 Lt_Obj,
 30.0,
 GX_SP_COS2 );
GXLoadLightObjImm(
 Lt_obj,
 GX_LIGHT0 );
GXSetChanAmbColor(
 GX_COLOR0,
 ScaleColor(White, 0.25)); // ambient scale
GXSetChanCtrl(
 GX_COLOR0,
 GX_ENABLE,
 GX_SRC_REG,  // ambient color source
 GX_SRC_VTX,  // material color source (pre-lit color)
 GX_LIGHT0,
 GX_DF_CLAMP,
 GX_AF_SPOT );
```

6.6 Specular Lighting

The GP supports the computation of specular lighting. Specularity is actually a surface property that is commonly equated with “shininess.” Specular highlights result when a surface is angled such that it reflects the light from the light source toward the eye point. Specular lighting is implemented by creating an infinite specular light source and modifying the angle attenuation control appropriately. In the equations below, “H” refers to the half-angle between the vectors to the light and to the eye, as shown in Figure 6-5.

Figure 6-5 Specular Lighting Vectors



Equation 6-11 Specular Attenuation

$$Atten_i(j) = GX\_AF\_NONE\ ?\ 1 : \frac{Clamp0\left(a_{2_j} AAtt_i(j)^2 + a_{1_j} AAtt_i(j) + a_{0_j}\right)}{k_{2_j} d_i(j)^2 + k_{1_j} d_i(j) + k_{0_j}}$$

$$AAtt_i(j) = \vec{N} \cdot \vec{L} > 0\ ?\ Clamp0(\vec{N} \cdot H_i) : 0\ //\ GX\_AF\_SPEC$$

$$d_i(j) = \vec{N} \cdot \vec{L} > 0\ ?\ Clamp0(\vec{N} \cdot H_i) : 0\ //\ GX\_AF\_SPEC$$

One may specify a specular light in the `GXSetChanCtrl` function by setting the `GXAttnFn` parameter to `GX_AF_SPEC`. Since specular light sources are infinite, distance attenuation does not apply to them. You specify the specular light direction using `GXInitSpecularDir`. It will compute and store the half-angle and light direction. You may also specify the half-angle directly using the function `GXInitSpecularDirHA`. Setting a light this way overwrites the position and direction from a diffuse light source. You should not use `GXInitLightDir` or `GXInitLightPos` with a specular light source.

A specular light’s half-angle is stored in the “light direction” field of the light object, while the specular light direction is stored in the “light position” field of the light object. The specular light direction is first multiplied by 2^20 before being stored. This factor does not affect the specular computation, since the direction is normalized first, and the factor allows the same light object to be used as a diffuse, non-directional light source (when used with a different channel). The specular-computed light and the diffuse-computed light can only be combined in the TEV.

You can use the macro `GXInitLightShininess` to control the specular attenuation function:

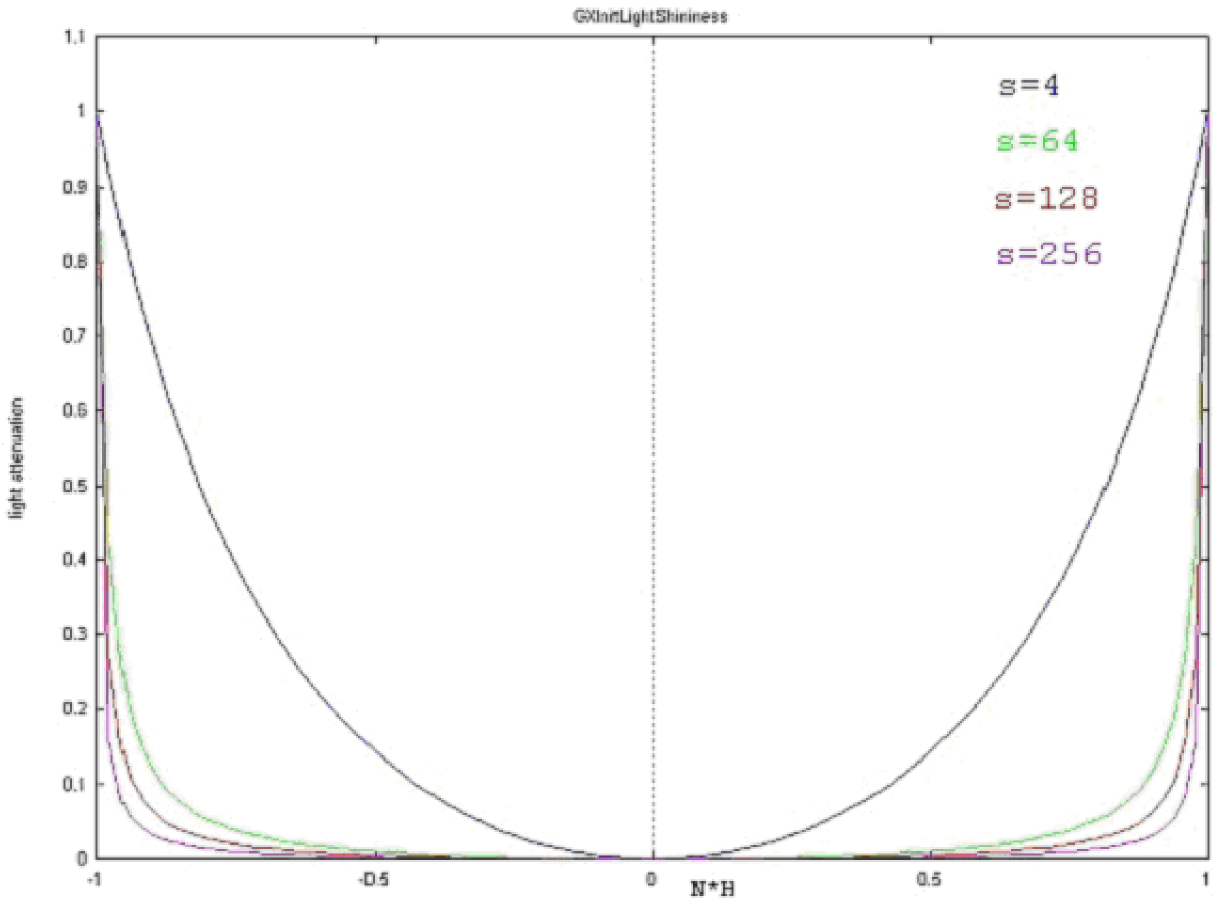
Code 6-8 GXInitLightShininess()

```
GXInitLightShininess(lt_obj, shininess);
```

This macro effectively controls how sharp the specular highlight appears on the lit surface. It sets both the distance and angle attenuation coefficients. Care should be taken when using this macro with other functions that set the attenuation coefficients, such as `GXInitLightAttn`, `GXInitLightAttnA`, `GXInitLightAttnK`, `GXInitLightDistAttn`, and `GXInitLightSpot`.

A plot of the shininess attenuation function is shown below for various values of shininess *s*. The plot shows that higher values of shininess result in a sharper fall-off in the attenuation function. Minimum attenuation occurs when $\hat{N} \cdot \hat{H}$ is +/- 1.0.

Figure 6-6 GXInitLightShininess Values



6.7 Vertex performance

"Table 1 - Vertex Performance" on page 51 lists the peak vertex data rate for various combinations of vertex data and lighting. Actual performance depends on a number of game-specific variables and should be determined empirically (see "6.8 Lighting Performance" on page 51 for an example). This table takes into account transform, lighting, and setup performance.

- **#T**: Number of (*s, t*) textures.
- **#PT**: Number of (*s, t, q*) projected textures.
- **#L**: Number of local diffuse or specular lights.
- **#BM**: Number of generated bump maps.
- **#C**: Host-supplied color.

Table 1 - Vertex Performance

Vertex Mode	Performance (Vertices per second)
1C Vertex	48.6 MV/s
1T Vertex	40.5 MV/s
[1 PT] [1 L] Vertex	30.5 MV/s
2T 1L Vertex	22.1 MV/s
3T 1L Vertex	17.4 MV/s
3PT 1L Vertex	17.4 MV/s
2T 2L Vertex	22.1 MV/s
2T 4L Vertex	14.3 MV/s
4PT 4L Vertex	14.3 MV/s
8T 4L Vertex	8.4 MV/s
8PT 4L Vertex	8.4 MV/s
2T 1BM 1L Vertex	12.8 MV/s
3T 2BM 1L Vertex	7.8 MV/s
3T 2BM 4L Vertex	6.9 MV/s

6.8 Lighting Performance

In hardware, the lighting pipeline computes three components per light. If a color channel is being computed, the components correspond to R/G/B. If an alpha channel is being computed, the components correspond to A/A/A. Each light costs four cycles (at 243 MHz) for all the lights that will contribute to a channel, plus one cycle. The discussion below applies only to local diffuse lighting (no texture coordinate generation or bump mapping). It is assumed that the vertex supplies only a color for each active channel.

For example, if channel `GX_COLOR0` uses two lights and channel `GX_ALPHA0` uses one light, then the performance is 2*4 + 1*4 + 1 = 13 cycles, or 243/13 million vertices/second (18.7 million vertices/second). If channel 0 (`GX_COLOR0` and `GX_ALPHA0`) uses three lights and channel 1 (`GX_COLOR1` and `GX_ALPHA1`) uses four lights, the total performance is 2*3*4 + 2*4*4 + 1 = 57 cycles, or 243/57 million vertices/second (4.3 million vertices/second).

You can use the performance counter function described in "14 Performance Metrics" on page 151 to determine the number of Graphics Processor clocks required per vertex, given the current lighting and texture coordinate generation settings. You can also use the online GX calculator, titled *Vertex Performance Calculator*, in the *Revolution Function Reference Manual* (HTML).

7 Texture Coordinate Generation

7.1 Specifying Texgens

The GP has many ways to generate texture coordinates. This section briefly describes the major aspects of the `GXSetTexCoordGen` function. For details on applications of texture coordinate generation, see the *Graphics Library (Advanced Rendering)* manual.

Code 7-1 GXSetTexCoordGen

```
GXSetTexCoordGen(
    GXTexCoordID  dst_coord,
    GXTexGenType  func,
    GXTexGenSrc   src_param,
    u32           mtx );
```

The texture coordinate generation function takes this general form:

Equation 7-1 Texture Coordinate Generation

$$dst\_coord = func(src\_param, mtx)$$

The input data described by the current vertex descriptor is transformed into a texture coordinate. The most common function of texture coordinate generation is to transform the *src_param* by a 2x4 or 3x4 texture matrix, *mtx*. In this case, *func* is set to either `GX_TG_MTX2x4` or `GX_TG_MTX3x4`.

Equation 7-2 Transforming *src_param* by 2x4 and 3x4 Matrices

$$\begin{bmatrix} s \\ t \end{bmatrix} = [GX\_TEXMTX*][InputCoord] \quad //GX\_TG\_MTX2x4$$

or

$$\begin{bmatrix} s \\ t \\ q \end{bmatrix} = [GX\_TEXMTX*][InputCoord] \quad //GX\_TG\_MTX3x4$$

The parameter *q* indicates the vertical distance from the light's focus. When conversion is done using a 3x4 texture matrix, the calculated *s* and *t* are divided by *q* before being passed to the next process. For details, see the *Graphics Library (Advanced Rendering)* manual.

Input coordinates, *src_param*, are one of:

Equation 7-3 Input Coordinates

$$\begin{aligned}
&(s_x \quad t_x \quad 1.0 \quad 1.0) \quad // GX_TG_TEX*\\
&(x \quad y \quad z \quad 1.0) \quad // GX_TG_POS\\
&(n_x \quad n_y \quad n_z \quad 1.0) \quad // GX_TG_NRM\\
&(b_x \quad b_y \quad b_z \quad 1.0) \quad // GX_TG_BINRM\\
&(t_x \quad t_y \quad t_z \quad 1.0) \quad // GX_TG_TANGENT
\end{aligned}$$

Input coordinates for `GX_TG_MTX2x4` and `GX_TG_MTX3x4` functions are the ***untransformed*** vertex data.

At a minimum, the GP always transforms input texture coordinates by a matrix. By default, the available texture matrices are described by the enumeration `GXTexMtx`. To pass input texture coordinates unchanged into output coordinates, use the `GX_IDENTITY` matrix. You must always generate a consecutive number of texture coordinates starting at `GX_TEXCOORD0`.

In addition to transforming texture coordinates, you can also transform positions and normals to create output texture coordinates. Transforming an input texture coordinate using a 2x4 matrix is useful for translation and rotation effects. A 3x4 matrix is useful for projecting textures and reflection mapping.

You can change the order of output texture coordinates from the input coordinate order using the `GXSetTexCoordGen` function. In addition, you can use a single input parameter to generate multiple texture coordinates. You can also ignore certain input parameters. This allows you to turn various layers of texture on and off without needing to build a new display list for each version (at the cost of sending vertex data you don't use).

The bump mapping function (*func* = `GX_TG_BUMP*`) of texture coordinate generation supports the embossing style of bump mapping. This style of bump mapping is useful when the surface geometry of an object is being animated. Texture coordinate generation carried out with the bump mapping function also affects the vertex lighting hardware. A maximum of three `GX_TG_BUMP*` texture coordinates are supported simultaneously. See the *Graphics Library (Advanced Rendering)* manual for more details on embossed bump mapping.

Texture coordinates can also be generated from the red and green components of a particular lighting channel (*func* = `GX_TG_SRTG`). These can be used to create "cartoon" lighting functions, with sharp transitions between arbitrary colors. Normally, the red component represents the intensity function of a single local diffuse light. The green channel is programmed as a material control. The red channel is mapped to the *s*-coordinate. The green channel is mapped to the *t*-coordinate. The *s*-coordinate (the diffuse light intensity) is mapped by an arbitrary 1D texture lookup into a color. The *t*-coordinate can be used to select which 1D table to use from an array of tables (a 2D texture). A maximum of two texture coordinates may be generated using `GX_TG_SRTG`. The first one must be used with color channel 0, while the second can only be used with color channel 1. See the *Graphics Library (Advanced Rendering)* manual for more details on cartoon lighting.

Generated texture coordinates must be sorted by function. That is, texture coordinates generated by simple transforms should occur first, followed by bump map coordinate generation, and finally the generation of any texture coordinates based on lighting results.

Table 7-1 Texture Coordinate Generation Order

Required Order	Texture Coordinate Generation Function
First	`GX_TG_MTX2x4`, `GX_TG_MTX3x4`
Next	`GX_TG_BUMP0-7`
Last	`GX_TG_SRTG` (up to 2 texture coordinates can be generated)

In addition to setting the texture-coordinate generation functions, one must also specify how many coordinates are being generated. This is performed by the function in Code 7-2.

Code 7-2 GXSetNumTexGens

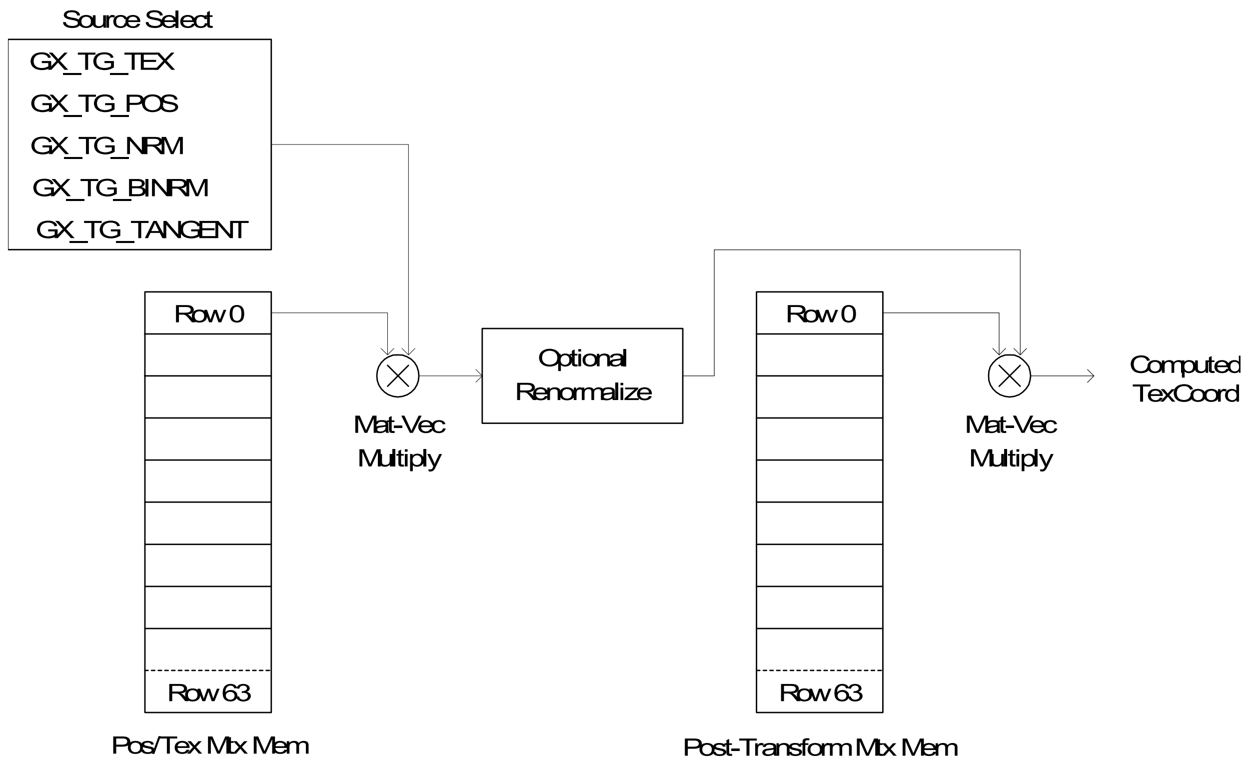
```
GXSetNumTexGens( u8 nTexGens );
```

The default number of texgens (set in `GXInit`) is one. If no vertex lighting is enabled (`GXSetNumChans(0)`), then at least one texture coordinate must be generated. If at least one channel is being lit (`GXSetNumChans(1)`), then the number of texgens can be set to zero.

7.2 Renormalization and “post-transform” Matrices Added for Texgens

It is also possible to use renormalization and post-transformation in the texture coordinate generation computation path.

Figure 7-1 Texgen Computation Path



It is possible to add an optional renormalization step followed by a second “post-transform” matrix-vector multiplication.

There are various ways to make use of these extra features. You can choose to provide twice as many matrices for position transformation and use the extra memory for texgens (by selecting the identity matrix for the first transformation). You can also use this feature to provide more efficient projected textures. In this case, you use the position multiplied by the position matrix, then multiplied by a reprojection/rescale matrix. This feature may also be used to make environment mapping easier. You use the normal multiplied by a regular normal matrix (but stored in the Pos/Tex Mtx memory), then renormalized and multiplied by a post-transform matrix that rescales the normal into texture space.

This feature is accessed by using `GXSetTexCoordGen2`, as shown in Code 7-3.

Code 7-3 GXSetTexCoordGen2

```
GXSetTexCoordGen2(
    GXTexCoordID  dst_coord,
    GXTexGenType  func,
    GXTexGenSrc   src_param,
    u32           mtx,
    GXBool        renormalize,
    u32           pt_mtx );
```

7.3 Other Texture Coordinate Generation Issues

In the GP, texture coordinates must be scaled to the size of the texture to which they are being applied. Normally, this is taken care of automatically by GX. However, you can also control the texture coordinate scaling manually by calling the function in Code 7-4.

Code 7-4 GXSetTexCoordScaleManually

```
void GXSetTexCoordScaleManually(GXTexCoordID coord, GXBool enable, u16 ss, u16 ts)
```

One application of this function is discussed in "10 Indirect Texture Mapping" on page 107.

The GP also has a feature that allows cylindrical geometry to be wrapped with a texture without having to specify different texture coordinates for the seam vertices. This is known as cylindrical texture wrapping, and it can be enabled with the function in Code 7-5.

Code 7-5 GXSetTexCoordCylWrap

```
void GXSetTexCoordCylWrap(GXTexCoordID coord, GXBool s_enable, GXBool t_enable)
```

In effect, this function computes the spread between each vertex's texture coordinate and the minimum coordinate, then subtracts 1 from any coordinate where the spread exceeds 0.5.

7.4 Texture Coordinate Generation Performance

The functions that are based on lighting, `GX_TG_BUMP*` and `GX_TG_SRTG`, exhibit performance similar to lights (see "6 Vertex Lighting" on page 41 for vertex lighting performance details). Each `GX_TG_SRTG` is equivalent to a light, or four clocks for each coordinate generated plus one clock. Bump mapping is equivalent to two lights, or eight clocks for each coordinate generated plus one clock.

The `GX_MTX_2x4` and `GX_MTX_3x4` functions take three clocks per component. For the special case of a single (*s*, *t*) texture coordinate transform, only two clocks are required.

In general, the actual vertex performance is a function of lighting, texture coordinate generation, input data, and output data parameters. To compute the performance for a particular configuration, use the *Vertex Performance Calculator* in the Performance section of the GX API pages in the *Revolution Function Reference Manual* (HTML).

8 Texture Mapping

Wii has powerful texture mapping features, including single-cycle mipmapping, compressed color texture and color index textures, anisotropic texture filtering, multitexture support (in multiple cycles), indirect textures, and Z textures. The graphics processor (GP) textures four pixels per clock at a 243 MHz clock speed, for a peak mipmapped texture rate of 972 megapixels/second.

A highly-flexible 1MB Texture Memory (TMEM) can be configured as multiple texture caches, color index lookup tables (TLUTs), or it can be preloaded with textures. Texture caches can automatically prefetch texels from main memory as needed. The TMEM is embedded high-speed 1TSRAM and is physically separate from the embedded frame buffer (EFB). Textures are prefetched in parallel with rendering.

The GP can be configured to Z-buffer before texturing, eliminating texture fetches for pixels that are not visible.

The texture hardware performs perspective correction, Level of Detail (LOD), and coordinate operations like clamping, repeating, or mirroring at the rate of 4 pixels per clock per image. To support multitexturing, the state describing up to eight active textures is stored in the GP. Multitexturing is accomplished by processing a quad of pixels (2x2) through the pipeline over multiple cycles.

`GXInit` provides a default configuration of the texture pipeline that abstracts away many of the more complex texture features. (You can override this configuration easily, as described later in this chapter and in "9 Texture Environment (TEV)" on page 91.) Code 8-1 shows a simple program that loads and uses a texture.

8.1 Example: Drawing a Textured Triangle

Code 8-1 Simple Texture Example

```
/*---------------------------------------------------------------------------*
  Project:  Dolphin/Revolution gx demo
  File:     smp-texexample.c

  Copyright 1998 - 2006 Nintendo.  All rights reserved.

  These coded instructions, statements, and computer programs contain
  proprietary information of Nintendo of America Inc. and/or Nintendo
  Company Ltd., and are protected by Federal copyright law.  They may
  not be disclosed to third parties or copied or duplicated in any form,
  in whole or in part, without the prior written consent of Nintendo.
*---------------------------------------------------------------------------*/

#include <demo.h>

#define BALL64_TEX_ID          8

/*---------------------------------------------------------------------------*
  Model Data
 *---------------------------------------------------------------------------*/

static s8 Vert_s8[] ATTRIBUTE_ALIGN(32) =
{
    -100,  100, 0,  // 0
     100,  100, 0,  // 1
    -100, -100, 0   // 2
};

static u32 Colors_u32[] ATTRIBUTE_ALIGN(32) =
{
//    r g b a
    0xff0000ff, // 0
    0x00ff00ff, // 1
```

```
    0x0000ffff  // 2
};

//  Array of texture coordinates
static u8 TexCoords_u8[] ATTRIBUTE_ALIGN(32) =
{
    0x00, 0x00, // 0
//    s     t        fixed-point format is unsigned 8.0
    0x01, 0x00, // 1
    0x00, 0x01  // 2
};

/*---------------------------------------------------------------------------*
   Forward references
 *---------------------------------------------------------------------------*/

static void CameraInit( Mtx v );

/*---------------------------------------------------------------------------*
   Application main loop
 *---------------------------------------------------------------------------*/

void main ( void )
{
    PADStatus    pad[4];   // Controller state
    GXTexObj     texObj;   // texture object
    Mtx          v;        // view matrix
    u8           i;        // loop variable
    TPLPalettePtrtpl = 0; // texture palette

    pad[0].button = 0;

    DEMOInit(NULL);    // Init os, pad, gx, vi

    CameraInit(v);
    GXLoadPosMtxImm(v, GX_PNMTX0);

    GXSetNumChans(1);  // Enable light channel; by default = vertex color

    GXClearVtxDesc();
    GXSetVtxDesc(GX_VA_POS,  GX_INDEX8);
    GXSetVtxDesc(GX_VA_CLR0, GX_INDEX8);
    // Add an indexed texture coordinate to the vertex description
    GXSetVtxDesc(GX_VA_TEX0, GX_INDEX8);

    GXSetArray(GX_VA_POS,  Vert_s8, 3*sizeof(s8));
    GXSetArray(GX_VA_CLR0, Colors_u32, 1*sizeof(u32));
    GXSetArray(GX_VA_TEX0, TexCoords_u8, 2*sizeof(u8));

    GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_POS,  GX_POS_XYZ,  GX_S8,    0);
    GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_CLR0, GX_CLR_RGBA, GX_RGBA8, 0);
    //  Describe the texture coordinate format
    //  fixed-point format is unsigned 8.0
    GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_TEX0, GX_TEX_ST,   GX_U8,    0);

    //  Load the texture palette

    (file loading omitted)

    TPLBind(tpl);
    //  Initialize a texture object to contain the correct texture
    TPLGetGXTexObjFromPalette(tpl, &texObj, BALL64_TEX_ID);
    //  Load the texture object; tex0 is used in stage 0
    GXLoadTexObj(&texObj, GX_TEXMAP0);
```

```
    //  Set the Texture Environment (Tev) Mode for stage 0
    //  GXInit sets default of 1 TexCoordGen
    //  Default TexCoordGen is texcoord(n) from tex(n) with 2x4 identity mtx
    //  Default number of tev stages is 1
    //  Default stage0 uses texcoord0, texmap0, color0a0
    //  Only need to change the tevop
    GXSetTevOp(GX_TEVSTAGE0, GX_DECAL);

    OSReport("\n\n********************************\n");
    OSReport("to quit:\n");
    OSReport("    hit the start button\n");
    OSReport("********************************\n");

    while(!(pad[0].button & PAD_BUTTON_MENU))
    {
        DEMOBeforeRender();
        // Draw a triangle
        GXBegin(GX_TRIANGLES, GX_VTXFMT0, 3);
        for (i = 0; i < 3; i++)
        {
            GXPosition1x8(i);
            GXColor1x8(i);
            // Add texture coordinate
            GXTexCoord1x8(i);
        }
        GXEnd();
        DEMODoneRender();
        PADRead(pad);
    }
    OSHalt("End of demo");
}

/*---------------------------------------------------------------------------*
   Functions
 *---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*
   Name:         CameraInit

   Description:  Initialize the projection matrix and load into hardware.
                 Initialize the view matrix

   Arguments:    v   view matrix

   Returns:      none
 *---------------------------------------------------------------------------*/

static void CameraInit ( Mtx v )
{
    Mtx44 p;
    Vec   camPt = {0.0F, 0.0F, 800.0F};
    Vec   at    = {0.0F, 0.0F, -100.0F};
    Vec   up    = {0.0F, 1.0F, 0.0F};

    MTXFrustum(p, 240.0F,-240.0F,-320.0F, 320.0F, 500, 2000);
    GXSetProjection(p, GX_PERSPECTIVE);
    MTXLookAt(v, &camPt, &up, &at);
}
```

The GX API requires the following basic steps to use texturing:

1. Load textures into main memory.
2. Allocate a texture object (`GXTexObj`) structure for each texture.
3. Initialize the texture object (`GXInitTexObj`) to describe the texture.
4. Load a texture object (`GXLoadTexObj`) into the GP to activate the texture.

For color index textures, you should perform the following additional steps:

5. Load TLUTs into main memory.
6. Allocate a TLUT object (`GXTlutObj`) structure for each TLUT.
7. Initialize the TLUT object (`GXInitTlutObj`) to describe the TLUT.
8. Load the TLUT (`GXLoadTlut`) into one of the named TLUT regions of texture memory (`GXTluts`).
9. Associate a TLUT name with a texture when initializing the color-index texture object.

These steps are explained in more detail in the sections that follow.

8.2 Loading a Texture into Main Memory

The first step in using a texture is to load it into main memory from the optical disc. The software released with the Revolution SDK includes a program called `TexConv.exe` for converting common images files to a “Texture Palette” (TPL, extension `.tpl`) format. TPL files can be stored on disc and loaded using the Texture Palette library. This library will load a TPL file and return a texture object that can be used by the GX API.

8.3 Describing a Texture Object

The `GX API` uses a `GXTexObj` structure to describe the various parameters associated with a texture, and it is the user’s responsibility to allocate the memory for this structure. Parameters include:

- A pointer (aligned to 32B) to the texture image data.
- The texture’s format.
- The width and height of the texture.
- The texture filter modes.
- The texture wrapping controls.

The user initializes or changes a GXTexObj using GXInitTexObj (for non-color index textures) or GXInitTexObjCI (for color-index textures).

Code 8-2 Initializing or Changing a Texture Object

```
 GXInitTexObj(
    GXTexObj*      obj,
    void*          image_ptr,
    u16            width,
    u16            height,
    GXTexFormats   format,
    GXTexWrapModes wrap_s,
    GXTexWrapModes wrap_t,
    GXBool         mipmap );

GXInitTexObjCI(
    GXTexObj*      obj,
    void*          image_ptr,
    u16            width,
    u16            height,
    GXCITexFmt     format,
    GXTexWrapModes wrap_s,
    GXTexWrapModes wrap_t,
    GXBool         mipmap,
    u32            tlut_name );
```

Additional mipmap controls (which are set to default values by the GXInitTexObj* functions listed above) can be set using GXInitTexObjLOD. The GXInitTexObj* functions pre-compile their parameters into hardware register state settings for optimum performance. The GXTexObj structure is used to store these pre-compiled values and so are not directly readable by the application. Instead, you should use the GXGet* functions to read the contents of a GXTexObj.

8.3.1 Texel Formats

The parameter *format* in `GXInitTexObj` and `GXInitTexObjCI` sets the texture format. The table below lists the possible formats. `GXInitTexObjCI` should only set color-index formats and `GXInitTexObj` should only set non-color-index texture formats.

Table 8-1 Texel Formats

Texture Format Name	Description	Mipmap Filter Modes
GX_TF_I4	Intensity 4 bit	all
GX_TF_I8	Intensity 8 bit	all
GX_TF_IA4	Intensity + Alpha 8 bit (4+4)	all
GX_TF_IA8	Intensity + Alpha 16 bit (8+8)	all
GX_TF_C4	Color Index 4 bit	LIN_MIP_NEAR
GX_TF_C8	Color Index 8 bit	LIN_MIP_NEAR
GX_TF_C14X2	Color Index 16 bit (14b index)	LIN_MIP_NEAR
GX_TF_RGB565	RGB 16 bit (565)	all
GX_TF_RGB5A3	When MSB = 1, RGB555 format (opaque) When MSB = 0, RGBA4443 format (transparent)	all
GX_TF_RGBA8	RGBA 32 bit (8888)	all
GX_TF_CMPR	Compressed 4 bits/texel, ~RGB8A1	all

The texture filter always processes 4-channel colors, RGBA, where each channel is 8 bits wide. In general, when the size of the texel component is less than 8 bits, the most significant bits (MSBs) of the texel are copied into the least significant bits (LSBs) of the color channel:

Code 8-3 Texture Component Promotion to 8 bits

```
Input Texel = 0xa5a5 (RGB565 format)
                       Hex  Binary
 Input Channel Red  = 0x14, 10100
Filter Channel Red  = 0xa5, 10100_101

 Input Channel Grn  = 0x2d, 101101
Filter Channel Grn  = 0xb6, 101101_10

 Input Channel Blu  = 0x05, 00101
Filter Channel Blu  = 0x29, 00101_001
```

This method guarantees that the entire color range from 0 to 255 is utilized. For the intensity formats, the intensity component is copied to all four of the RGBA channels. For the intensity/alpha texels, the intensity value of the texel is copied into only the RGB channels. For the RGB texture formats, the alpha channel is set to opaque (A = 0xff).

RGB5A3 is a 16-bit format that uses the MSB of each texel to indicate if the texel is opaque (`MSB == 1`) or transparent (`MSB == 0`). When the texel is opaque, the remaining 15 bits are assumed to be in a 5/5/5 RGB format. When the texel is transparent, the format is assumed to be 4/4/4/3 RGBA. The 8-bit alpha channel is formed as described above; the three bits of alpha are replicated starting at the MSBs until an 8-bit field is created. This format allows eight equal levels of transparency from fully transparent (`Alpha = 0x00`) to fully opaque (`Alpha = 0xff`). This format has two ways to represent opaque: when the MSB is 1 or when the MSB is 0 and the three bits of alpha are all 1's.

Compressed textures (`GX_TF_CMPR`) are stored as such in TMEM. Decompression occurs after texture lookup and before filtering. This results in storage savings in TMEM and main memory, and lower main memory to TMEM bandwidth requirements.

8.3.2 Texture Lookup Table (TLUT) Formats

Table 8-2 lists the possible TLUT formats. There is no support for 24-bit or 32-bit TLUT formats.

Table 8-2 TLUT Formats

TLUT Format Name	Description
GX_TL_IA8	Intensity + Alpha 16-bit (I8 + A8)
GX_TL_RGB565	RGB 16-bit (R5 + G6 + B5)
GX_TL_RGB5A3	When MSB = 1, RGB555 format (opaque) When MSB = 0, RGBA4443 format (transparent)

The color index lookup occurs *before* filtering. The color that is looked up from the TLUT is converted into an 8-bit per-component (RGBA) format for filtering in the same manner described in the previous section.

8.3.3 Texture Image Formats

Textures are stored in main memory as a row-column matrix of tiles with the following attributes:

- Each tile is a small sub-rectangle from the texture image.
- Each tile is a 4x4, 4x8, or 8x8 texel rectangle.
- Each tile is 32B, corresponding to the texture cache line size.
- Textures must be aligned to 32B in main memory and be a multiple of 32B in size.
- Texels are packed differently within a 32B tile depending on their type.

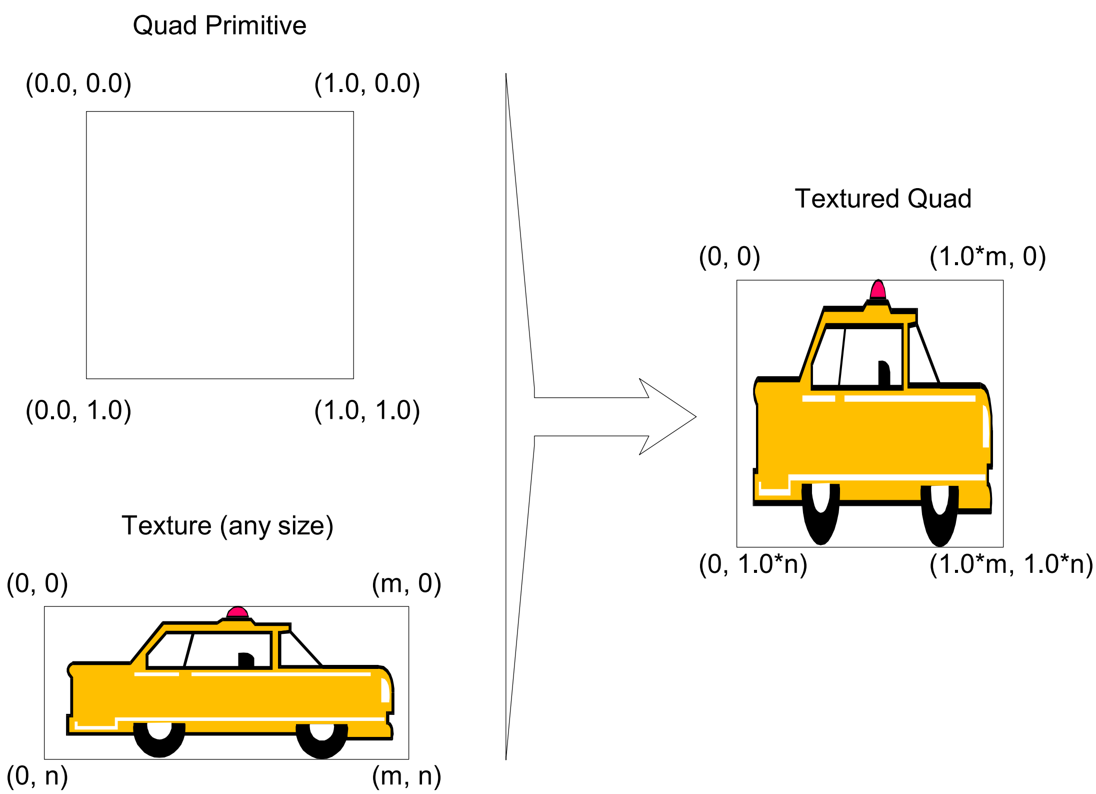
The level of detail maps for a mipmap are stored together one after another in main memory. The mipmap image data is referenced by the base pointer of LOD 0. The size of a mipmap refers to the size of the LOD 0 map. Mipmaps must be a power of 2 texels in width and height but not necessarily square. All the other LOD addresses can be calculated by the hardware from size and format information, and from the LOD 0 base pointer.

See Appendix D for texture formatting details. The texture conversion tool, `TexConv.exe`, can convert from common image formats to Wii texture formats.

8.3.4 Texture Coordinate Space

Input texture coordinates (those supplied by the application to the GP) are map-relative (normalized) coordinates. The following diagram shows how to map a single texture image onto a quad:

Figure 8-1 Map-relative Texture Coordinates



The GP will scale the input map-relative coordinates into *texel coordinate space* by multiplying the *s*-coordinate by the texture width and by multiplying the *t*-coordinate by the texture height. Each texture coordinate is associated with a texture map using the `GXSetTevOrder` function described in "9 Texture Environment (TEV)" on page 91.

The texel coordinate space ranges between +/-64K-texels. The largest texture image size is 1K by 1K-texels. The maximum texel coordinate extent across a primitive is 128K texels regardless of the image size. This implies a maximum number of repeats that can occur across a single primitive that depends on the texture size. For example, a 1K-texel image can repeat a maximum of 128 times across a single primitive.

The parameters *wrap_s* and *wrap_t* of the `GXInitTexObj` and `GXInitTexObjCI` functions control the texture coordinate operation. Texture coordinates can be operated on independently in one of three ways: either clamped (`GX_CLAMP`), repeated (`GX_REPEAT`), or mirrored (`GX_MIRROR`).

When clamping (`GX_CLAMP`), the texture coordinate is clamped within the bounds of the image. If the *s*-coordinate is negative, column 0 of the image is used. If the *s*-coordinate exceeds the width of the image, then the last column of the image is used. This clamping operation also occurs independently for the *t*-coordinate using the height of the image. When mipmapping, the width and height of the image should be powers of two, but the image does not have to be square.

Texture borders are sometimes used to piece together larger textures from smaller textures (tiled). The border area contains texels from the adjacent texture(s) so that filtering will use the correct information. The Wii system has no support for texture borders. For planar (non-mipmapped) images, the border can be included in the image itself because the GP supports arbitrary image width and height. Mipmapped images do not support borders, so they cannot be tiled without visible seams (unless borders are carefully designed into the textures themselves).

When repeating (`GX_REPEAT`), the (*s, t*) coordinates are modulo'd by the width/height of the image. Repeating is only valid for power of 2 image sizes. You can repeat textures to replicate a small texture over a large surface.

When mirroring (`GX_MIRROR`), the (*s, t*) coordinates are modulo'd by the width/height of the image, similar to `GX_REPEAT`. In addition, the coordinates are 1's complemented on every other wrap. Mirroring a texture is useful when an image is symmetrical about the (*s, t*) axis, like a tree texture. Mirroring is also useful for eliminating the seams that naturally occur when `GX_REPEAT`-ing a small texture.

The figures below illustrate the coordinate operations for a texture that is four texels wide for both linear and nearest filters. When using linear filtering, a ½ texel offset is subtracted to ensure proper spatial alignment of mipmap levels.

Figure 8-2 Linear Filter—Clamp, Repeat, Mirror

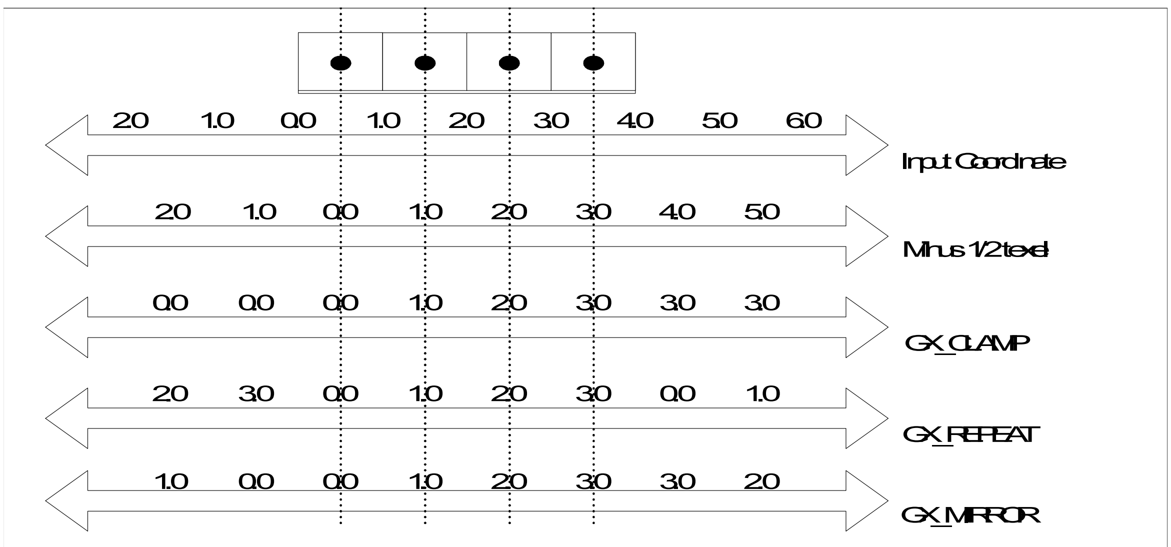
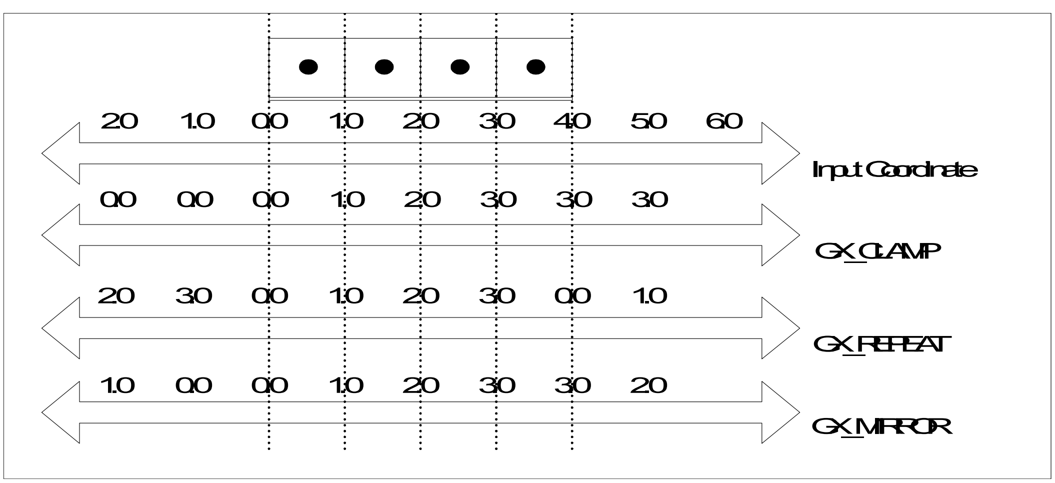


Figure 8-3 Nearest Filter—Clamp, Repeat, Mirror



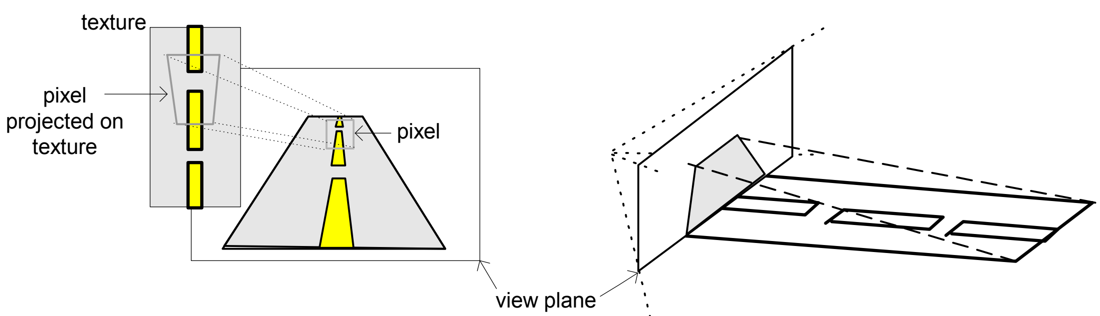
8.3.5 Filter Modes and LOD Controls

Filter modes and LOD controls are set to default values based on the state of the *mipmap* flag in the `GXInitTexObj` and `GXInitTexObjCI` functions. These default values can be overridden by a subsequent call to `GXInitTexObjLOD`. The parameters described in this section are set using the `GXInitTexObjLOD` function.

Code 8-4 GXInitTexObjLOD

```
GXInitTexObjLOD(
    GXTexObj*     obj,
    GXTexFilters  min_filt,
    GXTexFilters  mag_filt,
    f32           min_lod,
    f32           max_lod,
    f32           lod_bias,
    GXBool        bias_clamp,
    GXBool        do_edge_lod,
    GXAnisotropy  max_aniso );
```

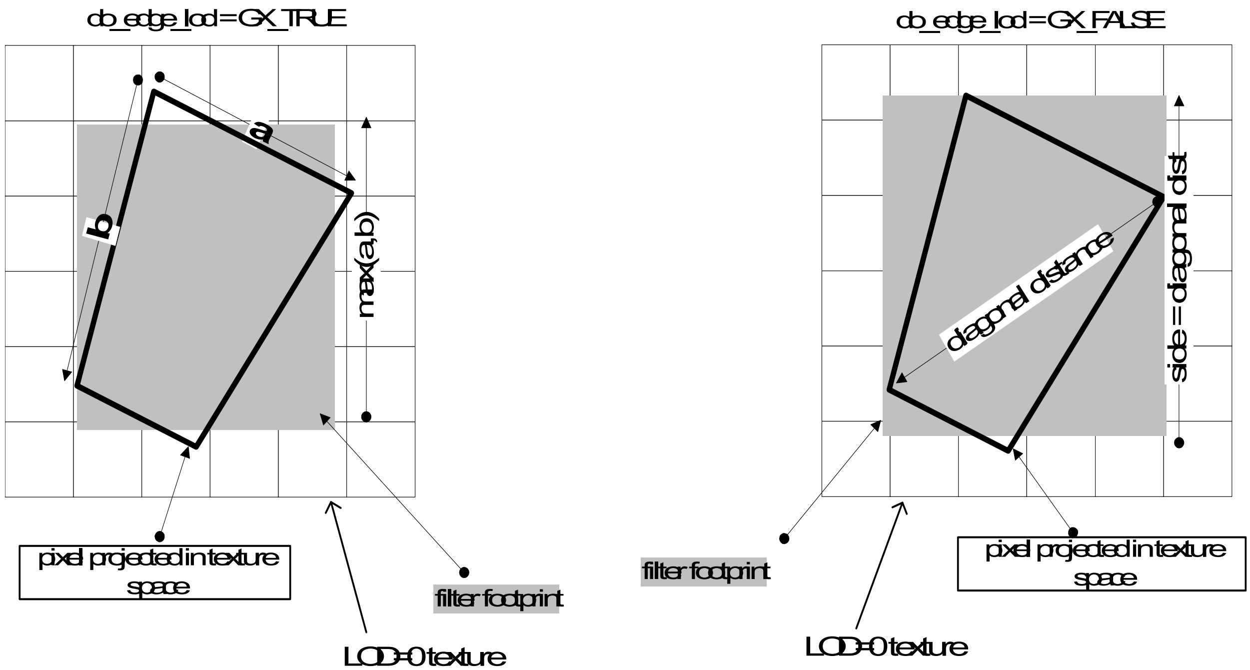
Figure 8-4 Pixel Projected in Texture Space Example



In the example above, the road texture is projected onto the view plane such that a square pixel on the screen maps to an elongated quadrilateral in texture space. The following discussion will reference this image of the pixel projected in texture space.

The level-of-detail calculation computes the effective texel-to-pixel ratio for a quad (2x2) of pixels. A log base 2 function converts the ratio into a corresponding mipmap level and a fraction, called *LOD*. Negative LOD indicates magnification, while positive LOD indicates minification. There are two methods of computing LOD. By setting *do_edge_lod* to `GX_TRUE`, LOD is computed using the distance in *(s, t)* between *adjacent* pixels in the quad. If you set *do_edge_lod* to `GX_FALSE`, LOD is computed using the distance in *(s, t)* between *diagonal* pixels in the quad.

Figure 8-5 LOD Calculation

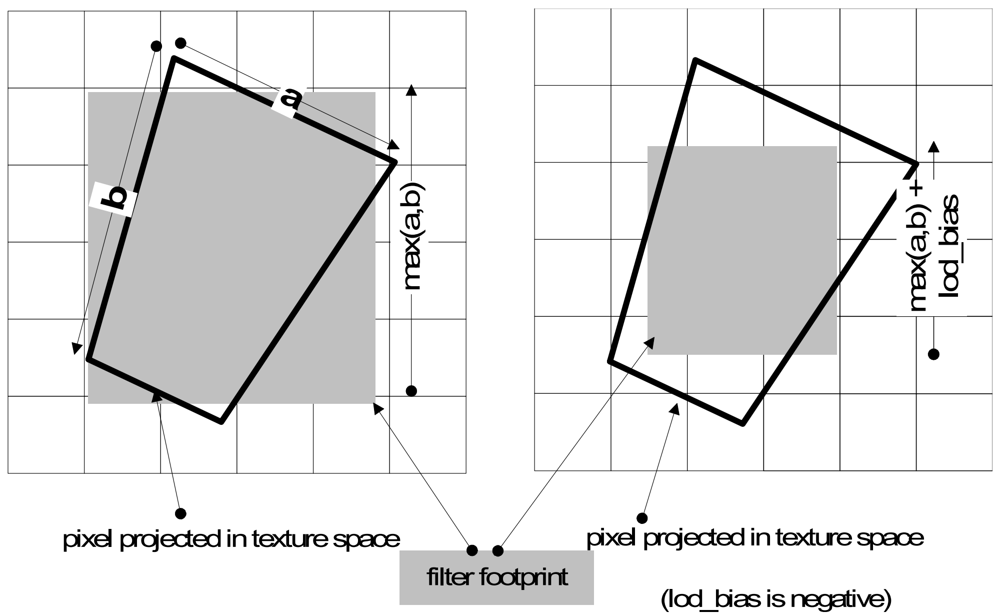


In other words, the LOD calculation assumes that a pixel projects to a square in texture space. This is only true if the polygon is roughly facing the viewer. The square filter pattern used by normal mipmapping is called an *isotropic* (uniform shape) filter. When the polygon to be rendered is oblique to the viewer, the pixel projected into texture space is distorted into a quadrilateral. In this case, the LOD calculation makes the square encompass the quadrilateral, resulting in excessive blurring in one of the coordinate directions.

There are two methods available in the GP for improving the overblurring behavior of the LOD computation*: LOD biasing* and *anisotropic filtering*.

The computed LOD value can be adjusted using the *lod_bias* parameter of the `GXInitTexObjLOD` function. *Lod_bias* can be used to prevent texture from becoming too blurry due to the conservative nature of the LOD calculation. *Lod_bias* must be in the range –4.0 to +3.99. The result of LOD + *lod_bias* is clamped between the *min_lod* and *max_lod* parameters. The *min_lod* and *max_lod* parameters define the usable region of the texture pyramid and can range from 0.0 to 10.0. The following figure shows how a negative *lod_bias* can be added to the computed LOD to effectively shrink the filter footprint in texture space.

Figure 8-6 LOD Bias

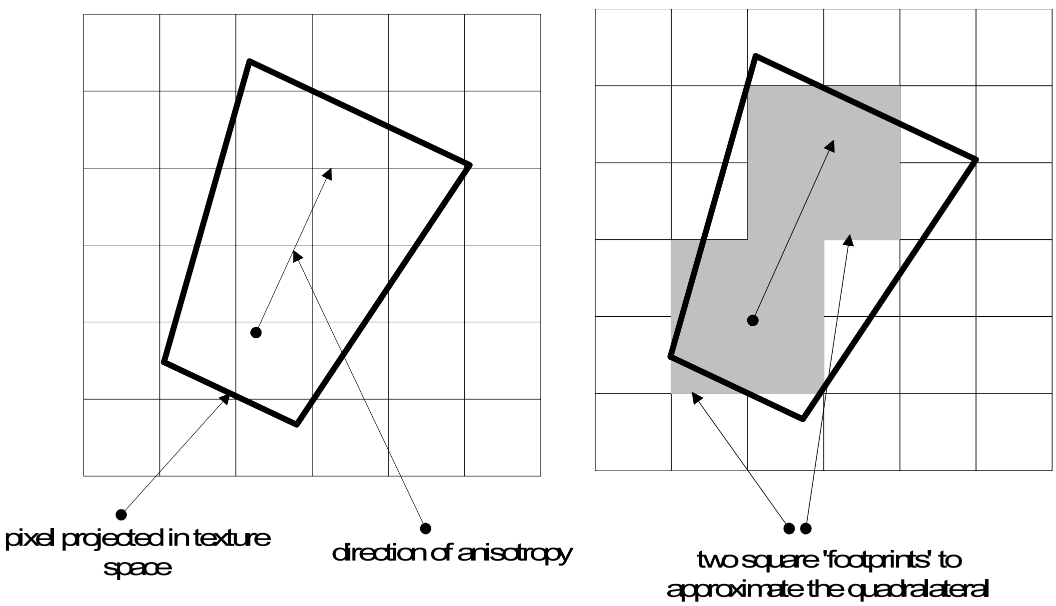


LOD bias will sharpen the texture when the polygon is oblique (the desired effect) but also when it is not. To alleviate this problem, the *bias_clamp* parameter can be used to lessen the effect of *lod_bias* when the polygon is more perpendicular to the view direction. When *bias_clamp* is enabled, the biased LOD will be clamped to the minimum extent of the pixel projected in texture space.

Multiple square trilinear texture filter “footprints” can be iterated to approximate the shape of the quadrilateral. This type of filter produces a sharper pixel and is said to be *anisotropic* (non-uniform shape). The maximum number of “footprints” allowed is programmable using the *max_aniso* parameter. Setting *max_aniso* to `GX_ANISO_1` allows only one square footprint and is the standard isotropic mipmap filter. Setting max_aniso to `GX_ANISO_2` and `GX_ANISO_4` allows a maximum of two or four footprints per pixel, respectively. The actual number of footprints used for each pixel is determined by the anisotropy of the pixel as computed by the hardware. Trilinear filtering should be enabled (*min_filt* = `GX_LIN_MIP_LIN`) when using `GX_ANISO_2` or `GX_ANISO_4`. Also, edge LOD must be enabled in order for anisotropic filtering to take place.

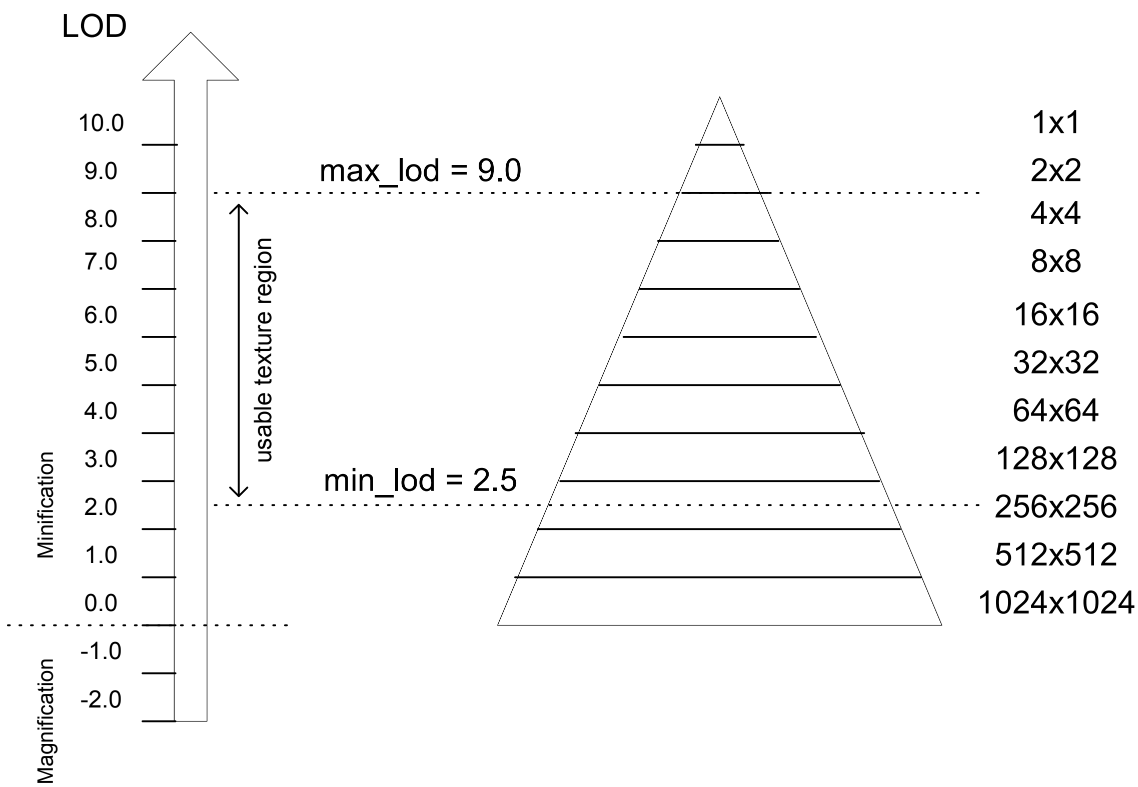
Multiple iterations of the texture filter require multiple internal cycles of the texture hardware. For example, if `GX_ANISO_4` is enabled, and the polygon being rendered requires the maximum four filter steps, the peak fill rate will be divided by a factor of four. Anisotropic filtering does not lower the number of available TEV stages (see "9 Texture Environment (TEV)" on page 91).

Figure 8-7 Anisotropic Filtering



The maximum image size is 1K x 1K-texels, so the largest mipmap pyramid is 11 levels of detail. LOD 0 always refers to the highest resolution LOD, regardless of the image size. In the case of the 1K x 1K-texel mipmap, LOD 10 is the coarsest resolution (1 x 1 texel).

Figure 8-8 Mipmap Pyramid for the Largest Texture Size

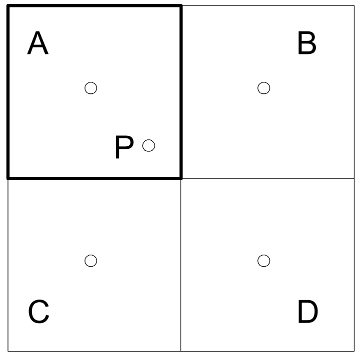


When LOD is negative, the LOD is said to be in the *magnification* region. In other words, the pixel projected into texture space covers only a fraction of a texel. This causes the LOD 0 texture to be magnified on the display device.

When the texture is in the magnification region, you can choose between `GX_NEAR` and `GX_LINEAR` filter modes using the *mag_filt* parameter.

Within an image, the `GX_NEAR` filter mode indicates that the closest texel to the pixel's *s*- and *t*-coordinates is chosen, as shown in the figure below:

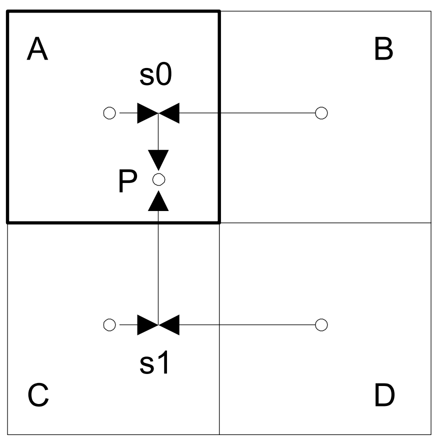
Figure 8-9 GX_NEAR



Pixel Color P = A

The `GX_LINEAR` filter mode indicates that the nearest four texels to the pixel's (*s*, *t*) coordinates should be bilinearly interpolated using the (*s*, *t*) fractional bits, as shown in Figure 8-10.

Figure 8-10 GX_LINEAR



s0 = A + (B-A)*s_fraction

s1 = C + (D-C)*s_fraction

Pixel Color P = s0 + (s1-s0)*t_fraction

When LOD is positive, the LOD is said to be in the *minification* region. In other words, the pixel projected into texture space covers more than one texel, and to achieve a 1:1 texel to pixel ratio these texels must be filtered. The texture will appear smaller on the display device.

As illustrated in the table below, when the texture is in the minification region, you can choose between several filter modes using the *min_filt* parameter.

Table 8-3 Mipmap Minimum Filter Modes

Name	Within LODn, LODn+1 (based on s, t fraction)	Between LODn, LODn+1 (based on LOD fraction)
GX_NEAR_MIP_NEAR	Use texel nearest to sample point.	Use nearest LOD.
GX_NEAR_MIP_LIN	Use texel nearest to sample point.	Interpolate between two closest LODs.
GX_LIN_MIP_NEAR	Bilinearly interpolate four texels surrounding sample point.	Use nearest LOD.
GX_LIN_MIP_LIN	Bilinearly interpolate four texels surrounding sample point.	Interpolate between two closest LODs

Color-index textures *cannot* use the `GX_NEAR_MIP_LIN` or `GX_LIN_MIP_LIN` filter modes. All other texture types, including compressed texture, can use any *min_filt* filter mode.

8.4 Loading Texture Objects

As shown in "9 Texture Environment (TEV)" on page 91, the default texture pipeline configuration accepts up to eight texture coordinates, each associated with a texture map. To associate a texture with `GX_TEXMAP0`, for example, you use the function in Code 8-5.

Code 8-5 GXLoadTexObj

```
GXLoadTexObj(GX_TEXMAP0, &myTexObj);
```

Loading a texture object only loads the state describing the texture into the hardware. The default GX configuration allocates texture memory as caches and automatically assigns a cache to use with this texture when you call `GXLoadTexObj`. Once you have loaded a texture object, you may render polygons using that texture. Up to eight texture objects can be loaded at once for multitexturing. There is no need to synchronize the texture state with primitives explicitly. The GP automatically synchronizes state changes with pixels in the rendering pipeline.

Refer to "9 Texture Environment (TEV)" on page 91 for more advanced control of texture coordinate ordering.

8.5 Loading Texture Lookup Tables (TLUTs)

Code 8-6 Loading TLUTs

```
GXInitTlutObj(
    GXTlutObj*    obj,
    void*         lut,
    GXTlutFmt     fmt );

GXLoadTlut(
    GXTlutObj*    obj,
    u32           tlut_name,
    u16           start_entry,
    u16           n_entries );
```

In order to use a color-indexed texture, you must first load a texture lookup table (TLUTs) into TMEM. The default configuration of TMEM allows for 20 TLUTs, 16 with 256 entries, and four with 1kb entries (see the `GXTluts` enumeration). To load a TLUT, you follow steps similar to loading a texture object:

1. Describe the location and format of the TLUT in main memory using `GXInitTlutObj`.
2. Load the TLUT into one of the named TLUTs in texture memory using `GXLoadTlut`.
3. Use `GXInitTexObjCI` to describe the color-indexed texture.
4. Set the *tlut_name* parameter to the TLUT name you have loaded.

Pointers to TLUTs in main memory must be aligned to 32 bytes. TLUTs must be a multiple of 16 entries, with 16 bits for each entry. The total number of entries loaded using `GXLoadTlut` must also be a multiple of 16.

8.6 How to Override the Default Texture Configuration

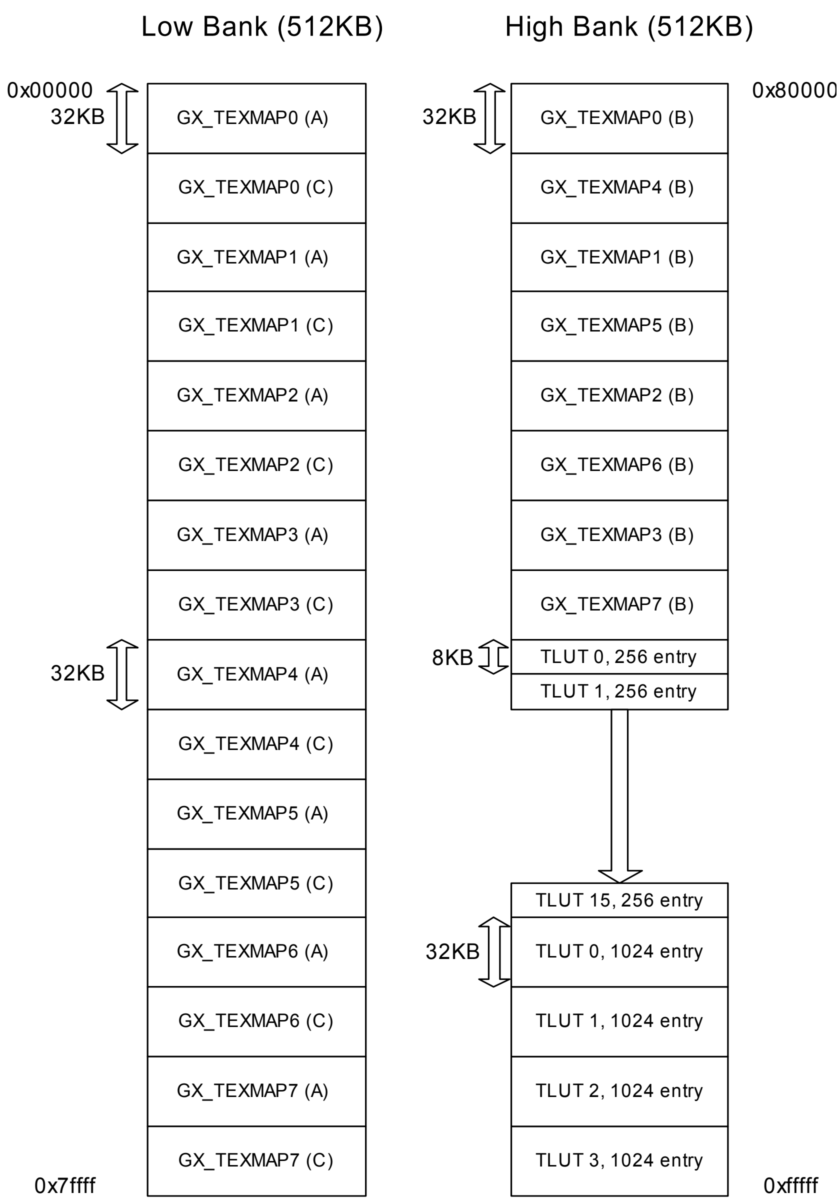
So far, we have discussed the texture pipeline only in the context of the default configuration set by `GXInit`. The purpose of this configuration is to abstract some of the complexities of the GX API in order to allow the developer to concentrate on the basic features. However, developers may easily override the default configuration in order to tailor the system to fit their specific applications more closely. This section discusses how a developer can take advantage of the GX API's additional flexibility to configure the following options:

- Allocation of TMEM.
- Binding of textures to texture caches and preloaded textures at run time.

8.6.1 Texture Regions

The TMEM is a 1MB high-speed 1TSRAM that can be configured to contain texture caches, preloaded textures, and TLUTs. The GX API uses a simple TMEM management scheme that is set up by `GXInit`. This scheme assumes that texture caches and TLUT *regions* are pre-allocated in TMEM. The default configuration does not allow use of pre-loaded textures. Regions are simply defined by a TMEM pointer and size. At runtime, textures are bound to a particular cache region by calling `GXLoadTexObj` according to information such as the texture ID of the target and the texture format. TLUT data is loaded into TLUT regions by calling `GXLoadTlut`. `GXInit` configures TMEM as shown in Figure 8-11.

Figure 8-11 Default TMEM Configuration



Logically, the 1MB TMEM is split into two 512KB low and high banks. The default configuration defines 16 texture caches for the low bank and eight texture caches for the high bank. When textures are used, each texture ID (GX_TEXMAP0, GX_TEXMAP1, ... , GX_TEXMAP7) is allocated two texture caches from the low bank and one from the high bank.

TLUTs that are used for color index format are allocated in the remainder of the high bank. There are 16x256-entry and 4x1024-entry TLUTs defined. TLUTs must always be allocated in the high bank.

When texture is without mipmapping and does not have a 32-bit format, only one cache region in the low bank (part A in Figure 8-10) is used. Cache regions in both the low bank and the high bank (parts A and B in Figure 8-10) are used only when a 32-bit format is used.

When 32-bit formatted mipmapped textures that are not color index formatted are used, even LODs are cached in one bank while odd LODs are cached in the opposite bank. Thus, one cache region from the low bank and one cache region from the high bank (parts A and B in Figure 8-10) are used.

When color index textures with mipmapping are used, both odd LODs and even LODs are allocated in the low bank (parts A and C in Figure 8-10). This is because the color index texture region must always be allocated in the low bank (that is, they must be in the bank opposite from the TLUTs).

When 32-bit format with mipmapping is used, it is necessary to have twice the continuous cache region for the odd LODs and even LODs as compared to when normal texturing is used. In this case, the necessary regions (parts A and C in Figure 8-10) are allocated from the low bank. However, because a sufficient number of regions cannot be secured in the high bank, cache regions and adjoining textures with different map IDs are shared. No operational problems are caused when one cache region is shared by two or more textures. But performance may be affected.

The GX API allows the application to configure texture regions using `GXInitTexCacheRegion` and `GXInitPreLoadRegion`. Application developers can also define their own region-binding schemes by registering a callback function with `GXSetTexRegionCallback` or `GXSetTlutRegionCallback`.

8.6.2 Cached Regions

Texture regions describe areas of TMEM that can be used as texture caches or preloaded textures.

Code 8-7 GXInitTexCacheRegion

```
GXInitTexCacheRegion(
    GXTexRegion*    region,
    GXBool          is_32b_mipmap,
    u32             tmem_even,
    GXTexCacheSize  size_even,
    u32             tmem_odd,
    GXTexCacheSize  size_odd );
```

For cached images, the *s*/*t* coordinates are translated into cache tag memory addresses. If the addressed tag indicates the texture is resident in TMEM, the *s*/*t* coordinates are translated into TMEM addresses, and the texture is accessed. If the texture is not resident, the GP will issue memory requests to copy the texture from main memory to TMEM. Once the texture is resident, the *s* and *t* coordinates are translated into TMEM addresses and the texture is accessed. The unit of texture access is a texture cache line (32 bytes). All textures are stored as a multiple of this line size. The GP makes texture requests ahead of time and stores the TMEM addresses in a FIFO. Prefetching allows rendering to proceed during a cache miss.

The GP uses main memory addresses for tags, so caches can be shared among textures (lower performance) without needing to invalidate between texture loads. Mipmaps that are to be trilinearly filtered must allocate a cache region in both the low and high banks. The *tmem_even* parameter defines the location in TMEM where even LODs will be cached. The *tmem_odd* parameter defines the location in TMEM where odd LODs will be cached. Usually, the two types of LODs must be placed in opposite banks. For non-mipmapped (planar) textures (except `GX_TF_RGBA8`), only the **_even* parameters need to be defined. Planar (except color index) texture regions can be allocated in either the low or high bank, but cannot span both banks. Color-indexed texture regions (both planar and mipmapped) must always be allocated in the low TMEM bank. The *size_even* and *size_odd* parameters describe the size of the cache region for their respective sets of LODs. Unused odd parameters should be set to 0 (address) and `GX_TEXCACHE_NONE` (size).

Full-color textures (`GX_TF_RGBA8`) can only be mipmapped in two cycles, with the even LOD accessed on the first cycle and the odd LOD accessed on the next cycle. In this case, the *tmem_even* cache (which must be in the low bank) is used to store the AR (alpha and red) components of the texture and the *tmem_odd* cache (which must be in the high bank) is used to store the GB (green and blue) components. (Please note that this explanation is simplified; the actual storage is slightly different.) Within each bank, the even LODs are cached first, followed by the odd LODs. For this case, the *size_even* and *size_odd* parameters refer to the size of the cache region for their respective LODs within *one* bank. Thus the actual cache memory usage will be *twice* the sum of the even and odd sizes. The parameter *is_32b_mipmap* indicates the region will be used in this manner.

TMEM caches are sized in terms of superlines (512B or 4x4 lines). TMEM caches can be only one of three sizes: 32KB, 128KB, and 512KB. Each cache pointer must be aligned to 2KB (2x2 superlines).

The default cached regions created by `GXInit` are 8x8 superlines (32KB).

8.6.3 TLUT Regions

Code 8-8 GXInitTlutRegion

```
GXInitTlutRegion(
    GXTlutRegion*    region,
    u32              tmem_addr,
    GXTlutSize       tlut_sz );
```

TLUTs must be allocated in the high bank of TMEM. Color-indexed texture regions must be allocated in the low bank of TMEM. Each 16-bit entry of a TLUT in main memory is replicated into 16 copies during the TLUT load. Therefore, the total memory in bytes that needs to be allocated for a TLUT is *tlut_sz* * 16 * 2B. A 256-entry TLUT requires an 8KB TLUT region.

The *tmem_addr* for the TLUT region must be aligned to 512B (16 entries * 16 * 2B). Furthermore, *tmem_addr* must be aligned to the size of TLUT. For example, a 256-entry TLUT should be aligned to an 8KB TMEM address.

The TLUT region size can be any power of 2 ranging from 16 entries to 16kb entries. The *tmem_addr* is bitwise OR'd with the texel index to determine the address of the entry. This makes it possible to create index sizes that are smaller than the texel type indicates.

For example, you can create a 1024-entry TLUT and access it using the `GX_TF_C14X2` (16-bit) texture type. Since a 1024-entry table requires a 10-bit index, the most significant 6 bits of each index in the texture should be set to zero.

8.6.4 Preloaded Regions

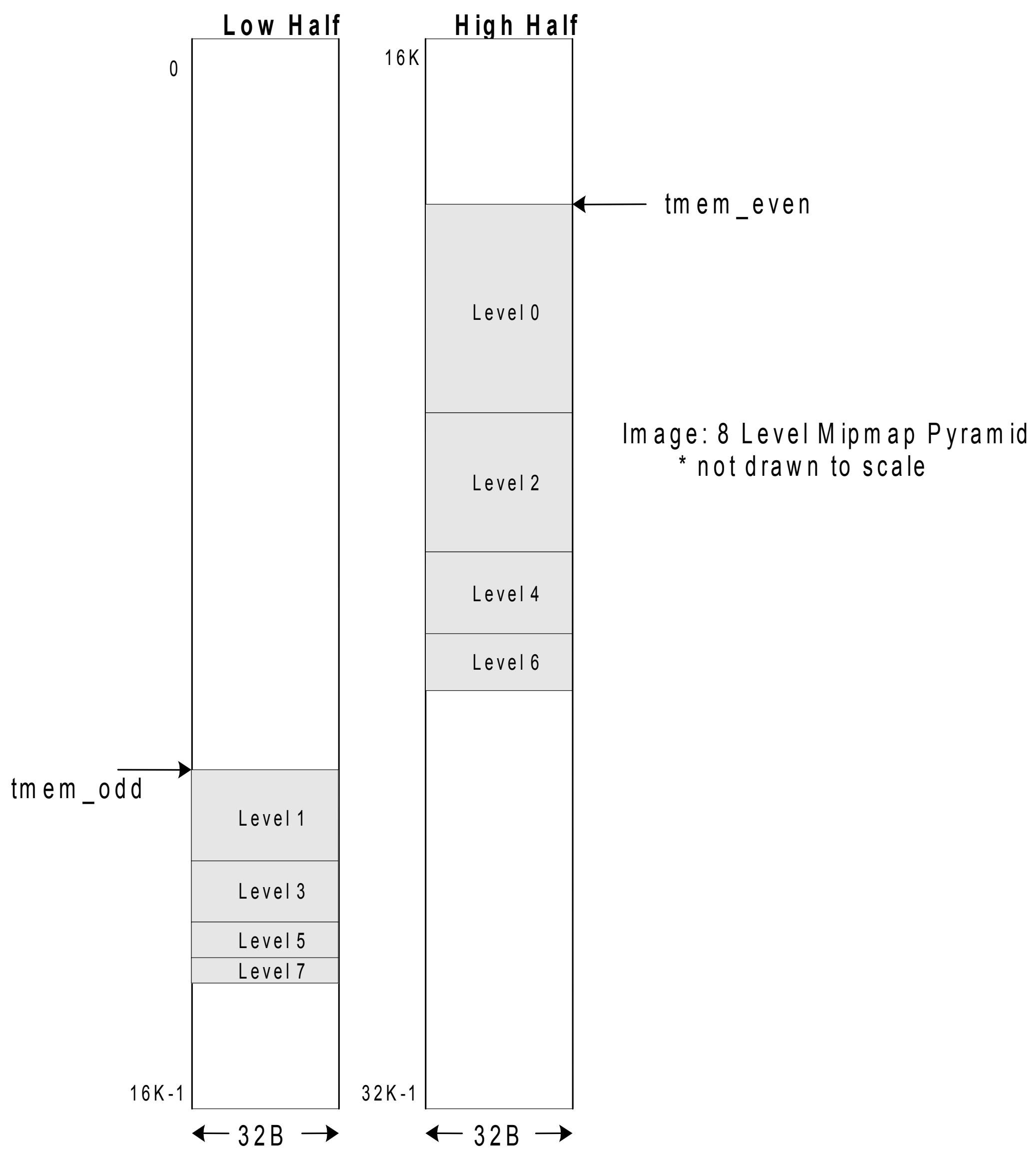
Code 8-9 GXInitTexPreLoadRegion()

```
void GXInitTexPreLoadRegion(
    GXTexRegion*      region,
    u32               tmem_even,
    u32               size_even,
    u32               tmem_odd,
    u32               size_odd );
```

Preloaded textures are loaded explicitly by the application into a TMEM region. The texture is stored in TMEM in the same format as main memory. Unlike cached regions, the texture cache tag memory is not checked when accessing a preloaded region. Small, frequently-used textures are good candidates for preloading. Preloaded textures are not supported by the default configuration of TMEM (the entire TMEM is mapped as caches or TLUTs).

The *size_even* and *size_odd* parameters to `GXInitTexPreLoadRegion` specify the size of the texture regions in bytes. The *size_even* region is used for even LODs and the *size_odd* is used for odd LODs when mipmapping. When preloading non-mipmapped (non-`GX_TF_RGBA8`) images, you only need to specify the *_even* parameters.

Figure 8-12 Mipmap in TMEM



The *tmem_even* and *tmem_odd* parameters to `GXInitTexPreLoadRegion` for a preloaded texture are required only to be 32B-aligned. For preloaded mipmaps, *tmem_even* and *tmem_odd* must be in opposite TMEM banks. The *tmem_even* value will define the location of all even LODs and *tmem_odd* will define the location of all odd LODs.

Note: Even LODs will use more memory in their bank of TMEM than the odd LODs in the opposite bank.

Figure 8-13 Planar Texture in TMEM

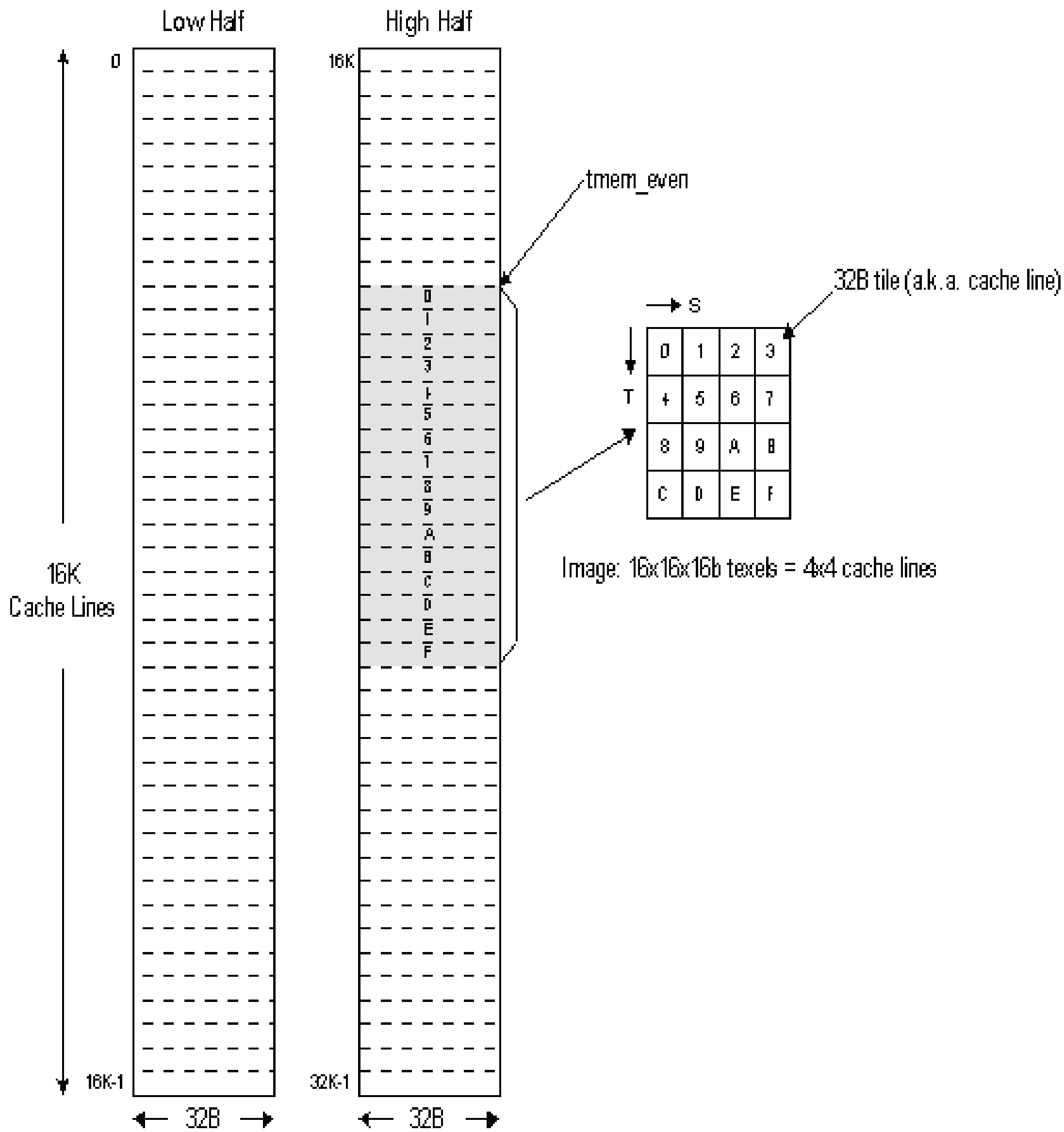
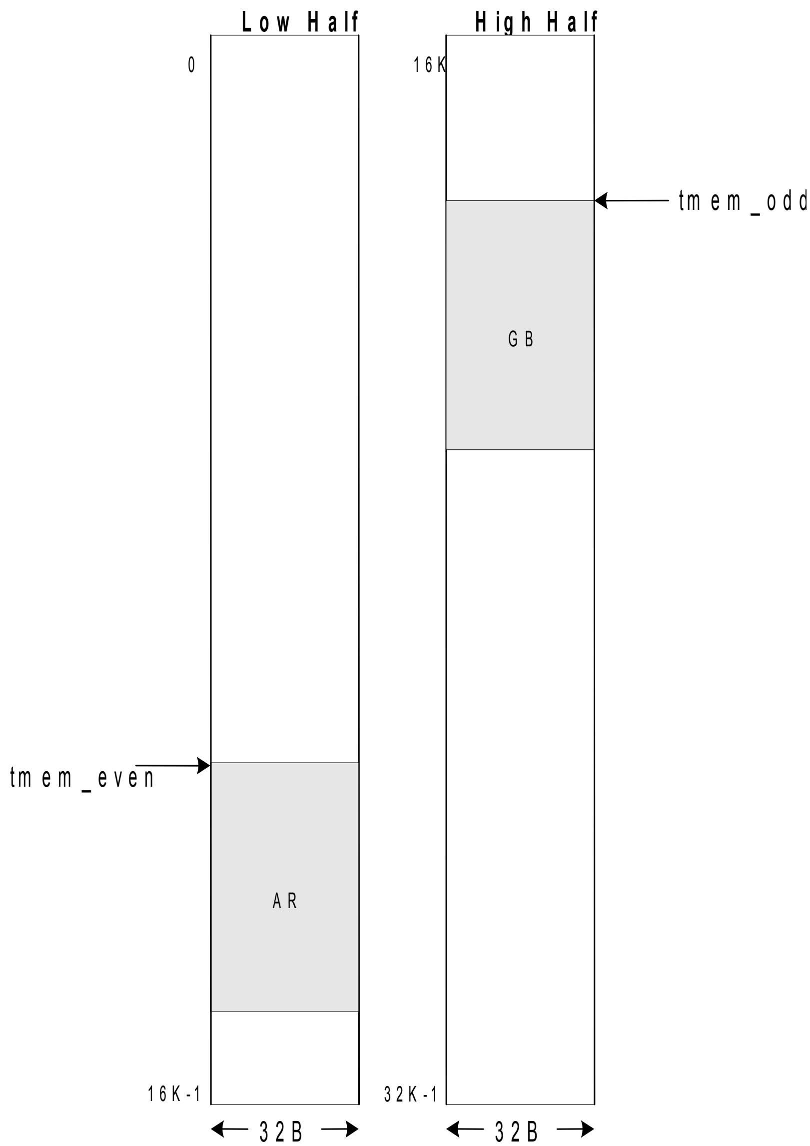


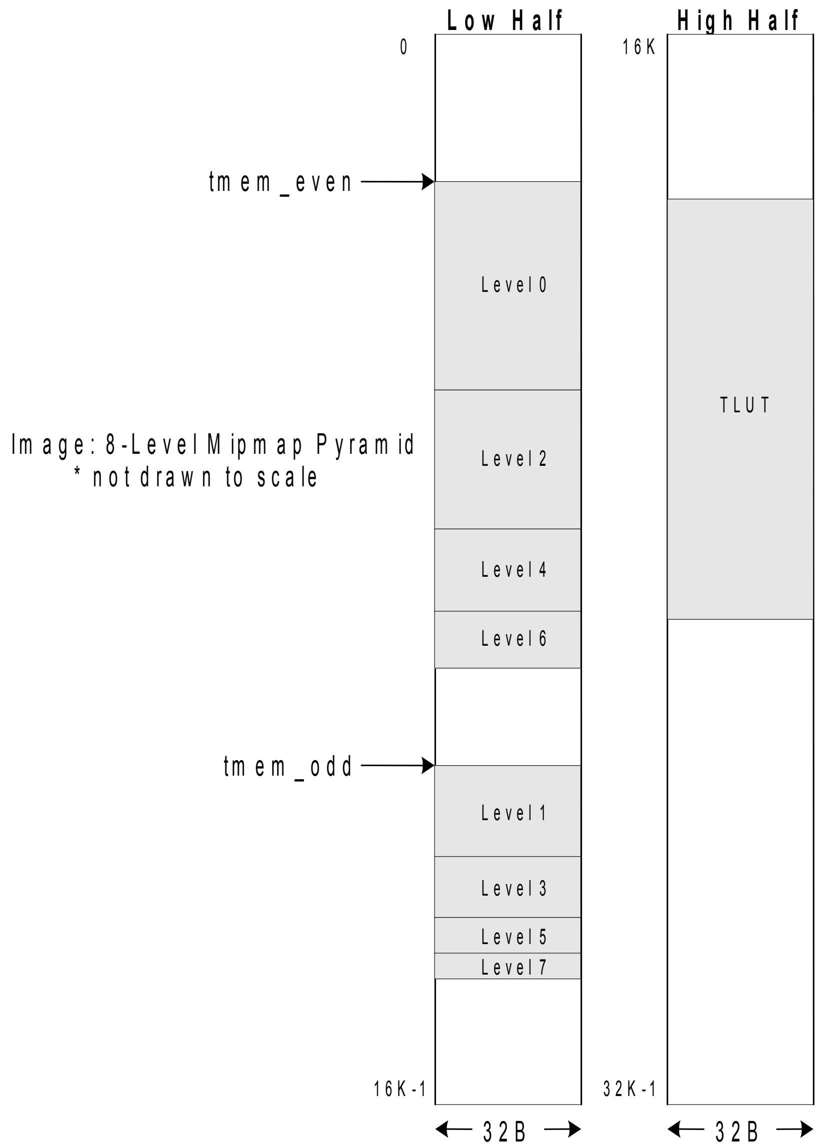
Figure 8-14 32-bit Planar Texture in TMEM



For planar non-color index textures (except 32-bit), only *tmem_even* is used, and may be located in either the high or low bank of TMEM. For color index textures, *tmem_even* must be in the low bank of TMEM.

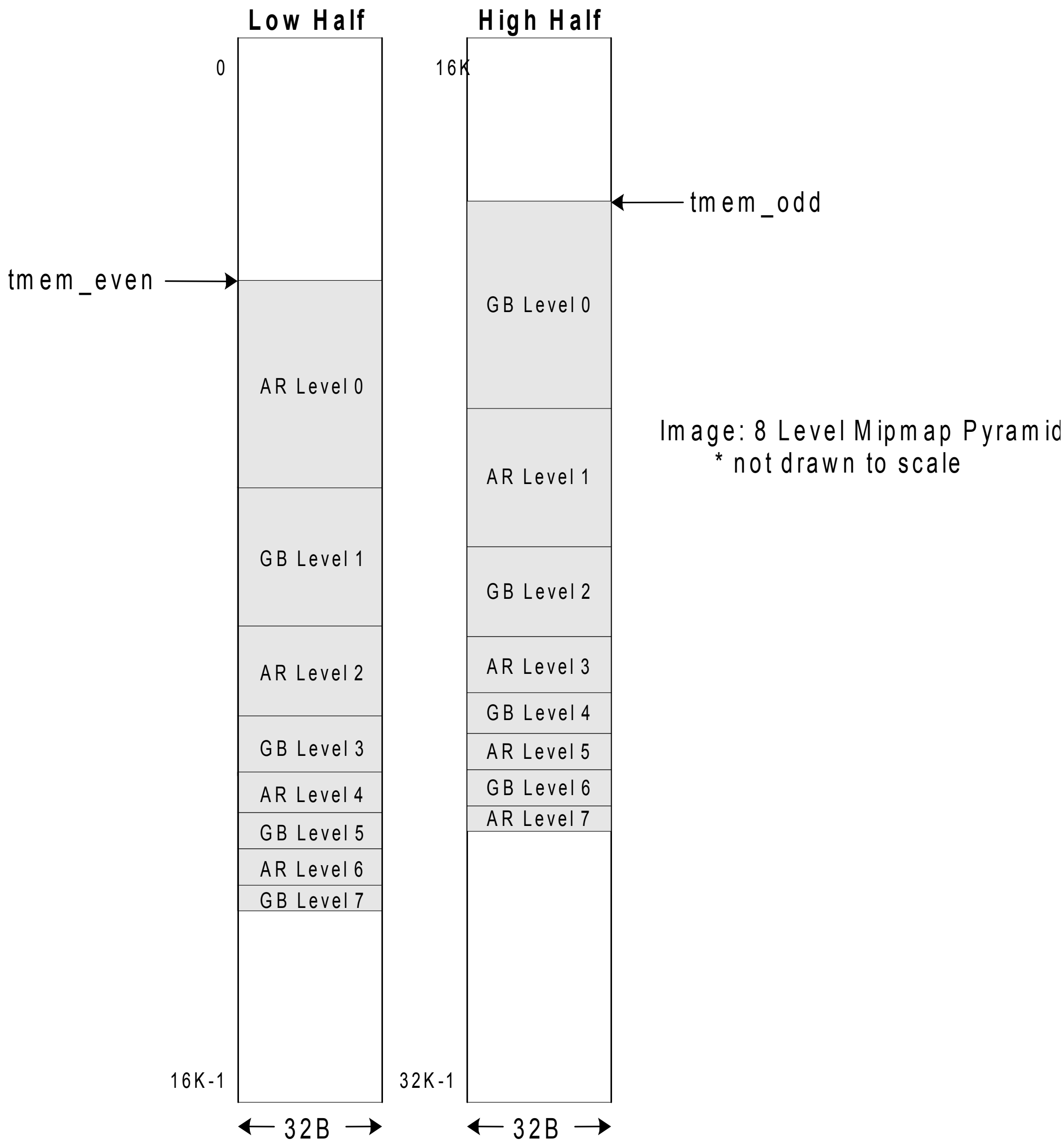
For 32-bit planar textures (`GX_TF_RGBA8`), the *tmem_even* is the address of the AR tiles and must be in the low bank of TMEM. The GB tiles are located at *tmem_odd* and must be in the high bank of TMEM.

Figure 8-15 Color Index Mipmap in TMEM



For color index preloaded textures, *tmem_even* and *tmem_odd* must be in the low bank of TMEM. The hardware will load all even LODs at the *tmem_even* address and all odd LODs at the *tmem_odd* address.

Figure 8-16 32-bit Mipmap in TMEM



`GX_TF_RGBA8` textures are stored as interleaved AR and GB tiles (32B/tile). When preloading a `GX_TF_RGBA8` texture, the AR and GB are written to opposite TMEM banks. The *tmem_even* (AR) tiles should be in the low TMEM bank and the *tmem_odd* (GB) tiles should be in the high TMEM bank.

Note: 32-bit textures use the same amount of memory in the low and high banks.

To actually load the texture into TMEM, call the function in Code 8-10.

Code 8-10 GXPreLoadEntireTexture()

```
GXPreLoadEntireTexture(
    GXTexObj*    obj,
    GXTexRegion* region );
```

To associate a hardware texture map ID (`GXTexMapID`) with the preloaded region, use:

Code 8-11 GXLoadTexObjPreLoaded()

```
void GXLoadTexObjPreLoaded(
    GXTexObj*         obj,
    GXTexRegion*      region,
    GXTexMapID        id );
```

8.6.5 Texture Cache Allocation

You can override the default TMEM allocation by replacing the callback function that binds texture objects to texture regions, as shown in Code 8-12.

Code 8-12 GXSetTexRegionCallback

```
typedef GXTexRegion *(*GXTexRegionCallback)( GXTexObj *tex_obj );

GXTexRegionCallback GXSetTexRegionCallback( GXTexRegionCallback f );
```

This function is called by `GXLoadTexObj` and is expected to return a pointer to a `GXTexRegion` to use for this texture. These texture regions can be statically allocated or dynamically allocated according to the needs of the application. The TMEM allocation scheme must also consider color index TLUT and preloaded texture memory requirements. `GXSetTexRegionCallback` returns the callback that was set prior to its invocation.

The programmer can use the `GXInitTexObjUserData` function to set a pointer to user data in the texture object. This data may be needed in order to implement a better TMEM allocation strategy. The data can be retrieved using `GXGetTexObjUserData`.

8.6.6 TLUT Allocation

If color index textures are to be used, the TMEM allocation scheme must consider TLUT region allocation in addition to texture region allocation. As described in "8.6.3 TLUT Regions" on page 76, there are more restrictions on the placement of color index textures and TLUTs in TMEM than on non-color index textures. To override the default TLUT allocation scheme, the application must replace the callback function that associates a TLUT *name* with a TLUT region as shown in Code 8-13.

Code 8-13 GXSetTlutRegionCallback

```
typedef GXTlutRegion *(*GXTlutRegionCallback)(u32 name) );

GXTlutRegionCallback GXSetTlutRegionCallback( GXTlutRegionCallback f );
```

The callback function is called by `GXLoadTexObj` to associate the TLUT ***name*** (`GXInitTexObjCI`) with a TLUT region. `GXLoadTlut` also calls the callback function. `GXSetTlutRegionCallback` returns the callback that was set prior to its invocation.

8.7 Invalidating Texture Cache

The texture hardware maintains a Cache Tag Memory that maps a texture cache line's main memory address to its TMEM address. Because of this, textures can share a cached region without address collisions. However, the following situations will require invalidating the texture cache:

- The texture is moved to a new main memory location.
- A new texture is copied into the memory occupied by a previously used texture.
- The application modifies some texels of a texture in main memory.

Invalidating the texture cache requires resetting the state of certain tag bits in the cache tag memory. This will force the reloading of the affected texture. Functions for invalidating either a texture region or the entire TMEM are shown in Code 8-14.

Code 8-14 Invalidating Texture Memory

```
GXInvalidateTexRegion( GXTexRegion* region );
GXInvalidateTexAll( void );
```

It is not necessary to invalidate TLUT regions (see "8.6.3 TLUT Regions" on page 76) and preloaded regions, because they are explicitly loaded into TMEM. If the data in a TLUT or preloaded texture is changed, the application must reload it for the change to take effect.

8.8 Changing the Usage of TMEM Regions

Sometimes an application will change the use of a particular region of TMEM from preloaded to cached or from TLUT to cached. In these cases—and only these—the application should call `GXTexModeSync` to ensure all texels currently in the pipeline are flushed before the change in usage goes into effect. The call should be made prior to drawing any primitives that will use the TMEM region in the new mode. When changing a TMEM region from cached to preloaded (or TLUT), the command to load the TMEM region will synchronize the pipeline automatically.

8.9 Creating Textures by Copying the Embedded Frame Buffer

Textures can be created by copying the Embedded Frame Buffer (EFB) to main memory using the `GXCopyTex` function. This is useful when creating dynamic shadow maps, environment maps, motion blur effects, etc.

All non-color index texture types except compressed textures (`GX_TF_CMPR`) can be created during the copy. The texture copy operation will create the correct tiling and formatting of the texture so it can be read directly by the hardware. Optionally, you can apply a box filter to the image in the EFB in order to create a lower level of detail (LOD) texture. The box filter can be used to create mipmaps from the EFB data.

The EFB can be used in either of two basic modes: antialiased (pixel format of `GX_PF_RGB565_Z16`) and non-antialiased (pixel format of `GX_PF_RGB8_Z24` or `GX_PF_RGBA6_Z24`). You can copy color textures in either mode, but Z textures can only be copied from non-antialiased frame buffers. See "12 Video Output" on page 129 for more details on the EFB modes.

Table 8-4 Texture Copy Formats and Conversion Notes

Format	Conversion
GX_TF_I4	4-bit intensity (see Note 1).
GX_TF_IA4	4-bit intensity and 4-bit alpha (see Note 1).
GX_TF_I8	8-bit intensity (see Note 1).
GX_TF_IA8	8-bit intensity and 8-bit alpha (see Note 1).
GX_TF_RGB565	16-bit color (RGB565).
GX_TF_RGB5A3	16-bit color and alpha. This is RGB555 (opaque) when the MSB is 1 and RGBA4443 (transparent) when the MSB is 0.
GX_TF_RGBA8	32-bit full color RGBA (8 bits per component).
GX_TF_Z8	Copy the upper 8 bits from the Z buffer to an 8-bit format.
GX_TF_Z16	Copy the upper 16 bits from the Z buffer to a 16-bit format (see Note 2).
GX_TF_Z24X8	Copy all 24 bits in the Z buffer into a 32-bit format. The upper 8 bits will be set to `0xFF`.
GX_CTF_R4	Copy 4 bits from the R component. The result will be handled in I4 format.
GX_CTF_RA4	Copy 4 bits from the R component and 4 bits from the alpha component. The result will be handled in IA4 format.
GX_CTF_RA8	Copy 8 bits from the R component and 8 bits from the alpha component. The result will be handled in IA8 format.
GX_CTF_A8	Copy 8 bits from the alpha component. The result will be handled in I8 format.
GX_CTF_R8	Copy 8 bits from the R component. The result will be handled in I8 format.
GX_CTF_G8	Copy 8 bits from the G component. The result will be handled in I8 format.
GX_CTF_B8	Copy 8 bits from the B component. The result will be handled in I8 format.

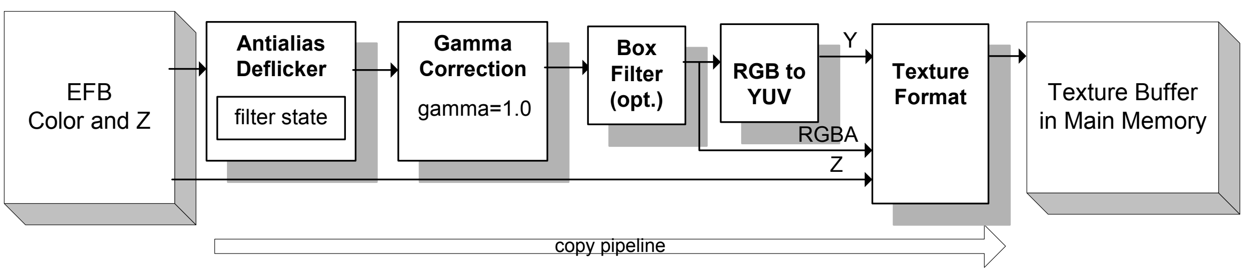
Table 8-4 Texture Copy Formats and Conversion Notes

GX_CTF_RG8	Copy 8 bits from the R component and 8 bits from the G component. The result will be handled in IA8 format.
GX_CTF_GB8	Copy 8 bits from the G component and 8 bits from the B component. The result will be handled in IA8 format.
GX_CTF_Z4	Copy the upper 4 bits of the Z value into a 4-bit format.
GX_CTF_Z8M	Copy the central 8 bits of the Z value into an 8-bit format.
GX_CTF_Z8L	Copy the lower 8 bits of the Z value into an 8-bit format.
GX_CTF_Z16L	Copy the lower 16 bits of the Z value into a 16-bit format (see Note 2).

Note 1: When copying textures to an intensity format (`GX_TF_I4`, `GX_TF_I8`, `GX_TF_IA4`, `GX_TF_IA8`), the Y value resulting from the RGB-to-YUB conversion will be used as the intensity value. This means that the output range is linearly compressed into the range between 16 and 235. If you want to get accurate values from the EFB, you can use a format such as `R8` or `RA8`.

Note 2: Hardware configuration dictates that when a texture is copied into a 16-bit Z format, the texture is laid down in reverse byte order. These results can therefore not be used directly as Z textures.

Figure 8-17 Texture Copy Data Path



The RGB-to-YUV conversion that takes place in the texture copy pipeline is the same as that which is used for the display copy pipeline for video output (see "12 Video Output" on page 129). As a result, the intensity is scaled: 16 <= Y <= 235.

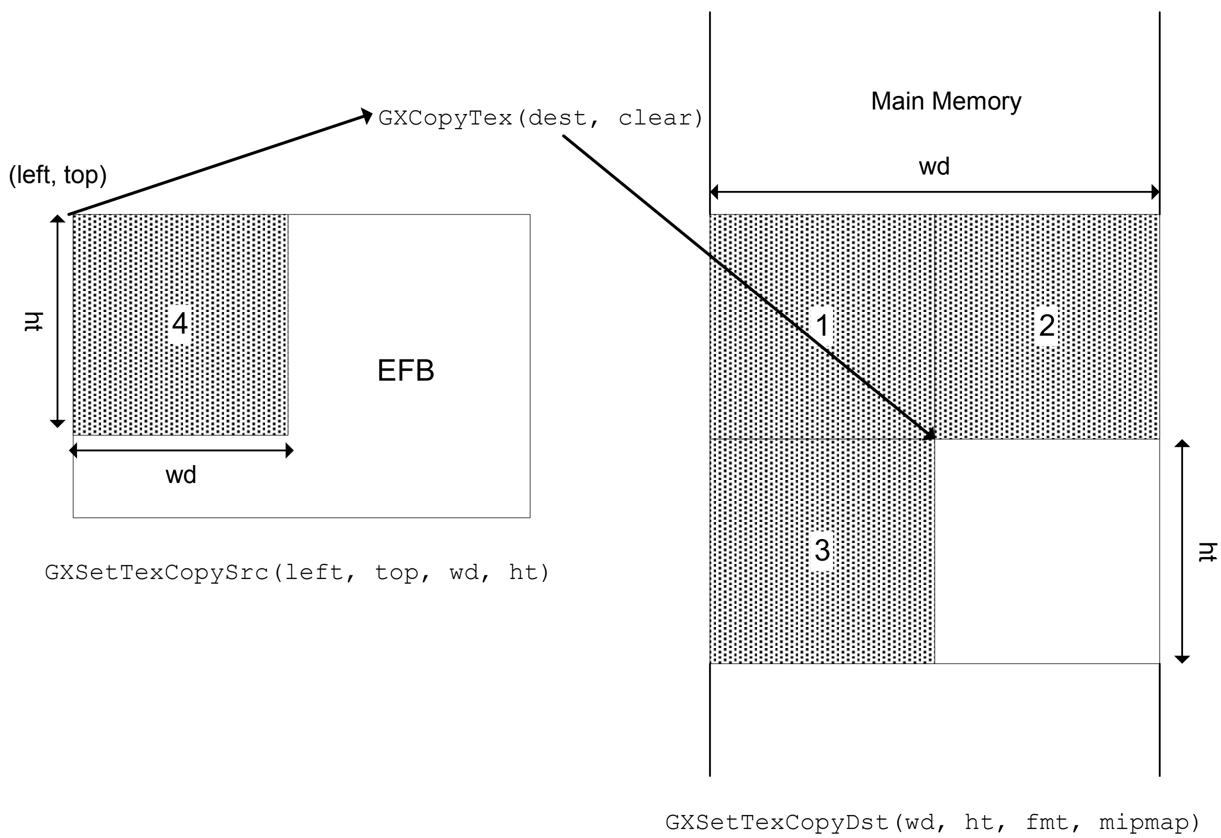
The following functions control the copying of textures from the EFB to main memory:

Code 8-15 Texture Copy Functions

```
void GXSetTexCopySrc(
    u16         left,
    u16         top,
    u16         wd,
    u16         ht );

void GXSetTexCopyDst(
    u16         wd,
    u16         ht,
    GXTexFmt    fmt,
    GXBool      mipmap );

void GXCopyTex(
    void*       dest,
    GXBool      clear );
```

`GXSetTexCopySrc` specifies the source rectangle to copy from the EFB. All parameters to `GXSetTexCopySrc` must be multiples of two pixels. `GXSetTexCopyDst` specifies the destination rectangle in main memory. Normally, the source and destination rectangles would have the same size. However, when copying small textures that will be composited into a larger texture, the source and destination rectangles may differ.

Figure 8-18 Copying Small Textures into a Larger Texture in Main Memory



The *fmt* argument of GXSetTexCopyDst can specify a few subtle types of texture copy operations. The format GX_CTF_A8 is used specifically to copy the *alpha* channel from the EFB into a GX_TF_I8 formatted texture. GX_TF_I8 will copy the scaled *luminance* of the EFB into a GX_TF_I8 texture. When reading a texture GX_CTF_A8 and GX_TF_I8 are equivalent.

When color textures are converted from an GX_PF_RGB8_Z24 pixel format to a lower-resolution color format, such as GX_TF_RGB565, the least significant bits (LSBs), of the 8-bit colors are truncated. When color textures are converted from a lower-resolution pixel format, say GX_PF_RGB565_Z16, to a higher resolution texture format, say GX_TF_RGBA8, the most significant bits (MSBs) of each pixel are replicated in the LSBs of each texel. This conversion process distributes the estimation error evenly and allows each texel to represent the minimum or maximum value.

In general, you should only copy textures containing alpha from an EFB with format GX_PF_RGBA6_Z24. When copying a texture containing alpha from an EFB without alpha, alpha will be set to its maximum value.

The GX_TF_Z24X8 format can be used to copy the 24-bit Z buffer to a 32-bit texture (equivalent format to GX_TF_RGBA8, see Appendix D). The next section describes how this “Z texture” can be used. It is not legal to copy out 8-bit or 16-bit Z textures, and you may not copy Z from an antialiased EFB (see "12 Video Output" on page 129 for more details on EFB formats).

The function GXCopyTex initiates the copy operation. It can conditionally clear the EFB at the same time if *clear* is true. The update enable flags for each buffer to be cleared must also be enabled (see GXSetColorUpdate, GXSetAlphaUpdate, and GXSetZMode). This allows individual buffers to be conditionally cleared. The copy filter in effect at the time the texture is copied (see GXSetCopyFilter) will be applied to the EFB data during the conversion process.

To copy a texture, the application must first allocate a buffer in main memory that is the size of the texture to be copied. This size can be determined using GXGetTexBufferSize. This function takes texture padding and texture type into account in its calculations.

Before a copied texture can be applied to a textured primitive, you must ensure that the copy operation has finished. The command GXPixModeSync may be called after GXCopyTex in order to guarantee this. GXPixModeSync flushes the pixel pipeline, ensuring that the copy is finished before a new primitive is started.

Sometimes it is useful to determine the screen rectangle that encloses a group of rendered geometry. The functions GXClearBoundingBox and GXReadBoundingBox can determine the bounding box of geometry rendered in the EFB. Call GXClearBoundingBox first to reset the bounding box, then render the geometry. The GP will update the minimum and maximum screen-space (x, y) continually for each pixel drawn. After rendering the geometry, the bounding box values can be read back using GXReadBoundingBox. You can use these values to compute the arguments to GXSetTexCopySrc.

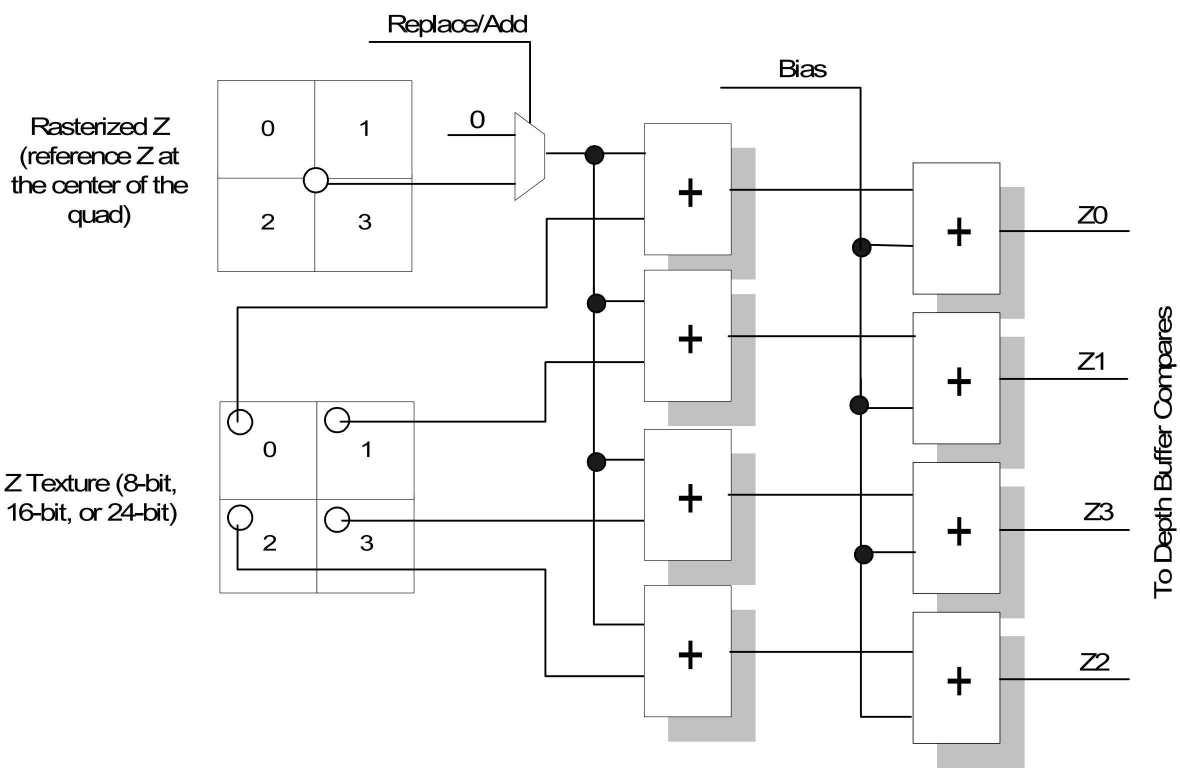
8.10 Z Textures

The GP supports combining a color texture and a Z texture into the Embedded Frame Buffer (EFB). This feature can be used to facilitate image-based rendering, in which a frame buffer is a composite of smaller color and depth images, like sprites with depth. Each of the sprites can be computed at independent frame rates, but the final frame buffer is composited at the target frame rate.

You can create Z textures by copying a region of the EFB using GXCopyTex with the pixel format (see GXSetTexCopyDst) set to GX_TF_Z24X8. Additionally, 8- and 16-bit Z textures can be created offline or by using the CPU. The GXCopyTex function *cannot* create 8- or 16-bit Z textures. Moreover, Z textures *cannot* be copied from an antialiased EFB.

The texture input to the *last* active TEV stage is used as the Z texture, so the application must be careful to arrange the TEV equation so the Z texture is output by this stage. When Z texturing is enabled, color is output from the *last* active TEV stage, but the texture input to the last stage is not available (because it is used for the Z texture).

Figure 8-19 Z Texture Block Diagram



The Z texture can either replace or offset the rasterized Z of the polygon. Normally, each pixel's Z in a quad is computed by adding Z slopes to a reference Z computed at the center of the quad. When Z texture is enabled, the Z texels will offset the reference Z (the Z slopes will not be added, and thus the computed Z accuracy will be per-quad, not per-pixel). In addition, a constant bias can be added to the result. Finally, if Z buffering is enabled, the resulting Z is compared with the EFB current Z. The pipeline must be configured to Z buffer *after* texture lookup when using Z textures. The Z-texture adders do not clamp, so the programmer must make sure that there is no overflow.

Code 8-16 GXSetZTexture

```
void GXSetZTexture(
        GXZTexOps       op,
        GXTexFmt        fmt,
        u32             bias );
```

The *fmt* argument above can be one of the following:

- GX_TF_Z8
- GX_TF_Z16
- GX_TF_Z24X8

8.11 Texture Performance

This section documents peak texture performance assuming a 243 MHz graphics processor clock speed.

Table 8-5 Texture Performance

Texture Count	Performance
1	972 megapixels/second
2	486 megapixels/second
3	324 megapixels/second
4	243 megapixels/second
5	194 megapixels/second
6	162 megapixels/second
7	139 megapixels/second
8	122 megapixels/second

Note: 32-bit textures filter bilinearly at 972 megapixels/second and trilinearly at 486 megapixels/second. Expect less than 15% degradation for properly sampled cached textures.

9 Texture Environment (TEV)

9.1 Description

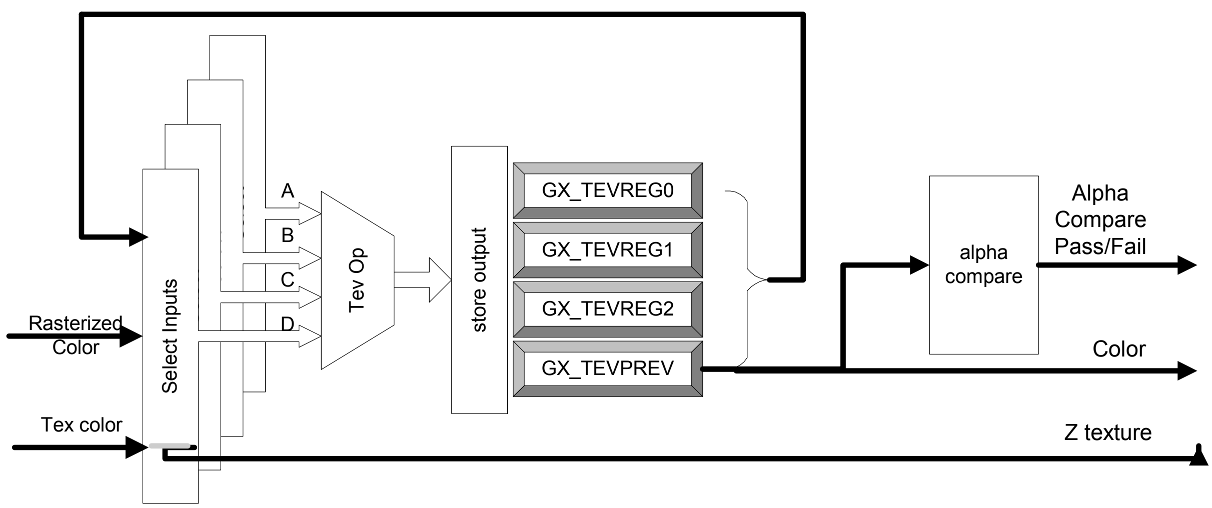
The Texture Environment (TEV) combines per-vertex lighting, textures, and constant colors to form the pixel color (before fogging and blending). The color and alpha components have independent TEV units with independent controls. There is only one set of TEV color/alpha-combiners implemented in hardware. To implement multi-texture, the TEV hardware is reused over multiple cycles, called TEV stages. Each TEV stage has independent controls and a maximum of 16 TEV stages are supported. A consecutive number of TEV stages may be enabled in order to perform multi-texturing. The resulting pixel color is output from the last active TEV stage. The last stage *must* send its output to the `GX_TEVPREV` register.

A set of four input/output color registers are provided to store temporary results, pass results from one stage to the next, or to supply user-defined constant colors. These color registers are shared among all TEV stages. The last stage *must* send its output to the `GX_TEVPREV` register.

The alpha produced by the last TEV stage is input to an alpha-compare equation. The result of the alpha compare can be used to conditionally mask color (and possibly Z) writes to the frame buffer.

Fog, if enabled, is applied to the pixel values output from the last active TEV stage. Blending operations occur after fogging. (Fog and blending are described in later chapters.)

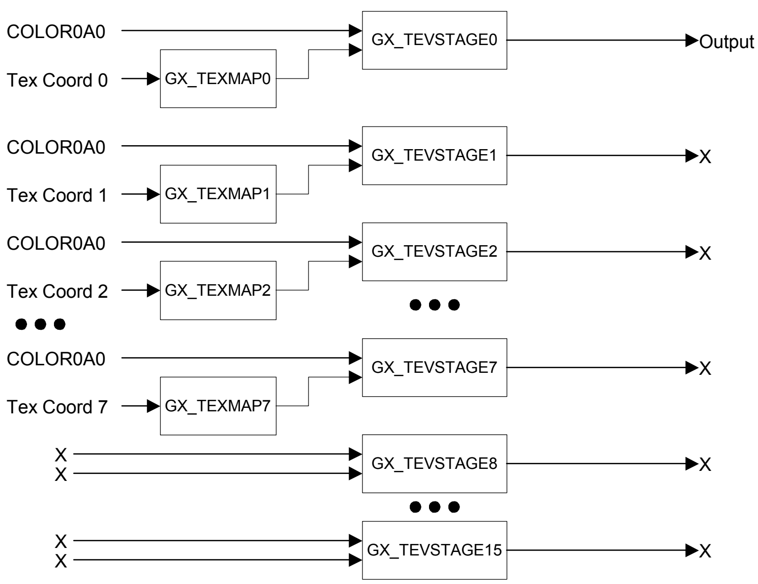
Figure 9-1 TEV Block Diagram



9.2 Default Texture Pipeline Configuration

By default, the texture pipeline is configured to look like the diagram below. Each texture coordinate enabled for a vertex (using `GXSetVtxDesc`) is sent down the pipeline in the input order. Each coordinate accesses a texture, and the resulting texel is fed to the corresponding TEV stage. Since only eight unique textures are available, only eight TEV stages are configured this way; the remaining eight stages are initialized with null texture and color inputs. The number of active TEV stages and the number of texture coordinates that are generated must be set by the application (see `GXSetNumTevStages` and `GXSetNumTexGens`). By default, only stage 0 produces an output. The default operation for stage 0 is `GX_REPLACE`, meaning only the texture color is output (see "9.4 GXSetTevOp" on page 92).

Figure 9-2 Default Texture Pipeline



The texture pipeline is very configurable. The user can override this simple default configuration as described in the following sections.

9.3 Number of Active TEV Stages

To program the number of active TEV stages, use the function in Code 9-1.

Code 9-1 GXSetNumTevStages

```
GXSetNumTevStages( u8 stages );
```

There must always be at least one TEV stage enabled. TEV stages are enabled consecutively. A maximum of 16 TEV stages may be enabled.

9.4 GXSetTevOp

We provide the function `GXSetTevOp` to simplify initial programming demos. It determines the color processing that occurs at the specified TEV stage. This function calls `GXSetTevColorIn`, `GXSetTevAlphaIn`, `GXSetTevColorOp`, and `GXSetTevAlphaOp` (described later).

Code 9-2 GXSetTevOp

```
GXSetTevOp( GXTevStageID stage, GXTevMode mode );
```

Table 9-1 lists the `GXTevMode` types and the implied operation.

Table 9-1 GXTevMode Types

Name	Color Channel Op	Alpha Channel Op
GX_MODULATE	$C_v = C_r C_t$	$A_v = A_r A_t$
GX_DECAL	$C_v = (1 - A_t) C_r + A_t C_t$	$A_v = A_r$
GX_BLEND	$C_v = (1 - C_t) C_r + C_t$	$A_v = A_t A_r$
GX_REPLACE	$C_v = C_t$	$A_v = A_t$

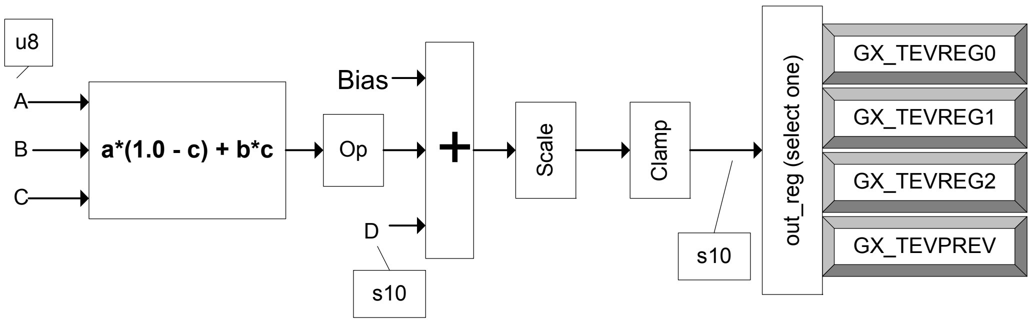
1. For stage 0, subscript *r* is the rasterized color (from a lighting channel) for this stage.
2. For other stages, subscript *r* is the previous stage output color.
3. Subscript *t* is the texture value for this stage.
4. Subscript *v* is the output color of this stage.

9.5 Color/alpha Combine Operations

As mentioned, `GXSetTevOp` is a simplifying function that sets the inputs and operation for a given TEV stage. In order to use the full power of the TEV, you must set all of these parameters independently. First we'll describe the operation that takes place within a TEV stage, followed by how to set the various inputs for the operation.

The figure below shows the operation that can be programmed in a given TEV stage. Remember that there are two operations happening in a given cycle: one for the color components, and one for the alpha component.

Figure 9-3 TEV Operations



Op = optional negate Scale = multiply by 1/2, 1, 2, or 4

Code 9-3 GXSetTevColorOp, GXSetTevAlphaOp

```
GXSetTevColorOp(
GXTevStageID stage,
GXTevOp      op,
GXTevBias    bias,
GXTevScale   scale,
GXBool       clamp_enable,
GXTevRegID   out_reg );
GXSetTevAlphaOp(
GXTevStageID stage,
GXTevOp      op,
GXTevBias    bias,
GXTevScale   scale,
GXBool       clamp_enable,
GXTevRegID   out_reg );
```

Each TEV stage has its own set of operation controls. The color and alpha operations are set independently using `GXSetTevColorOp` and `GXSetTevAlphaOp`.

The TEV operation begins by performing a linear interpolation between *A* and *B* inputs, using *C* as the interpolation factor. The inputs *A, B,* and *C* are always unsigned 8-bit values having a range [0 <= *A,B,C* <= 255]. The result of the interpolation can be optionally negated using *op*.

The input *D* and a *bias* value (0, +0.5, or –0.5) are then added to the result. The *D* input is a signed 10-bit number having a range [-1024 <= *D* <= 1023]. The result of each TEV stage can be a signed 10-bit value, so this input is provided for accumulating out-of-range values.

A constant *scale* (1, 2, 4, or 0.5) is then applied. The result is optionally clamped and written to an output register. (See section "9.5.1 Compare Mode" on page 95 for more details on clamping.)

For color operations, the same sequence is applied in parallel to the RGB color components. Alpha operations are independent of the color operations.

The output registers are available as inputs for the next TEV stage. For example, `GX_TEVREG0` corresponds to `GX_CC_C0` for the color TEV input, and `GX_CA_A0` corresponds to the alpha TEV input. The output registers can store a signed 10-bit number. The output register `GX_TEVPREV` is used by convention to pass the results of one TEV stage to the next. `GX_TEVPREV` *must* be the output register of the last active TEV stage.

Table 9-2 Correspondence Between TEV Input and Output Register Names

TEV Register Input Color Name	TEV Register Input Alpha Name	TEV Register Output Name
GX_CC_C0	GX_CA_A0	GX_TEVREG0
GX_CC_C1	GX_CA_A1	GX_TEVREG1
GX_CC_C2	GX_CA_A2	GX_TEVREG2
GX_CC_CPREV	GX_CA_APREV	GX_TEVPREV

9.5.1 Compare Mode

With GXTevOp, it is possible to use the comparison feature in addition to sign specification for addition and subtraction. When this feature is specified, the TEV output equation is modified as follows:

Equation 9-1 Regular TEV output

$$\text{output} = (d \pm ((1 - c) * a + c * b) + \text{bias}) * \text{scale}$$

Equation 9-2 Compare TEV output

$$\text{output} = d + ((a \text{ OP } b) \text{ ? } c{:}0)$$

You have a choice of 8-, 16-, or 24-bit wide compares, or an 8-bit per-component compare. When a compare operation is performed, the output scale factor can only be set to one, and the bias can only be set to zero.

The following tables describe the different operations. For the 16- and 24-bit compares, the MSB is listed first.

Table 9-3 Color or Alpha Compare Operations

Color or Alpha Compare	Operation
GX_TEV_COMP_R8_GT	a[red] > b[red] ?
GX_TEV_COMP_R8_EQ	a[red] == b[red] ?
GX_TEV_COMP_GR16_GT	a[green, red] > b[green, red] ?
GX_TEV_COMP_GR16_EQ	a[green, red] == b[green, red] ?
GX_TEV_COMP_BGR24_GT	a[blue, green, red] > b[blue, green, red] ?
GX_TEV_COMP_BGR24_EQ	a[blue, green, red] == b[blue, green, red] ?

Table 9-4 Color-only Compare Operations

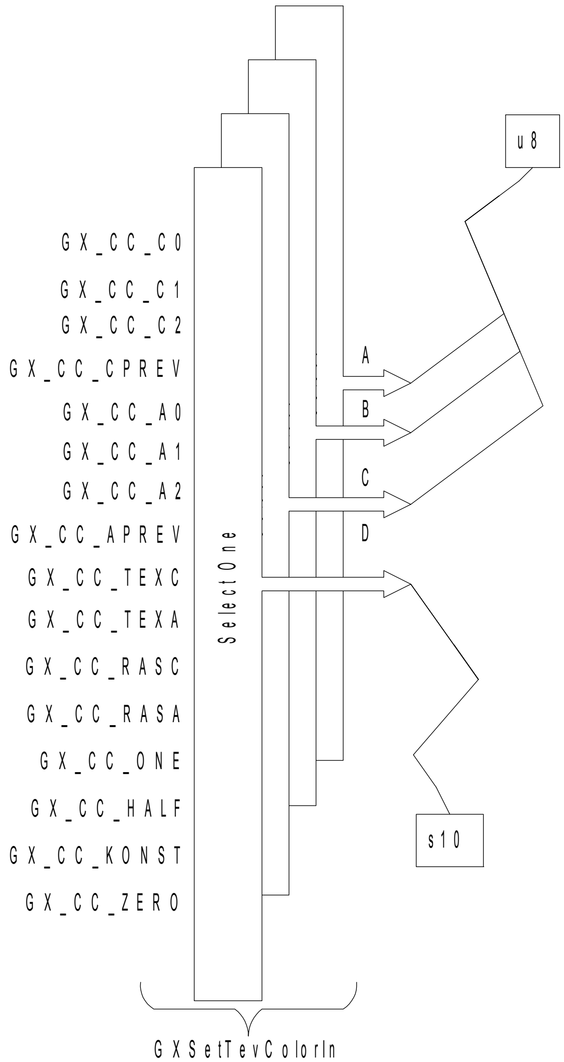
Color-only Compare	Operation
GX_TEV_COMP_RGB8_GT	per-component: a[component] > b[component] ?
GX_TEV_COMP_RGB8_EQ	per-component: a[component] == b[component] ?

Table 9-5 Alpha-only Compare Operations

Alpha-only Compare	Operation
GX_TEV_COMP_A8_GT	a[alpha] > b[alpha] ?
GX_TEV_COMP_A8_EQ	a[alpha] == b[alpha] ?

9.6 Color Inputs

Figure 9-4 TEV Stage Color Inputs



Code 9-4 GXSetTevColorIn

```
GXSetTevColorIn(
    GXTevStageID  stage,
    GXTevColorArg a,
    GXTevColorArg b,
    GXTevColorArg c,
    GXTevColorArg d );
```

The TEV allows for many color input sources including constant (register) colors and alphas, texture color/alpha, rasterized color/alpha (the result of per-vertex lighting), and a few useful constants. Notice that the input controls are independent for each TEV stage.

The register colors can be programmed directly as constant colors or used to store the results of TEV operations. For example, `GX_TEVREG0` corresponds to `GX_CC_C0` for the color TEV input. Unlike the rest of the TEV controls, the TEV registers are *shared* among all the TEV stages. The application must be careful to partition the use of registers as input colors and output colors when designing a color-combining equation. Code 9-5 sets register 1 to a constant 8-bit (per component) color and register 2 to a constant 10-bit (per component) color.

Code 9-5 Setting Constant Color

```
GXColor cyan = { 0x00, 0xff, 0xff, 0xff };
GXSetTevColor(GX_TEVREG1, cyan);

GXColorS10 coffset = { -128, -128, -128, 0 };
GXSetTevColorS10(GX_TEVREG2, coffset);
```

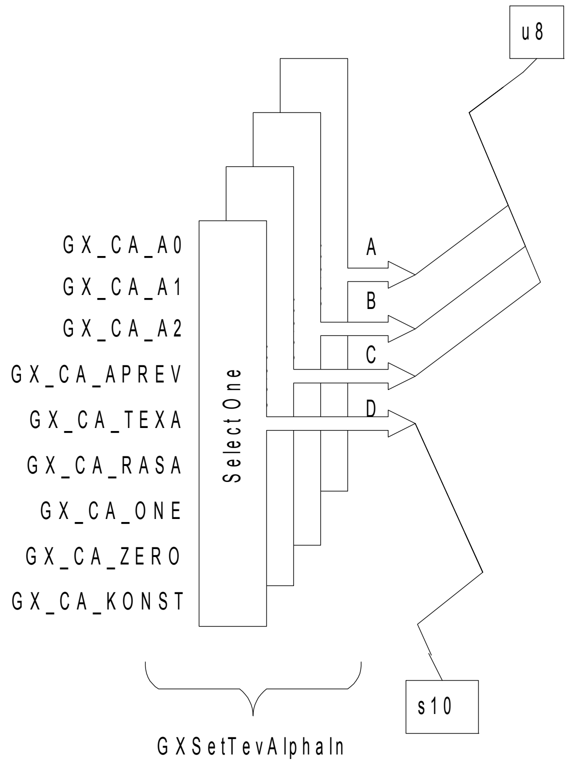
The default texture pipeline uses the `GX_TEVPREV` register to pass the output of one TEV stage to the input of the next TEV stage. This is only a convention of the default configuration assumed by `GXSetTevOp`. You can use `GX_TEVPREV` as a general-purpose register when programming your own TEV equations. However, at least one register is required to pass results from one stage to the next when multi-texturing. Note the `GX_TEVPREV` *must* be the output register for the last active TEV stage. This register is wired directly to the fogging and blending functions.

Note also that the inputs *A, B*, and *C* are unsigned 8-bit values, but the register inputs `GX_CC_C0-2` and `GX_CC_CPREV` can be signed 10-bit values [-1024...+1023]. When one of these registers is selected for the *A*, *B*, or *C* inputs, the *least* significant 8 bits of the register are used.

The rasterized color and alpha inputs are the result of per-vertex lighting on one of the lighting channels, either `GX_COLOR0A0` or `GX_COLOR1A1`. It is not possible to use the color from one lighting channel with the alpha from the other lighting channel (for example `GX_COLOR0` with `GX_ALPHA1`) into the *same* TEV stage. In the default texture pipeline, `GX_COLOR0A0` is directed to the first eight TEV stages. To change the default, you must call `GXSetTevOrder`, described later in this chapter.

9.7 Alpha Inputs

Figure 9-5 TEV Stage Alpha Inputs



Each TEV stage combines alpha channels independently of color channels. There are fewer alpha inputs than color inputs, and there are no color channels available in the alpha combiner.

Code 9-6 GXSetTevAlphaIn

```
void GXSetTevAlphaIn(
        GXTevStageID   stage,
        GXTevAlphaArg  a,
        GXTevAlphaArg  b,
        GXTevAlphaArg  c,
        GXTevAlphaArg  d );
```

The inputs *A*, *B*, and *C* are unsigned 8-bit values, but the register inputs `GX_CA_A0-2` and `GX_CA_APREV` can be signed 10-bit values [-1024...+1023]. When one of these registers is selected by the *A*, *B*, or *C* inputs, the *least* significant 8 bits of the register are used.

9.8 Color Component Swap Mode

The swap feature switches the color components of the texture color and rasterize color that are input to the TEV. This feature includes four tables in which the swap pattern is registered, and it can carry out the swap based on these tables. In various TEV stages, you can use the `GXSetTevSwapMode` function to select entries from the raster color input and texture color input tables. The swap tables are configured with the `GXSetTevSwapModeTable` function. Normally, the swap pattern uses only three of these four tables.

Code 9-7 GXSetTevSwapMode, GXSetTevSwapModeTable

```
GXSetTevSwapMode(
    GXTevStageID  stage,
    GXTevSwapSel  ras_sel,
    GXTevSwapSel  tex_sel );

GXSetTevSwapModeTable(
    GXTevSwapSel   select,
    GXTevColorChan red,
    GXTevColorChan green,
    GXTevColorChan blue,
    GXTevColorChan alpha );
```

9.9 TEV “constant” Color Registers

Four new registers are available as TEV inputs. They can be treated as RGBA registers or as a set of four scalar registers. These registers cannot be used as TEV outputs, and thus they are referred to as “constant” (or “konstant”) registers, although they can easily be modified by a GX set command. New actual constant values are provided in addition to the new register choices.

The new color inputs are selected through a two-level selection system. First, using `GXSetTevColorIn` (or `GXSetTevAlphaIn`), you select the `GX_CC_KONST` (or `GX_CA_KONST`). Then, using `GXSetTevKColorSel` (or `GXSetTevKAlphaSel`) you select the constant selection desired for each TEV stage.

Table 9-6 Color and Alpha Constant Register Values

Color KONST values	Alpha KONST values	Description
GX_TEV_KCSEL_1	GX_TEV_KASEL_1	1.0 constant
GX_TEV_KCSEL_7_8	GX_TEV_KASEL_7_8	7/8 constant
GX_TEV_KCSEL_3_4	GX_TEV_KASEL_3_4	3/4 constant
GX_TEV_KCSEL_5_8	GX_TEV_KASEL_5_8	5/8 constant
GX_TEV_KCSEL_1_2	GX_TEV_KASEL_1_2	1/2 constant
GX_TEV_KCSEL_3_8	GX_TEV_KASEL_3_8	3/8 constant
GX_TEV_KCSEL_1_4	GX_TEV_KASEL_1_4	1/4 constant
GX_TEV_KCSEL_1_8	GX_TEV_KASEL_1_8	1/8 constant
GX_TEV_KCSEL_K0	--	K0 [RGB] register
GX_TEV_KCSEL_K1	--	K1 [RGB] register
GX_TEV_KCSEL_K2	--	K2 [RGB] register
GX_TEV_KCSEL_K3	--	K3 [RGB] register
GX_TEV_KCSEL_K0_R	GX_TEV_KASEL_K0_R	K0 [RRR] register
GX_TEV_KCSEL_K1_R	GX_TEV_KASEL_K1_R	K1 [RRR] register
GX_TEV_KCSEL_K2_R	GX_TEV_KASEL_K2_R	K2 [RRR] register
GX_TEV_KCSEL_K3_R	GX_TEV_KASEL_K3_R	K3 [RRR] register

Table 9-6 Color and Alpha Constant Register Values (Continued)

Color KONST values	Alpha KONST values	Description
GX_TEV_KCSEL_K0_G	GX_TEV_KASEL_K0_G	K0 [GGG] register
GX_TEV_KCSEL_K1_G	GX_TEV_KASEL_K1_G	K1 [GGG] register
GX_TEV_KCSEL_K2_G	GX_TEV_KASEL_K2_G	K2 [GGG] register
GX_TEV_KCSEL_K3_G	GX_TEV_KASEL_K3_G	K3 [GGG] register
GX_TEV_KCSEL_K0_B	GX_TEV_KASEL_K0_B	K0 [BBB] register
GX_TEV_KCSEL_K1_B	GX_TEV_KASEL_K1_B	K1 [BBB] register
GX_TEV_KCSEL_K2_B	GX_TEV_KASEL_K2_B	K2 [BBB] register
GX_TEV_KCSEL_K3_B	GX_TEV_KASEL_K3_B	K3 [BBB] register
GX_TEV_KCSEL_K0_A	GX_TEV_KASEL_K0_A	K0 [AAA] register
GX_TEV_KCSEL_K1_A	GX_TEV_KASEL_K1_A	K1 [AAA] register
GX_TEV_KCSEL_K2_A	GX_TEV_KASEL_K2_A	K2 [AAA] register
GX_TEV_KCSEL_K3_A	GX_TEV_KASEL_K3_A	K3 [AAA] register

The command `GXSetTevKColor` may be used to modify the values of the four new constant color registers. `GXSetTevKColor` works the same way as `GXSetTevColor`. Unlike the existing TEV registers, the new registers have only 8 bits per component.

9.10 Example Settings

Equation 9-3 Pass Texture Color

$$C_v = C_t$$

Code 9-8 Pass Texture Color

```
GXSetTevColorIn(          GXSetTevColorOp(
     GX_TEVSTAGE0,               GX_TEVSTAGE0,
     GX_CC_ZERO,   // a          GX_TEV_ADD,    // op
     GX_CC_ZERO,   // b          GX_TB_ZERO,    // bias
     GX_CC_ZERO,   // c          GX_CS_SCALE_1, // scale
     GX_CC_TEXC ); // d          GX_ENABLE,     // clamp 0-255
                                 GX_TEVPREV);   // output reg
```

Equation 9-4 Modulate

$$C_v = C_t C_r$$

Code 9-9 Modulate

```
GXSetTevColorIn(          GXSetTevColorOp(
     GX_TEVSTAGE0,               GX_TEVSTAGE0,
     GX_CC_ZERO,   // a          GX_TEV_ADD,      // op
     GX_CC_TEXC,   // b          GX_TB_ZERO,      // bias
     GX_CC_RASC,   // c          GX_CS_SCALE_1,  // scale
     GX_CC_ZERO ); // d          GX_ENABLE,       // clamp 0-255
                                 GX_TEVPREV);     // output reg
```

Equation 9-5 Modulate 2X

$$C_v = 2\ C_t C_r$$

Code 9-10 Modulate 2X

```
GXSetTevColorIn(          GXSetTevColorOp(
     GX_TEVSTAGE0,               GX_TEVSTAGE0,
     GX_CC_ZERO,   // a          GX_TEV_ADD,     // op
     GX_CC_TEXC,   // b          GX_TB_ZERO,     // bias
     GX_CC_RASC,   // c          GX_CS_SCALE_2, // scale
     GX_CC_ZERO ); // d          GX_ENABLE,      // clamp 0-255
                                 GX_TEVPREV);    // output reg
```

Equation 9-6 Add

$$C_v = C_t + C_r$$

Code 9-11 Add

```
GXSetTevColorIn(          GXSetTevColorOp(
     GX_TEVSTAGE0,               GX_TEVSTAGE0,
     GX_CC_TEXC,   // a          GX_TEV_ADD,     // op
     GX_CC_ZERO,   // b          GX_TB_ZERO,     // bias
     GX_CC_ZERO,   // c          GX_CS_SCALE_1, // scale
     GX_CC_RASC ); // d          GX_ENABLE,      // clamp 0-255
                                 GX_TEVPREV);    // output reg
```

Equation 9-7 Subtract

$$C_v = C_t - C_r$$

Code 9-12 Subtract

```
GXSetTevColorIn(          GXSetTevColorOp(
     GX_TEVSTAGE0,               GX_TEVSTAGE0,
     GX_CC_RASC,   // a          GX_TEV_SUB,     // op
     GX_CC_ZERO,   // b          GX_TB_ZERO,     // bias
     GX_CC_ZERO,   // c          GX_CS_SCALE_1, // scale
     GX_CC_TEXC ); // d          GX_ENABLE,      // clamp 0-255
                                 GX_TEVPREV);    // output reg
```

Equation 9-8 Blend

$$C_v = C_t(1.0 - A_t) + C_r A_t$$

Code 9-13 Blend

```
GXSetTevColorIn(          GXSetTevColorOp(
     GX_TEVSTAGE0,               GX_TEVSTAGE0,
     GX_CC_TEXC,   // a          GX_TEV_ADD,    // op
     GX_CC_RASC,   // b          GX_TB_ZERO,    // bias
     GX_CC_TEXA,   // c          GX_CS_SCALE_1, // scale
     GX_CC_ZERO ); // d          GX_ENABLE,     // clamp 0-255
                                 GX_TEVPREV);   // output reg
```

9.11 Alpha Compare Function

You can apply the alpha compare operation to the alpha output from the last active TEV stage.

Code 9-14 GXSetAlphaCompare

```
GXSetAlphaCompare(
    GXCompare   comp0,
    u8          ref0,
    GXAlphaOp   op,
    GXCompare   comp1,
    u8          ref1 );
```

The alpha compare operation actually consists of two separate compares, the results of which can be combined using several logical operations.

Equation 9-9 Alpha Compare

$$(A_{source} \quad comp0 \quad A_{ref0}) \quad op \quad (A_{source} \quad comp1 \quad A_{ref1})$$

For example, the following compare is possible:

Equation 9-10 Sample Alpha Compare

$$(A_{source} \geq A_{ref0}) \quad AND \quad (A_{source} \leq A_{ref1})$$

The result of the alpha compare is a Boolean condition, *true* or *false*. The result of the alpha compare is used to conditionally write the pixel color (and possibly Z) to the frame buffer.

The following compare operations are possible: *never*, *less*, *less or equal*, *equal*, *not equal*, *greater*, *greater or equal*, *always*. The following combine operations are possible: *and*, *or*, *exclusive-or*, *exclusive-nor*.

The effect of the alpha compare on the Z buffer depends on whether Z buffering occurs before or after texture lookup (see `GXSetZCompLoc`). If Z buffering occurs *before* texture lookup, then the Z write condition is determined only by the Z compare. If Z buffering occurs *after* texture lookup, then the alpha compare result and the Z compare result are logically ANDed to determine whether the color and Z are written to the frame buffer. In general, if alpha compare is enabled, Z buffering should occur *after* texture lookup.

9.12 Z Textures

Z textures are always looked up in the last TEV stage. The texture input of the last active TEV stage is connected directly to the Z buffer logic; therefore, it is not possible to apply TEV operations to the Z texture. The color is still output from the last TEV stage when Z textures are enabled, but the texture input of the last stage is occupied by the Z texture so it cannot be used as a color source. However, all other color inputs and all TEV operations of the last stage can be used. The alpha side of the TEV stage is not affected by Z textures.

Refer to "8 Texture Mapping" on page 57 for information on using Z textures. Refer to *Advanced Rendering* for information on applications of Z textures.

9.13 Texture Pipeline Configuration

Each TEV stage requires:

- A texture map (`GXTexMapID`)
- A texture coordinate (`GXTexCoordID`) to use for the texture map
- A color channel (`GX_COLOR0A0 or GX_COLOR1A1`) to rasterize
- Modes and controls for the TEV stage itself

The first three items are supplied using the function in Code 9-15.

Code 9-15 GXSetTevOrder

```
GXSetTevOrder(
    GXTevStageID    stage,
    GXTexCoordID    coord,
    GXTexMapID      map,
    GXChannelID     color );
```

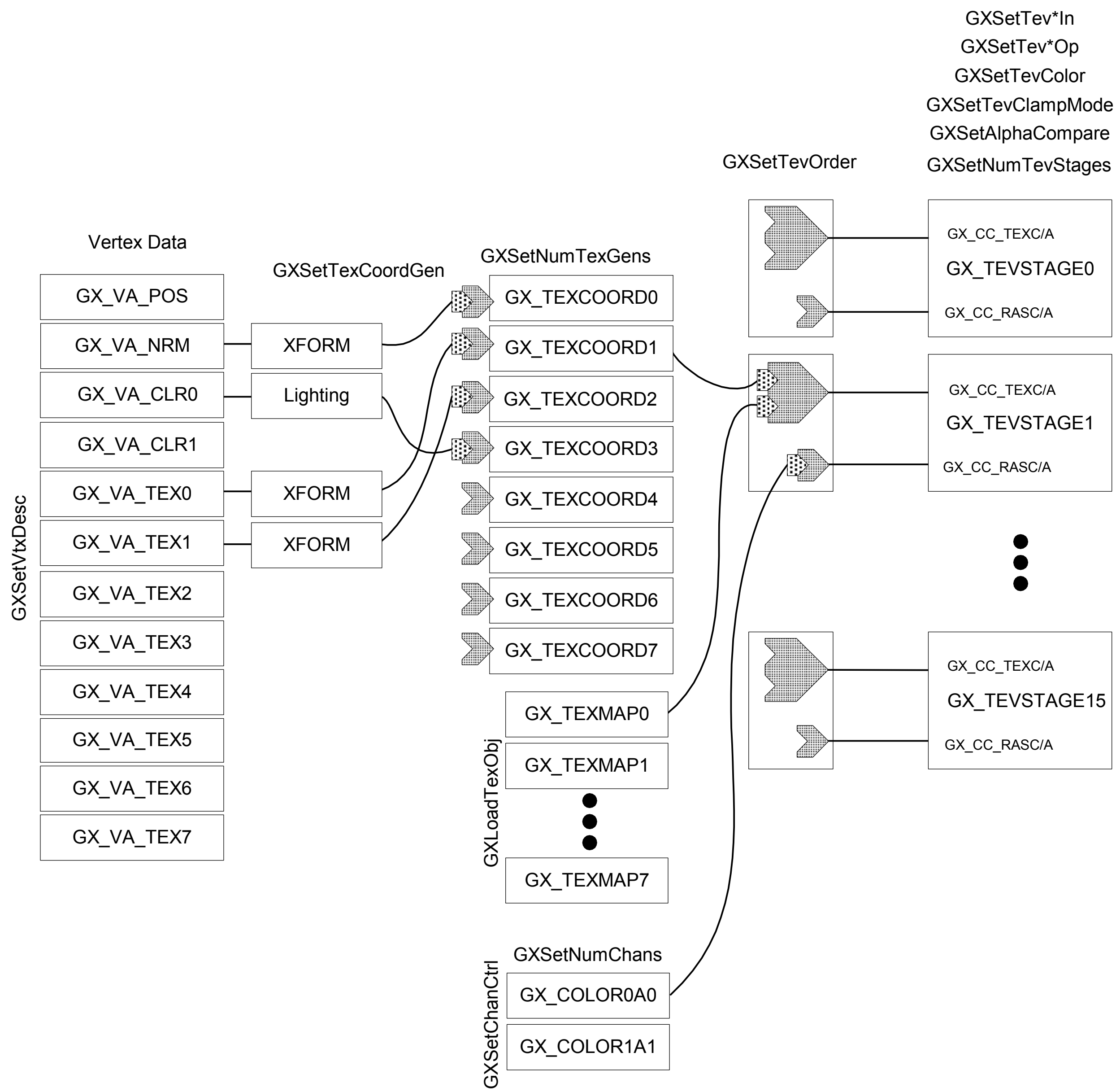
Let's take each parameter in turn. The *stage* parameter is the TEV stage that you are configuring. The *coord* parameter is the name of the texture coordinate used to look up the texture. The *coord* is generated according to the function `GXSetTexCoordGen`. The *map* parameter is the name of the texture to use. If no *coord* or *map* is to be used in this stage, they should be set to `GX_TEXCOORD_NULL` and `GX_TEXMAP_NULL`, respectively. This *map* ID is associated with a texture using the `GXLoadTexObj` function. The *color* parameter is used to name the output color channel of the vertex lighting hardware. The GP can only rasterize one color channel per clock, so you must choose whether to rasterize `GX_COLOR0A0` or `GX_COLOR1A1` for this TEV stage.

You can think of the `GXSetTevOrder` function as a switchboard with which you can connect all the interested parties together. Similarly, the function `GXSetTexCoordGen` can be seen as a switchboard that connects and transforms input vertex data to texture coordinates.

While only eight texmaps and eight texcoords may be specified, up to 16 different texture lookups can be performed. However, since texture coordinates are scaled based on the size of the associated texture map, you can only re-use a texture coordinate to texture maps that have the same size. Each TEV stage will perform its own texture lookup based upon all of its settings.

The following diagram shows the relationships between different parts of the texture pipeline and the functions that control them and their associations.

Figure 9-6 Texture Pipeline Control



`GXInit` sets the default texture coordinate generation capability. This function configures texture coordinate generation to copy each input texture coordinate directly to an output texture coordinate using an identity matrix. `GXInit` sets the number of color channels to 0 (`GXSetNumChans`).

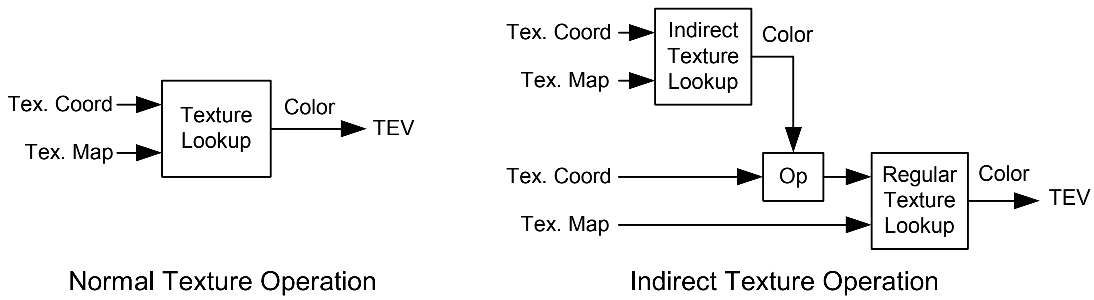
Once you understand the default configuration, you can choose to override the texture coordinate generation using `GXSetTexCoordGen`, or the TEV wiring using `GXSetTevOrder`, or both.

`GXSetTevOrder` determines which texture and rasterized color inputs are available to each TEV stage, while `GXSetTevColorIn` and `GXSetTevAlphaIn` determine how the various inputs plug into the TEV operation for each stage. Also, `GXSetTevColorOp`/`GXSetTevAlphaOp` determine how the stages connect together.

10 Indirect Texture Mapping

The Graphics Processor has a powerful indirect texture feature. It allows the colors looked up from one (indirect) texture lookup to be transformed into offsets that are then added to the texture coordinates for another (regular) texture lookup. Various operations are possible when combining the output of the indirect stage with the coordinates for the regular stage.

Figure 10-1 Indirect Texture Operation



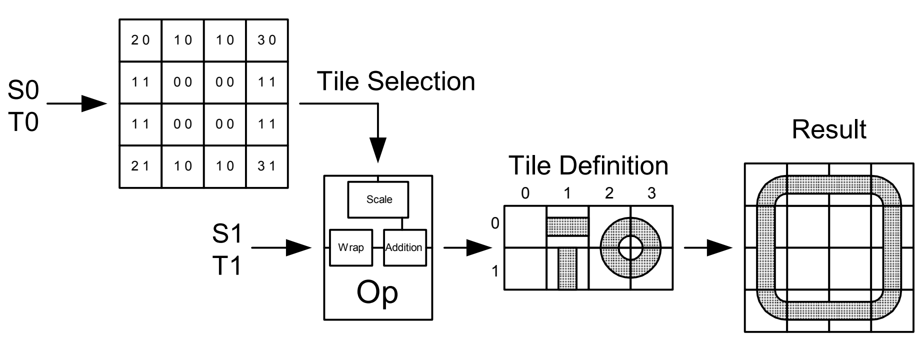
Indirect textures have several possible applications:

- Texture warping
- Texture tile maps
- Pseudo-3D textures
- Environment-mapped bump mapping

Using indirect textures for texture warping effects is the simplest application of the indirect feature. In this case, the indirect texture is used to stretch or otherwise distort the surface texture. You can achieve a dynamic distortion effect by swapping indirect maps (or by modifying the indirect map or coordinates). You may apply this effect to a given surface within a scene, or you can take this one step further and apply the effect to the entire scene. In the latter case, the scene is first rendered normally and then copied to a texture map. You then draw a big rectangle that can be mapped to the screen using an indirect texture. Texture warping allows for shimmering effects, special lens effects, and various psychedelic effects.

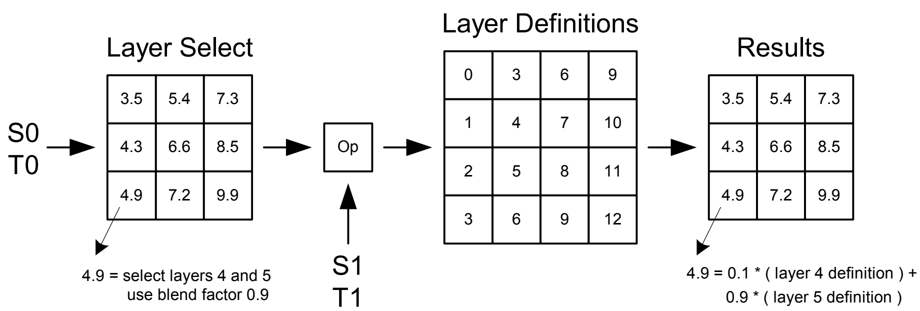
The indirect feature also lets you draw texture tile maps. In this case, one texture map holds the base definition for a variety of tiles. An indirect texture map is then used to place specific tiles in specific locations over a 2D surface. Normally, this effect is accomplished without indirect textures by drawing a polygon for each desired tile. With indirect textures, only one polygon need be drawn.

Figure 10-2 Tiled Texture Mapping



Tile mapping can be extended to become a pseudo-3D texture effect. Consider all the tiles to be part of a stack. Rather than drawing just a single tile layer from the stack, you can select two adjacent layers and blend between them. You can use this technique to cover a large surface with non-repeating patterns that blend together smoothly. You might imagine covering a beach with such a texture, where the layers vary in appearance from fine sand to small pebbles to larger rocks.

Figure 10-3 Pseudo-3D Textures



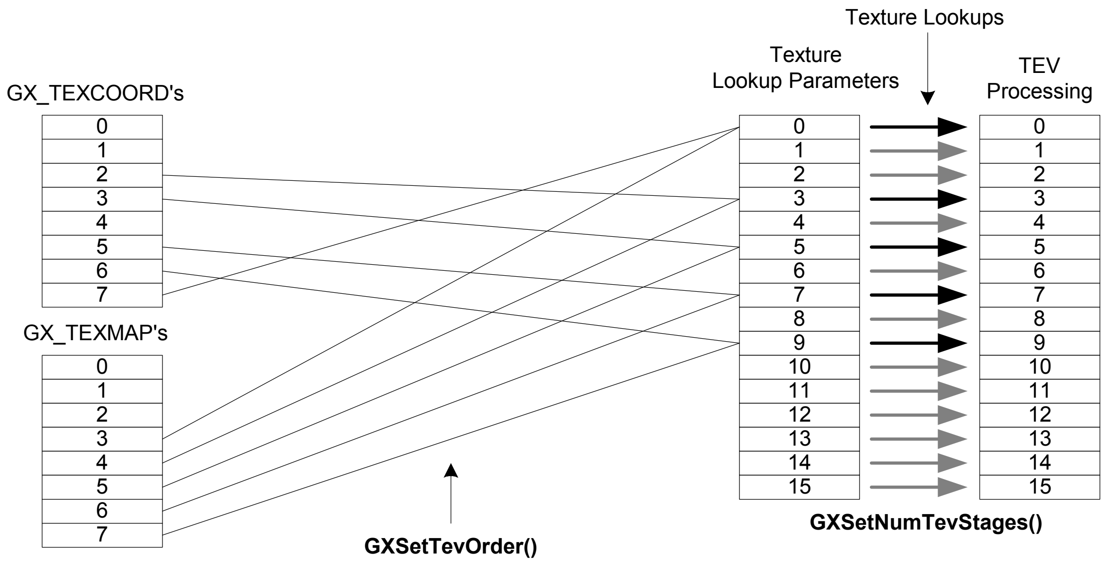
Environment-mapped bump mapping (EMBM) is another use of indirect textures. Regular emboss bump mapping considers only the interaction between the bump map and a single diffuse light source. With environment-mapped bump mapping, you use an indirect bump texture to offset texture coordinates generated via surface normals. The perturbed normals are then used to look up an environment texture. The environment texture may contain complicated lighting effects, or it may be a reflection map of the environment. (One caveat is that the environment map is viewpoint-dependent and must be regenerated whenever the viewpoint changes.) There are two EMBM methods:

1. The normal perturbations are modeled with respect to a flat surface (*dS*, *dT*), and during runtime they are transformed onto arbitrary surfaces. This method requires three TEV stages plus one indirect stage to calculate the modified texture coordinates.
2. The normal perturbations are modeled in 3D (*dX*, *dY*, *dZ*), and during runtime they are matched with specific surfaces and transformed into eye space. This method requires only one TEV stage plus one indirect stage to compute.

10.1 Setting Up Indirect Texture Stages

Figure 10-4 illustrates the texture hardware as described in the previous chapters. For each TEV stage, you may assign a texcoord and a texture map to be looked up. Texture lookup and processing occurs in each TEV stage.

Figure 10-4 Regular Texture Functional Diagram

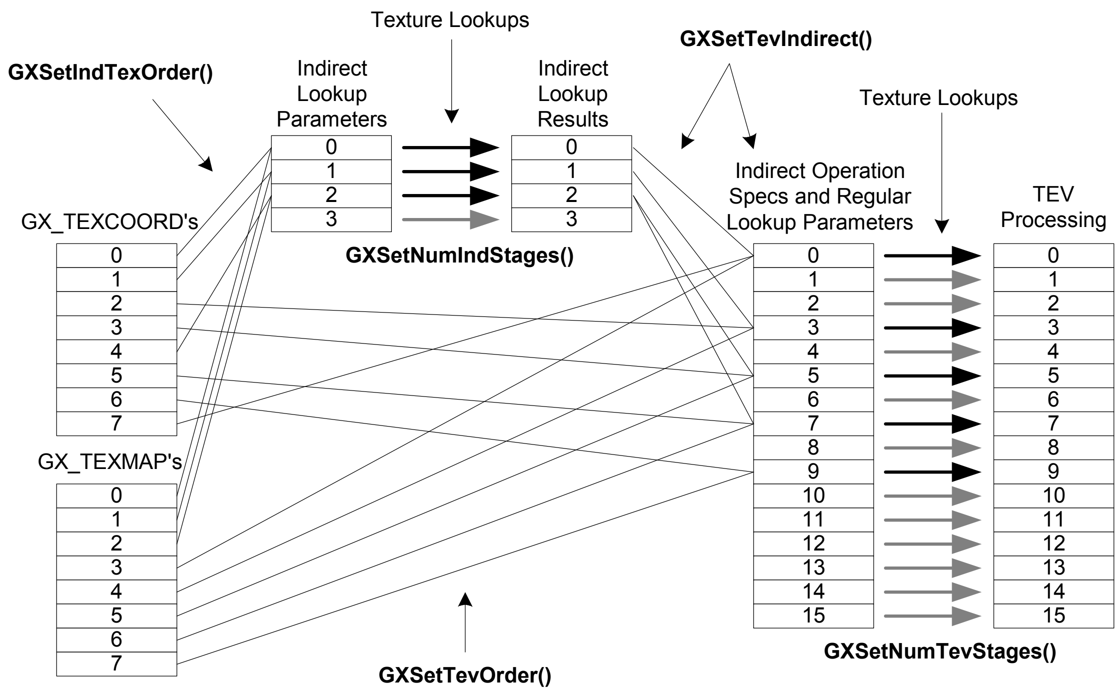


We now describe the indirect texture hardware in more detail. As mentioned above, an indirect lookup consists of an indirect lookup stage, after which the results are combined with the regular texture coordinates for another (regular) lookup stage. There can be up to four different indirect lookup stages. The results from any indirect lookup may be used in any TEV stage to modify a regular texture lookup. Thus there can be up to 16 modified regular lookups. The total number of texture maps available remains at 8. In addition, any texture map that is designated as being used for an indirect lookup cannot also be used as a target for a regular lookup (and vice versa—a given texture map can be only one or the other).

Performance-wise, adding an indirect stage is like adding another TEV stage. It increases the time required to process each quad (2x2 pixels) by an additional clock cycle.

In the figure below, we add in the indirect hardware as it has been described thus far.

Figure 10-5 Regular and Indirect Texture Functional Diagram



The figure also shows some of the relevant functions. First is `GXSetNumIndStages`, which controls how many indirect texture lookups there will be. For each indirect stage, you must specify a texture coordinate and map. This is done using `GXSetIndTexOrder`:

Code 10-1 GXSetIndTexOrder

```
void GXSetIndTexOrder(GXIndTexStageID stage,
                      GXTexCoordID    coord,
                      GXTexMapID      map);
```

As mentioned, a map that is assigned to an indirect stage in this way cannot also be assigned to a regular TEV stage as well. A given map can only be indirect or regular.

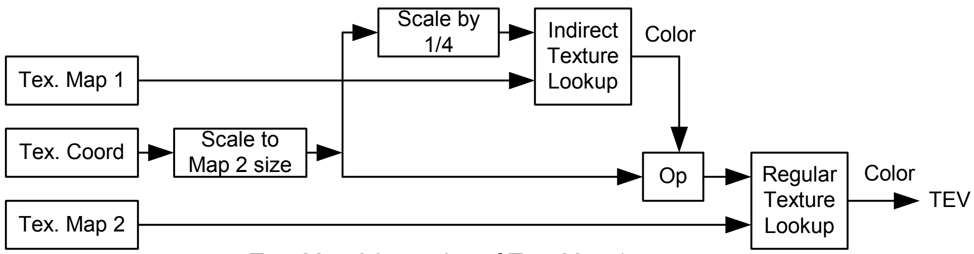
On the other hand, a given texture coordinate *can* be shared by both an indirect lookup and a regular lookup. If the indirect map is the same size as the regular map, then the sharing is straightforward. If the sizes differ, there is still the possibility to share. If the regular map is larger than the indirect map, you may scale it down by a power of two for use with the indirect lookup while still using the unscaled texture coordinate for the regular lookup. This is set by the call in Code 10-2.

Code 10-2 GXSetIndTexCoordScale

```
void GXSetIndTexCoordScale(GXIndTexStageID             indStage,
                           GXIndTexScale               scaleS,
                           GXIndTexScale               scaleT);
```

The figure below shows an example for this case. The block labeled “Scale to Map 2 size” is the regular texture coordinate scaling that the hardware does. The block labeled “Scale by 1/4” would happen as a result of calling `GXSetIndTexCoordScale` with parameters specifying 1/4 scaling.

Figure 10-6 Texture Coordinate Sharing Example



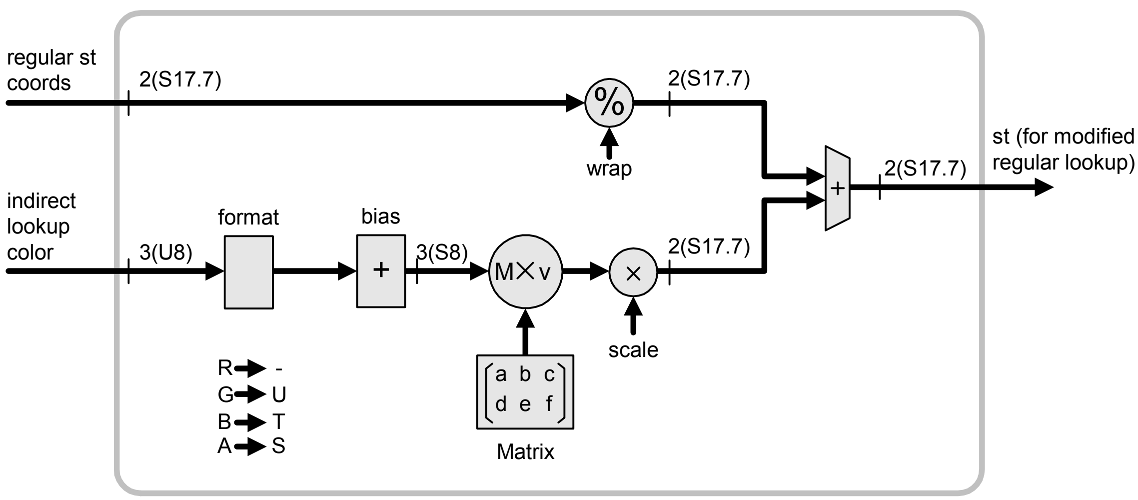
Tex. Map 2 is 4x size of Tex. Map 1

If we share texture coordinates for this example and set the indirect scaling to 1, then the texture coordinate would either wrap or clamp as it accesses the indirect texture. The behavior depends upon the settings for the indirect map being looked up.

10.2 Basic Indirect Texture Processing

This section describes what is possible inside the “Op” block shown above, where the colors coming from the indirect lookup are processed and combined with the regular texture coordinate. The figure below shows some of the major components of the indirect texture processing block. More details will be shown in a later section.

Figure 10-7 Indirect Texture Processing, Part 1



When the indirect looked-up color enters the processing box, the first thing that happens is a mapping of the color components to the (*s*, *t*, *u*) texture offsets. As shown, red (R) is discarded, green (G) maps to *u*, blue (B) maps to *t*, and alpha (A) maps to *s*. Some remapping may be done later through the use of the indirect matrix (see section "10.3 Basic Indirect Texture Functions" on page 112). We chose this initial mapping in order to make efficient use of IA8 texture maps.

Next, the texture offsets go through the format block. You may choose whether some or all of the bits of each component are actually used; that is, specifically, you may choose whether the lower 3, 4, 5, or 8 bits of each component are used. Normally, you will use all 8. (The reason for the partial choices is described in "10.5.1 Texture Tiling and Pseudo-3D Texturing" on page 115.)

Next, you can apply a bias. If 8 bits were chosen in the format block, then the bias is -128, allowing for signed offsets. If a smaller number of bits was chosen in the format block, then the bias is +1.

The next operation is a matrix-vector multiply. You may configure up to three different static matrices and choose which one to use for a given indirect texture operation. The components of the matrix are in the range [-1 … +0.999] (a sign bit plus 10 bits of mantissa). As shown, the matrix has two rows and three columns, and it appears on the left side of the multiply. On the right side is a column vector consisting of the (*s*, *t*, *u*) offsets. The matrix may be used for rotation, scaling, and remapping of the offsets.

A scale value is associated with each static matrix. You may choose a scale value by specifying an exponent of 2 in the range of [-17 … +46]. This scale value can be used to stretch the offsets over the size of the regular texture map that will be associated with this indirect operation. You can also think of the scale as the fixed-point exponent for the matrix values. Since a value equal to +1.0 cannot be stored in the matrix, you can store 0.5 and use an exponent of 1 greater than the desired value in order to compensate.

You can optionally wrap the regular texture coordinate used with this lookup operation. You may specify a wrap value of 0, 16, 32, 64, 128, 256, or none. Using 0 effectively zeroes out the regular texture coordinate values.

Once all of the above processing has taken place, the offsets are added to the regular texture coordinates. This becomes the effective texture coordinate that will be used in the texture lookup for the associated TEV stage.

10.3 Basic Indirect Texture Functions

We now describe more indirect texture functions. Use the function in Code 10-3 to set an indirect matrix and scale value:

Code 10-3 GXSetIndTexMtx

```
void GXSetIndTexMtx( GXIndTexMtxID mtx_id,
                     f32 offset[2][3],
                     s8 scale_exp);
```

As mentioned above, there are three indirect matrices, and *mtx_id* specifies which one to set. The *offset* values must be in the range of [-1 … +0.999] (sign plus 10-bit mantissa), and *scale_exp* must be in the range of [-17 … +46].

The following group of functions will set up the indirect hardware to enable a particular special effect. These functions only set the indirect hardware, and you must still set up the TEV, texture maps, texgens, and other parts of the system in order to achieve the desired effect.

10.3.1 Texture Warping

Code 10-4 GXSetTevIndWarp

```
GXSetTevIndWarp(
        GXTevStageID        tevStage,        // Name of TEV stage being modified
        GXIndTexStageID     indStage,        // The indirect stage that specifies the warp map
        GXBool              signedOffsets,   // True for s8 offsets, false for u8 offsets
        GXBool              replaceMode,     // True to replace, false to modify regular coords
        GXIndTexMtxID       matrixSel);      // Which indirect matrix and scale to use
```

This function provides the ability to warp a regular texture lookup. The indirect map should contain 8-bit offsets. The *signedOffsets* parameter controls whether or not a bias of -128 is applied to the offsets. The *replaceMode* parameter controls whether the offsets completely replace the regular texture coordinates (`GX_TRUE`) or if they merely offset them (`GX_FALSE`). In effect, this selects a zero wrap value or a wrap value of none. The *matrixSel* parameter chooses which of the available indirect matrices (and associated scale values) to use.

The SDK graphics demo `ind-warp` shows one way to use this function.

10.3.2 Environment-mapped Bump-mapping (EMBM) (*dX*, *dY*, *dZ*)

Code 10-5 GXSetTevIndBumpXYZ

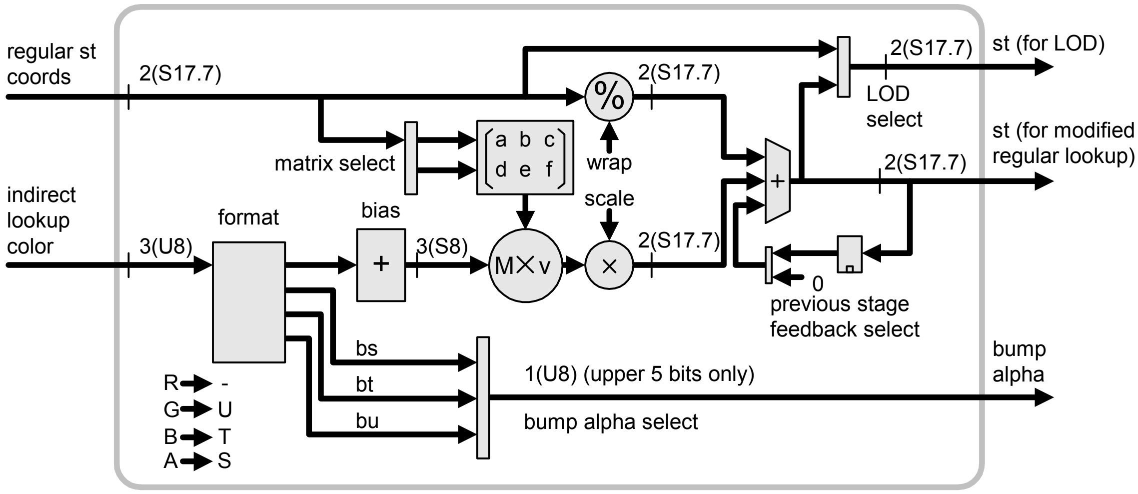
```
GXSetTevIndBumpXYZ(
       GXTevStageID            tevStage,    // Name of TEV stage being modified
       GXIndTexStageID         indStage,    // The indirect stage that specifies the bump map
       GXIndTexMtxID           matrixSel);  // Which matrix/scale slot to use
```

`GXSetTevIndBumpXYZ` sets up an environment-mapped bump-mapped (*dX*, *dY*, *dZ*) texture lookup. This is basically a perturbed lookup into a spherical reflection map. As an indirect operation, this is just a warp with signed offsets. The bump map must contain biased normal offsets in 3D model space. It must be an RGBA texture with *dX* in A, *dY* in B, and *dZ* in G. You must load the indirect matrix with a transform for normals that goes from model space to eye space. The scale value must contain the size of the reflection map divided by 2 (and thus the reflection map must be a square power of 2 size). You must set up a normal-based texgen for the regular texture coordinate. Alternately, you may avoid sending vertex normals by putting normals (not just offsets) in the texture map. See *Advanced Rendering* for more details on bump mapping. Also, refer to the demo `ind-bump-xyz`.

10.4 Advanced Indirect Texture Processing

Before moving on to the next indirect functions, let's examine more details in the indirect texture processing block. These details are illustrated in Figure 10-8.

Figure 10-8 Indirect Texture Processing, Part 2



The figure above shows four additional features of the indirect processing hardware:

- You may select a "bump alpha" value to be extracted from the indirect lookup color.
- A dynamic matrix may be created, based upon the incoming regular texture coordinates.
- You may select whether the original or modified texture coordinate is used to compute texture LOD.
- You may feed the texture coordinate calculated from one stage as an additive input to the next.

10.4.1 Selecting "bump alpha"

You may specify that certain bits from the indirect looked-up color be available as "bump alpha." Which bits are used depends upon the format selected for the indirect offsets. If you chose to use 3, 4, or 5 bits from the colors as offsets, then the remaining 5, 4, or 3 bits are available for selection as bump alpha. If you chose the 8-bit format, then the upper 5 bits may be duplicated as bump alpha. In any case, you choose whether to extract the bits from the *s*, *t*, or *u* component; the resulting bits are left-aligned in the bump-alpha 8-bit field. Bump alpha is available as a TEV-stage color input (by using `GXSetTevOrder`). You may select the plain bump alpha, or a "normalized" bump alpha (a bump alpha multiplied by a factor of 255/248). There is a restriction on this feature: bump alpha is not available for TEV stage 0.

10.4.2 Dynamic Matrices

There are more choices for the indirect matrix. You may select from the three static matrices, two types of dynamic matrices, or a matrix of all zeros. For each matrix selection (except the zero matrix), there is a corresponding scale selection as well. If you use a dynamic matrix, you may choose one of the three scale values associated with the static matrices. Dynamic matrices are set based upon the incoming regular s/t coordinate values:

Equation 10-1 Dynamic Indirect Matrices

$$\text{Type "}s\text{" matrix}: \begin{Bmatrix} s/256 & 0 & 0 \\ t/256 & 0 & 0 \end{Bmatrix} \qquad \text{Type "}t\text{" matrix}: \begin{Bmatrix} 0 & s/256 & 0 \\ 0 & t/256 & 0 \end{Bmatrix}$$

10.4.3 Selecting Texture Coordinates for Texture LOD

You may select to use the original (unmodified) or the modified texture coordinates for the MIPMAP LOD computation. When doing texture tiling (see section 10.5.1), you should use the unmodified texture coordinates for this purpose. In most other cases, you should use the modified coordinates.

10.4.4 Adding Texture Coordinates from Previous TEV Stages

You may choose to add in the texture coordinate computed in the previous TEV stage. This allows even more complicated expressions to be built up over multiple stages. It also allows a complicated result to be reused for more than one lookup.

10.5 Advanced Indirect Functions

We now describe functions that take advantage of the features mentioned in "10.4.4 Adding Texture Coordinates from Previous TEV Stages" on page 115.

10.5.1 Texture Tiling and Pseudo-3D Texturing

Code 10-6 GXSetTevIndTile

```
GXSetTevIndTile(
        GXTevStageID        tevStage,        // Name of TEV stage being modified
        GXIndTexStageID     indStage,        // The indirect stage that specifies the tile map
        u16                 tileSizeS,       // Size of tile in S dimension
        u16                 tileSizeT,       // Size of tile in T dimension
        u16                 tileSpacingS,    // Tile spacing in S dimension
        u16                 tileSpacingT,    // Tile spacing in T dimension
        GXIndTexFormat      format,          // Format of indirect offsets
        GXIndTexMtxID       matrixSel,       // Which indirect matrix slot to use
        GXIndTexBiasSel     biasSel,         // For pseudo-3D, selects tile-stacking direction
        GXIndTexAlphaSel    alphaSel);       // For pseudo-3D, selects bump alpha
```

This function specifies texture tiling or pseudo-3D texture lookup. You specify the tile size and spacing separately. Using a spacing different than the tile size allows borders for mipmapping purposes. Depending upon the height of the mipmap stack, texels outside of the tile area may be included in the filtering calculations for mipmapping. This function will set up the matrix values and scale value appropriately based upon the given inputs; you need only to specify which matrix slot to use. The *biasSel* and *alphaSel* are used only for pseudo-3D lookups (see section 10.5.3). You set these to `GX_ITB_NONE` and `GX_ITBA_OFF`, respectively, for normal 2D tiling.

Texture tiling can take advantage of the same texture coordinate for use with the indirect map and the regular map. There is a complication, however, since the desired scale values for the regular texture coordinates are not directly related to the size of the regular map, which contains the tile definitions (refer back to "Figure 10-2 Tiled Texture Mapping" on page 107). Normally, GX will set the texture coordinate's scale size to the size of the map being looked up, with preference for the regular map size if a texture coordinate is being shared. Since you need to use a different scale altogether with texture tiling, you must use the function in Code 10-7.

Code 10-7 GXSetTexCoordScaleManually

```
GXSetTexCoordScaleManually(
        GXTexCoordID        texCoord,        // Name of the texcoord being affected
        GXBool              enable,          // GX_TRUE = manual scaling; GX_FALSE = automatic
        u16                 ss,              // Manual scale value for S dimension
        u16                 ts );            // Manual scale value for T dimension
```

Once GXSetTexCoordScaleManually has been called with *enable* set to GX_TRUE, the given texture coordinate scale values are fixed until the function is called again. If the function is called with *enable* set to GX_FALSE, then automatic texture coordinate scaling takes over once again for that texcoord. When you are manually scaling, you should also call GXSetTexCoordBias. The bias is normally set automatically by the GX API, but when a texture coordinate is being scaled manually, the bias is no longer modified by GX and will be stale from the last time it was set.

For texture tiling, the desired texture coordinate scale is the tile size multiplied by the size of the indirect map. You then use GXSetIndTexCoordScale to divide out the tile size for use in accessing the indirect map. For an example of texture tiling, refer to the demo ind-tile-test.

In order to support pseudo-3D texture lookup, you must call GXSetTevIndTile for two adjacent TEV stages. The first stage resembles a normal 2D tiling specification. For the second stage, you specify a bias select and alpha select. The bias is used to select the tile-stacking direction. You use GX_ITB_S when the next tile is offset in *s*, and GX_ITB_T when the next tile is offset in *t*. You then choose a bump alpha in order to blend between the tile from the first lookup and the tile from the second lookup. You cannot use the 8-bit format for pseudo-3D. Instead, you must use the 3-, 4-, or 5-bit format. These formats use a bias value of +1 instead of -128. The +1 bias is used to get the “next” tile in the second stage. Refer to the demo ind-pseudo-3d to see one way to use this feature.

10.5.2 Environment-mapped Bump-mapping (EMBM) (*dS*, *dT*)

Code 10-8 GXSetTevIndBumpST

```
GXSetTevIndBumpST(
        GXTevStageID            tevStage,       // Name of first TEV stage to insert EMBM lookup
        GXIndTexStageID         indStage,       // The indirect stage that specifies the bump map
        GXIndTexMtxID           matrixSel);     // Which scale/matrix slot to use
```

This function sets up an environment-mapped, bump-mapped (*dS*, *dT*) texture lookup. Similar to GXSetTevIndBumpXYZ, this sets up a perturbed lookup into a spherical reflection map. The difference is that the bump map in this case contains deltas for (*s*, *t*). Such a lookup requires three TEV stages to compute the offset texture coordinates. The resulting texture coordinate will be available two stages after the one specified in the call. This function makes use of the dynamic matrices in order to transform the (*s*, *t*) offsets to be relative to the incoming regular (*s*, *t*) (which come from the object normals).

You must set up the desired offset scale value using GXSetIndTexMtx. The scale value must contain the size of the reflection map divided by 2 (and thus the reflection map must be a square power of 2 in size). No static matrix is actually used in the texture coordinate computation. Only dynamic matrices and the scale value are used.

The geometry associated with this lookup must include normals, binormals, and tangents. You must set up three normal-based texgens for the regular texture coordinates. The binormal texgen goes to the first TEV stage, the tangent texgen goes to the second stage, and the normal texgen goes to the third stage. An additional texgen is used for the indirect coordinate. You should use an IA8 texture format for the bump map, with *s* offsets in alpha (A) and *t* offsets in intensity (I). You must set up the first two TEV stages used so that they do not actually look up the textures (by using GX_TEX_DISABLE).

Refer to the demo ind-bump-st to see this kind of environment-mapped bump-mapping in action. Also refer to *Advanced Rendering* for more details on bump mapping.

Having used three TEV stages to compute the texture coordinate for an EMBM (*dS*, *dT*) lookup, you can use the result to do more than one lookup. In order to perform successive lookups without taking three stages to compute each one, use the texture coordinate feedback feature. You may call `GXSetTevIndRepeat` to set this up, as shown in Code 10-9.

Code 10-9 GXSetTevIndRepeat

```
GXSetTevIndRepeat(
        GXTevStageID            tevStage);      // Name of TEV stage being modified
```

This function allows you to use the texture coordinates computed in the previous TEV stage for the named TEV stage. It is typically used only after `GXSetTevIndBumpST`.

10.5.3 General Indirect Texturing

The functions described so far implement the most obvious uses of the indirect texture hardware. There is one more function available to set the indirect texture hardware directly, in case developers think of additional uses for the hardware.

Code 10-10 GXSetTevIndirect

```
GXSetTevIndirect(
        GXTevStageID        tevStage;        // TEV stage name
        GXIndTexStageID     indStage;        // the indirect stage to be used with this TEV stage
        GXIndTexFormat      format;          // format of the indirect texture offsets
        GXIndTexBiasSel     biasSel;         // Selects which offsets (S, T) receive a bias
        GXIndTexMtxID       matrixSel;       // Selects which indirect matrix and scale to use
        GXIndTexWrap        wrapS;           // Wrap value of regular S coordinate
        GXIndTexWrap        wrapT;           // Wrap value of regular T coordinate
        GXBool              addPrev;         // Add output from previous stage to texture coords?
        GXBool              utcLOD;          // Use unmodified texture coordinates for LOD calc?
        GXIndTexAlphaSel    alphaSel);       // Selects indirect texture alpha output
```

`GXSetTevIndirect` allows you to set the various parameters for a given indirect operation manually. The indirect texture coordinate processing block is used for all TEV stages. If you do *not* want to perform a modified lookup, select the zero texture matrix and turn off wrapping, feedback, and bump alpha. We provide a convenience function to do this:

Code 10-11 GXSetTevDirect

```
GXSetTevDirect(
       GXTevStageID   tevStage);     // TEV stage name
```

11 Fog, Z-compare, Blending, and Dithering

The final pixel output operations include fog, Z-compare, blending, and dithering. These operations are the final steps of the pixel pipeline before a pixel is either written to the frame buffer or discarded. Fog allows pixel values to be blended with the fog color based upon the distance from the viewer. The Z-compare operation determines whether or not rasterized pixels will be written to the frame buffer based upon a Z value comparison. The blending operation allows the rasterized pixel color to be mixed with the color existing in the frame buffer. Logic operations are also possible in the blender. Dithering takes place last.

11.1 Fog

Fog, if enabled, blends a constant fog color with the pixel color output from the last active Texture Environment (TEV) stage. The percentage of fog color blended depends on the *fog density*, which is a function of the distance from the viewpoint to a quad (2x2 pixels).

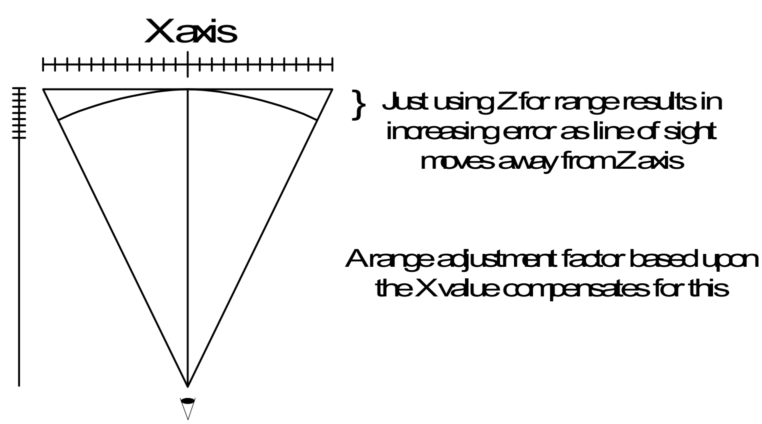
There are five possible fog density functions:

- Linear
- Exponential
- Exponential squared
- Reverse exponential
- Reverse exponential squared

You may program a near and far Z for the fog function independent of the clipping near and far Z.

The eye-space Z used for fog computations does not represent the correct range unless the viewer is facing the same direction as the *z*-axis. The GP can compensate for this with a range adjustment factor based upon the *x* position of the pixels being rendered. This boosts the eye-space Z value used for the fog computation (the *y* direction is not compensated), effectively increasing the fog density towards the edges of the screen in order to make the effect more realistic.

Figure 11-1 Fog Range Adjustment



11.1.1 Fog Curves

These curves show the fog density as a function of range with *startz* = 50 and *endz* = 100.

Figure 11-2 Linear Fog Curve

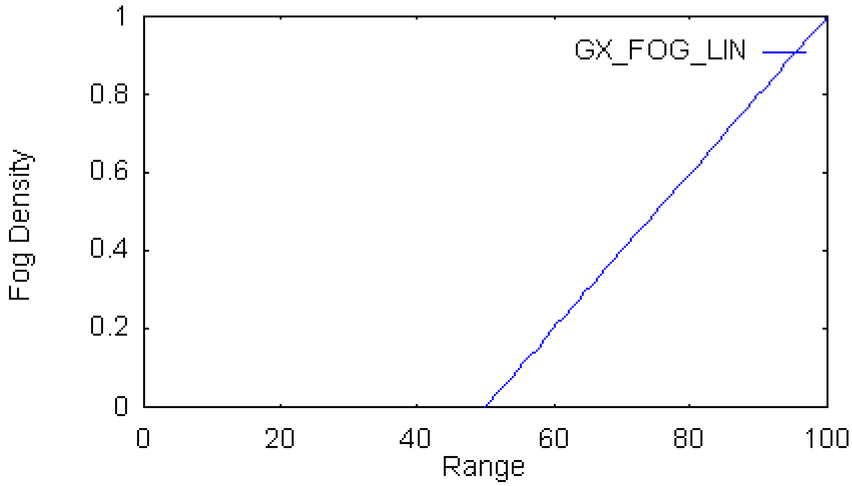


Figure 11-3 Exponential Fog Curve

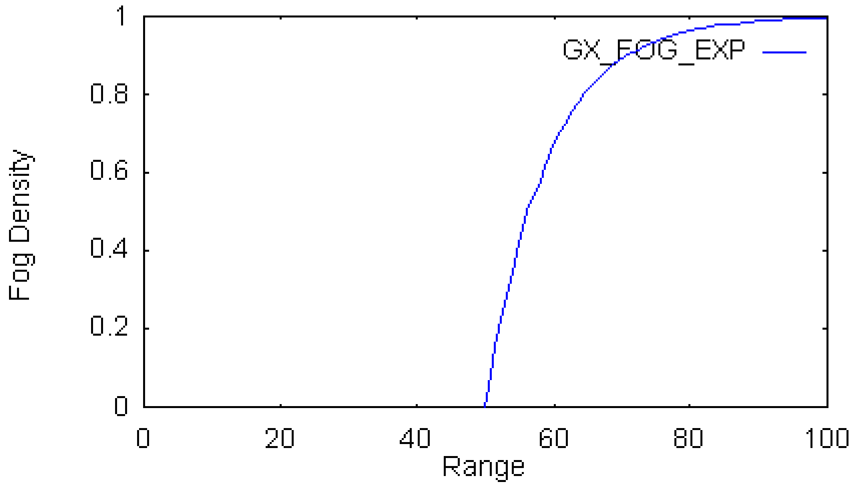


Figure 11-4 Exponential Squared Fog Curve

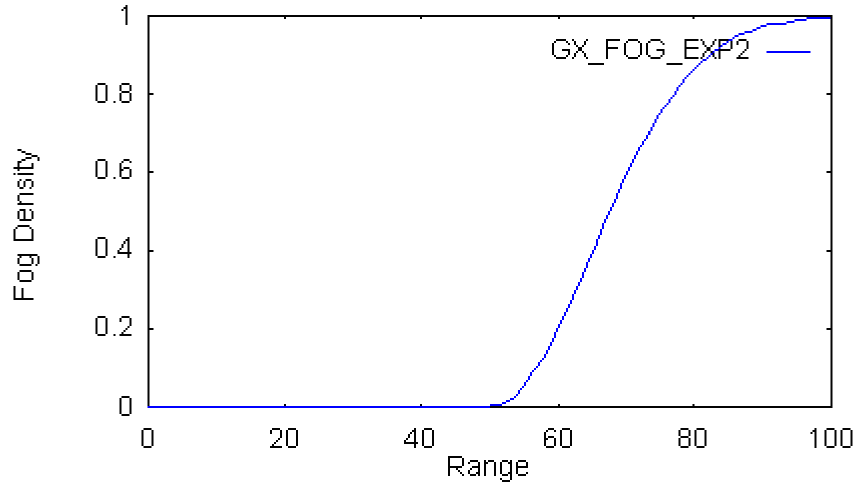


Figure 11-5 Reverse Exponential Fog Curve

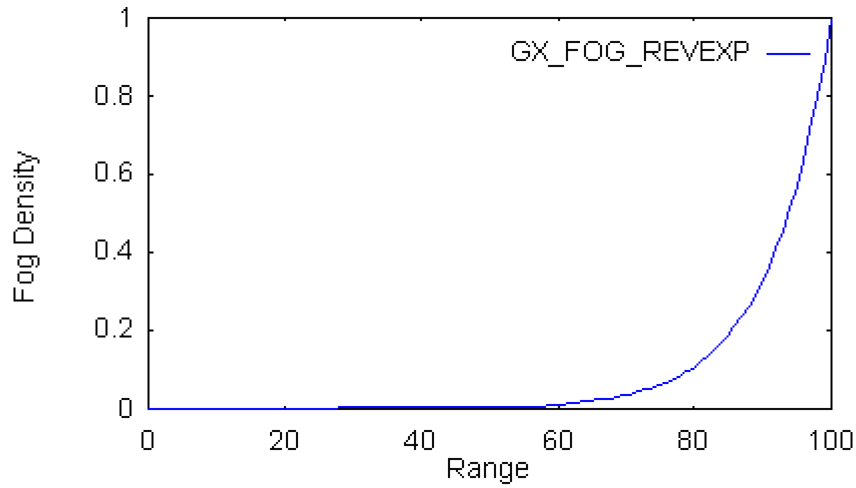
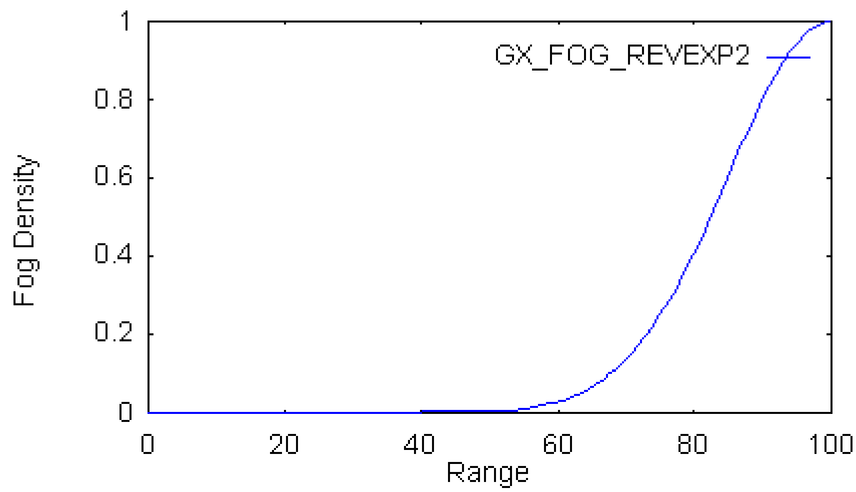


Figure 11-6 Reverse Exponential Squared Fog Curve



11.1.2 Fog Parameters

You can control fog by using the function in Code 11-1.

Code 11-1 GXSetFog

```
void GXSetFog(GXFogType type,
              f32       startz,
              f32       endz,
              f32       nearz,
              f32       farz,
              GXColor   color);
```

The parameters *startz* and *endz* control where the fog function starts and ends, respectively. Usually, the *endz* value is set to the far plane Z. The *nearz* and *farz* are needed to convert the rasterized screen space Z value into eye-space Z for fog computations. *Color* is the color of the pixel when the fog density is 1.0.

The horizontal fog range adjustment is turned off by default in `GXInit`. In order to use this feature, you must call the following two functions:

Code 11-2 Fog Range Adjustment Functions

```
void GXInitFogAdjTable(
        GXFogAdjTable*          table,
        u16                     width,
        f32                     projmtx[4][4] );

void GXSetFogRangeAdj(
        GXBool                  enable,
        u16                     center,
        GXFogAdjTable*          table );
```

The first function is used to compute the adjustment table. The user must provide the allocated space for this *table*. The *width* parameter specifies the width of the viewport. The *projmtx* parameter is the projection matrix that will be used to render into the viewport. This parameter is needed for the function to compute the viewport's horizontal extent in eye space.

Once the *table* has been computed, it can be passed to `GXSetFogRangeAdj`. The *enable* parameter indicates whether horizontal fog range adjustment is enabled or not. The *center* parameter should be the *x*-coordinate at the center of the viewport. The range adjust function is symmetric about *center*.

11.2 Z-compare

You may write to the frame buffer conditionally by comparing the Z value for the rasterized pixel against the Z value for the pixel already in the frame buffer. This comparison may happen at one of two places within the graphics pipeline: the comparison may take place before a pixel is textured, or it may take place after texturing has been done (see "Figure 3-1 Schematic of the GP" on page 10). Normally, the Z-compare occurs before texturing because this typically enhances performance; that is, memory bandwidth is reduced by not looking up textures for pixels that will not be visible. However, the alpha-compare logic is tied together with the Z-compare logic, so using alpha-compare requires placing the Z-compare after texturing. Using Z textures also requires that the Z-compare occur after texturing. The function `GXSetZCompLoc` sets whether the Z-compare happens before or after texturing. The compare is set to "before" in `GXInit`.

The function `GXSetZMode` is used to control the Z-compare:

Code 11-3 GXSetZMode

```
void GXSetZMode(
        GXBool     compare_enable,
        GXCompare  func,
        GXBool     update_enable );
```

The *compare_enable* parameter can be used to disable the Z-compare altogether. When *compare_enable* is false, writes to the Z buffer are also disabled. The *func* parameter sets the comparison function (never, less, less or equal, equal, not equal, greater or equal, greater, or always). The *update_enable* parameter sets whether new Z values are written to the Z buffer when Z-compares are enabled.

11.2.1 Z Buffer Format

The Z buffer is 24 bits wide in non-antialiased mode and 16 bits wide in antialiased mode. When using a 16-bit Z buffer, a number of different compressed formats are available to make better use of the limited number of bits. The compression algorithm performs a type of reverse floating point encoding because the properties of screen space Z necessitates clumping most of the resolution towards the high end of the number scale, whereas conventional floating point notation clumps most of the resolution towards the low end of the number scale.

The system supports various compression schemes, with selection being made based on the far-to-near ratio. For orthographic projection or small far-near ratios, you can use a linear format; it just strips the lower 8 bits from the input Z. For medium far-near ratios, you can use a 14e2 format that has an effective resolution of 15 bits at the near plane and 17 bits at the far plane. For high far-near ratios, you can use a 13e3 format that has an effective resolution of 14 bits at the near plane and 20 bits at the far plane. A 12e4 format is also available.

The 16-bit formats available are listed below:

Table 11-1 16-bit Z Buffer Formats

Format Name	Description
`GX_ZC_LINEAR`	Linear
`GX_ZC_NEAR`	14e2
`GX_ZC_MID`	13e3
`GX_ZC_FAR`	12e4

It is always best to use as little compression as possible (that is, use as many mantissa bits in the Z format as possible). You get less precision with higher compression. The “far” in the above text does not necessarily refer to the far clipping plane. You should think of it as the farthest object you want correct occlusion for.

The Z buffer format is set using `GXSetPixelFmt`. This function affects both the color buffer format and the Z buffer format. For information about color buffer formats, refer to "12 Video Output" on page 129. Changing the frame buffer format requires flushing the pixel pipeline.

11.3 Blending

The blending operation combines the pixel color (output from the TEV and fog operations), also called *the source color*, with Embedded Frame Buffer (EFB) color, also called the *destination color*. The pixel's alpha, or *source alpha*, can always be used as a factor in the blending operation. In addition, if the EFB format is `GX_PF_RGBA6_Z24`, then you can blend the source alpha channel with the EFB alpha, also called *destination alpha*.

When rendering non-antialiased images, four pixels per clock are blended. When rendering antialiased images, six samples, or two pixels per clock, are blended.

You can set the main blending controls using:

Code 11-4 GXSetBlendMode

```
void GXSetBlendMode(
        GXBlendMode    type,
        GXBlendFactor  src_factor,
        GXBlendFactor  dst_factor,
        GXLogicOp      op );
```

The *type* parameter selects between blending operations, `GX_BM_BLEND`, or logical operations, `GX_BM_LOGIC`. Setting *type* to `GX_BM_NONE` writes the source pixel directly to the EFB. Call `GXSetColorUpdate` and `GXSetAlphaUpdate` to enable the writing of the blending result to the EFB.

11.3.1 Blend Equation

The blend equation is:

Equation 11-1 Blending

$$pix\_color = Clamp\ (src\_factor \times src\_color + dst\_factor \times dst\_color)$$

RGB blending is performed using the equation above. Alpha blending differs in that a constant alpha can override the result of the blend equation using `GXSetDstAlpha`.

The *src_factor* and *dst_factor* are both normalized. The normalization process adds the most significant bit (MSB) of the factor to itself, thus 0 through 127 are unchanged, and 128-255 will map to 129 to 256. The 8-bit color is multiplied by the 9-bit factor and the result is rounded. The result of the two multiplies are added and the sum is clamped to 255. This technique allows colors to be multiplied by 1.0 (255 = 1.0), preserving the high end of the color range.

It is also possible to specify a subtractive blend mode as a special blend operation, as shown below.

Equation 11-2 Subtractive Blend Operation

$$final\ pixel\ color = destination\ color - source\ color$$

You select this blend operation using `GXSetBlendMode` and choosing a *blend_mode* of `GX_BM_SUBTRACT`. Unlike additive blending, you cannot specify source and destination factors (coefficients) in this equation. This blend operation is also available with writes from the CPU to the EFB, and selectable using `GXPokeBlendMode`.

11.3.2 Blending Parameters

The following table lists the possible values of *src_factor* and *dst_factor*:

Table 11-2 Blending Parameters

Source or Destination Factor	Red	Green	Blue	Alpha
GX_BL_ZERO	0	0	0	0
GX_BL_ONE	0xff	0xff	0xff	0xff
GX_BL_SRCCLR	Rs	Gs	Bs	As
GX_BL_INVSRCCLR	0xff – Rs	0xff – Gs	0xff – Bs	0xff – As
GX_BL_DSTCLR	Rd	Gd	Bd	Ad
GX_BL_INVDSTCLR	0xff – Rd	0xff – Gd	0xff – Bd	0xff – Ad
GX_BL_SRCALPHA	As	As	As	As
GX_BL_INVSRCALPHA	0xff – As	0xff – As	0xff – As	0xff – As
GX_BL_DSTALPHA	Ad	Ad	Ad	Ad
GX_BL_INVDSTALPHA	0xff – Ad	0xff – Ad	0xff – Ad	0xff – Ad

11.3.3 Logic Operations

These logical operations are supported:

Table 11-3 Logic Operations

Logic Op Name	Operation
GX_LO_CLEAR	0x00
GX_LO_SET	0xff
GX_LO_COPY	Source
GX_LO_INVCOPY	~Source
GX_LO_NOOP	Destination
GX_LO_INV	~Destination
GX_LO_AND	Source & destination
GX_LO_NAND	~(Source & destination
GX_LO_OR	Source | destination
GX_LO_NOR	~(Source | destination)
GX_LO_XOR	Source ^ destination
GX_LO_EQUIV	~(Source ^ destination)
GX_LO_REVAND	Source & ~destination
GX_LO_INVAND	~Source & destination
GX_LO_REVOR	Source | ~destination
GX_LO_INVOR	~Source | destination

Logic operations and blend operations cannot be used simultaneously.

11.4 Dithering

The pixel can be dithered after blending when the pixel format is either `GX_PF_RGBA6_Z24` or `GX_PF_RGB565_Z16`. There is no performance penalty for turning on dithering. Each 8-bit color component is scaled and normalized appropriately (see Equasion 11-4 through 11-7) and the corresponding entry in the standard 4x4 Bayer matrix is added. The Bayer matrix is screen-aligned and repeated over the entire screen.

Equation 11-3 Bayer Matrix

$$x \longrightarrow \qquad y \downarrow \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$

The following equations compute scaling and normalizing for dithering:

Equation 11-4 5-bit Dithering (ideal)

$$C_{out} = ((C_{in} * 247/255) << 1) + \text{dither}[x \& 3, y \& 3]) >> 4$$

Equation 11-5 5-bit Dithering (approximation actually used)

$$C_{out} = ((C_{in} - (C_{in} >> 5)) << 1) + \text{dither}[x \& 3, y \& 3]) >> 4$$

Equation 11-6 6-bit Dithering (ideal)

$$C_{out} = ((C_{in} * 251/255) << 2) + \text{dither}[x \& 3, y \& 3]) >> 4$$

Equation 11-7 6-bit Dithering (approximation actually used)

$$C_{out} = ((C_{in} - (C_{in} >> 6)) << 2) + \text{dither}[x \& 3, y \& 3]) >> 4$$

You can enable dithering by calling the `GXSetDither` function. Dithering is enabled by default in `GXInit`.

Code 11-5 GXSetDither

```
void GXSetDither( GXBool enable );
```

12 Video Output

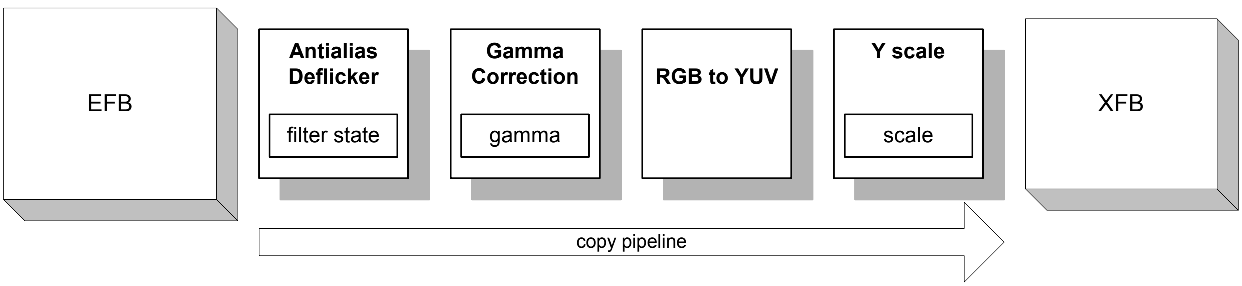
The Embedded Frame Buffer (EFB) cannot send pixel data directly to the video interface; therefore, the frame buffer must first be copied out to main memory. We call the frame buffer in main memory the External Frame Buffer (XFB). In this section, we will fully describe the copy operation necessary to transfer EFB to XFB.

Since the GP copy operation relates closely to video interface functionality, the remainder of this section often refers to Video Interface library (VI). For a complete description of this library, see *Video Interface Library (VI)*.

12.1 The Copy Pipeline

The diagram below illustrates the operations applied during the frame buffer copying process.

Figure 12-1 EFB-to-XFB Copy Pipeline



12.1.1 Copy Source

The EFB source of the copy operation is specified by the function in Code 12-1.

Code 12-1 GXSetDispCopySrc

```
void GXSetDispCopySrc(
    u16 left,
    u16 top,
    u16 wd,
    u16 ht );
```

`GXSetDispCopySrc` defines a sub-region of pixels in the EFB memory as the source for the copy operation. Since the GP works on regions of 2x2 pixels, there is the restriction that all of the source copy parameters be even numbers.

12.1.2 Antialiasing and Deflickering

The GP performs antialiasing in two parts. During rendering, it can rasterize to a super-sampled EFB. During the copy operation, the multiple samples per pixel can be filtered together to create the final pixel output color. In fact, samples from more than one row of pixels can be filtered together, enabling deflickering to be performed during the copy operation as well. Samples from up to three rows of pixels can be filtered together (with some restrictions). A more detailed discussion of Wii antialiasing and deflickering may be found in *Advanced Rendering*.

The antialiasing mode is determined by the frame buffer pixel format. "12.4 Embedded Frame Buffer Formats" on page 137 discusses how to set this format. The function in Code 12-2 sets all super-sample locations and sample filter weights.

Code 12-2 GXSetCopyFilter

```
u32 GXSetCopyFilter(
    GXBool aa,
    u8     sample_pattern[12][2],
    GXBool vf,
    u8     vfilter[7] );
```

The *aa* parameter indicates whether to use the supplied *sample_pattern* or a default pixel-centered pattern. The *sample_pattern* parameter indicates the exact location of each pixel subsample. The *vf* parameter indicates whether to use the supplied *vfilter* or a default single-line filter. The *vfilter* indicates the weights to use for each sample. Refer to *Advanced Rendering* for more details on these parameters.

When rendering in non-antialiased mode, the sample pattern must be set to a pixel-centered pattern, or else visual anomalies will result. Similarly, the vertical filter must always be set correctly depending upon the render mode chosen. Note, however, that the vertical filter only samples from the adjacent *rendered* pixels (in the EFB). When rendering in field mode, such pixels are *not* adjacent during scan-out (since the odd and even fields from the XFB will be interlaced with each other); therefore, the vertical filter should not be used in this mode. Moreover, the vertical filter is unnecessary in double-strike mode. See below for further discussion about render modes.

The same copy hardware is used for the video copy path as well as the texture copy path, so it might be necessary to change the sample pattern and vertical filter within a frame when doing a texture copy followed by a video copy.

Clamping is generally required when copying the first and last scan lines from the source rectangle. This makes sure that the GP uses valid data when sampling above the first scan line and below the last. However, there may be times when it is necessary to disable clamping (see "12.2.7 Interlaced, Antialiased, Frame-rendering, Deflicker Mode" on page 135). Clamping is controlled by the call in Code 12-3.

Code 12-3 GXSetCopyClamp

```
u32 GXSetCopyClamp( GXFBClamp clamp );
```

12.1.3 Gamma Correction

Pixel values may be gamma-corrected during the copy operation. Three choices of gamma correction are available: 1.0, 1.7, and 2.2. The default gamma set in `GXInit` is 1.0. The gamma for texture copy is fixed at 1.0. The display-copy gamma can be set with the function in Code 12-4.

Code 12-4 GXSetDispCopyGamma

```
u32 GXSetDispCopyGamma( GXGamma gamma );
```

Determining the appropriate gamma correction will depend on your design for the game. For many games, you may find it preferable to maximize low intensity color resolution by setting the gamma correction at 1.0.

12.1.4 RGB to YUV

A luma/chroma YUV format stores nearly the same visual quality pixel as RGB does, but requires only two-thirds of the memory. Therefore, we convert RGB EFB to YUV XFB during the copy operation to save on the amount of main memory used for the frame buffer. There is a corresponding savings of main memory bandwidth as well (for both the copy operation and the XFB video scan-out). For conversion details, see "12.5 External Frame Buffer Format" on page 138.

As an example, consider a 640x240 (150,000 pixels) frame buffer. A frame buffer this size requires 900KB to double-buffer using 24-bit/pixel RGB format. However, a 16-bit/pixel YUV format requires only 600KB, which saves a difference of 300KB in main memory.

12.1.5 Y Scale

Wii can arbitrarily scale a rendered image both horizontally and vertically. Vertical *y* scale occurs during the copy process, while horizontal *x* scale occurs during the video display. For more details on scaling (*x*, *y*) for display, see “Initialization” in *Video Interface Library (VI)*. The function in Code 12-5 sets the *y* scale factor.

Code 12-5 GXSetDispCopyYScale

```
u32 GXSetDispCopyYScale( f32 yscale );
```

The function returns the number of lines that will be copied, which can be used to compute the XFB size.

12.1.6 Copy Destination

The destination of the copy operation is defined by:

Code 12-6 GXCopyDisp

```
void GXSetDispCopyDst( u16 width, u16 height );
void GXCopyDisp( void *dest, GXBool clear );
```

`GXSetDispCopyDst` must be called in order to set the proper stride for the copy operation. `GXCopyDisp` specifies the XFB destination in main memory and actually issues the copy command. The destination XFB must begin at a 32-byte aligned address; moreover, the amount of memory required depends on width alignment. For a complete list of rules for allocating the correct amount of XFB memory, see “4.2.2 Frame Buffer Allocation” in *Video Interface Library (VI)*.

12.1.7 Clear Color and Z for Next Frame

The copy operation can clear the color frame buffer and the Z buffer during the copy. This eliminates clear time when rendering the next frame.

To perform a clear during the copy operation, use the `clear` parameter in `GXCopyDisp`. The `clear` parameter is only effective if the buffer has been enabled for update (see `GXSetColorUpdate`, `GXSetAlphaUpdate`, and `GXSetZMode`). This allows individual buffers to be cleared during the copy operation.

The function in Code 12-7 specifies the clear color and clear Z values to use during the copy operation.

Code 12-7 GXSetCopyClear

```
void GXSetCopyClear( GXColor clear_clr, u32 clear_z);
```

The *clear_clr* parameter is in RGBA8 format, while the *clear_z* parameter is in 24-bit format. The parameters are converted into the proper format during the clear operation. The constant `GX_MAX_Z24` specifies the maximum depth value.

12.2 Predefined Render Modes

The set of controls necessary to correctly configure the GX and VI libraries is complex. Many game designers like to be able to customize these controls, so the GX and VI libraries provide all of the necessary controls to give developers maximum control. The GX API provides a set of predefined rendering modes containing all of the parameters necessary to make this task simpler.

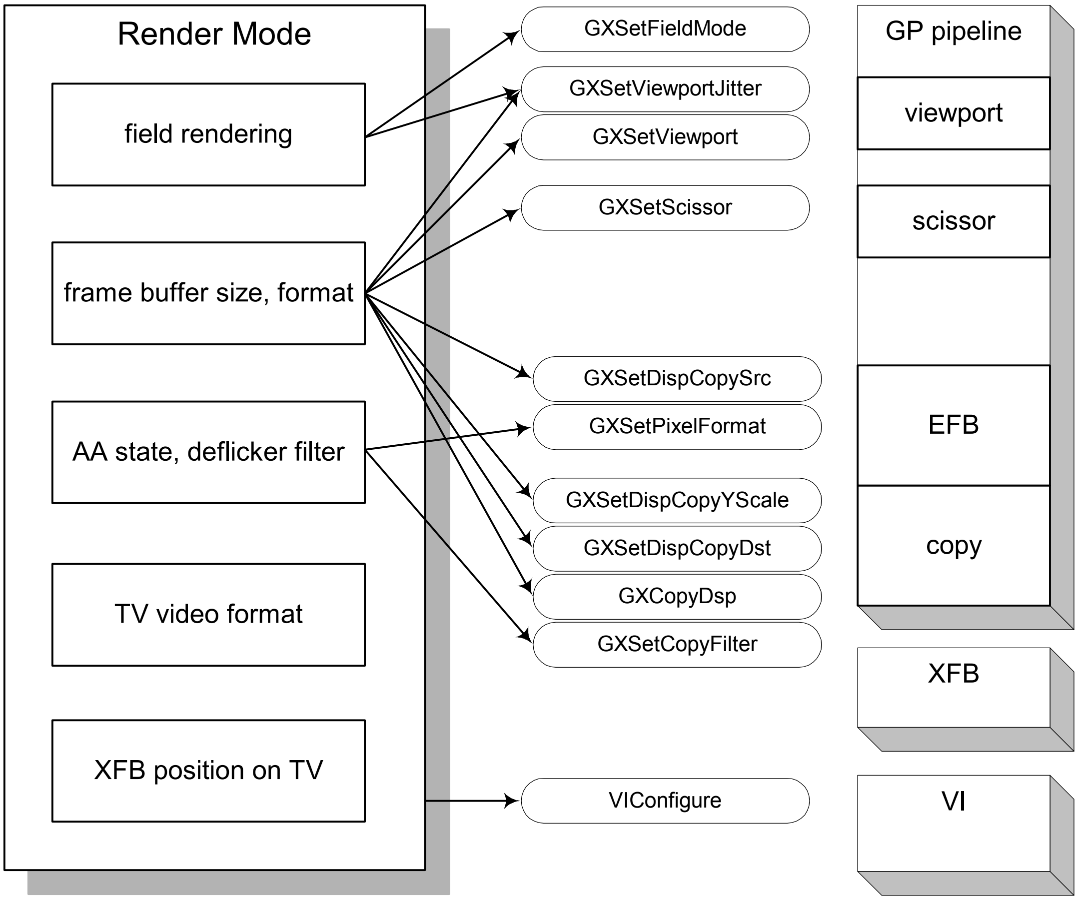
Each rendering mode contains the following data:

- EFB and XFB size and format information
- Position of the XFB on TV screen
- TV video format: interlaced or non-interlaced (double-strike) mode for NTSC, PAL, and EURGB60; or progressive mode (with 480 valid scan lines, using a standard D2 connector) for NTSC and EURGB60.
- Antialiasing state and deflicker filter
- Field-rendering mode (enabled or not)

For information on TV video formats and field-rendering, refer to *Video Interface Library (VI)*.

The diagram below illustrates the relationships between the render mode structure, related GX calls, and parts of the graphics hardware pipeline:

Figure 12-2 Render Mode Structure, Related Calls and Hardware Modules

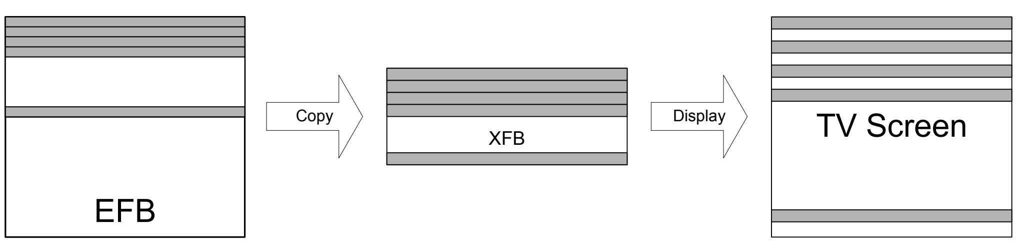


In the following sections, we describe the basic rendering modes defined for each of the video modes.

12.2.1 Double-strike, Non-antialiased Mode

In this mode, VI outputs a double-strike, or non-interlaced, signal. This mode turns off antialiasing (AA) to speed up fill rate. For NTSC, this mode renders 640x240 lines at 60Hz.

Figure 12-3 Double-strike, Non-antialiased Mode



12.2.2 Double-strike, Antialiased Mode

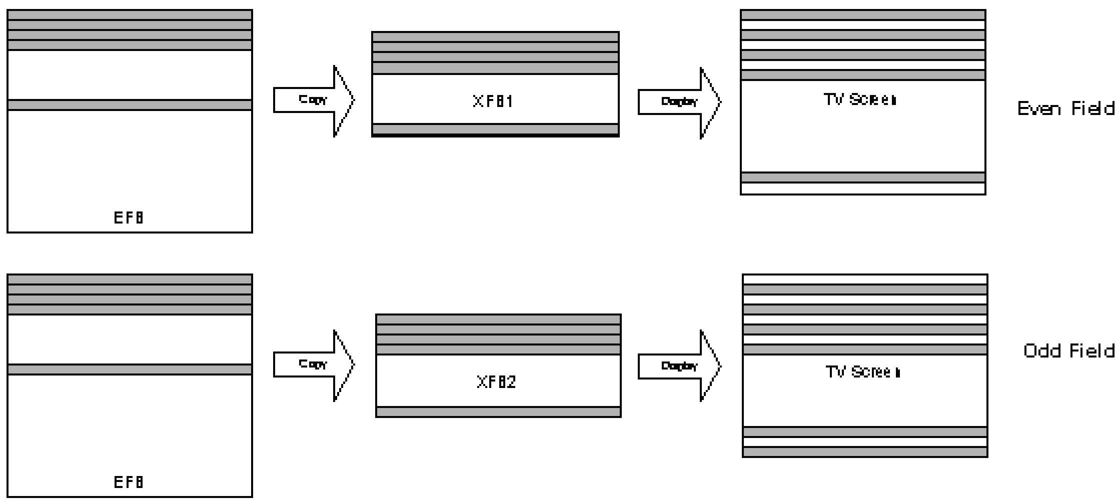
This mode is similar to the preceding one; however, it supports antialiasing (at a potentially reduced fill rate).

12.2.3 Interlaced, Non-antialiased, Field-rendering Mode

In this mode, VI outputs an interlaced signal (the rendering alternates between even and odd fields to support field-rendering). Antialiasing is turned off for maximum fill rate. For NTSC, this mode renders 640x240 lines at 60Hz.

Please note that deflicker is not possible in this mode. Also note that the fields must be completed and swapped before vertical retrace, or else the incorrect field is displayed during the next display interval. Otherwise, incorrect fields will be displayed during the following display periods.

Figure 12-4 Interlaced, Non-antialiased, Field-rendering Mode



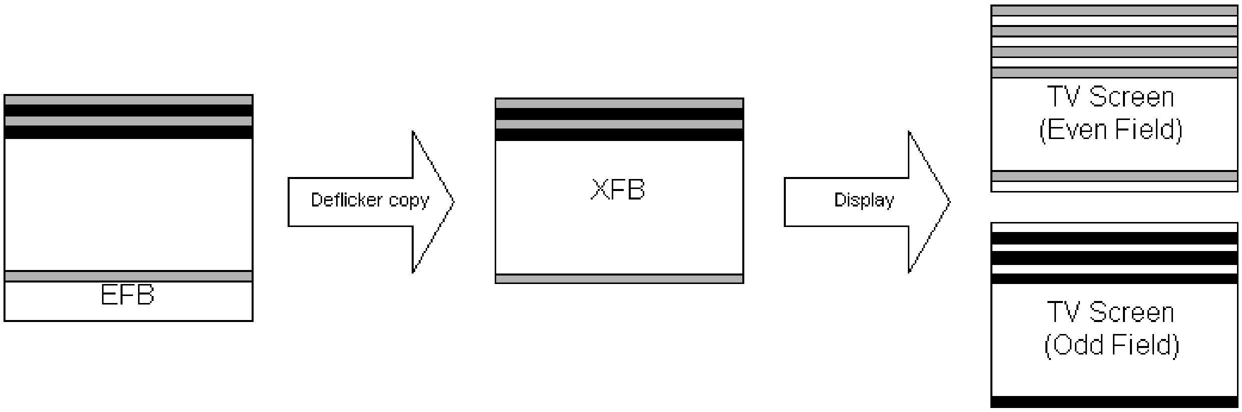
12.2.4 Interlaced, Antialiased, Field-rendering Mode

This is the same as the preceding mode, except that it supports antialiasing (at a potentially reduced fill rate).

12.2.5 Interlaced, Non-antialiased, Frame-rendering, Deflicker Mode

In this mode, VI outputs an interlaced signal. The entire frame is copied from EFB to XFB with a deflickering filter. The video interface hardware can select whether to display the even or odd field within this frame buffer. For NTSC, this mode renders 640x480 lines at any frame rate.

Figure 12-5 Interlaced, Non-antialiased, Frame-rendering, Deflicker Mode



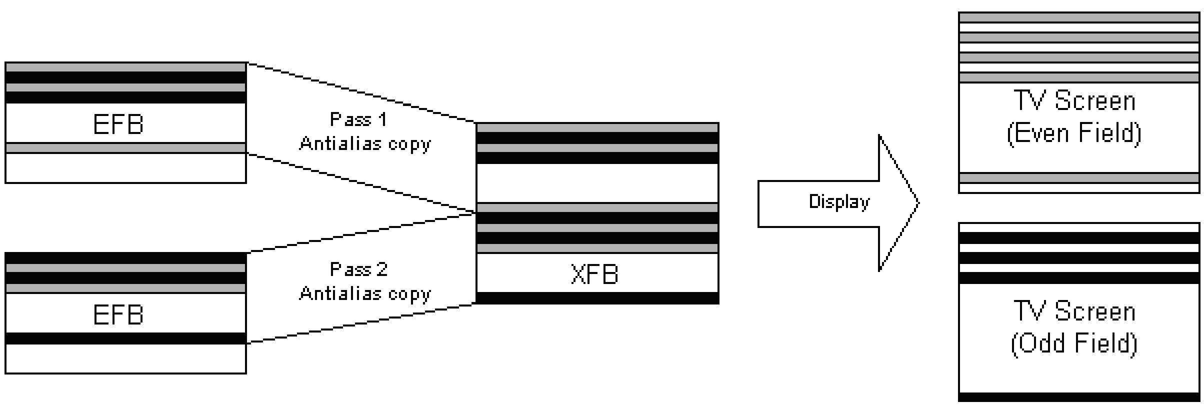
12.2.6 Interlaced, Non-antialiased, Frame-rendering, Non-deflicker Mode

Similar to interlaced, frame-rendering, deflicker mode, except that this mode does not support deflickering.

12.2.7 Interlaced, Antialiased, Frame-rendering, Deflicker Mode

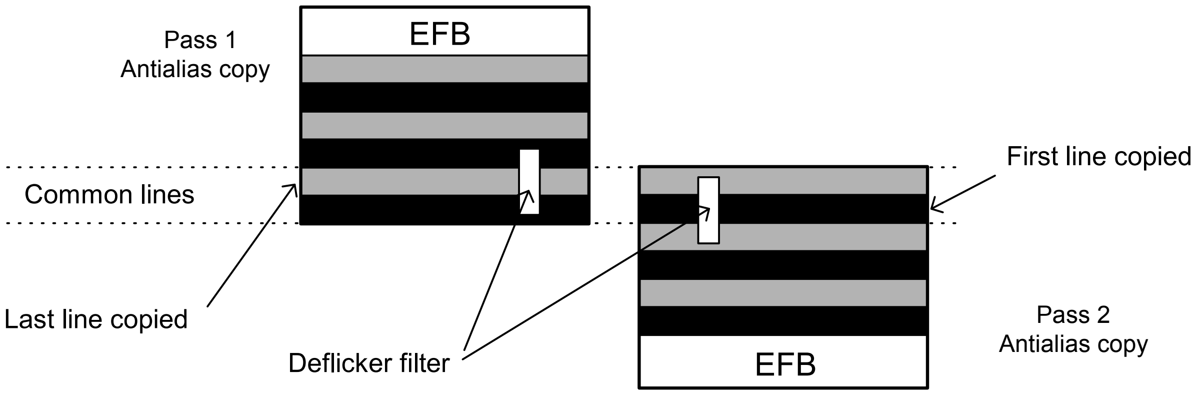
When rendering a large antialiased frame, the embedded frame buffer is not big enough to hold the entire frame, so two rendering passes are necessary to construct a single complete frame buffer.

Figure 12-6 Interlaced, Antialiased, Frame-rendering, Deflicker Mode



Since the vertical deflicker filter spans three lines, it is necessary to have some overlap in each pass where the images will be joined, as illustrated in "Figure 12-7 Overlapping Copy" on page 136. This diagram shows one extra line in each pass for a total of two lines of overlap. Due to the restriction that only even numbers of lines can be copied, you must actually have two extra lines from each pass, resulting in a total overlap of four lines. Please note that copy clamping is not possible at the bottom of the first pass copy and at the top of the second pass copy.

Figure 12-7 Overlapping Copy



Drawing the top and bottom halves of the screen correctly involves adjusting the viewing frustum. See the demo `frb-aa-full` for an example.

It is also possible to shift the area of the scissor box within the space of the EFB memory. Normally, the upper-left corner of the scissor box maps to the same corner in EFB space. You can now specify an offset that is subtracted from the computed pixel's location before it is stored in the EFB. Thus, you can shift the area of the scissor box up and/or left within the EFB space.

The main purpose of this feature is to simplify dual-pass rendering for antialiasing. You can now maintain the same viewport for rendering the upper and lower halves of the screen. The upper half is drawn with the scissor box set to the upper half of the viewport and the offset set to zero. The lower half is drawn by moving the scissor box to the bottom half of the viewport and adjusting the offset to place the scissor box area within the EFB's valid area (since the EFB is only half the screen height in antialiased frame-rendering mode).

The offset is adjusted using the function `GXSetScissorBoxOffset`, as shown in Code 12-8.

Code 12-8 GXSetScissorBoxOffset

```
GXSetScissorBoxOffset( u32 xoff, u32 yoff );
```

12.2.8 Progressive Mode

This mode renders 480 lines and displays all lines at 60 Hz. The frame rate will be 60 Hz or less. Vertical filtering is turned off in this mode, resulting in sharp image output.

12.2.9 Progressive, "Soft" Mode

This mode renders 480 lines and displays all lines at 60 Hz. The frame rate will be 60 Hz or less. Vertical filtering is turned on in this mode, resulting in image output that is "softer" than normal progressive mode.

12.2.10 Progressive Antialised Mode

This mode renders 480 lines and displays all lines at 60 Hz. The frame rate will be 60 Hz or less. Two-pass rendering is required to render the entire frame in this mode. Vertical filtering is turned off to process antialiasing correctly.

12.3 GX API Default Render Mode

`GXInit` queries `VIGetTVFormat` to determine the GX API default render mode. The default mode may be one of the following, depending on format:

- `GXNtsc480IntDf`
- `GXPal528IntDf`
- `GXEurgb60Hz480IntDf`

These modes feature:

- Full-screen frame-based rendering
- Non-antialiased for the fastest fill rate performance
- Deflickered display to reduce flickering artifacts
- Interlaced display output

If `DEMOInit` is called with a non-null render-mode pointer, then the referenced render mode is used. If the pointer is null, then a default render mode is used. In addition, if a default mode is used, `DEMOInit` will call `GXAdjustForOverscan` and trim 16 scan lines off the top and bottom of the screen. This adjustment is for demonstration purposes only; it is not guaranteed to make the viewport visible on all television sets.

12.4 Embedded Frame Buffer Formats

The EFB has a maximum memory capacity of 2027520B = 640 x 528 x (3B(color) + 3B(Z)). The maximum pixel width and height of the frame buffer is determined by the size of each pixel. There are two different pixel sizes:

- 48-bit color and Z
- 96-bit super-sampled color and Z

To set these formats, you must call `GXSetPixelFmt`, shown in Code 12-9. Setting the pixel format also controls the antialiasing mode. You must also call `GXSetCopyFilter` (see Code 12-2) when changing mode.

Code 12-9 GXSetPixelFmt

```
void GXSetPixelFmt( GXPixelFmt pix_fmt, GXZFmt16 z_fmt );
```

Changing pixel formats causes a flush of the rendering pipeline. Also, data existing in the frame buffer is not converted when you change formats, so mixed-format rendering is not possible in this manner. As a result, it may be necessary to clear the frame buffer again after changing modes.

12.4.1 48-bit Format – Non-antialiasing

The 48-bit format is intended for non-antialiasing; it has the following features:

- 24-bit color (either 8/8/8 with no alpha, or 6/6/6/6 with 6 bits of alpha)
- 24-bit Z

This format can support a maximum resolution of 640x528. The width must be between 0-640 and the EFB stride is fixed at 640 pixels.

12.4.2 96-bit Super-sampling Format – Antialiasing

The 96-bit pixel format is for antialiasing and has the following features:

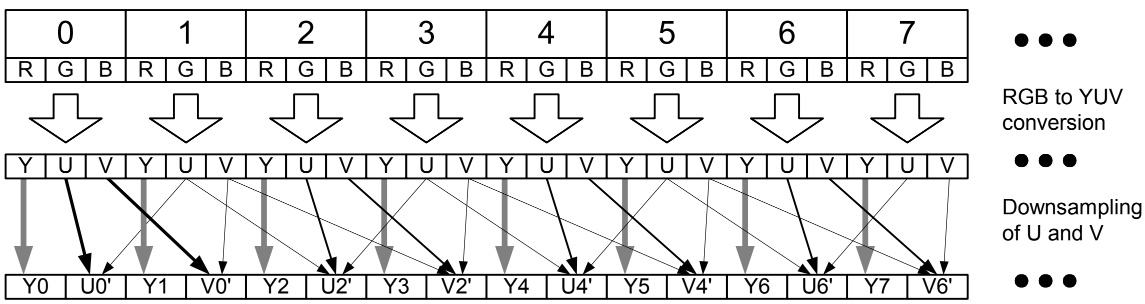
- 16-bit color frame buffer per sample (5/6/5, no alpha)
- 16-bit Z buffer per sample

This format can support a maximum resolution of 640x264. The width must be between 0-640 and the EFB stride fixed at 640 pixels.

12.5 External Frame Buffer Format

Pixels in the XFB are stored as illustrated in Figure 12-8.

Figure 12-8 XFB Format in Main Memory



$$U(i) = 1/4 * U(i-1) + 1/2 * U(i) + 1/4 * U(i+1)$$

$$V(i) = 1/4 * V(i-1) + 1/2 * V(i) + 1/4 * V(i+1)$$

In the down-sampling process indicated above, clamping is included for the left and right edges.

The following computations illustrate the conversion of RGB to YUV:

Equation 12-1 RGB to YUV conversion

$$Y = 0.257*R + 0.504*G + 0.098*B + 16$$

$$U = -0.148*R - 0.291*G + 0.439*B + 128$$

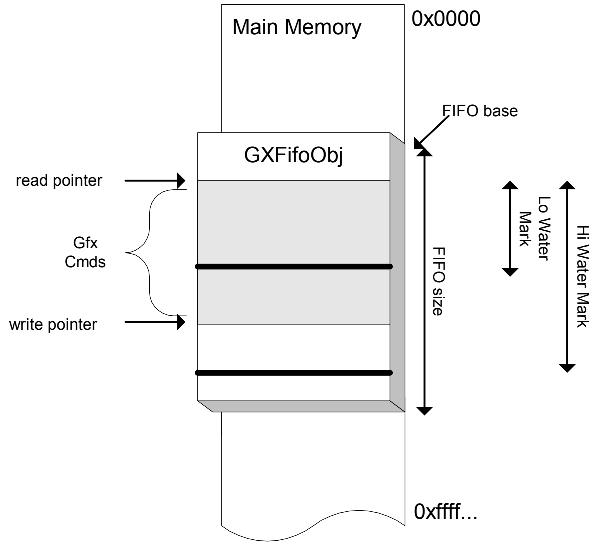
$$V = 0.439*R - 0.368*G - 0.071*B + 128$$

The range for Y is only 16 <= Y <= 235. This is in order to meet the requirements of the video encoder.

13 Graphics FIFO

13.1 Description

Figure 13-1 GXFifoObj

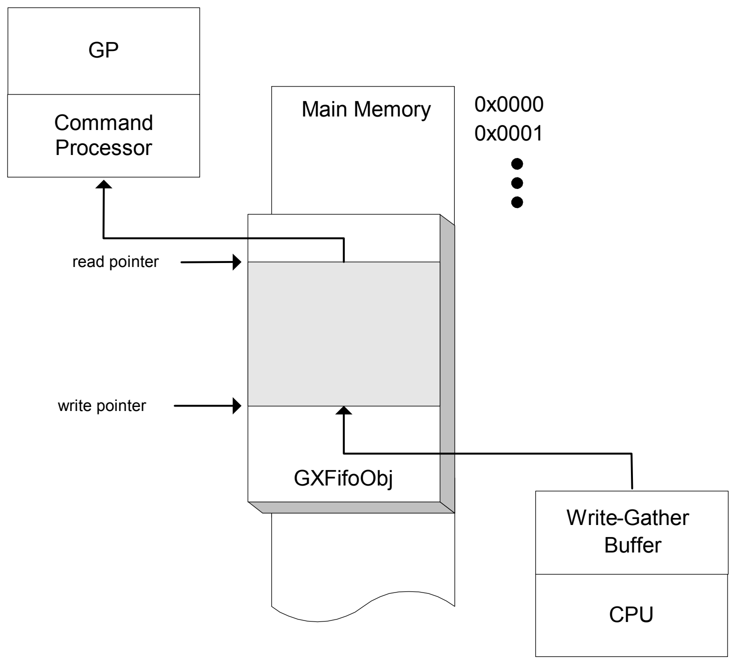


The GX API transmits commands from the CPU to the Graphics Processor (GP) using a `GXFifoObj` structure. The `GXFifoObj` structure describes a region of main memory, allocated by the application, set aside for storing graphics commands. The FIFO can be attached to either the CPU or GP or both. When the FIFO is attached to the CPU, GX commands will be written to the FIFO. There is always one—and only one—FIFO attached to the CPU. When the FIFO is attached to the GP, the GP will read and process graphics commands. Only one FIFO can be attached to the GP at a time.

The purpose of the FIFO is to allow the CPU and GP to work in parallel at close to their peak rates. There are two basic methods of using the FIFO to achieve parallelism: *immediate mode* and *multi-buffer mode*.

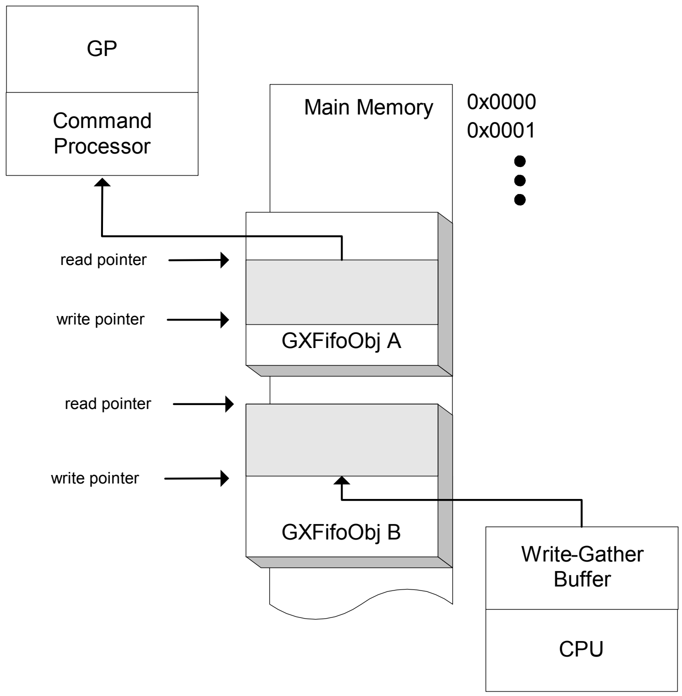
When a single FIFO is attached to both the CPU and the GP, the system is said to be in *immediate mode*. The FIFO read and write pointers are managed by hardware as a true FIFO. As the CPU writes graphics commands to the FIFO, the GP will process them in order. The hardware contains special flow control logic to prevent writes from over running reads and to wrap the read and write pointers from the last address of the buffer back to the first address. `GXInit` sets up the system to use immediate mode by default. Immediate mode is generally easier to use, because once it is set up, no further management by the application is required.

Figure 13-2 Immediate Mode



It is also possible to connect one FIFO to the CPU while the GP is reading from a *different* FIFO. This is called *multi-buffer mode*. In this case, the FIFOs are managed more like buffers than FIFOs, since there are no simultaneous reads and writes to a FIFO. You may choose multi-buffer mode if you require dynamic memory management of FIFOs; however, there are complications that make this choice less desirable. These will be described below.

Figure 13-3 Multi-Buffer Mode



The CPU must always write graphics commands to the FIFO in 32-byte units. To do this, the CPU has a special write-gather function that automatically packs graphics commands into 32-byte words. The GP always reads graphics commands from the FIFO in 32-byte units.

13.2 Creating a FIFO

The GX API declares a static `GXFifoObj` structure internally. This structure is initialized when `GXInit` is called.

Code 13-1 GXFifoObj

```
GXFifoObj* GXInit(void* base, u32 size);
```

The FIFO *base* pointer must be aligned to 32 bytes. Use the memory allocation functions provided by the OS and MEM libraries to allocate a FIFO memory region that is guaranteed to be 32-byte aligned.

Note: The FIFO region must be allocated from internal system memory (MEM1 region). It *cannot* be allocated from external memory (MEM2 region).

The *size* parameter passed to `GXInit` is the size of the FIFO in bytes, which must be a multiple of 32. The minimum FIFO size is 64KB. `GXInit` sets up the FIFO for immediate-mode graphics; both the CPU and GP are connected to the FIFO, the read and write pointers are initialized to the base pointer, and the high- and low-water marks (see "13.4 FIFO Status" on page 143) are enabled.

`GXInit` returns a pointer to the initialized `GXFifoObj` to the application.

If the application wants to operate in multi-buffered mode, then it must allocate additional FIFOs. The application must allocate the memory for each additional FIFO and initialize a `GXFifoObj` as well. The functions in Code 13-2 initialize the `GXFifoObj`.

Code 13-2 FIFO Initialization Functions

```
void GXInitFifoBase(
        GXFifoObj* fifo,
        void*      base,
        u32        size);
void GXInitFifoPtrs(
        GXFifoObj* fifo,
        void*      read_ptr,
        void*      write_ptr );
void GXInitFifoLimits(
        GXFifoObj* fifo,
        u32        hi_water_mark,
        u32        lo_water_mark );
```

Normally, the application only needs to initialize the FIFO read and write pointers to the base address of the FIFO. Once initialized, the system hardware will control the read and write pointers automatically.

The application only needs to call `GXInitFifoLimits` when the FIFO will be used in immediate mode. This function sets the high and low water marks for the FIFO, which are not available in multi-buffer mode (see "13.5 FIFO Flow Control" on page 144).

These APIs are intended for use on FIFOs that are not attached to the CPU or GP. This is to prevent any temporary inconsistencies in the pointers and water mark values. These APIs will cause assertion failures if they are used on attached FIFOs.

The following inquiry functions may be used to retrieve the data set above. To get the current FIFO pointers, refer to "13.4 FIFO Status" on page 143.

Code 13-3 FIFO Basic Inquiry Functions

```
void* GXGetFifoBase( const GXFifoObj* fifo);
u32   GXGetFifoSize( const GXFifoObj* fifo);
void  GXGetFifoLimits( const GXFifoObj* fifo, u32* hi, u32* lo);
```

13.3 Attaching and Saving FIFOs

Once a FIFO has been initialized, it can be attached to the CPU or the GP or both. Only one FIFO may be attached to either the CPU or GP at the same time. Once a FIFO is attached to the CPU, the CPU may issue GX commands to the FIFO. When a FIFO is attached to the GP, it will be enabled to read graphics commands from the FIFO. The following functions attach FIFOs:

Code 13-4 FIFO Attachment Functions

```
void GXSetCPUFifo( const GXFifoObj* fifo ) ;
void GXSetGPFifo( const GXFifoObj* fifo );
```

You may inquire which FIFO objects are currently attached with these functions:

Code 13-5 FIFO Attachment Inquiry Functions

```
GXBool GXGetCPUFifo( GXFifoObj* fifo );
GXBool GXGetGPFifo( GXFifoObj* fifo );
```

In multi-buffer mode, when the CPU is finished writing GX commands, the FIFO should be “saved” before switching to a new FIFO. The following function saves the CPU FIFO:

Code 13-6 GXGetCPUFifo

```
GXBool GXGetCPUFifo( GXFifoObj*  fifo );
```

This function gets the current state of the CPU FIFO as a structure object specified by *fifo*. If it is successfully obtained, `GX_TRUE` will be returned; otherwise, if the CPU FIFO has not been configured, `GX_FALSE` will be returned.

Note: This function gets the current state of the CPU FIFO directly, unlike the function of the same name in the Nintendo GameCube SDK, which only returned a pointer to the `GXFifoObj` structure previously set by `GXSetCPUFIfo`.

Since SDK 3.1 patch 3, this function has been changed to not flush the write-gather buffer internally. To get the same behavior as before, call `GXFlush` once before `GXGetCPUFifo` following the last command insertion into the FIFO.

There is no save function for the GP. Once the GP is attached, graphics commands will continue to be read until either:

- The FIFO is empty
- A FIFO breakpoint is encountered (see "13.6.3 FIFO Breakpoint" on page 146)
- The GP is preempted using `GXAbortFrame` (see "13.6.4 Abort Frame" on page 147)

13.4 FIFO Status

You can use the following functions to read the status of a FIFO and the GP:

Code 13-7 FIFO Status Functions

```
void GXGetGPStatus(
        GXBool* overhi,
        GXBool* underlow,
        GXBool* readIdle,
        GXBool* cmdIdle,
        GXBool* brkpt );

void GXGetFifoPtrs(
        GXFifoObj*  fifo,
        void**      readPtr,
        void**      writePtr );
```

Use `GXGetGPStatus` to get the status of the GP (regardless of the FIFO that is attached to it). The minimum requirement before attaching a new GP FIFO is to wait for the *readIdle* status to be `GX_TRUE`. Normally, additional requirements would include making sure that all graphics commands have been rendered into the EFB, and that the EFB has been copied to main memory. The *cmdIdle* status provides the additional information that the command processor is idle.

The parameters *underlow* and *overhi* indicate where the write pointer is, relative to the high and low water marks. They do not indicate any error in processing.

`GXGetFifoPtrs` may be used to request the read and write pointers of the given FIFO.

13.5 FIFO Flow Control

When a FIFO is attached to both the CPU and GP (immediate mode), care must be taken so that the CPU stops writing commands when the FIFO is too full. The high water mark defines how full the FIFO can get before graphics commands will no longer be written to it. Since there may be up to 16KB of buffered graphics commands in the CPU, we recommend that you set the high water mark to the FIFO size less 16KB. (This 16KB figure comes into play if the locked-cache mechanism is used to write to a FIFO.)

When the high water mark is encountered, the program will be suspended, but other interrupt-driven tasks such as audio will continue. However, in a multi-threaded program, the library will have to choose a particular thread to suspend. By default, the thread that called `GXInit` is suspended (which, in a single-threaded application, would be the main loop). However, you may designate a different thread to be suspended with the APIs in Code 13-8.

Code 13-8 APIs to Get and Set the Current GX Thread

```
OSThread* GXGetCurrentGXThread ( void );
OSThread* GXSetCurrentGXThread ( void );
```

`GXSetCurrentGXThread` will designate the calling thread as the current GX thread and return a pointer to the previous GX thread. `GXGetCurrentGXThread` will return a pointer to the current GX thread. It is a programming error to call `GXSetCurrentGXThread` while the previous GX thread is suspended waiting for a low water mark. This condition indicates that, potentially, your program has two threads generating GX data. An assertion failure will occur in this situation.

The low water mark defines how empty the FIFO must become after reaching a high water mark before the program (or GX thread) is allowed to continue. We recommend that the low water mark be set to (FIFO size / 2). The low water mark prevents frequent context switching in the program, since it does not need to poll some register or constantly receive overflow interrupts when the amount of new command data stays close to the high water mark.

When in multi-buffered mode, the high and low water marks are disabled. When a FIFO is attached to the CPU, and the CPU writes more commands than the FIFO will hold, the write pointer will be wrapped from the last address back to the base address. Previous graphics commands in the FIFO will be overwritten. It is possible to detect only when the write pointer wraps over the top of the FIFO (which indicates an overflow *only* if the FIFO's write pointer was initialized to the base of the FIFO before commands were sent).

In order to prevent FIFO (buffer) overflow in multi-buffered mode, the application must use a software-based checking scheme. The program should keep its own counter of the buffer size, and before any group of commands is added to the buffer, the program should check and see if there is room. If room is available, the size of the group should be added to the buffer size. If room is not available, the buffer should be flushed and a new one allocated.

Instead of using multi-buffered mode, it may be preferable to use a single large FIFO along with breakpoints to simulate multi-buffering. Breakpoints are discussed in section 13.6.3.

13.6 Draw Synchronization Functions

The rendering pipeline consists of several asynchronous components. Among them are the CPU generating graphics commands, the GP consuming the commands and producing frame buffers, and the VI displaying the frame buffers. We provide several mechanisms to synchronize these components, allowing for various programming models (with different levels of complexity).

The CPU must be coordinated with the GP since not all of the graphics data goes through the FIFO. All the indexed data and texture data that the CPU provides must remain available until the GP has read it, after which it can be altered for the next frame or deleted as necessary. The GP must be coordinated with the VI so that the EFB is copied only to an inactive external frame buffer XFB, and so that VI will switch to scanning out the new XFB at the right time, freeing up the previously scanned-out XFB.

First, we describe the mechanisms that are available for synchronization. Then we show how to use the different mechanisms for various synchronization schemes with varying levels of complexity and efficiency.

13.6.1 GXDrawDone

We have mentioned `GXDrawDone` briefly already. `GXDrawDone` is actually a wrapper around two synchronization functions: `GXSetDrawDone` and `GXWaitDrawDone`. The former sends a draw-done token into the FIFO and flushes it, while the latter waits for the pipeline to flush and the token to appear at the bottom of the pipe. Instead of waiting for the token, you can also make use of a callback that occurs as a result of a draw-done interrupt. This callback runs with interrupts disabled, and thus must complete quickly. The function to set the callback routine also returns the old callback function pointer. The draw-done functions are summarized in Code 13-9.

Code 13-9 GXDrawDone Synchronization Commands

```
void GXDrawDone();
void GXSetDrawDone();
void GXWaitDrawDone();
typedef void (*GXDrawDoneCallback)(void);
GXDrawDoneCallback GXSetDrawDoneCallback(GXDrawDoneCallback cb);
```

Note: The `GXDrawDone` function may hang under certain conditions. If execution has hanged, you can try various ways to deal with the situation: attempt to reduce the number of write operations used, switch the order in which matrices are loaded, and so on. Another possibility is that the XF stall bug has occured. This is an XF bug that sometimes causes the GP to hang when rendered primitives are clipped immediately after a raster state command is sent. With the GX library, you can work around the XF stall bug by using the `GXSetMisc` (`GX_MT_XF_FLUSH` or `GX_XF_FLUSH_SAFE`) settings. However, these settings adversely affect speed and memory usage, so we recommend that you try them only when the XF stall bug has occurred.

13.6.2 GXDrawSync

In order to detect that the pipeline has completely rendered geometry, you can use the functions `GXSetDrawSync` and `GXReadDrawSync`. Use `GXSetDrawSync` to send a token (a 16-bit number of your choosing) down the pipeline after rendering the geometry. This token will be stored in a special token register when it reaches the bottom of the pipeline. Use `GXReadDrawSync` to poll the token register. When the token register value returned matches the token you sent, the geometry has been rendered completely.

It is also possible to receive an interrupt when the draw token reaches the bottom of the pipeline. The application can register a callback, using `GXSetDrawSyncCallback`, that will be called by the interrupt handler. The callback's argument is the value of the most-recently-encountered token. Since it is possible to miss tokens (because graphics processing does not stop while the callback is running), your code should be capable of deducing if any tokens have been missed (for example, by using monotonically increasing values).

The draw-sync mechanism is similar to the draw-done mechanism, with two major differences. First, draw-sync allows you to insert a numbered (16-bit) token into the pipe and read the token value when it reaches the pipe bottom. Second, draw-sync does not force the pipe to be flushed, and thus does not create a "bubble" of idle cycles within the pipe. The draw-sync functions are summarized below:

Code 13-10 GXDrawSync Synchronization Commands

```
void GXSetDrawSync(u16 token);
u16 GXReadDrawSync();
typedef void (*GXDrawSyncCallback)(u16 token);
GXDrawSyncCallback GXSetDrawSyncCallback(GXDrawSyncCallback cb);
```

13.6.3 FIFO Breakpoint

Sometimes it is useful to write two or more frames of graphics to the *same* FIFO. The breakpoint feature will cause GP FIFO reads to be disabled when the FIFO read pointer matches the breakpoint value. The breakpoint can be set using:

Code 13-11 GXEnableBreakPt

```
void GXEnableBreakPt( void* break_pt );
```

You can re-enable GP FIFO reads by using:

Code 13-12 GXDisableBreakPt

```
void GXDisableBreakPt( void );
```

For example, after writing frame A of graphics, read the current CPU FIFO write pointer using `GXGetFifoPtrs`. Make this the current breakpoint using `GXEnableBreakPt`. Continue writing frame B of graphics into the FIFO. GP FIFO reads will be disabled when the read pointer reaches the break point. The *readIdle* status can be polled for this event. The application can enable processing of frame B graphics by calling `GXDisableBreakPt`.

There is also a CPU interrupt that is associated with the break point. You can define a callback to be executed when this interrupt occurs. This callback will run with interrupts disabled, and thus it is necessary for the callback to run as quickly as possible. The callback can be set using `GXSetBreakPtCallback`, which also returns the previous callback:

Code 13-13 GXSetBreakPtCallback

```
typedef void (*GXBreakPtCallback)(void);

GXBreakPtCallback GXSetBreakPtCallback( GXBreakPtCallback cb );
```

For an illustration of using the same FIFO to manage two frames of graphics with breakpoints, see `mgt-fifo-brkpt` in the `gxdemo/Management` branch of the source tree.

13.6.4 Abort Frame

GX allows you to halt the GP and flush all commands currently in the FIFO up to the next break point or the end of the FIFO (if no break point is set). The command in Code 13-14 achieves this.

Code 13-14 GXAbortFrame

```
void GXAbortFrame( void );
```

This command resets all state in the GP. Textures loaded in texture memory are retained, but if a load was in progress, you must make sure it was not aborted (you should use draw syncs to verify this). When starting the next frame, you should send down new, complete state information (you should not assume that any state has been retained from the aborted frame).

13.6.5 VI Synchronization

The preceding functions synchronize the CPU with the GP. In order to synchronize the CPU with the video output, you may use the functions in Code 13-15.

Code 13-15 VI Synchronization Commands

```
void VIWaitForRetrace();
typedef void (*VIRetraceCallback)(u32 retraceCount);
VIRetraceCallback VISetPreRetraceCallback(VIRetraceCallback cb);
VIRetraceCallback VISetPostRetraceCallback(VIRetraceCallback cb);
```

The `VIWaitForRetrace` function suspends the current thread until a vertical retrace occurs. When the retrace does occur, an interrupt is sent to the CPU. The handler for this interrupt will first call the “pre” retrace callback. It will next update the VI hardware registers, and then it calls the “post” retrace callback. Finally, it wakes any threads that are waiting for retrace. The callback routines run with interrupts off, and therefore must complete quickly. The functions to set the callback routines also return the old callback function pointer. For more details on these functions, refer to the *Video Interface Library (VI)*.

13.7 Draw Synchronization Methods

13.7.1 Double-Buffering

The simplest programming model uses double-buffering and `GXDrawDone` to coordinate the CPU with the GP. Having the EFB and one XFB would seem to be enough to provide for double-buffering; however, due to synchronization issues, it is simpler to have two XFBs. This allows the copy operation to be performed after graphics rendering is finished without having to stall until vertical retrace occurs. The end-of-frame code sequence in Code 13-16 illustrates simple double-XFB synchronization.

Code 13-16 Double-Buffer Copy Synchronization

```
// ... draw the image in the EFB, then:
GXCopyDisp(xfb, GX_TRUE);
GXDrawDone();
VISetNextFrameBuffer(xfb);
VIFlush();
VIWaitForRetrace();
xfb = (xfb == xfb1) ? xfb2 : xfb1;
```

With only one XFB, the copy operation must be performed during vertical retrace. The copy operation can be completed in about 0.5 milliseconds, thus finishing before the next video field begins to display. Since the copy command must be issued through the FIFO, you must hold up the FIFO until vertical retrace occurs. In addition, the copy command must be guaranteed to issue immediately upon vertical retrace. The latter requirement means that the copy command must be issued during a vertical retrace interrupt callback. This method is illustrated in Code 13-17.

Code 13-17 Single-Buffer Copy Synchronization

```
// main loop:                               // vertical retrace interrupt "post" callback:
// ... draw the image in the EFB, then:
     if (do_copy) {
        GXDrawDone();                           GXCopyDisp(xfb, GX_TRUE);
        do_copy = GX_TRUE;                      GXFlush();
        VIWaitForRetrace();                     do_copy = GX_FALSE;
     }
```

13.7.2 Triple-Buffering

Both of the above methods idle the CPU and the GP while waiting for vertical retrace. You can avoid this idling by using triple-buffering. This allows the graphics pipeline to run without having to wait for video refresh. You can perform triple-buffering in various ways. We will describe a method that uses only two XFBs.

When you decide not to wait for draw-done and vertical retrace before continuing drawing, various complications arise. One is that indexed data and dynamic texture data must remain available until you know the GP is done with it. Another is that you may finish drawing two frames before VI has scanned out only one. In this case, the second frame will have nowhere to go since both XFBs will be occupied. The second copy must wait until the vertical retrace period. Fortunately, these problems can be solved by use of the synchronization commands provided.

The breakpoint must be used in order to hold back the frame buffer copy commands. The draw sync token must be used to detect when the copy commands have completed, and the vertical retrace callbacks must be used to coordinate the copies and buffer swaps. To see all of this in action, refer to the `mgt-triple-buf.c` demo, located in the `gxdemo/Management` branch of the source tree.

13.8 Graphics FIFO vs. Display List

Writing graphics commands to a command FIFO differs from writing graphics commands to a display list in several respects. When writing commands to a FIFO you may use `GXCallDisplayList` to call a display list. Display lists (bracketed by `GXBeginDisplayList`/`GXEndDisplayList`) may not themselves call a display list.

Using `GXSetGPFifo`, the application attaches a FIFO to the GP to enable processing of the FIFO's graphics commands. A display list is called using `GXCallDisplayList`. A FIFO may be attached to the CPU and the GP simultaneously. The CPU creates a display list, a call command is issued into a command FIFO, and the GP reads and processes the display list.

13.9 Notes About the Write-Gather Pipe

The CPU write-gather pipe is a mechanism for doing fast uncached writes to main memory. It consists of a 128-byte circular queue organized as four 32-byte cache lines. It collects together the writes to a single memory address and stores them in the queue. When a cache line is filled, it is scheduled to be written to memory while another cache line continues to absorb the writes. When combined with the graphics processor's FIFO mechanism, this allows fast writing to arbitrary areas of memory, provided that all the writes are 32-byte aligned.

Flushing the write-gather/FIFO mechanism is typically done in GX by writing 32 NOPs. In general, these NOPs are not cleared out whenever the write-gather is redirected (when using `GXBeginDisplayList` or `GXEndDisplayList`). The `GXResetWriteGatherPipe` function is a special-purpose function intended to be called after the `GXBeginDisplayList` function. This function ensures that no residual zeroes will be written to the start or end of display lists generated after this point. This guarantees a consistent display list length.

Since the write-gather pipe is a handy way to blast data into memory, a couple of extra APIs have been created to make this easy:

Code 13-18 APIs to Control the Write-Gather Pipe

```
volatile void*   GXRedirectWriteGatherPipe ( void * ptr );
void             GXRestoreWriteGatherPipe ();
```

The first API allows the write-gather pipe to be redirected to an arbitrary, 32-byte aligned address. It returns a pointer to the write-gather register to which the application should actually write the data. The second API restores the write-gather pipe to where it had been before it was redirected.

These APIs handle flushing differently than the rest of GX. They will clear out any extra zeros that were used to flush the write-gather pipe. Also, the restore function flushes by writing 31 zeros, thus avoiding the possibility of writing out one too many cache lines.

You cannot issue most GX commands while the write-gather pipe has been redirected.

13.10 GX Verify

The debug version of the GX library has a verify feature that can be used to check for certain state-setting errors. This checking happens when `GXBegin` is called. The APIs in Code 13-19 control this feature.

Code 13-19 APIs to Control Verification

```
typedef enum {
    GX_WARN_NONE,       // no warnings reported
    GX_WARN_SEVERE,     // reports only severest warnings
    GX_WARN_MEDIUM,     // reports severe and medium warnings
    GX_WARN_ALL         // reports any and all warning info
} GXWarningLevel;

void GXSetVerifyLevel( GXVerifyLevel level );

typedef void (*GXVerifyCallback)(GXVerifyLevel level,
                                 u32           id,
                                 char*         msg);

GXVerifyCallback GXSetVerifyCallback( GXVerifyCallback cb );
```

As you can see, you can control the level of checking that occurs. The higher the verification *level* is set, the longer it takes. Even setting the highest level of verification will not uncover all possible errors; there are still many kinds of errors for which the verification function does not check.

When the verification function finds an error, it calls the GX verification callback. There is a default callback function that simply prints the *level*, *id*, and *msg* out to the debug console. You may change the error-reporting behavior by substituting your own callback function using `GXSetVerifyCallback`.

14 Performance Metrics

Application developers can access internal performance counters in the Graphics Processor. Statistics gathered using the performance counters might be useful in tuning the application for the highest performance, or for making load-balancing decisions at runtime.

14.1 Types of Metrics

The GP metrics are grouped into different categories:

- GP front-end and texture-related metrics
- Vertex cache metrics
- Pixel metrics
- Memory metrics

A set of GX functions exists to handle each category of metrics. The functions themselves fall into three categories:

- Set functions choose exactly which metric to count.
- Read functions read the value of the counter.
- Clear functions clear the counter back to zero.

The pixel and memory metrics do not have any “set” functions since all of the available counters may be read at once.

14.2 GP Front-End and Texture-Related Metrics

The following functions are used to control the performance counters for various GP-related events:

Code 14-1 GP Metric Functions

```
void GXSetGPMetric( GXPerf0 perf0, GXPerf1 perf1 );
void GXReadGPMetric( u32* cnt0, u32* cnt1 );
void GXClearGPMetric( void );

//   macros to deal with counter 0 only:
void GXSetGP0Metric( GXPerf0 perf0 );
u32  GXReadGP0Metric( void );
void GXClearGP0Metric( void );

//   macros to deal with counter 1 only:
void GXSetGP1Metric( GXPerf1 perf1 );
u32  GXReadGP1Metric( void );
void GXClearGP1Metric( void );
```

There are two counters which are set, read, and cleared at the same time. The functions that deal with counters 0 or 1 are generally macros that set the desired counter and turn off the other one. You cannot clear one counter without clearing the other at the same time.

The subsequent sections of this chapter describe the various counters.

14.2.1 GP Counter 0 Details

GX_PERF0_VERTICES

This metric returns the number of vertices processed by the GP as measured by the transform engine (XF unit).

GX_PERF0_CLIP_VTX

Returns the number of vertices that were clipped by the GP.

GX_PERF0_CLIP_CLKS

Returns the number of GP clocks spent clipping.

The transform engine (XF) in the GP is a pipeline that has an input stage, parallel transform and lighting stages, and a “bottom of pipe” processor which merges the results of lighting and texture coordinate generation. The following performance counters measure how many GP cycles are spent in each stage of the XF.

GX_PERF0_XF_WAIT_IN

Measures how many cycles the XF has been waiting on input. If the XF is waiting a large percentage of the total time, it may indicate that the CPU is not supplying data fast enough to keep the GP busy.

GX_PERF0_XF_WAIT_OUT

Measures how many cycles the XF waits to send its output to the rest of the GP pipeline. If the XF cannot output, it may indicate that the GP is currently fill-rate limited.

GX_PERF0_XF_XFRM_CLKS

Indicates the number of cycles that the transform engine is busy.

GX_PERF0_XF_LIT_CLKS

Indicates the number of cycles that the lighting engine is busy.

GX_PERF0_XF_BOT_CLKS

Indicates the number of cycles that the bottom of the pipe is busy.

The XF contains various state registers that control its processing. The registers are normally set using various functions of the GX API. The following counters measure state-register accesses.

GX_PERF0_XF_REGLD_CLKS

Measures how many cycles are spent loading (writing to) XF registers.

GX_PERF0_XF_REGRD_CLKS

Measures how many cycles are spent reading XF state registers.

GX_PERF0_TRIANGLES*

The triangle metrics allow the counting of triangles under specific conditions or with specific attributes.

- GX_PERF0_TRIANGLES counts all triangles.
- GX_PERF0_TRIANGLES_CULLED counts triangles that failed the front/backface culling test.
- GX_PERF0_TRIANGLES_PASSED counts triangles that passed the front/backface culling test.
- GX_PERF0_TRIANGLES_SCISSORED counts the triangles that are scissored.
- GX_PERF0_TRIANGLES_*TEX count triangles based on the number of texture coordinates supplied.
- GX_PERF0_TRIANGLES_*CLR count triangles based on the number of colors supplied.

GX_PERF0_QUAD*

The quad metrics allow you to count the number of quads (2x2 pixels) the GP processes. The term coverage is used to indicate how many pixels in the quad are actually part of the triangle being rasterized. For example, a coverage of four means that all pixels in the quad intersect the triangle. A coverage of one indicates that only one pixel in the quad intersects the triangle.

- GX_PERF0_QUAD_0CVG indicates the number of quads having 0 coverage.
- GX_PERF0_NON0CVG counts the number of quads that have greater than zero coverage values.
- GX_PERF0_QUAD_[1-4]CVG counts the quads having the given coverage.
- GX_PERF0_AVG_QUAD_CNT indicates the average quad count (number of pixels covered divided by 4).

GX_PERF0_CLOCKS

GX_PERF0_CLOCKS counts the number of GP clocks that have elapsed since the previous call to GXReadGP0Metric.

GX_PERF0_NONE

This metric disables counting on GP counter 0 and clears the current count.

14.2.2 Counter 1 Details

GX_PERF1_TEXELS

This metric returns the number of texels processed by the GP.

GX_PERF1_TX_IDLE

Returns the number of clocks that the texture unit (TX) is idle.

GX_PERF1_TX_REGS

Returns the number of GP clocks spent writing to state registers in the TX unit.

GX_PERF1_TX_MEMSTALL

Returns the number of GP clocks the TX unit is stalled waiting for main memory.

GX_PERF1_TC_CHECK1_2
GX_PERF1_TC_CHECK3_4
GX_PERF1_TC_CHECK5_6
GX_PERF1_TC_CHECK7_8
GX_PERF1_TC_MISS

These metrics can be used to compute the texture cache (TC) miss rate. The TC_CHECK* parameters count how many texture cache lines are accessed for each pixel. In the worst case, for a mipmap, up to eight cache lines may be accessed to produce one textured pixel. GX_PERF1_TC_MISS counts how many of those accesses missed the texture cache. To compute the miss rate, calculate:

Equation 14-1 Miss-Rate Calculation

```
                         GX_PERF1_TC_MISS
-------------------------------------------------------------------------------------------
GX_TC_PERF1_TC_CHECK1_2+GX_PERF1_TC_CHECK3_4+GX_PERF1_TC_CHECK_5_6+GX_PERF1_TC_CHECK_7_8
```

GX_PERF1_VC_ELEMQ_FULL

Counts vertex cache stalls due to its element queue being full.

GX_PERF1_VC_MISSQ

Counts vertex cache stalls due to its miss queue being full.

GX_PERF1_VC_MEMREQ_FULL

Counts vertex cache stalls due to too many outstanding main memory requests.

GX_PERF1_VC_STATUS7

Counts vertex cache stalls due to too many elements in the element queue depending upon a single cache line.

GX_PERF1_VC_MISSREP_FULL

Counts vertex cache stalls due to a cache miss with all sets still in use (no replacement available).

GX_PERF1_VC_STREAMBUF_LOW

Counts vertex cache stalls due to the near-empty FIFO (streaming buffer) having priority over the vertex cache.

GX_PERF1_VC_ALL_STALLS

Counts all of the above-mentioned vertex cache stall conditions.

GX_PERF1_VERTICES

This metric returns the number of vertices processed by the GP as measured by the vertex cache.

GX_PERF1_FIFO_REQ

This metric counts the number of lines (32B) read from the GP FIFO.

GX_PERF1_CALL_REQ

This metric counts the number of lines (32B) read from called display lists (GXCallDisplayList).

GX_PERF1_VC_MISS_REQ

This metric counts the number of vertex cache miss requests. Each miss requests a 32B transfer from main memory.

GX_PERF1_CP_ALL_REQ

This metric counts all requests (32B/request) from the GP command processor (CP). It should be equal to the sum of counts returned by GX_PERF1_FIFO_REQ, GX_PERF1_CALL_REQ, and GX_PERF1_VC_MISS_REQ.

GX_PERF1_CLOCKS

GX_PERF1_CLOCKS counts the number of GP clocks that have elapsed since the previous call to GXReadGP1Metric.

GX_PERF1_NONE

This metric disables counting on GP counter 1 and clears the current count.

14.3 Using Performance Counters

The performance counter functions directly access GP registers, and thus they only work in immediate mode. However, they measure information that is sent through the GP FIFO, and so some synchronization is necessary in order to make sure the desired data is measured properly. The code sequence in Code 14-2 illustrates one possible approach.

Code 14-2 Counting a Metric

```
u32 metric1, metric2;

// Set desired metrics
GXSetGPMetric(GX_PERF0_VERTICES, GX_PERF1_TEXELS);

// Clear counters
GXClearGPMetric();

// Send down non-end draw-sync token
GXSetDrawSync(0x0);

// Draw Object(s)
...

// Send down ending draw-sync token, wait for it
GXSetDrawSync(0xbeef);
while (0xbeef != GXReadDrawSync())
  ;

// Read the counters
GXReadGPMetric(&metric1, &metric2);
OSReport("Number of verts: %d  texels: %d\n", metric1, metric2);
```

14.4 Vertex Cache Metrics

Use the functions in Code 14-3 to control the performance counters for vertex cache-related events.

Code 14-3 Vertex Cache Metric Functions

```
void GXSetVCacheMetric( GXVCachePerf attr );
void GXReadVCacheMetric( u32* check, u32* miss, u32* stall );
void GXClearVCacheMetric( void );
```

The “set” function allows you to choose which vertex attribute will be measured. You can choose a value of `GX_VC_ALL` in order to measure all of the attributes at once. For any given attribute selection, three metrics are available:

- *check* indicates the total number of accesses to the vertex cache.
- *miss* indicates the total number of misses when accessing the vertex cache.
- *stall* indicates the number of GP clocks the GP is stalled waiting on the vertex cache.

The stall count measures how often the command processor of the GP must wait on cache requests being filled.

14.5 Pixel Metrics

The functions in Code 14-4 are used to control the performance counters for pixel-related events.

Code 14-4 Pixel Metric Functions

```
void GXReadPixMetric( u32* top_pixels_in,
                      u32* top_pixels_out,
                      u32* bot_pixels_in,
                      u32* bot_pixels_out,
                      u32* clr_pixels_in,
                      u32* copy_clks );
void GXClearPixMetric( void );
```

The GP can be configured to Z-buffer before or after texture lookup (see `GXSetZCompLoc`). The parameter *top_pixels_in* returns the number of pixels entering the Z compare before texture lookup. The parameter *top_pixels_out* indicates how many pixels passed this Z compare test.

The parameter *bot_pixels_in* counts the number of pixels entering the Z compare after texture lookup. The parameter *bot_pixels_out* indicates how many pixels passed this Z compare test.

The parameter *clr_pixels_in* counts the number of pixels processed by the blend unit in the last stage of the pipeline. This is normally the sum of *top_pixels_out* and *bot_pixels_out*.

The parameter *copy_clks* counts the number of GP clocks spent on copy operations, either from the EFB to a texture (see `GXCopyTex`) or from the EFB to a display buffer (see `GXCopyDisp`).

14.6 Memory Metrics

The functions in Code 14-5 are used to control the performance counters for memory-related events.

Code 14-5 Memory Metric Functions

```
void GXReadMemMetric( u32*           cp_req,
                      u32*           tc_req,
                      u32*           cpu_rd_req,
                      u32*           cpu_wr_req,
                      u32*           dsp_req,
                      u32*           io_req,
                      u32*           vi_req,
                      u32*           pe_req,
                      u32*           rf_req,
                      u32*           fi_req  );
void GXClearMemMetric( void );
```

The various metrics are explained in the following table:

Table 14-1 Memory Metrics

Metric	Purpose
`cp_req`	The command processor (CP) is responsible for reading the Graphics FIFO, reading display lists (`GXCallDisplayList`), and servicing vertex cache misses. This metric returns the number of memory read requests issued by the CP.
`tc_req`	Returns the number of memory read requests issued by the Texture Cache (TC).
`cpu_rd_req`	Returns the number of memory read requests made by the CPU.
`cpu_wr_req`	Returns the number of memory write requests made by the CPU.
`dsp_req`	Returns the number of memory requests made by the Audio DSP.
`io_req`	Returns the number of memory requests made by IO devices.
`vi_req`	Returns the number of memory read requests made by the Video Interface (VI).
`pe_req`	Returns the number of memory write requests made by the Pixel Engine (PE). These include texture copies (`GXCopyTex`) and display copies (`GXCopyDisp`).
`rf_req`	Returns the number of memory refresh requests.
`fi_req`	Returns the number of *Forced Idle* (FI) requests, which are dummy requests required to switch the bus direction (i.e., read to write, or write to read).

15 Limitations

This chapter outlines features that disable other features, or only work in restricted cases.

15.1 Antialiasing

- Antialiasing can use pixel format `GX_PF_RGB565_Z16` only.
- At peak, antialiased rendering runs at half the maximum fill rate of non-antialiased rendering, or 486 megapixels/second. (Beyond this reduction in the peak rate, the formula for computing the antialiased rendering speed is the same as for non-antialiased rendering; see "12 Video Output" on page 129 for more details.)
- Z textures cannot be copied from an antialiased frame buffer.

15.2 CPU Access to the Frame Buffer

The application must synchronize CPU access to the Embedded Frame Buffer (EFB) with normal rendering to the EFB.

15.3 Display Lists

When creating a display list at runtime by calling GX functions bracketed by `GXBeginDisplayList/GXEndDisplayList`, the following functions may not be used:

- `GXBeginDisplayList`
- `GXEndDisplayList`
- `GXCallDisplayList`

In addition, the following types of functions cannot be placed inside of a display list:

- `GXInit*`
- `GXRead*`
- `GXPeekARGB/GXPeekZ`
- `GXPokeARGB/GXPokeZ`
- `GXGet*`

These may be executed while a display list is being created; however, they will not be put into the display list.

15.4 Vertex Performance

Vertex performance depends on the lighting and texture coordinate features selected. See the “Vertex Performance Calculator” page in the *Revolution Function Reference Manual* (HTML).

15.5 Matrix Memory

Position and texture matrices share the same internal matrix memory. Normal matrices are stored in a separate memory.

15.6 Texture

Color index textures cannot be trilinearly-filtered.

Mipmaps using texel format `GX_TF_RGBA8` require two cycles to filter. These cycles are internal to the texture filter hardware and do not effect the total number of TEV stages available.

15.7 Blending and Logic Operations

You must choose between blending and logical operations; you cannot do both at the same time.

15.8 Sharing Main Memory Resources

While the embedded frame buffer (EFB) eases the bandwidth requirements of the main memory, there are still several large main memory bandwidth hogs in the system, including the CPU, vertex input to the GP, texture fetches, and copies from the EFB.

Appendix A. GX API Functions

The GX API list has been removed from the programming manual. Please refer to the GX pages in the *Revolution Function Reference Manual*.

Appendix B. GXInit Defaults

This appendix lists the state set by `GXInit`.

Code B-1 GXInit Defaults

```
//  Color definitions

#define GX_DEFAULT_BG    {64, 64, 64, 255}
#define BLACK            {0, 0, 0, 0}
#define WHITE            {255, 255, 255, 255}

    //
    //  Render Mode
    //
    //  (set 'rmode' based upon VIGetTvFormat(); code not shown)

    //
    //  Geometry and Vertex
    //
    GXSetTexCoordGen(GX_TEXCOORD0, GX_TG_MTX2x4, GX_TG_TEX0, GX_IDENTITY);
    GXSetTexCoordGen(GX_TEXCOORD1, GX_TG_MTX2x4, GX_TG_TEX1, GX_IDENTITY);
    GXSetTexCoordGen(GX_TEXCOORD2, GX_TG_MTX2x4, GX_TG_TEX2, GX_IDENTITY);
    GXSetTexCoordGen(GX_TEXCOORD3, GX_TG_MTX2x4, GX_TG_TEX3, GX_IDENTITY);
    GXSetTexCoordGen(GX_TEXCOORD4, GX_TG_MTX2x4, GX_TG_TEX4, GX_IDENTITY);
    GXSetTexCoordGen(GX_TEXCOORD5, GX_TG_MTX2x4, GX_TG_TEX5, GX_IDENTITY);
    GXSetTexCoordGen(GX_TEXCOORD6, GX_TG_MTX2x4, GX_TG_TEX6, GX_IDENTITY);
    GXSetTexCoordGen(GX_TEXCOORD7, GX_TG_MTX2x4, GX_TG_TEX7, GX_IDENTITY);
    GXSetNumTexGens(1);
    GXClearVtxDesc();
    GXInvalidateVtxCache();

    GXSetLineWidth(6, GX_TO_ZERO);
    GXSetPointSize(6, GX_TO_ZERO);
    GXEnableTexOffsets( GX_TEXCOORD0, GX_DISABLE, GX_DISABLE );
    GXEnableTexOffsets( GX_TEXCOORD1, GX_DISABLE, GX_DISABLE );
    GXEnableTexOffsets( GX_TEXCOORD2, GX_DISABLE, GX_DISABLE );
    GXEnableTexOffsets( GX_TEXCOORD3, GX_DISABLE, GX_DISABLE );
    GXEnableTexOffsets( GX_TEXCOORD4, GX_DISABLE, GX_DISABLE );
    GXEnableTexOffsets( GX_TEXCOORD5, GX_DISABLE, GX_DISABLE );
    GXEnableTexOffsets( GX_TEXCOORD6, GX_DISABLE, GX_DISABLE );
    GXEnableTexOffsets( GX_TEXCOORD7, GX_DISABLE, GX_DISABLE );

    //
    //  Transformation and Matrix
    //
    //  (initialize 'identity_mtx' to identity; code not shown)

    //  Note: projection matrix is not initialized!
    GXLoadPosMtxImm(identity_mtx, GX_PNMTX0);
    GXLoadNrmMtxImm(identity_mtx, GX_PNMTX0);
    GXSetCurrentMtx(GX_PNMTX0);
    GXLoadTexMtxImm(identity_mtx, GX_IDENTITY, GX_MTX3x4);
    GXLoadTexMtxImm(identity_mtx, GX_PTIDENTITY, GX_MTX3x4);
    GXSetViewport(0.0F,                    // left
                  0.0F,                    // top
                  (float)rmode->fbWidth,   // width
                  (float)rmode->xfbHeight, // height
                  0.0F,                    // nearz
                  1.0F);                   // farz

    //
    //  Clipping and Culling
    //
```

```
GXSetCoPlanar(GX_DISABLE);
GXSetCullMode(GX_CULL_BACK);
GXSetClipMode(GX_CLIP_ENABLE);
GXSetScissor(0, 0, (u32)rmode->fbWidth, (u32)rmode->efbHeight);
GXSetScissorBoxOffset(0, 0);

//
//  Lighting - pass vertex color through
//
GXSetNumChans(0); // no colors by default

GXSetChanCtrl(
    GX_COLOR0A0,
    GX_DISABLE,
    GX_SRC_REG,
    GX_SRC_VTX,
    GX_LIGHT_NULL,
    GX_DF_NONE,
    GX_AF_NONE );

GXSetChanAmbColor(GX_COLOR0A0, BLACK);
GXSetChanMatColor(GX_COLOR0A0, WHITE);

GXSetChanCtrl(
    GX_COLOR1A1,
    GX_DISABLE,
    GX_SRC_REG,
    GX_SRC_VTX,
    GX_LIGHT_NULL,
    GX_DF_NONE,
    GX_AF_NONE );

GXSetChanAmbColor(GX_COLOR1A1, BLACK);
GXSetChanMatColor(GX_COLOR1A1, WHITE);

//
//  Texture
//
GXInvalidateTexAll();

// Allocate 8 32k caches for RGBA texture mipmaps.
// Equal size caches to support 32b RGBA textures.
//
// (code not shown)

// Allocate color index caches in low bank of TMEM.
// Each cache is 32kB.
// Even and odd regions should be allocated on different address.
//
// (code not shown)

//  Allocate TLUTs, 16 256-entry TLUTs and 4 1K-entry TLUTs.
//  256-entry TLUTs are 8kB, 1k-entry TLUTs are 32kB.
//
// (code not shown)

//
//  Set texture region and tlut region Callbacks
//
GXSetTexRegionCallback(__GXDefaultTexRegionCallback);
GXSetTlutRegionCallback(__GXDefaultTlutRegionCallback);

//
//  Texture Environment
//
```

```
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD0, GX_TEXMAP0, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD1, GX_TEXMAP1, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE2, GX_TEXCOORD2, GX_TEXMAP2, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE3, GX_TEXCOORD3, GX_TEXMAP3, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE4, GX_TEXCOORD4, GX_TEXMAP4, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE5, GX_TEXCOORD5, GX_TEXMAP5, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE6, GX_TEXCOORD6, GX_TEXMAP6, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE7, GX_TEXCOORD7, GX_TEXMAP7, GX_COLOR0A0);
GXSetTevOrder(GX_TEVSTAGE8, GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR_NULL);
GXSetTevOrder(GX_TEVSTAGE9, GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR_NULL);
GXSetTevOrder(GX_TEVSTAGE10,GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR_NULL);
GXSetTevOrder(GX_TEVSTAGE11,GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR_NULL);
GXSetTevOrder(GX_TEVSTAGE12,GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR_NULL);
GXSetTevOrder(GX_TEVSTAGE13,GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR_NULL);
GXSetTevOrder(GX_TEVSTAGE14,GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR_NULL);
GXSetTevOrder(GX_TEVSTAGE15,GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR_NULL);
GXSetNumTevStages(1);
GXSetTevOp(GX_TEVSTAGE0, GX_REPLACE);
GXSetAlphaCompare(GX_ALWAYS, 0, GX_AOP_AND, GX_ALWAYS, 0);
GXSetZTexture(GX_ZT_DISABLE, GX_TF_Z8, 0);

for (i = GX_TEVSTAGE0; i < GX_MAX_TEVSTAGE; i++) {
    GXSetTevKColorSel((GXTevStageID) i, GX_TEV_KCSEL_1_4 );
    GXSetTevKAlphaSel((GXTevStageID) i, GX_TEV_KASEL_1 );
    GXSetTevSwapMode ((GXTevStageID) i, GX_TEV_SWAP0, GX_TEV_SWAP0 );
}
GXSetTevSwapModeTable(GX_TEV_SWAP0,
                      GX_CH_RED, GX_CH_GREEN, GX_CH_BLUE, GX_CH_ALPHA);
GXSetTevSwapModeTable(GX_TEV_SWAP1,
                      GX_CH_RED, GX_CH_RED, GX_CH_RED, GX_CH_ALPHA);
GXSetTevSwapModeTable(GX_TEV_SWAP2,
                      GX_CH_GREEN, GX_CH_GREEN, GX_CH_GREEN, GX_CH_ALPHA);
GXSetTevSwapModeTable(GX_TEV_SWAP3,
                      GX_CH_BLUE, GX_CH_BLUE, GX_CH_BLUE, GX_CH_ALPHA);

// Indirect Textures.
for (i = GX_TEVSTAGE0; i < GX_MAX_TEVSTAGE; i++) {
    GXSetTevDirect((GXTevStageID) i);
}
GXSetNumIndStages(0);
GXSetIndTexCoordScale( GX_INDTEXSTAGE0, GX_ITS_1, GX_ITS_1 );
GXSetIndTexCoordScale( GX_INDTEXSTAGE1, GX_ITS_1, GX_ITS_1 );
GXSetIndTexCoordScale( GX_INDTEXSTAGE2, GX_ITS_1, GX_ITS_1 );
GXSetIndTexCoordScale( GX_INDTEXSTAGE3, GX_ITS_1, GX_ITS_1 );

//
//  Pixel Processing
//
GXSetFog(GX_FOG_NONE, 0.0F, 1.0F, 0.1F, 1.0F, BLACK);
GXSetFogRangeAdj( GX_DISABLE, 0, 0 );
GXSetBlendMode(GX_BM_NONE,
               GX_BL_SRCALPHA,    // src factor
               GX_BL_INVSRCALPHA, // dst factor
               GX_LO_CLEAR);
GXSetColorUpdate(GX_ENABLE);
GXSetAlphaUpdate(GX_ENABLE);
GXSetZMode(GX_TRUE, GX_LEQUAL, GX_TRUE);
GXSetZCompLoc(GX_TRUE); // before texture
GXSetDither(GX_ENABLE);
GXSetDstAlpha(GX_DISABLE, 0);
GXSetPixelFmt(GX_PF_RGB8_Z24, GX_ZC_LINEAR);
GXSetFieldMask( GX_ENABLE, GX_ENABLE );
GXSetFieldMode((GXBool)(rmode->field_rendering),
               ((rmode->viHeight == 2*rmode->xfbHeight) ?
                GX_ENABLE : GX_DISABLE));
```

```
//
//  Framebuffer
//
GXSetCopyClear(GX_DEFAULT_BG, GX_MAX_Z24);
GXSetDispCopySrc(0, 0, rmode->fbWidth, rmode->efbHeight);
GXSetDispCopyDst(rmode->fbWidth, rmode->efbHeight);
GXSetDispCopyYScale((f32)(rmode->xfbHeight) / (f32)(rmode->efbHeight));
GXSetCopyClamp((GXFBClamp)(GX_CLAMP_TOP | GX_CLAMP_BOTTOM));
GXSetCopyFilter(rmode->aa, rmode->sample_pattern, GX_TRUE, rmode->vfilter);
GXSetDispCopyGamma( GX_GM_1_0 );
GXSetDispCopyFrame2Field(GX_COPY_PROGRESSIVE);
GXClearBoundingBox();

//
//  CPU direct EFB access
//
GXPokeColorUpdate(GX_TRUE);
GXPokeAlphaUpdate(GX_TRUE);
GXPokeDither(GX_FALSE);
GXPokeBlendMode(GX_BM_NONE, GX_BL_ZERO, GX_BL_ONE, GX_LO_SET);
GXPokeAlphaMode(GX_ALWAYS, 0);
GXPokeAlphaRead(GX_READ_FF);
GXPokeDstAlpha(GX_DISABLE, 0);
GXPokeZMode(GX_TRUE, GX_ALWAYS, GX_TRUE);

//
//  Performance Counters
//
GXSetGPMetric(GX_PERF0_NONE, GX_PERF1_NONE);
GXClearGPMetric();
```

Appendix C. Display List Format

Display list commands are listed in <revolution/gx/GXCommandList.h>. The graphics processor (GP) can interpret the described display list format directly. The bulk of a display list is expected to contain information for describing primitives and their vertices. The display list format contains only limited state commands. Most state commands, such as the vertex descriptor or vertex attribute format, should be set by the appropriate GX API functions prior to calling the display list. We describe the format of certain state-setting commands below.

Display lists must be 32-byte aligned in main memory and be a multiple of 32 bytes long. Display lists can be padded with `GX_NOP` commands to fill out a 32-byte line.

Display lists are executed by calling the `GXCallDisplayList` function with a pointer to the display list and the number of bytes in the display list.

Note: There is no “end of display list” token, since you are providing an explicit length in the call.

C.1 Display List Opcodes

A display list consists of a stream of commands (opcodes) followed by their associated data. There can be any number of commands in any sensible sequence within a display list.

Table C-1 Display List Opcodes

Opcode Name	Opcode Bits[7:0]	Next Field	Followed By
GX_DRAW_QUADS	10000VatIdx[2:0]	VertexCount[15:0]	Vertex data stream
GX_DRAW_TRIANGLES	10010VatIdx[2:0]	VertexCount[15:0]	Vertex data stream
GX_DRAW_TRIANGLE_STRIP	10011VatIdx[2:0]	VertexCount[15:0]	Vertex data stream
GX_DRAW_TRIANGLE_FAN	10100VatIdx[2:0]	VertexCount[15:0]	Vertex data stream
GX_DRAW_LINES	10101VatIdx[2:0]	VertexCount[15:0]	Vertex data stream
GX_DRAW_LINE_STRIP	10110VatIdx[2:0]	VertexCount[15:0]	Vertex data stream
GX_DRAW_POINTS	10111VatIdx[2:0]	VertexCount[15:0]	Vertex data stream
GX_LOAD_BP_REG	01100001 (0x61)	Register[31:0]	none
GX_NOP	00000000	None	none

Note: VatIdx[2:0] is the 3-bit Vertex Attribute Format Table index. This format will be used to interpret the vertex data arrays.

VertexCount[15:0] is a 16-bit count of the number of vertices to follow this command.

C.2 Attribute Order Requirements

The current Vertex Descriptor (`GXSetVtxDesc`) is used to indicate the number and type of data or indices. Recall that indices may be 8 bits or 16 bits. The current attribute array's (`GXSetArray`) base pointers and strides are used for referencing indexed data. These should all be set before executing the display list.

Table C-2 specifies the required order of attribute values (immediate data or indices) in the display list. The order is identical to that specified for immediate mode primitives drawn using `GXBegin`/`GXEnd`.

Table C-2 Vertex Index Stream Order Requirements

Order	Attribute
0	GX_VA_POSMATIDX
1	GX_VA_TEX0MTXIDX
2	GX_VA_TEX1MTXIDX
3	GX_VA_TEX2MTXIDX
4	GX_VA_TEX3MTXIDX
5	GX_VA_TEX4MTXIDX
6	GX_VA_TEX5MTXIDX
7	GX_VA_TEX6MTXIDX
8	GX_VA_TEX7MTXIDX
9	GX_VA_POS
10	GX_VA_NRM
11	GX_VA_COLOR0
12	GX_VA_COLOR1
13	GX_VA_TEXCOORD0
14	GX_VA_TEXCOORD1
15	GX_VA_TEXCOORD2
16	GX_VA_TEXCOORD3
17	GX_VA_TEXCOORD4
18	GX_VA_TEXCOORD5
19	GX_VA_TEXCOORD6
20	GX_VA_TEXCOORD7

C.3 Example Display List (primitives only)

Table C-3 provides an example display list as a list of hexadecimal numbers. This display list could be called using "Code C-2 Code Necessary to Utilize Example Display List" on page 170.

Note: Since the display list only contains indices to attributes, the formats of the attributes could be changed without affecting the display list. However, the attribute format must be described accurately (the format of the data must match the format described) in order for the display list to be drawn correctly.

Table C-3 Example Display List

Description	Data
GX_DRAW_TRIANGLES, GX_VTXFMT0	0x90
number of verts = 6	0x0006
pos_indx (8b)	0x00
norm_indx (8b)	0x10
tex_coord_0 (16b)	0x0011
pos_indx (8b)	0x01
norm_indx (8b)	0x11
tex_coord_0 (16b)	0x0012
pos_indx (8b)	0x02
norm_indx (8b)	0x12
tex_coord_0 (16b)	0x0013
pos_indx (8b)	0x03
norm_indx (8b)	0x13
tex_coord_0 (16b)	0x0014
pos_indx (8b)	0x04
norm_indx (8b)	0x14
tex_coord_0 (16b)	0x0015
pos_indx (8b)	0x05
norm_indx (8b)	0x15

Table C-3 Example Display List

tex_coord_0 (16b)	0x0016
no_op	0x00
no_op	0x00
no_op	0x00
no_op	0x00
no_op, pad to 32B	0x00

Note: The display list described in "Table C-3 Example Display List" on page 169 assumes the code sequence shown in Code C-2 to execute.

Code C-2 Code Necessary to Utilize Example Display List

```
GXClearVtxDesc();
GXSetVtxDesc( GX_VA_POS, GX_INDEX8 );
GXSetVtxDesc( GX_VA_NRM, GX_INDEX8 );
GXSetVtxDesc( GX_VA_TEX0, GX_INDEX16 );
GXSetVtxAttrFmt( GX_VTXFMT0, GX_VA_POS, GX_POS_XYZ, GX_F32, 0 );
GXSetVtxAttrFmt( GX_VTXFMT0, GX_VA_NRM, GX_NRM_XYZ, GX_S8, 6 );
GXSetVtxAttrFmt( GX_VTXFMT0, GX_VA_TEX0, GX_TEX_ST, GX_U16, 5 );
GXSetArray( GX_VA_POS, &mypos, sizeof(f32)*3 );
GXSetArray( GX_VA_NRM, &mynrm, sizeof(s8)*3 );
GXSetArray( GX_VA_TEX0, &mytex, sizeof(u16)*2 );
GXCallDisplayList( &Example_Display_List, 32 );
```

C.4 State Commands

Inserting state commands into display lists requires a certain amount of coordination between the state that is set by immediate-mode API functions and the state that is set by the display list. You should keep in mind that various types of conflicts could arise.

The first state commands described here are those related to loading texture objects. These “commands” are all implemented by setting registers within the GP. Before describing these registers, we explain how the GX API loads texture objects.

Loading a texture object via `GXLoadTexObj` performs the following steps:

1. GX calls the texture region callback in order to obtain a region of TMEM to use with the texture.
2. GX records the desired texture ID into the texture object, which consists of GP texture registers.
3. The texture registers are written into the FIFO.
4. If this is a color-indexed texture, GX calls the TLUT region callback to obtain the TMEM address for the TLUT. This is encoded into a register in the texture object and written into the FIFO.
5. GX sets a flag to indicate that the texture coordinate scaling registers need to be updated.

Calling `GXLoadTexObjPreLoaded` is similar, except that step 1 above is omitted. In either case, step 5 is necessary, because the GP requires that the texture coordinates be scaled according to the texture that they will look up. Prior to any geometry being drawn, the update flag is examined and if set, a function is called to update the scaling registers (and the flag is then cleared). The update function behaves as follows:

1. If all texture coordinates are being scaled manually, it simply returns.
2. Loop over the indirect stages:
   For all non-manual coordinates, it sets the scale register according to the associated map size.
3. Loop over the normal TEV stages:
   For all non-manual coordinates, it sets the scale register according to the associated map size.
4. It sets the texture coordinate range bias appropriately as each scale register is adjusted.

Notes:

- If a texture coordinate is associated with both an indirect stage and a normal TEV stage, the coordinate is scaled according to the size of the texture for the normal TEV stage.
- `GXSetTevOrder` and `GXSetIndTexOrder` will also set the update flag.

Given the preceding information, we should emphasize that putting texture object-loading commands into a display list requires that all of the steps be done manually. In particular, you may wish to manage TMEM according to your own scheme. You should also consider calling `GXSetTexCoordScaleManually` for all of the texture coordinates.

The subsequent sections detail the display list commands (registers) for loading textures. In order to send any of these registers down, you must send down the following sequence for each register:

`0x61`	(1 byte)	`GX_LOAD_BP_REG`, the register load command.
`xxxx`	(4 bytes, big-endian)	The actual register as specified below.

Unmentioned register bits are reserved and should be set to 0.

C.4.1 Set_TextureMode0

Indicates texture lookup and filtering modes.

Code C-3 Set_TextureMode0

```
Bits  Content          Values
----- ---------------- -----------------------------------------------------
0-1   wrap_s           0: clamp
                       1: repeat
                       2: mirror
                       3: reserved
2-3   wrap_t           same values as wrap_s
4     mag_filter       0: near
                       1: linear
5-7   min_filter       0: near                    4: linear
                       1: near_mip_near           5: lin_mip_near
                       2: near_mip_lin            6: lin_mip_lin
                       3: reserved                7: reserved
8     diag_lod_enable  0: use edge LOD            1: use diagonal LOD
9-16  lod_bias         S2.5 (-4.0:3.99) (2's complement format)
19-20 max_aniso        0: 1
                       1: 2 (requires edge LOD)
                       2: 4 (requires edge LOD)
21    lod_clamp        0: off
      (bias_clamp)     1: on
24-31 opcode           0x80 + GXTexMapID (id <= 3)
                       0xa0 + GXTexMapID (id >= 4)
```

C.4.2 Set_TextureMode1

Indicates min/max LOD info.

Code C-4 Set_TextureMode1

```
Bits  Content          Values
----- ---------------- -----------------------------------------------------
0-7   min_lod          U4.4 (0:10.0)
8-15  max_lod          U4.4 (0:10.0)
24-31 opcode           0x84 + GXTexMapID (id <= 3)
                       0xa4 + GXTexMapID (id >= 4)
```

C.4.3 Set_TextureImage0

Indicates texture width, height, and format.

Code C-5 Set_TextureImage0

```
Bits  Content          Values
----- ---------------- -----------------------------------------------------
0-9   image_width      U10 (0:1023) value is (real width) - 1
10-19 image_height     U10 (0:1023) value is (real height) - 1
20-23 image_format     0: I4         4: RGB565     8: C4        12: reserved
                       1: I8         5: RGB5A3     9: C8        13: reserved
                       2: IA4        6: RGBA8     10: C14X2     14: CMP
                       3: IA8        7: reserved  11: reserved  15: reserved
24-31 opcode           0x88 + GXTexMapID (id <= 3)
                       0xa8 + GXTexMapID (id >= 4)
```

C.4.4 Set_TextureImage1

Indicates where even LODs are stored (or cached) in TMEM. This data normally comes from a `GXTexRegion` object.

Code C-6 Set_TextureImage1

```
Bits  Content          Values
----- ---------------- -----------------------------------------------------
0-14  tmem_offset      (address of even LODs in TMEM) >> 5
15-17 cache_width      3: 32KB
                       4: 128KB
                       5: 512KB
17-19 cache_height     must be equal to cache_width
20    image_type       0: cached
                       1: preloaded
24-31 opcode           0x8c + GXTexMapID (id <= 3)
                       0xac + GXTexMapID (id >= 4)
```

C.4.5 Set_TextureImage2

Indicates where odd LODs are stored in TMEM (unused by most planar textures). This data normally comes from a `GXTexRegion` object.

Code C-7 Set_TextureImage2

```
Bits  Content          Values
----- ---------------- -----------------------------------------------------
0-14  tmem_offset      (address of odd LODs in TMEM) >> 5
15-17 cache_width      3: 32KB
                       4: 128KB
                       5: 512KB
                       0: none (only where odd is not used)
17-19 cache_height     must be equal to cache_width
24-31 opcode           0x90 + GXTexMapID (id <= 3)
                       0xb0 + GXTexMapID (id >= 4)
```

C.4.6 Set_TextureImage3

For cached textures, where the texture is found in main memory.

Code C-8 Set_TextureImage3

```
Bits  Content          Values
----- ---------------- -----------------------------------------------------
0-20  image_base       (PHYSICAL address of texture in main memory) >> 5
24-31 opcode           0x94 + GXTexMapID (id <= 3)
                       0xb4 + GXTexMapID (id >= 4)
```

C.4.7 Set_TextureTLUT

For color-index textures, where the TLUT is found in the high TMEM bank.

Code C-9 Set_TextureTLUT

```
Bits  Content          Values
----- ---------------- -----------------------------------------------------
0-9   tmem_offset      (offset of TLUT from start of high bank in TMEM) >> 5
10-11 tlut_format      0: IA8  1: RGB565  2: RGB5A3
24-31 opcode           0x98 + GXTexMapID (id <= 3)
                       0xb8 + GXTexMapID (id >= 4)
```

C.4.8 SU_TS0

Indicates the texture coordinate scaling (*s* component). The point/line offset is normally set by `GXEnableTexOffsets`.

Code C-10 SU_TS0

```
Bits  Content          Values
----- ---------------- -----------------------------------------------------
0-15  ssize            U16  (s scale value - 1) for the specified texcoord
16    bs               Enables range bias for s (used with GX_REPEAT only)
17    ws               Enables cylindrical texcoord wrapping for s
18    lf               Enables texcoord offset for lines using this texcoord
19    pf               Enables texcoord offset for points using this texcoord
24-31 opcode           0x30 + GXTexCoordID * 2
```

C.4.9 SU_TS1

Indicates the texture coordinate scaling (t component).

Code C-11 SU_TS1

```
Bits  Content          Values
----- ---------------- -----------------------------------------------------
0-15  tsize            U16  (t scale value - 1) for the specified texcoord
16    bt               Enables range bias for t (used with GX_REPEAT only)
17    wt               Enables cylindrical texcoord wrapping for t
24-31 opcode           0x31 + GXTexCoordID * 2
```

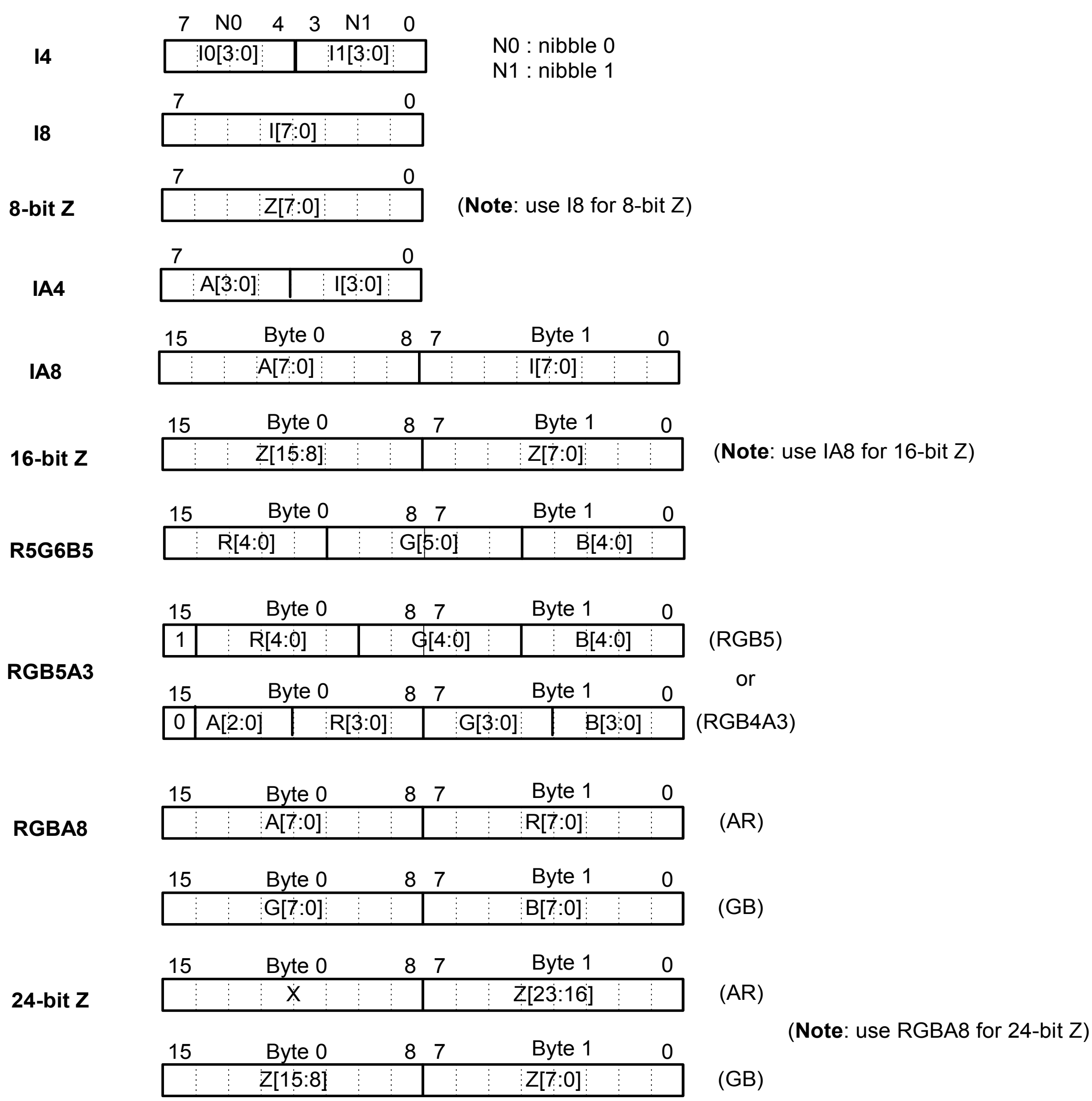
Appendix D. Wii Texture Formats

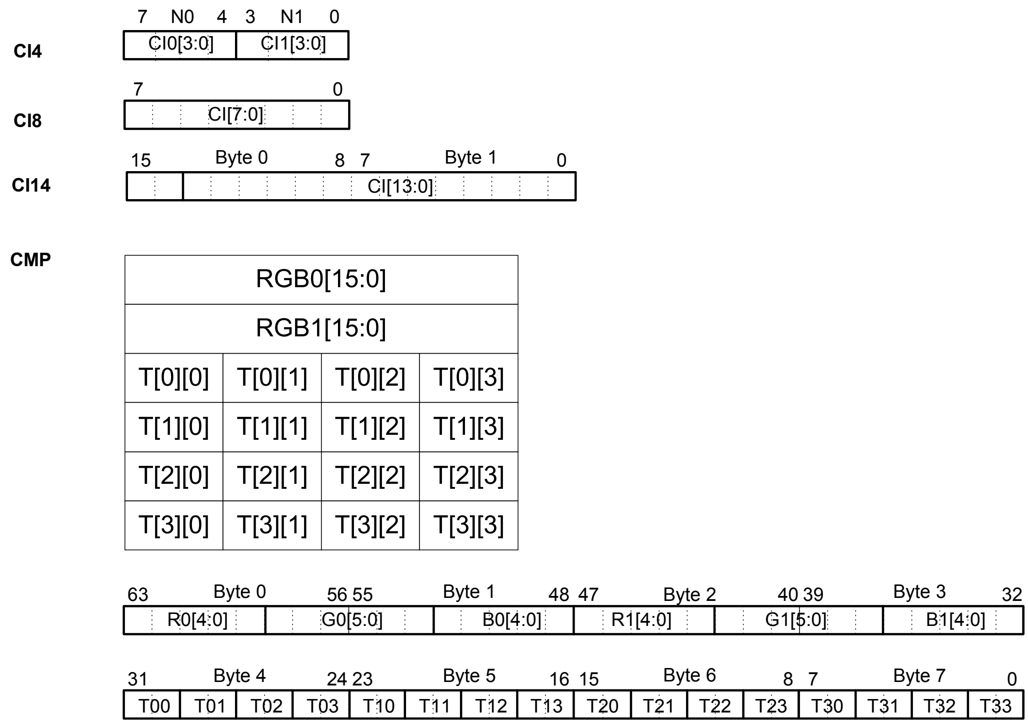
This appendix describes the Wii texture formats. The first section describes the bit ordering within a *texel*. The next section describes how texels are arranged in *tiles*. The texture cache hardware fetches texels in tile units. The final section describes how tiles are organized into *images*.

D.1 Texel Formats

The texturing hardware supports 14 native texture image formats specified by register `image_format[3:0]`. Supported types are Intensity, Intensity Alpha, RGB, RGBA, Color Index, Compressed, and Z. Supported texel sizes are 4-bit, 8-bit, 16-bit, and 32-bit. The following figures illustrate the packing of texel components.

Figure D-1 Texel Formats



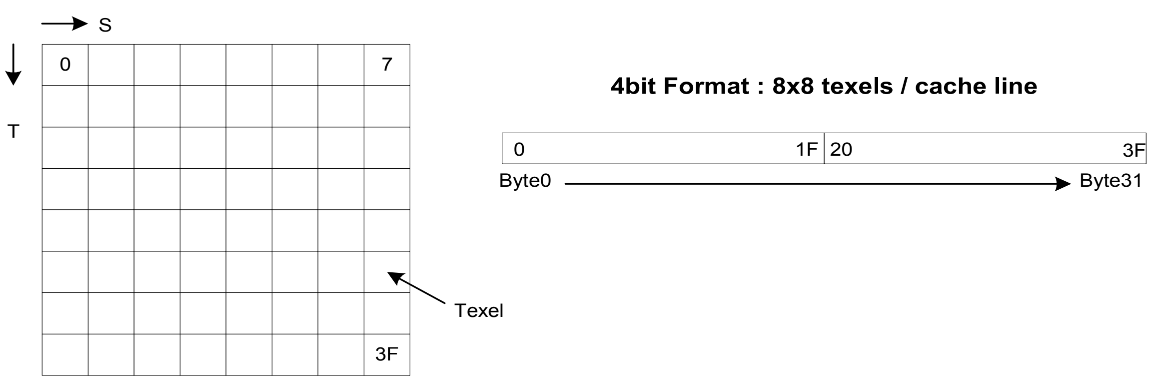


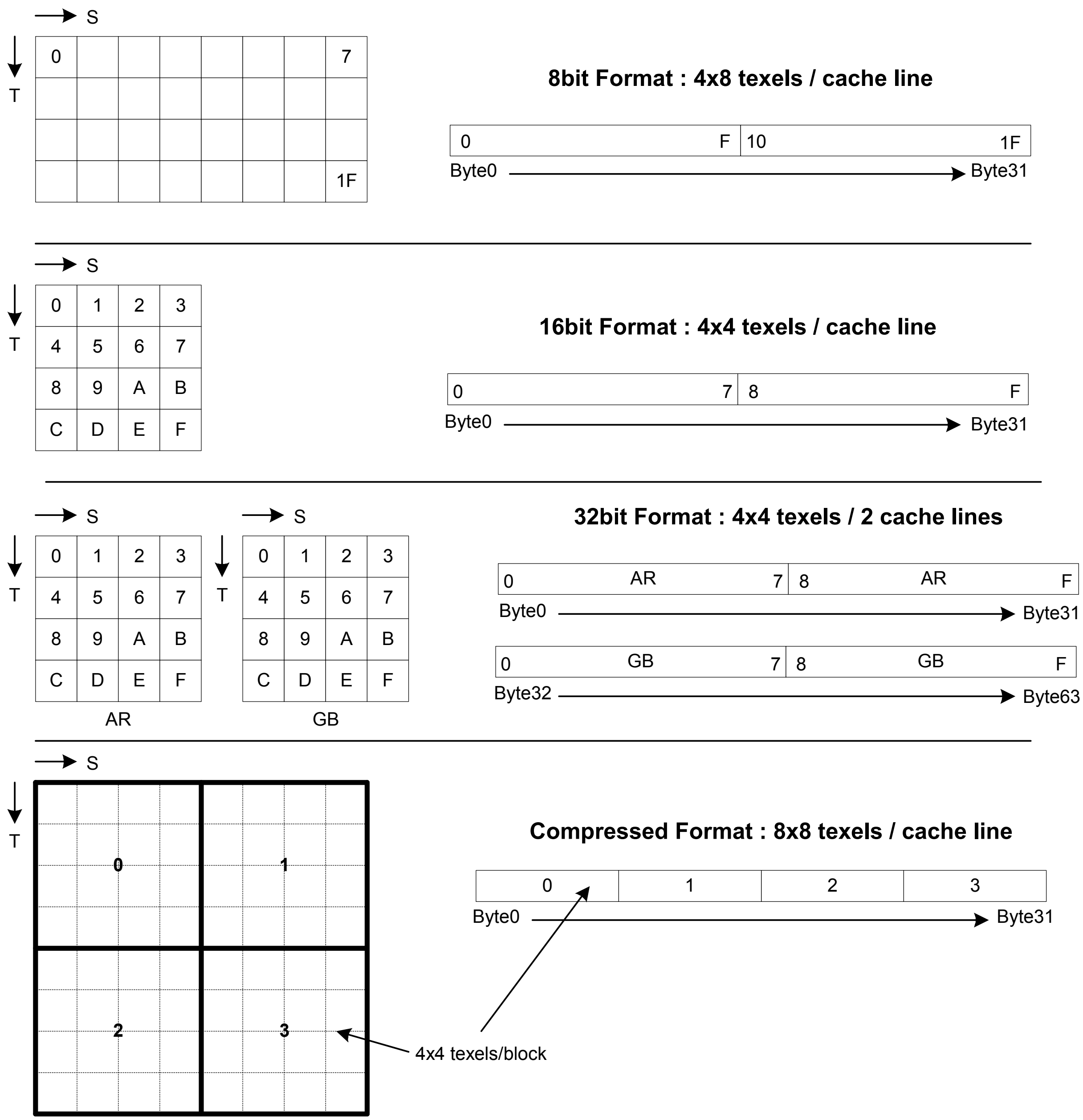
The compressed texture format allows for a 1-bit alpha. If a multi-bit alpha is desired, this can be accomplished through the use of multi-texture.

D.2 Texture Tile Formats

Texture images are organized as 32-byte tiles. Each 32-byte tile represents a 2D region of texels. For 4-bit texels, each tile represents 8x8 texels. For 8-bit texels, each tile represents 4x8 texels. For 16-bit texels, each tile represents 4x4 tiles. For 32-bit texels, a pair of 32-byte tiles represents 4x4 texels. This is illustrated in the following figures.

Figure D-2 Texture Tile Formats



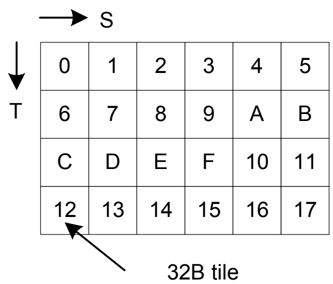


D.3 Texture Image Formats

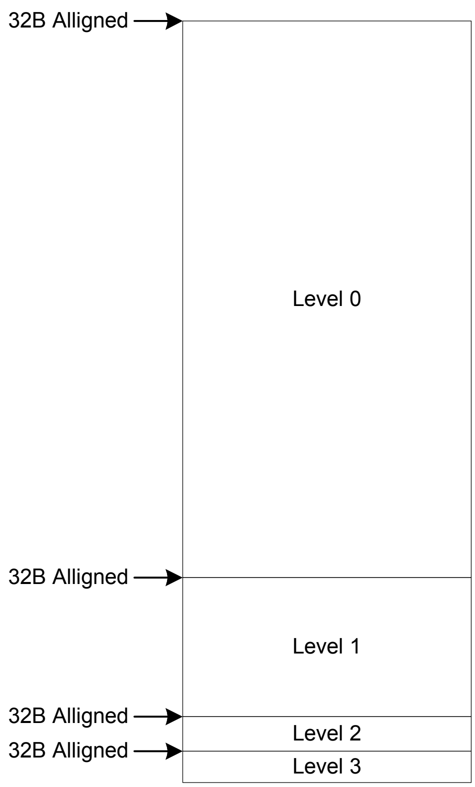
In main memory, the starting address of an image is aligned to 32 bytes. The tiles that make up the image are stored in row-column order. Both cached and pre-loaded images have the same organization in main memory.

Each row and column of an image is padded to a 32-byte tile. For 32-bit texels, each row is actually padded to a pair of 32-byte tiles. Each image is stored contiguously in main memory. For a mipmap pyramid, the levels are stored contiguously. The ordering of the images is from finest to coarsest. Each level of the mipmap pyramid is aligned to a 32-byte tile. This is illustrated in the following figures.

Figure D-3 Texture Image Formats



In main memory, images are alligned to 32B.
32B tiles are stored in row-column order.
Each row and column is padded to 32B.
For 32-bit format, each row is padded to a tile pair.



Mipmap images are stored contiguously.
Each level is alligned to 32B.

Appendix E. Memory Issues

The Graphics Processor has several data memory requirements, including alignment requirements for the following types of data:

- Texture and TLUT images
- Display lists
- Graphics FIFO
- External frame buffer (XFB)

E.1 Rules of Alignment

These data objects must be aligned because the GP is very fast; data from the main memory is transferred in 32-byte chunks. Data alignment allows for simple and fast hardware.

On other data objects, such as vertex, matrix and light arrays, additional hardware support eliminates the need for coarse alignment (these are 4-byte aligned). There are a large number of these data objects, and the memory consumption of each object is potentially low, so relaxing alignment restrictions helps to conserve memory.

Table E-4 outlines the alignment rules for these objects.

Table E-4 Memory Alignment Rules

Data Object	Alignment Rule	Function
Texture Map	• Base address = 32B • Width and height rounded up to tile boundary. Each tile is 32B. • Base address of each map in mipmap is aligned to 32B	• `GXInitTexObj(obj, image_ptr, …)` • `GXInitTexObjCI(obj, image_ptr, …)` • `GXCopyTex(dest, ..)`
Texture Lookup Table (TLUT)	• Base address = 32B • Length = 32B	• `GXInitTlutObj( , lut, )`
Display List	• Base address = 32B • Length = 32B	• `GXBeginDisplayList(list, size)` • `GXCallDisplayList(list, nbytes)`
Graphics FIFO	• Base address = 32B • Length = 32B	• `GXInitFifoBase(fifo, base, size)` • `GXInitFifoPtrs(fifo, read_ptr, write_ptr)` • `GXInitFifoLimits(fifo, hi_water_mark, lo_water_mark)`
External Frame Buffer (XFB)	• Base address = 32B • Width = 32B • Length = 32B	• `GXCopyDisp(dest, …)` • `VISetNextBuffer(framebuffer, …)`

E.2 Alignment Assistance Functions

The Revolution libraries contain functions that assist with alignment issues. Several examples appear in the table below.

Table E-5 Alignment Assistance Functions

Function	Purpose
`ATTRIBUTE_ALIGN(32)`	CodeWarrior compiler static variable alignment directive.
`OSRoundUp32B(x)`	Macro to round up a pointer to 32B.
`OSRoundDown32B(x)`	Macro to round down a pointer to 32B.
`OSAlloc(size)`	Memory heap allocation function. Always returns 32B base address and length of memory buffer.
`GXGetTexBufferSz(x, y, texel_type, mipmap)`	Function to compute correct amount of memory required to store a texture, based on the texture width, height, and texel type, and whether or not this texture is mipmapped.
`VIPadFramebufferWidth (width)`	Function to compute correct amount of memory required for the frame buffer, based on the pixel width of the frame buffer.

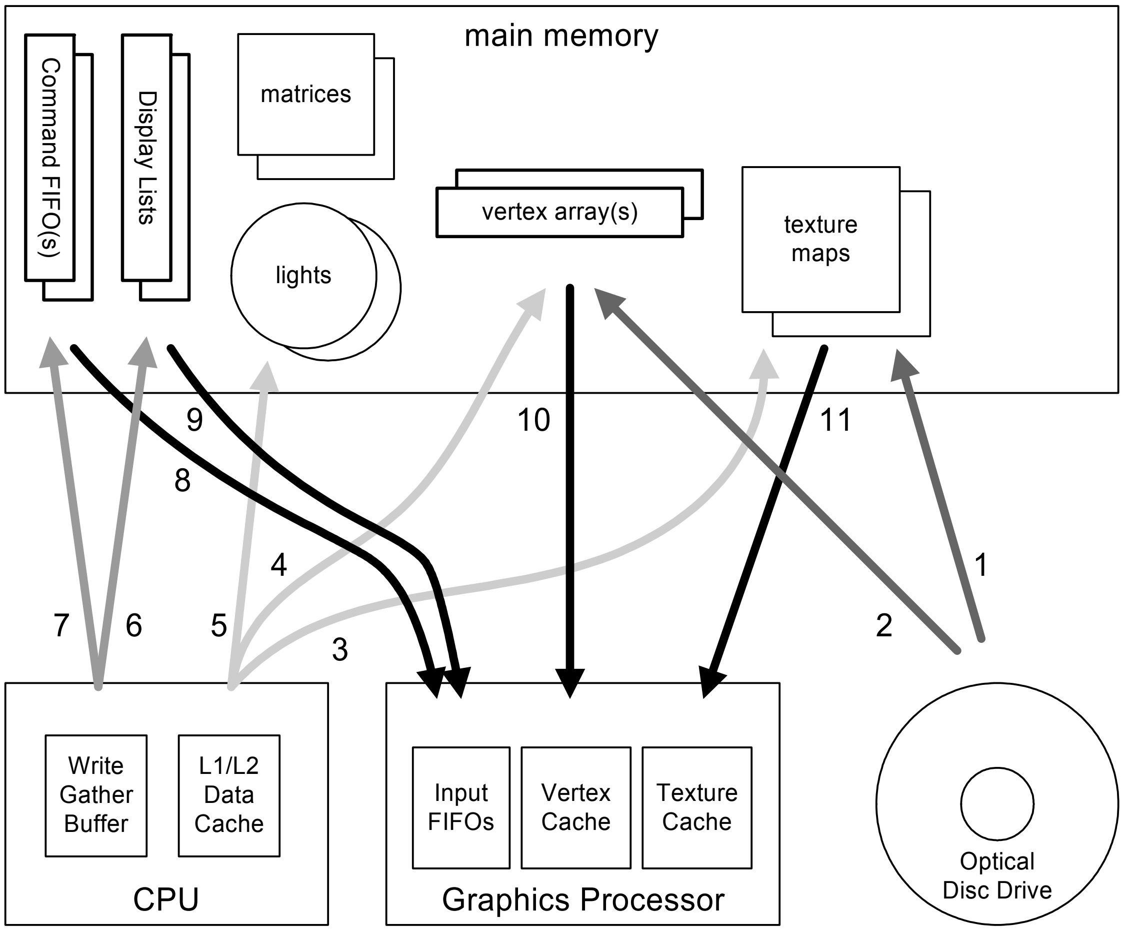
E.3 Data Coherency

Wii has multiple processors and hardware blocks that can update main memory. In addition, the CPU and GP contain various data caches. Since the hardware does not maintain coherency of the data in main memory and various associated caches, there are three potential sources of coherency problems:

- When the CPU modifies or generates data destined for the GP.
- When the CPU writes data through its write-gather buffer to cached memory.
- When loading new data destined for the GP from the disc into main memory.

Coherency problems may occur if the main memory used to store the data in these two latter cases were used for other graphics data.

Figure E-4 Data Coherency



The arrows in the Figure E-4 represent the following typical operations:

1. Loading texture images from the disc to main memory for a new game sector or level.
2. Loading geometry vertex display list from the disc to main memory for a new game sector or level.
3. Dynamic rendering of texture maps by the CPU.
4. Dynamic generation or modification of vertices by the CPU.
5. CPU animating lights and matrices.
6. CPU generating display lists.
7. CPU generating the graphics command stream.
8. GP reading graphics command stream.
9. GP reading display lists.
10. GP accessing vertices for rendering.
11. GP accessing textures for rendering.

In addition, other combinations are possible, such as loading display lists from the disc or writing command streams or display lists through the CPU cache.

E.3.1 Loading Graphics Data with the Optical Disc Drive Library

When the library loads data, the optical disc drive API automatically invalidates the loaded main memory portion that resides in the CPU data cache. This feature provides a safe method for programmers to modify the disc loaded data without worrying about CPU data cache coherency. This optical disc drive API feature activates by default; it can be deactivated by the programmer.

The graphical data loaded by the optical disc drive library may contain textures and vertices that have been already formatted for the GP to render. Therefore, invalidation of the vertex cache and texture cache regions may be necessary.

Code E-12 DVDSetAutoInvalidation

```
BOOL DVDSetAutoInvalidation(BOOL autoInval);
void GXInvalidateVtxCache( void );
void GXInvalidateTexRegion(GXTexRegion *region);
void GXInvalidateTexAll( void );
```

E.3.2 CPU Generating or Modifying Graphics Data

The CPU has two means of writing to main memory: the write-gather buffer and the CPU cache hierarchy. The write-gather buffer is normally used to "blast" graphics commands into memory without affecting the cache. As a result, information sent through the write-gather buffer is not cache coherent. Care must be taken when using the write-gather buffer to avoid writing to areas of memory that may be found in the CPU cache. The cache flushing instructions shown below may be used to force data areas out of the CPU cache.

If the CPU generates or modifies graphics data through its cache, the following memory types may end up containing stale data:

- Main memory.
- GP vertex cache and texture cache regions.

To send the correct data to the GP, we need to flush the CPU data cache as well as invalidate the GP vertex or texture cache. The CPU typically animates data one frame ahead of the GP, so efficient techniques to maintain data coherency include:

- Grouping all the CPU-modified graphics data in main memory sequentially, so that the block data cache flush is efficient.
- Invalidating the vertex cache, as well as the entire texture cache, at the beginning of each graphics frame.

Code E-13 Commands to Flush the CPU Data Cache

```
void DCFlushRange(void* startAddr, u32 nBytes); // write out & invalidate
void DCStoreRange(void* startAddr, u32 nBytes); // write out only
void DCInvalidateRange(void* startAddr, u32 nBytes); // invalidate only
```

E.3.2.1 Immediate Mode

If you use GX immediate mode APIs to update matrix or light data, you don't need to worry about coherency issues. These APIs copy the arguments into the graphics FIFO and include matrix and lighting functions with the form `GXLoad*Imm`.

E.3.2.2 Direct Data

If some vertex attributes in the vertex array descriptor are of the type `GX_DIRECT`, this data is copied directly into the graphics FIFO, so users must be aware of coherency considerations.

E.3.2.3 Indexed Data

If you use the GX indexed mode APIs to update matrix or light data, the software must flush the data cache in order to move the correct data into main memory. These APIs include matrix and lighting functions with the form `GXLoad*Indx`.

Furthermore, the hardware implements indexed matrices and lights by passing this data through the vertex cache; therefore, you must invalidate the vertex cache also. (The only reason for this is to simplify the hardware design.)

E.3.2.4 CPU Scratchpad

If the CPU L1 data cache is partitioned in scratchpad mode, you will need to DMA the modified data to main memory instead of flushing it from the normal data cache.