DMPGL 2.0 Specifications

Version 2.4

Digital Media Professionals Inc.

PROVISIONAL TRANSLATION

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

Table of Contents

Code

Tables

Figures

Equations

Revision History

Version	Revision Date	Description
2.4	2010/11/09	• Revised the explanation of DMP shadows. Added that the texture coordinate r is clamped when the shadow is referenced, the attenuation factor does not function correctly if a receiver is not rendered in the shadow buffer, that only the color g-component is used with the shadow accumulation pass, and made additional comments regarding the range of values for shadow texture components.
2.3	2010/10/07	• In sections 6.4.3 Shadow Reference Pass and 6.6.1 Gas Textures, clearly stated that shadow textures and gas textures cannot use mipmaps. • Fixed typo in the Japanese version.
2.2	2010/09/30	• Explained how the front and back faces are defined for the triangles rendered by points, lines, and particle systems. • Added supplementary information on culling. • Fixed typos.
2.1	2010/09/14	• Added information on the texture-wrapping mode for cube-map shadow filtering.
2.0	2010/08/20	• Added section 6.9.4 The w Buffer and Polygon Offset.
1.9	2010/07/30	• Revised number of texels fetched in section 2.8.2 Address and LOD Generation. • Added explanation that distance attenuation is disabled in fragment writing when `LIGHT_ENV_LAYER_CONFIG7_DMP` is set in section 6.3.2 Primary and Secondary Colors. • Added specification details in section 6.4.5 Shadow Accumulation Pass. • Added description of reading the stencil buffer in section 7.8 ReadPixels. • Added specifications in section 7.27 PolygonOffset. • Added new error codes in section 7.31 GetError. • Added section 7.33 Depth Information Textures. • Replaced some images to fix issue with broken images when creating a PDF version.
1.8	2010/07/07	• Revised the specifications of input values to lookup tables in sections 6.3.3 Lookup Tables (LUTs), 6.3.11 List of Reserved Uniforms, and Appendix B. • Added subsection 5.1.14 Texture Coordinate Precision under section 5.1 Texture Units in Chapter 5 Rasterization. • Added specifications to the emission and global ambient terms in the equation for calculating the fragment lighting primary color in section 6.3.2 Primary and Secondary Colors. • Added section 7.32 Obtaining Object Addresses to Chapter 7 Miscellaneous. • Added supplemental information about images in the HILO8 texture format in section 5.1.3 Texture Image Specifications.
1.7	2010/06/18	• Revised descriptions of the B and A components for `HILO8_DMP`. • Revised restrictions on 4-bit texture formats.
1.6	2010/06/04	• Added support in **`ReadPixels`** for reading out the depth buffer. • Added section 7.31 GetError to Chapter 7 Miscellaneous. • Added restrictions applying to 4-bit texture formats. • Added description of automatic texture mipmap generation in the new section 5.1.13 Automatically Generating Texture Mipmap Data.

Version	Revision Date	Description
1.5	2010/05/11	• Revised lookup table assignments for `LIGHT_ENV_LAYER_CONFIG3_DMP` fragment lighting. • Removed `LIGHT_ENV_LAYER_CONFIG8_DMP`, `LIGHT_ENV_LAYER_CONFIG9_DMP`, and `LIGHT_ENV_LAYER_CONFIG10_DMP` from the fragment lighting specifications. • Added restrictions imposed by specifications for early depth tests in section 5.5.1 Overview. • Added information on the maximum number of particles that can be generated by the particle system.
1.4	2010/04/23	• Added supplementary information about the display buffer to section 5.5.3 Block Mode. • Added error specifications to section 7.26 Uniform Location. • Fixed typos. • Added supplementary items about the shadow reference pass. • Added specifications related to the shadow texture format. • Revised explanations of `LIGHT_ENV_SP_DMP`.
1.3	2010/04/02	• Added descriptions of macros to specify the native formats for shadow and gas textures.
1.2	2010/03/19	• Fixed typos. • Revised settings for the `dmp_TexEnv[i].combineAlpha` uniform. • Standardized all clip coordinate and normalized device coordinate notation to the DMPGL 2.0 specifications. Added section 2.15 Coordinate Systems to Chapter 2 DMPGL 2.0 Pipeline to describe coordinate system differences with the OpenGL ES 1.1 specifications. • Revised information on compressed texture sizes. • Added sections 7.9 Framebuffer Objects, 7.21 CreateShader and CreateProgram, and 7.30 DepthRange to Chapter 7 Miscellaneous.
1.1	2009/11/30	• Revised explanations in section 5.1.4 Copying from the Framebuffer. • Changed the specifications for `ADD_MULT` in section 5.2.3 Other Combiner Features. • Added **`glStencilFuncSeparate`**, **`glStencilMaskSeparate`**, and **`glStencilOpSeparate`** to the list of unsupported functions. • Revised specifications for gas textures. • Revised specifications for selecting texture coordinate input. • Added sections 7.25 UniformMatrix, 7.26 Uniform Location, 7.27 PolygonOffset, and 7.28 LinkProgram. • Added section 5.5 Early Depth Tests. • Added Figure 5-6 Byte Swap for ETC Textures with Alpha. • Added the L4A4, L4, and A4 formats. • Removed descriptions related to the OpenGL ES Native Platform Graphics Interface (EGL).
1.0	2009/10/30	Initial version (branched from version 1.5 of the TEG2 document).

1 DMPGL 2.0 Overview

This document explains the DMPGL 2.0 specifications. The reader is expected to have an understanding of basic computer graphics algorithms and related graphics hardware or the OpenGL or OpenGL ES graphics system.

1.1 About This Document

This document explains the DMPGL 2.0 specifications in terms of their differences and extended specifications as compared to the OpenGL ES Common Profile Specification Version 2.0.23 (Full Specification). The basic DMPGL specifications comply with the aforementioned OpenGL ES 2.0 Specifications. DMP-specific features and extensions to the standard OES/ARB features are explained in terms of their differences as compared with the OpenGL ES 2.0 Specifications.

Unless specified otherwise, when this document refers to the "OpenGL ES 2.0 Specifications" it indicates the OpenGL ES Common Profile Specification Version 2.0.23 (Full Specification). In the same way, the "OpenGL ES 1.1 Specifications" indicate the OpenGL ES Common/Common-Lite Profile Specification Version 1.1.12 (Full Specification).

1.2 Structure of This Document

This document comprises the following chapters.

- Chapter 1 DMPGL 2.0 Overview
- Chapter 2 DMPGL 2.0 Pipeline
- Chapter 3 DMP Shaders
- Chapter 4 Primitives
- Chapter 5 Rasterization
- Chapter 6 Reserved Fragment Shaders
- Chapter 7 Miscellaneous

This is Chapter 1, which gives an overview of DMPGL 2.0. Chapter 2 describes the DMPGL 2.0 pipeline. Chapter 3 describes the DMPGL 2.0 shader environment. Chapter 4 describes the primitives handled by DMPGL 2.0. In addition to the standard primitives (points, lines, and triangles), DMPGL 2.0 can handle rendering units that are not supported by OpenGL ES 2.0, including adjacent triangles and subdivision patches. Chapter 5 mainly describes the texture environment and DMPGL-specific handling of textures in fragment processing. Chapter 6 focuses on the fixed pipeline feature that currently implements DMPGL 2.0-specific effects, from the programmable pipeline view. Chapter 7 gives other items that warrant special mention.

1.3 Single-Thread Model

The DMPGL 2.0 implementation does not consider a multi-threaded environment. As a result, DMPGL 2.0 functions must not be called from multiple threads. DMPGL 2.0 also does not support multiple rendering contexts. DMPGL 2.0 functions must be called on only a single rendering context.

1.4 Programmable Vertex Processing and Fixed Fragment Processing

DMPGL 2.0 provides a programmable pipeline view (from OpenGL ES 2.0) for vertex and fragment processing. However, the current implementation does not allow you to attach any user-defined fragment shader. You can only attach fragment shaders that have already been defined by DMPGL 2.0. The fragment pipeline is simply a fixed pipeline which has been given the appearance of a programmable view when considered through the API. You therefore cannot change fragment shaders. This document refers to these unchangeable fragment shaders as reserved fragment shaders. Changes to the context state by related existing OpenGL ES commands are emulated by changing the uniforms implemented by the reserved fragment shaders.

1.5 Creating Programmable Geometry

DMPGL 2.0 provides a programmable pipeline view for geometry processing that can create vertices. However, the current implementation does not allow you to attach any user-defined geometry shader. You can only attach geometry shaders provided by DMPGL 2.0. Unlike reserved fragment shaders, geometry shaders are implemented by programmable processing.

1.6 Examples and Notations

1.6.1 Showing Variables, Constants, Functions, and Reserved Uniforms

This document uses the `Courier New` font for function names, constant names, argument names, and reserved uniform names. Function names are shown in **`bold`** and argument names are shown in ***`bold italics`***.

Example:

```
void Command(type argument);
SYMBOLIC_CONSTANTS
```

1.6.2 Notation of Sets

Sets are sometimes shown in curly brackets {}. If a string is connected to a pair of brackets, it indicates a set that combines the elements of the string and set. For example, `example{A,B,C}` has the same meaning as `exampleA`, `exampleB`, and `exampleC`. If two or more sets of strings and brackets are connected, they indicate a set of every possible combination.

Some reserved uniforms include a number in square brackets, such as `[0]`, `[1]`, or `[2]`. Where the notation `[i]` is used with a range explicitly given in the form `(i is 0, 1, or 2)`, `[i]` indicates the set `{[0], [1], [2]}`.

Example 1) The notation "`example{0,1}.sub{X,Y,Z}`" indicates the set `{example0.subX, example0.subY, example0.subZ, example1.subX, example1.subY, example1.subZ}`.

Example 2) The notation "`example[i].sub{X,Y,Z} (i is 0, 1, or 2)`" indicates the set `{example[0].subX, example[0].subY, example[0].subZ, example[1].subX, example[1].subY, example[1].subZ, example[2].subX, example[2].subY, example[2].subZ}`.

2 DMPGL 2.0 Pipeline

DMPGL 2.0 follows the OpenGL ES 2.0 interface, but in the actual pipeline implementation, the structure of vertex processing most closely resembles the OpenGL ES 2.0 pipeline, and the structure of fragment processing most closely resembles the OpenGL ES 1.1 pipeline.

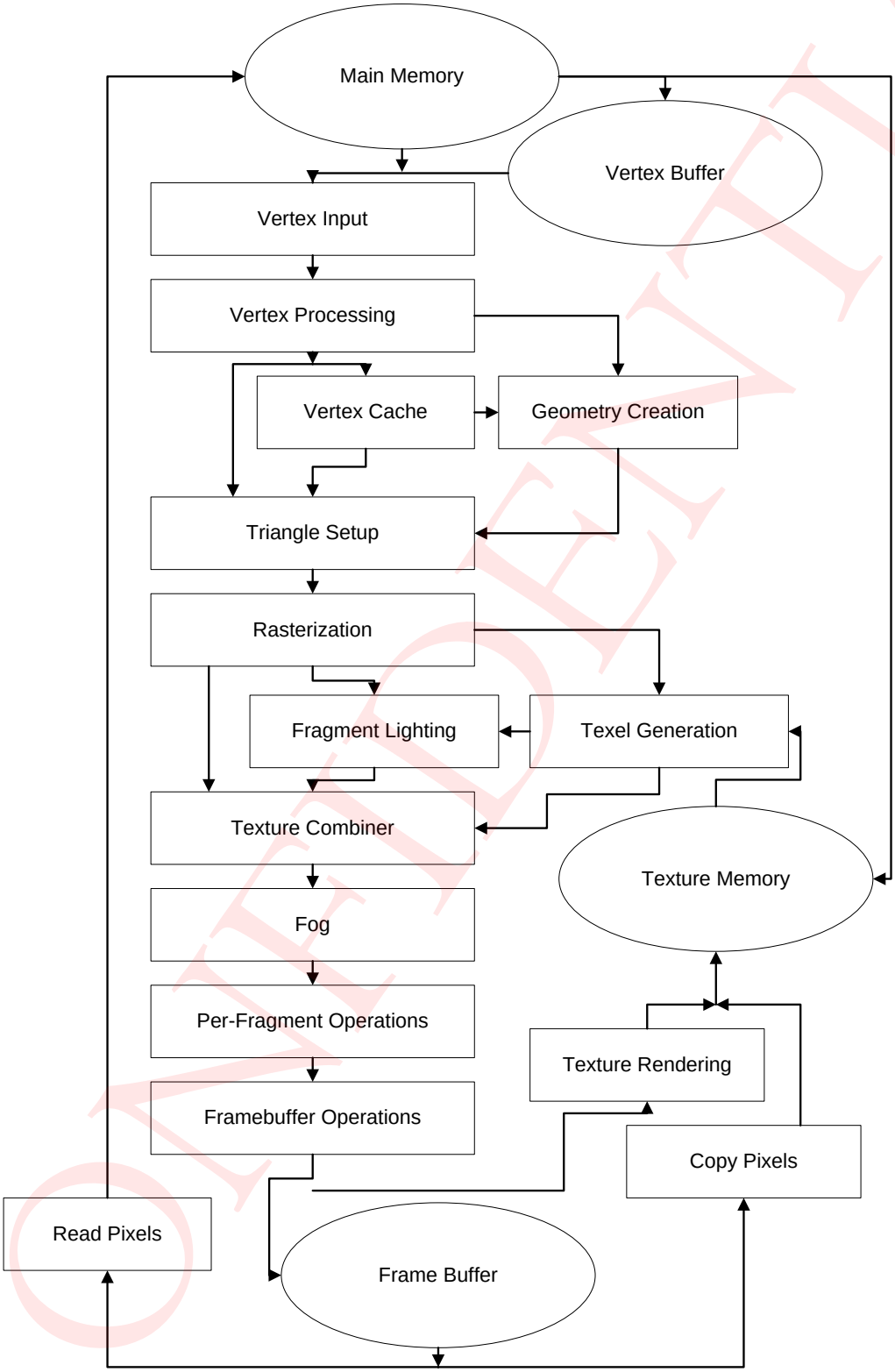
This chapter gives an overview of the DMPGL 2.0 pipeline.

2.1 Overview Figure for the DMPGL 2.0 Pipeline

Figure 2-1 gives an overview of the DMPGL 2.0 pipeline. Although the structure basically follows the OpenGL ES 2.0 and 1.1 pipelines, note that there are DMP-specific extensions.

Figure 2-1 Overview Figure for the DMPGL 2.0 Pipeline

The arrows indicate the flow of data.



The DMPGL 2.0 pipeline mainly comprises the following processes.

- Vertex input
- Vertex processing
- Vertex cache
- Geometry creation
- Triangle setup
- Rasterization
- Texel generation
- Procedural textures
- Fragment lighting
- Texture combiner
- Fog
- Per-fragment operations
- Framebuffer operations

The following sections explain the individual processes.

2.2 Vertex Input

Input processing for vertex data. In both DMPGL 2.0 and OpenGL ES 2.0, there is not a clear distinction between coordinates, normals, and so on in the handled vertex data. The aforementioned attributes are determined by how the data is mapped to specific vertex processing. However, the following general attributes are input as vertex data during graphics-specific processing.

- Vertex coordinates
- Normal vectors
- Tangent vectors
- Texture coordinates
- Vertex colors

In general, vertex coordinates are the only required attributes in the list above. Normal vectors and tangent vectors are not required in unlit environments. Texture coordinates are not required in environments that have disabled texture processing, nor when they are created during vertex processing. Vertex colors are not required in lit environments.

Vertex processing can run general-purpose calculations, so if you can assume that the next process will create the required attributes, it follows that no particular attributes are fixed as input attributes.

In DMPGL 2.0, the vertex input process converts any vertex data that is not stored as a `float` into a `float`. However, note that this process does not normalize the data.

Data can be input to either of the following locations with DMPGL 2.0.

- Vertex data that does not use the vertex buffer
- Vertex data that does use the vertex buffer

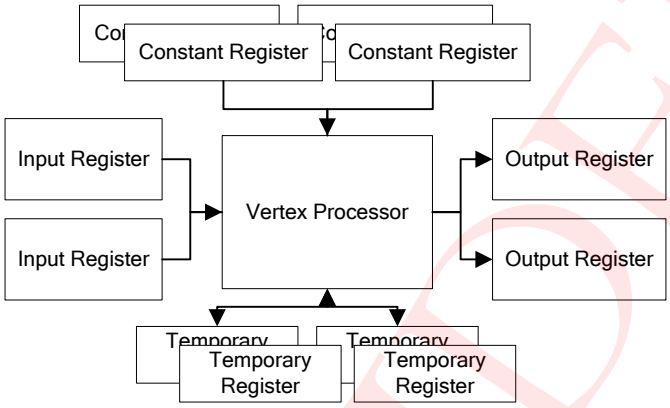
There is no support for simultaneously reading mixed vertex data, some of which uses and some of which does not use the vertex buffer.

Although it also depends on the type and other aspects of the vertex data, performance is generally better with vertex data that uses the vertex buffer.

2.3 Vertex Processing

Per-vertex processing is run on the input vertex data. DMPGL 2.0 uses a processor that can handle vector data comprising four 24-bit floats. This processor can run general-purpose calculations but it cannot read or write data from VRAM. It can read data from constant registers, and it can read from and write to temporary registers. Vertex attribute data loaded by the vertex-input process is written to output registers. The processor writes its output data to the output registers, and the data is then sent to the next process.

Figure 2-2 Vertex Processor and Registers Used



The DMPGL 2.0 implementation processes vertices in parallel on multiple vertex processors.

Vertex processors are ordinarily used to run graphics-specific processing. The following can be considered as vertex processing in a general graphics pipeline.

- Coordinate conversion from object space to eye space (modelview transform)
- Coordinate conversion from eye space to clip space (projection transform)
- Per-vertex color creation by lighting processes
- Texture-coordinate generation

To generate texture coordinates, you can transform input texture coordinates using matrices, or you can generate texture coordinates themselves from other vertex attributes. Vertex colors do not need to be generated when fragment lighting is enabled.

OpenGL ES 1.1 clearly defines the content of vertex processing and has a unique API for configuring the aforementioned processes. In contrast, like OpenGL ES 2.0, DMPGL 2.0 does not use an API to define the content of vertex processing.

In addition to instructions required for general-purpose calculations, the vertex processors have a combination of instructions optimal for 3D graphics processing such as the processes mentioned above. Programs run on the vertex processors are called vertex programs and are defined and loaded by the user. DMPGL 2.0 allows user-defined programs to be used, and implements this capability through the vertex shader mechanism.

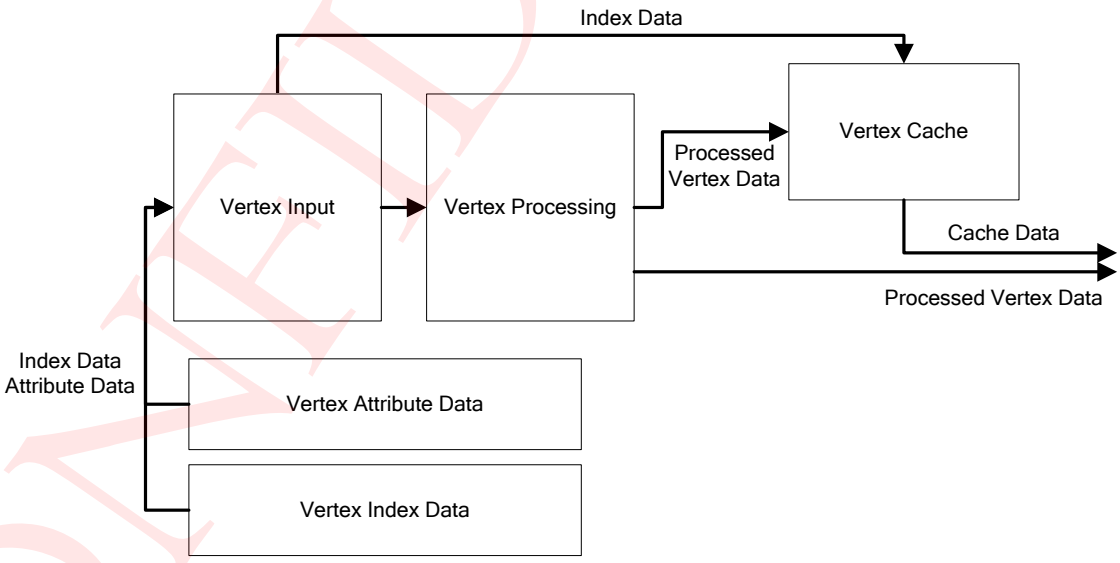
2.4 Vertex Cache

The cache is used to save some of the vertex data that is either created or handled by vertex processing. If the vertex input process determines that the input vertex data is the same as the original data for a vertex that exists in the cache, that input vertex data is not sent to vertex processing; instead, the already-processed data in the cache is sent on to the next process. In general, if vertex data is input using `TRIANGLES`, that vertex tends to be processed multiple times, but this duplicate processing can be omitted if there is a processed vertex in the cache.

DMPGL 2.0 has the following conditions for using the vertex cache.

- Vertex data must be input in a format that accesses the vertex index. In other words, vertex data must be input by a call to `DrawElements`.
- Vertex data must use the vertex buffer. In other words, data must be input using vertex buffer objects.

Figure 2-3 The Relationship Between Vertex Processing and the Vertex Cache



During vertex processing, if already-processed vertex data with the same index exists in the vertex cache, the data in the cache is used.

2.5 Geometry Creation

The vertex processing outlined in the previous section processes the input vertex data one vertex at a time, but it cannot create the vertices themselves. This geometry creation process is different because in addition to processing vertices in a way similar to vertex processing, it can create the vertices themselves and thus output more vertices than were input.

In DMPGL 2.0, all primitives other than triangles must be created by geometry shaders. For example, the geometry creation process creates four vertices when given a single input vertex for a point primitive. Likewise, it creates four vertices when given two input vertices for a line primitive.

Just like vertex processing, the content of geometry creation is not defined by the API. Programs run by this process are called geometry programs. They are generally defined and loaded by the user. Like vertex programs, they are implemented using the vertex shader mechanism, but the DMPGL 2.0 implementation does not allow user-defined geometry programs.

Geometry creation is not required by the pipeline and is omitted when only triangle primitives are handled.

Only one of the vertex processors is shared with the geometry processor by the DMPGL 2.0 implementation. In other words, a single processor is used both for vertices and geometry. Pipelines that use geometry creation therefore have a structure resembling the following figure. No other structure exists.

Figure 2-4 Processor Structure Without Geometry Creation

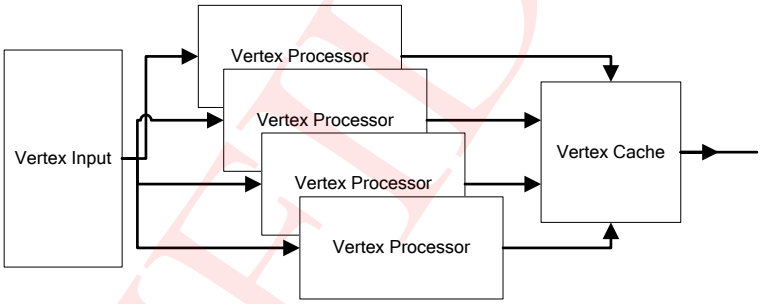
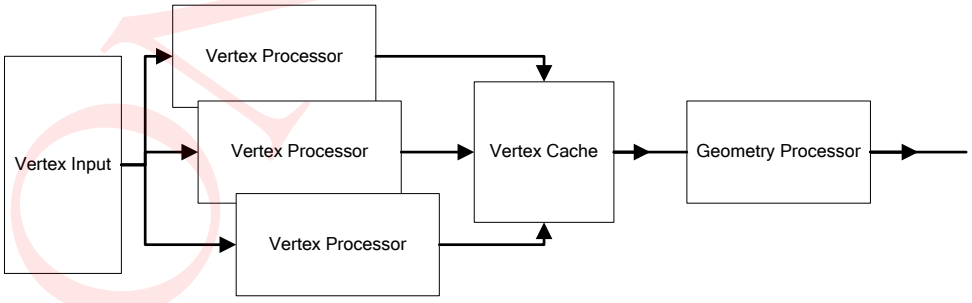


Figure 2-5 Processor Structure with Geometry Creation

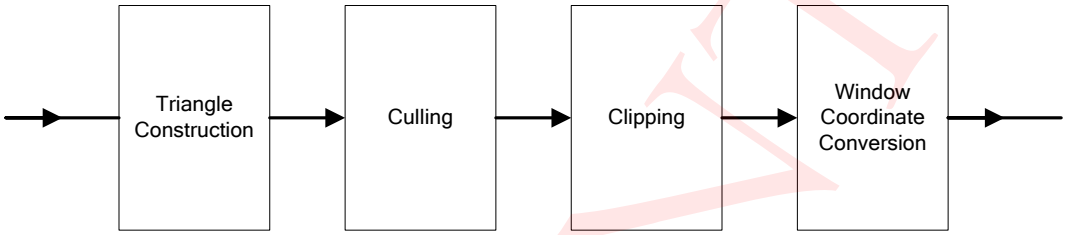


2.6 Triangle Setup

This process performs the following main steps.

- Triangle construction
- Culling
- Clipping
- Window coordinate conversion

Figure 2-6 Sub-Processes That Compose the Triangle Setup Process



2.6.1 Triangle Construction

This creates triangles from the individual vertices sent from vertex processing or the vertex cache. For `TRIANGLES`, this process creates a single triangle from three transferred vertices; for `TRIANGLE_STRIP` and `TRIANGLE_FAN`, it creates a single triangle from one transferred vertex.

DMPGL 2.0 handles all non-triangle primitives as triangles. For example, for a point primitive, geometry creation takes a single input vertex and generates four vertices. These four vertices generate two triangles. In other words, at this stage of processing there is no distinction between different primitives.

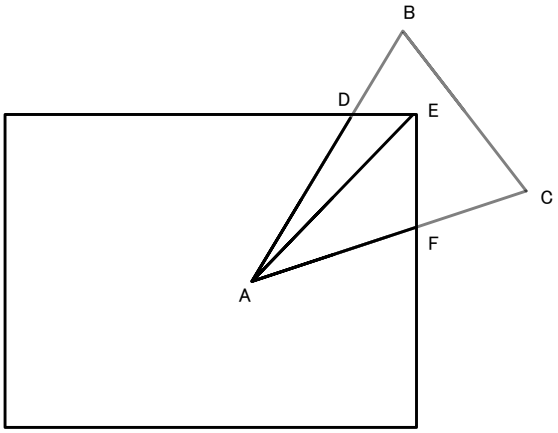
2.6.2 Culling

This process determines whether the generated triangles are front- or back-facing. When culling is enabled, triangles that are determined to be "culling faces" are not processed beyond this step. If a triangle is rendered identically regardless of whether culling is enabled or disabled, the time taken to process it is likewise unaffected by whether culling is enabled or disabled because a triangle’s facing (front or back) is always checked.

2.6.3 Clipping

Triangles that survive the culling process are clipped here to the view volume and user-defined clipping planes. The clipping implementation in DMPGL 2.0 involves the creation of new vertices and new triangles for clipped triangles.

Figure 2-7 Clipping to the View Volume



When the triangle ABC is clipped, the new vertices D, E, and F are created and the triangle ABC is split into triangles ADE and AEF.

2.6.4 Window Coordinate Conversion

Vertex coordinates are handled as homogenous 4-component coordinates but this process divides by the fourth coordinate, transforming the coordinates into normalized device coordinates and mapping the vertex into 2D space. This mapping and viewport transformation fixes the position of the triangle in the display and simultaneously creates a depth value. This depth value is used later by the depth test when it removes hidden surfaces.

In addition to supporting ordinary depth values that are divided by the fourth coordinate *w*, DMPGL 2.0 supports depth values that are not divided by *w* (the w buffer). In this case, a user-defined scaling coefficient is required.

In DMPGL 2.0, this sub-process handles polygon offsets.

DMPGL 2.0 clip coordinates and normalized device coordinates differ somewhat from the OpenGL ES 1.1 definition. (See section 2.15 Coordinate Systems.)

2.7 Rasterization

This process converts triangles into sets of fragments. Rasterization generates fragments that occupy the space inside the triangles, then interpolates the vertex attributes over the set of affected fragments and assigns the interpolated attribute values to individual fragments based on the fragment's position. All subsequent processes operate on the fragments that are generated in this step.

Unlike OpenGL ES 2.0, DMPGL 2.0 implements a fixed pipeline for subsequent processing and therefore there is a limited, fixed set of attributes that can be assigned to fragments. The following main attributes can be assigned.

- Window coordinates
- Depth value
- Texture coordinates and their partial differential values
- Quaternions
- View vectors
- Vertex colors

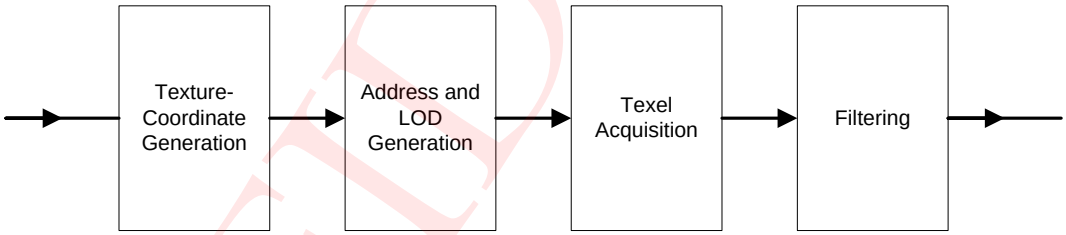
In DMPGL 2.0, scissoring is handled by this step, not by the per-fragment operations.

2.8 Texel Generation

Texel generation takes texture images loaded into memory and uses them to create texels corresponding to the texture coordinates of the fragments. This mainly comprises the following processes. In addition to ordinary 2D textures, DMPGL 2.0 can create texels from procedural textures. This section describes processing when handling 2D textures. Texel generation for 2D textures comprises the following sub-processes.

- Texture-coordinate generation
- Address and LOD generation
- Texel acquisition
- Filtering

Figure 2-8 Sub-Processes That Compose the Texel Creation Process



2.8.1 Texture-Coordinate Generation

This process generates texture coordinates. OpenGL ES 2.0 does not distinguish between vertex attributes and therefore both texture and vertex coordinates have four components. When handling multiple textures, there are only as many sets of texture coordinates as there are textures.

With an ordinary 2D texture, the first and second components of the texture coordinates are the components generated by this step. When a 2D texture is used as a perspective texture, the first and second components are divided by the fourth component when the texture coordinates are generated. When 2D textures are used to make a cubic environment map, the first, second, and third components are first used to select the textures to reference. (A cubic environment map is defined by six 2D textures, one of which is targeted for addressing) Next, texture coordinates are generated for the selected texture images.

DMPGL 2.0, on the other hand, uses a special method for generating texture coordinates. For all units other than unit 0, fragments save texture coordinates with only two components. In other words, only unit 0 can be used for cubic environment maps and perspective textures.

Even unit 0 saves only three components for texture coordinates. Although the aforementioned method generates texture coordinates for cubic environment maps, for perspective textures, the generated coordinates are divided by the third component. You must therefore be careful with texture coordinates generated by vertex processing.

2.8.2 Address and LOD Generation

This process generates addresses to use for accessing the texture images from the texture coordinates created by the previous process. If the texture is composed of mipmaps, an LOD is calculated; this LOD can then be evaluated to indicate which mipmap level of texture image to access. The LOD is generated from the partial derivative of the texture coordinates assigned to the fragment.

In DMPGL 2.0, one, two, four, or eight texels are obtained for a single fragment (the number of texels is affected by subsequent filtering), so the same number of corresponding addresses is generated.

2.8.3 Obtaining Texels

Texels are obtained from the image data in memory, using the addresses and LOD generated by the previous process. The DMPGL 2.0 implementation gets one texel per fragment per enabled texture when the filtering mode is point sampling, four texels simultaneously when binary filtering is enabled, and twice as many texels simultaneously when trilinear filtering is enabled for each mode.

2.8.4 Filtering

DMPGL 2.0 supports both point sampling (`NEAREST`, `NEAREST_MIPMAP_XXX`) and binary filtering (`LINEAR`, `LINEAR_MIPMAP_XXX`). Trilinear filtering (`XXX_MIPMAP_LINEAR`) can also be enabled for both, in which case the filtering effects of the two optimal mipmap level textures are interpolated to produce the final texel color.

When the texture format is `SHADOW_DMP` or `SHADOW_NATIVE_DMP`, filter operations are in a special mode and the aforementioned process is entirely replaced. For details, see section 6.4 DMP Shadows.

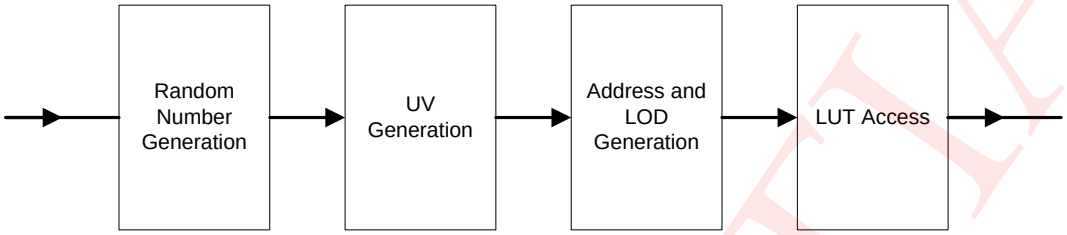
2.9 Procedural Textures

DMPGL 2.0 can handle procedural textures in addition to ordinary 2D textures. In OpenGL ES 2.0, a series of processes to create procedural textures is defined by the user as a fragment program; in DMPGL 2.0, procedural textures are handled by a fixed pipeline process. The following is the structure of sub-processes used to generate procedural textures.

- Random number generation
- UV generation
- Address and LOD generation

- LUT access

Figure 2-9 Sub-Processes for Generating Procedural Textures



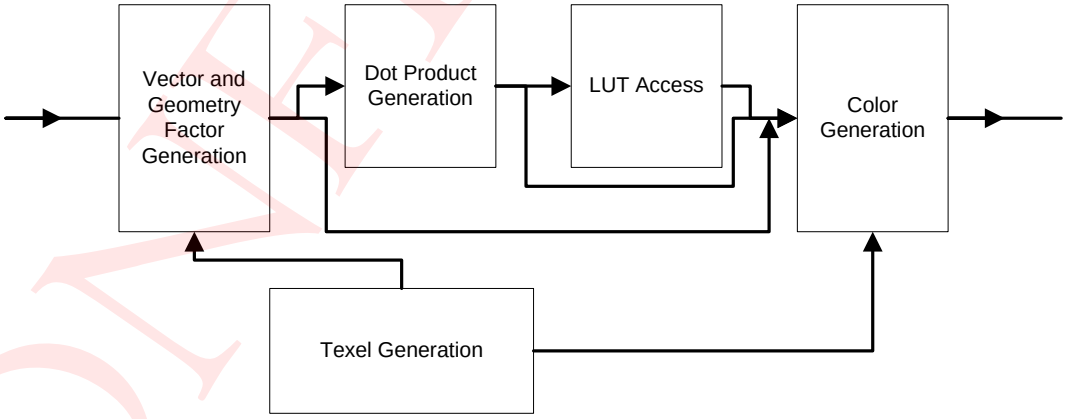
For details on each of the sub-processes, see section 6.2 Procedural Textures.

2.10 Fragment Lighting

DMPGL 2.0 has pipeline features that support per-fragment lighting calculations. Fragment lighting mainly comprises the following four sub-processes. Several of these processes can use the output of texel generation, and it is also possible to apply the contributions of bump mapping and shadows to color generation.

- Vector and geometry factor generation
- Dot product generation
- LUT (lookup table) access
- Color generation

Figure 2-10 Sub-Processes That Compose the Fragment Lighting Process



2.10.1 Vector Generation

This sub-process creates the various vectors used for fragment lighting calculations. The following vectors are created.

- Normal vectors
- Tangent vectors
- View vectors
- Half-angle vectors
- Light vectors
- Spotlight-direction vectors

All vectors are created as unit vectors. The results of texel generation can be used to add perturbations to the normal vectors and tangent vectors (bump mapping and tangent mapping).

Geometry factors are also created in this step.

2.10.2 Dot Product Generation

This process calculates the dot products used for color calculations and LUT access, based on the vectors generated by the previous process. This sub-process finds the dot products of the following vector pairs.

- Light vector and normal vector
- Half-angle vector and normal vector
- View vector and normal vector
- Half-angle vector and view vector
- Light vector and spotlight-direction vector
- Tangent vector and the reflection of the half-angle vector from the tangent plane

2.10.3 LUT Access

This sub-process gets values from the lookup tables, using the dot products generated by the previous sub-process as inputs.

2.10.4 Color Generation

This sub-process uses pipelined operations to calculate the final fragment colors. This sub-process can add shadow contributions to the results of texture generation. For details, see section 6.3 DMP Fragment Lighting.

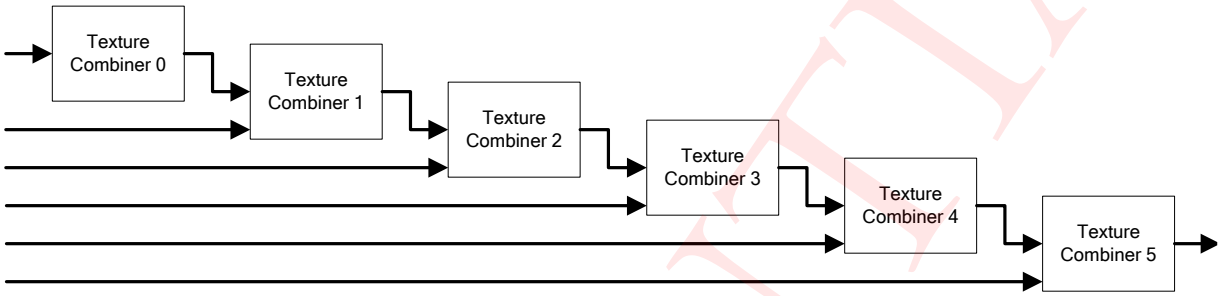
2.11 Texture Combiners

The texture combiners calculate the final fragment colors given three of the following inputs. The formulas they use are fixed, and the user selects which formulas to use. DMPGL 2.0 has the following main inputs.

- Output from texel generation
- Output from fragment lighting
- Vertex colors from rasterization
- User-defined constant colors
- Output from the previous-stage combiner unit

This process comprises six combiner units that operate on the aforementioned inputs and are connected in series.

Figure 2-11 Texture Combiner Structure



During texture combiner calculation, output from combiner unit *n*-1 is added to combiner unit *n* (*n* > 0).

DMPGL 2.0 has a special relationship connecting the texture combiners and texel generation process. In the same way, the relationship is even more complicated in a mode with gas shading. For details, see Chapter 5 Rasterization (on textures) and Chapter 6.6 Gas.

2.12 Fog

This process combines the results from the previous process with fog colors and blending coefficients. DMPGL 2.0 calculates the blending coefficients from values looked up in the fog table, which takes depth values as inputs. The table has a resolution of 128 levels.

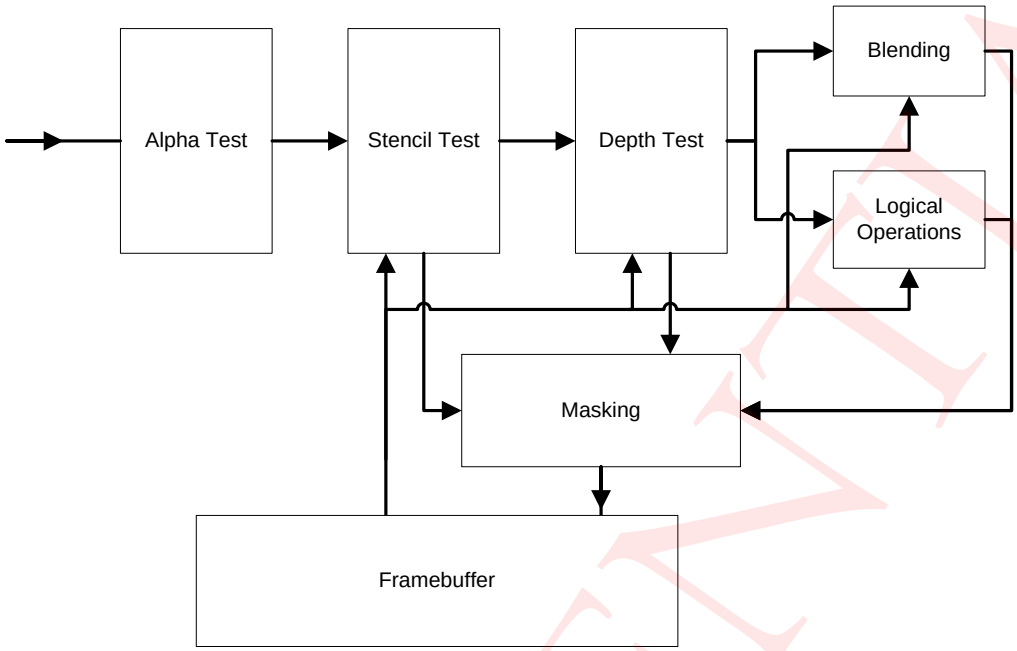
There is also a mode that supports gas shading. In that mode, fog processing has special behavior. For details, see section 6.6 Gas.

2.13 Per-Fragment Operations

In the DMPGL 2.0 default mode, per-fragment operations are comprised of the same processes as in OpenGL ES 2.0. Sub-processes are run sequentially.

- Alpha test
- Stencil test
- Depth test
- Blending or logical operations
- Masking of the various framebuffers

Figure 2-12 Default Mode for Per-Fragment Operations



DMPGL 2.0 supports logical operations. Blending and logical operations are mutually exclusive processes. Unlike OpenGL ES 2.0, scissoring is not performed in DMPGL 2.0.

In DMPGL 2.0, processing specific to rendering shadows and gaseous objects is done here in per-fragment operations. This processing entirely replaces the processes of the aforementioned default mode. For details on the mode for shadows and gas, see section 6.4 DMP Shadows and section 6.6 Gas.

2.14 Framebuffer Operations

The framebuffer operations all involve the framebuffer. This stage comprises the following sub-processes.

- Read pixels
- Copy pixels
- Render textures
- Clear the framebuffer

2.14.1 Read Pixels

This sub-process transfers the contents of the color buffer into main memory. The content of the depth and stencil buffers cannot be read.

2.14.2 Copy Pixels

This sub-process transfers all or part of the color buffer to texture memory. DMPGL 2.0 does not support transfers when the color buffer format and texture format do not match. Transfers with format conversions are not supported.

2.14.3 Render Textures

This sub-process supports direct writes to texture memory. DMPGL 2.0 implements this feature using framebuffer objects.

2.14.4 Clear the Framebuffer

Unlike in OpenGL ES 2.0, in DMPGL 2.0 the framebuffer is cleared entirely independent of the pipeline and is unaffected by scissor tests or masking.

2.15 Coordinate Systems

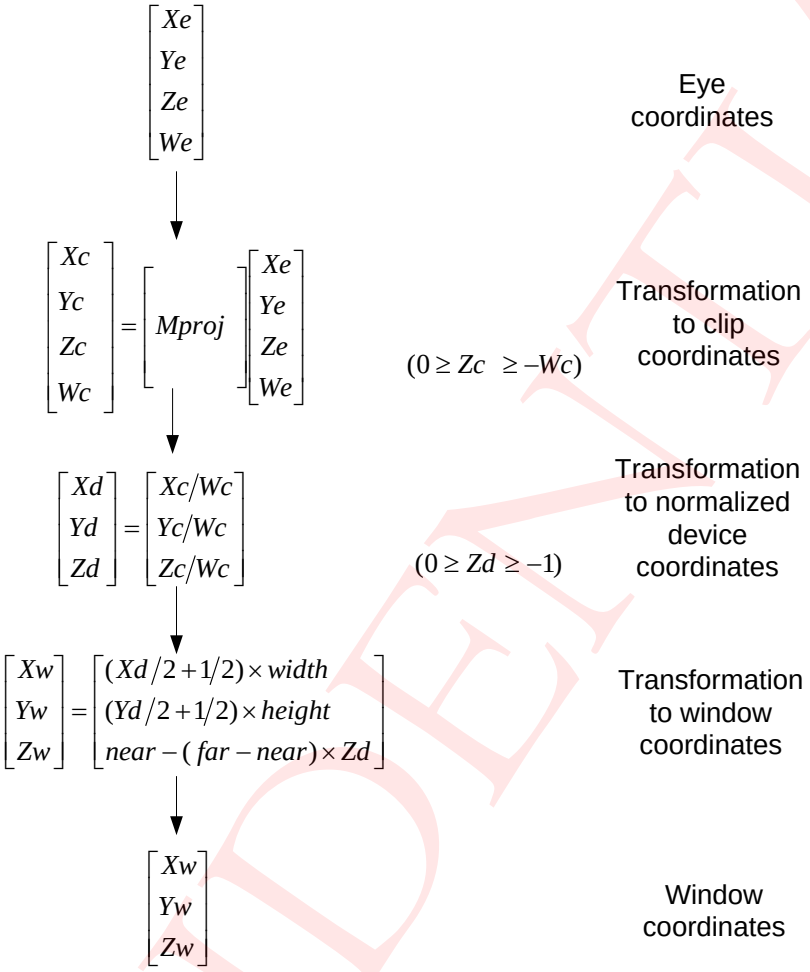
This section describes the DMPGL 2.0 coordinate systems. The table below shows the coordinate systems used in DMPGL 2.0 and the variable notation used in each. Both DMPGL 2.0 and OpenGL ES 2.0 generate the same window coordinates from the same eye coordinates, but the calculations themselves differ somewhat.

Table 2-1 DMPGL Coordinate Systems and Variables

	Variables	Description
Eye coordinates	Xe, Ye, Ze, We	Coordinate system after modelview transformation, with the viewpoint set as the origin
Clip coordinates	Xc, Yc, Zc, Wc	Coordinate system after projection transformation
Normalized device coordinates	Xd, Yd, Zd	Coordinate system after clipping and *w* division (also called perspective division)
Window coordinates	Xw, Yw, Zw	Coordinate system using X and Y for the window's pixel coordinates and Z for the depth values

Figure 2-13 shows the process for transforming from eye coordinates to window coordinates. Note that in DMPGL 2.0, the clip coordinate Zc and the normalized device coordinate Zd undergo different handling compared to the corresponding coordinates under OpenGL ES 1.1.

Figure 2-13 Conversion from Eye Coordinates to Window Coordinates



The relationship between the projection matrix $Mproj_{DMPGL}$ (that projects the eye coordinates into the clip coordinates used by DMPGL 2.0) and the projection matrix $Mproj_{OES}$ (that projects the eye coordinates into the clip coordinates used by OpenGL ES 1.1) can be expressed with the following equation.

Equation 2-1 Relationship Between DMPGL 2.0 and OpenGL ES 1.1 Projection Matrices

$$\begin{bmatrix} Mproj_{DMPGL} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.5 & -0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Mproj_{OES} \end{bmatrix}$$

3 DMP Shaders

As mentioned in Chapter 1 DMPGL 2.0 Overview, DMPGL 2.0 provides a programmable pipeline view (from OpenGL ES 2.0) for vertex and fragment processing. A programmable vertex creation view is also provided.

DMPGL 2.0 does not support shader programs written in the OpenGL ES Shading Language. Shader programs are written in a proprietary assembly language provided by DMPGL 2.0.

3.1 Reserved Uniforms

DMPGL 2.0 defines reserved uniforms (described later) to use with reserved shaders (fragments and geometry). Reserved uniforms start with the string "dmp_". To avoid collisions with reserved uniforms, the user's shaders must therefore not use uniforms that start with the string "dmp_".

3.2 Vertex Shaders

The series of procedures involved in the programmable vertex processing provided by DMPGL 2.0 complies with section 2.10 Vertex Shaders of the OpenGL ES 2.0 specifications, but the DMPGL 2.0 implementation does not support several of the OpenGL ES 2.0 features.

In DMPGL 2.0, any user-defined vertex shader object can be created and attached, but vertex shader objects are required to be attached to program objects. In DMPGL 2.0, the loading and compiling operations described in the OpenGL ES 2.0 specifications, section 2.10.1 Loading and Compiling Shader Source, are not supported. In other words, **`ShaderSource`** and **`CompileShader`** do not exist. User-defined shaders are always loaded by **`ShaderBinary`**.

Code 3-1 ShaderBinary

```
void ShaderBinary(sizei count, const uint *shader,
                  enum binaryformat, const void *binary, sizei length);
```

Specify `PLATFORM_BINARY_DMP` for the third argument, ***`binaryformat`***.

In DMPGL 2.0, the fourth argument in the following code cannot be set to `TRUE`.

Code 3-2 VertexAttribPointer

```
void VertexAttribPointer(uint index, int size, enum type,
                  boolean normalize, sizei stride, const void *pointer);
```

In DMPGL 2.0, normalization must be explicitly performed by a vertex shader. If you specify vertex attributes that require normalization, you must write code in the vertex shader to perform that normalization.

Any output attributes may be set for a vertex shader. Note, however, that sometimes there are required output attributes, depending on what reserved uniforms are set in the target program object. If the reserved uniform `dmp_FragmentLighting.enabled` is set to `TRUE`, for example, the vertex

shader must calculate valid quaternion attributes and configure them to be output. When a vertex shader is attached to a program object that targets a geometry shader that creates points, the `size` attribute is required to be output in addition to `position`. See each chapter for details on which output attributes need to be set for vertex shaders.

For details on the assembly language and vertex shaders provided by DMPGL 2.0, see the separate *Vertex Shader Reference Manual*.

3.3 Geometry Shaders

The series of procedures involved in the programmable geometry creation provided by DMPGL 2.0 complies with section 2.10 Vertex Shaders in the OpenGL ES 2.0 specifications, but like vertex shaders, several features are not supported.

You cannot create or attach arbitrary user-defined geometry shaders in DMPGL 2.0. You can only load precompiled shader programs provided by DMPGL 2.0 and attach those shader objects to program objects. DMPGL provides the following geometry shaders.

- Point shaders
- Line shaders
- Silhouette primitive shaders
- Catmull-Clark subdivision shaders
- Loop subdivision shaders
- Particle system shaders

Several shaders also create vertices and primitives based on vertex information obtained from vertex shaders.

`CreateShader` and **`DeleteShader`** are used to create and delete geometry objects. When creating a geometry object, set the first argument, ***`type`***, to `GEOMETRY_SHADER_DMP`.

Code 3-3 CreateShader

```
CreateShader(GEOMETRY_SHADER_DMP);
```

Just as with vertex shaders, **`ShaderSource`** and compilation using **`CompileShader`** are not supported. Shaders are loaded by **`ShaderBinary`**.

Code 3-4 ShaderBinary

```
void ShaderBinary(sizei count, const uint *shader,
         enum binaryformat, const void *binary, sizei length);
```

Specify `PLATFORM_BINARY_DMP` for the third argument, ***`binaryformat`***.

In DMPGL 2.0, you cannot use **`ShaderBinary`** to load geometry programs separately from vertex shader programs. To load geometry programs, call **`ShaderBinary`** with binary code that includes both a geometry program and a vertex program, as a pair. You must set the second argument,

shader, to a pointer to an array that has stored handles to both the vertex shader object and the geometry shader object.

The required vertex attributes are already decided for the geometry shaders provided by DMPGL 2.0, so you need only calculate and output the attributes targeted by the vertex shaders. The required vertex attributes differ between the various geometry shaders. For details, see Chapter 4 Primitives.

There are reserved uniforms in program objects with attached geometry shaders. These reserved uniforms all have undefined initial values. You must therefore always set values for the reserved uniforms. You cannot use the names of these reserved uniforms as symbols in user-defined vertex shaders. To get the location of a reserved uniform, use **`GetUniformLocation`**, just as with user-defined shaders. Because it is optional to attach geometry shaders, note that you may not be able to obtain uniform locations for some program objects.

When using geometry shaders to render, the ***mode*** argument for **`DrawElements`** and **`DrawArrays`** must be `GEOMETRY_PRIMITIVE_DMP`. Specifying any other value will generate an `INVALID_ENUM` error.

3.4 Fragment Shaders

The series of procedures involved in the programmable fragment processing provided by DMPGL 2.0 complies with section 2.10 Vertex Shaders in the OpenGL ES 2.0 specifications. However, some OpenGL ES 2.0 features are not supported.

You cannot create or attach arbitrary user-defined fragment shaders in the current DMPGL 2.0 implementation. You can only attach the reserved fragment shader object. In DMPGL 2.0, program objects must always have the reserved fragment shader attached to them.

The fragment shader object provided by DMPGL 2.0 has the following name.

- `DMP_FRAGMENT_SHADER_DMP`

Use **`AttachShader`** to attach the reserved fragment shader object. Set the second argument (the name of the shader object) to `DMP_FRAGMENT_SHADER_DMP`. The creation and deletion of shaders using **`CreateShader`** and **`DeleteShader`** is not supported. As with vertex shaders and geometry shaders, you cannot use **`ShaderSource`** or **`CompileShader`**. You also cannot use **`ShaderBinary`** for fragment shaders.

There are reserved uniforms in program objects that have attached the reserved fragment shader. These reserved uniforms do have initial values. You cannot use the names of these reserved uniforms as symbols in user-defined vertex shaders. For details on the initial values, see Appendix B Uniform State Table. To get the location of a reserved uniform, use **`GetUniformLocation`**, just as with user-defined shaders.

DMPGL 2.0 does not apply shader variables (section 3.8.1 of the OpenGL ES 2.0 specifications). It also does not use the shader inputs and shader outputs described in section 3.8.2 of the OpenGL ES 2.0 specifications. Texture access, described in the same section 3.8.2 of the OpenGL ES 2.0 specifications, entails the texture limitations given in Chapter 5 Rasterization of this document.

4 Primitives

This chapter explains how primitives are created. OpenGL ES 2.0 supports points, lines, and triangles. DMPGL 2.0 supports these features and also adds unique features that are not supported by OpenGL ES 2.0, such as silhouettes and subdivisions.

On the other hand, in DMPGL 2.0 some features are invoked differently. To render points and lines with OpenGL ES 2.0, **`DrawElements`** and **`DrawArrays`** are called with `POINTS`, `LINES`, `LINE_STRIP`, or `LINE_LOOP` specified as the ***`mode`*** argument. DMPGL 2.0 has a different method for rendering points and lines using reserved geometry shaders. Triangles are rendered in the same way as OpenGL ES 2.0. However, multisample rendering of triangles is not supported.

Section 4.1 Points and 4.2 Lines describe points and lines. Section 4.3 Silhouettes, 4.4 Subdivisions, and 4.5 Particle Systems describe silhouettes, subdivisions, and particle systems. For details on each of these topics, see the relevant sections.

DMPGL 2.0 introduces vertex state collections as an extended feature of vertex buffer objects. These are described in section 4.6 Vertex State Collections.

4.1 Points

This section describes functionality for rendering points. Points can be rendered either by taking each vertex as the center of a point, or as point sprites. You can set the size of each point as well as enable and disable distance attenuation on a per-point basis.

4.1.1 How to Use Points

To render a point with DMPGL 2.0, link a user vertex shader to one of the reserved geometry shaders `DMP_pointN.obj` (where `N` = 0,1,2,3,4,5,6). Specify a geometry shader whose number 0-6 at the end of its name is the same as the number of vertex attributes output by the vertex shader, not counting vertex coordinates and point size.

To use a point geometry shader, the vertex shader must output at least two vertex attributes, the vertex coordinates and point size, to predetermined registers. To set vertex coordinates as a vertex attribute to output, specify `position` for ***`data_name`***, corresponding to the vertex shader assembly code **`#pragma output_map`**. Likewise, to set the point size as an attribute to output, specify `generic` for ***`data_name`***, corresponding to **`#pragma output_map`**. When you set the ***`mapped_register`*** corresponding to **`#pragma output_map`**, you must set the lowest-numbered output register for the vertex coordinates and the next lowest-numbered output register for the point size. You can set all other vertex attributes freely with **`#pragma output_map`**. You must also set the reserved uniform `dmp_Point.viewport` by calling **`glUniform2fv`** with the 2-component array (1/viewport width, 1/viewport height) specified for ***`value`***.

The points described by section 2.6.1 Primitive Types of the OpenGL ES 2.0 specifications do not exist in DMPGL 2.0. You cannot call **`DrawElements`** and **`DrawArrays`** with `POINTS` specified for ***`mode`***.

4.1.2 Point Size

The description of point size in section 3.3 Points of the OpenGL ES 2.0 specifications does not apply to DMPGL 2.0. In DMPGL 2.0, the point size is specified by a vertex attribute. You must either use **`VertexAttrib`** to specify a fixed size or **`VertexAttribPointer`** to use a vertex array to specify a size for each point. There is no upper limit on point size. Behavior is undefined for a point size of 0 or less.

As a DMP-specific feature, you can enable or disable distance attenuation of individual point sizes. When distance attenuation is disabled, the point size is multiplied by the vertex clipping coordinate w_c and that effect is then canceled by w_c division during transformation to window coordinates. When distance attenuation is enabled, the aforementioned process is skipped. To configure distance attenuation, call **`Uniform1i`** on the reserved uniform `dmp_Point.distanceAttenuation` with `TRUE` or `FALSE` specified for ***`value`***. Distance attenuation is enabled by `TRUE` and disabled by `FALSE`.

4.1.3 Point Sprites

In a point sprite, a point's texture coordinates are replaced with texture coordinates for point sprites. To use point sprites, link a user vertex shader object to one of the reserved geometry shaders `DMP_pointSpriteN_T.obj` (where `N` = 0,1,2,3 and `T` = 1,2,3). Specify a reserved geometry shader whose number `N(0,1,2,3)` in its name `DMP_pointSpriteN_T.obj` is the same as the number of vertex attributes output by the vertex shader, not counting the vertex coordinates, point size, and texture coordinates. Also, the number `T(1,2,3)` in the specified reserved geometry shader's name must be the same as the number of texture coordinates.

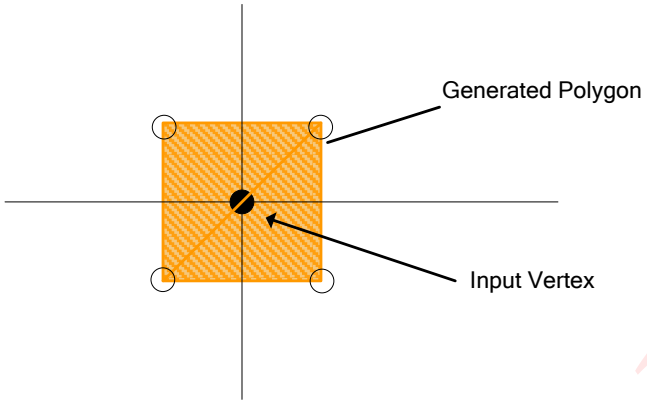
Vertex shaders must also use **`#pragma output_map`** to configure output for all texture coordinates used. When you configure output for texture coordinates in ***`mapped_register,`*** specify the lowest-numbered output register after the point size. When more than one texture coordinate is used, you must pack two texture coordinates into a single output register, using the register's XY components for the first set of texture coordinates and its ZW components for the second. The shader must output dummy values to output registers to which texture coordinates are assigned. This dummy output data is replaced with the point sprite's texture coordinates by the geometry shader.

In texture coordinates for point sprites, the texture coordinate *s* is 0 at the left edge of a point sprite and 1 at the right edge. In the same way, the texture coordinate *t* is 0 at the bottom edge of a point sprite and 1 at the top. The texture coordinates *r* and *q* are fixed at 0 and 1, respectively.

4.1.4 Point Rendering Method

In DMPGL 2.0, the geometry shader renders a point by taking the vertex input to it by the vertex shader as the center of the point, then rendering two triangles to form a square having the specified point size as the length of each side. This is not adjusted to match the vertex coordinate grid or in any other way. The two triangles are rendered to be front-facing with a counterclockwise winding (displayed when `CCW` is specified for `FrontFace`).

Figure 4-1 How Points Are Rendered



4.1.5 Point Clipping

Points are clipped according to the following equation.

Equation 4-1 Point Clipping

$$-w_c \leq z_c \leq 0$$

4.1.6 Multisample Rendering

DMPGL 2.0 does not incorporate the series of processes described in section 3.3.1 Point Multisample Rasterization of the OpenGL ES 2.0 specifications.

4.1.7 List of Reserved Uniforms

The following table shows settings for the reserved uniforms that are used by points.

Table 4-1 Reserved Uniform Settings for Points

Uniform	Type	Value
`dmp_Point.viewport`	`vec2`	(1/viewport width, 1/viewport height) Undefined by default
`dmp_Point.distanceAttenuation`	`bool`	TRUE FALSE Undefined by default

4.2 Lines

This section describes functionality for rendering lines. The line feature renders a line that connects two vertices. There are two types of lines: *separate lines* and *strip lines*. Separate lines are lines that are rendered independently for each pair of vertices. In other words, the first vertex specifies the

starting point of the first line and the next vertex specifies the ending point of the first line. The next two vertices specify the starting and ending points, respectively, of the next line. This is repeated for all vertices. A strip line, on the other hand, is rendered as a continuous line that connects all vertices. In other words, the first vertex specifies the starting point of the first line and the next vertex specifies both the ending point of the first line and the starting point of the next line. This is repeated for all vertices. You can also specify the width of lines.

4.2.1 How to Use Lines

To render lines in DMPGL 2.0, link a user vertex shader to one of the reserved geometry shaders: `DMP_separateLineN.obj` (where `N` = 0,1,2,3,4,5,6) for separate lines, or `DMP_stripLineN.obj` (where `N` = 0,1,2,3,4,5,6) for strip lines. Specify a reserved geometry shader whose number 0-6 at the end of its name is the same as the number of vertex attributes output by the vertex shader, not counting vertex coordinates.

To use a line geometry shader, the vertex shader must output at least the vertex coordinates. To set vertex coordinates as a vertex attribute to output specify `position` for ***data_name***, corresponding to the vertex shader assembly code **`#pragma output_map`**. When you set the ***mapped_register*** corresponding to **`#pragma output_map`**, you must set the lowest-numbered output register for the vertex coordinates. You can set all other vertex attributes freely with **`#pragma output_map`**.

The line strips, line loops, and separate lines discussed in section 2.6.1 Primitive Types of the OpenGL ES 2.0 specifications do not exist in DMPGL 2.0. You cannot call **`DrawElements`** or **`DrawArrays`** with `LINE_STRIP`, `LINE_LOOP`, or `LINES` specified for ***mode***.

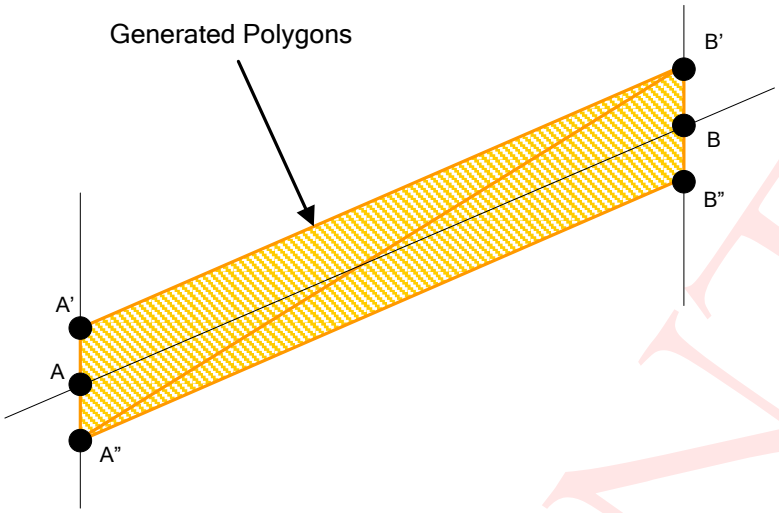
4.2.2 Line Width

To set the line width, call **`Uniform4fv`** on the reserved uniform `dmp_Line.width` with the 4 components (viewport width / line width, viewport height / line height, viewport width x viewport height, 2 / line width) set for ***value***. This reserved uniform is undefined by default and must be set. Behavior is undefined for a line width of 0.0 or less. **`LineWidth`**, mentioned in section 3.4 Line Segments of the OpenGL ES 2.0 specifications, does not exist in DMPGL 2.0.

4.2.3 Line Rendering Method

DMPGL 2.0 does not incorporate the series of processes described in sections 3.4.1 Basic Line Segment Rasterization and 3.4.2 Other Line Segment Features of the OpenGL ES 2.0 specifications. In DMPGL 2.0, the geometry shader renders a line by creating a rectangle formed of two polygons from the two vertices input to it by the vertex shader. The two newly created polygons comprise four vertices (A', A'', B', and B''), which are created in the y-direction from the two input vertices A and B (depending on the orientation of the line segment AB, the polygons might also be created in the x-direction). The line segments A'A'' and B'B'' are centered on A and B, respectively, and are as long as the line width. This is not adjusted to match the output vertex coordinate grid or in any other way. The two polygons are rendered to be front-facing with a counterclockwise winding (displayed when `CCW` is specified for `FrontFace`).

Figure 4-2 Line Rendering Method



4.2.4 Multisample Rendering

DMPGL 2.0 does not incorporate the series of processes described in section 3.4.4 Line Multisample Rasterization of the OpenGL ES 2.0 specifications.

4.2.5 List of Reserved Uniforms

The following table shows shows settings for the reserved uniforms that are used by lines.

Table 4-2 Reserved Uniform Settings for Lines

Uniform	Type	Value
`dmp_Line.width`	`vec4`	$\left(\frac{viewport\ width}{line\ width}, \frac{viewport\ height}{line\ width}, viewport\ width \times viewport\ height, \frac{2.f}{line\ width}\right)$ Undefined by default

4.3 Silhouettes

This section describes functionality for rendering silhouettes. Silhouettes introduce a new primitive type, *silhouette primitives*, which allow you to render silhouette edges around polygons. By combining the silhouette edges of polygons with shadow functionality, you can render *soft shadows*, also called *penumbras*.

4.3.1 How to Use Silhouettes

To render silhouettes, link a user vertex shader to the reserved geometry shader `DMP_silhouetteTriangle.obj` for silhouette triangles, or `DMP_silhouetteStrip.obj` for

silhouette strips. To use a silhouette geometry shader, the vertex shader must output three vertex attributes: the vertex coordinates, color, and normal (you cannot output any vertex attributes other than these). To set vertex coordinates as a vertex attribute to output, specify `position` for ***data_name***, corresponding to the vertex shader assembly code **`#pragma output_map`**. Likewise, to set the color as an attribute to output, specify `color` for ***data_name***, corresponding to **`#pragma output_map`**. To set the normal as an attribute to output, specify `generic` for ***data_name***, corresponding to **`#pragma output_map`**. When you set the ***mapped_register*** corresponding to **`#pragma output_map`**, you must start from the lowest-numbered output register for the vertex coordinates, then set the next-lowest registers for the color and the normal, in that order.

Calls to **`DrawArrays`** are not supported when silhouette shaders are in use. Although silhouette primitives (described in section 4.3.2 Silhouette Primitives) are used to render vertex data, the vertex data and index data must be used by **`glBindBuffer`** and **`glBufferData`** for the vertex buffer. You must also call **`Disable`** with `CULL_FACE` specified for ***cap*** to disable culling before you call **`DrawElements`**.

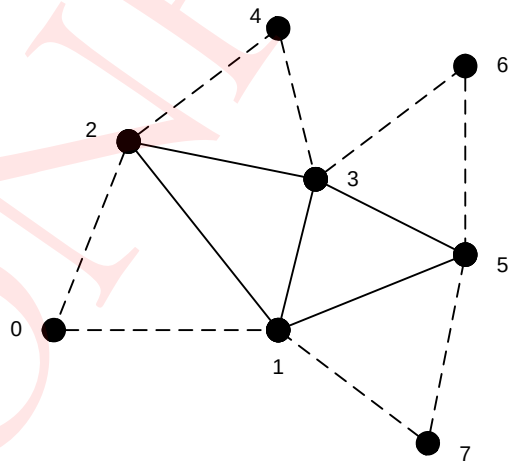
4.3.2 Silhouette Primitives

Silhouettes are are rendered using the new silhouette primitive type. A silhouette primitive comprises four triangles: one at the center and three around it, each sharing an edge with the center triangle. These four adjacent triangles together are called a *triangle with neighborhood* (hereinafter TWN). When an edge shared by the center triangle and an adjacent triangle is determined to be a silhouette edge, a rectangular polygon is generated there as the silhouette edge.

When TWN triangles have edges that are not shared with any other triangles, it is possible to specify another adjacent triangle formed of two points on those unshared edges and taking as its third vertex one of the vertices of the center triangle.

Triangles with edges shared by three or more triangles have not been taken into account.

Figure 4-3 Silhouette Primitive Example

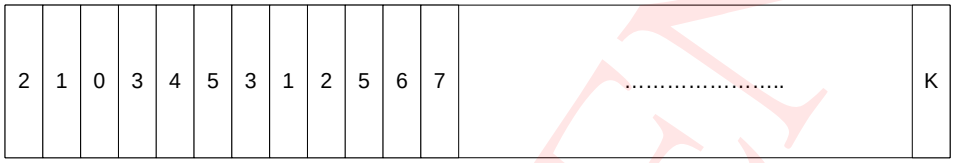


The center triangle (2,1,3) has the three adjacent triangles (2,0,1), (3,1,5), and (2,3,4).

You can specify TWN vertex indices as either silhouette triangles or silhouette strips.

With silhouette triangles, a single TWN comprises six vertices. To keep the center triangle front-facing, two of its vertices are taken as the first and second vertices of the TWN. The third vertex of the TWN is the remaining vertex of the adjacent triangle that shares an edge with these first two vertices. Next, the remaining vertex of the center triangle is taken as the fourth vertex. The fifth vertex is the remaining vertex of the adjacent triangle that shares the first and fourth vertices. Finally, the sixth vertex is the remaining vertex of the adjacent triangle that shares the second and fourth vertices. Figure 4-4 gives the TWN indices for the TWNs having the center triangles (2,1,3) and (3,1,5) in the silhouette primitive example shown in Figure 4-3.

Figure 4-4 Silhouette Triangle Indices



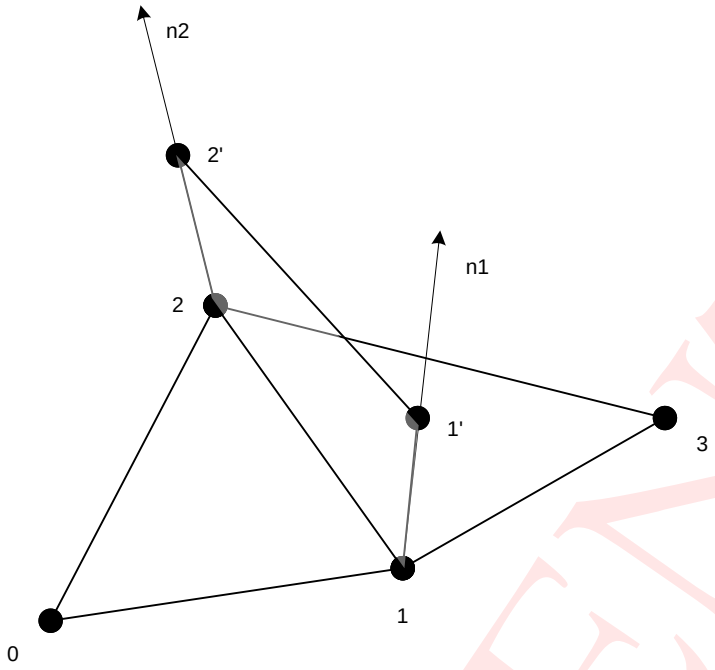
With silhouette strips, the first six vertices specify a single TWN and each following pair of vertices specifies a single TWN. The indices for the first six vertices are determined according to the same rules as silhouette triangles. Afterwards, the last adjacent triangle specified for the N'th TWN is taken as the center triangle of the N+1'th TWN. The second and third vertices of the center triangle for the N'th TWN are taken as the first and second vertices of the center triangle for the N+1'th TWN. Likewise, the last vertex of the N'th TWN is taken as the third vertex of the center triangle for the N+1'th TWN. The two vertices explicitly specified for the N+1'th TWN are the remaining vertices of the adjacent triangles that, respectively, share an edge with the first and third vertices and second and third vertices of the center triangle for that TWN. Figure 4-5 gives the TWN indices for the TWNs having the center triangles (2,1,3) and (3,1,5) in the silhouette primitive example shown in Figure 4-3.

Figure 4-5 Silhouette Strip Indices

4.3.3 Method for Creating Silhouette Edges

To render silhouette edges, new rectangular polygons are generated on the edges of the center triangle for a TWN. Silhouette edges are created when the center triangle for a TWN is front-facing and one of its adjacent triangles is not front-facing. The center triangle and the adjacent triangle share two vertices; two additional vertices are added along the normal vectors of those shared vertices, and those four vertices are used to generate the rectangular polygon that forms the silhouette edge.

Figure 4-6 Creation of a Silhoutte Rectangle by Edge 1-2 and Normal Vectors n1 and n2



Vertices 1' and 2' are the newly generated vertices. The generated silhouette rectangle comprises vertices 2, 1, 1', and 2'.

In the following equation, the vertices on the edge (1 and 2 in Figure 4-6) are given by *(x, y, z, w)* and the vertices along the normal vectors (1' and 2' in Figure 4-6) are given by *(x', y', z', w')*.

Equation 4-2 Relationship Between Silhouette Rectangle Vertices

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} x + xscale\_factor \times nx \times w\_scale \\ y + yscale\_factor \times ny \times w\_scale \\ z \\ w \end{pmatrix}$$

The normal vector output by the vertex shader is taken as *(nx, ny, nz)* in this equation. To set *xscale_factor* and *yscale_factor*, call **`Uniform2fv`** on the reserved uniform `dmp_Silhouette.width` with ***value*** specified as a 2-component floating-point vector. The following equation shows how *yscale_factor* is calculated.

Equation 4-3 Calculation of yscale_factor

$$yscale\_factor = xscale\_factor \times viewport.width \div viewport.height$$

When **`Uniform1i`** is called on the reserved uniform `dmp_Silhouette.scaleByW` with `TRUE` specified for ***value***, *w_scale* takes the value of the *w* component of the vertices on the edge (vertices 1 and 2 in Figure 4-6). A value of 1.0 is used for *w_scale* when `FALSE` is specified for ***value***.

4.3.4 Vertex Shaders When Silhouettes Are in Use

Vertex shaders must output normal vector data when silhouettes are in use. To obtain this output data, the vertex shader must take the vertex normal vectors input to it by the application, apply the modelview transformation, and normalize them over their *x* and *y* components. In other words, after the modelview transformation the output normal vector *n'* must be normalized from the eye-coordinate normal vector *n = (nx, ny, nz)* as follows.

Equation 4-4 Output Normal Vectors

$$n' = \frac{n}{\sqrt{nx^2 + ny^2}}$$

4.3.5 Silhouette Colors

The vertex color of the vertices lying on the center and adjacent triangles of the TWN is used for the color of those vertices of the silhouette edge. Set the color of the new vertices created along the normal vectors to form the silhouette edge by calling **`Uniform4fv`** on the reserved uniform `dmp_Silhouette.color` with the 4-component floating-point array (R, G, B, A) specified for ***`value`***. The configured values are clamped between 0 and 1.

4.3.6 Front-Facing Settings

When silhouettes are in use, you must call **`Uniform1i`** on the reserved uniform `dmp_Silhouette.frontFaceCCW` with ***`value`*** set to `TRUE` or `FALSE` to match the **`FrontFace`** setting. Specify `TRUE` when **`FrontFace`** is set to `CCW` and `FALSE` when it is set to `CW`.

4.3.7 Creating Silhouette Edges on Open Edges

When the center triangle of a TWN has an edge that is not shared with any other triangles (an open edge), the adjacent triangle for that edge is specified so that it comprises the two points on the edge and the remaining vertex of the center triangle. In other words, the adjacent triangle is specified so that it folds over and exactly overlaps the center triangle. To configure silhouette edges to either always or never be created on open edges, call **`Uniform1i`** on the reserved uniform `dmp_Silhouette.acceptEmptyTriangles` with ***`value`*** set to `TRUE` or `FALSE`. When it is `TRUE`, silhouette edges are always generated on the open edges of polygons, such as described above. When `FALSE`, silhouette edges are never generated on open edges.

Unlike normal silhouette edges, open-edge silhouettes are rendered by a line that connects the two vertices on the edge. In other words, they are rendered just like line primitives and are unaffected by the normal vectors for the vertices on the edge.

Open-edge silhouettes are configured separately from the settings for ordinary silhouette edges. You can configure the silhouette color, width of the silhouette edge, and bias in the view direction. You can also enable or disable scaling by the *w* component of vertex coordinates with respect to either or both the width of the silhouette edge and the bias in the view direction.

Open-edge silhouettes are rendered in a configured solid color; the edge's vertex colors are not used. To configure the color of open-edge silhouettes, call **`Uniform4fv`** on the reserved uniform `dmp_Silhouette.openEdgeColor` with ***`value`*** set to the color.

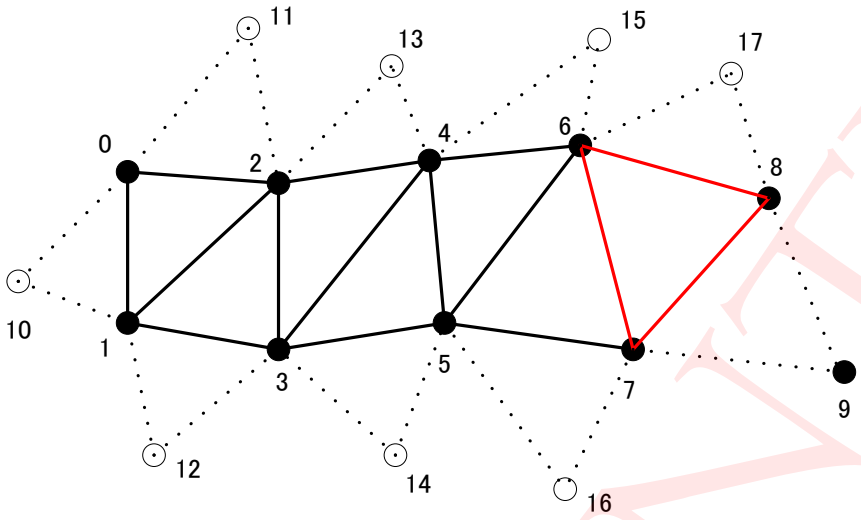
The width of an open-edge silhouette is configured in the same way as a line primitive. To set it, call **`Uniform4fv`** on the reserved uniform `dmp_Silhouette.openEdgeWidth` with ***`value`*** set to the four components (viewport width / silhouette width, viewport height / silhouette width, viewport width x viewport height, 2 / silhouette width). You can scale this silhouette width by the *w* component of the vertex coordinates. To scale by the *w* component, call **`Uniform1i`** on the reserved uniform `dmp_Silhouette.openEdgeWidthScaleByW` with ***`value`*** set to `TRUE`. To not scale the width of open-edge silhouettes by the *w* component, specify `FALSE`.

You can apply a bias in the view direction for open-edge silhouettes. To configure this, call **`Uniform1fv`** on the reserved uniform `dmp_Silhouette.openEdgeDepthBias` with ***`value`*** set to the bias factor. Setting a positive value will move the open-edge silhouette toward the viewpoint and a negative value will move it away. You can scale this bias factor by the *w* component of the vertex coordinates. To scale by the *w* component, call **`Uniform1i`** on the reserved uniform `dmp_Silhouette.openEdgeDepthBiasScaleByW` with ***`value`*** set to `TRUE`. To not scale the view-direction bias of open-edge silhouettes by the *w* component, specify `FALSE`.

4.3.8 Specifying Multiple Strip Arrays

To render silhouette strip arrays used for multiple silhouettes (hereafter called simply "strip arrays") with a single call to **`DrawElements`**, you must specify the end of each strip array. After the first six vertices in a strip array, even-numbered vertices always belong to center triangles. If the same vertex value is used for two even-numbered vertices in a row, however, it indicates that the end of that strip array has been reached. The strip array specifies the remaining vertex of the adjacent triangle whose edge is shared with the second and third vertices of the final center triangle, then the strip array ends. Another strip array can then be re-specified from its starting six vertices. When you reuse the first six vertices of a strip array to specify a second or subsequent strip array, its first center triangle can start with a different orientation than the **`FrontFace`** setting. In this case, specifying the first vertex of that strip array twice in a row indicates that it starts with the opposite orientation from the **`FrontFace`** setting.

Figure 4-7 Specifying the End of a Silhouette Strip Array



For the example in Figure 4-7, a silhouette strip array that starts with the triangle at vertices 0, 1, and 2 and ends with the triangle at vertices 6, 7, and 8 would have the vertex indices {0, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, **8**, 17, **8**, **9**}. If this silhouette strip array started with the opposite orientation from the **`FrontFace`** setting, it would have the vertex indices {**0**, **0**, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, **8**, 17, **8**, **9**}.

Although you can demarcate between multiple silhouette strip arrays by specifying the end of each in this way, failure to specify the end of the very last silhouette strip in this same way and instead connect it to another triangle sometimes causes double silhouettes to be generated on that edge. When applying alpha blending or similar operations on silhouettes, it is best to specify the end of every strip array, as described in this section.

4.3.9 List of Reserved Uniforms

The following table shows settings for the reserved uniforms that are used by silhouettes.

Table 4-3 Reserved Uniform Settings for Silhouettes

Uniform	Type	Value
`dmp_Silhouette.width`	`vec2`	Each component is 0.0 or greater Undefined by default
`dmp_Silhouette.scaleByW`	`bool`	TRUE FALSE Undefined by default
`dmp_Silhouette.color`	`vec4`	Each component is between 0.0 and 1.0 Undefined by default

Uniform	Type	Value
`dmp_Silhouette.frontFaceCCW`	`bool`	TRUE FALSE Undefined by default
`dmp_Silhouette.acceptEmptyTriangles`	`bool`	TRUE FALSE Undefined by default
`dmp_Silhouette.openEdgeColor`	`vec4`	Each component is between 0.0 and 1.0 Undefined by default
`dmp_Silhouette.openEdgeWidth`	`vec4`	$\left(\frac{viewport\ \ width}{silhouette\ \ width}, \frac{viewport\ \ height}{silhouette\ \ width}, viewport\ width \times viewport\ height, \frac{2.f}{silhouette\ \ width}\right)$ Undefined by default
`dmp_Silhouette.openEdgeDepthBias`	`float`	Any Undefined by default
`dmp_Silhouette.openEdgeWidthScaleByW`	`bool`	TRUE FALSE Undefined by default
`dmp_Silhouette.openEdgeDepthBiasScaleByW`	`bool`	TRUE FALSE Undefined by default

4.4 Subdivisions

This section describes subdivision features. DMPGL 2.0 provides a complete implementation for both Catmull-Clark subdivision and Loop subdivision. Both features use geometry shaders to subdivide primitives. The Catmull-Clark method splits up quads to create more detailed quads, while Loop subdivision splits up triangles to create more detailed triangles. Subdivisions use a new primitive type, the *subdivision patch*, to subdivide input polygons and thereby generate finer polygons to insert into a later stage of the fragment pipeline. You can set the subdivision level. Catmull-Clark and Loop subdivision differ in several ways, including how they are used, the definition of their subdivision patches, and their limitations.

4.4.1 Catmull-ClarkSubdivision

This section explains Catmull-Clark subdivision. When this section simply refers to "subdivision," it indicates Catmull-Clark subdivision.

4.4.1.1 How to Use Catmull-Clark Subdivision

To use Catmull-Clark subdivision, link a user vertex shader to one of the reserved geometry shaders `DMP_subdivisionN.obj` (where `N` = 0,1,2,3,4,5,6). The number of attributes sent by the vertex shader determines which of these to use. To use Catmull-Clark subdivision, the vertex shader must output at least the vertex coordinates. To set vertex coordinates as a vertex attribute to output, specify

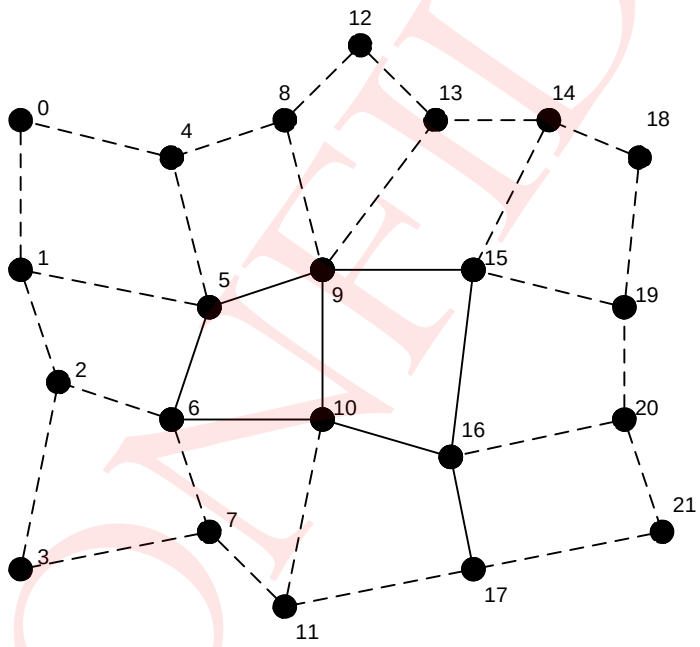
`position` for ***`data_name`***, corresponding to the vertex shader assembly code **`#pragma output_map`**. When you set the ***`mapped_register`*** corresponding to **`#pragma output_map`**, you must set the lowest-numbered output register for the vertex coordinates. You can set all other vertex attributes freely with **`#pragma output_map`**, but when quaternions are output by the vertex shader, you must set them to the next-lowest output register number after the vertex coordinates. The number (0-6) at the end of the reserved geometry shader name must have the same value as the number of vertex attributes, not counting the vertex coordinates.

Calls to **`DrawArrays`** are not supported when subdivision shaders are in use. Although Catmull-Clark subdivision patches (described in section 4.4.1.2 Definition of Catmull-Clark Subdivision Patches) are used to render vertex data, the vertex data and index data must be used by **`BindBuffer`** and **`BufferData`** for the vertex buffer.

4.4.1.2 Definition of Catmull-Clark Subdivision Patches

DMPGL 2.0 introduces a new primitive type, *Catmull-Clark subdivision patches*. Catmull-Clark subdivision patches are made up of all vertices in the polygon(s) to subdivide and all vertices that share an edge with those vertices. In Catmull-Clark subdivision, an *extraordinary vertex* is one that does not have 4 adjoining edges. The number of adjoining edges to a vertex is called the *valence* of the vertex. A Catmull-Clark subdivision patch must comprise only quads and must not have more than one extraordinary point. In general, after one or two subdivisions all subdivision patches are guaranteed to satisfy these two conditions.

Figure 4-8 Example of a Catmull-Clark Subdivision Patch



Vertex 9 is an extraordinary vertex with a valence of 5.

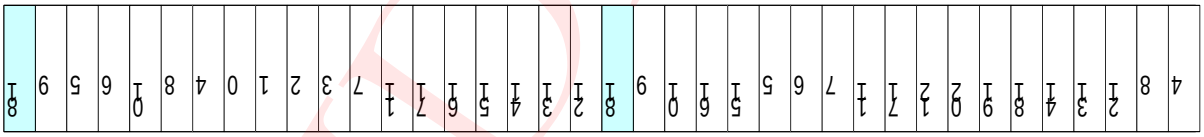
4.4.1.3 Vertex Indices for Catmull-Clark Subdivision Patches

The number of vertices in a patch depends on the valence of its extraordinary vertex. In other words, a subdivision patch is not a primitive with a fixed vertex count. As a result, the vertex count of a subdivision patch is stored at the start of the vertex indices used by that patch. The patch's vertex indices are stored next. A single patch can contain up to 32 vertices. A patch's behavior is undefined if it is specified with more than 32 vertices.

The order of the vertex indices that are stored subsequent to the patch's vertex count is determined according to the following rules.

1. The first vertex is either a vertex of the central quad to be subdivided or the extraordinary vertex, if one exists.
2. The second, third, and fourth vertices are the remaining 3 vertices of the central quad. They are numbered in either clockwise or counterclockwise order so that the quad is front-facing according to the **`FrontFace`** setting.
3. The fifth vertex is the vertex that forms an edge with the first vertex and a surface with both the first and second vertices.
4. The remaining vertices are specified in the same direction (clockwise or counterclockwise) as the central quad. Figure 4-9 shows the vertex indices for the example in Figure 4-8.

Figure 4-9 Array of Vertex Indices for a Patch



The patch size is stored at the start of the indices for each patch.

The indices for each patch must be stored so that patches sharing the same extraordinary vertex are consecutive. Vertex 9 is the extraordinary vertex in Figure 4-8, so the patches with the central quads (9,5,6,10), (9,10,16,15), (9,15,14,13), (9,13,12,8), and (9,8,4,5) must have their indices stored consecutively. If patches sharing the same extraordinary point are not consecutive in the index array, continuity between subdivision patches is not guaranteed and the mesh could develop holes.

4.4.2 Loop Subdivision

This section explains Loop subdivision. When this section simply refers to "subdivision," it indicates Loop subdivision.

4.4.2.1 How to Use Loop Subdivision

To use Loop subdivision, link a user vertex shader to one of the reserved geometry shaders `DMP_loopSubdivisionN.obj` (where `N` = 1,2,3,4). The number of output registers used by the vertex shader determines which of these to use. To use a geometry shader for Loop subdivision, the vertex shader must output at least the vertex coordinates and valences. The valence is the number of

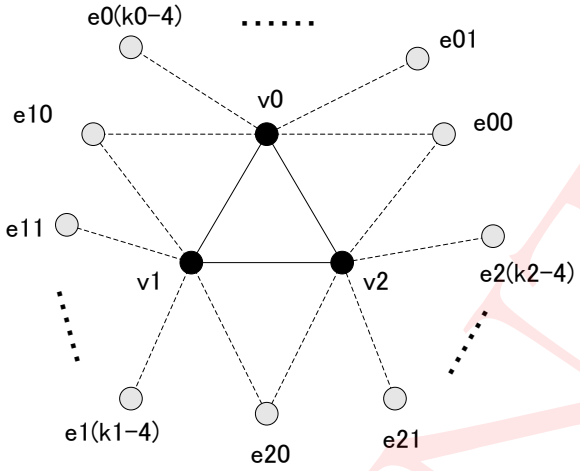
edges adjacent to a vertex. To set vertex coordinates as a vertex attribute to output, specify `position` for ***`data_name`***, corresponding to the vertex shader assembly code **`#pragma output_map`**. To set the valence as a vertex attribute to output, specify `generic` for ***`data_name`***, corresponding to the vertex shader assembly code **`#pragma output_map`**. When you set the ***`mapped_register`*** corresponding to **`#pragma output_map`**, you must set the lowest-numbered output register for the vertex coordinates. You can set all other vertex attributes freely with **`#pragma output_map`**, but when quaternions are output by the vertex shader, you must set them to the next-lowest output register number after the vertex coordinates. The valence must be set to the highest-numbered output register. In other words, you must use output registers first for the vertex coordinates and then for quaternions, miscellaneous attributes, and the valence, in that order. The number (1-4) at the end of the reserved geometry shader name must have the same value as the number of output registers used for vertex attributes, not counting the valence. Because the upper limit is 4, vertex shaders can use up to four output registers. Note that, unlike Catmull-Clark subdivision, this is the number of output *registers*, not the number of output *attributes*. Although you can use only up to four output registers, you can use more than four output attributes by setting more than one attribute in a single register (for example, you could set texture coordinate 0 in `o1.xy` and texture coordinate 1 in `o1.zw`). However, quaternions cannot be set to the same output register as another attribute.

Calls to **`DrawArrays`** are not supported when subdivision is in use. Although Loop subdivision patches (described in section 4.4.2.2 Definition of a Loop Subdivision Patch) are used to render vertex data, the vertex data and index data must be used by **`BindBuffer`** and **`BufferData`** for the vertex buffer.

4.4.2.2 Definition of a Loop Subdivision Patch

DMPGL 2.0 introduces a new primitive type, the *Loop subdivision patch*. Loop subdivision patches are made up of all vertices in the polygon(s) to subdivide and all vertices that share an edge with those vertices. Each vertex in a Loop subdivision patch must have a valence of at least 3 and no more than 12, and the sum of the valances of all vertices in each single polygon of a Loop subdivision patch must be 29 or less. If a Loop subdivision patch contains a vertex with a valence less than 3, this can be handled by adding imaginary vertices to increase its valence.

Figure 4-10 A Loop Subdivision Patch



In Figure 4-10, the polygon with vertices `v0`, `v1`, and `v2` is the target of subdivision. `k0`, `k1`, and `k2` are the valences of `v0`, `v1`, and `v2`, respectively. If two vertices share an edge, they are said to be adjacent. Vertices that are adjacent to `v0` but not adjacent to `v1` are named `e00, e01, … , e0(k0-4)` in counterclockwise order of adjacency, starting with the vertices adjacent to `v2`. In the same way, vertices that are adjacent to `v1` but not adjacent to `v2` are named `e10, e11, … , e1(k1-4)` in counterclockwise order of adjacency, starting with the vertices adjacent to `v0`. Vertices that are adjacent to `v2` but not adjacent to `v0` are named `e20, e21, … , e2(k2-4)` in counterclockwise order of adjacency, starting with vertices adjacent to `v1`. The patch comprises all of these vertices.

4.4.2.3 Vertex Indices of a Loop Subdivision Patch

The total number of vertices in a patch depends on the valence of the vertices. In other words, a subdivision patch is not a primitive with a fixed vertex count. As a result, the size of each subdivision patch is stored in the first vertex index that it uses. When a center triangle with the three vertices `v0`, `v1`, and `v2` is subdivided, the patch size is three vertices larger than the sum of all vertices that share an edge with vertex `v0`, `v1`, or `v2`.

Using Figure 4-10 as an example, the order of the vertex indices that are stored subsequent to the patch's vertex count is determined according to the following rules.

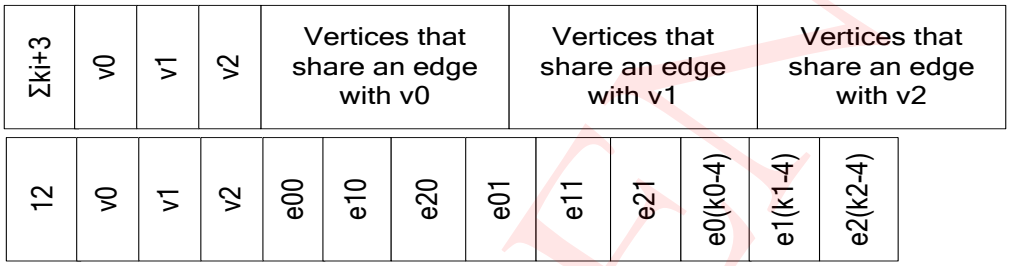
1. The three vertices `v0`, `v1`, and `v2` of the center triangle that is the target of subdivision are taken as the first, second, and third vertices in line with the **`FrontFace`** setting.
2. Next come all vertices that share an edge with `v0`, followed by all vertices that share an edge with `v1`, followed by all vertices that share an edge with `v2`, in order.
3. A fixed value of 12 is specified next, followed by the vertices `v0`, `v1`, and `v2` again.
4. The vertices `e00`, `e10`, and `e20` are specified after that. In this case, `e00` is the vertex that shares

an edge with both `v0` and `v2` but is not `v1`. In the same way, `e10` and `e20` correspond to the edges `v0v1` and `v1v2`, respectively.

5. The vertices `e01`, `e11`, and `e21` are specified after that. `e01` is the vertex that shares an edge with `v0` and is adjacent to `v0` next after `e00` in counterclockwise order. In the same way, `e11` and `e21` correspond to vertices `v1` and `v2`, respectively.

6. Last, the vertices `e0(k0-4)`, `e1(k1-4)`, and `e2(k2-4)` are specified. Here, `k0`, `k1`, and `k2` are the valences of `v0`, `v1`, and `v2`, respectively. `e0(k0-4)` is the vertex that shares an edge with `v0` and is adjacent to `v0` next after `e10` in clockwise order. In the same way, `e1(k1-4)` and `e2(k2-4)` correspond to vertices `v1` and `v2`, respectively.

Figure 4-11 Array of Vertex Indices for a Loop Subdivision Patch



Vertices that share an edge with either `v0`, `v1`, or `v2` may be specified in any order for that vertex, but they must use the same order in all patches. For example, vertices that share an edge with `v0` must have the same order in all patches that contain `v0`.

`v1`, `e00`, `e01`, and `e0(k0-4)` are also included in the vertices that share an edge with `v0`. These vertices and the corresponding vertices for `v1` and `v2` are specified more than once. This definition was nevertheless adopted for efficiency and to simplify the geometry shader implementation. Because the cache is hit after the vertices are processed, these duplications within the definition do not actually cause a performance penalty.

4.4.3 How to Process Subdivisions

The geometry shaders for Loop subdivision and Catmull-Clark subdivision accumulate the processed vertex data and subdivide a single patch as soon as all the vertex data for that single patch has been input. When subdivided, a polygon has new vertices created both inside it and on its edges and its original vertex positions are changed.

You can specify the subdivision level by calling **`Uniform1f`** on the reserved `uniform dmp_Subdivision.level` with ***`value`*** set to 0, 1, or 2. Increasing the level results in a finer subdivision. Although Loop subdivision does not generate vertices at level 0, the position of the vertex coordinates change and the resulting model therefore does not match the original. At level 0, Catmull-Clark subdivision generates a single new vertex only at the center of rectangular patches; the position of each of the vertex coordinates is also changed.

All vertex attributes other than the coordinates of the new vertices are calculated by interpolating the vertex attributes of the original polygon targeted for subdivision. You must configure the shader to either use or not use quaternions as vertex attributes, because quaternions need to be interpolated differently from the other vertex attributes. To configure the use of quaternions, call **`Uniform1i`** on the reserved uniform `dmp_Subdivision.fragmentLightingEnabled` with ***`value`*** set to either `TRUE` or `FALSE`. Specify `TRUE` to use quaternions and `FALSE` otherwise.

4.4.4 List of Reserved Uniforms

The following table shows settings for the reserved uniforms that are used by subdivisions.

Table 4-4 Reserved Uniform Settings for Subdivisions

Uniform	Type	Value

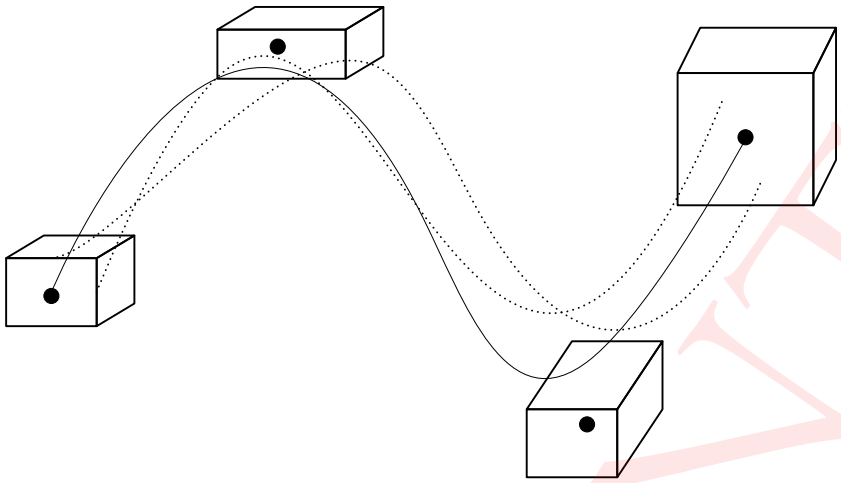
`dmp_Subdivision.level`	`float`	`0`
		`1`
		`2`
		Undefined by default

`dmp_Subdivision.fragmentLightingEnabled`	`bool`	`TRUE`
		`FALSE`
		Undefined by default

4.5 Particle Systems

This section describes particle system features. Particle systems are defined as a set of point sprites. In a particle system, multiple point sprites are placed randomly along a Bézier curve obtained from configured control points. The particle system configures bounding boxes centered on each of the control points, and the control points that define the Bézier curve move randomly inside of these bounding boxes. In other words, each individual point sprite's Bézier curves are defined randomly within the bounding boxes. The geometry shader generates the size, color, and texture coordinates of each of these point sprites. The point sprites generated by a particle system are hereafter called *particles*. Each particle is rendered using two triangles just like a point primitive. These two triangles are rendered to be front-facing with a counterclockwise winding (displayed when `CCW` is specified for `FrontFace`).

Figure 4-12 Particle Bezier Curve Trajectory Created by Control Points



4.5.1 How to Use Particle Systems

To use a particle system with DMPGL 2.0, link a user vertex shader to one of the reserved geometry shaders `DMP_particleSystem_X_X_X_X.obj` (where `X` = 0 or 1). Each `X` (0 or 1) in the reserved geometry shader name indicates whether a particle system feature is on or off. Respectively, these features are: particle time clamping; texture coordinate rotation; use of the RGBA particle color components (or only the alpha component when this feature is off); and texture coordinate 2. Details are given in Table 4-6.

Calls to **`DrawArrays`** are not supported when particle systems are in use.

4.5.2 Input of Control Points

Particle systems require four control points to define a Bézier curve. As a result, the vertex shader that is linked to the particle system's geometry shader must output both the vertex coordinates for the control point and the size of the bounding box to center on that control point.

The vertex coordinates are output in clip space. If the size of a bounding box is given as radii of `Rx`, `Ry`, and `Rz` in object coordinates for the X, Y, and Z directions, the bounding box size is input as the three vectors `(Rx, 0, 0)`, `(0, Ry, 0)`, and `(0, 0, Rz)` converted from object space to clip space. The vertex shaders sets the output attributes as follows.

Code 4-1 Output Attributes for Control Point Bounding Boxes

```
#pragma output_map ( position , o0 )
#pragma output_map ( generic , o1 )
#pragma output_map ( generic , o2 )
#pragma output_map ( generic , o3 )
#pragma output_map ( generic , o4 )
```

The vertex coordinates in clip space are output to `o0`. When the bounding box's radii in the XYZ directions are given by `Rx`, `Ry`, and `Rz` in object coordinates, `o1`-`o4` are calculated and output as follows.

Equation 4-5 Calculation of Control Point Bounding Box Size

$$\begin{pmatrix} o1.x & o1.y & o1.z & 0 \\ o2.x & o2.y & o2.z & 0 \\ o3.x & o3.y & o3.z & 0 \\ o4.x & o4.y & o4.z & 0 \end{pmatrix} = M_{proj} \times M_{modelview} \times \begin{pmatrix} Rx & 0 & 0 & 0 \\ 0 & Ry & 0 & 0 \\ 0 & 0 & Rz & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

In this equation, M_{proj} and $M_{modelview}$ correspond to the projection and modelview matrices, respectively.

4.5.3 Particle Colors

The color of each particle is interpolated from the control point color according to the particle's position on the Bézier curve. You can configure two types of control point colors. The first uses all of the RGBA components and the second uses only the alpha component. To set all of the RGBA components, use one of the reserved geometry shaders `DMP_particleSystem_X_X_1_X.obj` and call **`UniformMatrix4fv`** on the reserved uniform `dmp_PartSys.color`, specifying a pointer to a 4x4 matrix that stores the RGBA values for the four control points. In this 4x4 matrix, store the components in RGBA order, starting with the first control point and ending with the fourth control point.

To set only the alpha component, use one of the reserved geometry shaders `DMP_particleSystem_X_X_0_X.obj` and call **`UniformMatrix4fv`** on the reserved uniform `dmp_PartSys.aspect`, specifying a pointer to a 4x4 matrix that stores the alpha values for the four control points in its fourth column. Store the alpha components in the 4x4 matrix starting with the first control point in the first row and ending with the fourth control point in the fourth row.

Using all of the RGBA components results in worse performance than using only the alpha components. We recommend using only the alpha component when you do not need the RGB components.

4.5.4 Particle Size

The size of each particle is determined by interpolating the particle sizes specified for the control points according to the particle's position on the Bézier curve. To specify the particle size for each control point, call **`UniformMatrix4fv`** on the reserved uniform `dmp_PartSys.aspect`, specifying a pointer to a 4x4 matrix with the size of the four control points in its first column. Store the sizes in the 4x4 matrix starting with the first control point in the first row and ending with the fourth control point in the fourth row.

To set the minimum and maximum particle size, call **`Uniform2fv`** on the reserved uniform `dmp_PartSys.pointSize` with ***`value`*** set to an array storing the minimum and maximum size, in that order. Because you also need to take the screen size into consideration, you must also call **`Uniform2fv`** on the reserved uniform `dmp_PartSys.viewport` with ***`value`*** set to the array (1 / viewport width, 1 / viewport height).

When particles are configured to use distance attenuation, the attenuated size *derived_size* is shown by the following equation. The particle size is given by *size* and the distance from the viewpoint is given by *d*.

Equation 4-6 Distance-Attenuated Particle Size

$$derived\_size = size \times \sqrt{\frac{1}{a + b \times d + c \times d^2}}$$

To set the attenuation coefficients *a*, *b*, and *c*, call **`Uniform3fv`** on the reserved uniform `dmp_PartSys.distanceAttenuation` with ***value*** set to an array storing *a*, *b*, and *c*, in that order.

4.5.5 Generated Particle Count

To set the maximum number of particles that can be generated, call **`Uniform1fv`** on the reserved uniform `dmp_PartSys.countMax` with ***value*** set to an array storing a number one less than the maximum number of particles that can be generated. You must specify a value of 0.0 or greater. If you set `dmp_PartSys.countMax` to a maximum value greater than 256, however, only 255 particles are generated because 255 particles is the maximum number supported by the shader implementation.

4.5.6 Particle Running Time

Time settings are required for particle systems. As the time increases from 0, generated particles move from within the bounding box around the first control point to within the bounding box around the fourth control point. To set the current time, call **`Uniform1fv`** on the reserved uniform `dmp_PartSys.time` with ***value*** set to an array storing the current time. The specified current time is converted by a random value for each particle. As that final value changes within the range from 0 to 1, the particle moves between the first and fourth control points.

If one of the shaders `DMP_particleSystem_0_X_X_X.obj` is used as the reserved geometry shader, particles outside the range 0 to 1 are not rendered. When one of the shaders `DMP_particleSystem_1_X_X_X.obj` is used, the current time loops within the range from 0 to 1. In other words, particles that reach the fourth control point are re-generated by the first control point.

To set the speed of particle movement, call **`Uniform1fv`** on the reserved uniform `dmp_PartSys.speed` with ***value*** set to an array storing a value for the speed.

4.5.7 Generating Random Values

Random values generated by a random number function determine the particle execution time and the position of the control points that define the Bézier curve, which itself determines particle positions. The details of the random number function are dependent on the geometry shader implementation and are not given in this specification. The random number function does, however, use an implementation similar to the following algorithm for a pseudo-random number generator.

Equation 4-7 Algorithm for a Pseudo-Random Number Generator

$$X_{N+1} = (aX_N + b) \bmod m$$

Values must be set for *a*, *b*, and *m* in Equation 4-7. To set them, call **`Uniform4fv`** on the reserved uniform `dmp_PartSys.randomCore` with ***`value`*** set to the array (*a*, *b*, *m*, *1/m*). To set the random seed corresponding to X_0 in Equation 4-7, call **`Uniform4fv`** on the reserved uniform `dmp_PartSys.randSeed` with ***`value`*** set to an array with 4 components. The values set in the array are applied as random values to the x, y, and z components of the Bézier curve that determines the particle positions, and to the particle time.

4.5.8 Texture Settings

4.5.8.1 Usable Texture Units

Geometry shaders for particle systems are implemented to output texture coordinate 0 and texture coordinate 2. Tto use texture coordinate 2, set the reserved geometry shader to `DMP_particleSystem_X_X_X_1.obj;` to not use it, set the reserved geometry shader to `DMP_particleSystem_X_X_X_0.obj`.

4.5.8.2 Texture Coordinates

Particle systems automatically generate texture coordinates. Texture coordinate 0's *uv* components indicate the lower-left, lower-right, upper-left, and upper-right of a particle using `(0,0)`, `(1,0)`, `(0,1)`, and `(1,1)`, respectively. Texture coordinate 2's *uv* components indicate the lower-left, lower-right, upper-left, and upper-right of a particle using `(-1,-1)`, `(1,-1)`, `(-1,1)`, and `(1,1)`, respectively.

You can also rotate and scale texture coordinates (although only texture coordinate 2 allows scaling). Texture coordinates are not rotated or scaled when the reserved geometry shader is set to `DMP_particleSystem_X_1_X_X.obj`. The texture coordinates are rotated and scaled when the reserved geometry shader is set to `DMP_particleSystem_X_0_X_X.obj`.

The texture coordinates for each particle are rotated and scaled by values calculated by interpolating the corresponding settings of each control point by the particle's position on the Bézier curve. To configure the scaling value and rotation angle (in radians) for the texture coordinates at the four control points, call **`UniformMatrix4fv`** on the reserved uniform `dmp_PartSys.aspect` and specify a pointer to a 4x4 matrix, with the rotation angles in column 2 and the scaling values in column 3. The 4x4 matrix stores the settings for control points 1-4 respectively in rows 1-4.

Given a rotation angle of *A* and a scaling value of *R*, texture coordinate 0's *uv* components are the following.

- Lower left: $(0.5 \times (1.0 + (-\cos A + \sin A)), 0.5 \times (1.0 + (-\cos A - \sin A)))$(0.5×(1.0
- Lower right: $(0.5 \times (1.0 + (\cos A + \sin A)), 0.5 \times (1.0 + (-\cos A + \sin A)))$(0.5×(1.0
- Upper left: $(0.5 \times (1.0 + (-\cos A - \sin A)), 0.5 \times (1.0 + (\cos A - \sin A)))$(0.5×(1.0
- Upper right: $(0.5 \times (1.0 + (\cos A - \sin A)), 0.5 \times (1.0 + (\cos A + \sin A)))$(0.5×(1.0

Texture coordinate 2's *uv* components are the following.

- Lower left: $(R(-\cos A + \sin A),\ R(-\cos A - \sin A))$(R(
- Lower right: $(R(\cos A + \sin A),\ R(-\cos A + \sin A))$(R(cosA
- Upper left: $(R(-\cos A - \sin A),\ R(\cos A - \sin A))$(R(
- Upper right: $(R(\cos A - \sin A),\ R(\cos A + \sin A))$(R(cosA

4.5.9 List of Reserved Uniforms

The following table shows settings for the reserved uniforms that are used by particle systems.

Table 4-5 Reserved Uniform Settings for Particle Systems

Uniform	Type	Value
`dmp_PartSys.color`	`mat4`	Each component is in the range [0.0, 1.0] Undefined by default
`dmp_PartSys.aspect`	`mat4`	Four of the following vectors: $(particle\ size, rotation\ angle, scale, alpha)$ where $1.0 \leq particle\ size$ and $0.0 \leq alpha \leq 1.0$ No range is specified for the other values Undefined by default
`dmp_PartSys.time`	`float`	Unspecified range Undefined by default
`dmp_PartSys.speed`	`float`	A value larger than 0.0 Undefined by default
`dmp_PartSys.countMax`	`float`	0.0 or greater Undefined by default
`dmp_PartSys.randSeed`	`vec4`	Unspecified range Undefined by default
`dmp_PartSys.randomCore`	`vec4`	Unspecified range Undefined by default
`dmp_PartSys.distanceAttenuation`	`vec3`	Each component is 0.0 or greater Undefined by default
`dmp_PartSys.viewport`	`vec2`	$\left(\frac{1}{viewport.width}, \frac{1}{viewport.height}\right)$ Undefined by default
`dmp_PartSys.pointSize`	`vec2`	Each component is 0.0 or greater Undefined by default

4.5.10 Reserved Geometry Shaders

The reserved geometry shaders for particle systems are split into different files by the features they offer. Table 4-6 shows their filenames and corresponding features.

Table 4-6 Particle System Filenames and Features

Filename	Time Clamping	Texture Coordinate Rotation	RGBA Color	Texture Coordinate 2
*_0_0_0_0.obj	Clamps	Uses	—	—
*_0_0_0_1.obj	Clamps	Uses	—	Uses
*_0_0_1_0.obj	Clamps	Uses	Uses	—
*_0_0_1_1.obj	Clamps	Uses	Uses	Uses
*_0_1_0_0.obj	Clamps	—	—	—
*_0_1_0_1.obj	Clamps	—	—	Uses
*_0_1_1_0.obj	Clamps	—	Uses	—
*_0_1_1_1.obj	Clamps	—	Uses	Uses
*_1_0_0_0.obj	—	Uses	—	—
*_1_0_0_1.obj	—	Uses	—	Uses
*_1_0_1_0.obj	—	Uses	Uses	—
*_1_0_1_1.obj	—	Uses	Uses	Uses
*_1_1_0_0.obj	—	—	—	—
*_1_1_0_1.obj	—	—	—	Uses
*_1_1_1_0.obj	—	—	Uses	—
*_1_1_1_1.obj	—	—	Uses	Uses

Asterisks (*) in the **Filename** column stand for `DMP_particleSystem.`

4.6 Vertex State Collections

Vertex state collections are used to bind vertex buffer objects and set other vertex attributes at the same time, as a single batch operation. A vertex state collection is a pseudo-vertex state object that shares namespaces with vertex buffer objects.

4.6.1 Creating Vertex State Collections

To create a vertex state collection object, use either **`GenBuffers`** or **`BindBuffer`** to bind an unused name to `VERTEX_STATE_COLLECTION_DMP`. Name 0 is reserved as the default vertex state collection object.

4.6.2 Binding Vertex State Collections

To bind a vertex state collection, call **`BindBuffer with`** ***`target`*** set to `VERTEX_STATE_COLLECTION_DMP` and ***`buffer`*** set to the name of a vertex state object. The vertex state collection object for name 0 is bound by default.

When a vertex state collection is bound, all settings for array buffers, element array buffers, and vertex attributes that are bound thereafter are associated with that vertex state collection. Associations with this vertex state collection will continue until another vertex state collection is bound. Binding a vertex state collection has the same effect as binding the vertex buffer associated with that vertex state collection and setting all of the vertex attributes. Specifically, vertex attribute settings are the settings configured by **`EnableVertexAttribArray`**, **`DisableVertexAttribArray`**, **`VertexAttrib{1234}{fv}`**, and **`VertexAttribPointer`**.

4.6.3 Deleting Vertex State Collections

To delete a vertex state collection, specify its name to ***`buffers`*** in **`DeleteBuffers`** just as you would a normal vertex buffer object. Deleting a vertex state collection does not affect the vertex buffer objects associated with it. A bound vertex state collection is not deleted at the moment **`DeleteBuffers`** is called on it. It is deleted as soon as another vertex state collection is bound. A vertex state collection object upon which **`DeleteBuffers`** is called remains in use until it is actually deleted. Any attempt to delete the default vertex state collection object is ignored.

5 Rasterization

5.1 Texture Units

The DMPGL 2.0 texture units have five features: 2D textures, cube maps, shadows, projection textures, and procedural textures. A 2D texture samples colors from a single texture image allocated to it, using texture coordinates represented by by (*s, t, r*). A cube-map texture uses a predetermined method to sample colors from a single texture that is itself selected via calculations based on the texture coordinates (*s, t, r*) from among the six 2D texture images allocated to it. A shadow texture samples special values used by the shadow features of reserved fragment shaders from the texture coordinates (*s, t, r*) in the texture image allocated to it. A procedural texture generates colors using a formal procedure from the texture coordinates (*s, t, r*). Table 5-1 shows the relationship between the texture numbers and their features.

There are four built-in texture units. They each have different features as shown in Table 5-1.

Table 5-1 Texture Unit Types

Texture Unit	2D Textures	Cube-Map Textures	Shadow Textures	Projection Textures	Procedural Textures
`TEXTURE0`	**X**	**X**	**X**	**X**	
`TEXTURE1`	**X**				
`TEXTURE2`	**X**				
`TEXTURE3`					**X**

The following reserved fragment shaders are among those that request special texture settings: shadows (see section 6.4 DMP Shadows), gas (see section 6.6 Gas), bump mapping (see section 6.3.6 Bump Mapping), and fragment lighting (see section 6.3 DMP Fragment Lighting). This section explains the details of textures without addressing the textures used by these special reserved fragment shaders.

Note: You can only use up to three independent texture coordinates for the four texture units. For details, see section 5.1.11 Input of Coordinates to Texture Units.

5.1.1 Enabling Texture Units

To enable or disable a texture unit in OpenGL ES, call **`Enable`** or **`Disable`** with ***`cap`*** set to the constant `TEXTURE_2D`. In DMPGL 2.0, however, there are different procedures: to disable a texture unit, call **`Uniform1i`** on the reserved uniform `dmp_Texture[i].samplerType` with `FALSE` specified. To enable a texture unit, call **`Uniform1i`** on the reserved uniform `dmp_Texture[i].samplerType` with `TEXTURE_2D`, `TEXTURE_PROCEDURAL_DMP`, `TEXTURE_CUBE_MAP_DMP`, `TEXTURE_SHADOW_2D_DMP`, or `TEXTURE_SHADOW_CUBE_DMP` specified.

For details on reserved uniforms and reserved fragment shaders, see Chapter 6 Reserved Fragment Shaders.

The following values can be specified to `dmp_Texture[i].samplerType` (where `i` is 0, 1, 2, or 3) for each texture unit.

Table 5-2 Texture Unit samplerType

Uniform	Value

`dmp_Texture[0].samplerType`	`FALSE` (default)
	`TEXTURE_2D`
	`TEXTURE_CUBE_MAP`
	`TEXTURE_SHADOW_2D_DMP`
	`TEXTURE_PROJECTION_DMP`
	`TEXTURE_SHADOW_CUBE_DMP`

`dmp_Texture[1].samplerType`	`FALSE` (default) `TEXTURE_2D`
`dmp_Texture[2].samplerType`	`FALSE` (default) `TEXTURE_2D`
`dmp_Texture[3].samplerType`	`FALSE` (default) `TEXTURE_PROCEDURAL_DMP`

5.1.2 Specifying Texture Units

Texture unit settings that do not use reserved uniforms are applied to the texture unit selected by `ACTIVE_TEXTURE`. Use the following command to set `ACTIVE_TEXTURE`.

Code 5-1 ActiveTexture

```
void ActiveTexture(enum texture);
```

In DMPGL 2.0, you can get the number of built-in texture units by calling **`GetIntegerv`** with `MAX_COMBINED_TEXTURE_IMAGE_UNITS` specified. Setting a value for `ACTIVE_TEXTURE` equal to or larger than `MAX_COMBINED_TEXTURE_IMAGE_UNITS` causes the error `INVALID_ENUM` to occur. Setting `ACTIVE_TEXTURE` to the texture unit `TEXTURE3`, which is used only for procedural textures, also causes the error `INVALID_ENUM` to occur. Table 5-1 shows the features and texture unit numbers for the `ACTIVE_TEXTURE` status.

5.1.3 Texture Image Specifications

Use **`TexImage2D`** to specify the texture images used by 2D textures and cube-map textures, as well as the texture images used by the gas textures and shadow textures used by some reserved fragment shader features.

Code 5-2 TexImage2D

```
void TexImage2D(enum target, int level,
                int internalformat, sizei width, sizei height,
```

```
        int border, enum format, enum type, void* data);
```

The argument ***target*** must be TEXTURE_2D for 2D images and one of the following for cube maps: TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z. Use ***format***, ***type***, and ***data*** to specify the image data format, data type, and a pointer to the image data, respectively. When ***data*** is set to 0, a texture region is allocated without an image specified.

Table 5-3 shows the allowed combinations for ***format*** and ***type***.

Table 5-3 format and type

format **argument**	*type* **argument**
RGBA	UNSIGNED_SHORT_4_4_4_4
RGBA	UNSIGNED_SHORT_5_5_5_1
RGBA	UNSIGNED_BYTE
RGB	UNSIGNED_SHORT_5_6_5
RGB	UNSIGNED_BYTE (only valid for TEXTURE_2D)
ALPHA	UNSIGNED_BYTE
ALPHA	UNSIGNED_4BITS_DMP
LUMINANCE	UNSIGNED_BYTE
LUMINANCE	UNSIGNED_4BITS_DMP
LUMINANCE_ALPHA	UNSIGNED_BYTE
LUMINANCE_ALPHA	UNSIGNED_BYTE_4_4_DMP
SHADOW_DMP	UNSIGNED_INT
GAS_DMP	UNSIGNED_SHORT
HILO8_DMP	UNSIGNED_BYTE
RGBA_NATIVE_DMP	UNSIGNED_SHORT_4_4_4_4
RGBA_NATIVE_DMP	UNSIGNED_SHORT_5_5_5_1
RGBA_NATIVE_DMP	UNSIGNED_BYTE
RGB_NATIVE_DMP	UNSIGNED_SHORT_5_6_5
RGB_NATIVE_DMP	UNSIGNED_BYTE (only valid for TEXTURE_2D)
ALPHA_NATIVE_DMP	UNSIGNED_BYTE
ALPHA_NATIVE_DMP	UNSIGNED_4BITS_DMP
LUMINANCE_NATIVE_DMP	UNSIGNED_BYTE
LUMINANCE_NATIVE_DMP	UNSIGNED_4BITS_DMP
LUMINANCE_ALPHA_NATIVE_DMP	UNSIGNED_BYTE
LUMINANCE_ALPHA_NATIVE_DMP	UNSIGNED_BYTE_4_4_DMP
SHADOW_NATIVE_DMP	UNSIGNED_INT
GAS_NATIVE_DMP	UNSIGNED_SHORT
HILO8_DMP_NATIVE_DMP	UNSIGNED_BYTE

You must specify data in the native PICA format to the ***data*** argument when ***format*** is set to one of the *_NATIVE_DMP values. For details, see section 5.4 Native PICA Format.

Use a negative value to specify the mipmap level to load in the ***level*** argument. Both 0 and -1 are treated as identical. All levels of a mipmap are loaded together; individual levels cannot be loaded separately. The image data specified by the ***data*** argument is stored contiguously level by level, starting from the lowest-level data. When automatically generating mipmaps, however, only the lowest level of data is stored in ***data***. Specifying a value higher than 0 for the ***level*** argument causes an INVALID_VALUE error.

Both ***width*** and ***height*** must be an integer power of 2 between 8 and 1024. Otherwise, an INVALID_VALUE error will occur.

When ***target*** is TEXTURE_CUBE_MAP_* and the ***width*** and ***height*** values are unequal, the error INVALID_OPERATION occurs for **TexImage2D**. Likewise, all values set for each of the TEXTURE_CUBE_MAP_* targets must be the same except for the ***data*** argument.

Use ***internalformat*** to specify the base internal format. Table 5-4 shows how the base internal format corresponds to the R, G, B, and A components included in an image. The NATIVE modifier has been omitted within this table to so that the distinction between formats with and without the modifier can be ignored.

Table 5-4 Conversion from RGBA Pixels into the Internal Texture Format

Base Internal Format	RGBA	Internal Components
ALPHA	A	A
LUMINANCE	R	L
LUMINANCE_ALPHA	R,A	L,A
RGB	R,G,B	R,G,B
RGBA	R,G,B,A	R,G,B,A
HILO8_DMP	R,G	Nx,Ny

If ***internalformat*** and ***format*** are not the same, INVALID_OPERATION occurs for **TexImage2D**.

The following explanations pertain to the HILO8_DMP format, the UNSIGNED_BYTE_4_4_DMP type, and the UNSIGNED_4BITS_DMP type, which are unique to DMPGL and are shown in some combinations in Table 5-3. Section 5.4 Native PICA Format explains the formats with the NATIVE modifier.

The HILO8_DMP format contains R and G components. The UNSIGNED_BYTE type can be specified for this format, and eight bits each are used for the R and G components. The B and A components for HILO8_DMP are output as 0 and 1, respectively.

The UNSIGNED_BYTE_4_4_DMP type is used when each pixel is stored in a single byte and has two 4-bit components. The upper four bits and the lower four bits are used for separate components of

the same pixel. DMPGL 2.0 allows you to use UNSIGNED_BYTE_4_4_DMP in combination with the LUMINANCE_ALPHA and LUMINANCE_ALPHA_NATIVE_DMP formats, in which case the upper four bits represent the luminance and the lower four bits represent the alpha value.

The UNSIGNED_4BITS_DMP type is used when two pixels are stored in a single byte and each pixel has one 4-bit component. DMPGL 2.0 allows you to use the UNSIGNED_4BITS_DMP type in combination with the LUMINANCE, ALPHA, LUMINANCE_NATIVE_DMP, and ALPHA_NATIVE_DMP formats.

When the UNSIGNED_4BITS_DMP type is used, the upper bits are stored in the higher addresses. Pixels are therefore packed starting at the lower four bits of the first byte, followed by the higher four bits of the first byte, followed by the lower four bits of the second byte, and so on.

Note: If you simultaneously enable texture formats that use UNSIGNED_4BITS_DMP (4-bit formats) and other texture formats (non-4-bit formats) and use them as multitextures, there are restrictions on the regions in which textures can be placed. Specifically, when data that uses a 4-bit format is placed in VRAM, any data that uses a non-4-bit format must be placed in a different memory region. VRAM-A and VRAM-B are handled as separate memory regions. Behavior becomes unstable if 4-bit and non-4-bit formats are stored in the same memory region. When data that uses a 4-bit format is placed in FCRAM, however, there are no restrictions on the placement of data that uses a non-4-bit format. ETC formats are handled as non-4-bit formats for the purposes of these restrictions.

5.1.4 Copying From the Framebuffer

Use **CopyTexImage2D** to copy data from the framebuffer into a texture.

Code 5-3 CopyTexImage2D

```
void CopyTexImage2D(enum target, int level,
                    enum internalformat, int x, int y, sizei width,
                    sizei height, int border);
```

The argument ***target*** must be TEXTURE_2D for 2D images and one of the following for cube maps: TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z. The texture selected as the target must use the RGB or RGBA format.

The specifications of the ***width***, ***height***, and ***border*** arguments for **CopyTexImage2D** are inherited from **TexImage2D**. However, the texture image is taken from the framebuffer instead of **TexImage2D**'s ***data*** argument. The error INVALID_VALUE occurs with **CopyTexImage2D** when the texture format is not compatible with the framebuffer format.

Note: The OpenGL ES 2.0 specifications allow copy operations that entail format conversions but DMPGL 2.0 does not. In DMPGL 2.0, you must set ***x*** and ***y*** to multiples of 8. If you specify a number that is not a multiple of 8, INVALID_VALUE occurs. You must also set ***level*** to 0. If you specify a number other than 0, INVALID_OPERATION occurs.

However, there is one exception: when automatically generating mipmaps, you can use a negative value to specify the mipmap level in ***level*** without causing INVALID_OPERATION. For details, see section 5.1.13 Automatically Generating Texture Mipmap Data.

You cannot use **CopyTexImage2D** with block-32 mode. Attempts to do so will not cause an error, but images will fail to be transferred correctly. For more details on block-32 mode, see section 5.5.3 Block Mode.

5.1.5 Partial Texture Images

Use **CopyTexSubImage2D** to limit **CopyTexImage2D** to processing a partial image. You must use **TexImage2D** to create the texture image to copy before calling **CopyTexSubImage2D**.

Code 5-4 CopyTexSubImage2D

```
void CopyTexSubImage2D(enum target, int level,
                       int xoffset, int yoffset, int x, int y, sizei width,
                       sizei height);
```

The argument ***target*** must be TEXTURE_2D for 2D images and one of the following for cube maps: TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z.

Note: In DMPGL 2.0, you must set ***xoffset***, ***yoffset***, ***x***, ***y***, ***width***, and ***height*** to multiples of 8. If you specify a number that is not a multiple of 8, INVALID_VALUE occurs. You must set ***level*** to 0. If you specify a number other than 0, INVALID_OPERATION occurs.

When automatically generating mipmaps, an additional restriction applies: you must set ***level*** to the same value set for ***level*** in the previous call to **TexImage2D** or **CopyTexImage2D**, or INVALID_OPERATION occurs. For details, see section 5.1.13 Automatically Generating Texture Mipmap Data.

You cannot use **CopyTexSubImage2D** with block-32 mode. Attempts to do so will not cause an error, but images will fail to be transferred correctly. For more details on block-32 mode, see section 5.5.3 Block Mode for Early Depth Tests. Likewise, **TexSubImage2D** is not supported.

5.1.6 Compressed Textures

You can use a compressed image as a texture.

Code 5-5 CompressedTexImage2D

```
void CompressedTexImage2D(enum target, int level,
                          int internalformat, sizei width, sizei height,
                          int border, sizei imageSize, void* data);
```

The argument ***target*** must be TEXTURE_2D for 2D images and one of the following for cube maps: TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X,

TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z.

You can set ***internalformat*** equal to ETC1_RGB8_NATIVE_DMP or ETC1_ALPHA_RGB8_A4_NATIVE_DMP.

When ETC1_RGB8_NATIVE_DMP is specified, a 4x4 pixel, 24-bit RGB image that is compressed into 64 bits is set in ***data***. Both ***width*** and ***height*** must be an integer power of 2 between 16 and 1024. If this range is exceeded, INVALID_VALUE occurs. Use a negative value to specify the mipmap level to load in the ***level*** argument.

When ETC1_ALPHA_RGB8_A4_NATIVE_DMP is specified, a 4x4 pixel, 32-bit RGBA image that is compressed into 128 bits is set in ***data***. Both ***width*** and ***height*** must be integer powers of 2 between 8 and 1024. An INVALID_VALUE error occurs if this range is exceeded. The first 64 bits of compressed data are alpha data and the last 64 bits are RGB data. The 64 bits of alpha data represent 4x4 texels using four bits of alpha data per texel. The relationship between texels and alpha data is explained in section 5.4.1.2 Byte Order for Compressed Textures.

If you set ***internalformat*** equal to some other value than those described here, INVALID_OPERATION occurs.

You must set ***imageSize*** equal to the size (in bytes) of the image data. The ETC1 texture size is given by the number of bytes yielded by Equation 5-1. The width and height of the original texture image are given by *w* and *h*, respectively.

The value of $blockSize$ is 8 when ***internalformat*** is ETC1_RGB8_NATIVE_DMP and 16 when ***internalformat*** is ETC1_ALPHA_RGB8_A4_NATIVE_DMP.

Equation 5-1 imageSize

$$ETCTextureSize = \frac{w}{4} \times \frac{h}{4} \times blockSize$$

The equation above produces the size of a single mipmap level. Take the total sum of these values for all mipmap levels, and specify that sum in the ***imageSize*** argument.

CompressedTexSubImage2D, which processes partial images, is not supported.

5.1.7 Lookup Tables

Procedural textures, fragment shading, fog, and gas all use reserved fragment shaders that are implemented using one-dimensional tables called *lookup tables*. The number of lookup tables can be obtained by specifying MAX_LUT_TEXTURES_DMP and calling the **GetIntegerv** function.

5.1.7.1 Lookup Table Numbers

A lookup table is indicated by the expression LUT_TEXTUREi_DMP. The i in this expression is an integer from 0 to the number of lookup tables minus 1 and is called the *lookup table number*. LUT_TEXTUREi_DMP is defined to be LUT_TEXTURE0_DMP +i.

5.1.7.2 Lookup Table Objects

The content of a lookup table is set by a special texture object called a lookup table object. A lookup table object is created using **GenTextures** just like a normal texture. Lookup table object 0 is reserved by DMPGL. Binding 0 to a lookup table object will detach the lookup table object. The detached lookup table will no longer be usable. If a lookup table is detached by a reserved fragment shader, an INVALID_OPERATION error will occur if you call **DrawElements** or **DrawArrays** with that table specified.

5.1.7.3 Setting the Content of a Lookup Table Object

Use **TexImage1D** to set the content of a lookup table object.

Code 5-6 TexImage1D

```
void TexImage1D(enum target, int level, int internalformat,
          sizei width,    int border, enum format, enum type, void* data);
```

target can only be set to LUT_TEXTUREi_DMP.

width cannot exceed the maximum number of entries in the lookup table. If the maximum number of entries is exceeded, **TexImage1D** causes an INVALID_VALUE error. The maximum number of entries in a lookup table can be obtained by specifying MAX_LUT_ENTRIES_DMP and calling the **GetIntegerv** function.

level is only allowed to be 0. If a nonzero value is specified, **TexImage1D** causes an INVALID_VALUE error.

type is only allowed to be FLOAT. If a type other than FLOAT is specified, **TexImage1D** causes an INVALID_ENUM error.

format and ***internalformat*** are only allowed to be LUMINANCEF_DMP. If a format other than LUMINANCEF_DMP is specified, **TexImage1D** causes an INVALID_ENUM error.

You can set the ***width*** argument to any value that does not exceed the maximum number of entries in the lookup table. When you load ***data*** with a size specified by ***width***, the data is loaded to the lookup table object bound to the LUT_TEXTUREi_DMP value specified to ***target***. Each reserved fragment shader that accesses a lookup table object has independent restrictions on the value of ***width*** and the content of ***data***. See the definitions in the various reserved fragment shaders.

5.1.7.4 Partially Setting the Content of a Lookup Table Object

Use **TexSubImage1D** to partially set a lookup table object. This function overwrites a subarray within the array set by **TexImage1D**. You must use **TexImage1D** to create the lookup table before calling **TexSubImage1D**.

Code 5-7 TexSubImage1D

```
void TexSubImage1D(enum target, int level, int xoffset,
                   sizei width, enum format, enum type, const void* data);
```

You can only set ***target*** to LUT_TEXTUREi_DMP. The lookup table object bound to the LUT_TEXTUREi_DMP specified as ***target*** is the lookup table whose content is partially replaced by this function.

xoffset indicates the number of the first element to overwrite within the already-configured array. ***width*** specifies the number of elements to overwrite. You can set the ***width*** and ***xoffset*** arguments to any value as long as their total does not exceed the maximum number of entries in the lookup table. If the maximum value is exceeded, **TexSubImage1D** causes an INVALID_VALUE error. The maximum number of entries in a lookup table can be obtained by specifying MAX_LUT_ENTRIES_DMP and calling the **GetIntegerv** function.

level is only allowed to be 0. If a nonzero value is specified, **TexSubImage1D** causes an INVALID_VALUE error.

type is only allowed to be FLOAT. If a type other than FLOAT is specified, **TexSubImage1D** causes an INVALID_ENUM error.

format is only allowed to be LUMINANCEF_DMP. If a format other than LUMINANCEF_DMP is specified, **TexSubImage1D** causes an INVALID_ENUM error.

Note: The commands **TexImage1D** and **TexSubImage1D** do not exist in the OpenGL ES 1.1 and 2.0 specifications. Their definitions are analogous to the OpenGL specifications, but they do not allow uses defined by OpenGL. In particular, these commands cannot be used to assign colors to object surfaces. DMPGL 2.0 uses 1D textures so that they can be referenced by procedural textures (see section 6.2 Procedural Textures), fragment lighting (see section 6.3 DMP Fragment Lighting), gas (see section 6.6 Gas), and fog (see section 6.5 Fog).

DMPGL 2.0 does not contain **CopyTexImage1D**, **CopyTexSubImage1D**, or other possible commands corresponding to **TexImage1D**. There are also no settings that use **TexParameter**.

5.1.7.5 Using Lookup Table Objects

Reserved fragment shaders use lookup tables. To specify a lookup table to use, set its lookup table number in a dedicated reserved uniform. The content of a configured lookup table is determined by the lookup table object bound to it. For details on the reserved uniforms used to set a lookup table for a reserved fragment shader, see the description of each reserved fragment shader.

5.1.7.6 Getting Bound Lookup Table Objects

To get the ID of the lookup table object that is currently bound to LUT_TEXTUREi_DMP, set the ***pname*** argument of **GetIntegerv** to TEXTURE_BINDING_LUTi_DMP.

5.1.8 Creating Textures

Use **GenTextures** to create texture objects and texture collection objects. (Texture collection objects are a new feature added to DMPGL 2.0. They do not exist in the OpenGL ES 2.0 specifications. For details, see section 5.3 Texture Collections.)

Code 5-8 GenTextures

```
void GenTextures(sizei n, uint* textures);
```

This generates an object for a 2D texture, cube map, lookup table, or texture collection.

5.1.9 Binding Textures

Use **BindTexture** to bind texture objects and texture collection objects. (Texture collection objects are a new feature added to DMPGL 2.0. They do not exist in the OpenGL ES 2.0 specifications. For details, see section 5.3 Texture Collections.)

Code 5-9 BindTexture

```
void BindTexture(enum target, int texture);
```

The ***target*** argument allows the following values: TEXTURE_2D, TEXTURE_CUBE_MAP, TEXTURE_COLLECTION_DMP, or LUT_TEXTUREi_DMP (where i is between 0 and the number of lookup tables minus one). If any other value is specified, **BindTexture** causes an INVALID_ENUM error.

5.1.10 Texture Parameters

Use **TexParameter{if}{v}** to specify parameters to texture objects. To give a parameter to a texture object, call **TexParameter** with ***target*** set to the target object, ***pname*** set to the parameter name, and ***param*** set to the parameter value.

Code 5-10 TexParameter

```
void TexParameter{if}(enum target, enum pname, T param);
void TexParameter{if}v(enum target, enum pname, T params);
```

The ***target*** argument is the same as the one specified to commands such as **TexImage2D** and **TexImage1D**. The ***pname*** argument can only use the names given in Table 5-5. As new features added to DMPGL 2.0 that do not exist in OpenGL ES 2.0, in this command you can specify border clamping, the Level of Detail (LOD) bias, the minimum LOD, and automatic mipmap generation. To activate border clamping, call **TexParameteri** with ***pname*** set to TEXTURE_WRAP_S and ***param*** set to CLAMP_TO_BORDER. Regions clamped from a texture use the specified border color.

To specify the border color, call **TexParameterfv** with ***pname*** set to TEXTURE_BORDER_COLOR and ***params*** set to the color.

To automatically generate mipmaps, call **TexParameteri** with ***pname*** set to GENERATE_MIPMAP and ***param*** set to TRUE. For details, see section 5.1.13 Automatically Generating Texture Mipmap Data.

Table 5-5 Corresponding Color and Texture Formats

Name	Type	Value
TEXTURE_WRAP_S	int	REPEAT (default) MIRRORED_REPEAT CLAMP_TO_EDGE CLAMP_TO_BORDER
TEXTURE_WRAP_T	int	The same as TEXTURE_WRAP_S
TEXTURE_MIN_FILTER	int	NEAREST (default) LINEAR NEAREST_MIPMAP_NEAREST NEAREST_MIPMAP_LINEAR LINEAR_MIPMAP_NEAREST LINEAR_MIPMAP_LINEAR
TEXTURE_MAG_FILTER	int	NEAREST (default) LINEAR
TEXTURE_BORDER_COLOR	vec4	Each component is in the range [0.0, 1.0] (0, 0, 0, 0) by default
TEXTURE_LOD_BIAS	float	In the range [-16.0,16.0] 0.0 by default
TEXTURE_MIN_LOD	int	Unspecified range -1000 by default
GENERATE_MIPMAP	bool	TRUE FALSE (default)

5.1.11 Input of Coordinates to Texture Units

A vertex shader (or geometry shader when one is in use) can output up to three texture coordinates. Texture coordinate 0 and texture coordinate 1 are supplied to texture unit 0 and texture unit 1, respectively.

Texture unit 2 can select either texture coordinate 1 or 2 as input. Set the reserved uniform dmp_Texture[2].texcoord to either TEXTURE1 or TEXTURE2 to input texture coordinate 1 or 2, respectively, into texture unit 2.

Texture unit 3 can select texture coordinate 0, 1, or 2 as input. Set the reserved uniform dmp_Texture[3].texcoord to TEXTURE0, TEXTURE1, or TEXTURE2 to input texture coordinate 0, 1, or 2, respectively, into texture unit 3.

The attribute name specified as ***data_name*** and the output registers specified in ***mapped_register*** to **#pragma output_map** assign a vertex shader's output registers and the texture coordinates sent from the vertex shader.

Code 5-11 #pragma output_map

```
#pragma output_map(data_name, mapped_register);
```

The texture coordinates output by the vertex shader are specified by ***data_name***. Table 5-6 shows the relationship between ***data_name*** and the texture coordinates sent from the vertex shader. For details on how texture coordinates are assigned to output registers by the vertex shader, see the *Vertex Shader Reference Manual*.

Table 5-6 Relationship Between data_name and Attributes Sent from the Vertex Shader

data_name	**Attribute Sent from the Vertex Shader**
`texture0`	Texture unit 0's *u* and *v* coordinates
`texture0w`	Texture unit 0's *w* coordinate
`texture1`	Texture unit 1's *u* and *v* coordinates
`texture2`	Texture unit 2's *u* and *v* coordinates

If you set the reserved uniform `dmp_Texture[0].samplerType`, which specifies the type of texture unit 0, to `TEXTURE_PROJECTION_DMP`, `TEXTURE_SHADOW_DMP`, `TEXTURE_CUBE_MAP`, or `TEXTURE_CUBE_SHADOW_DMP`, you must use **`#pragma output_map`** to assign an output register to both `texture0` and `texture0w` in the vertex shader. The texture unit's output is undefined if an output register has not been assigned to at least one of these. If **`#pragma output_map`** is used to assign an output register to `texture0w` in the vertex shader but the reserved uniform `dmp_Texture[0].samplerType` is set to `TEXTURE_2D`, the coordinate output to `texture0w` is simply ignored.

Assume that the coordinates (*u*, *v*, *w*) are supplied to a texture unit by the vertex shader. Only texture unit 0 can use texture coordinate *w*. Texture units 1, 2, and 3 cannot use *w*. When *w* is enabled, the coordinates (*u*, *v*) are divided by *w*. Specifying a value of `TEXTURE_PROJECTION_DMP` to the reserved uniform `dmp_Texture[0].samplerType` is the same as specifying `TEXTURE_2D` with *w* enabled.

5.1.12 Loading Texture Mipmap Data

To load more than one mipmap level of data for a 2D texture or cube-map texture, set the ***level*** argument of **`TexImage2D`** or **`CompressedTexImage2D`** to a negative value indicating the number of mipmap levels and the ***data*** argument to data that consecutively stores all the mipmap levels you want to load in order of data size, from the largest-sized level to the smallest. You cannot specify each level separately as you can with the standard OpenGL specifications.

5.1.13 Automatically Generating Texture Mipmap Data

It is possible to automatically generate mipmap data for some formats and sizes of textures. To automatically generate mipmap data, first set the texture parameter `GENERATE_MIPMAP` to `TRUE`, then specify -2 or smaller for ***level*** in **`TexImage2D`**, **`CopyTexImage2D`**, or **`CopyTexSubImage2D`**.

With **`TexImage2D`**, the automatically generated data extends from the lowest-level data, specified in ***data***, through to data for the highest level. If a second or higher level of data was already specified in

data, that previous data is invalidated and the new automatically generated mipmap data is used instead. If 0 was specified in ***data***, no mipmaps are generated.

With **CopyTexImage2D** and **CopyTexSubImage2D**, the image in the current color buffer is transferred into the texture's lowest-level data and that transferred data is used as the base for generating all the higher-level mipmap data.

Mipmap data is automatically generated for all the levels in ***level*** except the lowest level.

Only some texture formats support automatic mipmap generation. Each of these texture formats, corresponding to different upper limits on LOD, has a different minimum height and width of textures that can be generated in that format. Table 5-7 shows the format and type arguments with which this feature is supported, and the corresponding minimum texture dimensions.

Table 5-7 format and type Combinations Supporting Automatic Mipmap Generation

format Argument	*type* Argument	Minimum Automatically Generatable Size (Texels)
RGBA	UNSIGNED_SHORT_4_4_4_4	64
RGBA	UNSIGNED_SHORT_5_5_5_1	64
RGBA	UNSIGNED_BYTE	32
RGB	UNSIGNED_SHORT_5_6_5	64
RGB	UNSIGNED_BYTE	32
RGBA_NATIVE_DMP	UNSIGNED_SHORT_4_4_4_4	64
RGBA_NATIVE_DMP	UNSIGNED_SHORT_5_5_5_1	64
RGBA_NATIVE_DMP	UNSIGNED_BYTE	32
RGB_NATIVE_DMP	UNSIGNED_SHORT_5_6_5	64
RGB_NATIVE_DMP	UNSIGNED_BYTE	32

For example, if the size of the lowest-level data is 128x128, ***format*** is RGBA, and ***type*** is UNSIGNED_BYTE, two levels of data can be automatically generated: 64x64 and 32x32.

If you specify a value for ***level*** that conflicts with the limits imposed by the minimum generatable mipmap sizes shown in Table 5-7, the INVALID_OPERATION error occurs.

In calls to **CopyTexSubImage2D** when GENERATE_MIPMAP is TRUE, you must set ***level*** to the same value set for ***level*** in the previous call to **TexImage2D** or **CopyTexImage2D.** Setting a different value for ***level*** causes an error. In contrast, when GENERATE_MIPMAP is FALSE, setting any nonzero value for ***level*** causes an error.

5.1.14 Texture Coordinate Precision

The term *actual hardware environment* refers to all hardware environments (development or retail hardware) and excludes the POD environment. On the actual hardware environment, texture

coordinates within the texture unit are represented using 16-bit values that contain both the integral part and the decimal part. With this representation, fewer bits will be available for the fractional part as the absolute value of the integral part increases. The texture sampling precision relies on the bit precision of the fractional part.

Optimal texture sampling is possible only when the fractional part has sufficient bits available to represent the number of texels in the texture's width or height. When using bilinear subsampling, Nintendo recommends that an additional six bits be made available for the decimal part.

5.2 Texture Combiners

5.2.1 Overview

The OpenGL ES 1.1 specifications mention texture combiners, which use specified combiner functions to combine multiple texture outputs. Combiner functions were eliminated from the OpenGL ES specifications starting with OpenGL ES 2.0, but this feature is still provided by the reserved fragment shaders in DMPGL 2.0. These DMPGL 2.0 specifications use the term *texture combiner*, or simply *combiner*, to refer to the feature for combining textures, fragment light colors, and other attributes in the later stages of the texture units and fragment light units.

Six combiners are built into the pipeline. These combiners take colors and alpha values from three textures, three fragment lights, or three other entities; apply a combiner function to each input color and alpha value; and output the result. The details of this feature are almost entirely the same as the OpenGL ES 1.1 specifications, and all the differences are explained here.

Combiner buffers are implemented as a DMPGL 2.0-specific feature. Through a combiner buffer, a combiner can receive source data from the output of a combiner before the one that immediately precedes it in the pipeline. The combiner buffer is explained in section 5.2.2 Combiner Buffer.

When using combiners, set reserved uniforms with **`Uniform1i`**, **`Uniform2i`**, **`Uniform3i`**, and **`Uniform4f`**. Reserved uniforms carry out the same settings that used **`TexEnv{i}`** in the OpenGL ES 1.1 specifications. The reserved uniforms that correspond to the **`TexEnv{i}`** arguments are given by Table 5-8.

Table 5-8 Relationship Between TexEnv{i} and Reserved Uniforms

OpenGL ES 1.1 Specifications `glTexEnv{i}`	**DMPGL 2.0 Reserved Uniforms**
`COMBINE_RGB`	`dmp_TexEnv[i].combineRgb`
`COMBINE_ALPHA`	`dmp_TexEnv[i].combineAlpha`
`SRC{0,1,2}_RGB`	`dmp_TexEnv[i].srcRgb`
`SRC{0,1,2}_ALPHA`	`dmp_TexEnv[i].srcAlpha`
`OPERAND{0,1,2}_RGB`	`dmp_TexEnv[i].operandRgb`
`OPERAND{0,1,2}_ALPHA`	`dmp_TexEnv[i].operandAlpha`
`RGB_SCALE`	`dmp_TexEnv[i].scaleRgb`

OpenGL ES 1.1 Specifications glTexEnv{i}	DMPGL 2.0 Reserved Uniforms
ALPHA_SCALE	dmp_TexEnv[i].scaleAlpha
TEXTURE_ENV_COLOR	dmp_TexEnv[i].constRgba

In Table 5-8 and throughout this section, `i` is an integer that indicates combiner 0, 1, 2, 3, 4, or 5.

The combiner source uses one of the reserved uniforms `dmp_TexEnv[i].srcRgb` (where `i` is 0, 1, or 2) or `dmp_TexEnv[i].srcAlpha` (where `i` is between 0 and 5). You can set these variables to the following values.

- `TEXTURE0`
- `TEXTURE1`
- `TEXTURE2`
- `TEXTURE3`
- `CONSTANT`
- `PRIMARY_COLOR`
- `PREVIOUS`
- `FRAGMENT_PRIMARY_COLOR_DMP`
- `FRAGMENT_SECONDARY_COLOR_DMP`
- `PREVIOUS_BUFFER_DMP`

You can set three inputs simultaneously with **`Uniform3i`**.

If you specify `PREVIOUS` here for `dmp_TexEnv[0].srcRgb` or `dmp_TexEnv[0].srcAlpha`, an `INVALID_ENUM` error occurs. If you do not select `CONSTANT`, `PREVIOUS`, or `PREVIOUS_BUFFER_DMP` at least once for `dmp_TexEnv[i].srcRgb` (where `i` is between 1 and 5) and/or `dmp_TexEnv[i].srcAlpha` (where `i` is between 1 and 5), an `INVALID_ENUM` error occurs. Section 5.2.2 Combiner Buffer explains behavior when `PREVIOUS_BUFFER_DMP` is set.

The color operand uses the reserved uniform `dmp_TexEnv[i].operandRgb` (where `i` is between 0 and 5). You can set this variable to any of the following values.

- `SRC_COLOR`
- `ONE_MINUS_SRC_COLOR`
- `SRC_ALPHA`
- `ONE_MINUS_SRC_ALPHA`
- `SRC_R_DMP`
- `ONE_MINUS_SRC_R_DMP`
- `SRC_G_DMP`
- `ONE_MINUS_SRC_G_DMP`
- `SRC_B_DMP`
- `ONE_MINUS_SRC_B_DMP`

The alpha operand uses the reserved uniform `dmp_TexEnv[i].operandAlpha` (where `i` is between 0 and 5). You can set this variable to any of the following values.

- SRC_ALPHA
- ONE_MINUS_SRC_ALPHA
- SRC_R_DMP
- ONE_MINUS_SRC_R_DMP
- SRC_G_DMP
- ONE_MINUS_SRC_G_DMP
- SRC_B_DMP
- ONE_MINUS_SRC_B_DMP

You can set three inputs simultaneously with **Uniform3i**. Section 5.2.3 Other Combiner Features explains behavior when the reserved uniform dmp_TexEnv[i].operandRgb or dmp_TexEnv[i].operandAlpha is set equal to SRC_R_DMP, ONE_MINUS_SRC_R_DMP, SRC_G_DMP, ONE_MINUS_SRC_G_DMP, SRC_B_DMP, or ONE_MINUS_SRC_B_DMP.

Use the reserved uniform dmp_TexEnv[i].combineRgb to set the color combiner function. Use the reserved uniform dmp_TexEnv[i].combineAlpha to set the alpha combiner function. You can set each of these variables to the following values.

- REPLACE
- MODULATE
- ADD
- ADD_SIGNED
- INTERPOLATE
- SUBTRACT
- DOT3_RGBA
- ADD_MULT_DMP
- MULT_ADD_DMP

DOT3_RGB can also be set, but only for dmp_TexEnv[i].combineRgb.

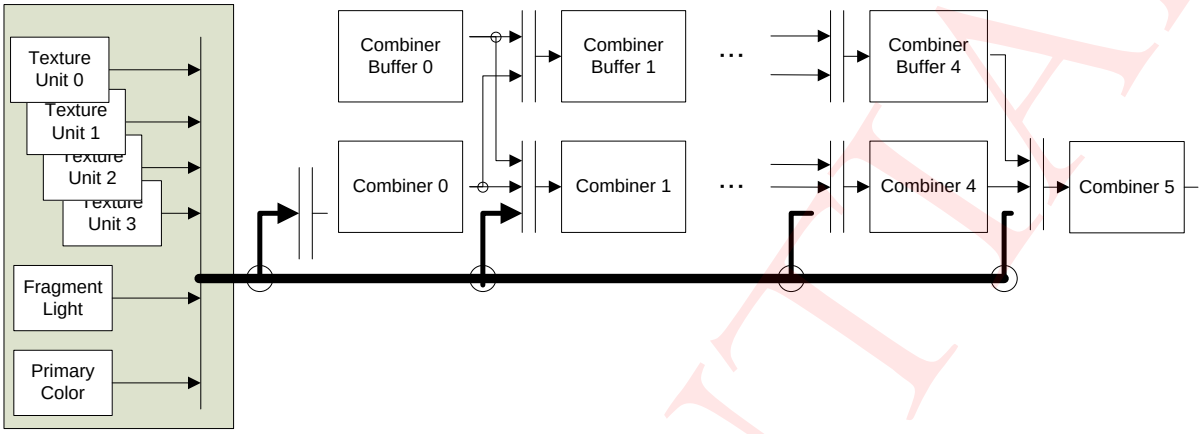
If you set either dmp_TexEnv[i].combineRgb or dmp_TexEnv[i].combineAlpha to DOT3_RGBA, you must set the other to DOT3_RGBA as well. Otherwise, combiner behavior is undefined.

Combiners also accept fragment light colors as sources. To set the primary or secondary color of a fragment light as a combiner source, assign FRAGMENT_PRIMARY_COLOR_DMP or FRAGMENT_SECONDARY_COLOR_DMP to the reserved uniform dmp_TexEnv[i].srcRgb or dmp_TexEnv[i].srcAlpha.

5.2.2 Combiner Buffers

Through a combiner buffer, a combiner can receive source data from the output of a combiner before the one that immediately precedes it in the pipeline. There are five built-in combiner buffers, in parallel with the first five combiners. You can choose the source of any combiner buffer except for the first one to be the output of either the previous combiner or the previous combiner buffer. A combiner buffer can store a single color and a single alpha value. The color and alpha for a combiner buffer can each come from different sources.

Figure 5-1 Relationship Between Combiners and Combiner Buffers



To set the source of a combiner buffer, use **`Uniform2i`** on the reserved uniform `dmp_TexEnv[i].bufferInput` (where `i` is 1, 2, 3, or 4) and configure the color and alpha simultaneously. A combiner buffer references output from the previous-stage combiner when the reserved uniform `dmp_TexEnv[i].bufferInput` is set equal to `PREVIOUS`, and references the previous-stage combiner buffer when the reserved uniform is set equal to `PREVIOUS_BUFFER_DMP`.

A combiner references the previous-stage combiner buffer if the reserved uniforms `dmp_TexEnv[i].srcRgb` or `dmp_TexEnv[i].srcAlpha`, which indicate the combiner source, have been set equal to `PREVIOUS_BUFFER_DMP`.

The output from the first combiner buffer can be configured as the combiner buffer color. The setting for the combiner buffer color is used by the reserved uniform `dmp_TexEnv[0].bufferColor`, which is set by **`Uniform4f`**.

5.2.3 Other Combiner Features

DMPGL 2.0 has implemented `ADD_MULT_DMP` and `MULT_ADD_DMP` in addition to the combiner functions defined by the OpenGL ES 1.1 specifications. `ADD_MULT_DMP` adds source 0 and source 1, clamps the sum between 0 and 1, multiplies the result by source 2. `MULT_ADD_DMP` multiplies source 0 and source 1, and then adds source 2 to the result. Both functions take three inputs. These functions can be applied to both colors and alpha values.

The new definitions `FRAGMENT_PRIMARY_COLOR_DMP` and `FRAGMENT_SECONDARY_COLOR_DMP` are used as sources when fragment lighting is in use. These constants can be accepted as both color and alpha sources.

If the reserved uniform `dmp_TexEnv[i].operandRgb` or `dmp_TexEnv[i].operandAlpha` is set equal to any of the following values, a single component is selected from the previous-stage combiner color and that component is used as a source.

- `SRC_R_DMP`
- `ONE_MINUS_SRC_R_DMP`
- `SRC_G_DMP`
- `ONE_MINUS_SRC_G_DMP`

- SRC_B_DMP
- ONE_MINUS_SRC_B_DMP

If you specify SRC_R_DMP, the source's R component is selected as source data. If you specify ONE_MINUS_SRC_R_DMP, the result of subtracting the source's R component from 1.0 is selected. The values SRC_G_DMP and ONE_MINUS_SRC_G_DMP behave the same way for the G component, and SRC_B_DMP and ONE_MINUS_SRC_B_DMP behave the same way for the B component.

5.2.4 List of Reserved Uniforms

Table 5-9 shows settings for the reserved uniforms that are used by combiners.

Table 5-9 Reserved Uniform Settings for Combiners

Uniform	Type	Value
dmp_TexEnv[i].combineRgb	int	• REPLACE **(default)** • MODULATE • ADD • ADD_SIGNED • INTERPOLATE • SUBTRACT • DOT3_RGB • DOT3_RGBA • ADD_MULT_DMP • MULT_ADD_DMP
dmp_TexEnv[i].combineAlpha	int	• REPLACE **(default)** • MODULATE • ADD • ADD_SIGNED • INTERPOLATE • SUBTRACT • DOT3_RGBA • ADD_MULT_DMP • MULT_ADD_DMP
dmp_TexEnv[0].srcRgb	ivec3	Each component: • TEXTURE0 • TEXTURE1 • TEXTURE2 • TEXTURE3 • CONSTANT **(default)** • PRIMARY_COLOR • FRAGMENT_PRIMARY_COLOR_DMP • FRAGMENT_SECONDARY_COLOR_DMP

Uniform	Type	Value
dmp_TexEnv[i].srcRgb (i is nonzero)	ivec3	Each component: • TEXTURE0 • TEXTURE1 • TEXTURE2 • TEXTURE3 • CONSTANT • PRIMARY_COLOR • PREVIOUS **(default)** • PREVIOUS_BUFFER_DMP • FRAGMENT_PRIMARY_COLOR_DMP • FRAGMENT_SECONDARY_COLOR_DMP At least one of either CONSTANT, PREVIOUS, or PREVIOUS_BUFFER_DMP must be used.
dmp_TexEnv[0].srcAlpha	ivec3	Identical to dmp_TexEnv[0].srcRGB
dmp_TexEnv[i].srcAlpha (i is nonzero)	ivec3	Identical to dmp_TexEnv[i].srcRGB (i is nonzero)
dmp_TexEnv[i].operandRgb	ivec3	Each component: • SRC_COLOR **(default)** • ONE_MINUS_SRC_COLOR • SRC_ALPHA • ONE_MINUS_SRC_ALPHA • SRC_R_DMP • ONE_MINUS_SRC_R_DMP • SRC_G_DMP • ONE_MINUS_SRC_G_DMP • SRC_B_DMP • ONE_MINUS_SRC_B_DMP
dmp_TexEnv[i].operandAlpha	ivec3	• SRC_ALPHA **(default)** • SRC_ONE_MINUS_SRC_ALPHA • SRC_R_DMP • ONE_MINUS_SRC_R_DMP • SRC_G_DMP • ONE_MINUS_SRC_G_DMP • SRC_B_DMP • ONE_MINUS_SRC_B_DMP
dmp_TexEnv[i].bufferInput (i is 1, 2, 3, or 4)	ivec2	• PREVIOUS • PREVIOUS_BUFFER_DMP **(default)**
dmp_TexEnv[i].scaleRgb	float	• 1.0 **(default)** • 2.0 • 4.0

Uniform	Type	Value
`dmp_TexEnv[i].scaleAlpha`	`float`	• `1.0` **(default)** • `2.0` • `4.0`
`dmp_TexEnv[i].constRgba`	`vec4`	Each component is in the range `[0.0, 1.0]` `(0, 0, 0, 0)` by default
`dmp_TexEnv[0].bufferColor`	`vec4`	Each component is in the range `[0.0, 1.0]` `(0, 0, 0, 0)` by default

5.3 Texture Collections

A texture collection is used to bind several texture objects all at once as a batch operation. A texture collection is a pseudo-texture object that shares a namespace with other texture objects.

5.3.1 Creating Texture Collections

To create a texture collection object, use either **`GenTextures`** or **`BindTexture`** to bind an unused name to `TEXTURE_COLLECTION_DMP`. Name 0 is reserved as the default texture collection object.

5.3.2 Binding Texture Collections

To bind a texture collection, set ***`target`*** to `TEXTURE_COLLECTION_DMP` and ***`texture`*** to the name of a texture object with **`BindTexture`**. The texture collection object for name 0 is bound by default.

When a texture collection is bound, all textures (2D textures, cube-map textures, and lookup tables) bound thereafter are associated with that texture collection. Associations with this texture collection continue until another texture collection is bound. Binding a texture collection has the same effect as binding all textures associated with that texture collection.

5.3.3 Deleting Texture Collections

To delete a texture collection, specify its name to ***`textures`*** with **`DeleteTextures`** just as you would a normal texture object. Deleting a texture collection does not affect the texture objects associated with it. A bound texture collection is not deleted at the moment **`DeleteTextures`** is called on it. It is deleted as soon as another texture collection is bound. That texture collection object is in use from the time **`DeleteTextures`** is used until it is actually deleted. Any attempt to delete the default texture collection object is ignored.

5.4 Native PICA Format

The PICA texture unit supports a different texture format than the one in the standard OpenGL specifications. The texture format that is actually supported by the PICA texture unit is called the *native PICA format*. DMPGL 2.0 has features to load this native PICA format.

The native PICA format and standard OpenGL format mainly differ in the following three ways.

1. Byte Order: The standard OpenGL format and native PICA format have different byte orders because of internal address processing.
2. V-Flipping: The relationship between the *u* and *v* coordinates and texel placement is opposite in the V-direction between the standard OpenGL format and the native PICA format.
3. Addressing: Texels and compressed blocks are entered in the opposite order because of differences between linear and block addressing.

The standard OpenGL format and native PICA format also handle compressed and uncompressed textures differently. As used here, the term *uncompressed texture* indicates a format loaded with **`TexImage2D`** and *compressed texture* indicates the ETC format loaded with **`CompressedTexImage2D`**.

To convert an uncompressed texture from standard OpenGL format to native PICA format, first convert its V-flip, then convert its addressing, and finally convert the byte order. For a compressed texture, first convert its V-flip, then run ETC compression, then convert its addressing, and finally convert the byte order. Note that you must run V-flip conversion before ETC compression.

5.4.1 Byte Order

This section explains the differences in byte order between the native PICA format and standard OpenGL format. Compressed and uncompressed textures have different specifications.

5.4.1.1 Byte Order for Uncompressed Textures

Uncompressed textures loaded by **`TexImage2D`** with the ***`type`*** argument set to `UNSIGNED_BYTE` have texels with a different internal byte order in the native PICA format and standard OpenGL format. You can use a "byte swap" to change the byte order per texel and thus switch between the standard OpenGL format and native PICA format. Table 5-10 shows the required byte swap for each format.

Table 5-10 Byte-Order Differences Between the Standard OpenGL and Native PICA Formats

Format	Type	Bytes	Byte Swap
`RGBA`	`UNSIGNED_BYTE`	4	4-byte swap
`RGB`	`UNSIGNED_BYTE`	3	3-byte swap
`RGBA`	`UNSIGNED_SHORT_5_5_5_1`	2	None
`RGBA`	`UNSIGNED_SHORT_4_4_4_4`	2	None
`RGB`	`UNSIGNED_SHORT_5_6_5`	2	None
`LUMINANCE_ALPHA`	`UNSIGNED_BYTE`	2	2-byte swap
`LUMINANCE`	`UNSIGNED_BYTE`	1	None
`ALPHA`	`UNSIGNED_BYTE`	1	None
`LUMINANCE_ALPHA`	`UNSIGNED_BYTE_4_4_DMP`	1	None
`LUMINANCE`	`UNSIGNED_4BITS_DMP`	0.5	None

Format	Type	Bytes	Byte Swap
ALPHA	UNSIGNED_4BITS_DMP	0.5	None

When the two formats are compared, the byte order is different in the native PICA format when ***type*** is UNSIGNED_BYTE and there is more than one color component. Byte swaps reverse the order of bytes within each set of the given number of bytes. This is shown by the following figures.

Figure 5-2 4-Byte Swap

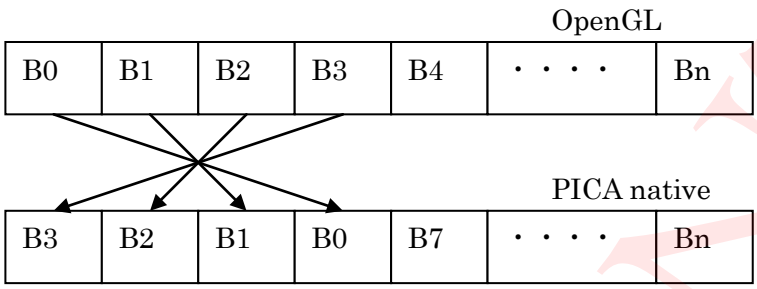


Figure 5-3 3-Byte Swap

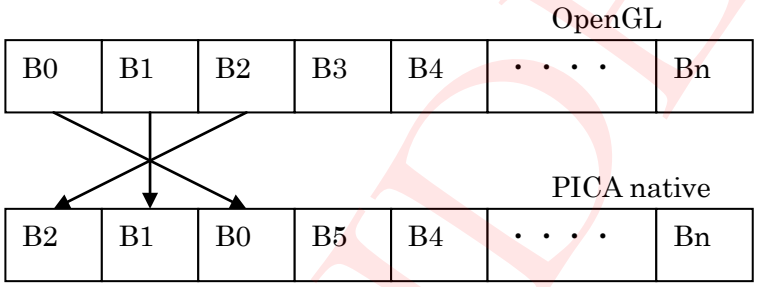
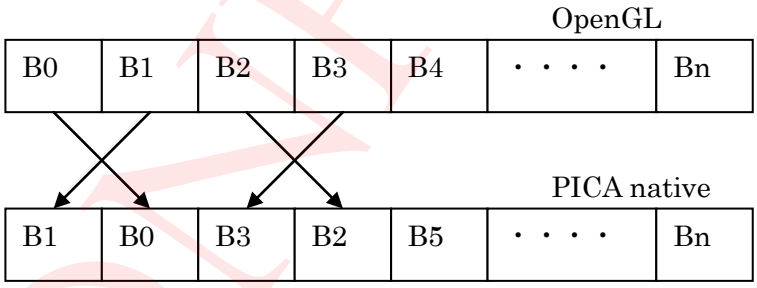


Figure 5-4 2-Byte Swap



5.4.1.2 Byte Order for Compressed Textures

Compressed textures in native PICA format are equivalent to the same data in standard OpenGL format divided into 8-byte chunks, with each pair of 8-byte chunks given in reverse order.

Figure 5-5 Byte Swap for Compressed Textures

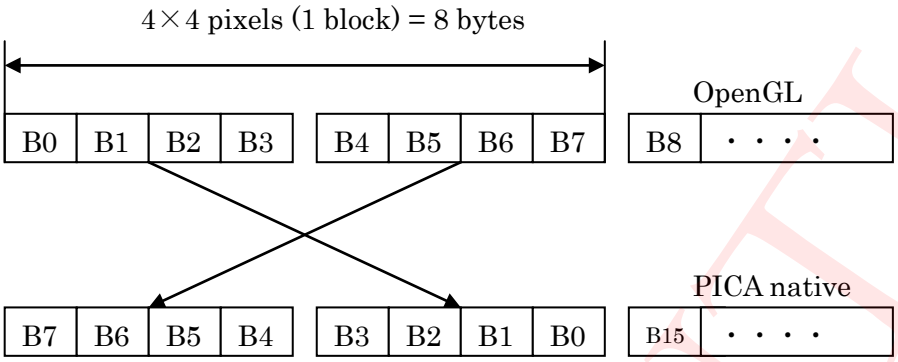
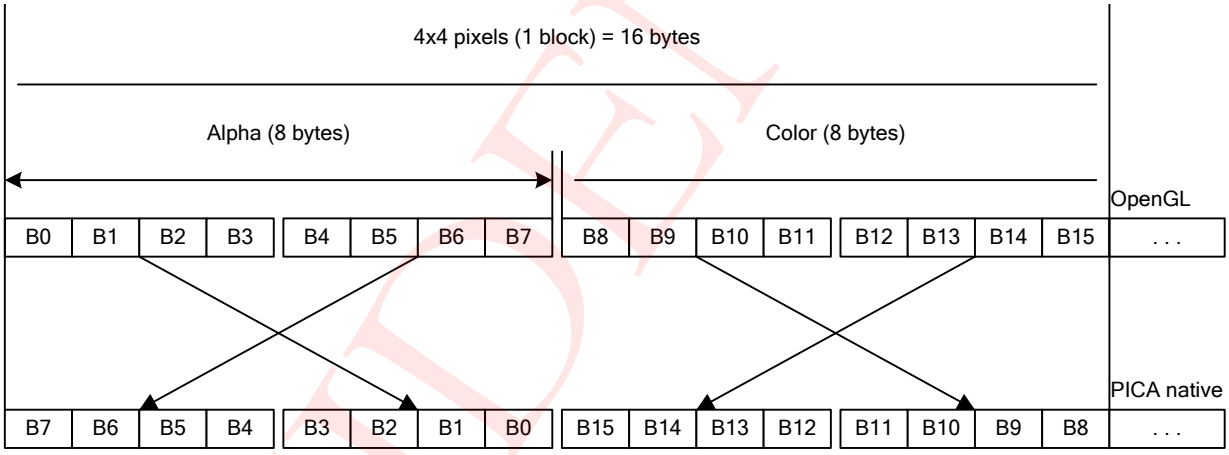


Figure 5-6 Byte Swap for ETC Textures with Alpha Components



When byte-swapping is taken into account, alpha data has the following relationship with texels.

Equation 5-2 Relationship Between Alpha Values and Texels

$$alpha = B7 + 256 \times \Big(B6 + 256 \times \Big(B5 + 256 \times \Big(B4 + 256 \times \Big(B3 + 256 \times \big(B2 + 256 \times (B1 + 256 \times B0) \big) \Big) \Big) \Big) \Big)$$

$$alpha(u,v) = bits[4 \times (4 \times u + v) + 3 \,..\, 4 \times (4 \times u + v) + 0]$$

In these equations, $B0–B7$ is a byte array. $alpha$ uses 64 bits to represent the alpha values for 4x4 texels. $bits[n..m]$ represent bits $n–m$ (where n is the more significant bit) in the 64 $alpha$ bits. $alpha(u,v)$ represents the 4-bit alpha value at position (u,v) in the 4x4 texels.

5.4.2 V-Flipping

This section explains differences in V-flipping between the native PICA format and standard OpenGL format. For both compressed and uncompressed textures, the OpenGL texture format stores the texel corresponding to the image coordinates (*u,v*) = (0.0, 0.0) at the starting address of texture data. The native PICA format, on the other hand, stores the texel with coordinates (*u,v*) = (0.0, 1.0) at the starting address of texture data.

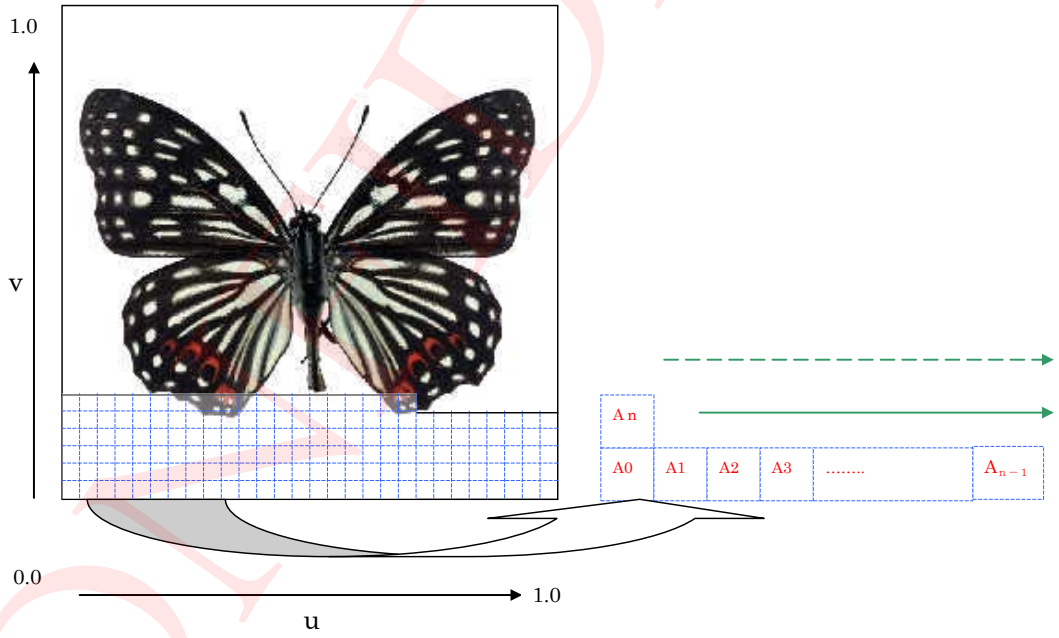
5.4.3 Addressing

This section explains addressing differences between the native PICA format and standard OpenGL format. The standard OpenGL format uses linear addressing, which stores data consecutively in the u direction starting with the texel at coordinates (*u,v*) = (0.0, 0.0). The native PICA format, on the other hand, uses block addressing to store blocks in a zig-zag pattern. Uncompressed and compressed textures have different block definitions and zig-zag patterns.

5.4.3.1 Addressing for Uncompressed Textures

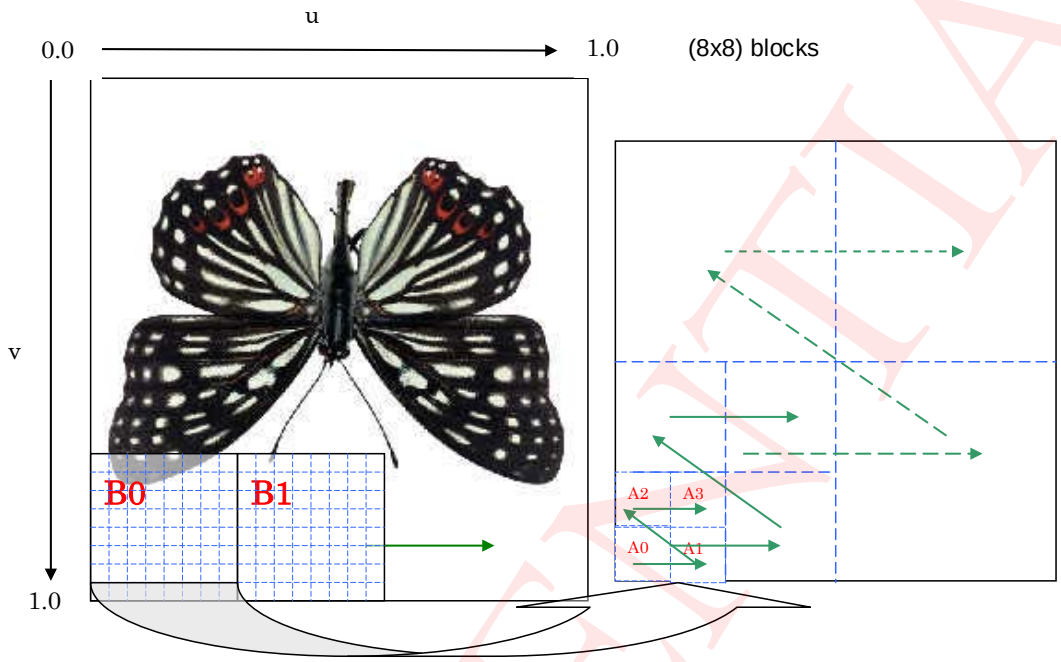
The standard OpenGL format's linear addressing stores data consecutively in the u direction starting with the texel at coordinates (*u,v*) = (0.0, 0.0). The native PICA format's block addressing, on the other hand, handles 8x8 texel data as a single block and stores blocks in a zig-zag pattern.

Figure 5-7 Linear Addressing in the OpenGL Format



In Figure 5-7, data is stored consecutively in the u direction starting with the texel at coordinates (*u,v*) = (0.0, 0.0).

Figure 5-8 Block Addressing in the Native PICA Format

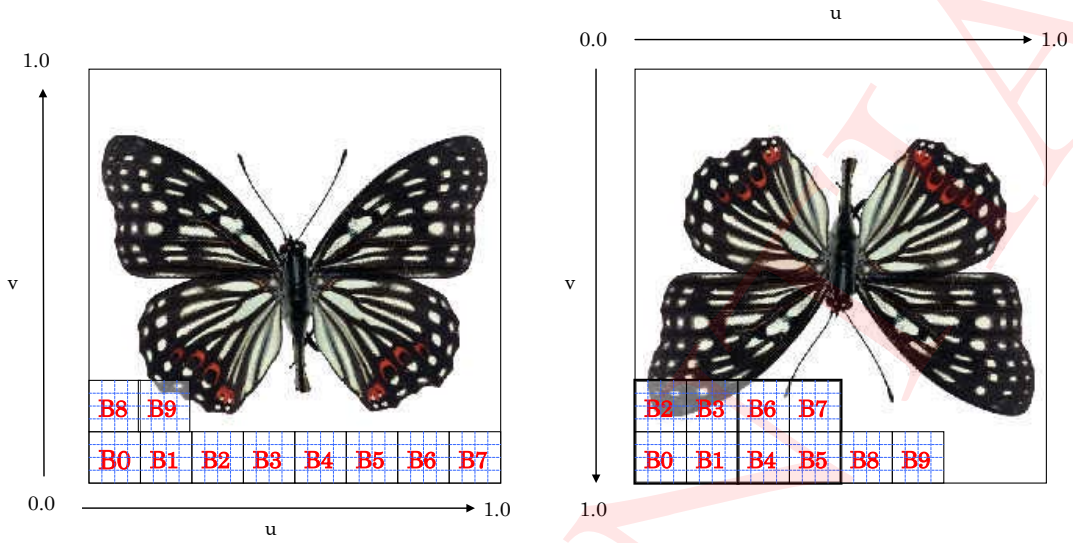


Using an 8x8 texel as a single block, Figure 5-8 shows how blocks are stored in the u direction starting with the one that includes the coordinates (*u,v*) = (0.0, 1.0). Each texel's data is stored in a zig-zag pattern within the block.

5.4.3.2 Addressing for Compressed Textures

In a compressed texture, a 4x4 collection of texels is called a *block*. The standard OpenGL format's linear addressing stores data consecutively in the u direction starting with the block that contains the coordinates (*u,v*) = (0.0, 0.0). In the native PICA format's block addressing, on the other hand, each 2x2 block (equivalent to an 8x8 collection of texel data) is called a *meta block*; data is stored consecutively in the u direction starting with the meta block that contains the coordinates (*u,v*) = (0.0, 1.0).

Figure 5-9 Linear Addressing (Left) and Block Addressing (Right) for Compressed Textures



In Figure 5-9, the labels B0, B1, … BN each indicate a block (4x4 texels). In block addressing, shown on the right, B0, B1, B2, and B3 constitute a single unit (a meta block), and data is stored in meta-block units along the u direction starting with the block that includes coordinates (*u,v*) = (0.0, 1.0).

5.5 Early Depth Tests

5.5.1 Overview

DMPGL 2.0 can perform a depth test before the texel-creation process, in addition to the depth test from the OpenGL ES standard. A depth test that precedes the texel-creation process is called an *early depth test*. By using early depth tests together with standard depth tests, you can reject unnecessary fragments earlier than you could have using only standard depth tests. Early depth tests are supported only in the actual hardware environment.

Early depth tests reject fragments at a lower precision than standard depth tests. Consequently, they may produce different results than standard depth tests. When a fragment mistakenly passes an early depth test it is called a *false pass*; if it mistakenly fails an early depth test it is called a *false fail*. Early depth tests are designed with the expectation that standard depth tests will correctly fail any fragment that is a false pass.

Note: Early depth tests must be configured in a way that prevents false fails. A fragment that should be rendered will not be when a false fail occurs.

Early depth tests must also be disabled when standard depth tests are disabled. False fails will occur if only early depth tests are enabled.

You cannot set the comparison function for early depth tests in the same operation that you use to set the comparison function for the standard depth tests; it requires a separate operation. You can set the

comparison function to LESS, LEQUAL, GREATER, or GEQUAL, but it must be the same function as the comparison function for standard depth tests. Setting different functions may cause false fails to occur. You cannot use early depth tests if standard depth tests are configured to use a comparison function, such as EQUAL or NEQUAL, that is not available to early depth tests.

You must always clear the early depth test buffer when you change the comparison function for early depth tests; otherwise, false fails will occur.

You cannot use early depth tests when the w buffer is enabled.

5.5.2 Clear Value for the Early Depth Buffer

When clearing the early depth buffer, you can specify any non-negative integer value between `0x000000` and `0xffffff`. Use `0x000000` when the standard depth buffer is cleared with a value of 0.0 and `0xffffff` when the standard depth buffer is cleared with a value of 1.0. If the standard depth buffer is cleared with any value other than the two just mentioned, multiply that value by `0xffffff` and then add an offset to get the value to use to clear the early depth buffer. You must provide an offset that accounts for the fact that early depth tests have lower precision than standard depth tests.

We recommend adding an offset of `0x1000` or greater when the early depth test function is LESS or LEQUAL, and an offset of `-0x1000` or less when the early depth test function is GREATER or GEQUAL. Values must be between `0x000000` and `0xffffff` after the offset has been added.

5.5.3 Block Mode for Early Depth Tests

You can choose either block-8 mode or block-32 mode for the render buffer. The render buffer is set to block-8 mode by default, but this must be changed to block-32 mode to use early depth tests.

Once rendering starts, it must continue to use the same block mode until the render buffer is cleared. You must clear the render buffer before rendering in a different block mode. Rendering results are not guaranteed if you render in a different block mode without clearing the render buffer. Consequently, all objects must be rendered with the render buffer configured to use block-32 mode when some objects have early depth tests enabled and others do not.

A buffer rendered in block-32 mode cannot be used for textures. Consequently, you cannot use early depth tests when rendering to textures.

In block-32 mode, you can render only to a render buffer with a width and height that are both multiples of 32 pixels. Also, the display buffer to which the render buffer's image is transferred must have a width and height that are also multiples of 32.

Note: Because block-32 mode limits sizes to multiples of 32, you cannot render to 240x400 or 320x400 framebuffers.

5.5.4 Enabling and Disabling Early Depth Tests

To enable early depth tests, call **`Enable`** with EARLY_DEPTH_TEST_DMP specified. To disable early depth tests, call **`Disable`** with EARLY_DEPTH_TEST_DMP specified. To determine whether early depth tests are enabled, call **`IsEnable`** with EARLY_DEPTH_TEST_DMP specified.

5.5.5 Setting the Comparison Function for Early Depth Tests

Use **EarlyDepthFuncDMP** to set the comparison function for early depth tests.

Code 5-12 EarlyDepthFuncDMP

```
void EarlyDepthFuncDMP(enum func);
```

To set the comparison function for early depth tests, call **EarlyDepthFuncDMP** with ***func*** set to GEQUAL, GREATER, LEQUAL, or LESS.

To get the comparison function for early depth tests, call **GetIntegerv** with ***pname*** set to EARLY_DEPTH_FUNC and the function will be given by ***params***.

5.5.6 Clearing the Early Depth Buffer

To clear the early depth buffer, call **Clear** with ***mask*** set to EARLY_DEPTH_BUFFER_BIT_DMP. To set the value by which to clear the buffer, call **ClearEarlyDepthDMP** with ***depth*** set to the clear value.

Code 5-13 ClearEarlyDepthDMP

```
void ClearEarlyDepthDMP(uint depth);
```

To get the clear value for the early depth buffer, call **GetIntegerv** with ***pname*** set to EARLY_DEPTH_CLEAR_VALUE_DMP and the value will be given by ***params***.

The depth value used to clear the depth buffer also affects the value used to clear the early depth buffer. The values are not necessarily equal; to calculate the early depth buffer's clear value, you must add an offset to the standard depth buffer's clear value that takes into account the lower accuracy of early depth tests. For more details on the offset, see section 5.5.2 Clear Value for the Early Depth Buffer

5.5.7 Changing to and Recovering from Block-32 Mode

Use **RenderBlockModeDMP** to set the pixel block mode for the render buffer.

Code 5-14 RenderBlockModeDMP

```
void RenderBlockModeDMP(enum mode);
```

You can set ***mode*** to RENDER_BLOCK8_MODE_DMP or RENDER_BLOCK32_MODE_DMP. To set the render buffer to block-32 mode, call **RenderBlockModeDMP** with ***mode*** set to RENDER_BLOCK32_MODE_DMP. To set the render buffer to block-8 mode, call **RenderBlockModeDMP** with ***mode*** set to RENDER_BLOCK8_MODE_DMP.

Call **GetIntegerv** with ***pname*** set to RENDER_BLOCK_MODE to get the render block mode in ***params***.

As with early depth tests, changes to the block mode are supported only in the actual hardware environment.

6 Reserved Fragment Shaders

6.1 Fragment Operations

This section explains the features of fragment operations. DMPGL 2.0 provides a feature to replace the standard OpenGL fragment pipeline (starting with the alpha test) with processing unique to DMPGL 2.0: the shadow and gas features. For more details, see section 6.4 DMP Shadows and section 6.6 Gas.

6.1.1 Switching Fragment Operations

To switch a fragment operation, call **`Uniform1i`** on the reserved uniform `dmp_FragOperation.mode`. Set ***`value`*** to `FRAGOP_MODE_GL_DMP`, `FRAGOP_MODE_SHADOW_DMP`, or `FRAGOP_MODE_GAS_ACC_DMP` to switch the fragment operation into standard, shadow, or gas mode, respectively.

6.1.2 List of Reserved Uniforms

Table 6-1 shows the settings for reserved uniforms that are used by fragment operations.

Table 6-1 Reserved Uniform Settings for Fragment Operations

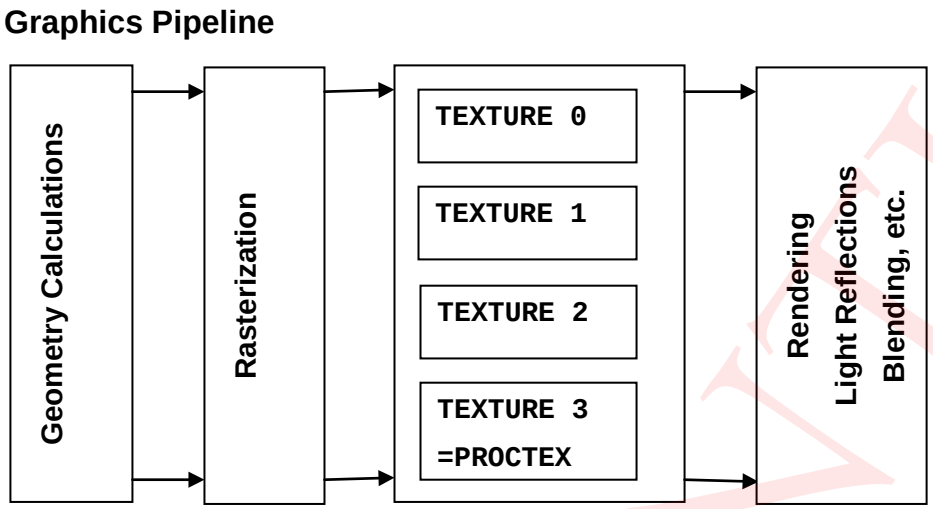
Uniform	Type	Value

`dmp_FragOperation.mode`	`int`	`FRAGOP_MODE_GL_DMP` (default)
		`FRAGOP_MODE_SHADOW_DMP`
		`FRAGOP_MODE_GAS_ACC_DMP`

6.2 Procedural Textures

This section explains the features of procedural textures. Procedural textures are one of the features of reserved fragment shaders in DMPGL 2.0. Unlike conventional texture units, which get colors by sampling texture images using texture coordinates, procedural textures calculate colors from texture coordinates using a formal procedure. The formal procedure includes conversions to polar coordinates and variations due to random numbers.

Figure 6-1 Graphics Pipeline for Procedural Textures



Four textures are built into the pipeline. TEXTURE3 is dedicated to procedural textures. The other texture units do not have procedural texture features. For details on the features of each texture unit, see section 5.1 Texture Units.

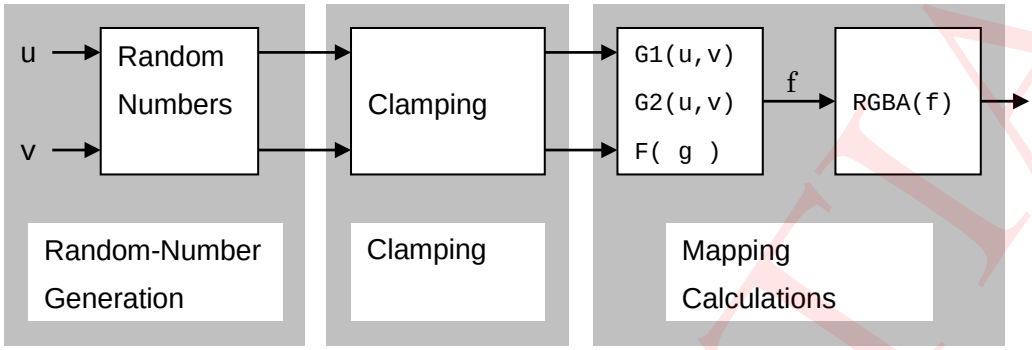
6.2.1 How to Use Procedural Textures

Although all the reserved uniforms `dmp_Texture[i]` (where `i` is 0,1,2, or 3) are defined for reserved fragment shaders, `dmp_Texture[3]` is the one used to configure procedural textures.

To enable procedural textures, call **`Uniform1i`** and set the reserved uniform `dmp_Texture[3].samplerType` to `TEXTURE_PROCEDURAL_DMP`. Texture combiners (see section 5.2 Texture Combiners) access the texels of a procedural texture as TEXTURE3.

The procedural texture unit accepts the same input texture coordinates (*u, v*) as other texture units. Afterwards, the texture coordinates are affected by random number generation and clamping, and are input to mapping calculations. Mapping calculations apply mapping and F functions to convert input coordinates into input values for the color table; the final texel color is obtained by accessing the color table. The mapping calculations can calculate color and alpha values independently. Figure 6-2 shows the flow of calculations.

Figure 6-2 Procedural Textures



Procedural textures comprise random number generation, clamping, and mapping calculations.

6.2.2 Creating and Assigning Lookup Tables

Procedural textures use lookup tables to specify the following functions.

- A function for random-number noise modulation (noise-modulation lookup table)
- The F function for mapping calculations (the F function lookup table)
- The RGBA value of the final texel (the color lookup table)

For details on data that must be set for each lookup table, see section 6.2.3 Random-Number Generation and section 6.2.5 Mapping Calculations.

6.2.3 Random-Number Generation

Random-number generation gets the (*u,v*) texture coordinates from the rasterizer and adds random numbers to them to get the new (*u',v'*) coordinates. The *noise* function that generates random numbers is defined as a black box.

Equation 6-1 Noise Function

$$u' = \begin{cases} |u| + NOISE \times scale\_u \ (u > 0) \\ |u| - NOISE \times scale\_u \ (u < 0) \end{cases}$$

$$v' = \begin{cases} |v| + NOISE \times scale\_v \ (v > 0) \\ |v| - NOISE \times scale\_v \ (v < 0) \end{cases}$$

$$NOISE = noise(\ (|u| + trans\_u) \times freq\_u, (|v| + trans\_v) \times freq\_v\ )$$

To enable or disable random numbers, use **`Uniform1i`** and set the reserved uniform `dmp_Texture[3].ptNoiseEnable` to TRUE or FALSE. Three variables control the types of random numbers that will be generated. If you consider the random numbers as a wave, these three variables correspond to the frequency, phase, and amplitude. Use the reserved uniforms `dmp_Texture[3].ptNoise{UV}` to control these variables. A set of these variables corresponds to a 3-value array with the frequency (*freq_{uv}*), phase (*trans_{uv}*), and amplitude (*scale_{uv}*) at indices 0, 1, and 2, respectively. To control the noise, call **`Uniform3f`** and set values for these reserved uniforms.

Table 6-2 Noise-Control Parameters

	k=0	k=1	k=2
`dmp_Texture[3].ptNoiseU[k]`	The frequency of U	The phase of U	The amplitude of U
`dmp_Texture[3].ptNoiseV[k]`	The frequency of V	The phase of V	The amplitude of V

Although the function that generates random numbers is a black box, as mentioned earlier, you can change one characteristic of the random numbers it generates: their continuity (this feature is called *noise modulation*). To control the noise modulation, register a noise modulation function in the noise modulation lookup table.

The noise modulation lookup table is accessed via `LUT_TEXTUREi_DMP`. To set it, call **`Uniformi`** on the reserved uniform `dmp_Texture[3].ptSamplerNoiseMap` and specify the appropriate lookup table number for `i` . The array given to the noise modulation lookup table has 256 floating-point entries; the first 128 must provide data for T and the last 128 must provide data for ΔT in Equation 6-2 (given $0 \leq k < 128$).

Equation 6-2 Noise Modulation

$$T_k = N\left(\frac{k}{128.0}\right)$$

$$\Delta T_k = N\left(\frac{k+1}{128.0}\right) - N\left(\frac{k}{128.0}\right)$$

The content of the specified lookup table is determined by the lookup table object bound to it and is configured by calling **`TexImage1D`** with ***`target`*** set to `LUT_TEXTUREi_DMP`, ***`level`*** set to 0, ***`internalformat`*** and ***`format`*** set to `LUMINANCEF_DMP`, ***`type`*** set to `FLOAT`, ***`width`*** set to 256, and ***`data`*** set to the T array given by Equation 6-2. For details on setting the content of lookup table objects, see section 5.1.7 Lookup Tables.

6.2.4 Clamping

During clamping, any u or v coordinate that does not fall within the range `[0,1]` is converted into a value that does. This feature resembles the wrap mode that can be configured for conventional texture objects. The clamping modes are: repeat mode, mirrored repeat mode, pulse mode, edge-clamp mode, and zero-clamp mode. Before being clamped by any of these modes, the u and v coordinates are shifted according to the method configured in the reserved uniform `dmp_Texture[3].ptShift{UV}`. You can set `EVEN_DMP`, `ODD_DMP`, or `NONE_DMP` as the shift method for either or both u and v. Table 6-3 shows how u and v are updated to u' and v' by the shift operation.

Table 6-3 Shift Mode Definitions

Shift Mode	Formula	Relationship Between *offset* and the Clamp Mode

`EVEN_DMP`	$u' = \left(\frac{\lfloor v \rfloor + 1}{2}\right) mod\ 2 + offset$
	$v' = \left(\frac{\lfloor u \rfloor + 1}{2}\right) mod\ 2 + offset$
		`MIRRORED_REPEAT 1.0`
		`SYMMETRICAL_REPEAT_DMP 0.5`
		`PULSE_DMP 0.5`
		`CLAMP_TO_EDGE 0.5`
		`CLAMP_TO_ZERO_DMP 0.5`

`ODD_DMP`	$u' = \left(\frac{\lfloor v \rfloor}{2}\right) mod\ 2 + offset$ $v' = \left(\frac{\lfloor u \rfloor}{2}\right) mod\ 2 + offset$	Identical to `EVEN_DMP`
`NONE_DMP`	$u' = u$ $v' = v$	None

After they are shifted, *u* and *v* are clamped according to the clamp mode setting. When *u* (or *v*) = *t*, *u* (or *v*) is updated according to the clamp mode by *t'* as shown in Table 6-4.

Table 6-4 Clamp Calculations for Each Mode

Clamp Mode	Clamped Coordinate
`SYMMETRICAL_REPEAT_DMP`	$t' = t - \lfloor t \rfloor$
`MIRRORED_REPEAT`	$t' = \begin{cases} t - \lfloor t \rfloor, & (\lfloor t \rfloor\ mod\ 2) == 0 \\ 1.0 - (t - \lfloor t \rfloor), & otherwise \end{cases}$
`PULSE_DMP`	$t' = \begin{cases} 1.0, & t > 0.5 \\ 0.0, & otherwise \end{cases}$
`CLAMP_TO_EDGE`	$t' = \begin{cases} 1.0, & t > 1.0 \\ t, & otherwise \end{cases}$
`CLAMP_TO_ZERO_DMP`	$t' = \begin{cases} 0.0, & t > 1.0 \\ t, & otherwise \end{cases}$

To set the clamp mode, call **`Uniform1i`** and specify `SYMMETRICAL_REPEAT_DMP`, `MIRRORED_REPEAT`, `PULSE_DMP`, `CLAMP_TO_EDGE`, or `CLAMP_TO_ZERO_DMP` for the reserved uniform `dmp_Texture[3].ptClamp{UV}`.

6.2.5 Mapping Calculations

The mapper takes the (*u*,*v*) coordinates output by the clamping calculations, applies various mapping calculations based on the mapping functions *G1*, *G2*, F_{rgb}, and F_a, and then calculates the final texel color by accessing the color lookup table. The mapping functions can take color and alpha values independently, or can take both at once (shared mode). The order of these calculations is shown by Figure 6-3 and Figure 6-4.

Figure 6-3 RGBA-Shared Mode for Mapping Calculations

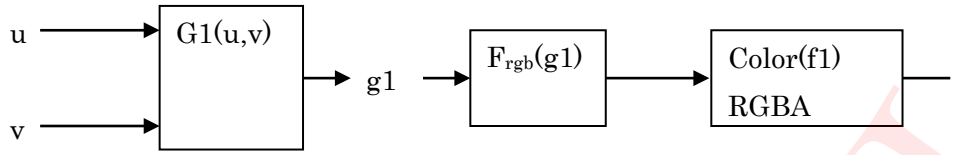
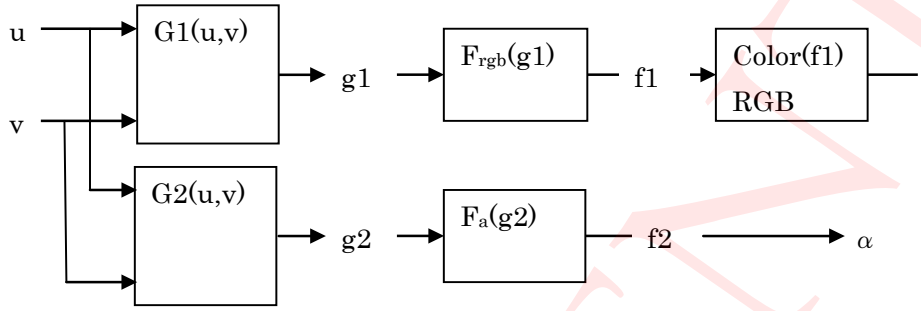


Figure 6-4 Independent Alpha Mode for Mapping Calculations



In RGBA-shared mode, the F function lookup table is accessed to convert the calculation results from the specified *G1* function. Next, the color lookup table is accessed to output the final texel color in RGBA format. In independent alpha mode, *G1* and *G2* are applied to *u* and *v*, yielding *g1* and *g2*. Then *g1* and *g2* are respectively looked up in the F function lookup tables F_{rgb} and F_a to obtain *f1* and *f2*. The final texel RGB color is obtained by looking up *f1* in the color lookup table. The *f2* value itself is the final texel alpha value. Equation 6-3 and Equation 6-4 show the calculation process.

Equation 6-3 RGBA-Shared Mode for Mapping Calculations

$$p_{RGBA} = Color\left(F_{RGBA}\left(G1(u,v)\right)\right)$$

Equation 6-4 Independent Alpha Mode for Mapping Calculations

$$p_{RGB} = Color\left(F_{RGB}\left(G1(u,v)\right)\right)$$

$$p_A = F_a\left(G2(u,v)\right)$$

To select RGBA-shared mode or indpendent alpha mode, use **`Uniform1i`** and set the reserved uniform `dmp_Texture[3].ptAlphaSeparate` to FALSE or TRUE, respectively. To set *G1* and *G2*, use **`Uniform1i`** and set the reserved uniforms `dmp_Texture[3].ptRgbMap` and `dmp_Texture[3].ptAlphaMap` to the parameters given in Table 6-5. The color lookup table can store more than one level of detail (LOD) and provides a filtering feature.

Table 6-5 G1 and G2 Modes

Parameter	Selected Function
PROCTEX_U_DMP	u
PROCTEX_V_DMP	v
PROCTEX_U2_DMP	u^2
PROCTEX_V2_DMP	v^2
PROCTEX_ADD_DMP	$\frac{1}{2}(u+v)$
PROCTEX_ADD2_DMP	$\frac{1}{2}(u^2+v^2)$
PROCTEX_ADDSQRT2_DMP	$\sqrt{\frac{1}{2}(u^2+v^2)}$
PROCTEX_MIN_DMP	$min(u,v)$
PROCTEX_MAX_DMP	$max(u,v)$
PROCTEX_RMAX_DMP	$\frac{1}{2}\left(\frac{1}{2}(u+v)+\sqrt{\frac{1}{2}(u^2+v^2)}\right)$

6.2.6 Lookup Tables for Mapping Calculations

Mapping calculations require an F function lookup table to implement F_{rgb} and F_a and a color lookup table to implement $Color$. They are all implemented as lookup tables using `LUT_TEXTUREi_DMP`.

6.2.6.1 F Function Lookup Tables

To set the F_{rgb} and F_a function lookup tables, call **`Uniformi`** on the reserved uniforms `dmp_Texture[3].ptSamplerRgbMap` and `dmp_Texture[3].ptSamplerAlphaMap` with the appropriate lookup table numbers specified. Arrays given to the lookup tables have 256 floating-point entries; the first 128 must be values for T and the last 128 must be values for ΔT in the following equation (given $0 \leq k < 128$).

Equation 6-5 F Function Lookup Table Arrays

$$T_k = F\left(\frac{k}{128.0}\right)$$

$$\Delta T_k = F\left(\frac{k+1}{128.0}\right) - F\left(\frac{k}{128.0}\right)$$

The content of the specified lookup table is determined by the lookup table object bound to it and is configured by calling **`TexImage1D`** with ***`target`*** set to `LUT_TEXTUREi_DMP`, ***`level`*** set to 0, ***`internalformat`*** and ***`format`*** set to `LUMINANCEF_DMP`, ***`type`*** set to `FLOAT`, ***`width`*** set to 256,

and ***data*** set to the T array given by Equation 6-5. For details on setting the content of lookup table objects, see section 5.1.7 Lookup Tables.

6.2.6.2 Color Lookup Tables

There are four color lookup tables, one for each of the R, G, B, and A channels. These lookup tables are accessed via `LUT_TEXTUREi_DMP`. To set them, call **`Uniformi`** on the reserved uniform `dmp_Texture[3].ptSampler{RGBA}` with the appropriate lookup table number specified. The content of the specified lookup table is determined by the lookup table object bound to it.

The lookup tables have different content depending on whether the LOD feature is used.

When LOD is not used, the array comprises T values (indicating the color component values) and ΔT values (indicating the difference between color component values). The array has a maximum length of 512 elements. The first 256 elements configure T and the last 256 elements configure ΔT.

Equation 6-6 Color Lookup Table Arrays

$$T_k = C\left(\frac{k}{256.0}\right)$$

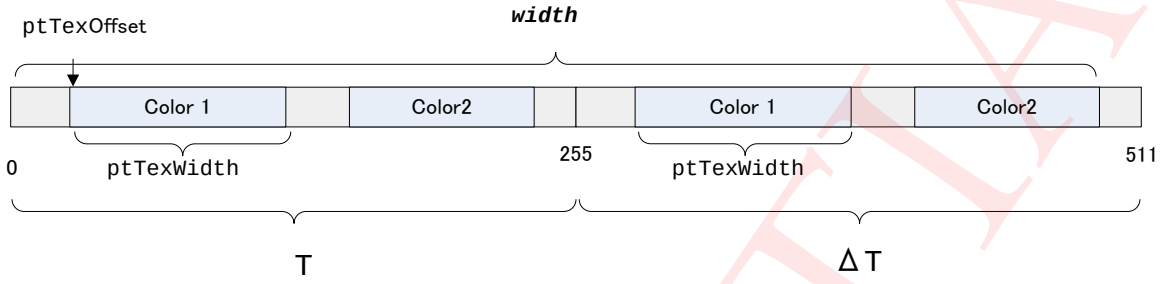
$$\Delta T_k = C\left(\frac{k+1}{256.0}\right) - C\left(\frac{k}{256.0}\right)$$

To set the content of a color lookup table, call **`TexImage1D`** with ***target*** set to `LUT_TEXTUREi_DMP`; ***level*** set to 0; ***internalformat*** and ***format*** set to `LUMINANCEF_DMP`; and ***type*** set to `FLOAT`. For ***width***, specify the number of elements until the last valid ΔT index in the array given by ***data***. The maximum value of ***width*** is 512. When configuring the content of the R channel, set ***data*** to a floating-point array of R components with the required number of data entries. The first 256 elements of the specified array set the channel color values and the last elements set the differences in color value.

The color lookup tables are used as subarrays when LOD is not in use. To specify the subarray to use, set its first element with `dmp_Texture[3].ptTexOffset` and its length with `dmp_Texture[3].ptTexWidth`. The subarray uses T values given by the `dmp_Texture[3].ptTexWidth` elements starting at index `dmp_Texture[3].ptTexOffset` (counted from the beginning of the array) and the ΔT values given by the `dmp_Texture[3].ptTexWidth` elements starting at index `dmp_Texture[3].ptTexOffset+256`. The value of `dmp_Texture[3].ptTexWidth` can be no greater than 128 and must be a power of 2.

To specify these characteristics of the subarray, call **`Uniform1i`** to set the reserved uniforms `dmp_Texture[3].ptTexOffset` and `dmp_Texture[3].ptTexWidth` each to an integer.

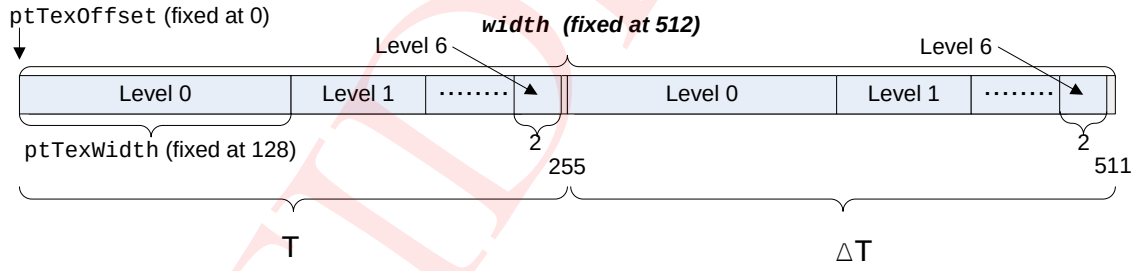
Figure 6-5 Color Lookup Table Settings



Specify the G, B, and A channels in the same way as the R channel.

Even when LOD is in use, the first 256 elements in the array represent the T values and the last 256 elements represent the ΔT values. The T and ΔT values must store all the LOD data, without any gaps. Data at the minimum LOD (level 0) and maximum LOD (level 6) must have widths of 128 and 2, respectively. When LOD is in use, `dmp_Texture[3].prTexOffset` must be 0 and `dmp_Texture[3].prTexWidth` must be 128.

Figure 6-6 Color Lookup Table Settings (When LOD Is in Use)



For details on setting the content of lookup table objects, see section 5.1.7 Lookup Tables. Table 6-6 shows the color lookup tables and corresponding uniforms.

Table 6-6 Color Lookup Table Assignments

Reserved Uniform	Content
`dmp_Texture[3].ptSamplerR`	Color lookup table number for R components
`dmp_Texture[3].ptSamplerG`	Color lookup table number for G components
`dmp_Texture[3].ptSamplerB`	Color lookup table number for B components
`dmp_Texture[3].ptSamplerA`	Color lookup table number for A components

You can apply filters when accessing color lookup tables. Apply the filters in the same way as with ordinary 2D textures. To set the filters, call **`Uniform1i`** on the reserved uniform `dmp_Texture[3].ptMinFilter` and assign the parameters shown below.

Table 6-7 Texture MinFilter Settings

Parameter	Application
`NEAREST`	Nearest point in the U and V directions. No LOD.
`LINEAR`	Linear interpolation in the U and V directions. No LOD.
`NEAREST_MIPMAP_NEAREST`	Nearest point in the U and V directions. Nearest point in the LOD direction.
`NEAREST_MIPMAP_LINEAR`	Nearest point in the U and V directions. Linear interpolation in the LOD direction.
`LINEAR_MIPMAP_NEAREST`	Linear interpolation in the U and V directions. Nearest point in the LOD direction.
`LINEAR_MIPMAP_LINEAR`	Linear interpolation in the U and V directions. Linear interpolation in the LOD direction.

A LOD bias applies when accessing a color lookup table. The LOD bias is configured by calling **`Uniform1f`** and assigning a number in the range `[0.0,6.0]` to `dmp_Texture[3].ptTexBias`. To disable the bias, call **`Uniform1f`** and set the same reserved uniform to `0.0`.

6.2.7 List of Reserved Uniforms

The following table shows the settings for reserved uniforms that are used by procedural textures.

Table 6-8 Reserved Uniform Settings for Procedural Textures

Uniform	Type	Values
`dmp_Texture[3].ptRgbMap`	`int`	• `PROCTEX_U_DMP` **(default)** • `PROCTEX_V_DMP` • `PROCTEX_U2_DMP` • `PROCTEX_V2_DMP` • `PROCTEX_ADD_DMP` • `PROCTEX_ADD2_DMP` • `PROCTEX_ADDSQRT2_DMP` • `PROCTEX_MIN_DMP` • `PROCTEX_MAX_DMP` • `PROCTEX_RMAX_DMP`
`dmp_Texture[3].ptAlphaMap`	`int`	Identical to `dmp_Texture[3].ptRgbMap`
`dmp_Texture[3].ptAlphaSeparate`	`bool`	• `TRUE` • `FALSE` **(default)**

Uniform	Type	Values
dmp_Texture[3].ptClampU	int	• SYMMETRICAL_REPEAT_DMP • MIRRORED_REPEAT • PULSE_DMP • CLAMP_TO_EDGE **(default)** • CLAMP_TO_ZERO_DMP
dmp_Texture[3].ptClampV	int	Identical to dmp_Texture[3].ptClampU
dmp_Texture[3].ptShiftU	int	• EVEN_DMP • ODD_DMP • NONE_DMP **(default)**
dmp_Texture[3].ptShiftV	int	Identical to dmp_Texture[3].ptShiftU
dmp_Texture[3].ptMinFilter	int	• NEAREST • LINEAR **(default)** • NEAREST_MIPMAP_NEAREST • NEAREST_MIPMAP_LINEAR • LINEAR_MIPMAP_NEAREST • LINEAR_MIPMAP_LINEAR
dmp_Texture[3].ptTexWidth	int	[0,128] 0 by default
dmp_Texture[3].ptTexOffset	int	[0,128] 0 by default
dmp_Texture[3].ptTexBias	float	0.0 or greater 0.5 by default
dmp_Texture[3].ptNoiseEnable	bool	• TRUE • FALSE **(default)**
dmp_Texture[3].ptNoiseU	vec3	Unspecified range (0.0, 0.0, 0.0) by default
dmp_Texture[3].ptNoiseV	vec3	Unspecified range (0.0, 0.0, 0.0) by default
dmp_Texture[3].ptSamplerRgbMap	int	[0,31] Undefined by default
dmp_Texture[3].ptSamplerAlphaMap	int	[0,31] Undefined by default
dmp_Texture[3].ptSamplerNoiseMap	int	[0,31] Undefined by default
dmp_Texture[3].ptSamplerR	int	[0,31] Undefined by default
dmp_Texture[3].ptSamplerG	int	[0,31] Undefined by default

Uniform	Type	Values
`dmp_Texture[3].ptSamplerB`	`int`	`[0,31]` Undefined by default
`dmp_Texture[3].ptSamplerA`	`int`	`[0,31]` Undefined by default

6.3 DMP Fragment Lighting

DMP fragment lighting is a feature that calculates fragment colors separately for each fragment. This fragment lighting calculates the primary and secondary colors. Fragment lighting includes perturbation of normal vectors referenced from textures ("bump mapping"), brightness adjustments due to shadows (the shadow feature), attentuation due to spotlights, and attenuation due to the distance between fragments and lights.

DMP fragment lighting calculations use multiple functions and combination methods. Fragment lighting uses functions that calculate the dot product of two vectors specified as the source; look up that dot product in a lookup table; and output the value yielded by the table. Fragment lighting can combine the values output by the functions via a bitwise AND or a bitwise OR operation. Fragment lighting outputs these AND or OR combined values as primary and secondary colors.

DMPGL 2.0 allows you to select from preset combinations of source vectors and lookup tables for each of the various functions as well as choosing between the preset combination methods. Arbitrary combinations are not allowed.

Section 6.3.1 View Coordinate System describes the coordinate system for input vectors. Section 6.3.2 Primary and Secondary Colors explains primary and secondary colors. Section 6.3.3 Lookup Tables (LUT) explains possible combinations of lookup tables and source values. Section 6.3.5 Shadow Attenuation explains shadow attenuation and section 6.3.6 Bump Mapping explains bump mapping. Geometric factors and fresnel factors are also available to reproduce special surface materials. See section 6.3.4 Geometry Factors for more information on geometric factors and section 6.3.7 Fresnel Factors for more information on fresnel factors. Section 6.3.9 Texture Combiner Input explains the appropriate texture combiner settings that are required to use fragment lighting.

To use DMP fragment lighting, set the reserved uniform `dmp_FragmentLighting.enabled` to TRUE. You must also compute valid quaternion attributes in the vertex shader and configure vertex shader output. Note that for lighting that takes tangent vectors into account, you must calculate quaternions that include the tangent component.

Various reserved uniforms are set to appropriate values as part of DMP fragment lighting. Note that the names of functions used to set reserved uniform values have different endings according to the uniform values that they set. For example, you would use **`Uniform4fv`** to set a reserved uniform with a constant type of `vec4`. The function name ending to use follows the description of uniform variables in section 2.10.4 of the OpenGL ES 2.0 specifications. For details on the reserved uniform types and their initial values, see section 6.3.10 List of Reserved Uniforms.

6.3.1 Eye Coordinate System

DMP fragment lighting operations are based on the eye coordinate system (also called *eyespace*), so the following several vectors are expected to be configured in eye coordinates. The vectors that you use do not necessarily have to be in the eye coordinate system, but they must all be in the same coordinate system. In many cases, vertex processing is carried out in the eye coordinate system and thus expects that reserved uniforms are specified as vectors in the eye coordinate system.

6.3.2 Primary and Secondary Colors

DMP fragment lighting generates two colors for each fragment. The first is the primary color cf_{pri}, which indicates the diffuse, ambient, and emissive colors. The second is the secondary color cf_{sec}, which indicates the specular color.

The primary color cf_{pri} is calculated by the following equation.

Equation 6-7 Primary Color

$$cf_{pri} = \sum \left((d_{cli} \times d_{cm} \times Lf_i \cdot Nf \times SdwAtt\,\mathrm{Pr}\, i_i + a_{cm} \times a_{cli}) \times Spot_i \times DistAtt_i \right) + e_{cm} + a_{cm} \times a_{cs}$$

Here, Lf_i is the light vector and Nf is the normal vector. e_{cm}, a_{cm}, d_{cm}, a_{cs}, a_{cli}, d_{cli}, and Lf_i are each set by the reserved uniforms in Table 6-9.

Table 6-9 Reserved Uniforms Related to Primary Color Settings (i Is an Integer from 0 to 7)

Parameter	Reserved Uniform
e_{cm}	`dmp_FragmentMaterial.emission`
a_{cm}	`dmp_FragmentMaterial.ambient`
d_{cm}	`dmp_FragmentMaterial.diffuse`
a_{cs}	`dmp_FragmentLighting.ambient`
a_{cli}	`dmp_FragmentLightSource[i].ambient`
d_{cli}	`dmp_FragmentLightSource[i].diffuse`
Lf_i	`dmp_FragmentLightSource[i].position`

The □□□ + □□□ × □□□ term will be 0 if light source 0 is disabled (in other words, if `dmp_FragmentLightSource[0].enabled` is set to `FALSE`).

Lf_i is specified by `dmp_FragmentLightSource[i].position` (where `i` is an integer between 0 and 7). The specified vectors must be defined in eye coordinates. Normalized vectors do not need to be specified here, but Lf_i is normalized within the equations. In the same way as vertex lighting, the fourth component of the specified vector determines whether a light source is parallel or a point.

$SdwAttPri_i$ is the shadow attenuation term. When the reserved uniforms `dmp_LightEnv.shadowPrimary` and `dmp_FragmentLightSource[i].shadowed` (`i` is an integer between 0 and 7) are set to TRUE, the brightness values reference the shadow map. When either of these reserved uniforms is set to FALSE, the brightness is `(1.0, 1.0, 1.0)`.

When the reserved uniform `dmp_FragmentLightSource[i].twoSideDiffuse` (`i` is an integer between 0 and 7) is set to FALSE, negative dot products are clamped to 0. The operations described above are only run on the R, G, and B components.

When the reserved uniform `dmp_LightEnv.invertShadow` is TRUE, 1.0 - $SdwAttPri_i$ is applied instead of $SdwAttPri_i$.

$DistAtt_i$ is the attenuation caused by the distance between a fragment and a light. This value either is referenced from a lookup table or is 1.0 (indicating that there is no attenuation contributed by the light) when TRUE or FALSE, respectively, are set in the reserved uniform `dmp_FragmentLightSource[i].distanceAttenuationEnabled` (`i` is an integer between 0 and 7). The distance attenuation term is disabled when `LIGHT_ENV_LAYER_CONFIG7_DMP` is set in the `dmp_LightEnv.config` reserved uniform.

The secondary color cf_{sec} is calculated by the following equation.

Equation 6-8 Secondary Color

$$cf_{sec} = \sum\left((s0_{cm} \times s0_{cli} \times D0 \times GF0_i \times s1_{cli} \times R \times D1 \times GF1_i) \times f_i \times SdwAttSec_i \times Spot_i \times DistAtt_i\right)$$

$s0_{cm}$, $s0_{cli}$, and $s1_{cli}$ are each set by the reserved uniforms in Table 6-10.

Table 6-10 Reserved Uniforms Related to Secondary Color Settings (i Is an Integer, 0–7)

Parameter	Reserved Uniform
$s0_{cm}$	`dmp_FragmentMaterial.specular0`
$s0_{cli}$	`dmp_FragmentLightSource[i].specular0`
$s1_{cli}$	`dmp_FragmentLightSource[i].specular1`

R is the reflection factor (color). $D0$ and $D1$ are distribution factors (real numbers). $GF0_i$ and $GF1_i$ are geometry factors (real numbers). These factors are either constants or are obtained from the lookup table, depending on the reserved uniform settings. $SdwAttSec_i$ is the attenuation caused by shadows.

When the reserved uniforms `dmp_LightEnv.shadowSecondary` and `dmp_FragmentLightSource[i].shadowed` (where `i` is an integer between 0 and 7) are set to TRUE, the brightness values reference the shadow map. When either of these reserved uniforms is set to FALSE, the brightness is `(1.0, 1.0, 1.0)`.

When the reserved uniform dmp_LightEnv.invertShadow is TRUE, 1.0 - $SdwAttSec_i$ is applied instead of $SdwAttSec_i$. The operations described above are only run on the R, G, and B components.

When the reserved uniform dmp_LightEnv.clampHighlights is TRUE and $Lf_i \cdot Nf$, is negative, then f_i is 0.0. In other words, that light has no specularity. If $Lf_i \cdot Nf$ is positive or dmp_LightEnv.clampHighlights is FALSE, $f_i f$ is 1.0.

$Spot_i$ and $DistAtt_i$ are the same as the terms used in the equation for the primary color.

With DMP fragment lighting, you can set different lighting parameters for each light. The DMPGL 2.0 implementation has a maximum of eight lights. To enable or disable each light, set the reserved uniform dmp_FragmentLightSource[i].enabled (i is an integer between 0 and 7) to TRUE or FALSE. If a light is disabled, its corresponding item is removed from the lighting equation. Even if all of the lights are disabled, setting the reserved uniform dmp_FragmentLighting.enabled to TRUE enables DMP fragment lighting.

6.3.3 Lookup Tables (LUTs)

Lookup tables are used for each reflection component—red (RR), green (RG), and blue (RB)—distribution 0 (D0), distribution 1 (D1), spotlight attenuation (SP), and fresnel factors (FR). The DMPGL 2.0 implementation allows 23 lookup tables to be accessed at once.

Table 6-11 shows which lookup tables are used for the individual items in the lighting equation. These are decided by the setting for the reserved uniform dmp_LightEnv.config. All of the lights use the same lookup tables.

Table 6-11 Configuration Provided by DMP Fragment Lighting

Configuration Type	Lookup Table Assignments						
	Rr	Rg	Rb	D0	D1	Fr	Spot
LIGHT_ENV_LAYER_CONFIG0_DMP	RR	RR	RR	D0	N/A	N/A	SP
LIGHT_ENV_LAYER_CONFIG1_DMP	RR	RR	RR	N/A	N/A	FR	SP
LIGHT_ENV_LAYER_CONFIG2_DMP	RR	RR	RR	D0	D1	N/A	N/A
LIGHT_ENV_LAYER_CONFIG3_DMP	N/A	N/A	N/A	D0	D1	FR	N/A
LIGHT_ENV_LAYER_CONFIG4_DMP	RR	RG	RB	D0	D1	N/A	SP
LIGHT_ENV_LAYER_CONFIG5_DMP	RR	RG	RB	D0	N/A	FR	SP
LIGHT_ENV_LAYER_CONFIG6_DMP	RR	RR	RR	D0	D1	FR	SP
LIGHT_ENV_LAYER_CONFIG7_DMP	RR	RG	RB	D0	D1	FR	SP

Entries marked as "N/A" indicate either the real number 1.0 or the color (1.0, 1.0, 1.0). When the reserved uniform dmp_LightEnv.lutEnabledRefl is FALSE, R is overwritten by the value set for the reserved uniform dmp_FragmentMaterial.specular1 and the lookup table is not used. By default, dmp_LightEnv.lutEnabledRefl is TRUE. Lookup tables RR, RG, RB, D0, D1, and FR are set by the reserved uniforms dmp_FragmentMaterial.sampler{RR,RG,RB,D0,D1,FR}.

A different SP lookup table is assigned for each light. A different distance attenuation lookup table is always assigned to each light regardless of the configuration type. SP and distance attenuation are configured by the reserved uniforms `dmp_FragmentLightSource[i].sampler{SP,DA}` (`i` is an integer between 0 and 7).

Use **`TexImage1D`** to specify lookup tables.

Code 6-1 TexImage1D

```
void TexImage1D(enum target, int level,
          int internalformat, sizei width, int border,
          enum format, enum type, void *data );
```

Each of the arguments follows the lookup table specifications in section 5.1.7 Lookup Tables, but the lookup table instance used by DMP fragment lighting is an array of 512 floating-point numbers. The first 256 elements are the sampling values for the lookup table and the next 256 elements are the differences between the individual sampling values. Set ***`width`*** to 512, ***`type`*** to FLOAT, and ***`data`*** to a pointer to a 512-element array of floating-point values.

The layout of the lookup table differs depending on whether its input values fall in the range `[0, 1.0]` or the range `[-1.0, 1.0]`. The former layout is possible when you access the lookup table using absolute-value input values. In this case, the input value is multiplied by 256 and then clamped to 255; the integer part of the resulting value is the table index. This index value is used as the address of a sampling value (one of the first 256 elements set by **`TexImage1D`**) and a difference value (one of the next 256 elements) in the lookup table. To get the output from the lookup table, multiply the input value by 256, take the fractional part of the result and multiply it by the difference value, then add the sampling value.

The same calculation method is used for input values that fall in the range `[-1.0, 1.0]`. The one difference is that the input value is multiplied by 128 and the integer part of that product is converted into a two's complement number before being used as an index. This further subdivides the lookup table into two parts. The first part is used for non-negative input values (sampling values for indices from `0` to `0x7f`) and the next part is used for negative input values (sampling values for indices from `0x80` to `0xff`).

To establish a mapping between a lookup table and each of the terms in the shading equations, set the reserved uniforms `dmp_FragmentMaterial.sampler{RR,RG,RB,D0,D1,FR}` and `dmp_FragmentLightSource[i].sampler{SP,DA}` (`i` is an integer between 0 and 7) equal to lookup table numbers, thus binding lookup table object numbers to the corresponding lookup tables. For details, see section 5.1.7 Lookup Tables.

You can select from the following preset combinations of source data required by the various fragment shader functions. Set the values in the reserved uniforms `dmp_LightEnv.lutInput{RR,RG,RB,D0,D1,SP,FR}`.

Table 6-12 Lookup Table Input Values

Input Value for Lookup Table Access	Meaning
LIGHT_ENV_NH_DMP	Cosine of the angle between the normal vector and half-angle vector
LIGHT_ENV_VH_DMP	Cosine of the angle between the view vector and half-angle vector
LIGHT_ENV_NV_DMP	Cosine of the angle between the normal vector and view vector
LIGHT_ENV_LN_DMP	Cosine of the angle between the light vector and the normal vector
LIGHT_ENV_SP_DMP	Cosine of the angle between the inverse light vector and the spotlight direction vector
LIGHT_ENV_CP_DMP	Cosine of the angle between the reflection of the half-angle vector on the tangent plane and the tangent vector

Table 6-12 shows which values can be specified. Note that LIGHT_ENV_CP_DMP here can only be used when the reserved uniform dmp_LightEnv.config is set to LIGHT_ENV_LAYER_CONFIG7_DMP. In addition, LIGHT_ENV_SP_DMP and LIGHT_ENV_CP_DMP can only be used with the reserved uniforms dmp_LightEnv.lutInput{D0,D1,SP}. The values that can be specified here are not used as input values to the distance attenuation lookup tables. For details, see section 6.3.9 Distance Attenuation Term. When dmp_FragmentMaterial.samplerFR uses a setting that is dependent on the light position (all input values other than LIGHT_ENV_NV_DMP are dependent on the light position), the light vector of the highest-numbered light among those currently enabled is used to access the lookup table.

When the reserved uniform dmp_FragmentLightSource[i].twoSideDiffuse (i is an integer between 0 and 7) has a value of FALSE, negative input values are clamped to 0. Lookup tables map input values in the range [-1.0, 1.0] into output values in the range [0, 1.0]. The reserved uniforms dmp_LightEnv.lutScale{RR,RG,RB,D0,D1,SP,FR} each specify a scaling value to apply to the output values of their respective lookup tables. You can currently set the following six values: 0.25, 0.5, 1.0, 2.0, 4.0, and 8.0.

Instead of using output values from a lookup table, you can also set the reserved uniforms dmp_LightEnv.lutEnabledD0 and dmp_LightEnv.lutEnabledD1 to FALSE to set $D0$ and $D1$ in Equation 6-8 to 1.0.

6.3.4 Geometry Factors

The reserved uniforms dmp_FragmentLightSource[i].geomFactor0 and dmp_FragmentLightSource[i].geomFactor1 control how the geometry factors $GF0_i$ and $GF1_i$ in Equation 6-8 are generated. $GF\{0,1\}_i$ is 1.0 when dmp_FragmentLightSource[i].geomFactor{0,1} are FALSE. $GF\{0,1\}_i$ is approximated by $max\left(0, \frac{Lf_i \cdot Nf}{(Lf_i+Vf)^2}\right)$ when dmp_FragmentLightSource[i].geomFactor{0,1} are TRUE. Actually, $GF\{0,1\}_i$ approximates the geometry factors of the Cook-Torrance lighting model. Vf indicates the view vector. Lf_i, Vf, and Nf are all normalized in within the equation.

Note: The defined $GF\{0,1\}_i$ is approximated using a method unique to DMPGL 2.0 that accounts for a hardware implementation of the geometry factors for the Cook-Torrance lighting model. This method was modeled on a formula (15.6) on p. 732 of *Principles of Digital Image Synthesis* by Andrew Glassner.

6.3.5 Shadow Attenuation Terms

Colors generated by lighting can be affected by shadow attenuation via the shadow attenuation terms $SdwAttPri_i$ and $SdwAttSec_i$. The values $SdwAttPri_i$ and $SdwAttSec_i$ are looked up from a shadow map; the reserved uniform `dmp_LightEnv.shadowSelector` specifies their texture unit.

When the reserved uniforms `dmp_FragmentLightSource[i].shadowed` (i is an integer between 0 and 7) and `dmp_LightEnv.shadowPrimary` are `TRUE`, the value looked up from the texture unit specified by `dmp_LightEnv.shadowSelector` is applied to $SdwAttPri_i$. When these reserved uniforms are `FALSE`, `(1.0, 1.0, 1.0)` is assigned to $SdwAttPri_i$.

In the same way, when the reserved uniforms `dmp_FragmentLightSource[i].shadowed` (i is an integer between 0 and 7) and `dmp_LightEnv.shadowSecondary` are `TRUE`, the value looked up from the texture unit specified by `dmp_LightEnv.shadowSelector` is applied to $SdwAttSec_i$. When these reserved uniforms are `FALSE`, `(1.0, 1.0, 1.0)` is assigned to $SdwAttSec_i$.

When the reserved uniform `dmp_LightEnv.invertShadow` is `TRUE`, the looked-up values are each subtracted from `(1.0, 1.0, 1.0)` to generate the two shadow attenuation terms.

6.3.6 Bump Mapping

DMP fragment lighting allows you to perturb normals and tangents that use a normal map. The reserved uniform `dmp_LightEnv.bumpSelector` specifies the texture unit that contains perturbation data for the normals and tangents. The reserved uniform `dmp_LightEnv.bumpMode` specifies the perturbation method. When `dmp_LightEnv.bumpMode` is `LIGHT_ENV_BUMP_NOT_USED_BUMP`, the normals and tangents are unaffected by perturbations. When the setting is `LIGHT_ENV_BUMP_AS_BUMP_DMP`, only the normals are perturbed.

Perturbation vectors are stored with their *x*, *y*, and *z* components encoded in the R, G, and B channels of texture samples (in other words, `-1.0` is encoded as the minimum brightness component and `1.0` is encoded as the maximum brightness component). When the setting is `LIGHT_ENV_BUMP_AS_TANG_DMP`, only the tangents are perturbed. When the reserved uniform `dmp_LightEnv.bumpRenorm` is `TRUE`, the *z* component of the perturbation vectors is recalculated according to Equation 6-9 below, instead of using the B component of the texture samples. Here, (b_x, b_y, b_z) indicates the bump-mapped vector.

Equation 6-9 Recalculating the Z Component of the Bump Map Perturbation Vectors

$$b_z = \sqrt{1.0 - \left(b_x^2 + b_y^2\right)}$$

6.3.7 Fresnel Factors

The equations described above are applied only to the R,G, and B components of the primary and secondary colors; the alpha components are set to `1.0`. However, it is possible for the alpha components to be overwritten by the real numbers indicated by `Fr` in Table 6-11. These values were originally intended to be used as approximations for the Fresnel factors, but they can also be used for other purposes. These values are generated in the same way as distribution 0 and distribution 1 ($D0$ and $D1$).

The content of the `Fr` factors is set by the corresponding reserved uniform `dmp_FragmentMaterial.samplerFr`. The alpha components of both the primary and secondary color are overwritten by `Fr` when the reserved uniform `dmp_LightEnv.fresnelSelector` is `LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP`. The alpha component is overwritten only for the secondary color with `LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP` and only for the primary color with `LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP`. The alpha component for the primary and secondary colors remains at `1.0` with `LIGHT_ENV_NO_FRESNEL_DMP`.

When the reserved uniform `dmp_LightEnv.fresnelSelector` is not `LIGHT_ENV_NO_FRESNEL_DMP` and `dmp_LightEnv.shadowAlpha` is `TRUE`, the corresponding alpha component can be attenuated by shadows (multiplied by shadow attenuation). It is also possible to flip the shadow texture output (this is controlled by `dmp_LightEnv.invertShadow`). When multiple lights are enabled, the light vector of the highest-numbered light among those currently enabled is used to generate a dot product as the input value to the lookup table `Fr`.

6.3.8 Spotlight Attenuation Term

Lookup tables are also used to generate the spotlight attenuation term $Spot$ in Equation 6-7 and Equation 6-8. Set the spotlight direction with the reserved uniform `dmp_FragmentLightSource[i].spotDirection` (i is an integer between 0 and 7). Just like a light-source vector, the vector indicating the spotlight direction is assumed to be in eye coordinates. Unlike a light-source vector, the vector specified for spotlight direction must already be normalized.

You can enable and disable spotlights individually by setting the reserved uniforms `dmp_FragmentLightSource[i].spotEnabled` (`i` is an integer between 0 and 7) equal to `TRUE` or `FALSE`. A separate lookup table is applied to each light using the lookup table number specified by `dmp_FragmentLightSource[i].samplerSP` (`i` is an integer between 0 and 7). However, even then the values set by `dmp_LightEnv.absLutInputSP`, `dmp_LightEnv.lutInputSP`, and `dmp_LightEnv.lutScaleSP` are applied commonly to all lights.

6.3.9 Distance Attenuation Term

Lookup tables are also used to generate the distance attenuation term $DistAtt_i$ in Equation 6-7 and Equation 6-8. You can enable and disable distance attenuation for individual lights by setting the reserved uniforms `dmp_FragmentLightSource[i].distanceAttenuationEnabled` equal to `TRUE` or `FALSE`. A separate lookup table is applied to each light using the lookup table number

specified by the reserved uniform `dmp_FragmentLightSource[i].samplerDA` (`i` is an integer between 0 and 7).

The lookup tables' data layout must be of the type that uses input values in the range `[0, 1.0]`. In other words, the same sampling is applied as when absolute values are used as the input values.

The following equation is applied to yield the lookup tables' input values.

Equation 6-10 Finding the Distance Attenuation Lookup Table Input Values

$$Scale \times \sqrt{(f_{position} - l_{position})^2} + Bias$$

$f_{position}$ is the fragment position and $l_{position}$ is the light position; both are assumed to be in eye coordinates. $Scale$ and $Bias$ are set by the reserved uniforms `dmp_FragmentLightSource[i].distanceAttenuationScale` and `dmp_FragmentLightSource[i].distanceAttenuationBias` (`i` is an integer between 0 and 7), respectively.

Distance attenuation does not account for parallel light sources. The value of $l_{position}$ only has meaning for point light sources.

The distance attenuation term is disabled when `LIGHT_ENV_LAYER_CONFIG7_DMP` is set in the `dmp_LightEnv.config` reserved uniform.

6.3.10 Texture Combiner Input

The primary and secondary colors generated by DMP fragment lighting are used as a single input source by the texture combiner. When the reserved uniform `dmp_TexEnv[i].srcRgb` or `dmp_TexEnv[i].srcAlpha` (i is between 0 and 5) is set to `FRAGMENT_PRIMARY_COLOR_DMP`, the primary color is defined as the texture combiner's input. When `FRAGMENT_SECONDARY_COLOR_DMP` is specified, the secondary color is defined as the texture combiner's input.

A value of (`0.0`, `0.0`, `0.0`, `1.0`) is output to both the primary and secondary colors when the reserved uniform `dmp_FragmentLighting.enabled` is `FALSE`.

6.3.11 List of Reserved Uniforms

Table 6-13 shows the settings (and default values) for reserved uniforms that are used by DMP fragment lighting (where `i` is 0, 1, 2, 3, 4, 5, 6, or 7). These reserved uniforms can be set for each light.

Table 6-13 Reserved Uniform Settings for Each Light

Uniform	Type	Value
`dmp_FragmentLightSource[i].enabled`	`bool`	• TRUE • FALSE **(default)**

Uniform	Type	Value
dmp_FragmentLightSource[i].ambient	vec4	Each value is in the range [0.0, 1.0] (0.0, 0.0, 0.0, 0.0) by default
dmp_FragmentLightSource[i].diffuse	vec4	Each value is in the range [0.0, 1.0] (0.0, 0.0, 0.0, 0.0) by default However, light 0 alone is (1.0, 1.0, 1.0, 1.0)
dmp_FragmentLightSource[i].specular0	vec4	Each value is in the range [0.0, 1.0] (0.0, 0.0, 0.0, 0.0) by default However, light 0 alone is (1.0, 1.0, 1.0, 1.0)
dmp_FragmentLightSource[i].specular1	vec4	Each value is in the range [0.0, 1.0] (0.0, 0.0, 0.0, 0.0) by default
dmp_FragmentLightSource[i].position	vec4	Unspecified range (0.0, 0.0, 0.0, 0.1) by default
dmp_FragmentLightSource[i].spotDirection	vec3	Unspecified range (0.0, 0.0, 0.0, -1.0) by default
dmp_FragmentLightSource[i].shadowed	bool	• TRUE • FALSE **(default)**
dmp_FragmentLightSource[i].geomFactor0	bool	• TRUE • FALSE **(default)**
dmp_FragmentLightSource[i].geomFactor1	bool	• TRUE • FALSE **(default)**
dmp_FragmentLightSource[i].twoSideDiffuse	bool	• TRUE • FALSE **(default)**
dmp_FragmentLightSource[i].spotEnabled	bool	• TRUE • FALSE **(default)**
dmp_FragmentLightSource[i].distanceAttenuationEnabled	bool	• TRUE • FALSE **(default)**
dmp_FragmentLightSource[i].distanceAttenuationBias	float	Unspecified range 0.0 by default
dmp_FragmentLightSource[i].distanceAttenuationScale	float	Unspecified range 1.0 by default

Uniform	Type	Value
dmp_FragmentLightSource[i].samplerSP	int	[0,31] Undefined by default
dmp_FragmentLightSource[i].samplerDA	int	[0,31] Undefined by default

Table 6-14 shows the reserved uniforms and values that can be set for materials using DMP fragment lighting, as well as their default values.

Table 6-14 Reserved Uniform Settings for Materials

Uniform	Type	Value
dmp_FragmentMaterial.sampler{D0,D1,RR,RG,RB,FR}	int	[0,31] Undefined by default
dmp_FragmentMaterial.ambient	vec4	Each value is in the range [0.0, 1.0] (0.2, 0.2, 0.2, 1.0) by default
dmp_FragmentMaterial.emission	vec4	Each value is in the range [0.0, 1.0] (0.0, 0.0, 0.0, 1.0) by default
dmp_FragmentMaterial.diffuse	vec4	Each value is in the range [0.0, 1.0] (0.8, 0.8, 0.8, 1.0) by default
dmp_FragmentMaterial.specular0	vec4	Each value is in the range [0.0, 1.0] (0.0, 0.0, 0.0, 1.0) by default
dmp_FragmentMaterial.specular1	vec4	Each value is in the range [0.0, 1.0] (0.0, 0.0, 0.0, 1.0) by default

Table 6-15 shows the reserved uniforms and values that can be set in the light environment of DMP fragment lighting, as well as their default values.

Table 6-15 Reserved Uniforms and Values That Can Be Set in the Light Environment

Uniform	Type	Value
dmp_FragmentLighting.enabled	bool	• TRUE **(default)** • FALSE
dmp_FragmentLighting.ambient	vec4	Each value is in the range [0.0, 1.0] (0.2, 0.2, 0.2, 1.0) by default

Uniform	Type	Value
dmp_LightEnv.absLutInput{D0,D1,RR,RG,RB,SP,FR}	bool	• TRUE • FALSE **(default)**
dmp_LightEnv.lutInput{D0,D1,SP}	int	• LIGHT_ENV_NH_DMP **(default)** • LIGHT_ENV_VH_DMP • LIGHT_ENV_NV_DMP • LIGHT_ENV_LN_DMP • LIGHT_ENV_SP_DMP • LIGHT_ENV_CP_DMP
dmp_LightEnv.lutInput{RR,RG,RB,FR}	int	• LIGHT_ENV_NH_DMP **(default)** • LIGHT_ENV_VH_DMP • LIGHT_ENV_NV_DMP • LIGHT_ENV_LN_DMP
dmp_LightEnv.lutScale{D0,D1,RR,RG,RB,SP,FR}	float	• 0.25 • 0.5 • 1.0 **(default)** • 2.0 • 4.0 • 8.0
dmp_LightEnv.shadowSelector	int	• TEXTURE0 **(default)** • TEXTURE1 • TEXTURE2 • TEXTURE3
dmp_LightEnv.bumpSelector	int	• TEXTURE0 **(default)** • TEXTURE1 • TEXTURE2 • TEXTURE3
dmp_LightEnv.bumpMode	int	• LIGHT_ENV_BUMP_NOT_USED_DMP **(default)** • LIGHT_ENV_BUMP_AS_BUMP_DMP • LIGHT_ENV_BUMP_AS_TANG_DMP
dmp_LightEnv.bumpRenorm	bool	• TRUE • FALSE **(default)**
dmp_LightEnv.config	int	• LIGHT_ENV_LAYER_CONFIG0_DMP **(default)** • LIGHT_ENV_LAYER_CONFIG1_DMP • LIGHT_ENV_LAYER_CONFIG2_DMP • LIGHT_ENV_LAYER_CONFIG3_DMP • LIGHT_ENV_LAYER_CONFIG4_DMP • LIGHT_ENV_LAYER_CONFIG5_DMP • LIGHT_ENV_LAYER_CONFIG6_DMP • LIGHT_ENV_LAYER_CONFIG7_DMP
dmp_LightEnv.invertShadow	bool	• TRUE • FALSE **(default)**

Uniform	Type	Value
dmp_LightEnv.shadowPrimary	bool	• TRUE • FALSE **(default)**
dmp_LightEnv.shadowSecondary	bool	• TRUE • FALSE **(default)**
dmp_LightEnv.shadowAlpha	bool	• TRUE • FALSE **(default)**
dmp_LightEnv.fresnelSelector	int	• LIGHT_ENV_NO_FRESNEL_DMP **(default)** • LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP • LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP • LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP
dmp_LightEnv.clampHighlights	bool	• TRUE **(default)** • FALSE
dmp_LightEnv.lutEnabledD0	bool	• TRUE • FALSE **(default)**
dmp_LightEnv.lutEnabledD1	bool	• TRUE • FALSE **(default)**
dmp_LightEnv.lutEnabledRefl	bool	• TRUE **(default)** • FALSE

6.4 DMP Shadows

DMPGL 2.0 shadows are expected to be rendered by two passes: first, the shadow accumulation pass creates a shadow buffer and then the reference pass accesses the shadow buffer to cast shadows.

DMPGL 2.0 allows shadows that use shadow textures. To enable shadow texture filtering, set the reserved uniform `dmp_Texture[0].samplerMode` to `SHADOW_2D_DMP` or `SHADOW_CUBE_DMP`.

Note that you must configure the vertex shader to calculate and output the R component of texture coordinates when using DMP shadows.

6.4.1 DMP Shadow Overview

The basic concept of shadow generation provided by DMPGL 2.0 is the same as the OpenGL 2-pass shadow generation algorithm that uses depth textures.

In the first pass, the scene is rendered in the light source's coordinate system. Depth information is stored in *shadow textures*, but these are not textures; they are the collected depth values representing the scene as viewed from the light source. (These are also called *shadow maps*.)

In the second pass, objects are determined to either be shaded (occluded from) or hit by the light source based on the depth values gathered in the first pass. Next, the scene is rendered. An object is

considered to be in shadow during the second pass if its distance from the light source's coordinates is larger than the depth value stored in the buffer generated during the first pass.

6.4.2 Shadow Texture Units

In some cases, only some texture units support shadow texture processing. In DMPGL 2.0, only texture unit 0 supports shadow texture processing. This is the reason for reserved uniform numbers to be fixed at 0 for all reserved uniforms relating to texture units later in this document.

6.4.3 Shadow Reference Pass

DMPGL 2.0 can only handle shadow textures using `SHADOW_DMP` or `SHADOW_NATIVE_DMP` for the texture format and `UNSIGNED_INT` for the type. Mipmapping of shadow textures is not supported. A shadow texture includes both depth values and shadow strength. The depth and shadow strength included in the shadow texture's texels are first compared to the R component of the texture coordinates, then converted to yield the final shadow strength.

The per-fragment depth value in light-source coordinates is calculated by Equation 6-11, which linearly interpolates the R component of the texture coordinates in eye space.

Equation 6-11 Per-Fragment Depth Value

$$z_l = r_{fr} - bias_{zl}$$

Here, $bias_{zl}$ is the actual offset (note the negative sign in front of the offset value!). Call **`Uniform1f`** to set the reserved uniform `dmp_Texture[0].shadowZBias` to the offset value. r_{fr} is the per-fragment R component value from the texture coordinates. *clamp* clamps in the range [0.0, 1.0].

When **`Uniform1i`** has been called to set the reserved uniform `dmp_Texture[0].perspectiveShadow` to FALSE and the reserved uniform `dmp_Texture[0].samplerMode` to `SHADOW_2D_DMP`, the texture coordinates s and t can be used directly to access shadow textures. Otherwise, Equation 6-12 is used to calculate the shadow texture position for each fragment.

Equation 6-12 Per-Fragment Shadow Texture Position

$$s = \frac{s_{fr}}{r_{fr}}$$

$$t = \frac{t_{fr}}{r_{fr}}$$

Here, s_{fr} and t_{fr} are the texture coordinates interpolated for each fragment. Note, however, that the texture coordinates *s,t,r* for each vertex must be multiplied by the *w* buffer's scaling factor. This is because the depth values rendered from the light source's position are assumed to be generated by linear interpolation in eye space.

If perspective projection is used when rendering to shadow textures (that is, if `dmp_Texture[0].perspectiveShadow` is set to `TRUE`), the $bias_{zl}$ parameter is determined by Equation 6-13 below.

Equation 6-13 Bias Parameter

$$bias_{zl} = alpha \times \frac{n}{f - n} + beta$$

Here, n and f are the distances to the near and far projection clipping planes specified when rendering shadow textures. $alpha$ is in the vicinity of 1.0 and $beta$ is in the vicinity of 0.0. The *w* buffer's scaling factor is assumed to be $1.0/f$. Setting $alpha$ to 1.0 or greater and $beta$ to 0.0 or greater is effective in controlling self-shadow aliasing when determining whether something is in shadow.

When calculating the texture coordinates might generate positions outside of the range from 0.0 to 1.0, the border color and texture wrapping mode can be configured to control the shadow texture's sampling results. When the texture wrapping mode is set to `CLAMP_TO_BORDER` for the *s* and *t* components of the texture coordinates, setting the shadow border color to (1.0, 1.0, 1.0, 1.0) or (0.0, 0.0, 0.0, 0.0) guarantees that the shadow texture's sampling results generate that color for fragment texture coordinates outside of the range from 0.0 to 1.0. Sampling results are undefined by the implementation if the wrapping mode is set to a value other than `CLAMP_TO_BORDER`. Sampling results are also undefined when the border color is set to a value other than (1.0, 1.0, 1.0, 1.0) or (0.0, 0.0, 0.0, 0.0).

The final shadow strength is obtained by comparing z_l to the depth value for the corresponding texel in the shadow texture. If the shadow texture's depth value is greater, the final shadow strength is 0.0; if z_l is greater, the original shadow strength of the texel in the shadow texture is retained as the final shadow strength. The texture unit outputs this value for each of the RGBA components.

6.4.4 Cube-Map Shadow Filtering

When rendering, you can use shadow textures together with cube mapping if both these two conditions have been met.

1. The target of **`TexImage2D`** was set to `TEXTURE_CUBE_MAP_POSITIVE_{X,Y,Z}` or `TEXTURE_CUBE_MAP_NEGATIVE_{X,Y,Z}` when the texture(s) bound to the texture unit were generated.
2. For the sampling mode during filtering, **`Uniform1i`** was called with the reserved uniform `dmp_Texture[0].samplerMode` set to `SHADOW_CUBE_MAP`.

If these conditions are met, the texture coordinate *s*, *t*, or *r* that has the largest absolute value is used for the depth comparison with the shadow texel depth value. The shadow texel position is calculated from the remaining two texture coordinates, and you can then use cube mapping to implement an omnidirectional shadow map. The shadow texel calculations follow section 3.7.5 of the OpenGL ES

2.0 specifications. The texture-wrapping mode is set equal to `CLAMP_TO_EDGE` for the *s* and *t* components of the texture coordinates with cube-map shadow filtering.

6.4.5 Shadow Accumulation Pass

Call **`Uniform1i`** with `FRAGOP_MODE_SHADOW_DMP` specified for the reserved uniform `dmp_FragOperation.mode` to replace all of the functionality described in Chapter 4 Per-Fragment Operations and the Framebuffer in the OpenGL ES 2.0 specifications with processing that accumulates shadow information in a shadow buffer.

The shadow buffer is implemented by attaching the shadow texture to the `COLOR_ATTACHMENT0` attachment point of the current framebuffer. In that situation, the texture or render buffers attached to the `DEPTH_ATTACHMENT` and `STENCIL_ATTACHMENT` attachment points are ignored.

Shadow information is accumulated into the shadow buffer in the light source's coordinate system. The accumulated information consists of the depth (distance) from the light source and the shadow strength (expressed as the g-component of color). Both the shadow depth information and strength information are values in the range of [0.0, 1.0].

The color with a g-component of 0.0 corresponds to an opaque, hard shadow. Any color with a g-component not 0.0 corresponds to a translucent shadow, and with 1.0 corresponds to a complete lack of shadows. The r-, b-, and a-components of color do not affect rendering results. You must set the *w* buffer's scale factor with the reserved uniform `dmp_FragOperation.wScale` because the depth values must be generated using linear interpolation in eye space. Call **`Uniform1i`** to set the reserved uniform `dmp_FragOperation.wScale` to any scale factor. For details, see section 6.9 w Buffer. The initial value is 0.0.

When rendering opaque hard shadows of color with a g-component of 0.0, only shadow depth information is updated, and shadow strength information is not updated. The depth for an opaque hard shadow fragment is compared against the depth of the corresponding pixels in the shadow buffer using the **LESS** comparison function (passing if the values are lower), and if the shadow fragment passes, the shadow depth information in the shadow buffer is updated for new fragments.

When rendering translucent shadows for a color that does not have a g-component of 0.0, only shadow strength information is updated and shadow depth information is not updated. The depth for a translucent shadow fragment is compared against the depth of the corresponding pixels in the shadow buffer using the **LESS** comparison function. If the fragment passes, the shadow strength information in the shadow buffer is updated for new fragments.

The shadow buffer is cleared with the color (1.0, 1.0, 1.0, 1.0) before the accumulation process starts. This initializes (clears) both the depth values and shadow strength. Be aware that the clear value is enabled for all r-, g-, b-, and a-components.

Several rendering passes are sometimes required to accumulate the necessary shadow information. The accumulation process must accumulate information for opaque, hard shadows first before translucent shadows. Results are not guaranteed if the order is reversed or information is accumulated alternating between opaque and translucent shadows.

6.4.6 Attenuation Factors

It is also possible to use silhouette rendering to create soft shadow regions for which the shadow strength changes in stages until it vanishes completely. Opaque, hard shadows must be rendered before this silhouette rendering.

To set the silhouette color, call **`Uniform4fv`** on the reserved uniform `dmp_Silhouette.color` with the color specified. You must set the g-component to 1.0. To produce incremental changes in the shadow strength around a silhouette, configure the remaining settings to be the same as the rendering pass for opaque, hard shadows such that the silhouette edge uses the color g-component of 0.0. This allows you to approximate a partial shadow region, which interpolates the shadow strength from 0.0 to 1.0 along a rectangle cut widthwise.

While accumulating the soft-shadow regions generated by silhouette rendering, you can add modulation based on the relative distance to the occluding object before saving the shadow strength to the shadow buffer (this is applied to the fragments that passed the depth test, and only the depth test, using an implicit "LESS" function). The shadow strength is modulated by the attenuation factor calculated according to the following equation.

Equation 6-14 Shadow Strength Attenuation Factor

$$\frac{1}{bias + scale \times \frac{1 - z_{frag}}{z_{rec}}}$$

Here, z_{frag} is the fragment's depth value and z_{rec} is the depth value of the surface on which the light falls (saved in advance to the shadow buffer). $bias$ and $scale$ allow you to control the "hardness" to apply to the penumbra (the region of half shadow); the harder the penumbra is, the more narrow it is. Using Equation 6-14 allows you to decrease the width of the penumbra in proportion to its proximity to the occluding object (the object casting it), and thus approximate the characteristics of a real shadow.

You can adjust $bias$ and $scale$ by setting values for the reserved uniforms `dmp_FragOperation.penumbraBias` and `dmp_FragOperation.penumbraScale`.

If no receiver is rendered in the shadow buffer, the shadow strength attenuation effect is not applied correctly.

6.4.7 Shadow Artifacts

The following parameters are used to control shadow artifacts. Values set by the reserved uniform `dmp_Texture[0].shadowZScale` control the negative offset value calculated from z_l before the depth values stored in the shadow texel are compared. Call **`Uniform1f`** to set the reserved uniform `dmp_Texture[0].shadowZScale` to any value. This offset is calculated from the derivatives of the texture coordinates in screen space by the following equation.

Equation 6-15 Offset

$$max\left(abs\left(\frac{dr}{width \times d\left(\frac{s}{r}\right)}\right), abs\left(\frac{dr}{height \times d\left(\frac{t}{r}\right)}\right)\right)$$

Here, $width$ and $height$ indicate the width and height of the shadow texture.

The scale factor specified by `dmp_Texture[0].shadowZScale` is multiplied by the offset value. The reserved uniform `dmp_Texture[0].shadowZScale` does not have a default value and must be set. The per-fragment offset value increases along with the scale factor, causing the shadow to be further separated from the object casting it. This can suppress artifacts due to self-shadow aliasing, but it unfortunately separates the shadow from the object casting it. The scale factor uses a value multiplied by the inverse of the shadow texture size.

6.4.8 Shadow Texture Format

The shadow depth information and strength information are stored in the shadow texture, and both have values in the range [0.0, 1.0] but the accuracy is different. To get the content of a shadow texture, call **`ReadPixels`** with `RGBA` specified as the format and `UNSIGNED_BYTE` specified as the type. The depth information and shadow strength included in the obtained shadow texture are formatted differently on the actual hardware environment versus the PICA on Desktop environment.

The actual hardware environment uses 8 bits for the shadow strength and 24 bits for the depth value. The R component represents the shadow strength as a value between `0x00` and `0xff`. A value of `0xff` indicates the absence of a soft shadow region and any other value indicates the presence of a soft shadow region. In other words, the shadow strength value [0.0, 1.0] corresponds to [0x00, 0xff]. The G, B, and A components together represent the depth value, holding the depth value's lower 8 bits, middle 8 bits, and upper 8 bits, respectively. The depth value is a value between `0x000000` and `0xffffff` and is scaled by the *near* and *far* values. Note that this scaling is uniform when the *w* buffer is enabled in the shadow accumulation path. The shadow depth value [0.0, 1.0] corresponds to [0x0, 0xffffff].

PICA on Desktop uses 8 bits for the shadow strength and 16 bits for the depth value. The R component represents the shadow strength; the B and A components together represent the depth value, holding the depth value's lower 8 bits and upper 8 bits, respectively. Despite this difference in format, the shadow strength and depth value have the same meaning as they do in the actual hardware environment. The shadow depth value [0.0, 1.0] corresponds to [0x0, 0xffff].

An unshadowed region produced by the shadow accumulation path has a shadow strength of `0xff` and a depth value of either `0xffffff` on the actual hardware or `0xffff` on PICA on Desktop. A region with only hard shadows has a shadow strength of `0xff` and any depth value other than `0xffffff` (on the actual hardware) or `0xffff` (on PICA on Desktop). A region with both hard and soft shadows has any shadow strength other than `0xff` and any depth value other than either `0xffffff` (actual hardware) or `0xffff` (PICA on Desktop).

6.4.9 List of Reserved Uniforms

Table 6-16 shows the settings (and default values) for reserved uniforms that are used by DMP shadows. Because only texture unit 0 allows shadow filtering in DMPGL 2.0, the `i` in `dmp_Texture[i]` is only set to 0.

Table 6-16 Reserved Uniform Settings for DMP Shadows

Name	Type	Value
`dmp_Texture[0].samplerType`	`int`	• `FALSE` **(default)** • `TEXTURE_2D` • `TEXTURE_CUBE_MAP` • `TEXTURE_SHADOW_2D_DMP` • `TEXTURE_SHADOW_CUBE_DMP` • `TEXTURE_PROJECTION_DMP`
`dmp_Texture[0].perspectiveShadow`	`bool`	• `TRUE` **(default)** • `FALSE`
`dmp_Texture[0].shadowZScale`	`float`	A value larger than 0.0 Undefined by default
`dmp_Texture[0].shadowZBias`	`float`	Arbitrary value 0.0 by default
`dmp_FragOperation.mode`	`int`	• `FRAGOP_MODE_GL_DMP` **(default)** • `FRAGOP_MODE_GAS_ACC_DMP` • `FRAGOP_MODE_SHADOW_DMP`
`dmp_FragOperation.penumbraScale`	`float`	Arbitrary value 0.0 by default
`dmp_FragOperation.penumbraBias`	`float`	Arbitrary value 1.0 by default
`dmp_FragOperation.wScale`	`float`	Arbitrary value 0.0 by default

6.5 Fog

Fog adjusts fragment colors according to their depth values and is nearly identical to the feature in OpenGL ES 1.1. This feature was removed from OpenGL ES 2.0 but can be used with DMPGL 2.0. The required parameters were set by **`Fog{if}`** in OpenGL ES 1.1 but are set by reserved uniforms in DMPGL 2.0.

The relationship between the depth and color values was specified by choosing between `LINEAR`, `EXP`, and `EXP2` in OpenGL ES 1.1, but is set by lookup tables in DMPGL 2.0.

The fog feature has two modes, fog and gas, and is also used by the gas feature. For details on the gas mode, see section 6.6 Gas.

6.5.1 Enabling Fog

To use the fog feature, enable either fog mode or gas mode by calling **`Uniform1i`** with the reserved uniform `dmp_Fog.mode` set to `FOG` or `GAS_DMP`. To disable the fog feature, set `dmp_Fog.mode` to `FALSE`.

6.5.2 Setting Lookup Table Content

Lookup tables set the relationship between depth values and fog. Lookup tables also implement `LINEAR`, `EXP`, and `EXP2`, which are defined in OpenGL ES 1.1.

These lookup tables are accessed via `LUT_TEXTUREi_DMP`. To set them, call **`Uniformi`** on the reserved uniform `dmp_Fog.sampler` with the appropriate lookup table number specified. The arrays given to these lookup tables have 256 floating-point entries; the first 128 must provide data for T and the last 128 must provide data for ΔT in Equation 6-15 (given $0 \leq k \leq 127$).

Equation 6-16 Lookup Table Array

$$T_k = N\left(\frac{k}{128.0}\right)$$

$$\Delta T_k = N\left(\frac{k+1}{128.0}\right) - N\left(\frac{k}{128.0}\right)$$

The content of the specified lookup table is determined by the lookup table object bound to it and is configured by calling **`TexImage1D`** with ***`target`*** set to `LUT_TEXTUREi_DMP`, ***`level`*** set to 0, ***`internalformat`*** and ***`format`*** set to `LUMINANCEF_DMP`, ***`type`*** set to `FLOAT`, ***`width`*** set to 256, and ***`data`*** set to the T array given by Equation 6-15. For details on setting the content of lookup table objects, see section 5.1.7 Lookup Tables.

6.5.3 Lookup Table Input Values

The *z* value in window coordinates is used as input to the lookup table. This differs from OpenGL ES 1.1, which uses the Z value in eye coordinates.

6.5.4 Specifying the Fog Color

To set the fog color, call **`Uniform3fv`** on the reserved uniform `dmp_Fog.color` with the color specified.

6.5.5 Fog Calculations

Fog calculations update the fragment color Cr to Cr'.

Equation 6-17 Fog Fragment Color

$$Cr' = f \times Cr + (1 - f) \times Cf$$

Here, Cf is the fog color and f is the output value from the lookup table.

6.5.6 Fog Z-Flipping

DMPGL 2.0 provides a z-flip mode that uses 1-z instead of *z* (where *z* is the depth value) to access the fog lookup tables. To enable or disable the z-flip mode, call **`Uniform1i`** on the reserved uniform `dmp_Fog.zFlip` and set it to `TRUE` or `FALSE`.

6.5.7 List of Reserved Uniforms

The following table shows the values set for reserved uniforms that are used by fog.

Table 6-17 Reserved Uniform Settings for Fog

Uniform	Type	Value
`dmp_Fog.mode`	`int`	• `FALSE` **(default)** • `FOG` • `GAS_DMP`
`dmp_Fog.color`	`vec3`	Each value is in the range `[0.0, 1.0]` `(0, 0, 0)` by default
`dmp_Fog.zFlip`	`bool`	• `TRUE` • `FALSE` **(default)**
`dmp_Fog.sampler`	`int`	`[0,31]` Undefined by default

Note: The lookup table specifications used by the fog feature in DMPGL 2.0 are easily misunderstood by people familiar with OpenGL. In OpenGL, the fog feature affects the z coordinate in eye space, but in DMPGL 2.0 it affects the depth value following the adjustment of the depth values by a perspective projection. As a result, changing the near or far clipping planes alters the fog effect. Also, using the same lookup table produces a different effect depending on whether the *w* buffer (see section 6.9 w Buffer) or normal depth buffer is used.

6.6 Gas

This section explains the gas feature. DMPGL 2.0 provides the gas feature to render gaseous objects. Gaseous objects have density and depth and are rendered pixel-by-pixel based on their foreground/background relationship with polygon models. Gaseous objects are rendered in multiple passes, including one that renders density and one that performs shading.

The density-rendering pass renders the density values of a gaseous object. Density rendering accounts for intersections with the depth values stored in the depth buffer. This pass generates gas textures (see section 6.6.1 Gas Textures) and runs other operations.

The shading pass shades gaseous objects based on the density values rendered by the density-rendering pass. This pass usually accesses gas textures.

6.6.1 Gas Textures

DMPGL 2.0 provides a special texture format for gases. Textures in this special format are called gas textures. To use gas textures, call **`TexImage2D`** with ***`target`*** set to `TEXTURE_2D`; ***`internalformat`*** and ***`format`*** set to `GAS_DMP` or `GAS_NATIVE_DMP`; and ***`type`*** set to `UNSIGNED_SHORT`. The gas texture's content is rendered by the density pass; each texel is a 32-bit value that comprises D1 and D2. D2 is a density value that accounts for intersections with the depth values stored in the depth buffer and D1 is a density value that does not account for these intersections. Gas textures always use point sampling, regardless of the `TEXTURE_MIN_FILTER` or `TEXTURE_MAG_FILTER` setting configured by **`TexParameter`**. Mipmapping of gas textures is not supported.

6.6.2 Rendering Density Values

DMPGL 2.0 provides features that render the density values of gaseous objects to the color buffer. The density information rendered here is used by the shading pass and is therefore usually rendered on gas textures.

6.6.2.1 Switching the Per-Fragment Operations

To render density values, you must switch the per-fragment operations into density-rendering mode. Call **`Uniform1i`** on the reserved uniform for per-fragment operations, `dmp_FragOperation.mode`, with ***`value`*** set to `FRAGOP_MODE_GAS_ACC_DMP` to switch per-fragment operations into density-rendering mode.

6.6.2.2 Density-Rendering Mode

In density-rendering mode, the alpha test (and subsequent processes) is replaced by density rendering. Density rendering stores 32-bit values that comprise two different values, $D1$ and $D2$, into the color buffer. These values have the same format as gas textures. Density rendering additively blends $D1$ and $D2$. In additive blending, the following equation is applied to $D1$ to yield $D1'$

Equation 6-18 Additive Blending of D1

$$D1' = D1 + Df$$

Here, Df uses the R component of the input fragment color.

A different set of equations, shown below, is applied to $D2$ to yield $D2'$.

Equation 6-19 Additive Blending of D2

$$DZ = (Zb - Zf < 0.0)\ ?\ 0.0 : (Zb - Zf) \times EZ$$

$$ATT = (DZ > 1.0)?\ 1.0 : DZ$$

$$D2' = D2 + Df \times ATT$$

EZ here is set by a call to **`Uniform1f`** on the reserved uniform `dmp_Gas.deltaZ` with ***`value`*** set to a floating-point value. Zb is the depth value stored in the depth buffer and Zf is the fragment's depth

value. If the depth test is run in GREATER or GEQUAL mode, additive blending of $D2$ uses the following equations.

Equation 6-20 Additive Blending of D2 With GREATER or GEQUAL Depth Tests

$$DZ = (Zb - Zf < 0.0)\ ?\ 0.0 : (Zb - Zf) \times EZ$$

$$ATT = (DZ > 1.0)?\ 1.0 : DZ$$

$$D2^{'} = D2 + Df \times (1.0 - ATT)$$

6.6.3 Shading

DMPGL 2.0 provides a feature to shade gaseous objects based on gas texture information.

6.6.3.1 Switching the Fog Mode

To shade gases, configure fog to use the gas mode. Call **`Uniform1i`** on the reserved uniform for fog, `dmp_Fog.mode`, with ***`value`*** set to `GAS_DMP` and the fog mode will be set to gas mode.The fog lookup table and shading lookup table are used when fog is configured to use gas mode. For information on setting the fog lookup table, see section 6.5 Fog. The fog output (Gr, Gg, Gb, Ga) is used for gas shading. Gr,Gg, and Gb are output from the shading lookup table and Ga is output from the fog lookup table.

6.6.3.2 Fog Input

When fog is configured to use gas mode, the output color from the next-to-last texture combiner (for example, `dmp_TexEnv[1]` when there are three texture combiners) is taken as the fog input color (r,g,b,a). Only the R component of the input color is used for shading. Also, the third argument of `dmp_TexEnv[last].srcRgb` and the third argument of `dmp_TexEnv[last].srcAlpha`, both from the final texture combiner, are also given as fog input (`"last"` would be 5 when there are six texture combiners). The latter two input values are taken and used as gas texture format values (density values $D1$ and $D2$) when fog is in gas mode.

6.6.3.3 Density Values Used for Shading

Shading uses $d1$ and $d2$, which are calculated based on the density values $D1$ and $D2$ given as fog input. $d1$ is yielded by the following equation.

Equation 6-21 Density Value d1

$$d1 = dk \times INVERTED\_ACC\_MAX1$$

To select between $D1$ or $D2$ for dk, use **`Uniform1i`** on the reserved uniform `dmp_Gas.shadingDensitySrc` with ***`value`*** set to `GAS_PLAIN_DENSITY` or `GAS_DEPTH_DENSITY`, respectively. If **`Uniform1i`** has been used to set the reserved uniform `dmp_Gas.autoAcc` to `TRUE`, $INVERTED\_ACC\_MAX1$ is the inverse of the maximum $D1$ value obtained by additive blending during density rendering. When `dmp_Gas.autoAcc` has been set to `FALSE`, $INVERTED\_ACC\_MAX1$ is the floating-point value that **`Uniform1f`** has set for the reserved uniform `dmp_Gas.accMax`.

$d2$, on the other hand, is yielded by the following equation.

Equation 6-22 Density Value d2

$$d2 = D2 \times GAS\_ATT$$

GAS_ATT here is set by a call to **`Uniform1f`** on the reserved uniform `dmp_Gas.attenuation` with ***`value`*** set to a floating-point value.

6.6.3.4 Shading Lookup Tables

The shading lookup table is used when fog is operating in gas mode. The shading lookup table comprises eight RGB colors and eight difference colors (together, RGB_i and RGB_DIF_i, where i = 0,1,…,7). If $Shading\_FUNC(x)$ is defined as the shading color given by the lookup table, the RGB colors and difference colors must be yielded by the following equation.

Equation 6-23 Shading Lookup Table Elements

$$RGB_i = Shading\_FUNC\left(\frac{i}{8}\right)$$

$$RGB\_DIF_i = Shading\_FUNC\left(\frac{i+1}{8}\right) - Shading\_FUNC\left(\frac{i}{8}\right)$$

There are three lookup tables, one for each of the R, G, and B channels. These lookup tables are accessed via `LUT_TEXTUREi_DMP`. To set them, call **`Uniform1i`** on the reserved uniform `dmp_Gas.samplerT{RGB}` with the appropriate lookup table number specified. The content of the specified lookup table is determined by the lookup table object bound to it and is configured by calling **`TexImage1D`** with ***`target`*** set to `LUT_TEXTUREi_DMP`, ***`level`*** set to 0, ***`internalformat`*** and ***`format`*** set to `LUMINANCEF_DMP`, ***`type`*** set to `FLOAT`, and ***`width`*** set to 16. To set the content of the R channel, specify ***`data`*** to be a floating-point array that comprises the R component of eight colors and eight difference values. The G and B channels are set in the same way. For details on setting the content of lookup table objects, see section 5.1.7 Lookup Tables.

6.6.3.5 Input to the Shading Lookup Tables

Using **`Uniform1i`**, set the reserved uniform `dmp_Gas.colorLutInput` to `GAS_DENSITY_DMP` or `GAS_LIGHT_FACTOR_DMP` to configure the shading lookup table's input value to be either $d1$ or the shading intensity II. The shading intensity II is defined by the following equation.

Equation 6-24 Shading Intensity

$$II = IG + IS$$

Here, IG is called the *planar shading intensity* and IS is called the *view shading intensity*. The planar shading intensity defines planar shading and is calculated by the following equations.

Equation 6-25 Planar Shading Intensity

$$Perturbation = (1.0 - lightAtt \times d1)$$

$$ig = r \times Perturbation$$

$$IG = (1.0 - ig) \times lightMin + ig \times lightMax$$

Here, $lightMin$ (the minimum intensity), $lightMax$ (the maximum intensity), and $lightAtt$ (the attenuation for density) are three floating-point numbers between 0.0 and 1.0 that control planar shading and are set by using **`Uniform3fv`** on the reserved uniform `dmp_Gas.lightXY`. r is the input R component for fog, as described in section 6.6.3.2 Fog Input. The view shading intensity IS, on the other hand, defines view shading and is calculated by the following equation.

Equation 6-26 View Shading Intensity

$$Perturbation = (1.0 - scattAtt \times d1)$$

$$is = LZ \times Perturbation$$

$$IS = (1.0 - is) \times scattMin + is \times scattMax$$

Here, $scattMin$ (the minimum intensity), $scattMax$ (the maximum intensity), $scattAtt$ (the attenuation for density), and LZ (the light direction in relation to the Z axis in the eye coordinate system) are four floating-point numbers between 0.0 and 1.0 that control view shading and are set by using **`Uniform4fv`** on the reserved uniform `dmp_Gas.lightZ`.

6.6.3.6 RGB Shading Values

The RGB shading values $G_{RGB} = (Gr, Gg, Gb)$ are output from the shading lookup table.

Equation 6-27 RGB Shading Values

$$G_{RGB} = Shading\_LUT(d1)$$

OR

$$G_{RGB} = Shading\_LUT(II)$$

6.6.3.7 Alpha Shading Value

The alpha shading value Ga is output from the fog lookup table. Ga is shown by the following equation that uses $d2$.

Equation 6-28 Alpha Shading Value

$$Ga = Fog\_LUT(d2)$$

6.6.4 List of Reserved Uniforms

The following table shows the values set for reserved uniforms that are used for gas.

Table 6-18 Reserved Uniform Settings for Gas

Uniform	Type	Value
`dmp_Gas.deltaZ`	`float`	Unrestricted range `10.0` by default
`dmp_Gas.shadingDensitySrc`	`int`	• `GAS_PLAIN_DENSITY_DMP` **(default)** • `GAS_DEPTH_DENSITY_DMP`
`dmp_Gas.autoAcc`	`bool`	• `TRUE` **(default)** • `FALSE`
`dmp_Gas.accMax`	`float`	Greater than or equal to `0.0` `1.0` by default
`dmp_Gas.colorLutInput`	`int`	• `GAS_DENSITY_DMP` • `GAS_LIGHT_FACTOR_DMP` **(default)**
`dmp_Gas.lightXY`	`vec3`	Each value is in the range `[0.0, 1.0]` `(0.0, 0.0, 0.0)` by default
`dmp_Gas.lightZ`	`vec4`	Each value is in the range `[0.0, 1.0]` `(0.0, 0.0, 0.0, 0.0)` by default
`dmp_Gas.samplerT{RGB}`	`int`	`[0,31]` Undefined by default
`dmp_Gas.attenuation`	`float`	Greater than or equal to `0.0` `1.0` by default

6.7 Alpha Tests

This section describes the alpha-test feature. DMPGL 2.0 provides features corresponding to the alpha tests defined in the OpenGL ES 1.1 specifications. You can use alpha tests to discard any fragment that does not pass a comparison of its alpha component with a configured reference value.

6.7.1 Enabling and Disabling Alpha Tests

To enable or disable alpha tests, call **`Uniform1i`** on the reserved uniform `dmp_FragOperation.enableAlphaTest` with ***`value`*** set to `TRUE` or `FALSE`. When alpha tests are disabled, the pipeline behaves as if all fragments always pass the alpha test. Alpha tests are disabled by default.

6.7.2 Setting Reference Values Used by Alpha Tests

To set the reference value with which to compare the fragment's alpha component during an alpha test, call **`Uniform1f`** on the reserved uniform `dmp_FragOperation.alphaRefValue` with ***`value`*** set to a floating-point number. The default value is 0.0. This value is clamped between 0 and 1 before it is used.

6.7.3 Controlling Alpha Test Comparisons

To set the comparison method used by alpha tests, call **`Uniform1i`** on the reserved uniform `dmp_FragOperation.alphaTestFunc` with ***`value`*** set to one of the values given by Table 6-19.

Table 6-19 Alpha Test Comparison Methods

Value	Meaning
`NEVER`	The fragment *never* passes.
`ALWAYS`	The fragment *always* passes.
`LESS`	The fragment passes if its alpha value is *smaller than* the reference value.
`LEQUAL`	The fragment passes if its alpha value is *less than or equal to* the reference value.
`EQUAL`	The fragment passes if its alpha value is *equal to* the reference value.
`GEQUAL`	The fragment passes if its alpha value is *greater than or equal to* the reference value
`GREATER`	The fragment passes if its alpha value is *greater than* the reference value.
`NOTEQUAL`	The fragment passes if its alpha value is *not equal to* the reference value.

The default setting is `ALWAYS`.

6.7.4 List of Reserved Uniforms

The following table shows the values set for reserved uniforms that are used by alpha tests.

Table 6-20 Reserved Uniform Settings for Alpha Tests

Uniform	Type	Value
`dmp_FragOperation.enableAlphaTest`	`bool`	• `TRUE` • `FALSE` **(default)**
`dmp_FragOperation.alphaRefValue`	`float`	`[0.0, 1.0]` `0.0` by default
`dmp_FragOperation.alphaTestFunc`	`int`	• `NEVER` • `ALWAYS` **(default)** • `LESS` • `LEQUAL` • `EQUAL` • `GEQUAL` • `GREATER` • `NOTEQUAL`

6.8 Clipping

This section describes clipping features. DMPGL 2.0 provides features that are nearly identical to clipping as defined in the OpenGL ES 1.1 specifications. Clipping features clip primitives to a clipping

volume. The clipping volume is the intersection of the viewing volume, which is a closed region, and all half-spaces defined by additional arbitrary clipping planes.

6.8.1 Clipping Volumes

The clipping volume is defined to be the intersection of the viewing volume and all half-spaces created by additional clipping planes. The following sections define the details of the viewing volume and half-spaces created by any additional clipping planes.

6.8.1.1 Definition of the Viewing Volume

The viewing volume is defined by Equation 6-29 in clip coordinates. This equation differs from the definition in the OpenGL ES 1.1 specifications. (See section 2.15 Coordinate Systems.)

Equation 6-29 Viewing Volume

$$-w_c \leq x_c \leq w_c$$

$$-w_c \leq y_c \leq w_c$$

$$-w_c \leq z_c \leq 0$$

6.8.1.2 Definition of a Clipping Plane

A clipping plane is defined by four coefficients, shown below.

Equation 6-30 Clipping Plane Coeffients

$$(p_1 \quad p_2 \quad p_3 \quad p_4)$$

The half-space created by the clipping planes is defined as the set of all points that satisfy the following equation.

Equation 6-31 Clipping Plane Half-Space

$$(p_1 \quad p_2 \quad p_3 \quad p_4) \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} \geq 0$$

To set these four coefficients, call **`Uniform4f`** on the reserved uniform `dmp_FragOperation.clippingPlane` with ***v0***, ***v1***, ***v2***, and ***v3*** each set to a floating-point value. The default value for all of these coefficients is 0.0. To enable or disable clipping planes, call **`Uniform1i`** on the reserved uniform `dmp_FragOperation.enableClippingPlane` with ***value*** set to TRUE or FALSE. Clipping planes are disabled by default.

6.8.2 List of Reserved Uniforms

The following table shows the values set for reserved uniforms that are used for clipping.

Table 6-21 Reserved Uniform Settings for Clipping

Uniform	Type	Value
`dmp_FragOperation.clippingPlane`	`vec4`	Unrestricted range (0.0, 0.0, 0.0, 0.0) by default
`dmp_FragOperation.enableClippingPlane`	`bool`	• TRUE • FALSE **(default)**

6.9 w Buffer

This section explains *w* buffer features. DMPGL 2.0 provides a feature to calculate depth values (the *z* values in window coordinates) without a perspective transformation.

6.9.1 Depth Values When the w Buffer Is Enabled

The following equation shows a depth value z_w when the *w* buffer is enabled.

Equation 6-32 w Buffer Depth Values

$$z_w = -sc\_w \times z_c$$

Here z_c is the *z* value in clip coordinates. sc_w is the floating-point number set by calling **`Uniform1f`** on the reserved uniform `dmp_FragOperation.wScale`. The value of sc_w must be set so as to keep z_w within the range from 0.0 to 1.0.

6.9.2 Enabling and Disabling the w Buffer

The *w* buffer is disabled when the reserved uniform `dmp_FragOperation.wScale` is set to 0.0 and enabled when the uniform is set to a nonzero value. It is disabled by default.

6.9.3 The w Buffer and the Depth Range

When the *w* buffer is enabled, the depth range configured by **`DepthRange`** is not used.

6.9.4 The w Buffer and Polygon Offset

You can use z-value offsets through `PolygonOffset`, regardless of the *w* buffer settings. When the *w* buffer is enabled, however, the value used as the offset is the product of W_c (the *w* value in clip coordinates) and the value specified as ***`units`*** to `PolygonOffset`.

6.9.5 List of Reserved Uniforms

The following table shows the values set for reserved uniforms that are used by the *w* buffer.

Table 6-22 Reserved Uniform Settings for the w Buffer

Uniform	Type	Value
`dmp_FragOperation.wScale`	`float`	Unspecified range `0.0` by default

7 Miscellaneous

DMPGL 2.0 is defined as an interface that resembles OpenGL ES 2.0, but there are several differences. This chapter summarizes these differences. All differences related to textures are described in Chapter 5 Rasterization. All other differences are described in this chapter.

7.1 Logical Operations

OpenGL ES 1.1 can perform logical operations on images between a source and destination image. These features were removed from the OpenGL ES 2.0 specifications but are usable with DMPGL 2.0. To enable or disable logical operations on images, call **`Enable`** or **`Disable`** with the ***`cap`*** argument set to `COLOR_LOGIC_OP`. To set the logical operation, call **`LogicOp`** with the ***`op`*** argument set to the operator. This specification is identical to OpenGL ES 1.1.

Code 7-1 LogicOp

```
void LogicOp(enum op);
```

The following table shows the operators that you can specify as the ***op*** argument. In the table, *s* indicates the *source* and *d* indicates the *destination*.

Table 7-1 Logical Operators for Images

Argument Value	Operation
CLEAR	0
AND	$s \wedge d$
AND_REVERSE	$s \wedge \neg d$
COPY	s
AND_INVERTED	$\neg s \wedge d$
NOOP	d
XOR	$s\ xor\ d$
OR	$s \vee d$
NOR	$\neg(s \vee d)$
EQUIV	$\neg(s\ xor\ d)$
INVERT	$\neg d$
OR_REVERSE	$s \vee \neg d$
COPY_INVERTED	$\neg s$
OR_INVERTED	$\neg s \vee d$
NAND	$\neg(s \wedge d)$
SET	$all\ 1$

To get the current setting, call **GetIntegerv** with the ***pname*** argument set to LOGIC_OP_MODE.

7.2 Flush and Finish

DMPGL 2.0 does not distinguish between **Flush** and **Finish**. They have the same implementation.

Code 7-2 Flush and Finish

```
void Flush(void);
void Finish(void);
```

7.3 Enable and Disable

The specifications for **Enable** and **Disable** are almost identical to OpenGL ES 2.0. In DMPGL 2.0, you cannot specify TEXTURE_2D or TEXTURE_CUBE_MAP to the **Enable** or **Disable** function. For details, see section 5.1.1 Enabling Texture Units.

7.4 DrawElements and DrawArrays

DrawArrays and **DrawElements** are used to render primitives. However, in DMPGL 2.0 the usable argument range differs from OpenGL 2.0.

Code 7-3 DrawElements and DrawArrays

```
void DrawArrays(enum mode, int first, sizei count);
void DrawElements(enum mode, sizei count, enum type, void *indices);
```

The ***mode*** argument specifies the mode for rendering primitives. You cannot specify POINTS, LINE_STRIP, LINE_LOOP, or LINES to the ***mode*** argument in DMPGL 2.0. Points and lines use the reserved geometry shader with the new GEOMETRY_PRIMITIVE_DMP specified. For details, see section 3.3 Geometry Shaders. TRIANGLES, TRIANGLE_STRIP, and TRIANGLE_FAN can be used in the same way as the OpenGL ES 2.0 specifications. You also specify GEOMETRY_PRIMITIVE_DMP when using the following new features that do not exist in the OpenGL ES 2.0 specifications: silhouettes, subdivision patches, and particle systems. For details on silhouettes, subdivision patches, and particle systems, see Chapter 4 Primitives.

7.5 LineWidth

LineWidth exists in OpenGL ES 2.0 but not in DMPGL 2.0.

7.6 PixelStorei

PixelStorei exists in OpenGL ES 2.0 but not in DMPGL 2.0.

7.7 SampleCoverage

SampleCoverage exists in OpenGL ES 2.0 but not in DMPGL 2.0.

7.8 ReadPixels

Use **ReadPixels** to read the content of the framebuffer.

Code 7-4 ReadPixels

```
void ReadPixels(int x, int y, sizei width, sizei height,
          enum format, enum type, void *data);
```

You can configure the DMPGL 2.0 render buffer to use one of two block modes. Ordinarily the mode used is block-8 mode, but block-32 mode is used with early depth tests. For further details on the block modes, see section 5.5.3 Block Mode for Early Depth Tests.

The block mode in effect when when objects are rendered must be the same as the block mode in effect when **ReadPixels** is used. The behavior of **ReadPixels** is not guaranteed when these block modes differ.

By specifying DEPTH_COMPONENT for ***format***, it is possible to read out the contents of the depth buffer. When doing so, specify either UNSIGNED_INT, UNSIGNED_INT_24_DMP, UNSIGNED_SHORT, or UNSIGNED_BYTE for ***type*** to get depth values in (respectively) 32 bits, 24 bits, 16 bits, or 8 bits per pixel. Each pixel's depth value is obtained with the lowest-level bits first. For example, if ***type*** is UNSIGNED_INT_24_DMP, a single pixel's value is obtained in three bytes, in this order: lower 8 bits, middle 8 bits, upper 8 bits. If the actually-rendered depth values have a different bit width than the ***type*** setting, this function gets values that have been converted to the width of ***type***.

When STENCIL_INDEX is specified for ***format***, the stencil buffer content can be read. In such cases, UNSIGNED_BYTE must be specified for ***type***, and the current depth buffer must be in DEPTH24_STENCIL8_EXT format. The stencil value for each pixel is read into each byte of ***data***, with each byte corresponding to one pixel.

When DEPTH24_STENCIL8_EXT is specified for ***format***, the depth buffer and stencil buffer content can be read in a combined format with 24 bits for the depth value and 8 bits for the stencil value. In such cases, UNSIGNED_INT must be specified for ***type***, and the current depth buffer must be in DEPTH24_STENCIL8_EXT format. One pixel is obtained as a four-byte value, in order as the least signficant 8 bits, middle 8 bits, and most significant 8 bits of the depth value, and then 8 bits for the stencil value.

An INVALID_ENUM error is generated when an invalid combination is specified for the ***format*** and ***type*** arugments.

In all other respects, this feature operates in the same way as it does in OpenGL ES 1.1 and 2.0.

7.9 Framebuffer Objects

DMPGL 2.0 handles framebuffer objects in compliance with section 4.4 Framebuffer Objects in the OpenGL ES 2.0 specifications, but does not support all of the OpenGL ES 2.0 specifications. There are also some DMP-specific extended specifications. In addition, the PICA on Desktop environment differs from the actual hardware environment. This section explains all of the differences in these specifications.

The DMPGL 2.0 implementation allows you to use the following image formats for the render buffer, in addition to those mentioned in section 4.4.5 of the OpenGL ES 2.0 specifications.

- `RGBA8_OES`
- `DEPTH_COMPONENT24_OES`
- `DEPTH24_STENCIL8_EXT`

The `STENCIL_INDEX8` format in the OpenGL ES 2.0 specifications is not supported by DMPGL 2.0.

Table 7-2 Image Formats for the Render Buffer

Sized Internal Format	Renderable Type	R bits	G bits	B bits	A bits	D bits	S bits
`DEPTH_COMPONENT16`	depth-renderable					16	
`DEPTH_COMPONENT24_OES`	depth-renderable					24	
`RGBA4`	color-renderable	4	4	4	4		
`RGB5_A1`	color-renderable	5	5	5	1		
`RGB565`	color-renderable	5	6	5			
`RGBA8_OES`	color-renderable	8	8	8	8		
`DEPTH24_STENCIL8_EXT`	depth and stencil-renderable					24	8

`DEPTH24_STENCIL8_EXT` is a 32-bit format that combines the 8-bit stencil buffer with the 24-bit depth buffer defined by the `EXT_packed_depth_stencil` extension.

You must set the ***`attach`*** argument to `DEPTH_STENCIL_ATTACHMENT` when using **`FramebufferRenderbuffer`** to attach a render buffer in the `DEPTH24_STENCIL8_EXT` format to the framebuffer.

You can use **`CopyTexImage2D`** or **`CopyTexSubImage2D`** to transfer the content of the render buffer that is attached to the color attachment point into a texture, but the same restrictions as in section 5.1.4 Copying From the Framebuffer apply. In other words, the internal format of the attached render buffer must be same as the internal format of the texture to which data is transferred. DMPGL 2.0 does not convert between formats when transferring data.

7.9.1 Specifications Particular to the PICA on Desktop Environment

After DMPGL initialization, the default render buffers are attached to the default framebuffer. These default render buffers cannot be attached to a non-default framebuffer. A non-default render buffer also cannot be attached to the default framebuffer.

When using PICA on Desktop on some host PC environments, you might not be able to properly attach render buffers in the `DEPTH24_STENCIL8_EXT` format to the framebuffer via **`FramebufferRenderbuffer`** with the ***`attach`*** argument set to `DEPTH_STENCIL_ATTACHMENT`. In this case, you must explicitly attach the render buffers to both `DEPTH_ATTACHMENT` and `STENCIL_ATTACHMENT` instead.

Depth tests always pass if the bound framebuffer does not have a depth buffer attached and depth tests are enabled. Similarly, stencil tests always pass if the bound framebuffer does not have a stencil buffer attached and stencil tests are enabled. If a depth buffer is not attached during gas density rendering, however, the depth buffer attached to the default framebuffer is used.

7.9.2 Specifications Particular to the Actual Hardware Environment

When the system API has initialized DMPGL, there are no render buffers are attached to the default framebuffer. The application must create and attach each of the render buffers.

In this environment, if the bound framebuffer does not have a depth buffer attached and depth tests are enabled, the previously enabled depth buffer is used. Similarly, if the bound framebuffer does not have a stencil buffer attached and stencil tests are enabled, the previously enabled stencil buffer is used. Behavior is undefined if a depth buffer or stencil buffer has never been enabled, as is the case immediately following DMPGL initialization. The same type of behavior applies when gas density rendering has been enabled.

A texture may also be attached as a depth buffer. See section 7.33 Depth Information Textures for details.

7.10 Uniform{1234}{if}(v)

DMPGL 2.0 error-checks reserved uniforms to see whether their configured values conform to the specifications. When a reserved uniform is set to a value that is outside of the range allowed by the specifications, an `INVALID_VALUE` error is generated. DMPGL 2.0 does not error-check the uniforms for the vertex shader and geometry shader.

7.11 GenerateMipmap

The DMPGL 2.0 implementation does not include calls to **`GenerateMipmap`**.

7.12 VertexAttribPointer

The DMPGL 2.0 implementation of this function does not allow FIXED or UNSIGNED_SHORT to be used for ***type***. If ***type*** is FLOAT or SHORT and ***pointer*** is not 4-byte aligned or 2-byte aligned, respectively, an INVALID_VALUE error is generated. ***normalize*** cannot be set to TRUE.

7.13 Clear

The DMPGL 2.0 implementation does not apply scissoring or buffer write mask settings when clearing a buffer with **Clear**. Also note that in the actual hardware environment, if you clear the stencil buffer you must also clear the depth buffer. In other words, if you specify STENCIL_BUFFER_BIT to the ***mask*** argument, you must specify DEPTH_BUFFER_BIT as well.

7.14 BlendFuncSeparate

The standard OpenGL ES specifications do not allow ***dstRGB*** or ***dstAlpha*** to be set to SRC_ALPHA_SATURATE, but DMPGL 2.0 allows this setting, although only with the drivers for the actual hardware. You cannot specify this setting in the PICA on Desktop (POD) environment.

7.15 Viewport

The DMPGL 2.0 implementation of this function does not support negative values for the ***x*** or ***y*** arguments. An INVALID_VALUE error is generated if a negative value is set.

7.16 Dithering

DMPGL 2.0 does not support any of the dithering-related specifications in standard OpenGL ES.

7.17 BufferData

The DMPGL 2.0 implementation of this function only supports STATIC_DRAW for ***usage***.

7.18 Vertex Buffers

When vertex data is rendered using a buffer object with **BindBuffer** or **BufferData**, some hardware restrictions apply to the arrangement of the vertex data set by **VertexAttribPointer**. If there is a conflict with these restrictions, an INVALID_OPERATION error is generated when **DrawArrays** or **DrawElements** is called. The vertex attributes and vertex indices used by a single draw operation cannot mix vertex data that uses buffer objects with vertex data that does not use buffer objects. An INVALID_OPERATION error is generated if they are mixed together. The following sections explain the restrictions on vertex data arrangement.

7.18.1 Restriction 1

Each vertex attribute must be aligned to a data length equal to the size of its own type. The following vertex data structure is explained as an example.

Code 7-5 Sample Vertex Data Structure (Padding for Alignment)

```
struct tagVertex
{
   GLbyte color[3];
   GLbyte padding1[1];    // The next attribute is 2-byte aligned
   GLshort       position[3];
   GLbyte padding2[2];    // The next attribute is 4-byte aligned
   GLfloat       normal[3];
};
```

In Code 7-5, `position` must be 2-byte aligned and `normal` must be 4-byte aligned. `padding1` and `padding2` are normally unnecessary because the compiler automatically adds padding.

7.18.2 Restriction 2

The stride of each vertex attribute must be a multiple of the size of the largest attribute type included in the same vertex attribute structure. The following vertex data structure is explained as an example.

Code 7-6 Sample Vertex Data Structure (Padding for Stride)

```
struct tagVertex
{
   GLfloat       position[3];
   GLbyte color[3];
   GLbyte padding[1];
} vertexArray[VERTEX_COUNT];
```

In Code 7-6, `sizeof(vertexArray[0])` must be a multiple of 4 bytes, the size of the largest type `GLfloat`. The `padding` is normally unnecessary because the compiler automatically adds padding.

7.18.3 Restriction 3

If you add more padding at the end of a vertex attribute than the minimum required amount satisfying conditions 1 and 2, the next vertex attribute must be placed not at the nearest 4-byte boundary following the end of the first vertex attribute but at one of the 4-byte boundaries following that.

The following vertex data structure is explained as an example.

Code 7-7 Sample Vertex Data Structure (Not Enough Extra Padding)

```
struct tagVertex
{
   GLshort       position[3];
```

```
    GLshort       extraPadding;
    GLshort       color[4];
};
```

In Code 7-7, `extraPadding` is unnecessary because restrictions 1 and 2 are satisfied even without it. In this case, restriction 3 is not satisfied because `color` is placed at the closest 4-byte boundary to the end of `position` (in other words, closest to `position[2]`). If this is rewritten as Code 7-8, it is acceptable because `color` is placed at the next 4-byte boundary after the one that is closest to the end of `position`.

Code 7-8 Sample Vertex Data Structure (Enough Extra Padding)

```
struct tagVertex
{
    GLshort       position[3];
    GLshort       extraPadding1;
    GLshort       extraPadding2[2];
    GLshort       color[4];
};
```

7.19 Getting the State

The OpenGL ES 2.0 specifications include functions to get the state, such as **`GetBooleanv`**, **`GetFloatv`**, **`GetIntegerv`**, and **`IsEnabled`**, but DMPGL 2.0 does not have support for getting all the states that can be obtained in OpenGl ES 2.0. The following states are not supported by DMPGL 2.0.

Table 7-3 Unsupported States

pname	**Description**
`LINE_WIDTH`	Unsupported
`SAMPLE_ALPHA_TO_COVERAGE`	Unsupported
`SAMPLE_COVERAGE`	Unsupported
`SAMPLE_COVERAGE_VALUE`	Unsupported
`SAMPLE_COVERAGE_INVERT`	Unsupported
`STENCIL_BACK_FUNC`	Unsupported
`STENCIL_BACK_VALUE_MASK`	Unsupported
`STENCIL_BACK_REF`	Unsupported
`STENCIL_BACK_FAIL`	Unsupported
`STENCIL_BACK_PASS_DEPTH_FAIL`	Unsupported
`STENCIL_BACK_PASS_DEPTH_PASS`	Unsupported

pname	Description
UNPACK_ALIGNMENT	Unsupported
PACK_ALIGNMENT	Unsupported
GENERATE_MIPMAP_HINT	Unsupported
ALIASED_POINT_SIZE_RANGE	Unsupported
ALIASED_LINE_WIDTH_RANGE	Unsupported
SAMPLE_BUFFERS	Unsupported
SAMPLES	Unsupported
MAX_VERTEX_UNIFORM_VECTORS	Unsupported For 4-element floating-point uniforms: 96 For Boolean uniforms: 15 For integer uniforms: 4
MAX_VARYING_VECTORS	Unsupported
MAX_VERTEX_TEXTURE_IMAGE_UNITS	Unsupported
MAX_TEXTURE_IMAGE_UNITS	Unsupported
MAX_FRAGMENT_UNIFORM_VECTORS	Unsupported

7.20 Hint

Hint exists in OpenGL ES 2.0 but not in DMPGL 2.0.

7.21 CreateShader and CreateProgram

Shader objects generated by **CreateShader** and program objects generated by **CreateProgram** share a namespace in OpenGL ES 2.0 but use different namespaces in DMPGL 2.0. Also, in DMPGL 2.0 program objects have a 13-bit namespace, so no more than 8191 names can be generated and exist at a single time. Although no more than 8191 program objects can exist at the same time, program objects can be created again after the **DeleteProgram** function is used to delete some program objects.

7.22 StencilFuncSeparate

StencilFuncSeparate exists in OpenGL ES 2.0 but not in DMPGL 2.0.

7.23 StencilMaskSeparate

StencilMaskSeparate exists in OpenGL ES 2.0 but not in DMPGL 2.0.

7.24 StencilOpSeparate

`StencilOpSeparate` exists in OpenGL ES 2.0 but not in DMPGL 2.0.

7.25 UniformMatrix

Although you cannot set ***`transpose`*** to `GL_TRUE` in OpenGL ES 2.0, you can do so in DMPGL 2.0. DMPGL 2.0 handles a vertex shader uniform as a `BOOL`, `INT_VEC3`, `FLOAT`, `FLOAT_VEC2`, `FLOAT_VEC3`, or `FLOAT_VEC4` value, or as an array of such values. **`UniformMatrix2fv`**, **`UniformMatrix3fv`**, and **`UniformMatrix4fv`** are respectively used to set `FLOAT_VEC2` array data with two or more elements, `FLOAT_VEC3` array data with three or more elements, and `FLOAT_VEC4` array data with four or more elements. These types of array data can also be set by **`Uniform{234}f(v)`**, so the following two function calls result in the same values.

- **`Uniform4fv`**`(location, 4, value);`
- **`UniformMatrix4fv`**`(location, 1, GL_TRUE, value);`

7.26 Location of Uniforms

The location obtained by **`GetUniformLocation`** is used to access that uniform's value with **`Uniform`** and **`GetUniform`**. By adding an offset value to ***`location`***, you can specify the elements to access in a uniform that is an array. For example, you can increment ***`location`*** by 1 to access the second element in an array uniform.

The location value is originally determined when **`LinkProgram`** is called, and the location value differs for each program object. The `Uniform` function generates an error if the location is associated with a program object that is not set as the current program. The `GetUniform` function generates an error if the location is associated with a program object other than ***`program`***. In DMPGL 2.0, there is only one case where `Uniform` and `GetUniform` do not generate these errors: for reserved fragment shader uniforms whose location value has been specified as `0xfff80000` using a bitwise OR.

7.27 PolygonOffset

DMPGL 2.0 does not support ***`factor`***. It does support ***`units`***, but since the vertex coordinate z value after vertex processing is implemented as a 24-bit floating-point number, the effect specified by ***`units`*** may not be fully realized depending on the polygon z value. If the z value is close to 1.0, the ***`units`*** value only has an effect if it is a multiple of 128. Consequently, you can be sure the ***`units`*** value will be effective if it is specified as a multiple of 128.

7.28 LinkProgram

In DMPGL 2.0, **`LinkProgram`** fails if more than 2048 uniforms for vertex shader objects and geometry shader objects are linked to a program object.

7.29 Functions to Set or Get Multiple Uniforms at Once

DMPGL 2.0 supports setting or getting multiple uniforms at once.

Code 7-9 UniformsDMP

```
void UniformsDMP(uint n, int* locations, sizei* counts, const uint* value);
```

This function sets multiple uniform values for the current program. ***n*** specifies the number of uniforms to set. ***locations*** specifies a pointer to an array holding the locations of the ***n*** uniforms. ***counts*** specifies a pointer to an array holding the number of elements in each of the ***n*** uniforms. This argument correlates to the ***count*** argument of the **Uniform{1234}fv** functions. In each element of ***counts***, specify the number of elements in the array to set (for array-type uniforms), or for non-array uniforms, specify 1. ***value*** specifies a pointer to an array holding the values to which the uniforms will be set. Each uniform might have a different number of data values, so the array indices for ***value*** might not match those for ***locations*** and ***counts***. With this function, you can set uniforms that have a mix of `float` and `int` data types. When specifying a `float` uniform, the corresponding ***value*** element holds the `float` data as 32 bits. This function performs no error checking. Specifying invalid values for any of the arguments will cause unstable operation.

The following function can be used to get multiple uniform vales that have been previously set.

Code 7-10 GetUniformsDMP

```
void GetUniformsDMP(
  uint program, uint n, int* locations, sizei* counts, uint* params);
```

program specifies the program object for the uniforms to get. ***n*** specifies the number of uniforms to get. ***locations*** specifies a pointer to an array holding the locations of the ***n*** uniforms. ***counts*** specifies a pointer to an array holding the number of elements in each of the ***n*** uniforms. In each element of ***counts***, specify the number of elements in the array to set (for array-type uniforms), or for non-array uniforms, specify 1. ***params*** specifies a pointer to an array that will hold the values obtained from the uniforms. Each uniform might have a different number of data values, so the array indices for ***params*** might not match those for ***locations*** and ***counts***. With this function, you can get uniforms that have a mix of `float` and `int` data types. When getting a `float` uniform, the corresponding ***params*** element holds the `float` data as 32 bits. This function performs no error checking. Specifying invalid values for any of the arguments will cause unstable operation.

7.30 DepthRange

In DMPGL 2.0, the relationship between window coordinate depth values Zw and normalized device coordinate depth values Zd is expressed by the following equation. This equation differs from the equation used in the OpenGL ES 1.1 specifications. (See section 2.15 Coordinate Systems.)

Equation 7-1 Relationship Between Zw and Zd

$$Zw = near - (far - near) \times Zd$$

When the reserved uniform variable `dmp_FragOperation.wScale` is set to some value other than 0.0, the depth values indicated by the ***near*** and ***far*** arguments to **DepthRange** are ignored. (See section 6.9 w Buffer.)

7.31 GetError

Use **GetError** to get the DMPGL error codes. In the actual hardware environment, the DMP-specific error `ERROR_COMMANDBUFFER_FULL_DMP` is generated when the 3D command buffer overflows or when it is unspecified. An `ERROR_COMMANDREQUEST_FULL_DMP` error is generated when the command request overflows or is not specified.

The DMPGL API may call the system API internally, in which case errors specific to the system API may be generated.

For details on command list objects, 3D command buffers, and command requests, see the *DMPGL 2.0 System API Specifications*.

7.32 Obtaining Object Addresses

In the actual hardware environment, it is possible to obtain the address of the data regions allocated for texture objects, vertex buffer objects, and render buffer objects.

- To get the address of a texture, bind the texture object and then call **GetTexParameteriv** with the ***pname*** argument set to `TEXTURE_DATA_ADDR_DMP`.
- To get the address of a vertex buffer, bind the vertex buffer object and then call **GetBufferParameteriv** with the ***pname*** argument set to `BUFFER_DATA_ADDR_DMP`.
- To get the address of a render buffer, call **GetRenderbufferParameteriv** with the ***pname*** argument set to `RENDERBUFFER_DATA_ADDR_DMP`.

In the PICA on Desktop environment, calling these functions with these arguments will generate an error. All addresses are obtained from a region that PICA accesses directly. The addresses in the "copy region" generated by the DMPGL driver will not be obtained.

7.33 Depth Information Textures

The DMPGL actual device environment provides a feature to use a texture storing a depth value. Texture data can also be created based on data from reading the rendered depth buffer using the **ReadPixels** function, but this section describes a different method.

7.33.1 Rendering Depth Information to Textures

With DMPGL, a texture can be attached to a framebuffer’s depth attachment point. In other words, depth information can be rendered to a texture. Only textures in certain formats can be attached to a depth attachment point. The format of the depth information depends on the format of the texture to attach. The table below gives the texture formats, correlating depth buffer formats, and the depth information stored in each texture component.

Table 7-4 Texture Formats and the Corresponding Depth Buffer Formats

Texture Format	Texture Type	Depth Buffer Format	Description of Each Component
RGBA RGBA_NATIVE_DMP	UNSIGNED_BYTE	DEPTH24_STENCIL8_EXT	R: 8-bit stencil value G: 24-bit depth value, bits [23:16] B: 24-bit depth value, bits [15:8] A: 24-bit depth value, bits [7:0]
RGB RGB_NATIVE_DMP	UNSIGNED_BYTE	DEPTH_COMPONENT24_OES	R: 24-bit depth value, bits [23:16] G: 24-bit depth value, bits [15:8] B: 24-bit depth value, bits [7:0]
HILO8_DMP HILO8_DMP_NATIVE_DMP	UNSIGNED_BYTE	DEPTH_COMPONENT16	R: 16-bit depth value, bits [15:8] G: 16-bit depth value, bits [7:0]

The textures of the formats listed above can be attached by calling the `FramebufferTexture2D` function, specifying `DEPTH_ATTACHMENT` for the ***`attachment`*** argument. The texture format `RGBA` values are used in a format that includes stencil values, so `FramebufferTexture2D` can also be called specifying `DEPTH_STENCIL_ATTACHMENT` for ***`attachment`***, but the effect is the same.

7.33.2 Copying to a Texture from a Depth Buffer

With DMPGL, you can use the current depth buffer by copying it to a texture. Call the **`CopyTexImage2D`** or **`CopyTexSubImage2D`** functions with the depth stencil copy feature enabled to copy a depth buffer to a texture. Call **`Enable`** specifying `DEPTH_STENCIL_COPY_DMP` to enable the depth stencil copy feature, and call **`Disable`** specifying `DEPTH_STENCIL_COPY_DMP` to disable the feature. Calling **`IsEnabled`** specifying `DEPTH_STENCIL_COPY_DMP` will return `TRUE` if the depth stencil copy feature is enabled, and `FALSE` if disabled.

The format of the texture being copied to is determined by the format of the current depth buffer. Format conversion at time of copying is not supported. The correlation between the format of the current depth buffer and the format of the texture being copied to is the same as shown in Table 7-4. For instance, if the format of the current depth buffer is `DEPTH24_STENCIL8_EXT`, then the ***`internalformat`*** argument to the **`CopyTexImage2D`** function must be either `RGBA` or `RGBA_NATIVE_DMP`. (Both `RGBA` and `RGBA_NATIVE_DMP` have the same effect.) When the texture format specified for ***`internalformat`*** does not correspond to the current depth buffer format, the call to **`CopyTexImage2D`** will generate an `INVALID_ENUM` error. When **`CopyTexSubImage2D`** is called on a texture object that has a texture format that does not correspond to the current depth buffer format, the function call will generate an `INVALID_OPERATION` error.

Appendix A DMPGL 2.0 Functions

The following functions exist in OpenGL ES 2.0 but not in DMPGL 2.0.

- `CompileShader`
- `CompressedTexSubImage2D`
- `GenerateMipmap`
- `GetProgramInfoLog`
- `GetShaderInfoLog`
- `GetShaderPrecisionFormat`
- `GetShaderSource`
- `Hint`
- `LineWidth`
- `PixelStorei`
- `ReleaseShaderCompiler`
- `SampleCoverage`
- `ShaderSource`
- `StencilFuncSeparate`
- `StencilMaskSeparate`
- `StencilOpSeparate`
- `TexSubImage2D`

The following table gives a list of functions that are supported in DMPGL 2.0, but with limited features.

Table A-1 Feature-Limited Functions

Function Name	Implementation
`BufferData`	Only `STATIC_DRAW` is supported for ***`usage`***.
`Clear`	Scissoring is not applied.
`CopyTexImage2D`	***`x`*** and ***`y`*** must be multiples of 8. ***`width`*** and ***`height`*** must be powers of 2.
`CopyTexSubImage2D`	***`x`***, ***`y`***, ***`width`***, and ***`height`*** must be multiples of 8.
`CreateProgram`	See section 7.21 CreateShader and CreateProgram. This uses a 13-bit namespace.
`DrawArrays`	`POINT`, `LINES`, `LINE_STRIP`, and `LINE_LOOP` cannot be specified.
`DrawElements`	`POINT`, `LINES`, `LINE_STRIP`, and `LINE_LOOP` cannot be specified.
`PolygonOffset`	***`factor`*** is not supported.
`RenderbufferStorage`	`STENCIL_INDEX8` cannot be used in the actual hardware environment but `DEPTH_COMPONENT24_STENCIL_INDEX8_DMP` is supported.
`ShaderBinary`	See Chapter 3 DMP Shaders.
`TexImage2D`	***`width`*** and ***`height`*** must be powers of 2.

Function Name	Implementation
`VertexAttribPointer`	***`normalize`*** cannot be used. `FIXED` and `UNSIGNED_SHORT` cannot be used for ***`type`***.
`Viewport`	***`x`*** and ***`y`*** must be 0 or greater.

Appendix B Uniform State Tables

This appendix lists uniform constants that are reserved by program objects in DMPGL 2.0. *Uniform Name* is the name specified to the **`GetUniformLocation`** function. *Type* is the type of the corresponding uniform and conforms to the descriptions in the GL ES Shading Language Version 1.0 specifications. *Values & Initial Value* indicates the values that can be specified for the corresponding uniform as well as the default values.

Table B-1 Texture Environment State Uniforms (i = 0, 1, 2)

Uniform Name	Type	Values & Initial Value	Description
`dmp_TexEnv[i] .combineRgb`	`int`	• `REPLACE` **(default)** • `MODULATE` • `ADD` • `ADD_SIGNED` • `INTERPOLATE` • `SUBTRACT` • `DOT3_RGB` • `DOT3_RGBA` • `ADD_MULT_DMP` • `MULT_ADD_DMP`	Texture combiner functions for colors
`dmp_TexEnv[i] .combineAlpha`	`int`	• `REPLACE` **(default)** • `MODULATE` • `ADD` • `ADD_SIGNED` • `INTERPOLATE` • `SUBTRACT` • `DOT3_RGBA` • `ADD_MULT_DMP` • `MULT_ADD_DMP`	Texture combiner functions for alpha values
`dmp_TexEnv[0] .srcRgb`	`ivec3`	• `TEXTURE0` • `TEXTURE1` • `TEXTURE2` • `TEXTURE3` • `CONSTANT` **(default)** • `PRIMARY_COLOR` • `FRAGMENT_PRIMARY_COLOR_DMP` • `FRAGMENT_SECONDARY_COLOR_DMP`	Color input to texture combiner 0

Uniform Name	Type	Values & Initial Value	Description
dmp_TexEnv[i] .srcRgb (i is nonzero)	ivec3	• TEXTURE0 • TEXTURE1 • TEXTURE2 • TEXTURE3 • CONSTANT • PRIMARY_COLOR • PREVIOUS **(default)** • PREVIOUS_BUFFER_DMP • FRAGMENT_PRIMARY_COLOR_DMP • FRAGMENT_SECONDARY_COLOR_DMP	Color input to the texture combiners (i is nonzero). There must be at least one input of PREVIOUS or CONSTANT.
dmp_TexEnv[0] .srcAlpha	ivec3	• TEXTURE0 • TEXTURE1 • TEXTURE2 • TEXTURE3 • CONSTANT **(default)** • PRIMARY_COLOR • FRAGMENT_PRIMARY_COLOR_DMP • FRAGMENT_SECONDARY_COLOR_DMP	Alpha input to texture combiner 0
dmp_TexEnv[i] .srcAlpha (i is nonzero)	ivec3	• TEXTURE0 • TEXTURE1 • TEXTURE2 • TEXTURE3 • CONSTANT • PRIMARY_COLOR • PREVIOUS **(default)** • PREVIOUS_BUFFER_DMP • FRAGMENT_PRIMARY_COLOR_DMP • FRAGMENT_SECONDARY_COLOR_DMP	Alpha input to the texture combiners (i is nonzero) There must be at least one input of PREVIOUS or CONSTANT.
dmp_TexEnv[i] .operandRgb	ivec3	• SRC_COLOR **(default)** • ONE_MINUS_SRC_COLOR • SRC_ALPHA • ONE_MINUS_SRC_ALPHA • SRC_R_DMP • ONE_MINUS_SRC_R_DMP • SRC_G_DMP • ONE_MINUS_SRC_G_DMP • SRC_B_DMP • ONE_MINUS_SRC_B_DMP	Color operands for the texture combiners.

Uniform Name	Type	Values & Initial Value	Description
dmp_TexEnv[i] .operandAlpha	ivec3	• SRC_ALPHA **(default)** • ONE_MINUS_SRC_ALPHA • SRC_R_DMP • ONE_MINUS_SRC_R_DMP • SRC_G_DMP • ONE_MINUS_SRC_G_DMP • SRC_B_DMP • ONE_MINUS_SRC_B_DMP	Alpha operands for the texture combiners.
dmp_TexEnv[i] .bufferInput (i = 1,2,3,4)	ivec2	• PREVIOUS • PREVIOUS_BUFFER_DMP **(default)**	Combiner buffer input.
dmp_TexEnv[i] .scaleRgb	float	• 1.0 **(default)** • 2.0 • 4.0	Scaling factor for the output colors from the texture combiners.
dmp_TexEnv[i] .scaleAlpha	float	• 1.0 **(default)** • 2.0 • 4.0	Scaling factors for the output alpha from the texture combiners.
dmp_TexEnv[i] .constRgba	vec4	Each component is in the range [0.0,1.0] Default: (0,0,0,0)	The CONSTANT input values to the texture combiners.
dmp_TexEnv[0] .bufferColor	vec4	Each component is in the range [0.0,1.0] Default: (0,0,0,0)	Combiner buffer color.

Table B-2 Fragment Lighting State Uniforms (i = 0, 1, 2, 3, 4, 5, 6, 7)

Uniform Name	Type	Values & Initial Value	Description
dmp_FragmentLighting .enabled	bool	• TRUE • FALSE **(default)**	Enables or disables fragment lighting
dmp_FragmentLighting .ambient	vec4	Each component is in the range [0.0,1.0] (0.2, 0.2, 0.2, 1.0) by default	Global ambient components
dmp_FragmentMaterial .sampler{D0,D1,RR,RG,RB,FR}	int	[0,31] Undefined by default	Lookup table numbers for the various factors in the lighting equation
dmp_FragmentMaterial .emission	vec4	Each component is in the range [0.0,1.0] (0.0, 0.0, 0.0, 1.0) by default	The material emission components
dmp_FragmentMaterial .ambient	vec4	Each component is in the range [0.0,1.0] (0.2, 0.2, 0.2, 1.0) by default	The material ambient components
dmp_FragmentMaterial .diffuse	vec4	Each component is in the range [0.0,1.0] (0.8, 0.8, 0.8, 1.0) by default	The material diffuse components
dmp_FragmentMaterial .specular0	vec4	Each component is in the range [0.0,1.0] (0.0, 0.0, 0.0, 1.0) by default	The first specular components of a material

Uniform Name	Type	Values & Initial Value	Description
`dmp_FragmentMaterial .specular1`	`vec4`	Each component is in the range [0.0,1.0] (0.0, 0.0, 0.0, 1.0) by default	The second specular components of a material
`dmp_FragmentLightSource[i] .enabled`	`bool`	• `TRUE` • `FALSE` **(default)**	Enables or disables the light at index `i`
`dmp_FragmentLightSource[i] .ambient`	`vec4`	Each component is in the range [0.0,1.0] (0.0, 0.0, 0.0, 0.0) by default	Ambient components of the light at index `i`
`dmp_FragmentLightSource[i] .diffuse`	`vec4`	Each component is in the range [0.0,1.0] (0.0, 0.0, 0.0, 0.0) by default However, only light 0 is (1.0, 1.0, 1.0, 1.0)	Diffuse components of the light at index `i`
`dmp_FragmentLightSource[i] .specular0`	`vec4`	Each component is in the range [0.0,1.0] (0.0, 0.0, 0.0, 0.0) by default However, only light 0 is (1.0, 1.0, 1.0, 1.0)	First specular components of the light at index `i`
`dmp_FragmentLightSource[i] .specular1`	`vec4`	Each component is in the range [0.0,1.0] (0.0, 0.0, 0.0, 0.0) by default	Second specular components of the light at index `i`
`dmp_FragmentLightSource[i] .position`	`vec4`	Unspecified range (0.0, 0.0, 1.0, 0.0) by default	The light direction vector or light position vector of the light at index `i`
`dmp_FragmentLightSource[i] .spotDirection`	`vec3`	Unspecified range (0.0, 0.0, -1.0) by default	Spotlight direction vector of the light at index `i`
`dmp_FragmentLightSource[i] .shadowed`	`bool`	• `TRUE` • `FALSE` **(default)**	Enables or disables shadows for the light at index `i`
`dmp_FragmentLightSource[i] .geomFactor0`	`bool`	• `TRUE` • `FALSE` **(default)**	Enables or disables the first geometry factor for the light at index `i`
`dmp_FragmentLightSource[i] .geomFactor1`	`bool`	• `TRUE` • `FALSE` **(default)**	Enables or disables the second geometry factor for the light at index `i`
`dmp_FragmentLightSource[i] .twoSideDiffuse`	`bool`	• `TRUE` • `FALSE` **(default)**	Enables or disables two-sided diffuse lighting
`dmp_FragmentLightSource[i] .spotEnabled`	`bool`	• `TRUE` • `FALSE` **(default)**	Enables or disables spotlights
`dmp_FragmentLightSource[i] .distanceAttenuationEnabled`	`bool`	• `TRUE` • `FALSE` **(default)**	Enables or disables distance attenuation

Uniform Name	Type	Values & Initial Value	Description
dmp_FragmentLightSource[i] .distanceAttenuationBias	float	Unspecified range 0.0 by default	Bias on input values for distance attenuation lookup tables
dmp_FragmentLightSource[i] .distanceAttenuationScale	float	Unspecified range 1.0 by default	Scaling factor on input values for distance attenuation lookup tables
dmp_FragmentLightSource[i] .samplerSP	int	[0,31] Undefined by default	Spotlight lookup table number
dmp_FragmentLightSource[i] .samplerDA	int	[0,31] Undefined by default	Distance attenuation lookup table number
dmp_LightEnv .absLutInput{D0,D1,RR,RG,RB ,SP,FR}	bool	• TRUE • FALSE **(default)**	Enables or disables use of absolute-value input values to the lookup tables for the various factors
dmp_LightEnv .lutInput{D0,D1,SP}	int	• LIGHT_ENV_NH_DMP **(default)** • LIGHT_ENV_VH_DMP • LIGHT_ENV_NV_DMP • LIGHT_ENV_LN_DMP • LIGHT_ENV_SP_DMP • LIGHT_ENV_CP_DMP	Lookup table input values for the various factors
dmp_LightEnv .lutInput{RR,RG,RB,FR}	int	• LIGHT_ENV_NH_DMP **(default)** • LIGHT_ENV_VH_DMP • LIGHT_ENV_NV_DMP • LIGHT_ENV_LN_DMP	Lookup table input values for the various factors
dmp_LightEnv .lutScale{D0,D1,RR,RG,RB,SP ,FR}	float	• 0.25 • 0.50 • 1.0 **(default)** • 2.0 • 4.0 • 8.0	Scaling factor on the lookup table output values for the various factors
dmp_LightEnv .shadowSelector	int	TEXTURE{0,1,2,3} TEXTURE0 by default	Texture unit to use for shadows
dmp_LightEnv .bumpSelector	int	TEXTURE{0,1,2,3} TEXTURE0 by default	Texture unit to use for bump mapping
dmp_LightEnv .bumpMode	int	• LIGHT_ENV_BUMP_NOT_USED_DMP **(default)** • LIGHT_ENV_BUMP_AS_BUMP_DMP • LIGHT_ENV_BUMP_AS_TANG_DMP	Perturbation mode for normals or tangents

Uniform Name	Type	Values & Initial Value	Description
dmp_LightEnv .bumpRenorm	bool	• TRUE • FALSE **(default)**	Enables or disables recalculation of a normal vector's third component
dmp_LightEnv .config	int	• LIGHT_ENV_LAYER_CONFIG0_DMP **(default)** • LIGHT_ENV_LAYER_CONFIG1_DMP • LIGHT_ENV_LAYER_CONFIG2_DMP • LIGHT_ENV_LAYER_CONFIG3_DMP • LIGHT_ENV_LAYER_CONFIG4_DMP • LIGHT_ENV_LAYER_CONFIG5_DMP • LIGHT_ENV_LAYER_CONFIG6_DMP • LIGHT_ENV_LAYER_CONFIG7_DMP	Per-factor configuration
dmp_LightEnv .invertShadow	bool	• TRUE • FALSE **(default)**	Enables or disables inversion (flipping) of the shadow attenuation item
dmp_LightEnv .shadowPrimary	bool	• TRUE • FALSE **(default)**	Enables or disables shadow contribution to the primary color
dmp_LightEnv .shadowSecondary	bool	• TRUE • FALSE **(default)**	Enables or disables shadow contribution to the secondary color
dmp_LightEnv .shadowAlpha	bool	• TRUE • FALSE **(default)**	Enables or disables shadow contribution to the alpha value
dmp_LightEnv .fresnelSelector	int	• LIGHT_ENV_NO_FRESNEL_DMP **(default)** • LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP • LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP • LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP	Fresnel factor output mode
dmp_LightEnv .clampHighlights	bool	• TRUE **(default)** • FALSE	Enables or disables clamping on the specular color
dmp_LightEnv .lutEnabledD0	bool	• TRUE • FALSE **(default)**	Enables or disables the application of lookup table output on the first distribution
dmp_LightEnv .lutEnabledD1	bool	• TRUE • FALSE **(default)**	Enables or disables the application of lookup table output on the second distribution

Uniform Name	Type	Values & Initial Value	Description
dmp_LightEnv .lutEnabledRefl	bool	• TRUE **(default)** • FALSE	Enables or disables the application of lookup table output on the reflection factor

Table B-3 Texture State Uniforms

Uniform Name	Type	Values & Initial Value	Description
dmp_Texture[0] .perspectiveShadow	bool	• TRUE **(default)** • FALSE	Enables or disables perspective projections with texture coordinate generation for shadow texture access
dmp_Texture[0] .shadowZScale	float	A value larger than 0.0 Undefined by default	Scaling factor on the evaluated derivatives of the depth values in the screen space of the light source's coordinate system
dmp_Texture[0] .shadowZBias	float	Unspecified range 0.0 by default	The bias value to subtract from the distance to the light source
dmp_Texture[0] .samplerType	int	• FALSE **(default)** • TEXTURE_2D • TEXTURE_CUBE_MAP • TEXTURE_SHADOW_2D_DMP • TEXTURE_SHADOW_CUBE_DMP • TEXTURE_PROJECTION_DMP	Sampling mode for texture unit 0
dmp_Texture[{1,2}] .samplerType	int	• FALSE **(default)** • TEXTURE_2D	Sampling mode for texture units 1 and 2
dmp_Texture[3] .samplerType	int	• FALSE **(default)** • TEXTURE_PROCEDURAL_DMP	Sampling mode for texture unit 3
dmp_Texture[2] .texcoord	int	• TEXTURE1 • TEXTURE2 **(default)**	Selects the input coordinates to texture unit 2
dmp_Texture[3] .texcoord	int	• TEXTURE0 **(default)** • TEXTURE1 • TEXTURE2	Selects the input coordinates to texture unit 3

Table B-4 Procedural Texture State Uniforms

Uniform Name	Type	Values & Initial Value	Description
dmp_Texture[3] .ptRgbMap	int	• PROCTEX_U_DMP **(default)** • PROCTEX_V_DMP • PROCTEX_U2_DMP • PROCTEX_V2_DMP • PROCTEX_ADD_DMP • PROCTEX_ADD2_DMP • PROCTEX_ADDSQRT2_DMP • PROCTEX_MIN_DMP • PROCTEX_MAX_DMP • PROCTEX_RMAX_DMP	Selects the function used for procedural calculations
dmp_Texture[3] .ptAlphaMap	int	• PROCTEX_U_DMP **(default)** • PROCTEX_V_DMP • PROCTEX_U2_DMP • PROCTEX_V2_DMP • PROCTEX_ADD_DMP • PROCTEX_ADD2_DMP • PROCTEX_ADDSQRT2_DMP • PROCTEX_MIN_DMP • PROCTEX_MAX_DMP • PROCTEX_RMAX_DMP	Selects the function used for procedural calculations
dmp_Texture[3] .ptAlphaSeparate	bool	• TRUE • FALSE **(default)**	Selects between shared-coordinate mode and separate-coordinate mode
dmp_Texture[3] .ptClampU	int	• SYMMETRICAL_REPEAT_DMP • MIRRORED_REPEAT • PULSE_DMP • CLAMP_TO_EDGE **(default)** • CLAMP_TO_ZERO_DMP	Clamping method
dmp_Texture[3] .ptClampV	int	• SYMMETRICAL_REPEAT_DMP • MIRRORED_REPEAT • PULSE_DMP • CLAMP_TO_EDGE **(default)** • CLAMP_TO_ZERO_DMP	Clamping method
dmp_Texture[3] .ptShiftU	int	• EVEN_DMP • ODD_DMP • NONE_DMP **(default)**	Coordinate shift method
dmp_Texture[3] .ptShiftV	int	• EVEN_DMP • ODD_DMP • NONE_DMP **(default)**	Coordinate shift method

Uniform Name	Type	Values & Initial Value	Description
dmp_Texture[3] .ptMinFilter	int	• NEAREST • LINEAR **(default)** • NEAREST_MIPMAP_NEAREST • NEAREST_MIPMAP_LINEAR • LINEAR_MIPMAP_NEAREST • LINEAR_MIPMAP_LINEAR	MinFilter method
dmp_Texture[3] .ptTexWidth	int	[0,128] 0 by default	Lookup table width
dmp_Texture[3] .ptTexOffset	int	[0,128] 0 by default	Color lookup table offset
dmp_Texture[3] .ptTexBias	float	0.0 or greater 0.5 by default	LOD bias
dmp_Texture[3] .ptNoiseEnable	bool	• TRUE • FALSE **(default)**	Enables or disables noise
dmp_Texture[3] .ptNoiseU	vec3	Unspecified range (0.0, 0.0, 0.0) by default	The frequency, amplitude, and phase of noise
dmp_Texture[3] .ptNoiseV	vec3	Unspecified range (0.0, 0.0, 0.0) by default	The frequency, amplitude, and phase of noise
dmp_Texture[3] .ptSamplerRgbMap	int	[0,31] Undefined by default	Lookup table number for the color function lookup table used by procedural calculations
dmp_Texture[3] .ptSamplerAlphaMap	int	[0,31] Undefined by default	Lookup table number for the alpha function lookup table used by procedural calculations
dmp_Texture[3] .ptSamplerNoiseMap	int	[0,31] Undefined by default	Lookup table number for noise modulation
dmp_Texture[3] .ptSamplerR	int	[0,31] Undefined by default	R component lookup table number
dmp_Texture[3] .ptSamplerG	int	[0,31] Undefined by default	G component lookup table number
dmp_Texture[3] .ptSamplerB	int	[0,31] Undefined by default	B component lookup table number
dmp_Texture[3] .ptSamplerA	int	[0,31] Undefined by default	A component lookup table number

Table B-5 Gas State Uniforms

Uniform Name	Type	Values & Initial Value	Description
`dmp_Gas .lightXY`	`vec3`	Each component is in the range [0.0, 1.0] (0.0, 0.0, 0.0) by default	Attenuation for the minimum intensity, maximum intensity, and density, all controlling planar shading
`dmp_Gas .lightZ`	`vec4`	Each component is in the range [0.0, 1.0] (0.0, 0.0, 0.0, 0.0) by default	Attenuation for the minimum intensity, maximum intensity, and density, all controlling view shading; also, the light direction along the z axis in eye coordinates
`dmp_Gas .deltaZ`	`float`	Unrestricted range 10.0 by default	Scaling factor given to the calculated distance in the view-vector direction while rendering the accumulation pass
`dmp_Gas .autoAcc`	`bool`	• `TRUE` **(default)** • `FALSE`	Enables or disables automatic calculation of the maximum density in the additive blending results
`dmp_Gas .accMax`	`float`	0.0 or greater 1.0 by default	Inverse of the density when the maximum density is given in the additive blending results
`dmp_Gas .shadingDensitySrc`	`int`	• `GAS_PLAIN_DENSITY_DMP` **(default)** • `GAS_DEPTH_DENSITY_DMP`	Selects the density used for shading
`dmp_Gas .colorLutInput`	`int`	• `GAS_DENSITY_DMP` • `GAS_LIGHT_FACTOR_DMP` **(default)**	Selects either the density or shading intensity for input to the shading lookup table
`dmp_Gas .samplerT{R,G,B}`	`int`	[0,31] Undefined by default	Shading lookup tables
`dmp_Gas .attenuation`	`float`	0.0 or greater 1.0 by default	Density attenuation coefficients for fog table input values

Table B-6 Fog State Uniforms

Uniform Name	Type	Values & Initial Value	Description
`dmp_Fog.mode`	`int`	• `FALSE` **(default)** • `FOG` • `GAS_DMP`	Fog mode
`dmp_Fog.color`	`vec3`	Each component is in the range [0.0, 1.0] (0, 0, 0) by default	Fog color
`dmp_Fog.zFlip`	`bool`	• `TRUE` • `FALSE` **(default)**	Enables or disables inversion (flipping) of fog input values

Uniform Name	Type	Values & Initial Value	Description
dmp_Fog.sampler	int	[0,31] Undefined by default	Fog lookup table number

Table B-7 Per-Fragment Operations State Uniforms

Uniform Name	Type	Values & Initial Value	Description
dmp_FragOperation .enableClippingPlane	bool	• TRUE • FALSE **(default)**	Enables or disables clipping by a clipping plane
dmp_FragOperation .clippingPlane	vec4	Unrestricted range (0.0, 0.0, 0.0, 0.0) by default	Clipping plane
dmp_FragOperation .enableAlphaTest	bool	• TRUE • FALSE **(default)**	Enables or disables alpha tests
dmp_FragOperation .alphaRefValue	float	[0.0, 1.0] 0.0 by default	Alpha test reference value
dmp_FragOperation .alphaTestFunc	int	• NEVER • ALWAYS **(default)** • LESS • LEQUAL • EQUAL • GEQUAL • GREATER • NOTEQUAL	Alpha test comparison function
dmp_FragOperation .mode	int	• FRAGOP_MODE_GL_DMP **(default)** • FRAGOP_MODE_SHADOW_DMP • FRAGOP_MODE_GAS_ACC_DMP	Per-fragment operations mode
dmp_FragOperation .wScale	float	Unspecified range 0.0 by default	This enables or disables the W buffer and is the scaling factor applied to depth values
dmp_FragOperation .penumbraScale	float	Unspecified range 0.0 by default	Scaling factor applied when calculating the "hardness" of a penumbra
dmp_FragOperation .penumbraBias	float	Unspecified range 1.0 by default	Bias applied when calculating the "hardness" of a penumbra

Table B-8 Point State Uniforms

Uniform Name	Type	Value & Initial Value	Description
dmp_Point .viewport	vec2	Unspecified range Undefined by default	Specifies the viewport (1 / viewport.width, 1 / viewport.height)

Uniform Name	Type	Value & Initial Value	Description
dmp_Point .distanceAttenuation	bool	• TRUE • FALSE Undefined by default	Enables or disables DMP-specific distance attenuation

Table B-9 Line State Uniforms

Uniform Name	Type	Value & Initial Value	Description
dmp_Line.width	vec4	Unspecified range Undefined by default	Specifies the line width (viewport width / line width, viewport height / line width, viewport width×viewport height, 2.f / line width)

Table B-10 Silhouette State Uniforms

Uniform Name	Type	Values & Initial Value	Description
dmp_Silhouette .width	vec2	Each component is 0.0 or greater Undefined by default	The separate scaling factors for the *x* and *y* components of normal vectors used during silhouette rectangle generation
dmp_Silhouette .scaleByW	bool	• TRUE • FALSE Undefined by default	Enables or disables multiplication of the vertex's w component with the x and y components of the normal vector during silhouette rectangle generation
dmp_Silhouette .color	vec4	Each component is in the range [0.0, 1.0] Undefined by default	Silhouette color
dmp_Silhouette .frontFaceCCW	bool	• TRUE • FALSE Undefined by default	Front-facing setting
dmp_Silhouette .acceptEmptyTriangles	bool	• TRUE • FALSE Undefined by default	Enables or disables silhouette edge generation on open edges
dmp_Silhouette .openEdgeColor	vec4	Each component is in the range [0.0, 1.0] Undefined by default	The silhouette color of an open edge
dmp_Silhouette .openEdgeWidth	vec4	Unspecified range Undefined by default	Width of the silhouette rectangle on an open edge (viewport width / silhouette width, viewport height / silhouette width, viewport width×viewport height, 2.f / silhouette width)

Uniform Name	Type	Values & Initial Value	Description
`dmp_Silhouette .openEdgeDepthBias`	`float`	Unspecified range Undefined by default	Bias applied to the depth values of the silhouette rectangle on an open edge
`dmp_Silhouette .openEdgeWidthScaleByW`	`bool`	• TRUE • FALSE Undefined by default	Enables or disables multiplication of the vertex's *w* component with the width of a silhouette rectangle on an open edge
`dmp_Silhouette .openEdgeDepthBiasScaleByW`	`bool`	• TRUE • FALSE Undefined by default	Enables or disables multiplication of the vertex's *w* component with the depth value bias of a silhouette rectangle on an open edge

Table B-11 Subdivision State Uniforms

Uniform Name	Type	Values & Initial Value	Description
`dmp_Subdivision .level`	`float`	• 0 • 1 • 2 Undefined by default	Subdivision level
`dmp_Subdivision .fragmentLightingEnabled`	`bool`	• TRUE • FALSE Undefined by default	Whether to use the quaternion vertex attribute

Table B-12 Particle System State Uniforms

Uniform Name	Type	Values & Initial Value	Description
`dmp_PartSys .color`	`mat4`	Each component is in the range [0.0, 1.0] Undefined by default	Colors of the four control points
`dmp_PartSys .aspect`	`mat4`	The particle size is 1.0 or greater The alpha color is in the range [0.0, 1.0] No range is specified for anything else Undefined by default	The following values for the four control points: (particle size, texture coordinate rotation angle, texture coordinate scaling factor, alpha color)
`dmp_PartSys .time`	`float`	Unspecified range Undefined by default	The current particle system time
`dmp_PartSys .speed`	`float`	A value larger than 0.0 Undefined by default	Particle movement speed
`dmp_PartSys .countMax`	`float`	0.0 or greater Undefined by default	One less than the generated particle count

Uniform Name	Type	Values & Initial Value	Description
`dmp_PartSys .randSeed`	`vec4`	Unspecified range Undefined by default	The seed for randomizing the current time and the *x,y,z* coordinates of the control points
`dmp_PartSys .randomCore`	`vec4`	Unspecified range Undefined by default	$(a, b, m, 1/m)$ in the random-number function $X_{N+1} = (aX_N + b) mod\ m$
`dmp_PartSys .distanceAttenuation`	`vec3`	Each component is 0.0 or greater Undefined by default	The distance attenuation coefficients a, b, c for particles
`dmp_PartSys .viewport`	`vec2`	Unspecified range Undefined by default	Specifies the viewport (1 / viewport width, 1 / viewport height)
`dmp_PartSys .pointSize`	`vec2`	Each component is 0.0 or greater Undefined by default	Specifies the maximum and minimum particle size (maximum size, minimum size)