CTR
DMPGL 2.0 Programming Guide

2012/11/01

Version 3.5

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

© 2009–2012 Nintendo
CONFIDENTIAL

CTR-06-0004-002-I
Released: January 14, 2013

Confidential

These coded instructions, statements, and computer programs contain proprietary information of Nintendo and/or its licensed developers and are protected by national and international copyright laws. They may not be disclosed to third parties or copied or duplicated in any form, in whole or in part, without the prior written consent of Nintendo.

Table of Contents

Code

© 2009–2012 Nintendo
CONFIDENTIAL

Tables

Figures

© 2009–2012 Nintendo
CONFIDENTIAL

1 About This Document

This document is a programming guide that explains basic programming using the DMPGL 2.0 API. The content of this document is aimed at people with knowledge about basic C programming and 3D graphics. For an overview of the DMPGL 2.0 pipeline and detailed specifications, see the *DMPGL 2.0 Specifications*.

1.1 Examples and Notation

This document abbreviates the **`glGetUniformLocation(`**`"uniform_name"`**`)`** syntax as `LOC(“uniform_name”)`.

2 Vertex Shaders

This chapter describes how to program vertex shaders.

2.1 Overview

Vertex shaders are a feature for taking vertex attribute data provided by the application and processing it in certain ways, such as transforming its coordinate system or shading it. Shader programs are written in an assembly language whose specifications are proprietary to DMPGL2.0 (this language will be referred to as "shader assembly language" or "shader assembly code" throughout the rest of the documentation). The programs are assembled and linked, and the binary file that is generated is loaded by the application and then used.

2.2 Loading Shaders

Shaders are loaded using **`glShaderBinary`**. For the third argument (*`binaryformat`*), specify `GL_PLATFORM_BINARY_DMP`. For the fourth argument (*`binary`*), specify a pointer to the data for a linked shader binary file. For the first argument (*`count`*), specify the number of shader objects within the linked shader assembly code that contain the `main` label. For the second argument (*`shader`*), specify a pointer to an array that stores the shader objects that were generated by **`glCreateShader`**. This binds the shader assembly code to the various shader objects, in the order that was specified to the linker. The link order for the shader assembly code can also be verified using the map files that the linker generates. See the Map Files chapter of the *Vertex Shader Reference Manual* for more details.

2.3 Attaching Shaders

To actually render images using the loaded shader objects, you must use **`glAttachShader`** to attach the shader objects to the program object that was generated using **`glCreateProgram`**, then link the program object using **`glLinkProgram`**. Use **`glUseProgram`** to apply successfully linked program objects to the vertex processing pipeline. The **`glUseProgram`** function also allows you to switch between multiple linked program objects. When you attach a different shader object to a program object, you must then relink it using **`glLinkProgram`**.

2.4 Inputting Vertex Data

The indices, names, and input registers of each vertex attribute are bound together into vertex data, which is then fed to a vertex shader as input. The application binds the vertex attribute index and the input data name by calling **`glBindAttribLocation`**, specifying the vertex attribute index as the second argument (*`index`*), and specifying the input data name (*`name`*) as the third argument. The shader assembly code then binds the input data name and the input registers by defining `#pragma bind_symbol` with the input data name as the first argument and the input registers as the second and third arguments. When specifying the input data name in shader assembly code, specify the same input data name specified by the application, then follow it with the components of the vertex attribute.

The application uses either **glVertexAttrib{1234}{f}v** or **glVertexAttribPointer** to enter data for bound vertex attribute numbers. The data can then be read from the input registers by the shader assembly code.

Code 2-1: Shader Assembly Code Sample

```
#pragma bind_symbol(AttribPosition.xyzw, v0, v0)
```

This binds the data name "`AttribPosition`" to the `xyzw` components of input register `v0`.

Code 2-2: Application Code Sample

```
glBindAttribLocation(program, 0, ”AttribPosition”);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, pointer);
```

This binds vertex attribute index 0 to the data that has the name `AttribPosition` and inputs the four vertex attribute components.

Vertex attribute indices and input register numbers are not related in any way, and do not have to be the same.

2.5 Outputting Vertex Data

To output data from vertex shaders after vertex processing and make the data available to subsequent pipeline stages, write the data to the output registers that have been mapped to the output vertex attributes. In your shader assembly code, define `#pragma output_map`, specifying an output vertex attribute name as the first argument, and specifying an output register for the second argument.

Code 2-3: Shader Assembly Code Sample

```
#pragma output_map(position, o0)

mov          o0,    v0
```

This maps output register `o0` to vertex coordinates. By writing data to `o0`, the data will be output as vertex coordinates to subsequent stages of the pipeline.

Vertex shaders finish processing and output their data when data has been written to all the output registers that were mapped using `#pragma output_map`. (An `end` instruction must be issued immediately as soon as all of the registers have been written to.) The output data can be overwritten as long as this is done before data is written to all the output registers. With DMPGL 2.0, only reserved shaders can be used as the fragment shader, so the attributes that can be output from vertex shaders are predetermined.

2.6 Configuring Uniforms

The application can set the various registers used in shader assembly code. The registers set from the application are all bound to names by `#pragma bind_symbol`, and the application recognizes these names as uniform names. Uniforms are set to values that are shared by all vertex operations that are run using a single call to **`glDrawElements`** or **`glDrawArrays`**. Due to the specifications of the assembler, only the following three types of uniforms can be set.

- Floating-point registers set by **`glUniform{1234}{f}{v}`**
- Boolean registers set by **`glUniform1i`**
- Integer registers set by **`glUniform3iv`**

2.6.1 Floating-Point Constant Registers

Floating-point constant registers store constants that are required for calculations; for example, constant transformation matrices such as the modelview or projection matrix, or the light-source colors and coordinates used in vertex lighting. These registers are set using **`glUniform{1234}{f}{v}`**. Their values are undefined if the application doesn't set them. Floating-point constant registers whose values are defined by the `def` instruction in shader assembly code cannot be bound to names by `#pragma bind_symbol`, so their values cannot be set from the application.

Code 2-4: Shader Assembly Code Sample

```
#pragma bind_symbol(ModelViewMatrix, c0, c3)

m4x4         r0,    v0,    c0
```

This binds the registers from `c0` through `c3` (each register having four components) to the name `ModelViewMatrix`. The example above illustrates how to calculate a modelview transformation, if vertex coordinates have been loaded into input register `v0`.

Code 2-5: Application Code Sample

```
GLfloat  modelview[16];

modelview[0] = 1.f; modelview[1] = 0.f; ...

glUniformMatrix4fv(LOC(”ModelViewMatrix”), 1, GL_FALSE, modelview);
```

This configures the content of `ModelViewMatrix`.

2.6.2 Boolean Registers

Boolean registers are used for branch instructions. The values of Boolean registers are set using **`glUniform1i`**. Their values are undefined if the application doesn't set them. Boolean registers whose values are defined by the `defb` instruction in shader assembly code cannot be bound to names by `#pragma bind_symbol`, so their values cannot be set from the application.

Code 2-6: Shader Assembly Code Sample

```
#pragma bind_symbol(LightingEnable, b0, b0)

ifb   b0
// Lighting calculation
...
endif
```

This binds `b0` to the name `LightingEnable`. The example above illustrates how to use the `b0` register as a switch to enable or disable lighting.

Code 2-7: Application Code Sample

```
glUniform1i(LOC(“LightingEnable”), GL_TRUE);
```

This sets `LightingEnable` to `GL_TRUE`.

2.6.3 Integer Registers

Integer registers are used for loop instructions. These registers are set using **`glUniform3iv`**. Three values are set: (1) the number of iterations minus one, (2) the initial value of the loop counter register, and (3) the amount by which to increase or decrease the loop counter register for each iteration. Integer register values are undefined if the application doesn't set them. Integer registers whose values are defined by the `defi` instruction in shader assembly code cannot be bound to names by `#pragma bind_symbol`, so their values cannot be set from the application.

Code 2-8: Shader Assembly Code Sample

```
#pragma bind_symbol(LightSourceCount, i0, i0)
#define LIGHT0_AMBIENT           c0
#define LIGHT0_SPECULAR          c1
#define LIGHT0_DIFFUSE           c2
#define LIGHT1_AMBIENT           c3
#define LIGHT1_SPECULAR          c4
...

loop  i0
// Calculation for each light source
mov          r0,    c0[aL]       // get ambient
mov          r1,    c0[aL + 1]   // get specular
mov          r2,    c0[aL + 2]   // get diffuse
...
endloop
```

This binds `i0` to the name `LightSourceCount`. The example above illustrates how to use a loop instruction to perform an operation once for every light source.

© 2009–2012 Nintendo
CONFIDENTIAL

Code 2-9: Application Code Sample

```
int loop_setting[3];
loop_setting[0] = LIGHT_SOURCE_COUNT – 1;
loop_setting[1] = 0;    // Set the initial value of the loop counter register to 0
loop_setting[2] = 3;    // Set the step size of the loop counter register to +3
glUniform3iv(LOC(“LightSourceCount”), 1, loop_setting);
```

In the shader assembly code sample, three settings for each light source are stored in registers, starting from register `c0`. A loop instruction then performs operations on each light source. The application sets the loop counter register to make sure the registers that hold the settings for each light source are accessed properly.

2.7 Precautions When Using Shader Programs

2.7.1 The Z-Component of the Output Vertex Coordinates

While standard OpenGL ES implementations clip the z-component of the clip coordinates output from the vertex shaders to the range [$-w_c$, w_c], DMPGL 2.0 clips the z-component to the range [0, $-w_c$]. (Note that the sign is inverted.) This means that if your application uses a projection matrix that is compatible with OpenGL ES, you must convert from the range [$-w_c$, w_c], to [0, $-w_c$] during projection transformation. There are two possible ways of doing this required operation.

Method #1: Change the projection matrix from the application.

This method makes the following changes to the projection matrix before loading it into a uniform.

```
GLfloat projection[16];
projection[2] = (projection[2] + projection[3]) × (-0.5f);
projection[6] = (projection[6] + projection[7]) × (-0.5f);
projection[10] = (projection[10] + projection[11]) × (-0.5f);
projection[14] = (projection[14] + projection[15]) × (-0.5f);
```

Method #2: Change the projection transformation operation using shader assembly code.

This method uses shader assembly code to perform the following projection transformation.

```
#pragma output_map(position, o0)
#pragma bind_symbol(attrib_position, v0)
#pragma bind_symbol(modelview, c0, c3)
#pragma bind_symbol(projection, c4, c7)
def   c8,    -0.5, -0.5, -0.5, -0.5

// Modelview transformation
dp4   r0.x,  v0,    c0
dp4   r0.y,  v0,    c1
dp4   r0.z,  v0,    c2
dp4   r0.w,  v0,    c3
// Projection transformation
dp4   o0.x,  r0,    c4
dp4   o0.y,  r0,    c5
mov   r1,    c6                  // Use （c7 + c8）×(-0.5f) as the
add   r1,    r1,    c7           // third row of the projection matrix
mul   r1,    r1,    c8           //
dp4   o0.z,  r0,    r1
dp4   o0.w,  r0,    c7
```

Between the two methods shown above, Method #1 involves more calculation on the application side, but reduces the load on the vertex shaders more than Method #2 does.

2.7.2 Normalization of Vertex Attributes

With PICA, there is no hardware support for the fourth argument to **`glVertexAttribPointer`**, which specifies whether to normalize values. This argument's setting is not applied, so normalization must be performed explicitly by the vertex shaders.

© 2009–2012 Nintendo
CONFIDENTIAL

3 Geometry Shaders

This chapter describes how to program geometry shaders.

3.1 Overview

Geometry shaders operate on the vertex attribute data that is output by the vertex shaders. They process it on the per-primitive level, and can output an arbitrary number of vertices. It is not possible to use a user-defined shader program written in the shader assembly language as a geometry shader. It is only possible to link user-defined vertex shader assembler objects to the intermediate assembler objects (OBJ files) that are distributed in pre-assembled form. These pre-assembled intermediate assembler objects are known as "reserved geometry shaders."

3.2 Loading Shaders

Geometry shaders, like vertex shaders, are loaded using **`glShaderBinary`**. However, to load geometry shaders, the second argument (*`shader`*) of **`glShaderBinary`** must be passed a shader object that was generated by specifying `GL_GEOMETRY_SHADER_DMP` to **`glCreateShader`**. With DMPGL 2.0, it is not possible to use a geometry shader object all by itself. You must always load a binary that also links a vertex shader so that geometry shader objects are used together with vertex shader objects.

```
GLuint shader[2];
shader[0] = glCreateShader(GL_VERTEX_SHADER);
shader[1] = glCreateShader(GL_GEOMETRY_SHADER_DMP);
glShaderBinary(2, shader, GL_PLATFORM_BINARY_DMP, binary, size);
// The 'binary' argument must be linked to both vertex shader assembly code
// and geometry shader assembly code.
```

3.3 Attaching Shaders

To actually render images using the loaded shader objects, you must use **`glAttachShader`** to attach the shader objects to the program objects that were generated using **`glCreateProgram`**, and then link the program objects using **`glLinkProgram`**. To attach a geometry shader, you must also simultaneously attach the shader objects of the vertex shaders you'll be using. Geometry shaders cannot operate on their own without at least one vertex shader.

3.4 Inputting Vertex Data

The data output from vertex shaders is the input data for geometry shaders. Each reserved geometry shader has fixed rules for input data, such as what vertex attributes it requires, what other vertex attributes are optional, and the input order of attributes. All of this data must be output correctly by the vertex shaders. Data is input to geometry shaders in order by register number, starting from the lowest-numbered register output by the vertex shaders. The vertex attributes to output are defined by **`#pragma output_map`**, but if a vertex attribute is used only by the geometry shaders and not handled by subsequent stages of the rendering pipeline, its attribute name is defined only as *`generic`*. As an example, the settings for point shaders and line shaders are shown below. For other types of geometry shaders, see the relevant sections within this document.

3.4.1 Point Shaders

When using a point shader, output the vertex attribute data in the following order: (1) vertex coordinates (from the vertex shaders), (2) point size, (3) other attributes. The vertex shaders must define the following data and output the data to the corresponding output register.

```
#pragma output_map(position, o0)    // Vertex coordinates
#pragma output_map(generic, o1)     // Point size
#pragma output_map(color, o2)       // Other attributes (vertex color, etc.)
```

3.4.2 Line Shaders

When using a line shader, output the vertex attribute data in the following order: (1) vertex coordinates (from the vertex shaders), (2) other attributes. The vertex shaders must define the following data and output the data to the corresponding output register.

```
#pragma output_map(position, o0)    // Vertex coordinates
#pragma output_map(color, o1)       // Other attributes (vertex color, etc.)
```

3.5 Configuring Uniforms

When using geometry shaders, all of the uniforms must be set. Refer to the specifications of each geometry shader and make sure all related uniforms are configured.

3.6 Rendering Using Geometry Shaders

When using geometry shaders, set the *`mode`* argument of the **`glDrawElements`** and **`glDrawArrays`** functions to `GL_GEOMETRY_PRIMITIVE_DMP`.

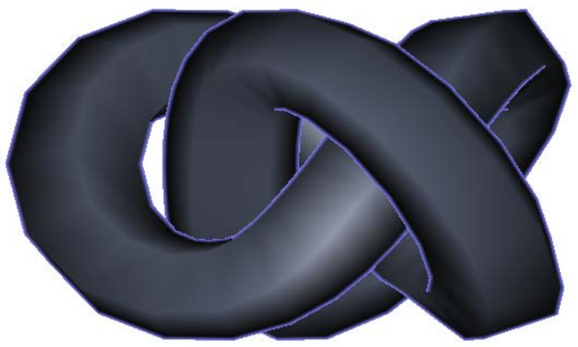
4 Silhouettes

This chapter explains how to program silhouettes that use the silhouette shader functionality provided by DMPGL 2.0.

4.1 Overview

Silhouettes generate silhouette lines along the boundaries of objects. The diagram below shows an example of a silhouette, where the silhouette lines are the areas rendered in blue that run around the boundaries of the object. Silhouette line generation is done by the pipeline's geometry processor, but doing so requires that primitives called "Triangles with Neighborhoods" (abbreviated as TWN throughout the rest of this document) be put into the pipeline.

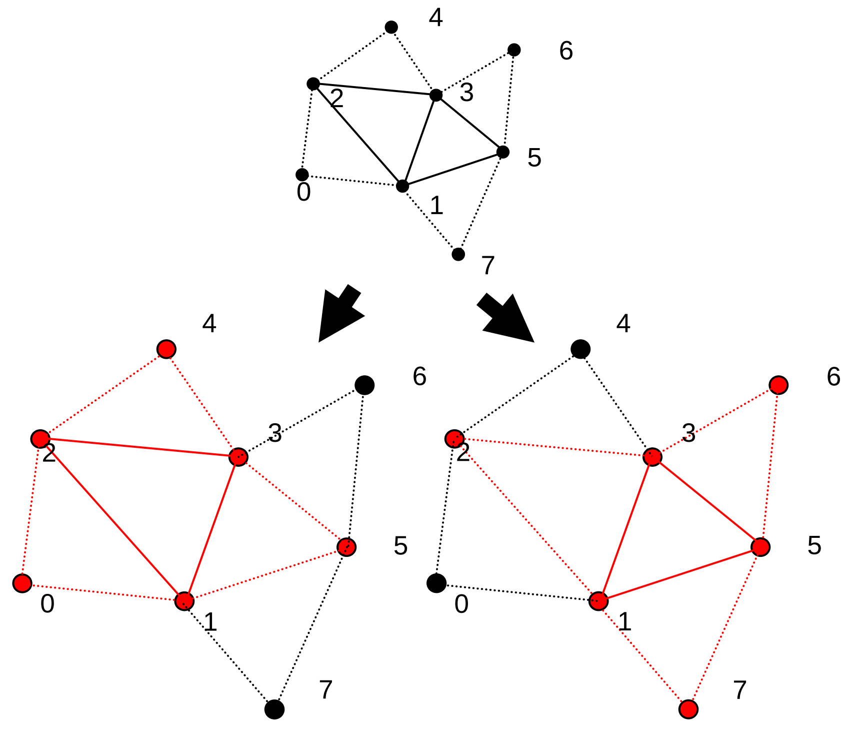
Figure 4-1: Example of Rendering Silhouette Lines



4.2 Triangles with Neighborhoods (TWNs)

TWNs consist of a target triangle (which we call the *center triangle*) and the three adjacent triangles that share edges with that triangle.

Figure 4-2: Examples of TWNs (Triangles with Neighborhoods)



If the triangle with vertices (1, 3, 2) is regarded as the center triangle, that triangle along with the triangles (0, 1, 2), (5, 3, 1), and (4, 2, 3) will form a single TWN made of four total triangles. If the triangle with vertices (5, 3, 1) is regarded as the center triangle, that triangle along with the triangles (7, 5, 1), (6, 3, 5), and (2, 1, 3) will form another TWN made of four total triangles.

TWN primitives are used to detect the silhouettes of center triangles. By creating objects out of TWN primitives, silhouette lines can be rendered for those objects.

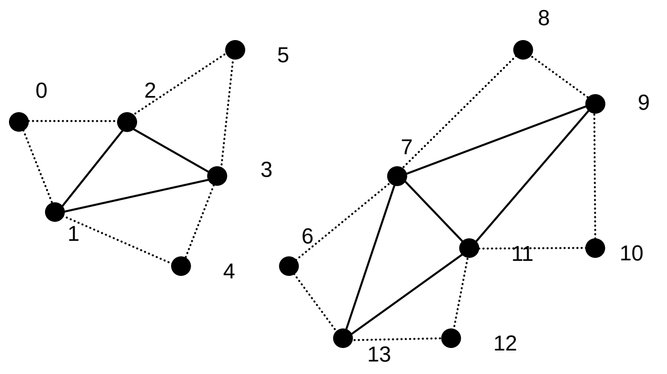
4.3 TWN Primitive Indices

TWNs can only be used with **`glDrawElements`**. Specify `GL_GEOMETRY_PRIMITIVE_DMP` for the *`mode`* argument. Rendering TWNs with **`glDrawArrays`** is not supported. In relation to TWNs, there are two types of reserved geometry shaders (`DMP_silhouetteTriangle.obj` and `DMP_silhouetteStrip.obj`), and the indices of each type are created in different ways. We assume that indices for both types are created based on the indices for `GL_TRIANGLES` and `GL_TRIANGLE_STRIP` that are used during normal triangle rendering.

4.3.1 DMP_silhouetteTriangle Indices

When using the `DMP_silhouetteTriangle.obj` reserved geometry shader, create indices using the method shown below.

Figure 4-3: Example for `silhouetteTriangle`



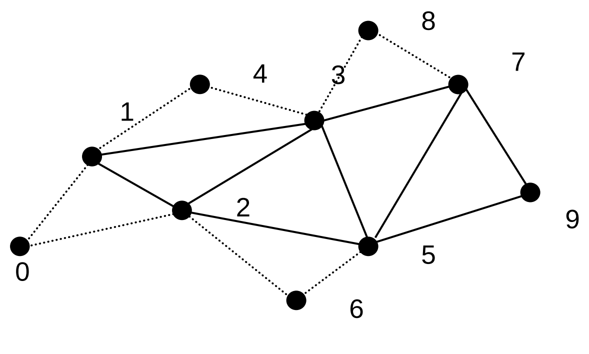
1. Specify the first and second vertices of the center triangle in an order that ensures that the center triangle is front-facing, then specify the remaining vertex of the adjacent triangle that shares the edge formed by the first two vertices.
2. Specify the third vertex of the center triangle, and then specify the remaining vertex of the adjacent triangle that shares the edge formed by the first and third vertices of the center triangle.
3. Finally, specify the remaining vertex of the adjacent triangle that shares the edge formed by the second and third vertices of the center triangle.

In the sample shown in the figure above, assuming **`glFrontFace`** is set to `GL_CCW`, the indices "3, 2, 5, 1, 4, 0" form one TWN primitive. The indices "11, 7, 9, 13, 12, 6" indicate the next primitive, and the indices "11, 9, 10, 7, 13, 8" indicate the one after that.

4.3.2 DMP_silhouetteStrip Indices

When using the `DMP_silhouetteStrip.obj` reserved geometry shader, create indices using the method shown below.

Figure 4-4: Example for silhouetteStrip



1. Specify the first TWN primitive.
   - o Specify the vertices of this first TWN primitive in the same order that was described for `DMP_silhouetteTriangle.obj` earlier.

2. Specify the (n+1)th TWN primitive. The center triangle of the (n+1)th primitive is the last of the adjacent triangles specified for the nth TWN primitive. The first, second, and third vertices of the center triangle in the (n+1)th TWN primitive are, respectively, the second and third vertices of the center triangle in the nth TWN primitive, followed by the very last vertex of the nth TWN primitive.
    - Specify the remaining vertex of the adjacent triangle that shares the edge formed by the first and third vertices of the center triangle in the (n+1)th TWN primitive.
    - Specify the remaining vertex of the adjacent triangle that shares the edge formed by the second and third vertices of the center triangle in the (n+1)th TWN primitive.
3. Repeat step 2 as many times as necessary.
4. There is a special method used to specify the final (Nth) TWN primitive when ending a strip.
    - Specify the remaining vertex of the adjacent triangle that shares the edge formed by the first and third vertices of the center triangle in the Nth TWN primitive.
    - Specify the third vertex of the center triangle in the Nth TWN primitive.
    - Specify the remaining vertex of the adjacent triangle that shares the edge formed by the second and third vertices of the center triangle in the Nth TWN primitive.
5. To specify a new strip array, go back to step 1. At this point in the procedure, if the first center triangle faces the opposite direction from the **`glFrontFace`** setting, specify its first vertex twice in a row to indicate this.

In the example shown in Figure 4-4, assuming that **`glFrontFace`** is set to `GL_CCW`, the TWN strip format is "1, 2, 0, 3, 4, 5, 6, 7, 8, 9, …". The first TWN primitive is defined by the "1, 2, 0, 3, 4, 5" portion, and the subsequent "6, 7" portion defines a new TWN primitive. This continues in "8, 9" and so on. To make the center triangle (3, 5, 7) the last one in the strip, specify (1, 2, 0, 3, 4, 5, 6, 7, 8, 7, 9). If you were to specify "1, 1, 2, 0, 3, 4, 5, 6, 7, 8, 7, 9", the center triangles would face in the opposite direction.

4.4 Inputting Vertex Data

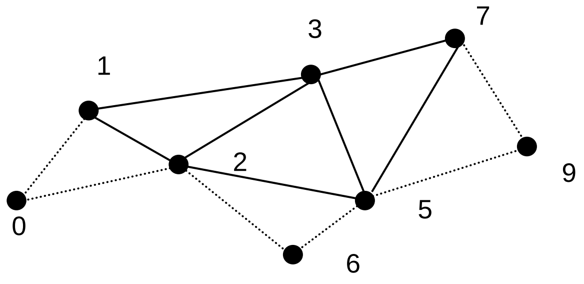
Input vertex attribute data to the silhouette shader in the following order: (1) vertex coordinates (from the vertex shader), (2) vertex color, (3) normal vector. Define the following in vertex shader assembly code and output the data to the corresponding output registers. The x- and y-components of the normal vector must be output as normalized data.

```
#pragma output_map(position, o0)    // Vertex coordinates
#pragma output_map(color, o1)       // Vertex color
#pragma output_map(generic, o2)     // Normal vector
```

4.5 Open Edges

If one of the edges of a center triangle doesn't share vertices with any other triangles, it is referred to as an "open edge." Adjacent triangles to open edges are specified as though they are folded over onto the center triangles. In the figure below, for example, if the 1-3 edge of the TWN whose center triangle is defined by vertices (1, 2, 3) is an open edge, the silhouette indices of that TWN are "1, 2, 0, 3, 2, 5."

Figure 4-5: Silhouette Indices of an Open Edge

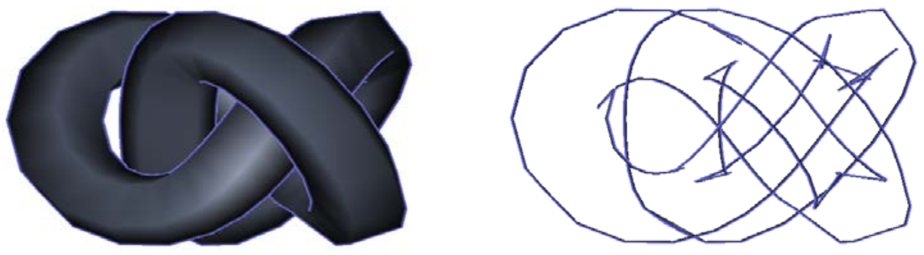


You can choose to either always render or never render silhouettes on open edges. This is set by calling **`Uniform1i`** and specifying `TRUE` or `FALSE` for the *`value`* argument of the `dmp_Silhouette.acceptEmptyTriangles` reserved uniform. If `TRUE` is specified, silhouette edges for this type of polygon will always be generated. If `FALSE` is specified, they will never be generated. Open-edge silhouettes are unlike regular silhouettes in that they don't use normal vectors and are rendered instead using the same method as line primitives. As a result, the appearance of open-edge silhouettes depends on the angles of the center triangle. You'll need to adjust the silhouette width and its bias along the Z-axis using the reserved uniforms `dmp_Silhouette.openEdgeWidth` and `dmp_Silhouette.openEdgeDepthBias`.

4.6 Examples of Rendered Silhouette Lines

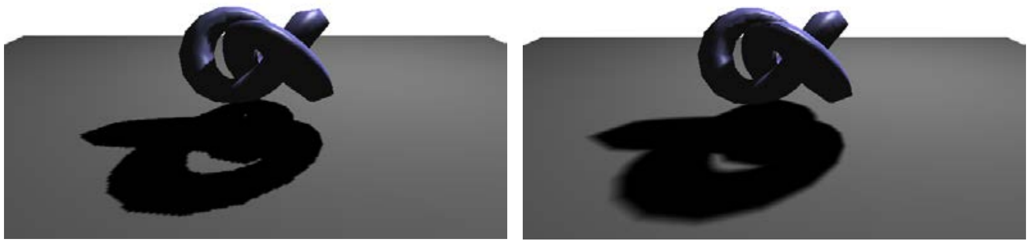
The figures below are examples of rendering using silhouettes.

Figure 4-6: Examples of Edges Rendered Using Silhouette Quads



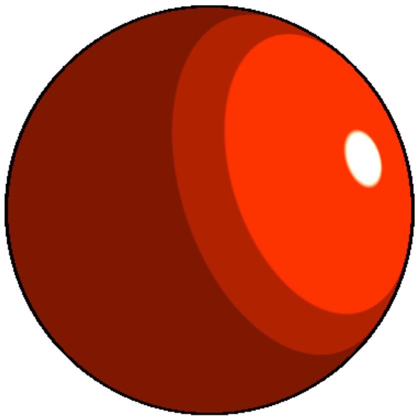
In Figure 4-6, the model on the left is a normal model rendered along with its silhouette lines. In the model on the right, only the silhouette lines are rendered.

Figure 4-7: Example of Soft Shadowing Using Silhouettes



(The image on the left has silhouettes disabled; the image on the right has silhouettes enabled.) For details, see Chapter 14 DMP Shadows.

Figure 4-8: Example of Edges Rendered Using Silhouette Quads



In Figure 4-8, we've used the silhouette feature to add a black edge to a sphere that has already been toon-shaded using fragment lighting.

4.7 Precautions When Rendering Silhouettes

Silhouette lines are only generated when the center triangles are front-facing. Make sure the specified index order of your center triangles and your **`glFrontFace`** setting reflect your intended behavior. When using `DMP_silhouetteTriangle.obj`, silhouettes are not generated if the center triangle is degenerate. When using `DMP_silhouetteStrip.obj`, degenerate polygons are interpreted as the last item in the strip array.

The vertex data used to render silhouettes requires the use of a vertex buffer. Call the **`glBindBuffer`** or **`glBufferData`** functions to use a vertex buffer.

© 2009–2012 Nintendo
CONFIDENTIAL

4.8 Silhouette-Rendering Performance

Performance differs between `DMP_silhouetteTriangle.obj` and `DMP_silhouetteStrip.obj`, even when rendering the same model. With `DMP_silhouetteTriangle.obj`, each TWN is specified with six vertices. With `DMP_silhouetteStrip.obj`, after the initial TWN is specified, only two vertices are required to specify each additional TWN. You can expect that `DMP_silhouetteStrip.obj` will give you twice or better the performance of `DMP_silhouetteTriangle.obj`.

5 Subdivision

This chapter explains how to program polygon subdivision using DMPGL 2.0's subdivision shader functionality.

5.1 Overview

Subdivision refers to a technique for subdividing surfaces into groups of vertices called *subdivision patches* (shown in the figures below) to make the surface appear smoother.

Figure 5-1: Example of Catmull-Clark Subdivision

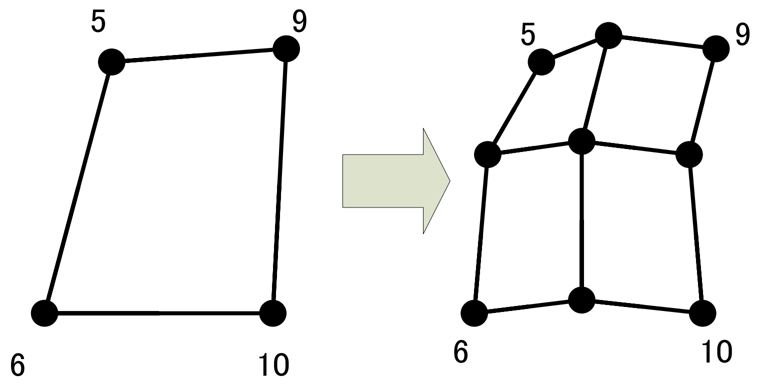
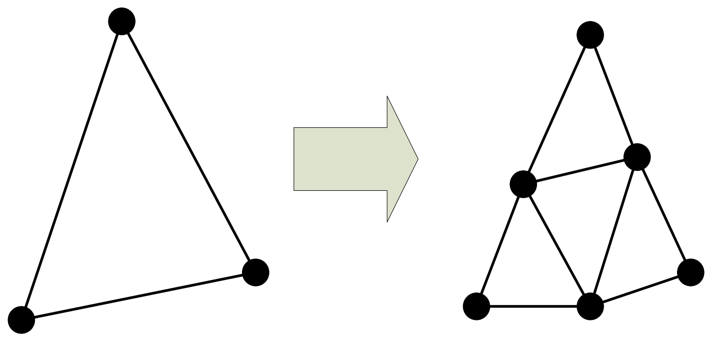


Figure 5-2: Example of Loop Subdivision



In DMPGL 2.0, subdivision patches are made from the vertices in the triangles or quadrilaterals to be subdivided, as well as from the vertices around their perimeter.

Figure 5-3: Example of Catmull-Clark Subdivision Patches

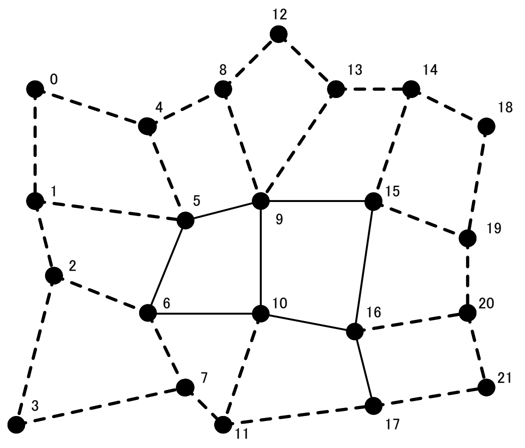
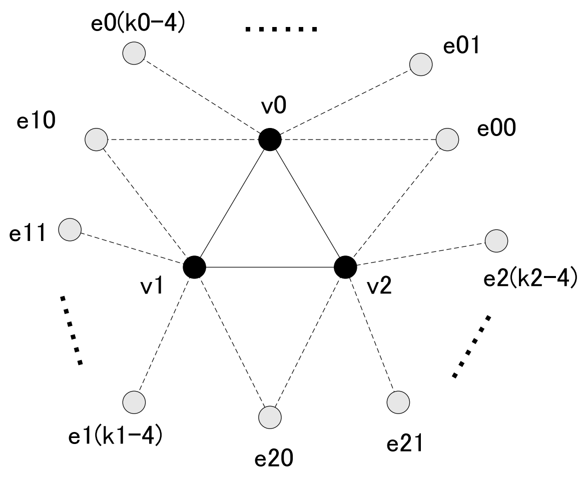


Figure 5-4: Example of a Loop Subdivision Patch

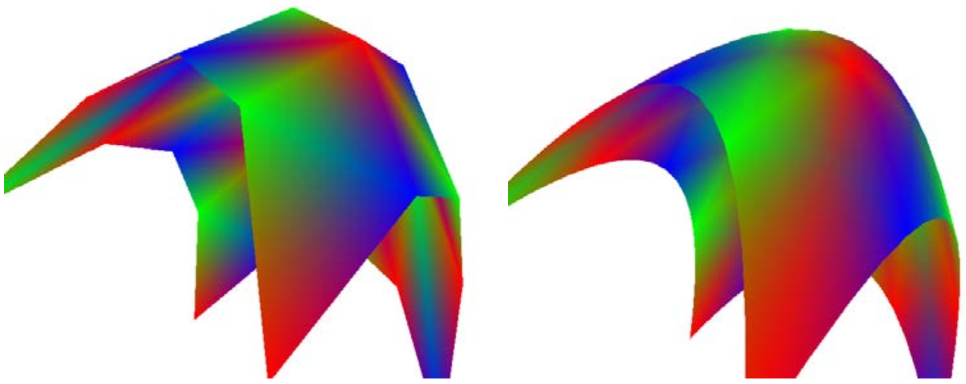


Subdivision refers to the technique of breaking polygons down starting from vertices on their perimeters. There are two types of subdivision shaders implemented by DMPGL 2.0: Catmull-Clark subdivision and Loop subdivision. These methods subdivide and generate polygons in the geometry shader according to their respective algorithms. This creates more detailed polygons than the polygons that were fed into the pipeline and sends them on to the rasterization pipeline.

5.2 Rendered Examples of Subdivision

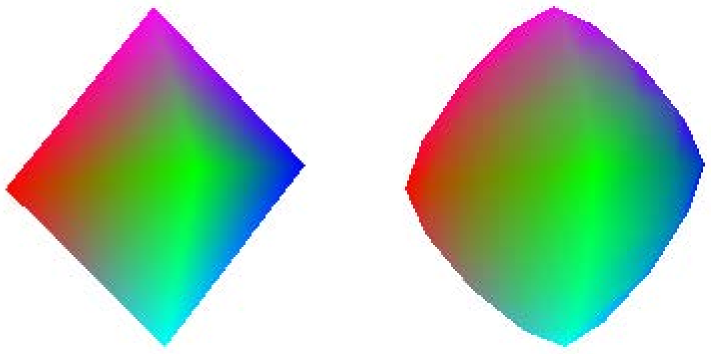
The figure below is a sample rendering that uses subdivision.

Figure 5-5: Subdivision Rendering Example 1



In Figure 5-5, the image on the left shows the result of rendering the original mesh. The image on the right shows the rendered result after applying Catmull-Clark subdivision. Note that the same vertex data was used to create both images. You can see that on the right the polygons are more finely subdivided, and the curved surfaces appear smoother than they do in the original.

Figure 5-6: Subdivision Rendering Example 2



In Figure 5-6, the image on the left shows the original mesh, and the image on the right shows the rendered result after applying Loop subdivision.

5.3 Catmull-Clark Subdivision

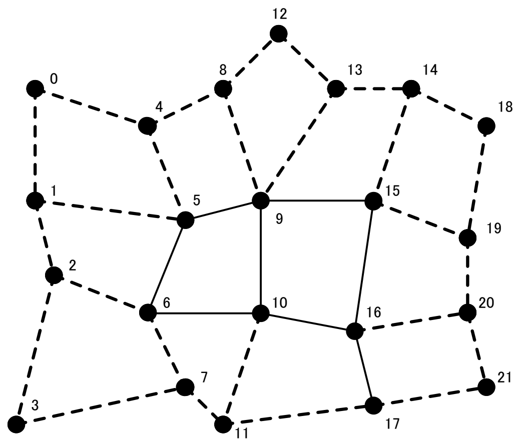
This section describes Catmull-Clark subdivision. Within this section, the word "subdivision" always indicates Catmull-Clark subdivision.

© 2009–2012 Nintendo
CONFIDENTIAL

5.3.1 Definition of Catmull-Clark Subdivision Patches

A *Catmull-Clark subdivision patch* is a set of polygons consisting only of quadrilaterals. A patch is the group of quads consisting of the quadrilateral to subdivide (the central quad) and all vertices that share an edge with the vertices in the central quad.

Figure 5-7: Example of Catmull-Clark Subdivision Patches



If (5, 6, 10, 9) is taken to be the central quad, the group of quads contained within the area defined by the vertices (5, 6, 10, 9, 8, 4, 0, 1, 2, 3, 7, 11, 17, 16, 15, 14, 13, 12) constitutes a patch. If (9, 10, 16, 15) is taken to be the central quad, the group of quads contained within the area defined by the vertices (9, 10, 16, 15, 8, 4, 5, 6, 7, 11, 17, 21, 20, 19, 18, 14, 13, 12) constitutes another patch.

When subdivision is performed, the central quad within each patch is broken down into smaller, finer polygons that form a smoother surface.

5.3.2 Restrictions on Catmull-Clark Subdivision Patches

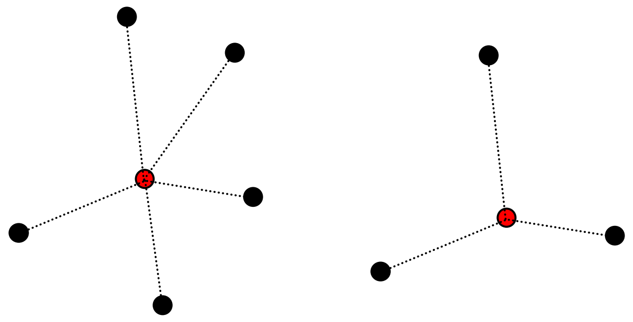
The following restrictions apply to subdivision patches when using Catmull-Clark subdivision.

- They must be made from quads.
- Central quads are allowed to contain irregular vertices, but each central quad must not contain more than a single irregular vertex.
- The valence of each irregular vertex must be between 3 and 12. (*Valence* refers to the number of edges that touch a given vertex.)

Here, *irregular vertices* are defined as vertices with valences other than 4.

The figure below shows irregular vertices with valences of 5 and 3.

Figure 5-8: Examples of Irregular Vertices



In the example on the left in Figure 5-8, the vertex marked in red has a valence of five (it has five adjacent edges). In the example on the right, the valence is only three (there are only three adjacent edges). Both of these vertices are irregular vertices.

In the representative example of subdivision patches shown in Figure 5-7, vertex 9 in the central quad (5, 6, 10, 9) is an irregular vertex.

5.3.3 Specifying Indices for Catmull-Clark Subdivision Patches

Subdivision patches can only be used with **`glDrawElements`**. Rendering with **`glDrawArrays`** is not supported. A vertex buffer must also be used (call a function like **`glBindBuffer`** or **`glBufferData`** to set up the buffer). The stored data that includes the vertex indices must be used only via the vertex buffer.

Before specifying the vertex indices of each subdivision patch, you must specify the overall patch size (the number of vertices that make up the patch). For example, if the central quad contains an irregular vertex of valence 3, the patch size is 14. If the central quad contains an irregular vertex of valence 5, the patch size is 18. If the central quad doesn't contain any irregular vertices, the patch size is 16. The patch size can be calculated as follows: 2×(valence of irregular vertex)+8. Specify this size at the beginning of the patch.

The order used to specify indices is shown below.

1. If there is an irregular vertex among the vertices that make up the central quad, specify that point first, as the *starting point*. If the central quad doesn't contain any irregular vertices, any of its vertices can serve as the starting point. Make sure to comply with the **`glFrontFace`** setting so that the central quad is front-facing.
2. Take the fifth vertex as the one that forms an edge with the first vertex (the starting point) and forms a surface with the first and second vertices.
3. After that, specify the remaining vertices (6th, 7th, 8th, ...), in the same direction (clockwise or counterclockwise) in which you specified the vertices in the central quad.

In the example of subdivision patches shown in Figure 5-7, the indices are as follows:

In the first patch, vertex 9 is an irregular vertex with a valence of 5. This patch contains 18 vertices, so first we store the number 18. Next, we store the indices 9, 5, 6, and 10 to indicate the central quad, using the irregular vertex as the starting point. Then we store vertex 8, which forms an edge with the first vertex and also forms a surface with the first and second vertices. After that, we store the indices that enclose the central quad, working around in the same direction in which the central quad was specified: 4, 0, 1, 2, 3, 7, 11, 17, 16, 15, 14, 13, and 12. At this point, we've stored 18 vertices, so the first patch is complete. We then go on to specify the second patch.

If two or more patches share the same irregular vertex, the indices of those patches must be stored consecutively. Vertex 9 is the irregular vertex in Figure 5-7, 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 irregular vertex are not consecutive in the index array, continuity between subdivision patches is not guaranteed and the mesh could develop holes.

5.3.4 Inputting Vertex Data for Catmull-Clark Subdivision

Input vertex attribute data to the subdivision shader in the following order: (1) vertex coordinates (from the vertex shader), (2) quaternions (if used), (3) other attributes. Define the following in vertex shader assembly code and output the data to the corresponding output registers. If you use quaternions, you must output them after the vertex coordinates.

```
#pragma output_map(position, o0) // Vertex coordinates
#pragma output_map(quaternion, o1)      // Quaternions
#pragma output_map(color, o2)    // Vertex colors and other such attributes
```

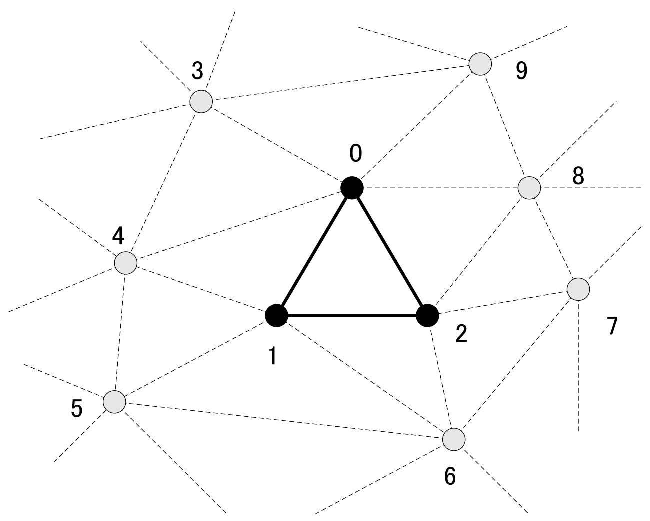
5.4 Loop Subdivision

This section explains Loop subdivision. Within this section, the word "subdivision" always indicates Loop subdivision.

5.4.1 Definition of a Loop Subdivision Patch

Loop subdivision patches are made from a single triangle and the surrounding vertices. These patches comprise a group of triangles that includes both the triangle to subdivide (the center triangle) and all vertices that share an edge with the vertices in the center triangle.

Figure 5-9: Example of a Loop Subdivision Patch



If the triangle defined by vertices (0, 1, 2) is to be subdivided, then the patch includes vertices 0, 1, and 2, along with the vertices that share edges with those vertices (3, 4, 5, 6, 7, 8, and 9).

When subdivision is performed, the center triangle within each patch is broken down into smaller, finer polygons that form a smoother surface.

5.4.2 Restrictions on Loop Subdivision Patches

The following restrictions apply to subdivision patches when using Loop subdivision.

- Vertices of center triangles must all have a valence (number of adjacent edges) falling within the range from 3 to 12.
- For all center triangles, the sum of the valences of all three vertices must be 29 or less.

In the patch example shown in Figure 5-9, the respective valences of vertices 0, 1, and 2 in the center triangle are 6, 5, and 5.

5.4.3 Specifying Indices for Loop Subdivision Patches

Subdivision patches can only be used with **`glDrawElements`**. Rendering with **`glDrawArrays`** is not supported. A vertex buffer must also be used (call a function like **`glBindBuffer`** or **`glBufferData`** to set up the buffer). The stored data that includes the vertex indices must be used only via the vertex buffer.

Before specifying the vertex indices of each subdivision patch, you must specify the overall patch size. The patch size is defined as the sum of the valences of the three vertices of the center triangle + 3. Specify this number at the beginning of the patch.

The order used to specify indices is shown below.

© 2009–2012 Nintendo
CONFIDENTIAL

1. Specify the three vertices that make up the center triangle as the first, second, and third elements. We'll call these v0, v1, and v2. Make sure to comply with the **`glFrontFace`** setting so that the center triangle is front-facing.
2. Specify all vertices that share edges with v0. (The order doesn't matter.)
3. Specify all vertices that share edges with v1. (The order doesn't matter.)
4. Specify all vertices that share edges with v2. (The order doesn't matter.)
5. Specify a fixed value of 12, and then specify the three vertices that make up the center triangle once again.
6. Specify the vertex that shares an edge with v0 and v2 but doesn't belong to the center triangle. In the same way, specify the vertex that shares an edge with v0 and v1 but doesn't belong to the center triangle, followed by the vertex that shares an edge with v1 and v2 but doesn't belong to the center triangle. We'll call these e00, e10, and e20, respectively.
7. Specify the vertex that shares an edge with v0 and is adjacent to v0 next after e00 in the counterclockwise direction. In the same way, specify the vertex that shares an edge with v1 and is adjacent to v1 next after e10 in the counterclockwise direction. Then specify the vertex that shares an edge with v2 and is adjacent to v2 next after e20 in the counterclockwise direction.
8. Specify the vertex that shares an edge with v0 and is adjacent to v0 next after e10 in the clockwise direction. In the same way, specify the vertex that shares an edge with v1 and is adjacent to v1 next after e20 in the clockwise direction. Then specify the vertex that shares an edge with v2 and is adjacent to v2 next after e00 in the clockwise direction.

In the example of a patch shown in Figure 5-9, the indices are as follows:

19, 0, 1, 2, 1, 2, 8, 9, 3, 4,   2, 0, 4, 5, 6,   0, 1, 6, 7, 8

12, 0, 1, 2, 8, 4, 6, 9, 5, 7, 3, 5, 7,

Vertices that share an edge with a vertex of the center triangle (indicated by the second, third, and fourth indices) may be specified in any order for that vertex, but they must use the same order in all patches. In Figure 5-9, for example, the index order of all vertices that share an edge with vertex 0 in the patch for center triangle (0, 1, 2) must be the same as the index order of all vertices that share an edge with vertex 0 in the patch for center triangle (3, 4, 0).

5.4.4 Inputting Vertex Data for Loop Subdivision

With subdivision shaders, you must input both vertex coordinates and valences from the vertex shaders to the geometry shaders. Output the vertex attribute data in the following order: (1) vertex coordinates, (2) quaternions (if used), (3) other attributes (if used), (4) valences. Define the following in vertex shader assembly code and output the data to the corresponding output registers. If you use quaternions, you must output them after the vertex coordinates.

```
#pragma output_map(position, o0)        // Vertex coordinates
#pragma output_map(quaternion, o1)      // Quaternions
#pragma output_map(color, o2)           // Vertex colors and other such attributes
#pragma output_map (generic, o3)        // Valences (use the "generic" attribute)
```

One of the limitations of the Loop subdivision geometry shader is that the data output from the vertex shaders (excluding valences) can occupy a maximum of only four output registers. To output more than four such vertex attributes, you must set multiple attributes to a single register. However, quaternions cannot be set to share an output register with another attribute.

```
#pragma output_map(position, o0)        // Vertex coordinates
#pragma output_map(quaternion, o1)      // Quaternions
#pragma output_map(view, o2.xyz)        // View vector
#pragma output_map(texture0w, o2.w)     // Texture coordinate w of texture 0
#pragma output_map(texture0, o3.xy)     // Texture coordinates of texture 0
#pragma output_map(texture1, o3.zw)     // Texture coordinates of texture 1
#pragma output_map ( generic , o4 )     // Valences
```

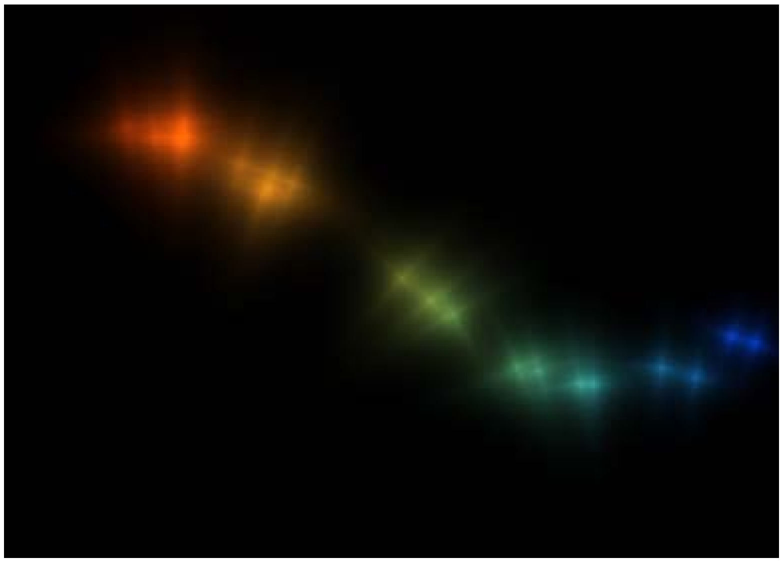
6 Particle Systems

This chapter explains how to program the particle system feature provided by DMPGL 2.0.

6.1 Overview

The particle system feature generates large quantities of point sprites along a Bézier curve. The point sprites that are generated by the particle system are called *particles*. Particle systems use a geometry shader. Specifically, particles are generated along the Bézier curve that is defined by the four control points configured in the geometry shader. Details about the particles (such as their color, size, and the rotation angle of the texture coordinates) are determined by interpolating the values set for each control point by the position of the particles along the Bézier curve. An example rendering is shown below.

Figure 6-1: Example of Rendering a Particle System



In this example, an RGBA color and a procedural texture are blended and then applied to the particles.

6.2 Setting the Control Point Coordinates

There are two types of settings for particle systems: those that are set directly in the geometry shaders using uniforms, and those that are output from the vertex shaders and then set. The settings that are output from the vertex shaders and then set are the vertex coordinates of the four control points that define the Bézier curve, and sizes of the bounding boxes. The positions of the control points during the actual rendering are set at random within the range of the bounding boxes.

The vertex coordinates and the sizes of the bounding boxes are transformed to the clip coordinate system by the vertex shaders, then output to the geometry shaders.

Set the output attributes in the vertex shader as follows.

```
#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 XYZ-axis radii are given by Rx, Ry, and Rz in object coordinates, `o1-o4` are calculated and output as follows.

Equation 6-1: Transforming the Bounding Box Size to Clip Space

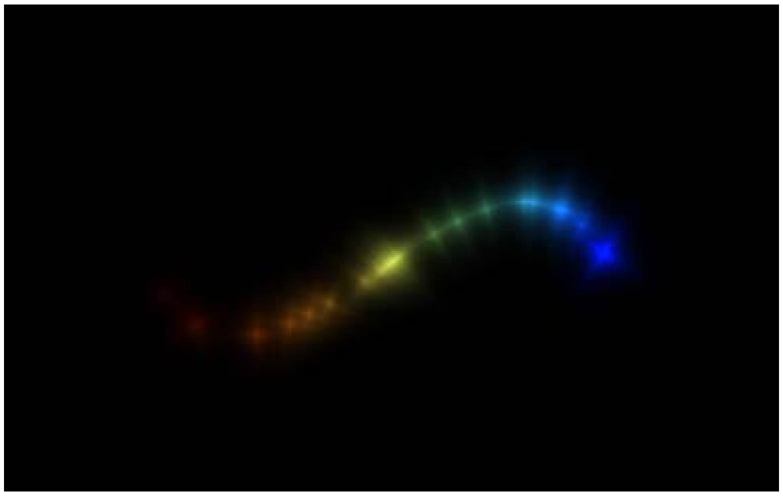
$$\begin{pmatrix} o1.x & o1.y & o1.z & 0 \\ o2.x & o2.y & o2.z & 0 \\ o3.x & o3.y & o3.z & 0 \\ o4.x & o4.y & o4.z & 0 \end{pmatrix} = M_{proj} \times M_{modelview} \times \begin{pmatrix} Rx & 0 & 0 & 0 \\ 0 & Ry & 0 & 0 \\ 0 & 0 & Rz & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

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

Because only four vertices worth of data are output from the vertex shaders, there are two ways of inputting the vertex coordinates and the sizes of the bounding boxes: (1) inputting them as vertex shader attributes and (2) setting them as uniforms.

The control points that define the Bézier curve are placed at random within the bounding boxes. The sample rendering below shows the effect of a change to the sizes of the bounding boxes.

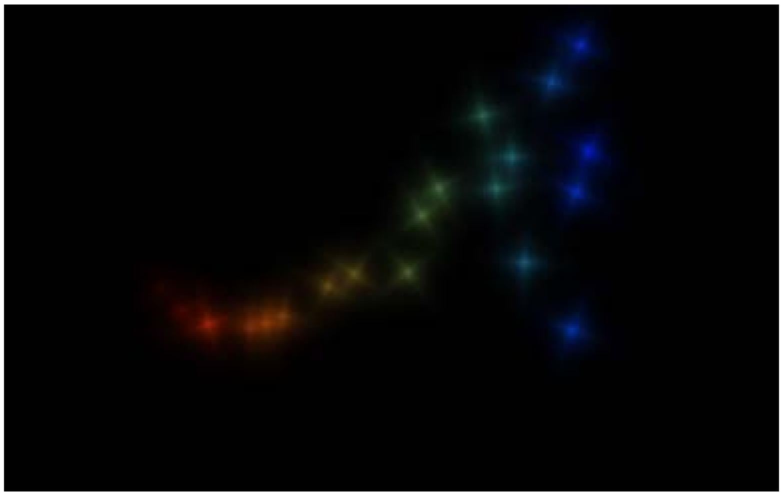
Figure 6-2: Rendering with the Bounding Box Sizes Set to Zero



In the figure above, the bounding boxes sizes are set to zero for all control points, so there is no variation in the particles; they are rendered exactly on top of a single Bézier curve.

© 2009–2012 Nintendo
CONFIDENTIAL

Figure 6-3: Rendering with the Bounding Box Sizes Changed



In the sample figure above, the control points are distributed from left to right. The size of the bounding box for the fourth (rightmost) control point is set to a large value, but the sizes of the bounding boxes for all other control points are set to zero. You can see that the closer we get to the control point with the large bounding box, the more widely the particles are being distributed.

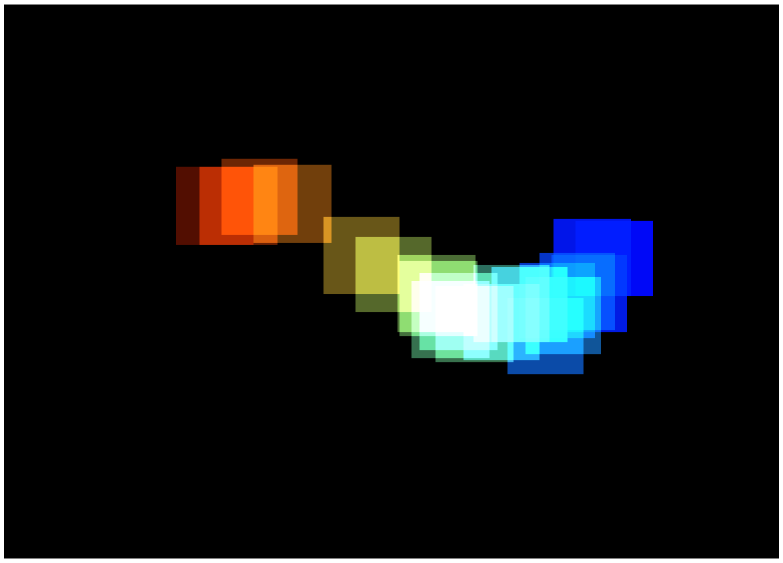
6.3 Color and Size of Particles

The color and size of the four control points are set using reserved uniforms. The color and size of each individual particle is calculated by interpolating the values set for the control points by the position of the particles along the Bézier curve.

When setting the color, different reserved uniforms are used depending on whether you specify all the RGBA components or just the alpha component. To set all of the RGBA components, use one of the reserved geometry shader objects `DMP_particleSystem_X_X_1_X.obj` (where “X” is either 0 or 1) and call **`glUniformMatrix4fv`** on the reserved uniform `dmp_PartSys.color`, specifying a pointer to a 4x4 matrix that stores the RGBA values for the four control points. The rows of the 4x4 matrix correspond to the control points: the color of the first control point is stored in the first row, the color of the second control point is stored in the second row, and so on. The columns of the 4x4 matrix correspond to the individual components, stored in the order R, G, B, A (the R values are stored in the first column, the G values are stored in the second column, and so on). To set only the alpha component, use one of the reserved geometry shaders `DMP_particleSystem_X_X_0_X.obj` and call **`glUniformMatrix4fv`** 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 the 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 only the alpha component results in better performance than using all of the RGBA components.

The figure below shows an example rendering with RGBA colors applied to the particles.

Figure 6-4: Rendering of Particles Using RGBA Color

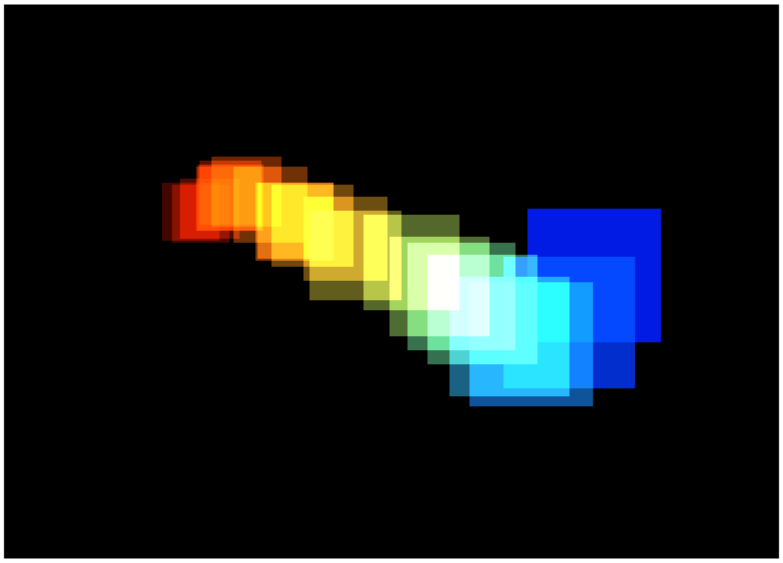


In Figure 6-4 above, the four control points are distributed from left to right. The RGB components of the colors for the control points are as follows. The first control point is set to (0.f, 0.f, 0.f), the second is set to (1.f, 0.f, 0.f), the third is set to (0.f, 1.f, 0.f), and the fourth is set to (0.f, 0.f, 1.f).

The size of the particles is set by calling **`glUniformMatrix4fv`** on the reserved uniform `dmp_PartSys.aspect`, specifying a 4x4 matrix whose first column stores the sizes of the four control points. Store the size 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.

In the example shown in Figure 6-4, the particle sizes for all control points have been set to the same value. The figure below shows what it looks like if each control point is set to a different size.

Figure 6-5: Rendering with the Particle Size Changed

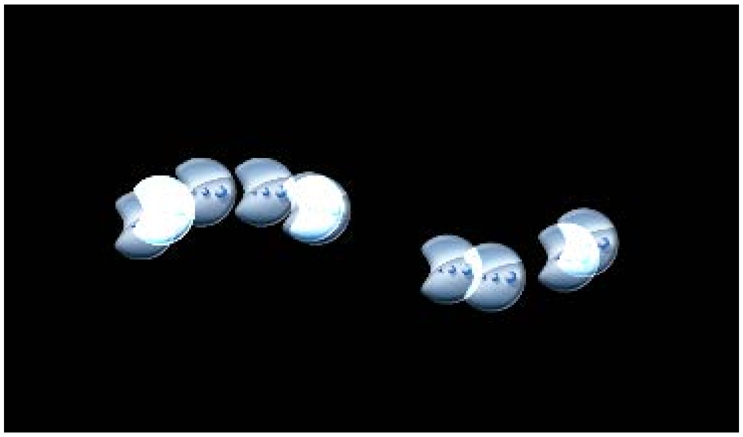


In Figure 6-5 above, the size of the control points are set as follows: the first control point is set to 3.0, the second is set to 5.0, the third is set to 7.0, and the fourth is set to 9.0.

6.4 Using Textures

Particle system geometry shaders generate texture coordinates 0 and 2 automatically. Texture coordinate 0 is always generated, whereas you can choose whether to generate texture coordinate 2. Texture coordinate 2 is generated if you use one of the reserved geometry shaders `DMP_particleSystem_X_X_X_1.obj`. It isn't generated if you use one of the reserved geometry shaders `DMP_particleSystem_X_X_X_0.obj`. For texture coordinate 0, the *uv* coordinates (0,0), (1,0), (0,1), and (1,1), indicate the lower-left, lower-right, upper-left, and upper-right of a particle, respectively. For texture coordinate 2, the *uv* coordinates (-1, -1), (1,-1), (-1,1), and (1,1) indicate the lower-left, lower-right, upper-left, and upper-right of a particle, respectively. The example shown in Figure 6-6 below uses texture coordinate 0.

Figure 6-6: Rendering of a Particle System with Texture Coordinate 0 Applied



The texture coordinates can be rotated and scaled. (Only texture coordinate 2 can be scaled.) If you use one of the reserved geometry shaders `DMP_particleSystem_X_1_X_X.obj`, texture coordinates are not rotated or scaled. If you use one of `DMP_particleSystem_X_0_X_X.obj`, texture coordinates are rotated and scaled. There is no texture coordinate rotation in the example in Figure 6-6.

The angle of rotation and the scaling value for texture coordinates are determined by interpolating the values set for the control points by the location of a given particle along the Bézier curve. To set these values, call **`glUniformMatrix4fv`** on the reserved uniform `dmp_PartSys.aspect`, specifying a pointer to a 4x4 matrix whose second column stores the rotation angles of the texture coordinates for the four control points (in radians), and whose third column stores the scaling factor. Store the settings for control points 1-4 in rows 1-4 of the 4x4 matrix, respectively.

Given a rotation angle of A and a scaling value of R, the *uv* coordinates for texture coordinate 0 are calculated as follows:

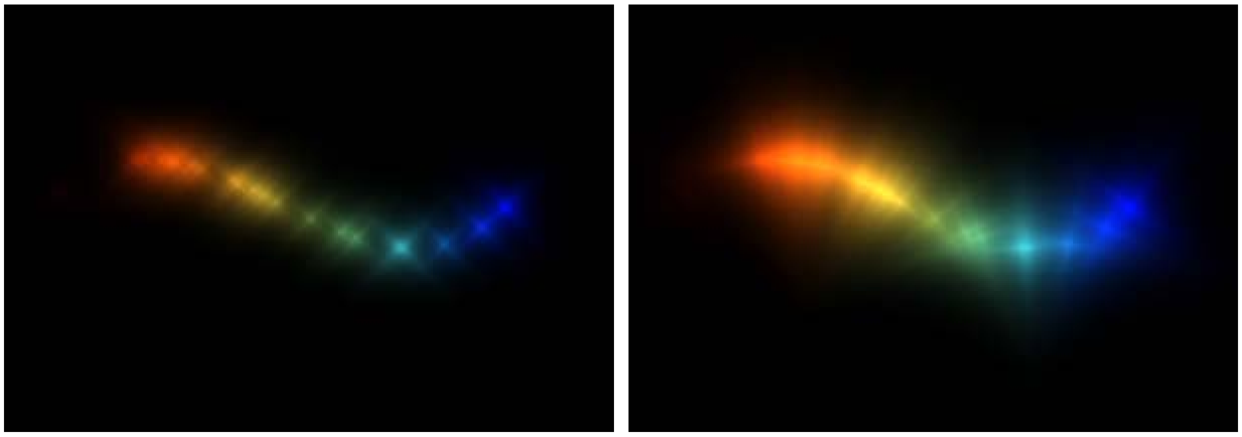
Corner	Formula for Rotation Angle
Lower left	$(0.5 \times (1.0 + (-\cos A + \sin A)), -\cos A - \sin A)))$
Lower right	$(0.5 \times (1.0 + (\cos A + \sin A)), -\cos A + \sin A)))$
Upper left	$(0.5 \times (1.0 + (-\cos A - \sin A)), \cos A - \sin A)))$
Upper right	$(0.5 \times (1.0 + (\cos A - \sin A)), \cos A + \sin A)))$

The *uv* coordinates for texture coordinate 2 are calculated as follows:

Corner	Formula for Rotation Angle
Lower left	$(R(-\cos A + \sin A), R(-\cos A - \sin A))$
Lower left	$(R(\cos A + \sin A), R(-\cos A + \sin A))$
Upper left	$(R(-\cos A - \sin A), R(\cos A - \sin A))$
Upper right	$(R(\cos A - \sin A), R(\cos A + \sin A))$

The figure below shows an example of texture coordinate rotation.

Figure 6-7: Texture Coordinate Rotation



The image on the left uses no rotation, and the image at the right uses both rotation and scaling. The particle size is set to the same value in both images.

6.5 Particle Time

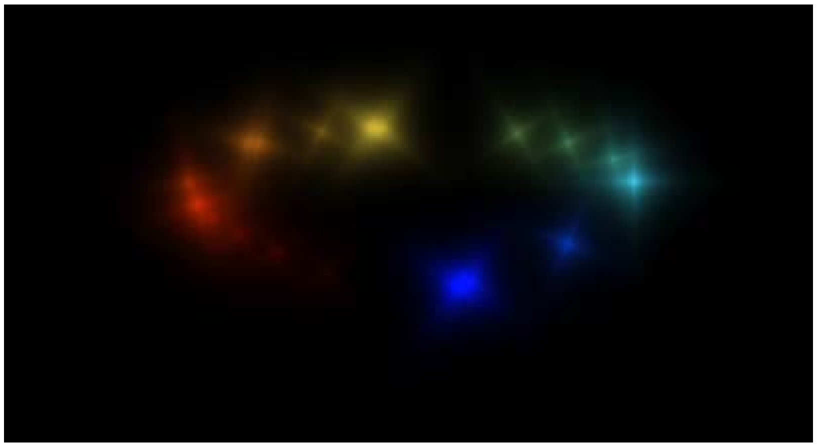
Particle systems use the concept of time. To set the current time, call **`glUniform1fv`** on the reserved uniform `dmp_PartSys.time` with *`value`* set to an array storing the current time. Time moves forward as the value increases, or moves backward as the value decreases. As the time moves forward, particles are generated at the first control point, and move toward the fourth control point. To set the speed of particle movement, call **`glUniform1fv`** on the reserved uniform `dmp_PartSys.speed` with *`value`* set to an array storing a value for the speed. The specified time is converted by a random value for each particle. If the final value for a particle falls within the range [0, 1], the particle is generated somewhere between the first and fourth control points.

You can set whether to generate particles when the time falls outside the range [0, 1]. If one of the reserved geometry shaders `DMP_particleSystem_0_X_X_X.obj` is used, particles that fall outside the range [0, 1] are not rendered. (This is referred to as "time clamping" from this point in the document onward.) When one of `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 at the first control point.

When using time clamping, simply letting the current time value increase will cause particles to stop being generated at a certain point in time. It is therefore necessary to reset the time from the application.

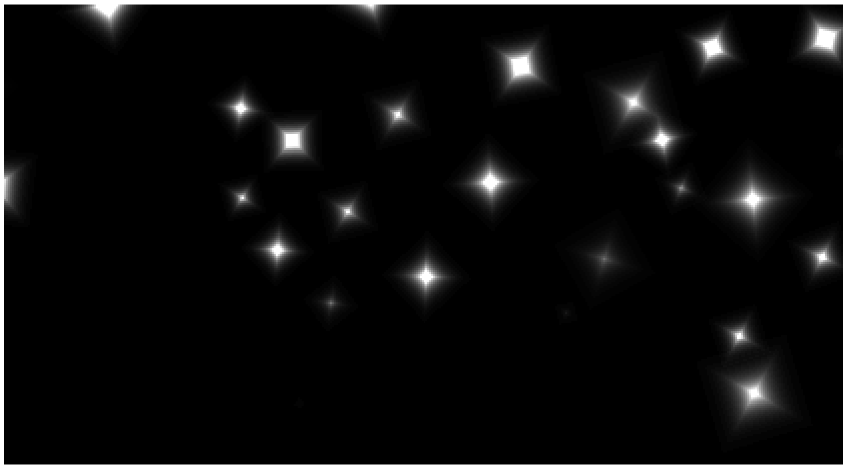
6.6 Sample Particle Renderings

Figure 6-8: Particles Generated in a Circular Pattern



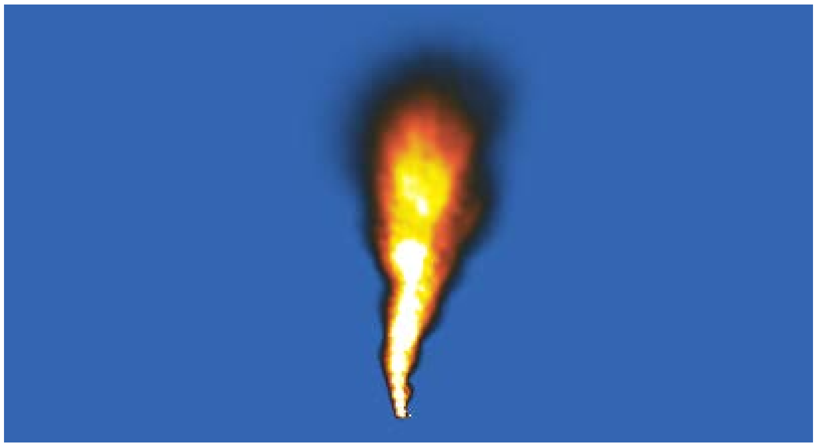
In this example, the first and fourth control points are set to the same vertex coordinates, and the particles are generated along a circular path.

Figure 6-9: Particles Generated to Simulate Falling Snow



In this example, the bounding boxes are set very wide in the horizontal direction. The first control point has been placed at the top of the screen, and the fourth control point has been placed at the bottom of the screen to approximate the appearance of falling snow.

Figure 6-10: Rendering Using a Particle System for the Density-Rendering Pass of a Gas



This can be used for effects like towering flames.

© 2009–2012 Nintendo
CONFIDENTIAL

7 Textures

Textures behave just like they do in OpenGL ES 2.0 and can be used without modification. However, the detailed aspects involve some DMPGL 2.0-specific restrictions, so please keep these in mind.

7.1 Enabling and Disabling Textures

In OpenGL ES 2.0, textures are enabled by calling **`glEnable`** with the argument `GL_TEXTURE_2D`. However, in DMPGL 2.0, this is done by setting the sampler type in a reserved uniform. The reserved uniform variable that is set is `dmp_Texture[i].samplerType`.

Code 7-1: Enabling Texture 0

```
glUniform1i(LOC(”dmp_Texture[0].samplerType”),GL_TEXTURE_2D);
```

Code 7-2: Disabling Texture 0

```
glUniform1i(LOC(”dmp_Texture[0].samplerType”),GL_FALSE);
```

In Code 7-1 above, texture unit 0 is enabled as a 2D texture. Other features can be used by changing the sampler type. If `FALSE` is specified, the texture in question is disabled. If you call **`glEnable`**`(GL_TEXTURE_2D)`, behavior is undefined.

7.2 Restrictions on Coordinates

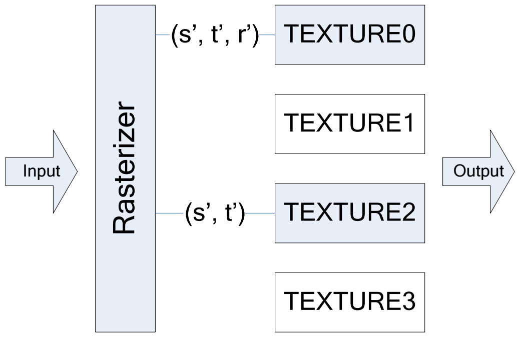
Here, we’ll use the notation `(s, t, r, g)` to refer to the set of texture coordinates that the vertex shaders take as input. Likewise, `(s’, t’, r’)` refers to the coordinates output to textures by the vertex shaders. With DMPGL 2.0, only texture unit 0 can take all three components `(s’, t’, r’)`. Note that all other texture units can only take two components `(s’, t’)` and will not divide by `r’`.

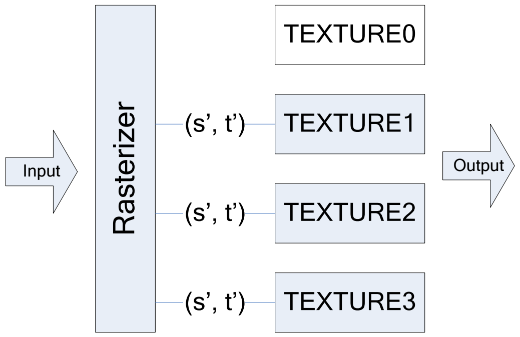
Although there are four texture units, the rasterizer can only provide three sets of texture coordinates independently.

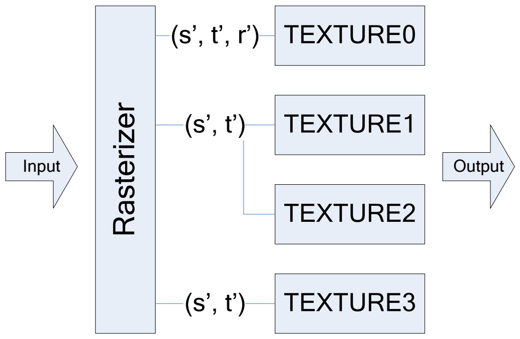
Texture coordinate 0 is provided to texture unit 0, and texture coordinate 1 is provided to texture unit 1. You can choose whether to provide texture coordinates 1 or 2 to texture unit 2. You can choose whether to provide texture coordinates 0, 1, or 2 to texture unit 3. Use the reserved uniforms `dmp_Texture[2].texcoord` and `dmp_Texture[3].texcoord` to set your selections for texture units 2 and 3.

Table 7-1 below shows how the rasterizer and texture units are connected when the rasterizer provides different sets of coordinates to the texture units.

Table 7-1: Examples of Providing Coordinates from the Rasterizer

Units Used	Connection to Rasterizer	Description
`TEXTURE0` `TEXTURE2`	 Input Rasterizer (s', t', r') TEXTURE0 TEXTURE1 (s', t') TEXTURE2 TEXTURE3 Output	This configuration tries to use texture 0 and texture 2. It is possible to provide `r'` to texture 0. Division by `r'` is possible for texture 0.

`TEXTURE1`
`TEXTURE2`
`TEXTURE3`
	 Input
	Rasterizer
	TEXTURE0
	(s', t')
	TEXTURE1
	(s', t')
	TEXTURE2
	(s', t')
	TEXTURE3
	Output
		If you provide texture coordinate 2 to texture unit 2, and provide texture coordinate 0 to texture unit 3, the provided texture coordinates can all be mutually independent.
		Although texture 0 is disabled, `r'` still cannot be provided to the other textures.

`TEXTURE0`
`TEXTURE1`
`TEXTURE2`
`TEXTURE3`
	 Input
	Rasterizer
	(s', t', r')
	TEXTURE0
	(s', t')
	TEXTURE1
	TEXTURE2
	(s', t')
	TEXTURE3
	Output
		When using all textures, either texture unit 2 or 3 will share texture coordinates with other texture units.

The texture coordinates sent from the vertex shaders are mapped to the vertex shader output registers by specifying their attribute name and output register to `#pragma output_map` as the *`data_name`* and *`mapped_register`* arguments, respectively.

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

For details on how texture coordinates are assigned to output registers by the vertex shaders, see the *Vertex Shader Reference Manual*. Table 7-2 shows how the value of `data_name` corresponds to texture coordinates sent from the vertex shaders.

Table 7-2: Values of `data_name` When Mapping Texture Coordinates to Output Registers

`data_name`	Attributes Sent From the Vertex Shader
`texture0`	uv components of texture coordinate 0
`texture0w`	`w` component of texture coordinate 0
`texture1`	uv components of texture coordinate 1
`texture2`	uv components of texture coordinate 2

The vertex attributes for `texture0` are the only ones that can contain the `w` coordinate in addition to the `u` and `v` coordinates. Be sure to always configure the `texture0w` vertex attribute for any features that require the `w` coordinate, such as cube mapping, shadows, and projective textures. If you do not configure `texture0w` when the *w* coordinate is required, the output of texture 0 is undefined. If texture 0 is being used as a 2D texture, and a value for `texture0w` is output by the vertex shaders, this value is simply ignored.

7.3 Precautions When the Filter Mode Is `GL_NEAREST`

Suppose the `GL_TEXTURE_MIN_FILTER` or `GL_TEXTURE_MAG_FILTER` texture parameter is set to `GL_NEAREST`, and an image with straight horizontal or vertical lines and clear boundaries between colors is used as the texture. If those straight lines are rendered to on-screen polygons in such a way that they are oriented parallel to or perpendicular to the screen's scanlines, the pattern will sometimes appear to be uneven, even though it should appear perfectly straight. This will occur, for example, if a texture with horizontal or vertical stripes is rendered to a rectangular polygon whose sides run parallel with the screen's scanlines.

This phenomenon is caused by the precision of the calculations used to interpolate texture coordinates within polygons.

Consider a row or column of fragments that run either perpendicular or parallel to the scanlines. If each fragment samples a row or column of texels in the texture, a straight line in the texture will be rendered as a straight line on the polygon.

However, if that row or column of fragments samples near the boundary between two neighboring rows or columns, the margin or error in the texture coordinates will cause some of those fragments to sample from one row or column, and others to sample from the other. When this happens, straight lines within the texture image will be rendered as uneven lines on the polygon.

You can prevent this phenomenon by adjusting the texture coordinates or the render area so that the fragments will sample the centers of the texels in the texture instead of the edges. For example, if you had a quad whose texture coordinates were 0 at one edge and 1 at the other, rendering that polygon at the same size as the texture would sample the centers of the texels.

7.4 Precautions When the Filter Mode Is `GL_XXX_MIPMAP_LINEAR`

Trilinear filtering is enabled when texture parameter `GL_TEXTURE_MIN_FILTER` is `GL_XXX_MIPMAP_LINEAR`. Trilinear filtering generates the final color by interpolating the texture colors of two mipmap levels, but calculation mistakes may occur due to the accuracy of the interpolation process. Similar mistakes may also occur when interpolating two identical colors.

When the selected mipmap level changes for a given pixel on a polygon, such as changing from level 0 to level 1, the color changes from the level 0 texture color to the interpolated color from the texture colors for levels 0 and 1, and then the level 1 texture color. If a mipmap level is selected to get the texture color of a single level, no colors are interpolated. Consequently, the interpolated color value and the non-interpolated color value may exhibit some slight discrepancies, with the interpolated color appearing slightly darker. On the polygon, the artifact of this difference causes lines that look like boundaries between the mipmap levels to appear.

Use the texture's fixed value color as a means of reducing this issue. For example, assume a texture in `GL_RGB` format has an alpha component with a fixed value of 1.0. This fixed-value component and the trilinear interpolated value will also be slightly different. In this case, the non-interpolated value will simply be 1.0. The texture combiner will then correct the color by multiplying the texture color by the difference in color values for this alpha component.

The following code shows an example of configuring the combiner.

Code 7-3: Combiner Settings

```
glUniform1i(LOC("dmp_TexEnv[0].combineRgb"), GL_MULT_ADD_DMP);
glUniform3i(LOC("dmp_TexEnv[0].operandRgb"),
  GL_SRC_COLOR, GL_ONE_MINUS_SRC_ALPHA, GL_SRC_COLOR);
glUniform3i(LOC("dmp_TexEnv[0].srcRgb"),
  GL_TEXTURE0, GL_TEXTURE0, GL_TEXTURE0);
```

The RGB component input uses the texture colors. The second input is `ONE_MINUS_SRC_ALPHA`, so 0 is entered for the non-interpolated portion, and the difference value is entered for the interpolated portion. The original texture color is multiplied by this difference value, and this is then added to the original texture color.

If there are no components storing fixed-value colors, as with the `GL_RGBA` format, then a similar method may be used. This would use another texture for the fixed-value color and use this as a multi-texture. The texture with the fixed-value color must have the same number of texels (i.e., be the same size), have the same number of mipmap levels, and have the same `uv` value entered as for the original texture. We recommend using ETC1 textures from a data size and data cache efficiency perspective.

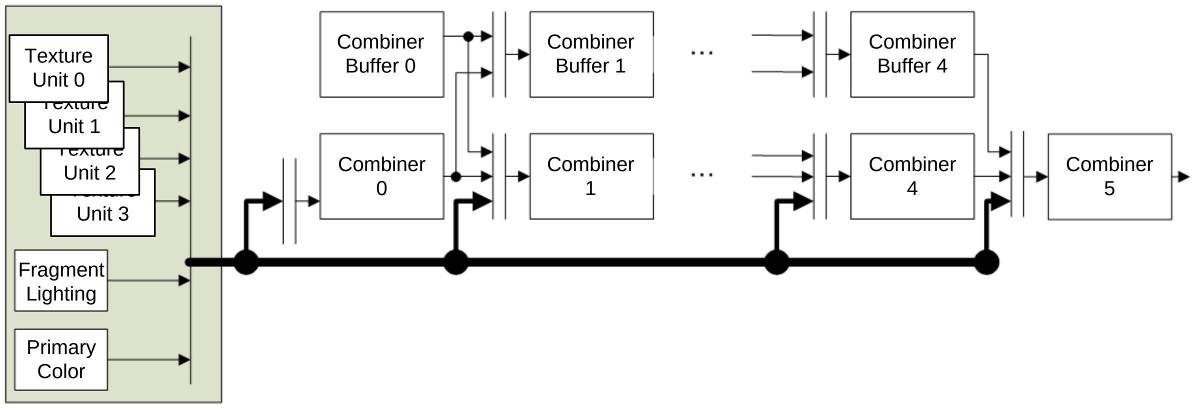
When texture colors are interpolated, their hues always fade by a fixed amount that depends on their color values. Large (bright) color values fade more than small (dark) color values. For texture colors between 0.0 and 1.0 that are represented as 8-bit colors between 0 and 255, the maximum texture color value of 255 fades the most: two color values of 255 interpolate to a value of 251 (a color loss of 4). Interpolating the same texture color between two mipmap levels entails a color loss of 4, 3, 2, 1, or 0 when the original value is in the range 225–255, 161–224, 97–160, 33–96, or 0–32, respectively.

8 Texture Combiners

DMPGL texture combiners provide nearly the same capabilities of texture combiners as defined by the OpenGL standard. Texture combiners make it possible to combine the following inputs using the hardware's arithmetic unit: primary colors provided by the rasterizer, texture colors sampled by the four texture units, fragment primary and secondary colors output by fragment lighting, output from the previous-stage texture combiners, and output from the previous-stage combiner buffers. Once these are combined, the result can be output.

For more information about the various units that provide colors to the texture combiners, see the corresponding sections within this document. DMPGL 2.0 uses a unique set of connections when the gas feature is being used. For details, see Chapter 16 Gas.

Figure 8-1: Texture Combiner Connections



Texture combiners are always enabled; there is no way to enable or disable them. DMPGL 2.0's default texture combiner configuration outputs the primary color. If you want to change the input sources (for example, when using textures), you must explicitly change the combiner settings.

Up to three colors can be chosen as sources for a single texture combiner; these colors can be blended using the selected combiner function. Combiners 0, 1, 2, 3, 4, and 5 are connected in a cascading fashion, with combiner 0 being closest to the input. Combiners can choose the output of the previous combiner as the source. (In other words, combiner `n` can choose the output of combiner `n-1` as its source.) Each combiner can also have a single constant color.

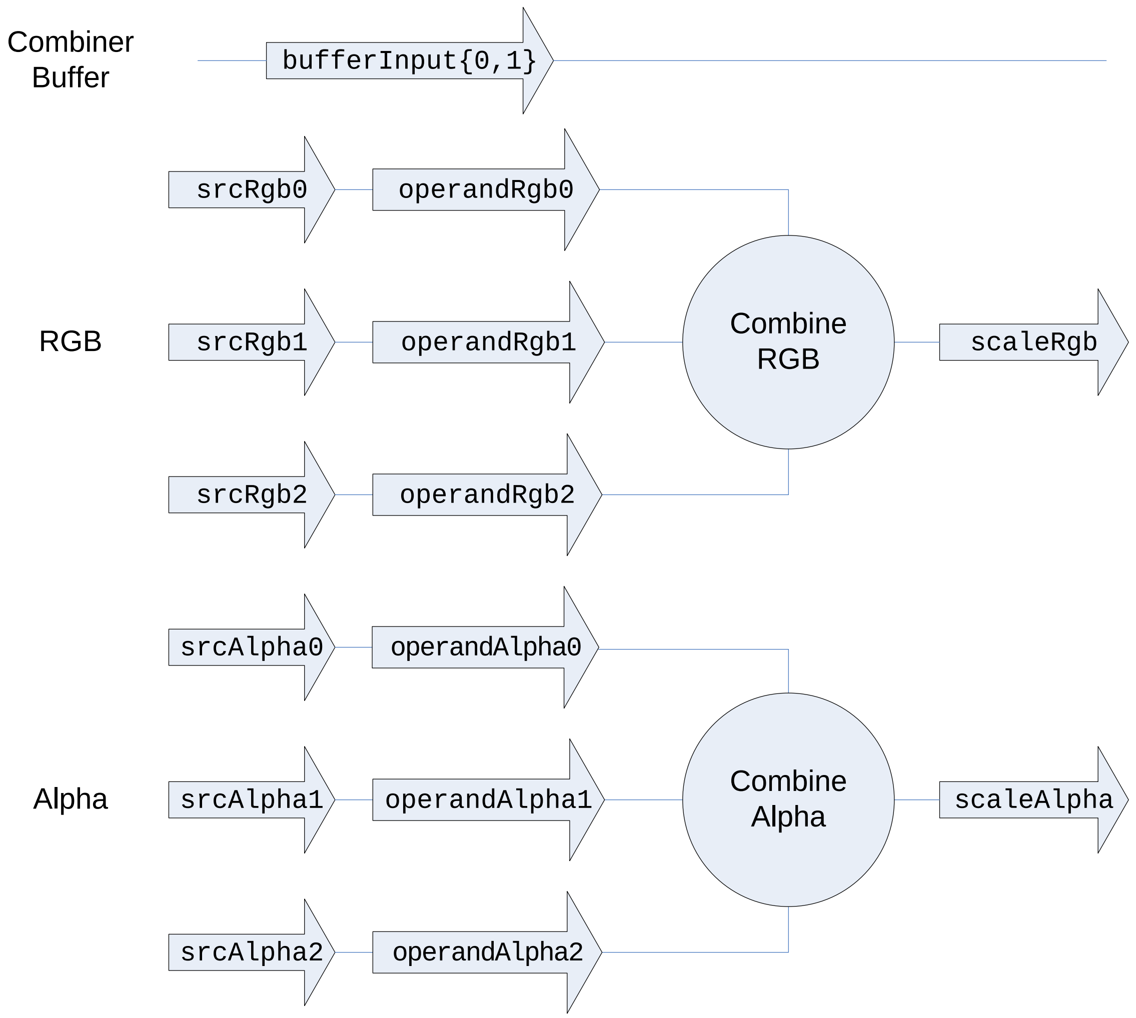
All combiners except for the final combiner (combiner 5) have a *combiner buffer* that can be used in parallel with the combiner itself. These combiner buffers are put there so that a combiner can access combiners other than the one that immediately precedes it. Each combiner buffer except for the first one can choose either the preceding combiner buffer or the output of the preceding combiner as input. In previous implementations, it was possible to use textures and output from the preceding combiner as input. With the new implementation, it is now also possible to use the output of the immediately preceding combiner buffer as input.

The output of the first combiner buffer is a constant color called the *buffer color*.

8.1 Texture Combiner Properties

The following properties can be set for each combiner: `srcRgb{0,1,2}`, `srcAlpha{0,1,2}`, `operandRgb{0,1,2}`, `operandAlpha{0,1,2}`, `combineRgb`, `combineAlpha`, `scaleRgb`, `scaleAlpha`, and `bufferInput{0,1}`. Figure 8-2 below shows the order in which the effects of these properties are applied.

Figure 8-2: Combiner Configuration



The property names shown in Figure 8-2 above are the names that serve as suffixes to the reserved uniform name string "`dmp_TexEnv[i].`" The notation `{0,1,2}` refers to the first, second, and third arguments to the reserved uniforms.

The `srcRgb{0,1,2}` and `srcAlpha{0,1,2}` properties set the sources that are combined to yield the RGB color and alpha, respectively. For each of these properties you can choose a texture color, fragment lighting color, or vertex color as the source.

The `operandRgb{0,1,2}` and `operandAlpha{0,1,2}` properties set whether to send their respective sources directly to the combiner function, or whether to invert them before doing so.

The `combineRgb` and `combineAlpha` properties specify the combiner functions that yield the RGB color and the alpha, respectively.

The `scaleRgb` and `scaleAlpha` properties set the constants by which the output colors are multiplied.

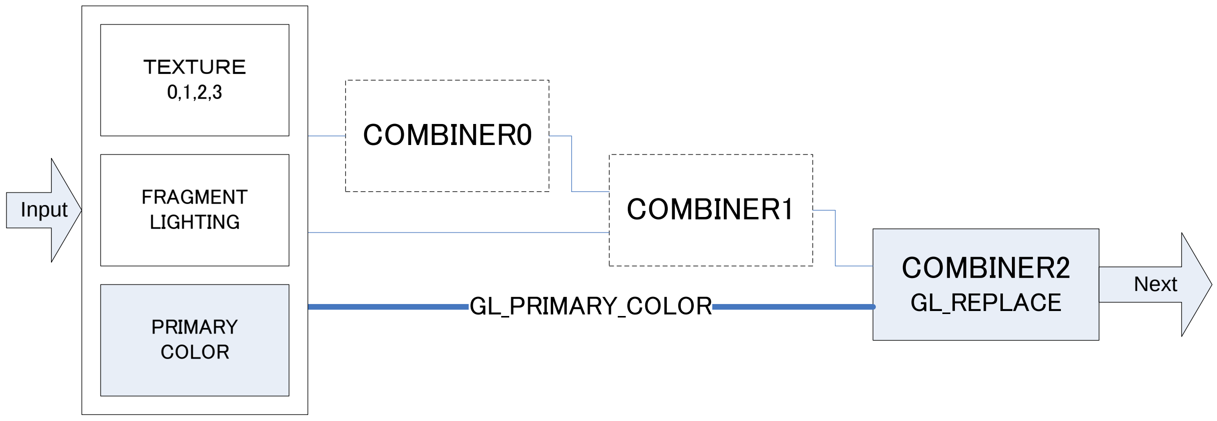
The `bufferInput{0,1}` property chooses the source that is input into the combiner buffer. The color and alpha values can be specified separately.

8.2 Sample Texture Combiner Settings

The example in Figure 8-3 below shows a way of rendering just the unmodified primary color. This is done by referencing the primary color from combiner 2 and outputting it directly to the next stage. To do this, first set the source 0 of combiner 2 to the primary color (`GL_PRIMARY_COLOR`), then set operand 0 to pass it through unmodified (`GL_SRC_COLOR`), set the combiner function to output source 0 without modification (`GL_REPLACE`), and set the scaling factor to 1.0. These settings will cause the texture combiners to output just the primary color and apply no other effects. In this case, combiners 0 and 1 will not affect the result at all.

The connections for this case are illustrated in Figure 8-3 below.

Figure 8-3: Outputting Only the Primary Color



These settings would look like the following when coded within a program:

© 2009–2012 Nintendo
CONFIDENTIAL

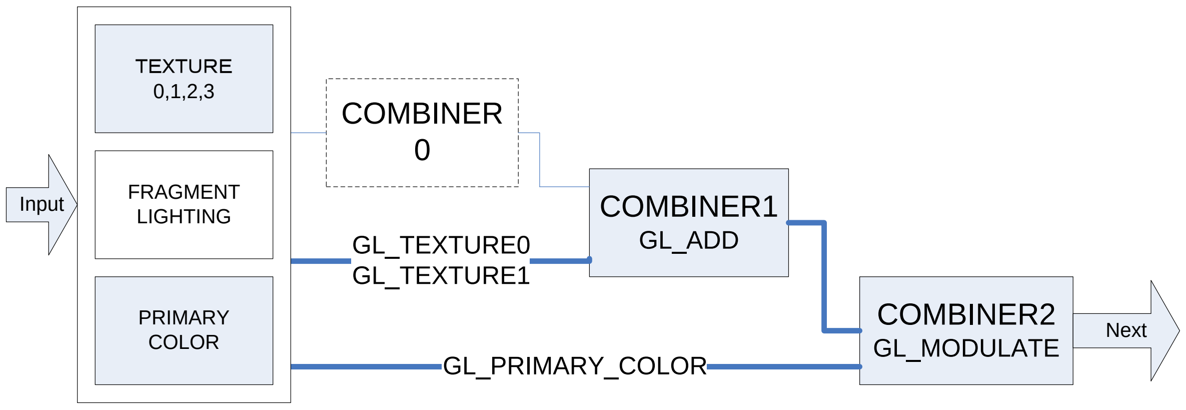
Code 8-1: Texture Combiner Settings for Outputting Only the Primary Color

```
  glUniform3i(LOC(”dmp_TexEnv[2].srcRgb”),GL_PRIMARY_COLOR,
  GL_PREVIOUS,GL_PREVIOUS);
  glUniform3i(LOC(”dmp_TexEnv[2].srcAlpha”),GL_PRIMARY_COLOR,
  GL_PREVIOUS,GL_PREVIOUS);
  glUniform1i(LOC(”dmp_TexEnv[2].combineRgb”),GL_REPLACE);
  glUniform1i(LOC(”dmp_TexEnv[2].combineAlpha”),GL_REPLACE);
  glUniform3i(LOC(”dmp_TexEnv[2].operandRgb”),GL_SRC_COLOR, GL_SRC_COLOR,
GL_SRC_COLOR);
  glUniform3i(LOC(”dmp_TexEnv[2].operandAlpha”),GL_SRC_ALPHA, GL_SRC_ALPHA,
GL_SRC_ALPHA);
  glUniform1f(LOC(”dmp_TexEnv[2].scaleRgb”),1.0);
  glUniform1f(LOC(”dmp_TexEnv[2].scaleAlpha”),1.0);
```

In the example in Code 8-1 above, the primary color is selected as source 0 for texture combiner 2, and passed through to output without any modification.

Next, let’s take a look at a more complicated configuration. In the following example, we’ve configured the texture combiners so that textures 0 and 1 are added in texture combiner 1, and the result is then multiplied by the primary color in texture combiner 2. Figure 8-4 shows these connections.

Figure 8-4: A More Complicated Example of Texture Combiner Connections



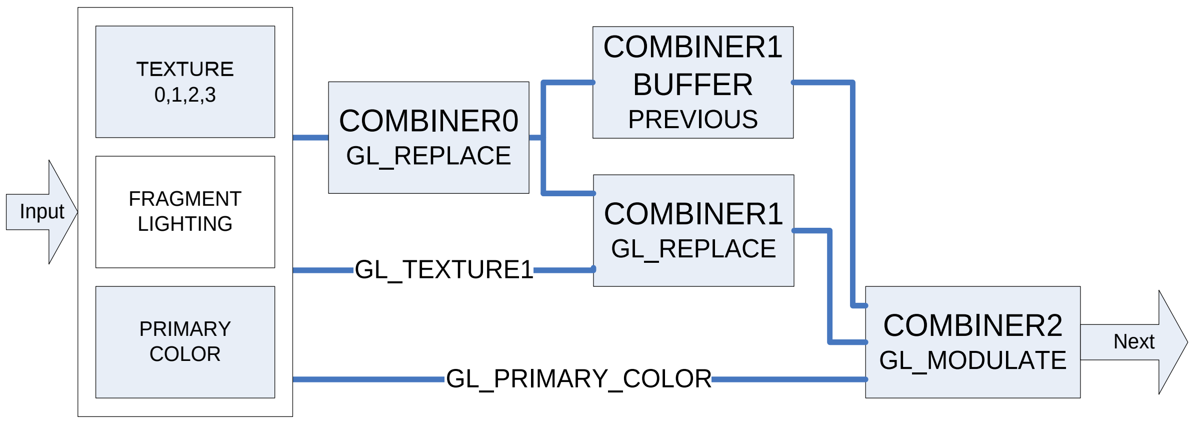
These connections would be coded as follows:

Code 8-2: Texture Combiner Settings for Figure 8-4

```
  glUniform3i(LOC(”dmp_TexEnv[1].srcRgb”),GL_TEXTURE0,
GL_TEXTURE1,GL_PREVIOUS);
  glUniform3i(LOC(”dmp_TexEnv[1].srcAlpha”),GL_TEXTURE0,
  GL_TEXTURE1,GL_PREVIOUS);
  glUniform1i(LOC(”dmp_TexEnv[1].combineRgb”),GL_ADD);
  glUniform1i(LOC(”dmp_TexEnv[1].combineAlpha”),GL_ADD);
  glUniform3i(LOC(”dmp_TexEnv[1].operandRgb”),GL_SRC_COLOR,
  GL_SRC_COLOR,GL_SRC_COLOR);
  glUniform3i(LOC(”dmp_TexEnv[1].operandAlpha”),GL_SRC_ALPHA,
  GL_SRC_ALPHA,GL_SRC_ALPHA);
  glUniform3i(LOC(”dmp_TexEnv[2].srcRgb”),GL_PREVIOUS,
GL_PRIMARY_COLOR,GL_PREVIOUS);
  glUniform3i(LOC(”dmp_TexEnv[2].srcAlpha”),GL_PREVIOUS,
  GL_PRIMARY_COLOR,GL_PREVIOUS);
  glUniform1i(LOC(”dmp_TexEnv[2].combineRgb”),GL_MODULATE);
  glUniform1i(LOC(”dmp_TexEnv[2].combineAlpha”),GL_MODULATE);
  glUniform3i(LOC(”dmp_TexEnv[2].operandRgb”),GL_SRC_COLOR,
  GL_SRC_COLOR,GL_SRC_COLOR);
  glUniform3i(LOC(”dmp_TexEnv[2].operandAlpha”),GL_SRC_ALPHA,
  GL_SRC_ALPHA,GL_SRC_ALPHA);
```

Let’s try to extend this further and use a combiner buffer to let combiner 2 access combiner 0. Combiner 0 will output texture 0, combiner 1 will output texture 1, and combiner 2 will `MODULATE` the output of the two preceding combiners (combiners 1 and 0).

Figure 8-5: Connections Using a Previous Buffer



Code 8-3: Texture Combiner Settings for Figure 8-5

```
glUniform3i(LOC(”dmp_TexEnv[0].srcRgb”),GL_TEXTURE0,
                                     GL_CONSTANT,GL_CONSTANT);
glUniform3i(LOC(”dmp_TexEnv[0].srcAlpha”),GL_TEXTURE0,
                                     GL_CONSTANT,GL_CONSTANT);
glUniform1i(LOC(”dmp_TexEnv[0].combineRgb”),GL_REPLACE);
glUniform1i(LOC(”dmp_TexEnv[0].combineAlpha”),GL_REPLACE);
glUniform3i(LOC(”dmp_TexEnv[0].operandRgb”),GL_SRC_COLOR,
                                     GL_SRC_COLOR,GL_SRC_COLOR);
glUniform3i(LOC(”dmp_TexEnv[0].operandAlpha”),GL_SRC_ALPHA,
                                     GL_SRC_ALPHA,GL_SRC_ALPHA);
glUniform3i(LOC(”dmp_TexEnv[1].srcRgb”),GL_TEXTURE1,
                                     GL_PREVIOUS,GL_PREVIOUS);
glUniform3i(LOC(”dmp_TexEnv[1].srcAlpha”),GL_TEXTURE1,
                                     GL_PREVIOUS,GL_PREVIOUS);
glUniform1i(LOC(”dmp_TexEnv[1].combineRgb”),GL_REPLACE);
glUniform1i(LOC(”dmp_TexEnv[1].combineAlpha”),GL_REPLACE);
glUniform3i(LOC(”dmp_TexEnv[1].operandRgb”),GL_SRC_COLOR,
                                     GL_SRC_COLOR,GL_SRC_COLOR);
glUniform3i(LOC(”dmp_TexEnv[1].operandAlpha”),GL_SRC_ALPHA,
                                     GL_SRC_ALPHA,GL_SRC_ALPHA);
glUniform2i(LOC(”dmp_TexEnv[1].bufferInput”),GL_PREVIOUS,
                                     GL_PREVIOUS);
glUniform3i(LOC(”dmp_TexEnv[2].srcRgb”),GL_PREVIOUS_BUFFER_DMP,
                                     GL_PREVIOUS,GL_PREVIOUS);
glUniform3i(LOC(”dmp_TexEnv[2].srcAlpha”),GL_PREVIOUS_BUFFER_DMP,
                                     GL_PREVIOUS,GL_PREVIOUS);
glUniform1i(LOC(”dmp_TexEnv[2].combineRgb”),GL_MODULATE);
glUniform1i(LOC(”dmp_TexEnv[2].combineAlpha”),GL_MODULATE);
glUniform3i(LOC(”dmp_TexEnv[2].operandRgb”),GL_SRC_COLOR,
                                     GL_SRC_COLOR,GL_SRC_COLOR);
glUniform3i(LOC(”dmp_TexEnv[2].operandAlpha”),GL_SRC_ALPHA,
                                     GL_SRC_ALPHA,GL_SRC_ALPHA);
```

The two values specified for `dmp_TexEnv[i].bufferInput` are the color and alpha, respectively.

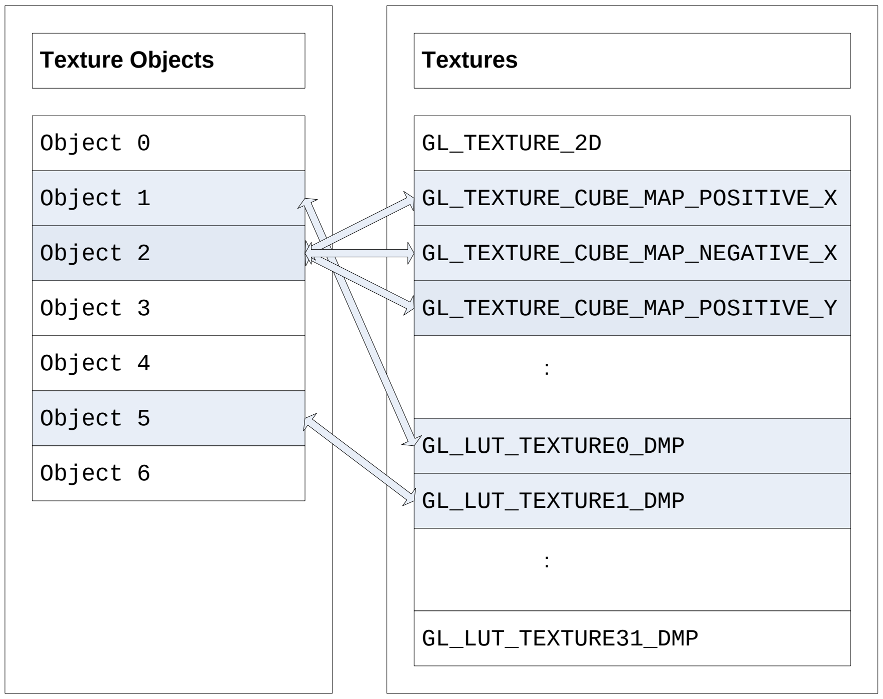
9 Texture Collections

Texture collections are a new feature provided by DMPGL 2.0. They do not exist in the OpenGL specification. Texture collections save texture bindings.

There are certain situations in which multiple bindings are required to render a single object. For example, if an object requires both a texture and a cube map, each must be bound. If different textures and cube maps are used for two objects A and B, two bindings would normally be required for both the texture and the cube map each time you switch objects. The texture collection feature can be used to simplify these bindings.

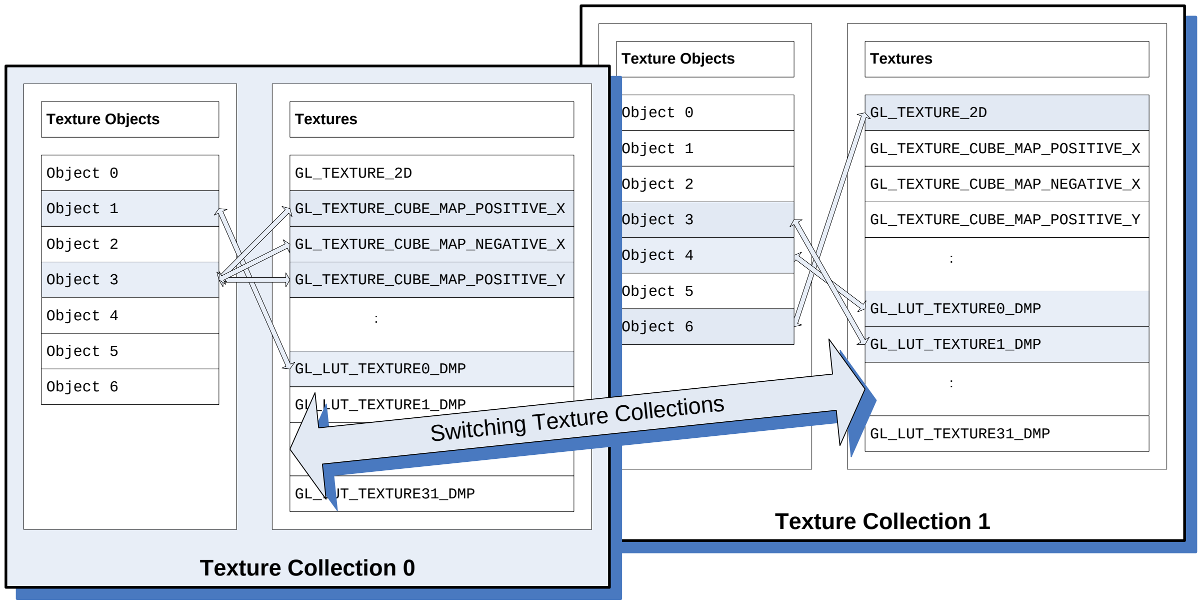
Texture collections recreate the state of a set of saved texture bindings each time you bind the texture collection. For example, if you bind a texture collection for object A and a texture collection for object B using different names, each time you bind the corresponding texture collection for an object, the texture bindings that were done for each object are recreated.

Figure 9-1: Texture Bindings Using OpenGL



As shown in the figure above, texture binding in OpenGL is done separately for each texture. With DMPGL 2.0, however, binding is done using texture collection objects. Unless you reconfigure the bindings in a texture collection, it will function just like OpenGL.

Figure 9-2: Binding Using Texture Collections



When texture collections are switched, the texture objects that were associated with that texture collection are all switched together.

When a new texture collection is created and bound, all bindings that have been done on the current textures are saved to the old texture collection, and all settings made thereafter are saved to the new texture collection. To restore the state saved in a texture collection, rebind the texture collection as shown in the figure above. This will recreate the texture bindings that were saved in the texture collection.

There are two ways to create a texture collection object. You can use either **`glGenTextures`** or **`glBindTexture`** to bind an unused name to `GL_TEXTURE_COLLECTION_DMP`. In the initial state of DMPGL 2.0, texture collection 0 (the default texture collection) is already bound.

Code 9-1: Creating a Texture Collection

```
GLint collection;
glGenTextures(1, &collection);
```

The texture collection namespace is shared with textures. The one exception is the name “0.” Although this name is shared by textures, cube-map textures, and texture collections as the default name for all three types, care must be taken with names other than “0” to avoid naming collisions between texture collections and texture objects. Code 9-2 below shows how to bind a texture collection. When a new texture collection is bound, the bindings for texture objects that were bound previously are saved in the previous texture collection.

Code 9-2: Binding a Texture Collection

```
glBindTexture(GL_TEXTURE_COLLECTION_DMP, collection);
```

To delete a texture collection object, use **`glDeleteTextures`**, just as you would for a texture object.

Code 9-3: Deleting a Texture Collection

```
glDeleteTextures(1, &collection);
```

If a texture collection object is deleted while it is still the currently bound object, the deletion process is deferred as long as the object is still in use. The actual deletion occurs when a different texture collection object is bound. The default texture collection (0) cannot be deleted.

© 2009–2012 Nintendo
CONFIDENTIAL

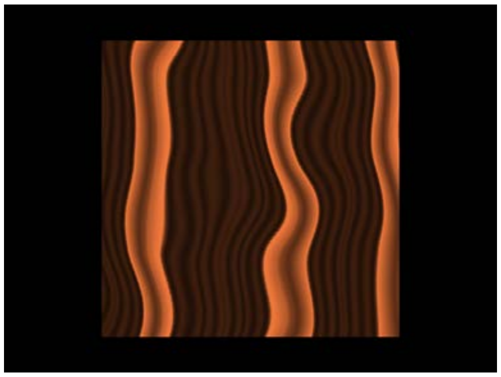
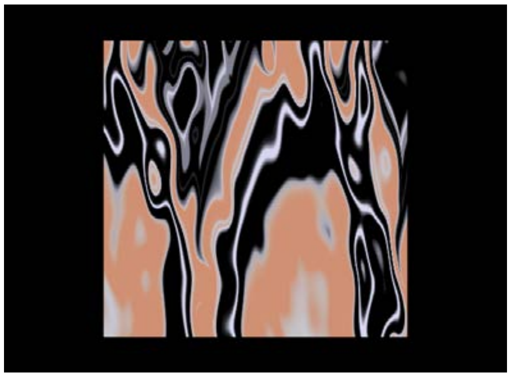
10 Procedural Textures

10.1 Overview of Procedural Textures

The procedural texture unit is located at the same position in the graphics pipeline as the texture units. NNProcedural textures are similar to standard textures in that they both determine the texel color at a certain pair of `(u, v)` texture coordinates.

However, they are different from traditional textures in that the texel color is determined through a procedural calculation instead of by accessing an image. The fact that texture images are not accessed has several benefits. For example, because competition for memory access is avoided, less memory is used, access to memory is sped up, and content size is reduced. Procedural textures work best on either completely systematic patterns or on patterns that have an element of randomness to them.

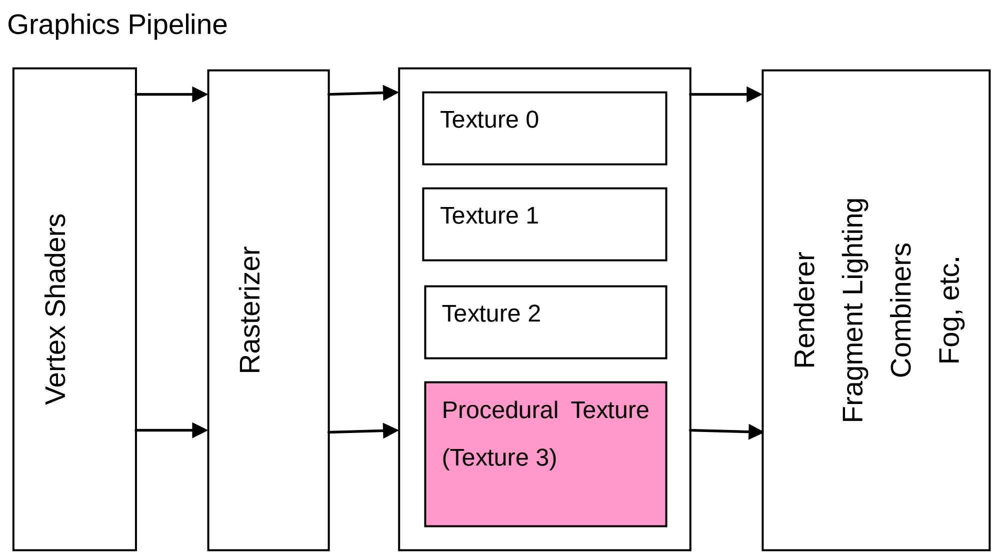
Figure 10-1: Rendered Results

Wood Grain	Stone
	

The procedural texture feature provided by DMPGL 2.0 is defined as the fourth texture unit (texture 3). The position of this step in the pipeline is shown in the conceptual diagram in Figure 10-2 below. Only texture 3 can be used as a procedural texture. Conversely, texture 3 cannot be used for anything other than procedural textures.

The parameters for procedural textures are set using the reserved uniform variables `dmp_Texture[3].*`.

Figure 10-2: Position of Procedural Textures in the Pipeline

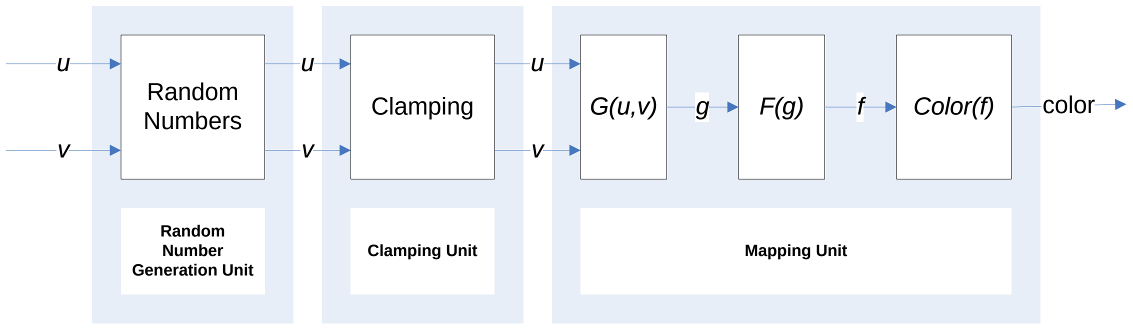


Note: With DMPGL 2.0, since mutually independent texture coordinates can only be provided for three of the four texture units, special methods can be used to provide coordinates to the texture units. For more information about these methods, refer to Chapter 7 Textures. To render the output of a procedural texture, that texture must be assigned as the source of a texture combiner. For information about how to configure texture combiners, see Chapter 8 Texture Combiners.

10.2 Elements Comprising the Procedural Texture Unit

The procedural texture unit is made up of three computational units. They are connected in series and are referred to as the random number generation unit, the clamping unit, and the mapping unit. The random number generation unit is the closest to the input. The connections between the units are shown in Figure 10-3 below. The random number generation unit provides some fluctuation to the `(u, v)` texture coordinates. The clamping unit determines the manner in which the pattern is mirrored and repeated. The mapping unit calculates the texel color based on the values of the `(u, v)` coordinates.

Figure 10-3: Internal Structure of the Procedural Texture Unit



When determining the image that is output by a procedural texture, the desired image can be obtained by setting the parameters according to the procedure listed below. This procedure is aimed at illustrating the required steps in an easy-to-understand manner. It does not take into account any of the restrictions imposed by the API.

1. Enable procedural textures.
2. Configure either RGBA-shared mode or independent alpha mode. This corresponds to the *G(u,v)* and *F(g)* settings in Figure 10-3 above.
3. Select the basic shape. This corresponds to the choice of *G(u,v)* in Figure 10-3.
4. Set the basic color. Colors are set as color lookup tables. This corresponds to the *Color(f)* setting in Figure 10-3.
5. Configure the relationship between the basic shape and the color lookup table. This determines how the basic shape from step 3 maps to the color lookup table from step 4. This corresponds to the *F(g)* setting in Figure 10-3.
6. Select the random-number parameters. If random numbers are necessary, enable them and set the amount of influence that the random numbers have in your procedural texture. If random numbers aren't needed, disable them. This corresponds to the random number generation unit in Figure 10-3.
7. Configure the settings for repetition and mirroring. This corresponds to the clamp settings in Figure 10-3.

This chapter describes the process of generating procedural textures in this order. If you can grasp these parameters, you will also have no problem understanding how to use the non-standard parameters.

10.3 Enabling and Disabling Procedural Textures

This explains how to start using procedural textures. All you have to do is set the uniform variable as shown below.

Code 10-1: Enabling Procedural Textures

```
glUniform1i(LOC(”dmp_Texture[3].samplerType”),GL_TEXTURE_PROCEDURAL_DMP);
```

To stop using procedural textures, disable them using the following syntax:

Code 10-2: Disabling Procedural Textures

```
glUniform1i(LOC(”dmp_Texture[3].samplerType”),GL_FALSE);
```

It is not possible to set `GL_TEXTURE_2D` for `dmp_Texture[3].samplerType`, unlike the other texture units (0, 1, and 2).

10.4 Choosing RGBA-Shared Mode or Independent Alpha Mode

Choosing RGBA-shared mode or independent alpha mode amounts to choosing whether the alpha component is subordinate to the color, or whether the alpha component is specified independently. In the case of independent alpha mode, the **`F`** and **`G`** functions shown below must each be set twice. If you just want to try producing an output image, we recommend using RGBA-shared mode, since it can be configured more easily.

Figure 10-4: Mapping Unit in RGBA-Shared Mode

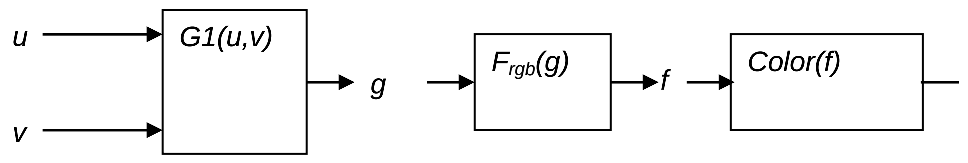
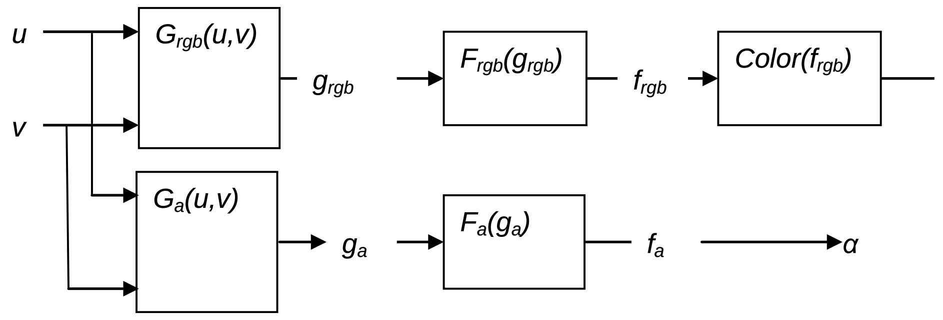


Figure 10-5: Mapping Unit in Independent Alpha Mode



The syntax to specify RGBA-shared mode or independent alpha mode is shown below.

Code 10-3: RGBA-Shared Mode

```
glUniform1i(LOC(”dmp_Texture[3].ptAlphaSeparate”),GL_FALSE);
```

Code 10-4: Independent Alpha Mode

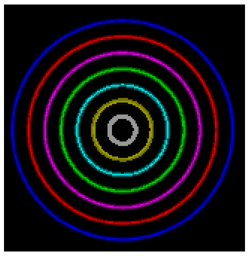
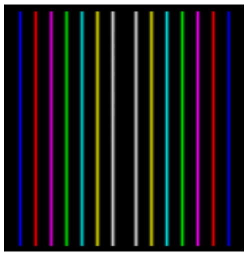
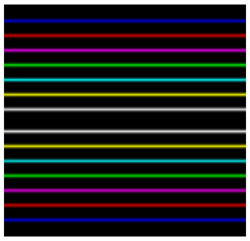
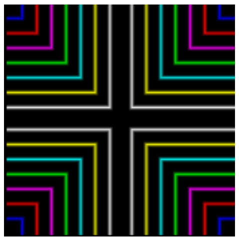
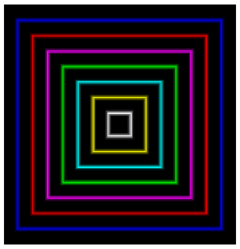
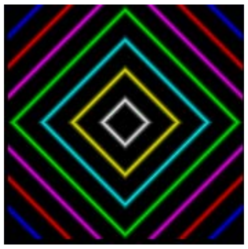
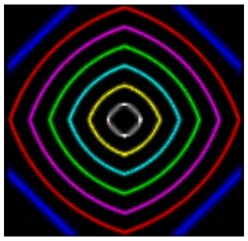
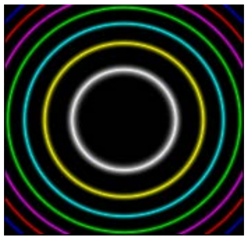
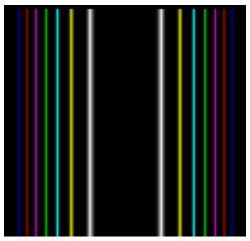
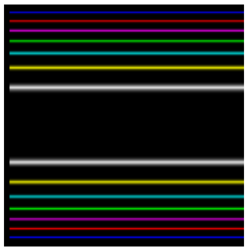
```
glUniform1i(LOC(”dmp_Texture[3].ptAlphaSeparate”),GL_TRUE);
```

10.5 Selecting the Basic Shape

Among the various elements that make up procedural textures, the **`G`** function is the one that determines the basic shape of the output image.

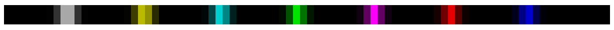
The basic shape is defined by a function that maps from a given set of `(u, v)` coordinates in the range [0.0, 1.0] to a single value `g`. The figure below illustrates the patterns of the basic shapes, with arbitrary colors assigned to them. The `(u, v)` coordinates are both in the range [-1.0, 1.0], and the clamping is set to “mirrored repeat” mode. (This is described in detail later in the document.)

Figure 10-6: Patterns Formed by the Basic Shapes

The 10 Basic Shapes That Can Be Chosen for Procedural Textures			
 ADDSQRT2	 U	 V	 MIN
 MAX	 ADD	 RMAX	 ADD2
 U2	 V2		

Here, *g* has a value in the range [0.0, 1.0], and the assignment of colors in Figure 10-6 follows the pattern shown below.

Figure 10-7: Relationship Between Scalar Values and Colors


0.0 0.5 1.0

Choose the basic shape from Figure 10-6 that most closely resembles the image you want to render in your procedural texture. If you want to render a random pattern, choose the shape whose non-random elements are closest to what you're looking for. For example, to render a wood-grain texture with parallel grain lines, you might choose the `U` or `V` patterns. Likewise, to render a texture that resembles tree stump rings, you might choose the `ADDSQRT` pattern.

To set the basic shape in the procedural texture unit, set the reserved uniform `dmp_Texture[3].ptRgbMap` to the constant whose name matches the basic pattern you want to use. If you're using independent alpha mode, you will also need to set the **G** function for the alpha component by configuring `dmp_Texture[3].ptAlphaMap` separately.

The constants are defined using the syntax `GL_PROCTEX_NameOfBasicShape_DMP`. For example, to select the `U` pattern (vertical stripes), use the following syntax:

Code 10-5: Setting the `G(u,v)` Mapping Function to the "U" Pattern

```
glUniform1i(LOC(”dmp_Texture[3].ptRgbMap”), GL_PROCTEX_U_DMP);
glUniform1i(LOC(”dmp_Texture[3].ptAlphaMap”), GL_PROCTEX_U_DMP);
```

If using RGBA-shared mode, any settings assigned to the reserved uniform for the alpha component (`dmp_Texture[3].ptAlphaMap`) are ignored.

10.6 Configuring the Color Lookup Tables

Use the color lookup tables to set the basic colors of the pattern generated by the procedural texture. The R, G, B, and A components of the color are set individually using **`glTexImage1D`**. The *`internalformat`* and *`format`* arguments of the **`glTexImage1D`** function can only be set to `GL_LUMINANCEF_DMP`, and the *`type`* argument can only be set to `GL_FLOAT`. Refer to the *DMPGL Specifications* for the detailed definitions. In the example below, the color is set by defining the functions **`func_R`**, **`func_G`**, **`func_B`**, and **`func_A`** over the domain [0, 1.0]. These functions yield the individual components of the desired color.

Code 10-6: Configuring the Color Lookup Tables for the R, G, B, and A Components

```
  GLfloat func_R(GLfloat x /* x must be in the range [0.0f , 1.0f] */ ){
        return /* Appropriate value for the R component of the F function
*/;
  }
  GLfloat func_G(GLfloat x /* x must be in the range [0.0f , 1.0f] */ ){
        return /* Appropriate value for the G component of the F function
*/;
  }
  GLfloat func_B(GLfloat x /* x must be in the range [0.0f , 1.0f] */ ){
        return /* Appropriate value for the B component of the F function
*/;
  }
  GLfloat func_A(GLfloat x /* x must be in the range [0.0f , 1.0f] */ ){
        return /* Appropriate value for the A component of the F function
*/;
  }

  // The size (SIZE) must be a power of two no greater than 128
  #define SIZE 128
  GLint ref_R = ...; // Lookup table number [0..31]
  GLint ref_G = ...; // Lookup table number [0..31]
  GLint ref_B = ...; // Lookup table number [0..31]
  GLint ref_A = ...; // Lookup table number [0..31]
  GLfloat map_color_R[SIZE+256]; // Lookup table
  GLfloat map_color_G[SIZE+256]; // Lookup table
  GLfloat map_color_B[SIZE+256]; // Lookup table
  GLfloat map_color_A[SIZE+256]; // Lookup table
  GLint texID[4];
  for(int i=0; i< SIZE; i++){ // Set the function values
  map_color_R[i] = func_R((GLfloat)i / (GLfloat)SIZE);
  map_color_G[i] = func_G((GLfloat)i / (GLfloat)SIZE);
  map_color_B[i] = func_B((GLfloat)i / (GLfloat)SIZE);
  map_color_A[i] = func_A((GLfloat)i / (GLfloat)SIZE);
  };
  for(int i=0; i< SIZE; i++){ // Set the deltas of the function values
  map_color_R[i+256] = func_R((i+1) / (GLfloat)SIZE)- func_R(i /
(GLfloat)SIZE);
  map_color_G[i+256] = func_G((i+1) / (GLfloat)SIZE)- func_G(i /
(GLfloat)SIZE);
  map_color_B[i+256] = func_B((i+1) / (GLfloat)SIZE)- func_B(i /
(GLfloat)SIZE);
```

```
  map_color_A[i+256] = func_A((i+1) / (GLfloat)SIZE)- func_A(i /
(GLfloat)SIZE);
  };
  glGenTextures(4,texID);
  glBindTexture(LUT_TEXTURE0_DMP + ref_R, texID[0]);
  glTexImage1D(GL_LUT_TEXTURE0_DMP + ref_R, 0,
  GL_LUMINANCEF_DMP, SIZE+256, 0, GL_LUMINANCEF_DMP, GL_FLOAT, map_color_R);
  glBindTexture(GL_LUT_TEXTURE0_DMP + ref_G, texID[1]);
  glTexImage1D(GL_LUT_TEXTURE0_DMP + ref_G, 0,
  GL_LUMINANCEF_DMP, SIZE+256, 0, GL_LUMINANCEF_DMP, GL_FLOAT, map_color_G);
  glBindTexture(GL_LUT_TEXTURE0_DMP + ref_B, texID[2]);
  glTexImage1D(GL_LUT_TEXTURE0_DMP + ref_B, 0,
  GL_LUMINANCEF_DMP, SIZE+256, 0, GL_LUMINANCEF_DMP, GL_FLOAT, map_color_B);
  glBindTexture(GL_LUT_TEXTURE0_DMP + ref_A, texID[3]);
  glTexImage1D(GL_LUT_TEXTURE0_DMP + ref_A, 0,
  GL_LUMINANCEF_DMP, SIZE+256, 0, GL_LUMINANCEF_DMP, GL_FLOAT, map_color_A);

  // To allow the reserved fragment shader to access the four lookup tables
we've
  // created, we set each reserved uniform to the number of the
corresponding
  // lookup table

  glUniform1i(LOC("dmp_Texture[3].ptSamplerR"), ref_R);
  glUniform1i(LOC("dmp_Texture[3].ptSamplerG"), ref_G);
  glUniform1i(LOC("dmp_Texture[3].ptSamplerB"), ref_B);
  glUniform1i(LOC("dmp_Texture[3].ptSamplerA"), ref_A);
  // Specify ptTexOffset
  glUniform1i(LOC("dmp_Texture[3].ptTexOffset"), 0);
  // Specify ptTexWidth
  glUniform1i(LOC("dmp_Texture[3].ptTexWidth"), SIZE);
```

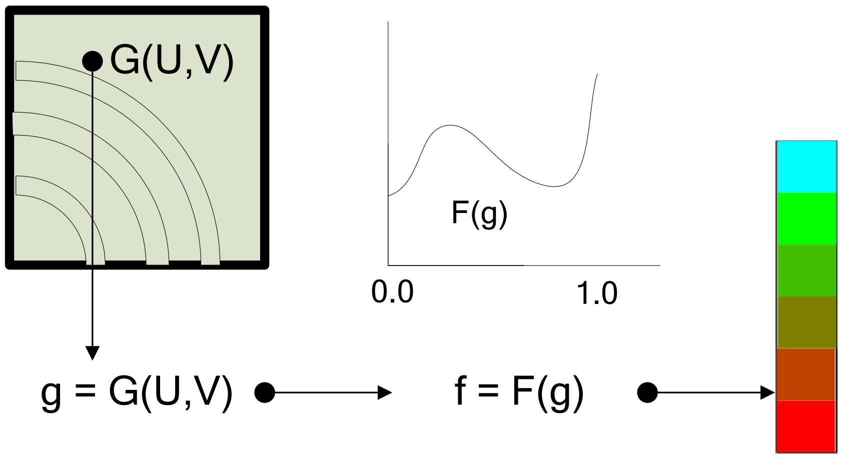
In RGBA-shared mode, the color lookup table set to `ptSamplerA` is referenced in the same way as the other components of the color. In independent alpha mode, the `ptSamplerA` setting is ignored.

10.7 Establishing the Relationship between the Basic Shape and the Color Lookup Tables

The relationship between the **`G`** function that was selected as the basic shape and the configured color lookup tables is determined by the **`F(g)`** function. The conceptual diagram in Figure 10-8 below illustrates the relationship in a visual form. First, the **`G`** function takes a pair of `(u, v)` coordinates and determines a single `g` value from them. The **`F`** function then maps the `g` value to another value, `f`.

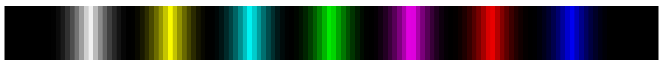
Finally, `f` is used to access the color lookup tables and look up the color. By cleverly defining your **`F(g)`** function, you can output images with different appearances, even given the same basic shape and the same color lookup tables as inputs. The domain and the range of the **`F(g)`** function are both defined as [0.0, 1.0].

Figure 10-8: Determining the Relationship between the `G` Function and the Color Lookup Table



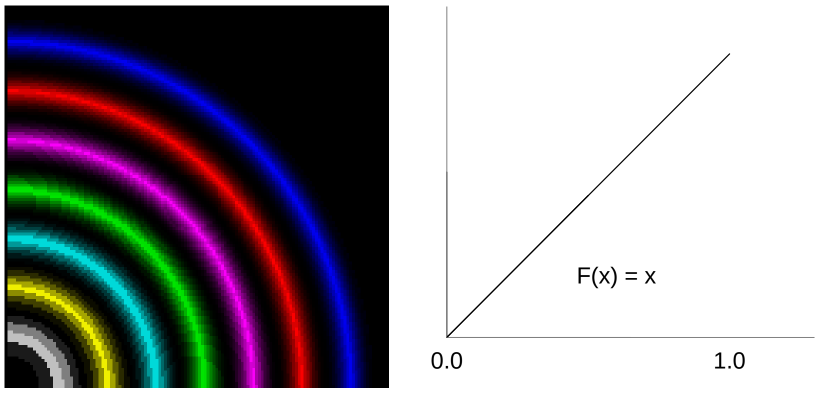
For example, let's assume that you've chosen `ADDSQRT2` as the basic shape and configured your color lookup tables as follows:

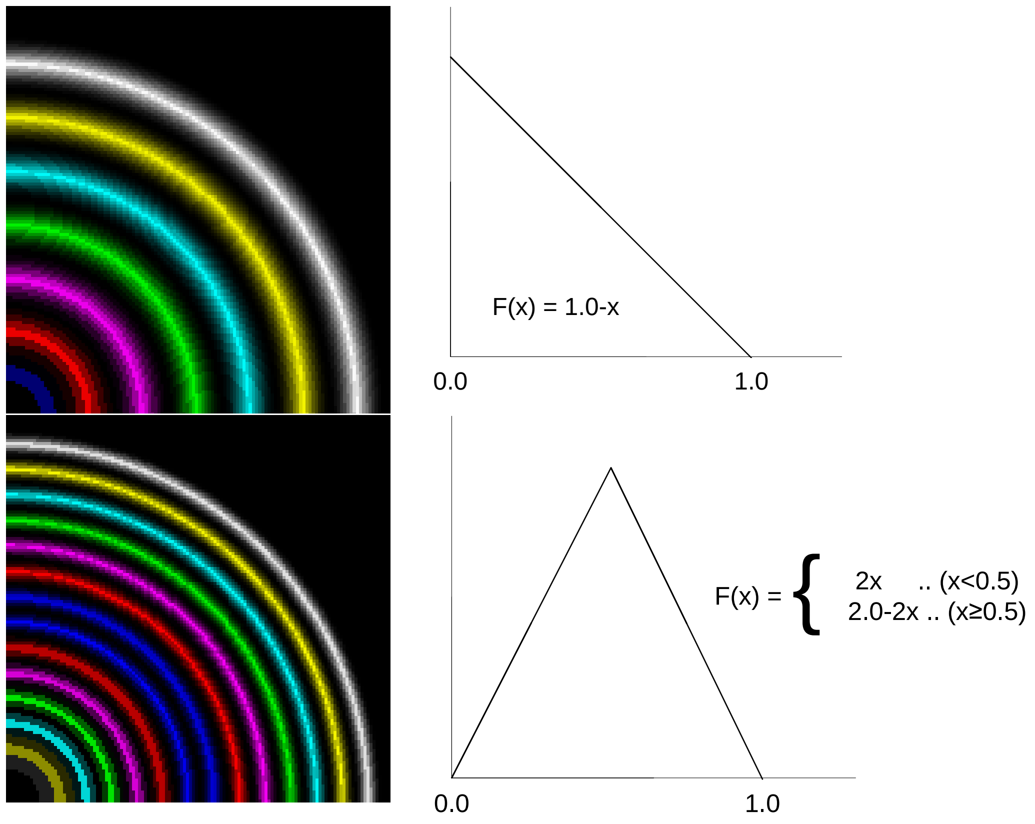
Figure 10-9: Sample Color Lookup Table



Given the color lookup tables in Figure 10-9, the images in the figure below show the effects of several possible definitions for the **`F`** function.

Figure 10-10: Effects of the `F(g)` Function





In RGBA-shared mode, a single `F` function determines the image that is output, but in independent alpha mode, an independent `F` function must be defined for the alpha component.

In the following example shown in Code 10-7, we create two `GLfloat` arrays of size 256 (one for the `F` function for the RGB components, and the other for the `F` function for the alpha component), then create the `F` function lookup tables. These arrays must be of size 256; the first 128 elements are used to store the function's values, and the last 128 elements are used to store the deltas of the function's values. This example assumes that we've already defined the functions **`func_RGB(x)`** and **`func_A(x)`**, which calculate the values we want to set in our tables. The integer variables `ref_RGB` and `ref_A` used in the example are identification numbers for the lookup tables. They are used when accessing the lookup tables from a reserved uniform, and can be set to any value between 0 and 31 (inclusive).

Code 10-7: Registering the `F(g)` Mapping Function

```
 GLfloat func_RGB(GLfloat x /* x must be in the range [0.0f , 1.0f] */ ){
      return /* Appropriate value for the F_RGB function  */;
 }
 GLfloat func_A(GLfloat x /* x must be in the range [0.0f , 1.0f] */ ){
      return /* Appropriate value for the F_A function  */;
 }

 // Main body of program
 GLint ref_RGB = ...; // Lookup table number [0..31]
 GLint ref_A = ...; // Lookup table number [0..31]
 GLfloat map_F_RGB[256];
 GLfloat map_F_A[256];
 GLint texID[2];
 for(int i=0;i<128;i++) // Set the function values
 map_F_RGB[i] = func_RGB(i/128.f);
 for(int i=0;i<128;i++) // Set the deltas of the function values
 map_F_RGB[i+128] = func_RGB((i+1)/128.f) – func_RGB(i/128.f);
 for(int i=0;i<128;i++) // Set the function values.
 map_F_A[i] = func_A(i/128.f);
 for(int i=0;i<128;i++) // Set the deltas of the function values
 map_F_A[i+128] = func_A((i+1)/128.f) – func_A(i/128.f);
 glGenTextures(2,texID);
 glBindTexture(GL_LUT_TEXTURE0_DMP + ref_RGB, texID[0]);
 glTexImage1D(GL_LUT_TEXTURE0_DMP + ref_RGB, 0,
 GL_LUMINANCEF_DMP, 256, 0, GL_LUMINANCEF_DMP, GL_FLOAT, map_F_RGB);
 glBindTexture(GL_LUT_TEXTURE0_DMP + ref_A, texID[1]);
 glTexImage1D(GL_LUT_TEXTURE0_DMP + ref_A, 0,
 GL_LUMINANCEF_DMP, 256, 0, GL_LUMINANCEF_DMP, GL_FLOAT, map_F_A);

 // To allow the reserved fragment shader to access the two lookup tables
we've
 // created, we set each reserved uniform to the number of the
corresponding
 // lookup table
 glUniform1i(LOC("dmp_Texture[3].ptSamplerRgbMap"), ref_RGB);
 glUniform1i(LOC("dmp_Texture[3].ptSamplerAlphaMap"), ref_A);
```

In RGBA-shared mode, there is no need for the `map_F_A[]` array used in the sample code. In independent alpha mode, both lookup tables are required (one for RGB and one for alpha).

The range of values that can be set for the **`F`** function is restricted to [0.0, 1.0]. The range of the values that can be set for the deltas is restricted to [-1.0, 1.0]. `GLfloat` is the only acceptable type for the array passed to **`glTexImage1D`**, `GL_FLOAT` is the only value that can be specified for the *`type`* argument, and `GL_LUMINANCEF_DMP` is the only value that can be specified for the *`format`* argument. To use a configured table as the **`F`** function for the RGB components of the procedural texture unit, set `dmp_Texture[3].ptSamplerRgbMap` to the number of the lookup table. To set the **`F`** function for the alpha component, set `dmp_Texture[3].ptSamplerAlphaMap` to the number of the lookup table. In the sample code, these table numbers are `ref_RGB` and `ref_A`. Note that these numbers aren't the name of the lookup table that was bound (`texID[]`).

10.8 Configuring the Noise

Once the relationship between the basic shape and the color lookup tables has been established, you'll want to provide some fluctuation using random numbers. In this section, this fluctuation is called *noise*. To enable noise, set the reserved uniforms as shown below.

Code 10-8: Enabling Noise

```
  glUniform1i(LOC(”dmp_Texture[3].ptNoiseEnable”),GL_TRUE );
```

Code 10-9: Disabling Noise

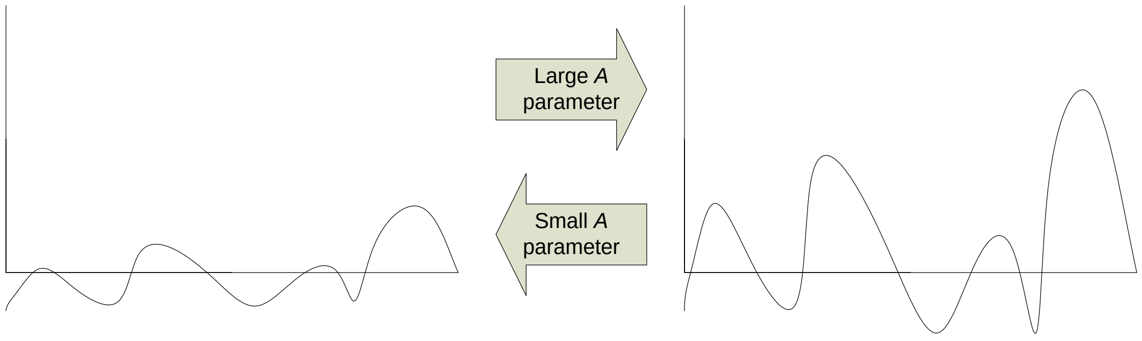
```
  glUniform1i(LOC(”dmp_Texture[3].ptNoiseEnable”),GL_FALSE );
```

Noise has three parameters (*F*, *A*, and *P*), and can be applied to change the *u* and *v* coordinates independently. The three parameters *F*, *A*, and *P* are set by reserved uniforms as shown below.

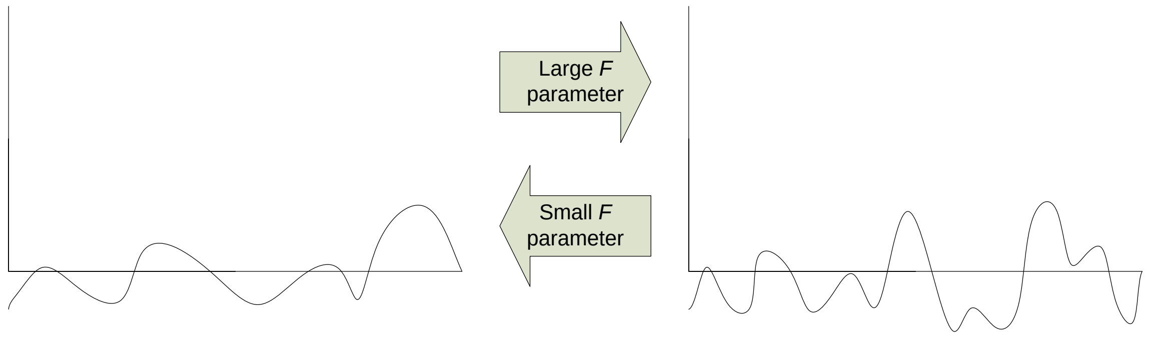
Code 10-10 Sample Settings for Noise-Control Parameters

```
  glUniform3f(LOC(”dmp_Texture[3].ptNoiseU”), 0.1, 0.2,
0.3);//F=0.1,P=0.2,A=0.3
  glUniform3f(LOC(”dmp_Texture[3].ptNoiseV”), 0.2, 0.4,
0.5);//F=0.2,P=0.4,A=0.5
```

The *A* parameter (*amplitude*) defines the strength of the fluctuation. Larger values for the *A* parameter cause the fluctuation to have more of an effect, meaning that the resulting image will look less like the original basic shape.



The *F* parameter (*frequency*) defines the fineness of the fluctuation. Larger values for the *F* parameter cause more sudden changes in fluctuation, whereas smaller values for the *F* parameter cause the fluctuation to change more gradually.



The *P* parameter (*phase*) shifts the starting position for the fluctuation. The way in which the function fluctuates can be changed by altering the *P* parameter.

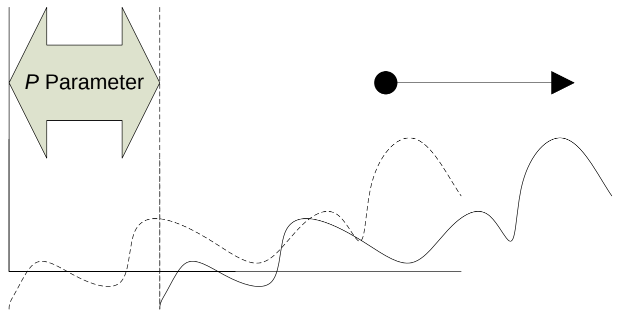
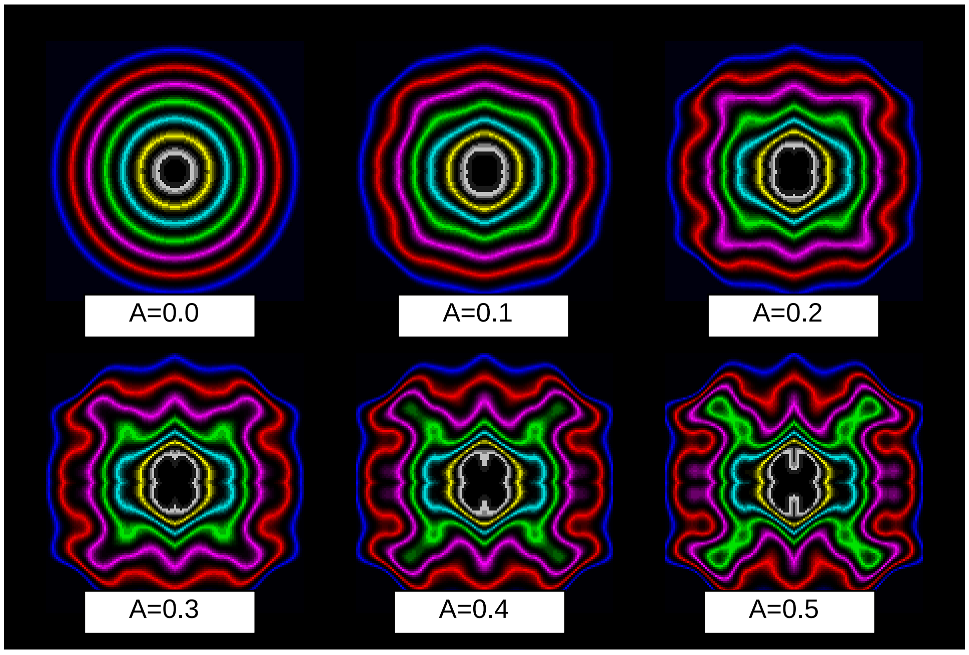


Figure 10-11 below shows six renderings; the settings used to create these renderings are identical except for the values for the *A* parameter. Without the random-number component, the settings for all these images would have produced perfect concentric circles. Therefore, any visible departure from concentric circles is caused by the random numbers.

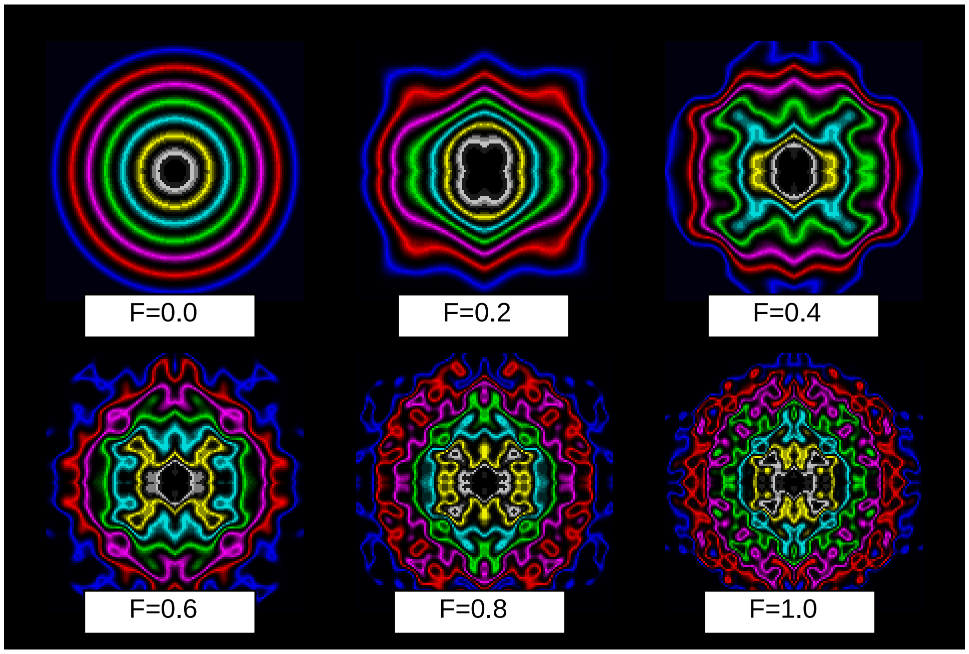
Figure 10-11: Relationship Between *A* Parameter and Shape



The parameters other than *A* were kept constant; *F* was set to 0.3, and *P* was set to 0.0. The higher the *A* value is set, the more prominent the waves of fluctuation become. When *A* is set to 0.0, no fluctuation occurs.

Figure 10-12 below shows six renderings; the settings used to create these renderings are identical except for the values of the *F* parameter. Without the random-number component, the settings for all these images would have produced perfect concentric circles. Therefore, any visible departure from concentric circles is caused by the random numbers.

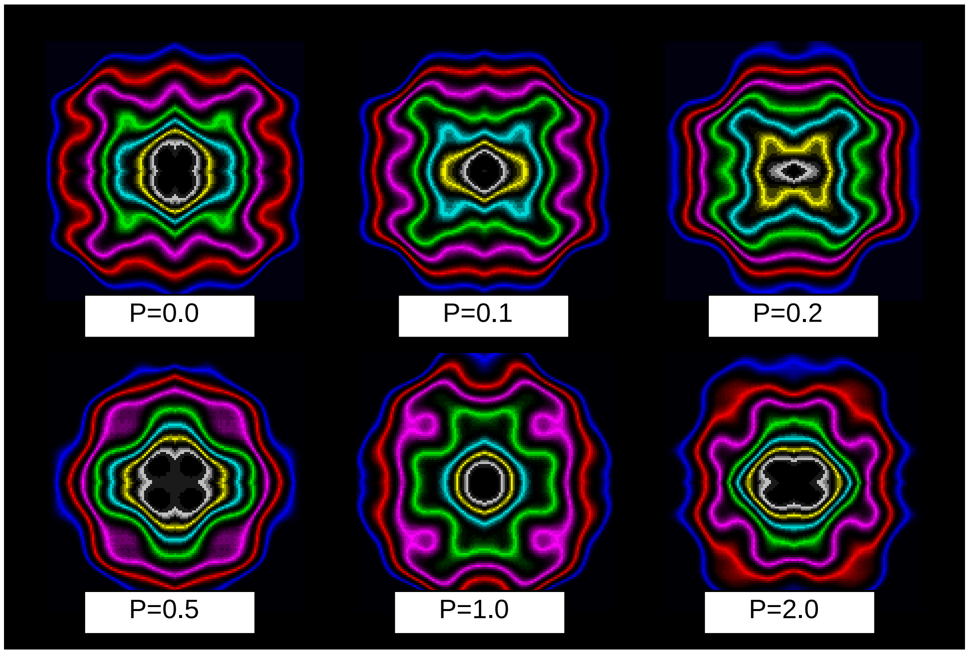
Figure 10-12: Relationship Between *F* Parameter and Shape



All parameters other than *F* were kept constant; for both the u and v components *P* was set to 0.0 and *A* was set to 0.3. The greater the value of *F*, the greater the frequency becomes, resulting in a more finely detailed fluctuation pattern. When *F* is set to 0.0, no fluctuation occurs.

Figure 10-13 below shows six renderings; the settings used to create these renderings are identical except for the values of the *P* parameter. Without the random-number component, the settings for all these images would have produced perfect concentric circles. Therefore, any visible departure from concentric circles is caused by the random numbers.

Figure 10-13: Relationship Between *P* Parameter and Shape



The parameters other than *P* were kept constant; *F* was set to 0.3, and *A* was set to 0.3. The only difference caused by changing *P* is the starting point for the fluctuation. This should be used when you don't always want to start out with the same random number; for example, change the value of *P* when you want to change the fluctuation pattern used in animations.

If you are changing *P* to produce an animation and set *P* equal to a large value, small variations in *P* will cease to affect the fluctuation. Because this phenomenon is caused by the accuracy of calculations in the hardware, if you change *P*—by adding a fixed value to it, for example—and then run an animation using the changed noise, you will need to restore *P* to a small value some time before it gets large. As long as *F* and *P* are positive numbers, you can get the same fluctuation results when *F*×*P* is a multiple of 16 and when 0 is specified for *P*. This allows you to maintain an animation's continuity by reverting *P* to 0 when you calculate *F*×*P*. However, the fluctuation may not be the same when *F*×*P* is 16 and when 0 is specified for *P* if a large value has been specified for *F*.

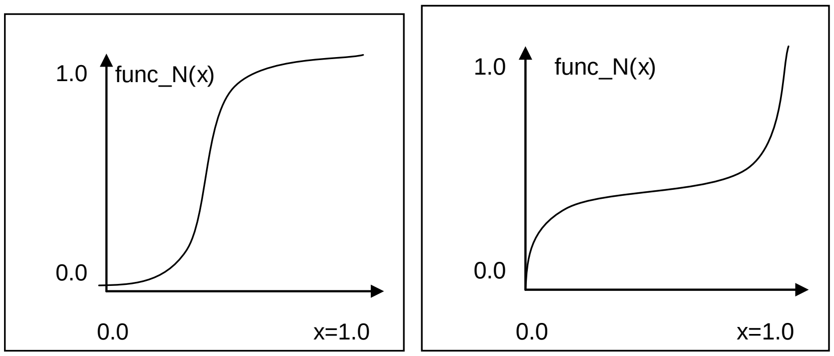
Aside from the *A*, *F*, and *P* parameters, you also need to configure a noise continuity or *noise modulation* function. The noise modulation function attempts to take the noise values that are calculated discretely, and convert them to natural, continuous values. A function that smoothes out the ends of the domain [0.0, 1.0] is therefore ideal.

In Code 10-11 below, we create a table for the noise modulation function by preparing a single `GLfloat` array of size 256. This array must be of size 256, the first 128 elements are used to store the function's values, and the last 128 elements are used to store the deltas of the function's values. This code sample assumes that we've already defined the function **`func_N(x)`**, which calculates the values we want to set in our table. The integer variable `ref_Noise` used in the sample is the lookup table's identification number. It is used when accessing lookup tables from a reserved uniform, and can be set to any value between 0 and 31 (inclusive).

Code 10-11: Configuring the Noise Modulation Lookup Table

```
 GLfloat func_N(GLfloat x /* x must be in the range [0.0f , 1.0f] */ ){
      return /* Appropriate value for the noise modulation function */;
 }

 GLfloat map_Noise[256];
 GLint ref_Noise = ...; // Arbitrary value in the range [0,31]
 GLint texID[1];
 for(int i=0; i<128; i++) // Set values for the function
 map_Noise[i] = func_N(i/128.f);
 for(int i=0; i<128; i++) // Set the deltas between the function's values
 map_Noise[i+128] = func_N((i+1)/128.f) – func_N(i/128.f);
 glGenTextures(1, texID);
 glBindTexture(GL_LUT_TEXTURE0_DMP + ref_Noise, texID[0]);
 glTexImage1D(GL_LUT_TEXTURE0_DMP + ref_Noise, 0,
 GL_LUMINANCEF_DMP, 256, 0, GL_LUMINANCEF_DMP, GL_FLOAT, map_Noise);

 // To allow the reserved fragment shader to use the lookup table we've
created,
 // we set the reserved uniform to ref_Noise.
 glUniform1i(LOC("dmp_Texture[3].ptSamplerNoiseMap"), ref_Noise);
```

Figure 10-14 Sample Noise Modulation Functions



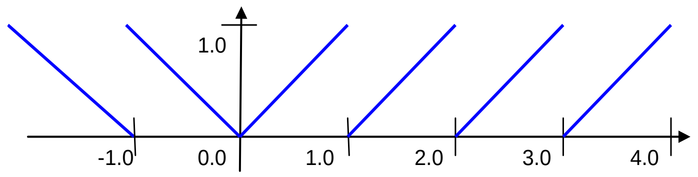
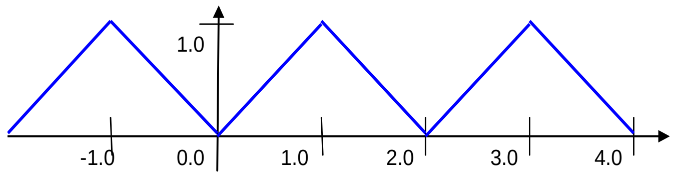
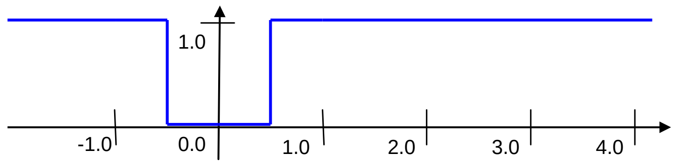
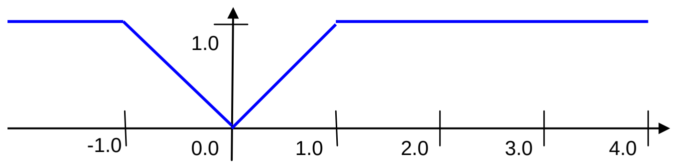
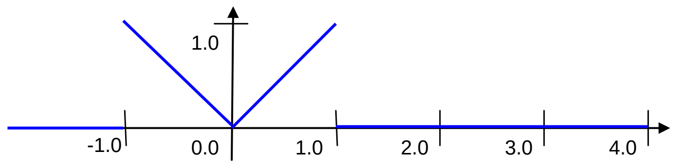
The graphs in Figure 10-14 above take **`func_N(x)`** as the example noise modulation function. If a function like the one shown to the left is provided as the noise modulation function, the noise will be concentrated around the value 0.5. With the noise modulation function shown on the right, the noise will tend to be scattered at the values 0.0 and 1.0. The recommended function is $3x^2 - 2x^3$, which is shown on the left in Figure 10-14. This will give the most natural appearance under most circumstances. However, the reference images that appear within this chapter simply use the function $f(x) = x$.

10.9 Clamp Settings

The clamping unit handles two operations: shift calculation and clamp calculation. Clamping is nearly identical to the normal wrapping modes for textures, except that features such as `GL_PULSE_DMP` and `GL_CLAMP_TO_ZERO_DMP` are added. The shift calculation occurs first, and clamping is then applied to the shifted coordinates. For the sake of convenience in describing these operations, we'll describe the clamp calculation first, and then move on to the shift calculation.

The clamp calculation converts coordinates outside the range [0.0, 1.0] to coordinates within this range. The table below shows the relationship between the various clamp modes and the resulting coordinates from each mode.

Table 10-1: Overview of Clamp Parameters

Clamp Mode	Clamped Coordinate
`GL_SYMMETRICAL_REPEAT_DMP`	
`GL_MIRRORED_REPEAT`	
`GL_PULSE_DMP`	
`GL_CLAMP_TO_EDGE`	
`GL_CLAMP_TO_ZERO_DMP`	

`GL_SYMMETRICAL_REPEAT_DMP` repeats the same image along a grid. `GL_MIRRORED_REPEAT` flips (mirrors) the image around each even number on the axis. `GL_PULSE_DMP` picks up the pixel at the edge of the texture that is closest to the pixel being rendered. `GL_CLAMP_TO_EDGE` references the image within the texture when its value falls within the range [-1.0, 1.0]; when the value falls outside of that range, it references the pixels at the edge of the texture. `GL_CLAMP_TO_ZERO_DMP` references the texture image when the value falls within the range (-1.0, 1.0); when the value falls outside of that range, it references the image at coordinate 0.

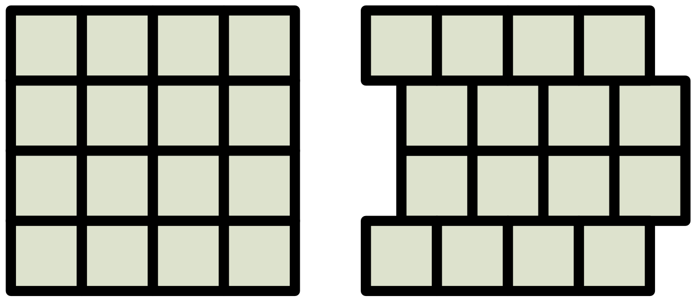
For example, to use `GL_SYMMETRICAL_REPEAT_DMP` for the clamp calculation, use the API to configure the following settings.

Code 10-12: Clamp Parameter Configuration

```
GLuint uclamp = GL_SYMMETRICAL_REPEAT_DMP;
GLuint vclamp = GL_SYMMETRICAL_REPEAT_DMP;
glUniform1i(LOC(”dmp_Texture[3].ptClampU”),uclamp);
glUniform1i(LOC(”dmp_Texture[3].ptClampU”),vclamp);
```

Bricklayers will often shift progressive rows of bricks by half a brick length. The shift calculation makes it possible to express this type of pattern. Here, a *block* refers to a collection of texels whose coordinates share the same integer portion. A *shift operation* refers to the act of shifting a block. The *shift width* refers to the distance a block is shifted. The conceptual diagram in Figure 10-15 below illustrates what a set of blocks looks like with and without shift calculation. Shift calculation is useful for alleviating visual monotony. The shift mode is set independently for the u and v coordinates. Shifting of blocks in the U direction is determined by the integer portion of each block’s v-coordinate. Shifting of blocks in the V direction is determined by the integer portion of each block’s u-coordinate.

Figure 10-15 Repeating Clamps Without (Left) and with (Right) Shift Calculation



Whether a given block is shifted by shift calculation depends on whether the shift mode is set to ODD or EVEN. The conceptual diagrams in Figure 10-16 and Figure 10-17 below illustrate shifting in both ODD and EVEN modes. In the figures, the u-coordinates are shifted based on the integer portion of the v-coordinates.

Figure 10-16: Shifting in ODD Mode

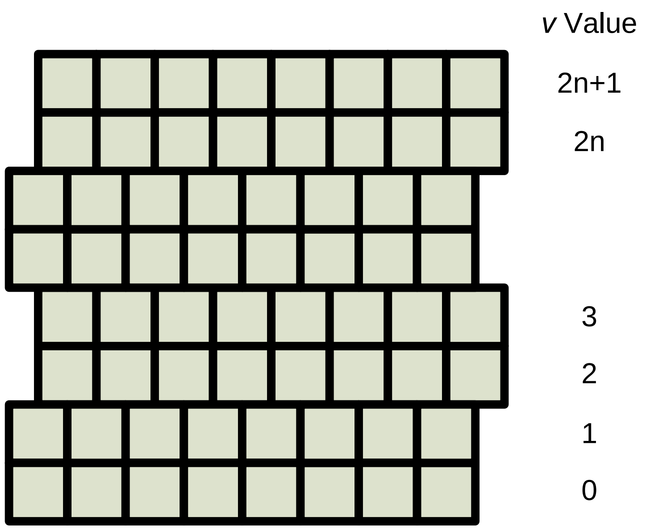
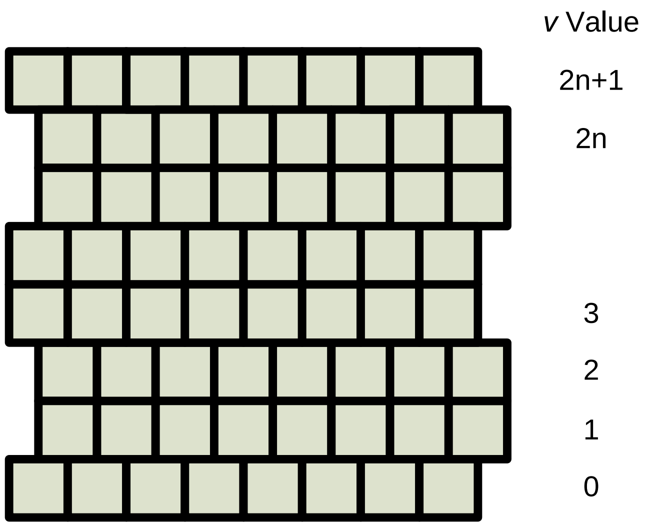
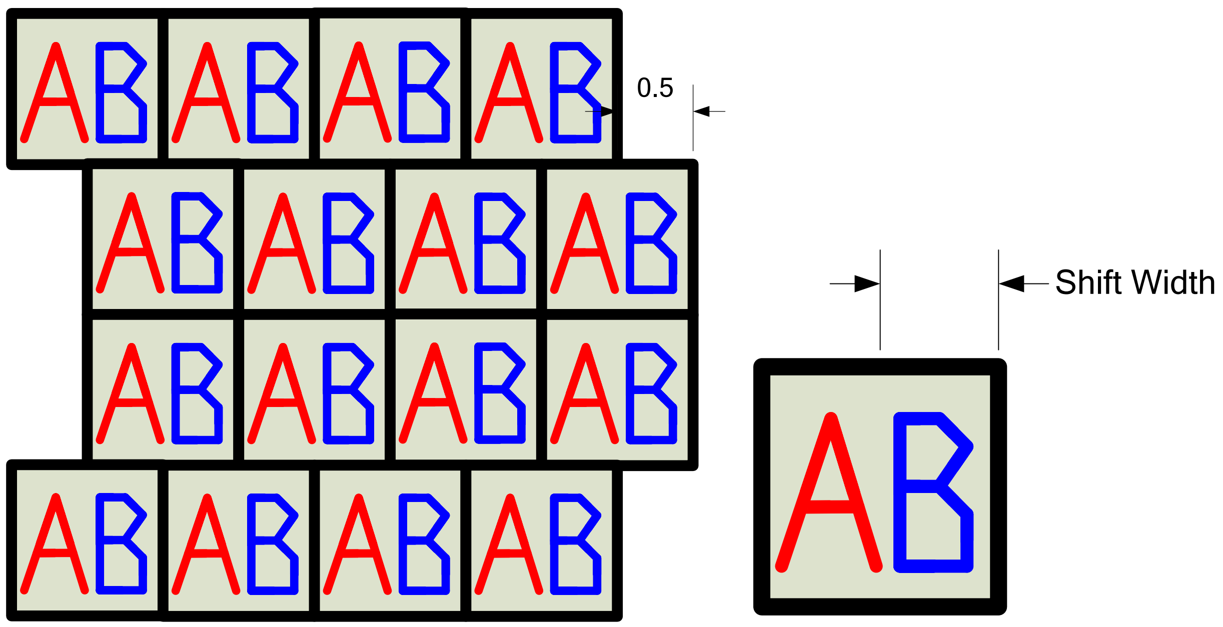


Figure 10-17: Shifting in EVEN Mode



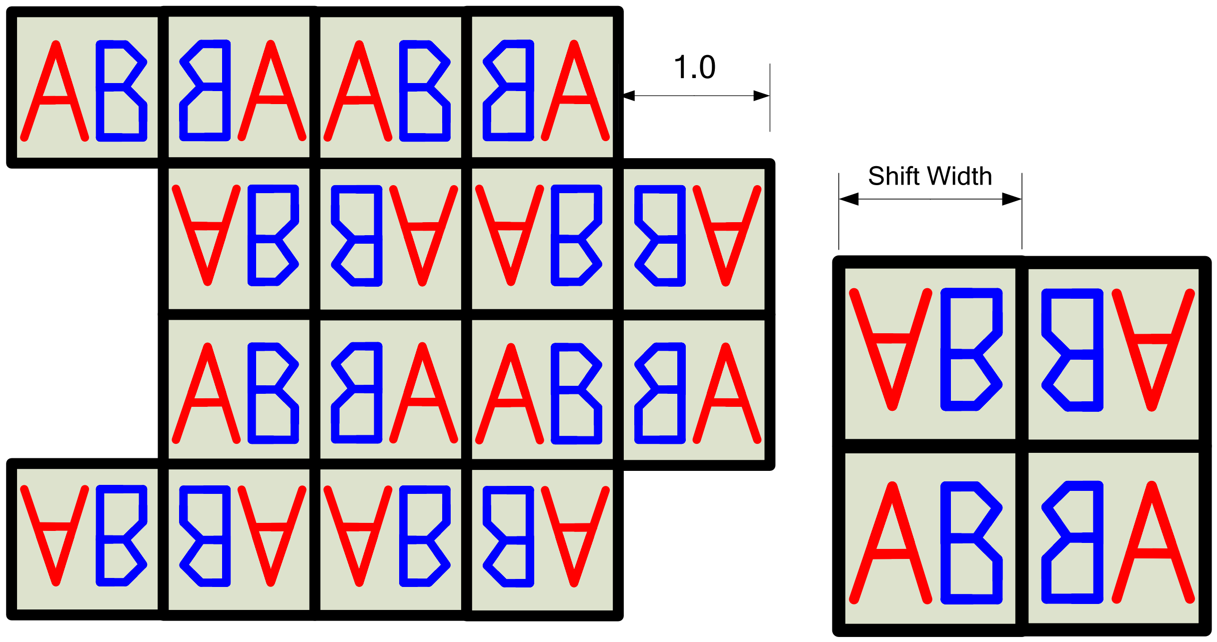
While it is not possible to adjust the shift width to a value of your choosing, the shift width varies depending on the clamp mode. The shift width is 0.5 (half a block) for all clamp modes except for mirrored repeat mode, as shown in Figure 10-18 below.

Figure 10-18: Shift Width in Modes Other Than Mirrored Repeat Mode



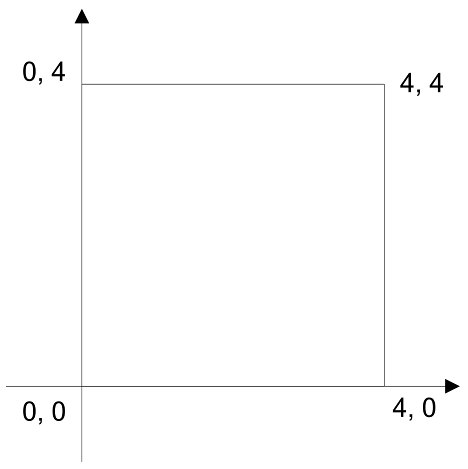
If clamping is set to mirrored repeat mode, the shift width is 1.0 (a whole block). This is because the broad sense of the term *a single block* is considered to also include the flipped (mirrored) side of the pattern.

Figure 10-19: Shift Width in Mirrored Repeat Mode



To illustrate the effects of the SHIFT-U, SHIFT-V, CLAMP-U, and CLAMP-V parameters, we established the coordinate system shown in Figure 10-20 for each polygon and rendered the appearance of various combinations.

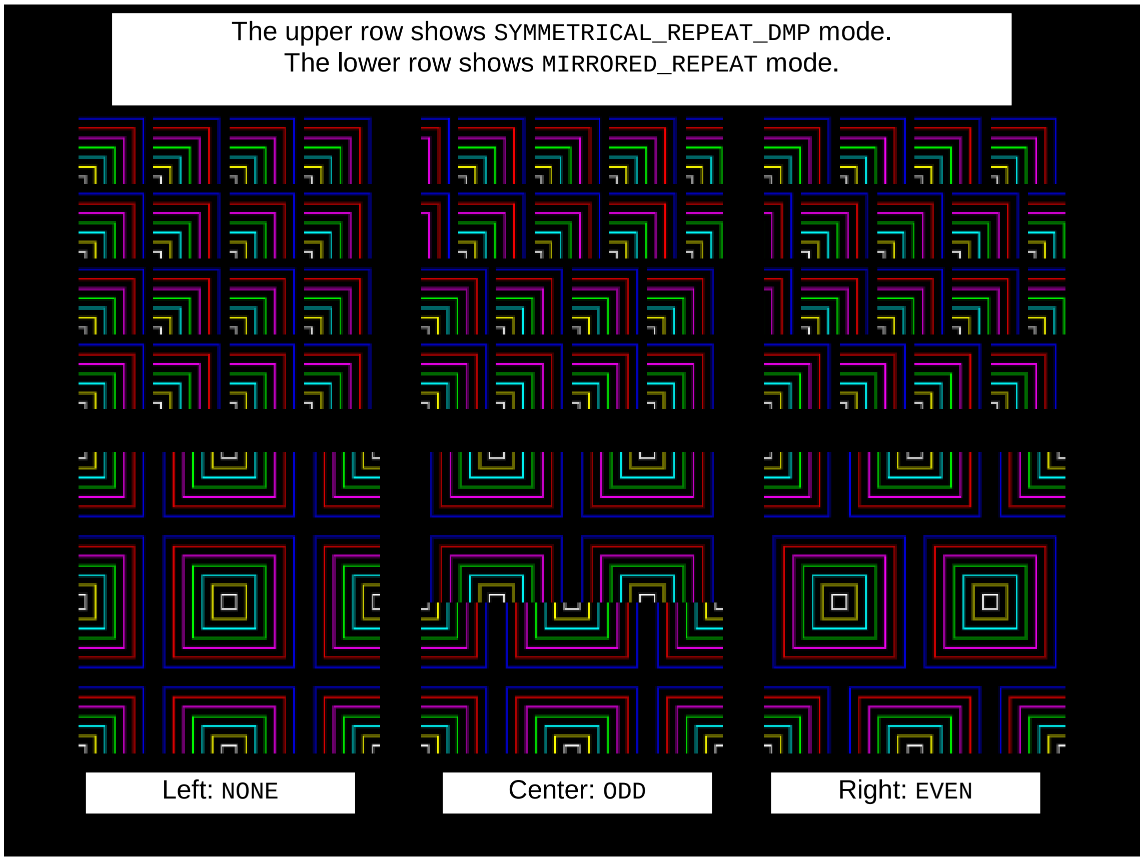
Figure 10-20: Polygon and Texture Coordinates



In the six rendered images shown in Figure 10-21, all settings are identical except for the shift and clamp modes.

© 2009–2012 Nintendo
CONFIDENTIAL

Figure 10-21: Renderings Illustrating the Effect of Shift Mode


The upper row shows SYMMETRICAL_REPEAT_DMP mode.
The lower row shows MIRRORED_REPEAT mode.
Left: NONE
Center: ODD
Right: EVEN

In the figures above, the only differences are the *`SHIFT-U`*, *`CLAMP-U`*, and *`CLAMP-V`* parameters for each polygon.

Table 10-2: CLAMP Parameters in Figure 10-21

	`CLAMP-U` **and** ***`CLAMP-V`*** **Parameters**
Top Row	`GL_SYMMETRICAL_REPEAT_DMP`
Bottom Row	`GL_MIRRORED_REPEAT`

Table 10-3: SHIFT Parameters in Figure 10-21

	`SHIFT-U` **Parameters**
Left	`GL_NONE_DMP`
Center	`GL_ODD_DMP`
Right	`GL_EVEN_DMP`

The images in the top row were rendered using repeat mode; the images in the bottom row were rendered using mirrored repeat mode. No shifting was done on the images in the left column.

11 DMP Fragment Lighting

This chapter explains the fragment lighting feature provided by DMPGL 2.0. When using DMP fragment lighting, the reserved fragment shader provided by DMPGL 2.0 must be attached to a valid program object.

Note that the term “OpenGL” within this chapter refers to OpenGL 1.x, which does not support the use of programmable pipelines. The term “Red Book” refers to the *OpenGL Programming Guide*, which was also written with OpenGL 1.x in mind.

The variable `i`, which appears within notation for the reserved uniforms for this chapter, `dmp_FragmentLightSource[i].*`, refers to the light number. `i` can take values from 0 to 7.

11.1 Overview

DMPGL 2.0 provides fragment lighting capabilities. In other words, the use of DMPGL 2.0 lets you control lighting on a per-fragment basis instead of a per-vertex basis.

DMP fragment lighting provides an interface for changing settings through reserved uniforms. This means that like OpenGL, the formula used for lighting calculation is fixed. The user can incorporate terms (consisting themselves of functions) into the lighting calculation formula. In other words, the user can choose both the actual functions that are used as terms of the calculation, as well as the arguments that are passed to these functions. The functions are specified as lookup tables. This method makes it possible to define a Phong, Cook-Torrance, or Schlick-like anisotropic lighting model, or any other shading model that can be represented by the DMP fragment lighting formula.

To enable DMP fragment lighting, set the reserved uniform as shown below.

```
glUniform1i(LOC(“dmp_FragmentLighting.enabled”), GL_TRUE);
```

You must also enable one or more light sources. For example, the following code can be used to enable `light0`.

```
glUniform1i(LOC(“dmp_FragmentLightSource[0].enabled”), GL_TRUE);
```

11.2 Scene Range

When using DMP fragment lighting, you must keep in mind a precaution about the range of the scene. The dimensions of the scene are limited to the range (-2^{15}, 2^{15}). The distance between the various objects that make up the scene and the viewpoint must be less than 2^{16}. When using point light sources, the same is true; the distance from the light to the viewpoint must be less than 2^{16}.

11.3 Lights and Materials

11.3.1 Ambient, Diffuse, Specular, and Emissive Light

Like OpenGL, DMP fragment lighting uses the concepts of ambient, diffuse, specular, and emissive light. The OpenGL Red Book includes definitions and real-world examples of each. As we stated earlier, a major difference between OpenGL and DMP fragment lighting is that with DMP fragment lighting, all of these types of light act on individual fragment units.

Another difference is that DMP fragment lighting uses two specular colors. These colors, called `specular0` and `specular1`, are expressed in RGBA format. The `specular1` color is used for modeling of monochromatic (black and white) representations. For more details about this type of model, see section 11.4 DMP Fragment Lighting Equations.

However, the biggest difference between DMP fragment lighting and OpenGL is the fact that DMP fragment lighting contains such a variety of ways of specifying how specular light is reflected off objects. This is the ultimate example of the flexibility that characterizes DMP fragment lighting settings.

11.3.2 Material Color

Similar to materials in OpenGL, materials in DMP fragment lighting also have ambient light color, diffuse light color, and specular light color. The difference between DMP fragment lighting and OpenGL in terms of material color is that in DMP, two specular lights can be specified (`specular0` and `specular1`). This difference makes two-layered representations possible (for example, two-toned paint jobs on a bicycle).

11.3.3 Creating Light Sources

All light source properties for DMP fragment shading are specified using the reserved uniforms `dmp_FragmentLightSource[i].*`. The table below shows the values that can be specified, along with the corresponding default values.

Table 11-1: Light Source Properties for DMP Fragment Lighting and Their Default Values

Reserved Uniform Name	Default Value	Description
`dmp_FragmentLightSource[i].ambient`	`(0.0,0.0,0.0,1.0)`	Sets the ambient light
`dmp_FragmentLightSource[i].diffuse`	`(1.0,1.0,1.0,1.0)` or `(0.0,0.0,0.0,1.0)`	Intensity of diffuse light (Default for light 0 is white; default for other lights is black)
`dmp_FragmentLightSource[i].specular0`	`(1.0,1.0,1.0,1.0)` or `(0.0,0.0,0.0,1.0)`	Intensity of specular light 0 (Default for light 0 is white; default for other lights is black)
`dmp_FragmentLightSource[i].specular1`	`(1.0,1.0,1.0,1.0)` or `(0.0,0.0,0.0,1.0)`	Intensity of specular light 1 (Default for light 0 is white; default for other lights is black)
`dmp_FragmentLightSource[i].position`	`(0.0,0.0,1.0,0.0)`	Position of light (x, y, z, w)
`dmp_FragmentLightSource[i].spotDirection`	`(0.0,0.0,-1.0)`	Direction of spotlight (x, y, z)

If you compare Table 11-1 above with Table 5-1 in the OpenGL Red Book, you'll notice the following differences:

1. There are two types of specular lights. This allows you to specify one more specular light than a strict OpenGL implementation would allow. (See section 11.3.1 Ambient, Diffuse, Specular, and Emissive Light.)
2. The features that correspond to the `GL_SPOT_EXPONENT` and `GL_SPOT_CUTOFF` parameters are not supported. DMP fragment lighting provides a flexible means of controlling the spread of the cone of illumination produced by a spotlight. This is done by letting the user define a function for calculating the cosine of the angle between the axis of the light cone and the fragment's light vector. This function is specified using a lookup table (for details, see section 11.4 DMP Fragment Lighting Equations).
3. DMP fragment lighting does not support features that correspond to `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION` in OpenGL. As with spotlights, users can define their own function to control attenuation based on the distance between a fragment and a light source.

The code below is an example of how to configure light sources using DMP fragment lighting.

Code 11-1: Sample Light Source Configuration

```
GLfloat la[] = {0.f, 0.f, 0.f, 1.f};
GLfloat ld[] = {1.f, 1.f, 1.f, 1.f};
GLfloat ls0[] = {0.35f, 0.35f, 0.35f, 1.f};
GLfloat ls1[] = {0.35f, 0.35f, 0.35f, 1.f};
GLfloat lpos0[] = {-35.5f, 0.f, 35.5f, 1.f};
GLfloat sd[] = {-1.f, -1.f, 0.f};
glUniform4fv(LOC(“dmp_FragmentLightSource[0].ambient”), 1, la );
glUniform4fv(LOC(“dmp_FragmentLightSource[0].diffuse”), 1, ld );
glUniform4fv(LOC(“dmp_FragmentLightSource[0].specular0”), 1, ls0);
glUniform4fv(LOC(“dmp_FragmentLightSource[0].specular1”), 1, ls1 );
glUniform4fv(LOC(“dmp_FragmentLightSource[0].position”), 1, lpos0);
glUniform3fv(LOC(“dmp_FragmentLightSource[0].spotDirection”), 1, sd);
```

Note that light sources (or groups of light sources) that are created in this way are per-fragment light sources (or groups thereof). These are distinct from per-vertex light sources.

11.3.4 Lighting Models

The OpenGL functions **`glLightModel*`** are used to control the four properties of the OpenGL lighting model. These four properties are the following: (1) the ambient light intensity in the scene, (2) whether the viewer is local to the scene, (3) whether to light one face or both faces of objects, and (4) whether the specular color should be distinguished from the other colors. The first of these properties (the global ambient) is also defined for DMP fragment lighting. It is set using a reserved uniform and is used to calculate the global ambient light for DMP fragment lighting. For details, refer to DMP Fragment Lighting Equations.

Code 11-2: Global Ambient Light Configuration

```
GLfloat al[] = {0.2f, 0.2f, 0.2f, 1.f};
glUniform4fv(LOC(“dmp_FragmentLighting.ambient”), 1, al);
```

The viewpoint calculation for DMP fragment lighting assumes that the viewer is always local to the scene.

Whether lighting calculations should be performed for both the front and back faces of objects is specified using the reserved uniform `dmp_FragmentLightSource[i].twoSideDiffuse`. Unlike OpenGL, DMP fragment lighting always calculates specular light independently. (See section 11.4 DMP Fragment Lighting Equations.)

11.3.5 Defining Material Properties

The reserved uniforms `dmp_FragmentMaterial*` are used to define material properties for DMP fragment lighting. Note that DMP fragment lighting does not support specification of different properties for the front and back faces. The names and default values of the uniforms that can be used to specify material parameters are shown below.

Table 11-2: Parameters Set When Defining Materials and Their Default Values

Reserved Uniform Name	Default Value	Description
`dmp_FragmentMaterial.ambient`	`(0.2, 0.2, 0.2, 1.0)`	Ambient color of material
`dmp_FragmentMaterial.diffuse`	`(0.8, 0.8, 0.8, 1.0)`	Diffuse color of material
`dmp_FragmentMaterial.specular0`	`(0.0, 0.0, 0.0, 1.0)`	Specular 0 color of material
`dmp_FragmentMaterial.specular1`	`(0.0, 0.0, 0.0, 1.0)`	Specular 1 color of material
`dmp_FragmentMaterial.emission`	`(0.0, 0.0, 0.0, 1.0)`	Emissive color of material
`dmp_FragmentMaterial.sampler{D0, D1,FR,RR,RG,RB}`		Lookup tables used for calculating DMP fragment lighting

Lookup tables for DMP fragment lighting each contain 512 entries. The first 256 entries are the lookup values, which must fall within the range [0, 1.0]. The following 256 entries are used to store the differences between each lookup value and the previous one. These differences (called “deltas” from this point onward) must fall within the range [-1.0, 1.0]. Spotlight distributions and distance attenuation distributions, like material properties, are defined by lookup tables that are loaded by the `glTexImage1D` function. (See section 11.3.3 Creating Light Sources.)

You may have already noticed that DMP fragment lighting does not support the `GL_SHININESS` parameter. In OpenGL, `GL_SHININESS` is used as the exponent of the dot product of a normal and a half-angle vector. DMP fragment lighting, on the other hand, allows you to specify any function of your choice, not just the exponent. If you want to use an exponential function like the one OpenGL uses to calculate specular light, just specify an exponential function to the lookup table. Code 11-3 below specifies an exponential function that behaves in the same way as if `GL_SHININESS` had been set to 30 in OpenGL. (For details, see section 11.4 DMP Fragment Lighting Equations.)

Code 11-3: Material Definition

```
  GLfloat mat_specular0[] = {1.f, 0.8f, 0.6f, 1.f};
  GLfloat mat_diffuse [] = {0.8f, 0.6f, 0.4f, 1.f};
  GLfloat mat_ambient [] = {0.f, 0.f, 0.f, 1.f};
  glUniform4fv(LOC(“dmp_FragmentMaterial.specular0”), 1, mat_specular0);
  glUniform4fv(LOC(“dmp_FragmentMaterial.ambient”), 1, mat_ambient);
  glUniform4fv(LOC(“dmp_FragmentMaterial.diffuse”), 1, mat_diffuse);

  GLfloat lut[512];
  int j;
  for (j = 0; j < 256; j++)
      lut[j] = powf((float)j/256.f, 30.f);
  for (j = 0; j < 255; j++)
      lut[j + 256] = lut[j + 1] - lut[j];
  lut[255 + 256] = 1.f - lut[255];
  glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerD1"), 0);
```

11.4 DMP Fragment Lighting Equations

This section explains the equations that are used when calculating lighting.

DMP fragment lighting operates on a per-fragment basis. Unlike OpenGL, DMP fragment lighting always calculates two colors: the primary and secondary colors. OpenGL only calculates both if the lighting model is set to `GL_SEPARATE_SPECULAR_COLOR`.

Equation 11-1: Calculation of Primary and Secondary Colors

$$\begin{aligned} Primary\ color &= Emissive\ light\ from\ material \\ &+ Effect\ of\ global\ ambient\ light \\ &+ Effect\ of\ ambient\ and\ diffuse\ light\ from\ light\ sources \\ Secondary\ color &= Effect\ of\ specular\ light\ from\ light\ sources \end{aligned}$$

11.4.1 Primary Color

The equation shown below explains how the primary color is calculated in greater detail.

Equation 11-2: Detailed Calculation of Primary Color

$$\begin{aligned} Primary\ color = {} & Emissive_{material} + Ambient_{lightingmodel} \times Ambient_{material} \\ & + \sum_{i=0}^{n-1} (Effect\ of\ spotlight_i) \times (Effect\ of\ distance\ attenuation_i) \\ & \times (Ambient_{light\_src\_i} \times Ambient_{material} + (max\{\boldsymbol{L_i} \cdot \boldsymbol{n}, 0\} \mid abs(\boldsymbol{L_i} \cdot \boldsymbol{n})) \\ & \times Diffuse_{light\_src\_i} \times Diffuse_{material} \times (Effect\ of\ shadow\ attenuation_i)) \end{aligned}$$

The value used for the $(max\{\boldsymbol{L_i} \cdot \boldsymbol{n}, 0\} \mid abs(\boldsymbol{L_i} \cdot \boldsymbol{n})\,)$ term will depend on the value of the reserved uniform `dmp_FragmentLightSource[i].twoSideDiffuse`. In other words, if `GL_LIGHT_ENV_TWO_SIDE_DIFFUSE` is true, the absolute value of $\boldsymbol{L} \cdot \boldsymbol{n}$ will be used. Otherwise, the greater of the two values $\boldsymbol{L} \cdot \boldsymbol{n}$ and 0 will be used. The latter case is analogous to the one-sided lighting feature of OpenGL.

The terms that appear within the equation are described below.

11.4.1.1 Effect of Spotlight

The $(Effect\ of\ spotlight_i)$ term is used for modeling the distribution of the light from a spotlight, just as it is with OpenGL. However, in DMP fragment lighting the distribution itself can be defined freely with a lookup table. The following procedure is used to enable spotlights.

Code 11-4: Enabling Spotlights

```
glUniform1i(LOC(“dmp_FragmentLightSource[i].spotEnabled”), GL_TRUE);
```

In this case, a spotlight will only be enabled for the `i`th light, and the `i`th light’s own lookup table will be used.

Use the following code to disable a spotlight. Spotlights are disabled by default.

Code 11-5: Disabling Spotlights

```
glUniform1i(LOC(“dmp_FragmentLightSource[i].spotEnabled”), GL_FALSE);
```

11.4.1.2 Effect of Shadow Attenuation

The $(Effect\ of\ shadow\ attenuation_i)$ term is an element for adjusting the shadow, and is sampled from the texture configured in the reserved uniform `dmp_LightEnv.shadowSelector`. The $(Effect\ of\ shadow\ attenuation_i)$ term can be replaced by a value of 1 using the following command.

Code 11-6: Replacing the Shadow Attenuation Term with One

```
glUniform1i(LOC(“dmp_FragmentLightSource[i].shadowed”), GL_FALSE);
```

11.4.1.3 Effect of Distance Attenuation

The $Effect\ of\ distance\ attenuation_i$ term is an element for adjusting distance attenuation and is used to get a distribution that depends on the distance between an object and a light source, just as it is with OpenGL. That said, the distribution itself can be defined freely with a lookup table. The code sample below shows how to enable distance attenuation.

Code 11-7: Enabling Distance Attenuation

```
glUniform1i(LOC(“dmp_FragmentLightSource[i].distanceAttenuationEnabled”),
GL_TRUE);
```

To disable distance attenuation, you would use the following code. Distance attenuation is disabled by default.

Code 11-8: Disabling Distance Attenuation

```
 glUniform1i(LOC(“dmp_FragmentLightSource[i].distanceAttenuationEnabled”),
GL_FALSE);
```

All other terms in the equation have the same meanings as those defined by OpenGL.

11.4.2 Secondary Color

The secondary color is calculated as follows.

Equation 11-3: Detailed Calculation of Secondary Color

$$\begin{aligned}&Secondary\ color = \\ &\sum_{i=0}^{n-1}(Effect\ of\ spotlight_i) \times (Effect\ of\ distance\ attenuation_i) \times (Effect\ of\ shadow\ attenuation_i) \\ &\qquad\qquad \times f_i \\ &\qquad\quad \times (Specular0_{light\_src\_i} \times Specular0_{material} \times Distribution0 \times GeometryFactor0_{light\_src\_i} \\ &\qquad\quad + Specular1_{light\_src\_i} \times (Reflection \mid Specular1_{material}) \times Distribution1 \\ &\qquad\qquad \times GeometryFactor1_{light\_src\_i})\end{aligned}$$

If you compare this with the equation that appears in the OpenGL Red Book for calculating the secondary color, you’ll notice that in the DMP fragment lighting secondary color equation there are two terms for the specular light:

$$\begin{aligned}Term_0 &= Specular0_{light\_src\_i} \times Specular0_{material} \times Distribution0 \times GeometryFactor0_{light\_src\_i} \\ Term_1 &= Specular1_{light\_src\_i} \times (Reflection \mid Specular1_{material}) \times Distribution1 \\ &\qquad \times GeometryFactor1_{light\_src\_i}\end{aligned}$$

In the corresponding OpenGL function, there is only one such term:

$$(max\{\boldsymbol{N} \cdot \boldsymbol{H}_i, 0\})^{shininess} \times Specular_{light\_src\_i} \times Specular_{material}$$

You can reduce the DMP fragment lighting equation to the same equation as the one used by OpenGL to calculate specular light. To do so, set $Distribution0$ to an exponential function (use $\boldsymbol{N} \cdot \boldsymbol{H_i}$ as the argument, and use the shininess as the exponent), set $GeometryFactor0_{light\_src\_i}$ to 1, and set $Specular1_{light\_src\_i}$ to zero. However, DMP fragment lighting has another term of the same type. This lets you construct two-layered models, which for example can be used to represent two-tone paint. Use of the two-term equation doesn’t only enable two-layer modeling; it could also be used to represent translucent models. In that case, you would use the first term to represent subsurface scattering, and the second term to represent the ordinary surface reflection.

The terms that appear within Equation 11-3 are described below. The terms $(Effect\ of\ spotlight_i)$, $(Effect\ of\ shadow\ attenuation_i)$, and $(Effect\ of\ distance\ attenuation_i)$ have the same meanings that were already provided by the description of the primary color in section 11.4.1 Primary Color.

11.4.2.1 Clamping of Unlit Areas

The function f_i is 1 if $L \cdot n$ >0 or 0 otherwise. This means that, like OpenGL, if a fragment is facing away from light $light\_src\_i$, the secondary color will be zero. This is enabled by default, or if the following instruction is issued.

Code 11-9: Enabling Clamping of Unlit Areas

```
glUniform1i(LOC(“dmp_LightEnv.clampHighlights“), GL_TRUE);
```

If the following instruction is issued, the value 1 is used instead of f_i.

Code 11-10: Disabling Clamping of Unlit Areas

```
glUniform1i(LOC(“dmp_LightEnv.clampHighlights“), GL_FALSE);
```

If you disable clamping of unlit areas, light can then pass through to areas where light would not normally reach. This setting is used for translucent materials that use strong subsurface scattering, such as wax, skin, and marble.

11.4.2.2 Geometry Factors

The terms $GeometryFactor0_{light\_src\_i}$ and $GeometryFactor1_{light\_src\_i}$ are required elements for constructing a Cook-Torrance shading model.

The term $GeometryFactorj_{light\_src\_i}$ (where j is 0 or 1) is included when the reserved uniform `dmp_FragmentLightSource[i].geomFactor`j is set to `GL_TRUE`. Include the code shown in Code 11-11 when you don’t want to include the geometry factors.

Code 11-11: Code for Not Using Geometry Factors

```
glUniform1i(LOC(“dmp_FragmentLightSource[i].geomFactor0“), GL_FALSE);
glUniform1i(LOC(“dmp_FragmentLightSource[i].geomFactor1“), GL_FALSE);
```

With these settings, the value 1 is used for the geometry factors during DMP fragment lighting calculation.

11.4.2.3 Distributions and Reflection

The terms $Distribution0$, $Distribution1$, and $Reflection$ are sampled values from lookup tables. $Distribution0$ and $Distribution1$ each indicates one sample value. $Reflection$ indicates a vector with three elements, each of which is interpreted as an RGB value. When this term is multiplied by $Specular1_{light\_src\_i}$, the multiplication is done individually for each element.

11.5 Defining the Lighting Environment

The reserved uniforms `dmp_LightEnv.*` and `dmp_FragmentLightSource[i].*` are used to specify how light sources interact with materials and properties.

11.5.1 Lighting Environment Parameters

The reserved uniforms in the table below are all set to one of two values, `GL_FALSE` or `GL_TRUE`.

Table 11-3: Boolean Lighting Environment Parameters and Their Default Values

Reserved Uniform Name	Default Value	Description
`dmp_LightEnv.lutEnabledRefl`	`GL_FALSE`	Sets whether to use the value `1` or a lookup table as the reflection component of term 1
`dmp_FragmentLightSource[i].twoSideDiffuse`	`GL_FALSE`	Sets whether to use $abs(\mathbf{L}\cdot\mathbf{n})$ *or* $max\{$L·n$, 0\}$
`dmp_LightEnv.bumpRenorm`	`GL_FALSE`	Enables or disables recalculation of the bump vectors
`dmp_LightEnv.clampHighlights`	`GL_TRUE`	Sets whether to clamp the specular color to `0` when $\mathbf{L}\cdot\mathbf{n} < 0$.
`dmp_FragmentLightSource[i].geomFactor0`	`GL_FALSE`	Sets whether to use the geometry factor in term 0
`dmp_FragmentLightSource[i].geomFactor1`	`GL_FALSE`	Sets whether to use the geometry factor in term 1
`dmp_LightEnv.lutEnabledD0`	`GL_FALSE`	Sets whether to use a lookup table or the value `1` as the distribution component of term 0
`dmp_LightEnv.lutEnabledD1`	`GL_FALSE`	Sets whether to use a lookup table or the value `1` as the distribution component of term 1
`dmp_FragmentLightSource[i].shadowed`	`GL_FALSE`	Configures the use of shadow
`dmp_LightEnv.shadowPrimary`	`GL_FALSE`	Applies shadow attenuation to the primary color
`dmp_LightEnv.shadowSecondary`	`GL_FALSE`	Applies shadow attenuation to the secondary color
`dmp_LightEnv.shadowAlpha`	`GL_FALSE`	Applies shadow attenuation to the alpha component
`dmp_LightEnv.invertShadow`	`GL_FALSE`	Inverts shadow attenuation
`dmp_LightEnv.absLutInput{D0,D1,FR,SP,RR RG,RB}`	`GL_FALSE`	Configures either `[0,1]` or `[-1,1]` as the range of the arguments of various lookup tables
`dmp_FragmentLightSource[i].spotEnabled`	`GL_FALSE`	Applies spotlights

© 2009–2012 Nintendo
CONFIDENTIAL

Reserved Uniform Name	Default Value	Description
`dmp_FragmentLightSource[i].distanceAttenuationEnabled`	`GL_FALSE`	Applies distance attenuation

The non-boolean reserved uniforms (as shown below) can take parameters other than `GL_FALSE` and `GL_TRUE`.

Table 11-4: Non-Boolean Lighting Environment Parameters and Their Default Values

Reserved Uniform Name	Values
dmp_LightEnv.lutInput{D0,D1,SP}	GL_LIGHT_ENV_NH_DMP, GL_LIGHT_ENV_VH_DMP, GL_LIGHT_ENV_NV_DMP, GL_LIGHT_ENV_LN_DMP, GL_LIGHT_ENV_SP_DMP, or GL_LIGHT_ENV_CP_DMP
dmp_LightEnv.lutInput{FR,RR,RG,RB}	GL_LIGHT_ENV_NH_DMP, GL_LIGHT_ENV_VH_DMP, GL_LIGHT_ENV_NV_DMP, or GL_LIGHT_ENV_LN_DMP
dmp_LightEnv.lutScale{D0,D1,SP,FR,RR,RG,RB}	1.0, 2.0, 4.0, 8.0, 0.5, or 0.25
dmp_LightEnv.config	GL_LIGHT_ENV_LAYER_CONFIG0_DMP, GL_LIGHT_ENV_LAYER_CONFIG1_DMP, GL_LIGHT_ENV_LAYER_CONFIG2_DMP, GL_LIGHT_ENV_LAYER_CONFIG3_DMP, GL_LIGHT_ENV_LAYER_CONFIG4_DMP, GL_LIGHT_ENV_LAYER_CONFIG5_DMP, GL_LIGHT_ENV_LAYER_CONFIG6_DMP, or GL_LIGHT_ENV_LAYER_CONFIG7_DMP
dmp_LightEnv.bumpMode	GL_LIGHT_ENV_BUMP_NOT_USED_DMP, GL_LIGHT_ENV_BUMP_AS_BUMP_DMP, or GL_LIGHT_ENV_BUMP_AS_TANG_DMP
dmp_LightEnv.fresnelSelector	GL_LIGHT_ENV_NO_FRESNEL_DMP, GL_LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP, GL_LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP, or GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP
dmp_LightEnv.bumpSelector dmp_LightEnv.shadowSelector	GL_TEXTURE0, GL_TEXTURE1, GL_TEXTURE2, or GL_TEXTURE3
dmp_FragmentLightSource[i].distanceAttenuationBias	Any float value
dmp_FragmentLightSource[i].distanceAttenuationScale	Any float value

© 2009–2012 Nintendo
CONFIDENTIAL

11.5.2 Lookup Tables

Up to 22 lookup tables used for lighting calculations can be bound at a given time. The corresponding reserved uniforms are shown below.

```
dmp_FragmentMaterial.samplerD0,
dmp_FragmentMaterial.samplerD1,
dmp_FragmentMaterial.samplerFR,
dmp_FragmentMaterial.samplerRR,
dmp_FragmentMaterial.samplerRG,
dmp_FragmentMaterial.samplerRB,
dmp_FragmentLightSource[i].samplerSP (where i is an integer from 0 to 7),
dmp_FragmentLightSource[i].samplerDA (where i is an integer from 0 to 7)
```

In order, these represent distribution 0, distribution 1, the Fresnel factors, the R, G, and B components of the reflection, the effect of spotlights for lights 0-7, and the effect of distance attenuation for lights 0-7.

11.5.3 Binding Lookup Tables

Lookup tables are managed by means of a texture object and a target for the lookup table. Instances of lookup tables are bound using these texture objects and targets.

In Code 11-12 below, we create a lookup table by preparing a single `float` array of size 512. This array must be of size 512; the first 256 elements are used to store the function’s values, and the last 256 elements are used to store the deltas of the function’s values. The roles of these blocks of elements cannot be changed. This code sample assumes that we’ve already defined the function **`func(x)`**, which calculates the values we want to set in our table. The integer variable `ref_func` used in the code sample is the lookup table's identification number. It is used when accessing the created lookup table from a reserved uniform, and can be set to any value between 0 and 31 (inclusive).

Code 11-12: Binding a Lookup Table

```
GLfloat func(GLfloat x /* x must be in the range [0.f , 1.f] */ )
{
    return /* Return as the value of the lookup table */;
}

GLfloat map_func[512];
GLint ref_func = ...; // Arbitrary value in the range [0,31]
GLint tex_name;
for (int i = 0; i < 256; i++)  // Set the function values
    map_func[i] = func((float)i/256.f);
for (int i = 0; i < 256; i++)  // Set the deltas of the function values
    map_func[i+256] = func((float)(i+1)/256.f) – func(i/256.f);
glGenTextures(1, &tex_name);
glBindTexture(GL_LUT_TEXTURE0_DMP+ref_func, tex_name);
glTexImage1D(GL_LUT_TEXTURE0_DMP+ref_func, 0,
    GL_LUMINANCEF_DMP, 512, 0, GL_LUMINANCEF_DMP, GL_FLOAT, map_func);

// To allow the reserved fragment shader to use the lookup table we've
created,
// set a reserved uniform to ref_func.
glUniform1i(LOC("ReservedUniformToBindWithLookupTable"), ref_func);
```

11.5.4 Layer Configuration

As explained in section 11.4 DMP Fragment Lighting Equations, lookup tables for materials are used to represent various characteristics of materials like the spotlight and reflection distributions. Layer configurations determine which lookup tables are used within the lighting equations for the primary and secondary colors, and where in the lighting equations those lookup tables are used.

Table 11-5 describes the layer configurations for the secondary color.

Table 11-5: Layer Configurations for the Secondary Color in DMP Fragment Lighting

Layer Configuration	Lookup Table Assignments							
	Rr	**Rg**	**Rb**	**D0**	**D1**	**Fr**	**Sp**	
`GL_LIGHT_ENV_LAYER_CONFIG0_DMP`	RR	RR	RR	D0	-	-	SP	1
`GL_LIGHT_ENV_LAYER_CONFIG1_DMP`	RR	RR	RR	-	-	FR	SP	1
`GL_LIGHT_ENV_LAYER_CONFIG2_DMP`	RR	RR	RR	D0	D1	-	-	1
`GL_LIGHT_ENV_LAYER_CONFIG3_DMP`	-	-	-	D0	D1	FR	-	1
`GL_LIGHT_ENV_LAYER_CONFIG4_DMP`	RR	RG	RB	D0	D1	-	SP	2
`GL_LIGHT_ENV_LAYER_CONFIG5_DMP`	RR	RG	RB	D0	-	FR	SP	2
`GL_LIGHT_ENV_LAYER_CONFIG6_DMP`	RR	RR	RR	D0	D1	FR	SP	2
`GL_LIGHT_ENV_LAYER_CONFIG7_DMP`	RR	RG	RB	D0	D1	FR	SP	4

The numbers in the third column in Table 11-5 indicate how many hardware cycles are required for the lighting calculations with the various layer configurations. For high-speed calculation, we recommend choosing a layer configuration with a low number.

The correspondences between the terms within the lighting equation for the secondary color and the assignments in Table 11-5 are shown below.

- `Rr`: R component of Reflection
- `Rg`: G component of Reflection
- `Rb`: B component of Reflection
- `D0`: Distribution0
- `D1`: Distribution1
- `Sp`: Effect of spotlights

You can choose only a single layer configuration, and it applies to all per-fragment light sources. For example, if you selected layer configuration 0 using the function below, the secondary color equation would be as shown in Equation 11-4.

Code 11-13: Selecting Layer Configuration 0

```
glUniform1i(LOC(“dmp_LightEnv.config”), GL_LIGHT_ENV_LAYER_CONFIG0_DMP);
```

Equation 11-4: Secondary Color Equation Using Layer Configuration 0

$$Secondary\ color =$$

$$\sum_{i=0}^{n-1} SP \times (Effect\ of\ shadow\ attenuation_i) \times (Effect\ of\ distance\ attenuation_i)$$

$$\times f_i$$

$$\times (Specular0_{light\_src\_i} \times Specular0_{material} \times D0 \times GeometryFactor0_{light\_src\_i}$$

$$+Specular1_{light\_src\_i} \times (RR\ \boldsymbol{or}\ Specular1_{material}) \times GeometryFactor1_{light\_src\_i})$$

Because no lookup table is assigned to `D1` in layer configuration 0 in Table 11-5, distribution 1 is removed from the equation altogether (it is replaced with the value `1`).

As an example, the code sample below uses a single light source (light source 0) and assigns the lookup tables `RR`, `D0`, and `SP` to light source 1.

Code 11-14: Assigning Lookup Tables RR, D0, and SP to Light Source 1

```
 glUniform1i(LOC(“dmp_LightEnv.config”), GL_LIGHT_ENV_LAYER_CONFIG0_DMP);
 glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut0);
 glUniform1i(LOC("dmp_FragmentMaterial.samplerRR"), 0);
 glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut1);
 glUniform1i(LOC("dmp_FragmentMaterial.samplerD0"), 1);
 glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut2);
 glUniform1i(LOC("dmp_FragmentLightSource[0].samplerSP"), 2);
```

With the lookup table assignments in the code sample above, the secondary color equation becomes the following.

Equation 11-5: Secondary Color Equation for Code 11-14

$$\begin{aligned}
Secondary\ color = \\
& lut2 \times (Effect\ of\ shadow\ attenuation_0) \times (Effect\ of\ distance\ attenuation_0) \times f_0 \\
& \times (Specular0_{light\_src\_0} \times Specular0_{material} \times lut1 \times GeometryFactor0_{light\_src\_0} \\
& + Specular1_{light\_src\_0} \times (\mathrm{lut0} \mid Specular1_{material}) \times GeometryFactor1_{light\_src\_0})
\end{aligned}$$

11.5.5 Selecting Lookup Table Arguments

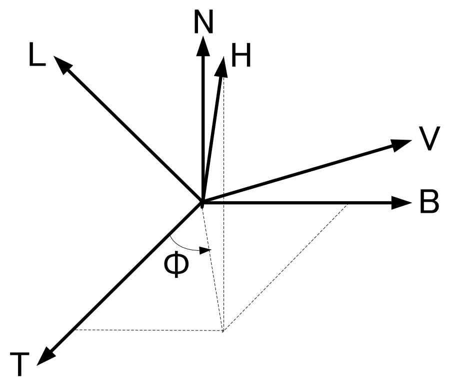
Table 11-6 below is a list of input values to the lookup tables. These values are constants that are set for the reserved uniforms `dmp_LightEnv.lutInput*`. (However, only `LIGHT_ENV_{NH,VH,NV,LN}_DMP` can be used as input to `dmp_LightEnv.lutInput{FR,RR,RG,RB}`.)

© 2009–2012 Nintendo
CONFIDENTIAL

Table 11-6: List of Constants Set for dmp_LightEnv.lutInput*

Constants Available as Parameters	Description
`LIGHT_ENV_NH_DMP`	Cosine of the angle formed by the normal and the half-angle vector $(\mathbf{N} \cdot \mathbf{H})$.
`LIGHT_ENV_VH_DMP`	Cosine of the angle formed by the view vector and the half-angle vector $(\mathbf{V} \cdot \mathbf{H})$.
`LIGHT_ENV_NV_DMP`	Cosine of the angle formed by the normal and the view vector $(\mathbf{N} \cdot \mathbf{V})$.
`LIGHT_ENV_LN_DMP`	Cosine of the angle formed by the light vector and the normal $(\mathbf{L} \cdot \mathbf{N})$.
`LIGHT_ENV_SP_DMP`	Cosine of the angle formed by the inverse light vector and the spotlight direction.
`LIGHT_ENV_CP_DMP`	Cosine of the angle ϕ formed by the projection of the half-angle vector on the tangent plane and the tangent vector ?? (see Figure 11-1 below).

Figure 11-1: Basic Vectors and Angles Used in Lighting Calculations



By assigning the arguments that serve as inputs to the lookup tables, in Code 11-15 we can complete the example of how to use lookup tables that we started in Code 11-14.

Code 11-15: How to Use Lookup Tables

```
  glUniform1i(LOC(“dmp_LightEnv.config”), GL_LIGHT_ENV_LAYER_CONFIG0_DMP);
  glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut0);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerRR"), 0);
  glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut1);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerD0"), 1);
  glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut2);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerSP"), 2);

  glUniform1i(LOC(“dmp_LightEnv.lutInputRR”), GL_LIGHT_ENV_NH_DMP);
  glUniform1i(LOC(“dmp_LightEnv.lutInputD0”), GL_LIGHT_ENV_NV_DMP);
  glUniform1i(LOC(“dmp_LightEnv.lutInputSP”), GL_LIGHT_ENV_LN_DMP);
```

In the following example, the secondary color equation becomes as shown when it uses the lookup tables and inputs set in Code 11-15.

Equation 11-6: Secondary Color Equation Using Lookup Tables as in Code 11-15

$$\begin{aligned}
&Secondary\ color\\
&\qquad = \mathrm{lut2}(\boldsymbol{L}\cdot\boldsymbol{N}) \times (Effect\ of\ shadow\ attenuation_0) \times (Effect\ of\ distance\ attenuation_0)\\
&\qquad \times f_0\\
&\qquad \times (Specular0_{light\_src\_0} \times Specular0_{material} \times \mathrm{lut1}(\boldsymbol{N}\cdot\boldsymbol{V}) \times GeometryFactor0_{light\_src\_0}\\
&\qquad + Specular1_{light\_src\_0} \times (\mathrm{lut0}(\boldsymbol{N}\cdot\boldsymbol{H}) \mid Specular1_{material}) \times GeometryFactor1_{light\_src\_0})
\end{aligned}$$

11.5.6 Adjusting Lookup Table Output Values

As we explained in section 11.3.5, material lookup tables store values in the range [0, 1]. Because many terms are multiplied together in the lighting equations, it may sometimes be necessary to increase the size of the values that are sampled from the lookup tables. In other words, there may be situations when you need your lookup tables to output values larger than the [0, 1] range. Scaling factors for lookup tables have been provided for this purpose. The following values can be used as scaling factors: 1.0, 2.0, 4.0, 8.0, 0.5, and 0.25.

11.5.7 Tangent Vectors

When using anisotropic representations or normal mapping, it may be necessary to specify tangent vectors in addition to the normals defined for the vertices. The normals and tangent vectors specified here are the basis vectors that construct the tangent space defined by the vertices. (The basis vectors of this tangent space are **T**, **B**, and **N**, as shown in Figure 11-1.) Furthermore, the vertex program must convert the rotation matrix constructed of these basis vectors (shown below) to a quaternion, and output that quaternion. The equation below is a rotation matrix that converts from tangent space to eye space.

© 2009–2012 Nintendo
CONFIDENTIAL

Equation 11-7: Rotation Matrix to Convert from Tangent Space to Eye Space

$$rotation\_matrix = \begin{bmatrix} Tx & Bx & Nx \\ Ty & By & Ny \\ Tz & Bz & Nz \end{bmatrix}$$

11.5.8 Normal Mapping

The normals of each fragment within a surface are calculated by interpolating between vertices. When using what is also known as *bump mapping*, it is necessary to apply perturbation to these “smooth” normals.

In this case, you select a texture to serve as the bump texture. The R, G, and B channels of the bump texture store the perturbation normals *x*, *y*, and *z* in the tangent space (the space that has the basis vectors **T**, **B**, and **N**, as shown in Figure 11-1). Each perturbation normal is a value in the range [-1.0, 1.0].

An example of normal mapping is shown below.

Code 11-16: Normal Mapping

```
glUniform1i(LOC(“dmp_LightEnv.bumpMode”), GL_LIGHT_ENV_BUMP_AS_BUMP_DMP);
glUniform1i(LOC(“dmp_LightEnv.bumpSelector”), GL_TEXTURE1);
glUniform1i(LOC(“dmp_LightEnv.bumpRenorm”), GL_TRUE);
```

The third line in this code sample generates the *z* component. This is useful in eliminating quantization noise, which can occur due to a lack of bit precision (8 bits per coordinate) in the bump image. This is also required when using textures in formats such as `GL_HILO8_DMP`, for which there is no *z* component.

The use of normal mapping requires that tangent vectors be assigned for the vertices (see section 11.5.7 Tangent Vectors for more information).

11.5.9 Tangent Mapping

It is also possible for tangent vectors to be perturbed. An example is shown below.

Code 11-17: Setting the Tangent

```
glUniform1i(LOC(“dmp_LightEnv.bumpMode”), GL_LIGHT_ENV_BUMP_AS_TANG_DMP);
glUniform1i(LOC(“dmp_LightEnv.bumpSelector”), GL_TEXTURE1);
glUniform1i(LOC(“dmp_LightEnv.bumpRenorm”), GL_FALSE);
```

The first line in Code 11-17 specifies that the bump texture is to be used as a tangent map. The second line specifies the texture to use. The third line disables generation of the *z* component. Generation of the *z* component assumes that the bump texture is being used for bump mapping; it is not necessary for tangent mapping.

11.6 Defining Quaternions

With DMP fragment lighting, normal vectors and view vectors are targets for rasterization just like vertex coordinates are. As a result, your vertex shaders must output both normal vectors and view vectors. Note that while view vectors can be output directly from the vertex shaders, the same is not true for normal vectors. Normal vectors are considered to be one of the basis vectors that construct the tangent space (a.k.a. *surface-local* space), and the tangent space is defined on a per-vertex basis. To make these basis vectors into vertex shader output attributes, they are first converted into quaternions.

The same is true when using normal vectors as vertex attributes, as is the case when using anisotropic reflection models or bump mapping. Like normal vectors, tangent vectors are also considered to be one of the basis vectors that construct the tangent spaces defined on a per-vertex basis. Tangent vectors are output as quaternions.

The C function in Code 11-18 below converts the rotation matrix `m` in Equation 11-7 to the quaternion array `quat`.

© 2009–2012 Nintendo
CONFIDENTIAL

Code 11-18: Function to Convert a Rotation Matrix to Quaternions

```
void matrix_to_quat( float quat[4], const float m[3][3] )
{
    float tr, s, q[4];
    int i, j, k;
    int nxt[3] = {1, 2, 0};

    tr = m[0][0]+m[1][1]+m[2][2];

    if ( tr > 0.f )
    {
        s = sqrtf(tr+1.f);
        quat[3] = s/2.f;
        s = 0.5f/s;
        quat[0] = (m[1][2]-m[2][1])*s;
        quat[1] = (m[2][0]-m[0][2])*s;
        quat[2] = (m[0][1]-m[1][0])*s;
    }
    else
    {
        i = 0;
        if (m[1][1] > m[0][0]) i = 1;
        if (m[2][2] > m[i][i]) i = 2;
        j = nxt[i];
        k = nxt[j];

        s = sqrtf(m[i][i]-(m[j][j]+m[k][k])+1.f);
        q[i] = s*0.5f;
        if (s != 0.f)  s = 0.5f/s;
        q[3] = (m[j][k]-m[k][j])*s;
        q[j] = (m[i][j]+m[j][i])*s;
        q[k] = (m[i][k]+m[k][i])*s;
        quat[0] = q[0];
        quat[1] = q[1];
        quat[2] = q[2];
        quat[3] = q[3];
    }
}
```

11.7 Lookup Table Configuration

As stated in section 11.5 Defining the Lighting Environment, each lookup table consists of an array of type `float` with 512 elements, the first 256 of which are used for storing the function’s values, and the last 256 of which are used for storing the deltas between the function's values. The size of the array cannot be changed, nor can the roles of these blocks of elements. If `dmp_LightEnv.absLutInput*` is set to `GL_TRUE`, the lookup table's output value is calculated as follows, given input value a in the range [0.0, 1.0]:

$$\begin{aligned} Output = {} & Table\ output\ value, using\ the\ integer\ portion\ of\ (a \times 256)\ as\ the\ index \\ & + Table\ output\ value, using\ (256 + the\ integer\ portion\ of\ (a \times 256))\ as\ the\ index \\ & \quad \times\ Fractional\ portion\ of\ (a \times 256) \end{aligned}$$

In other words, this is $Output\ =\ LUT(\lfloor a \times 256 \rfloor) + LUT(\lfloor a \times 256 \rfloor + 256) \times \{a \times 256\}$.

The fractional portion has a precision of 4 bits, and the maximum possible value of $(a \times 256)$ given input value a is only 255.9375. As a result, if you configure your lookup table in the manner shown in Code 11-12, it is not possible to obtain **func**`(1.0)` in return for the maximum input value.

To obtain **func**`(1.0)` in return for the maximum input value, you must revise your lookup table configuration. An example is shown below.

```
GLfloat func(GLfloat x /* x must be in the range [0.f , 1.f] */ )
{
    return /* Return as the value of the lookup table */;
}

GLfloat map_func[512];
GLint ref_func = ...; // Arbitrary value in the range [0,31]
GLint tex_name;
for (int i = 0; i < 256; i++)  // Set the function's values
    map_func[i] = func((float)i/256.f);
for (int i = 0; i < 255; i++)  // Set the deltas of the function's values
    map_func[i+256] = map_func[i+1] – map_func[i];
map_func[511] = (func(1.f)-map_func[255])*16.f/15;
```

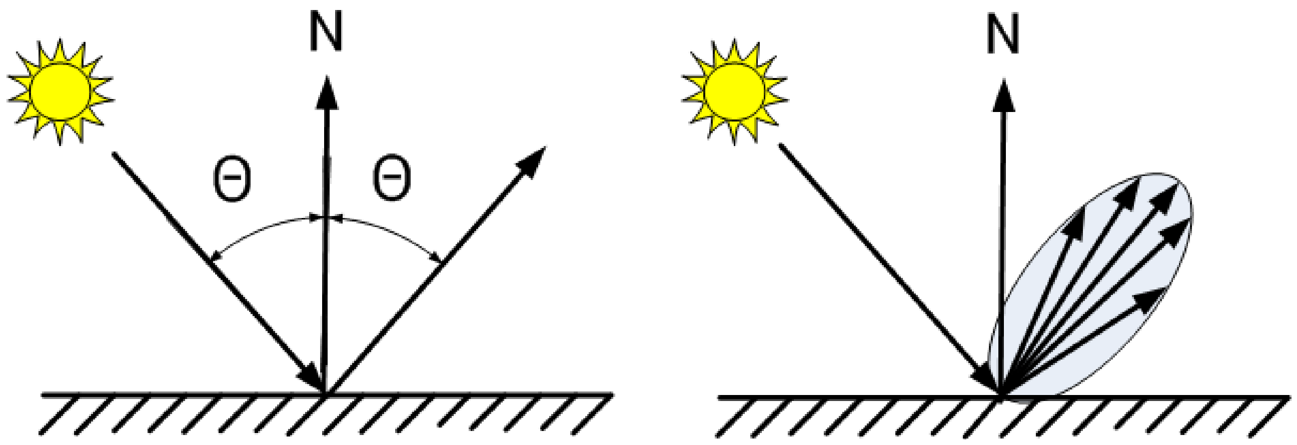
© 2009–2012 Nintendo
CONFIDENTIAL

12 Illumination Models in DMP Fragment Lighting

12.1 Specular and Diffuse Reflections

Different materials reflect light in different ways. For example, metals that have smooth, mirror-like surfaces perfectly reflect incident light by reflecting it all in a single direction. Light that is reflected in this way is called *specular* light.

Figure 12-1: Specular and Diffuse Reflections



The image on the left in Figure 12-1 shows the reflection of light off a mirror (specular reflection). The image on the right shows the scattered reflections off a polished metal (diffuse reflection).

The angle of reflection is the same as the angle of incidence. This principle is known as Snell's law.

If the surface of a material is rough, incoming light is reflected in a range of directions (see Figure 12-1). Because this range is extremely small, highlights can still be seen in polished metal. This slightly-scattered reflection is also specular, and is seen in any material that has a sheen or luster to it (for example, velvet and silk).

Specular highlights are represented in computer graphics using the Blinn-Phong equation[1].

Equation 12-1: Blinn-Phong Equation

$$specular = color \times (N \cdot H)^s$$

The exponent s is a constant known as shininess, and $N \cdot H$ represents the dot product of the normal N with the half-angle vector H, which is itself defined by Equation 12-2 below. L is a unit vector pointing at the light source, and V is a unit vector pointing at the viewer.

[1] Blinn, James F. Models of Light Reflection for Computer Synthesized Pictures, ACM Computer Graphics, SIGGRAPH 1977 Proceedings, 11(4), pp. 192-198

Equation 12-2: Calculation of the Half-Angle Vector

$$H = \frac{L + V}{|L + V|}$$

These unit vectors are shown in Figure 12-2 below. The color of the specular highlight is calculated by multiplying to yield each component.

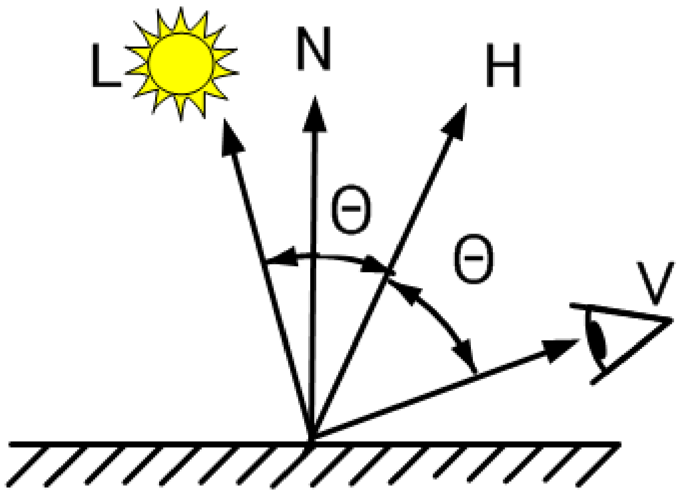
Equation 12-3: Color of Specular Highlights

$$color = m \times s = (m_r s_r, m_g s_g, m_b s_b)$$

In Equation 12-3, m indicates the color of the material, and s indicates the color of the light source.

The reason that the dot product $N \cdot H$ is used within the specular calculation is explained by the reflection model known as *microfacet reflection*.

Figure 12-2: Light Source Vector L, Half-Angle Vector H, and View Vector V



In contrast, with materials like chalk, incident light is reflected in an entirely different way. The light is scattered uniformly in all directions (see the image on the right in Figure 12-1). This type of reflection is referred to as either *diffuse light* or *Lambertian reflectance*, and is represented using the following equation.

Equation 12-4: Diffuse Light Equation

$$diffuse = color \times L \cdot N$$

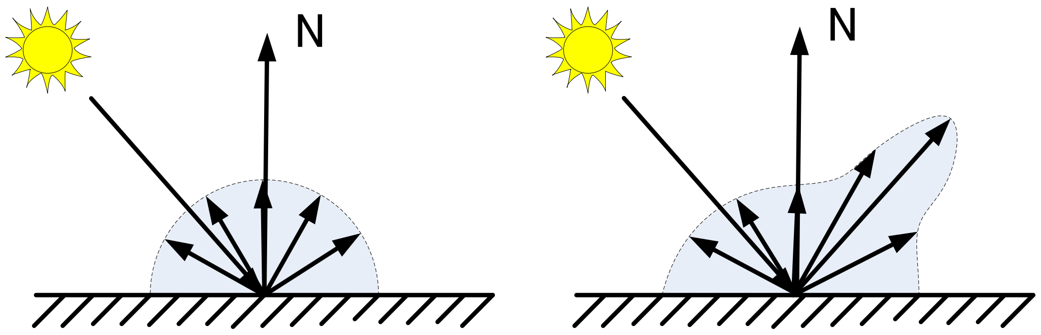
The $color$ term in Equation 12-4 is calculated using the same equation as we use to calculate specular light, Equation 12-3. Both L and N are unit vectors.

Equation 12-5: Dot Product of Light Source Vector and Normal Vector

$$L \cdot N = cos(\alpha)$$

The variable α represents the angle formed by L and N. The dot product $L \cdot N$ has a maximum value of 1.0 when α is 0° (in other words, when L and N are pointing in the exact same direction), and is 0.0 when α is 90° (in other words, when L is parallel to the surface). All real-world materials actually have both specular and diffuse properties. See the image on the right in Figure 12-3 below.

Figure 12-3: Difference Between Idealized Diffuse Reflection and Reflection in a Typical Material



The image on the left in Figure 12-3 shows idealized diffuse reflection. The image on the right shows typical light reflection for a typical material.

For this type of material with mixed characteristics, the reflection is represented by simply summing the specular and diffuse components. For more information, see section 12.3.2 Blinn-Phong Model.

Equation 12-6: Reflection in Materials with Both Diffuse and Specular Components

$$reflection = diffuse + specular$$

Even if the equations used to calculate the specular and diffuse components are the same, there are cases when the light source and the material have different combinations of specular and diffuse colors. This means that materials and light sources both use models in which the color of their specular light differs from the color of their diffuse light. This approach is used by OpenGL as well as many other lighting models.

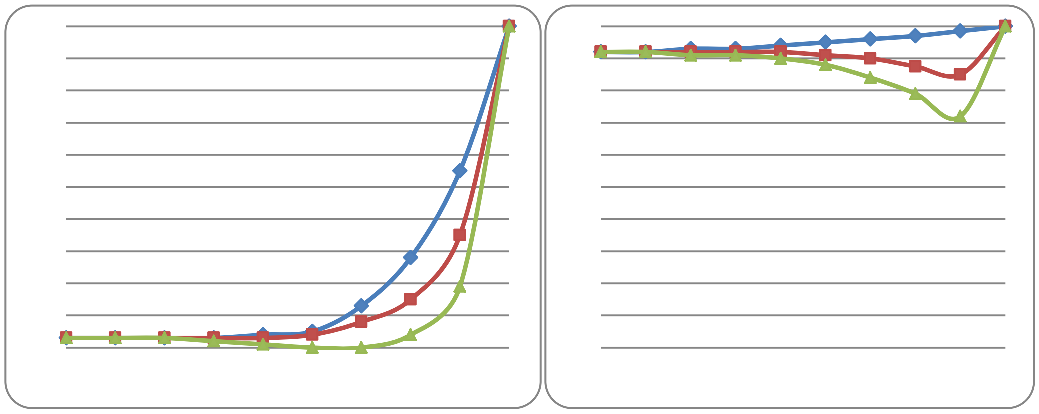
The reflection of light in the real world is not as simple as the aforementioned equations would seem to indicate. Reflection is typically represented as a function of the directions of the viewpoint and light sources. This function is known as the bidirectional reflectance distribution function (or "BRDF"). The next section considers how the BRDF is implemented using DMP fragment lighting. From this point forward, the BRDF will be referred to simply as *reflectance*.

12.2 Fresnel Reflectance

When light propagates through a medium other than a vacuum, it travels slower than it would in a vacuum. The factor that represents the ratio of these speeds is known as the *refractive index* (η). It is known that the refractive index and the reflectance of a given material depend on this ratio of the speed of light in that material versus the speed of light in a vacuum.

The French scientist Augustin-Jean Fresnel derived the formulas that are now known as the Fresnel equations. These equations describe the fact that the reflection and refraction of transmitted light are functions of the angle of incidence α and the refractive index η. Figure 12-4 below shows how Fresnel reflection relates to the angle of incidence. The graph on the left shows the relationship between reflection and the angle of incidence for glass. The graph on the right shows the same for aluminum.

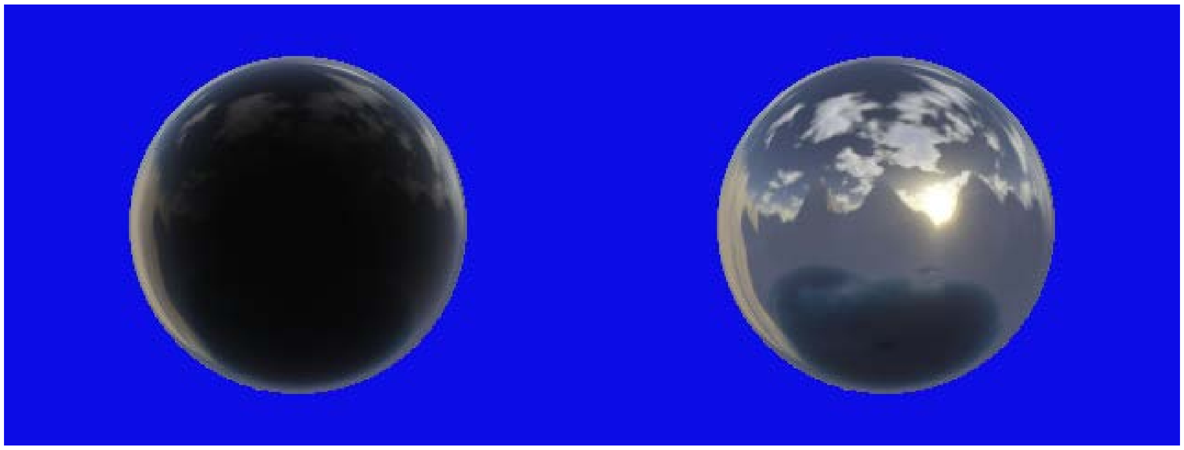
Figure 12-4: Relationship Between Angle of Incidence and Fresnel Reflection in a Dielectric Material (Left) and a Metal (Right)



The smooth surfaces of glass allow it to transmit some of the incident light and reflect the rest as specular light. In the same way, metals reflect light in accordance with the Fresnel equations, but metals absorb some of the incident rays of light. This absorption is represented mathematically by the *absorption coefficient* (κ). In other words, for metals, the Fresnel reflectance is determined by three factors: α, η, and the absorption coefficient κ.

As you can see from Figure 12-4 above, the greater the angle between the light source and the line of sight, the closer the reflection becomes to 1.0. The rendered images in Figure 12-5 illustrate the effect of this relationship.

Figure 12-5: Differences in Reflection When the Fresnel Equations Are Used (Left) and Not Used (Right)



To simplify the discussion, the Fresnel reflectance is represented as a cosine function of angle σ. The cosine of σ can be found easily by taking the dot product of the light vector and the normal.

Based on this fact, the Fresnel reflection function is expressed as $F(cos(\sigma),\eta)$ for dielectric materials, or as $F(cos(\sigma),\eta,\kappa)$ for metals.

12.3 Lighting Models

Lighting models refer to mathematical models that express how lighting occurs within a three-dimensional scene. These models include information such as the colors of the light sources, the colors of the materials, and how the light is reflected. A typical lighting model can be broken down into several components: ambient, emissive, diffuse, and specular.

Equation 12-7: Typical Lighting Model

$$reflected\_light = emissive + ambient + diffuse + specular$$

Below, we explain the specular-related aspects of the Blinn-Phong, Cook-Torrance, Schlick, and other lighting models. The first three terms in Equation 12-7 are almost identical for all lighting models; DMP fragment lighting uses the same values in all lighting models for these first three terms. (For more information, see Chapter 11 DMP Fragment Lighting.)

With DMP fragment lighting, reflected light is considered to be composed of two terms, the primary color and the secondary color.

Equation 12-8: Reflected Light in DMP Fragment Lighting

$$reflected\_light = primary + secondary$$

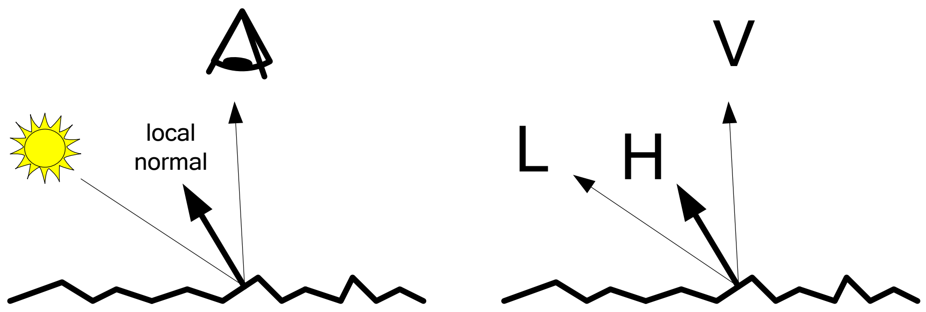
Note that the primary color is simply the sum of the ambient, emissive, and diffuse components. The secondary color, on the other hand, is used to represent various types of BRDFs (Blinn-Phong and Cook-Torrance, for example) in addition to the specular component. The specific BRDF implementations are covered in section 12.3.6.1 Implementation Using DMP Fragment Lighting.

12.3.1 Microfacet Reflection

Here, we consider reflections on rough surfaces. Rough surfaces can be thought of as a collection of many tiny flat surfaces. Each of these tiny surfaces has its own normal, which points in a different direction from the other normals. These surfaces are so small that they cannot be seen with the naked eye, and are known as *microfacets*.

Figure 12-6 below shows how light is reflected off of a microfacet, and how that light reaches the eye of the viewer. Because the individual microfacets are flat and smooth, the normals (labeled "local normal" in the figure below) must be identical to the half-angle vector of the light vector L and the view vector V in order for the light to be reflected completely toward the viewer. Furthermore, these normals also serve as the half-angle vector H described in section 12.1 Specular and Diffuse Reflections.

Figure 12-6: Reflection of Light in Microfacet Mode



In Figure 12-6, the lighting model is shown in the image to the left. The same scenario is represented using vectors in the image to the right.

The greater the number of microfacets there are that reflect light toward the viewer, the more light will be reflected in this direction. Here, we introduce the distribution function $D(X)$. This function represents the proportion of microfacets whose normals point in the X direction. The amount of light that is reflected toward the viewer is in turn proportional to $D(H)$. Microfacet reflections are represented by Equation 12-9 below, in which H represents the half-vector, and the tilde represents proportionality.

Equation 12-9: Microfacet Reflections

$$reflectance \sim D(\mathrm{H})$$

D is also known as the *slope distribution function* because it expresses how the slopes of the facets are distributed.

12.3.2 Blinn-Phong Model

The Blinn-Phong equation (Equation 12-1) in section 12.1 Specular and Diffuse Reflections is in accordance with Equation 12-9, the relationship derived in the previous section. The dot product $N \cdot H$ is equivalent to the cosine of the angle formed by the local normal of a microfacet and the average normal of the surface when viewed macroscopically. The value of the dot product is also an index that indicates how different a given local normal is from the average normal. This means that the equation below can be seen as following the microfacet approach.

Equation 12-10: Microfacet Approach to Blinn-Phong Model

$$reflectance = some\_constant \times D(\mathrm{N} \cdot \mathrm{H})$$

The simplest-case Blinn-Phong reflection model is a simple microfacet model that uses a power function as its distribution function. Section 0 provides a more accurate mathematical model conceived of by R. Cook and K. Torrance[2].

The generalized Blinn-Phong model uses $N \cdot H$ as an argument to another distribution function. The Gaussian function is one example of a possible distribution function.

[2] Cook, Robert L., and Torrance, Kenneth E. A Reflectance Model for Computer Graphics, ACM Computer Graphics, SIGGRAPH 1981 Proceedings, 15(4), pp. 307-316

Equation 12-11: Gaussian Distribution Function

$$D(N \cdot H) = \exp\left(-\frac{\alpha \cos(N \cdot H)^2}{m^2}\right)$$

When the value of $N \cdot H$ is close to 1.0 (in other words, at the center of highlights), using either a Gaussian function or a power function as the distribution function will yield similar behavior. Likewise, when the shininess s in Equation 12-1 is $= 2/m\hat{}2$, the values of the highlights will be nearly the same regardless of whether a Gaussian function or a power function is used as the distribution function.

The figure below shows the result of a sample implementation using DMP fragment lighting.

Figure 12-7: Result of a Sample Implementation Using DMP Fragment Lighting

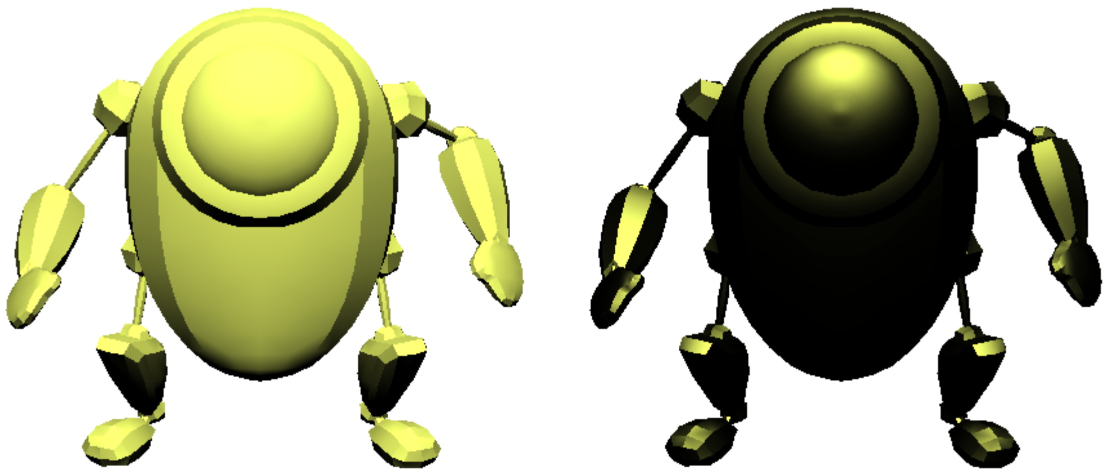


Figure 12-7 shows two examples of Blinn-Phong shading with a power distribution function, with the specular shininess s set to 1 (left) and 10 (right).

Implementation Using DMP Fragment Lighting

Code 12-1 below is an example implementation of the Blinn-Phong model using DMP fragment lighting. This example uses layer configuration 0.

Code 12-1: Example Implementation of the Blinn-Phong Model

```
GLfloat ms1[] = {0.f, 0.f, 0.f, 1.f};
glUniform4fv(LOC("dmp_FragmentMaterial.specular1")), 1, ms1);
glUniform1i(LOC("dmp_LightEnv.lutEnabledRefl")), GL_FALSE);

glUniform1i(LOC("dmp_LightEnv.lutEnabledD0"), GL_TRUE);
glUniform1i(LOC("dmp_LightEnv.absLutInputD0"), GL_TRUE);
glUniform1i(LOC("dmp_LightEnv.lutInputD0"), GL_LIGHT_ENV_NH_DMP);
glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor1"), GL_FALSE);

for (j = 0 ; j < 256 ; j++)
    lut[j] = powf((float)j/256.f, 2.0f);
for (j = 0 ; j < 255 ; j++)
    lut[j + 256] = lut[j+1] - lut[j];
lut[255 + 256] = 1.f - lut[255];
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut);
glUniform1i(LOC("dmp_FragmentMaterial.samplerD0"), 0);
```

This implementation uses a lookup table that ranges from 0.0 to 1.0. The arguments are absolute values; since the fronts of the surfaces face the light source, $N \cdot H$ will never evaluate to a negative value. The next section, 12.3.3 Cook-Torrance Model, explains the specification of the reserved uniform `dmp_FragmentLightSource[0].geomFactor1`.

Figure 12-7 shows the rendered results if the power function is used, with s set to the two values 1.0 and 10.0.

If the `specular1` component is set to zero, only one term is used for the secondary color of DMP fragment lighting. When two terms are present, this indicates a two-layered Blinn-Phong model. For example, if we consider the layer defined by $s = 90$ to be the second layer, we would assign some values to the `specular1` properties of the light and the material, and set the reserved uniform `dmp_LightEnv.lutEnabledD1` to `GL_TRUE`.

Code 12-2: Assigning Values to the "specular1" Properties of the Light and Material

```
GLfloat ls1[] = {1.f, 0.01f, 0.25f, 1.f};
GLfloat ms1[] = {1.f, 1.f, 1.f, 1.f};
glUniform4fv(LOC("dmp_FragmentLightSource[0].specular1"), 1, ls1);
glUniform4fv(LOC("dmp_FragmentMaterial.specular1"), 1, ms1);
glUniform1i(LOC("dmp_LightEnv.lutEnabledRefl"), GL_FALSE);
```

We then create a lookup table, bind the lookup table to distribution 1, and specify the arguments to the table.

Code 12-3: Creating a Lookup Table and Binding It to Distribution 1

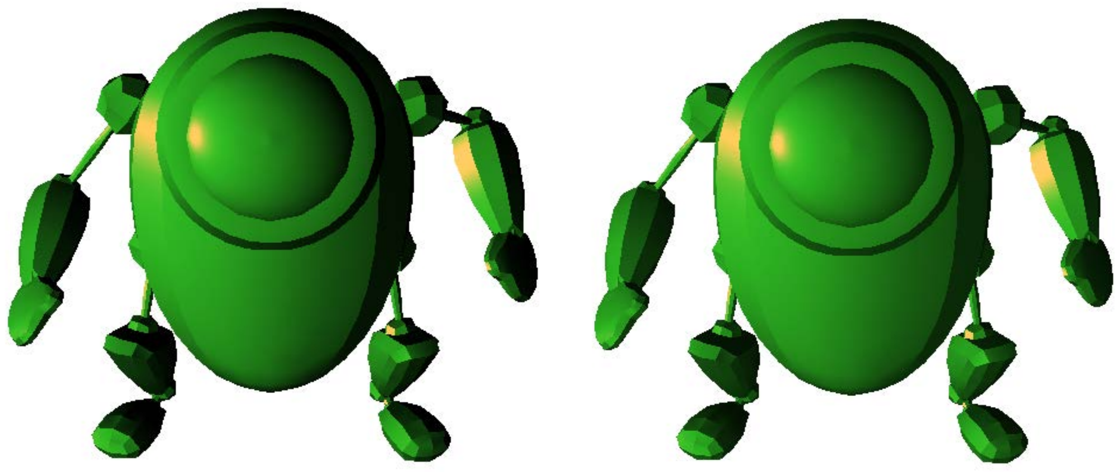
```
  glUniform1i(LOC("dmp_LightEnv.lutEnabledD1"), GL_TRUE);
  glUniform1i(LOC("dmp_LightEnv.lutInputD1"), GL_LIGHT_ENV_NH_DMP);
  glUniform1i(LOC("dmp_LightEnv.absLutInputD1"), GL_TRUE);
  glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor1"), GL_FALSE);

  for (j = 0 ; j < 256 ; j++)
      lut[j] = powf((float)j/256.f, 90.f);
  for (j = 0 ; j < 255 ; j++)
      lut[j + 256] = lut[j + 1] - lut[j];
  lut[255 + 256] = 1.f - lut[255];
  glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerD1"), 1);
```

In the case of the generalized Blinn-Phong model, other functions can be used in place of the power function.

For example, the right-hand image in Figure 12-8 below shows the results if `powf((float)j/256.f, 90.f)` is replaced with `gaussian((float)j/256.f, 0.149f)`, and `powf((float)j/256.f, 2.f)` is replaced with `gaussian((float)j/256.f, 1.f)`.

Here, a Gaussian function is used with m such that $s = 2/m$^2 to make the distribution at the center of the highlight appear the same as if a power function had been used as the distribution function.

Figure 12-8: Example of a Generalized Blinn-Phong Model



The image on the left in Figure 12-8 was created using a power function. The image on the right was created using a Gaussian function. The highlights resemble each other where $N \cdot H$ is close to 1, but the differences are more apparent in the darker areas.

12.3.3 Cook-Torrance Model

To represent the amount of reflected light more accurately, this lighting model multiplies the distribution function D by the Fresnel reflection F of the microfacets that reflect light toward the view vector V by the distribution function.

Equation 12-12: Basic Cook-Torrance Model

$$reflectance \sim F \times D$$

The F in Equation 12-12 is called the *Fresnel factor* or *Fresnel term*.

As stated in section 12.2 Fresnel Reflectance, the Fresnel reflection F is expressed as a cosine function of the angle formed by a light vector and a normal. Here, the normal is the local normal of a microfacet, and is the same as the half-angle vector H. As a consequence, $L \cdot N$ (the cosine of the angle formed by the light vector and the normal) is equivalent to $L \cdot H$.

In this model, the distribution function $D(N \cdot H)$ is multiplied by $F(L \cdot H, \eta)$. In DMP fragment lighting, $V \cdot H$ is used instead of $L \cdot H$. The reason for this is that H is the half-angle vector of L and V, so $L \cdot H$ is equivalent to $V \cdot H$.

Equation 12-13: Basic Cook-Torrance Model in DMP Fragment Lighting

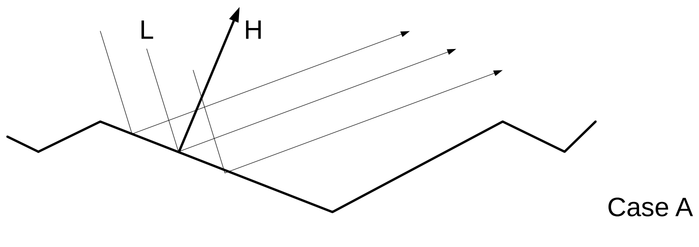
$$reflectance \sim F(V \cdot H, \eta) \times D(N \cdot H)$$

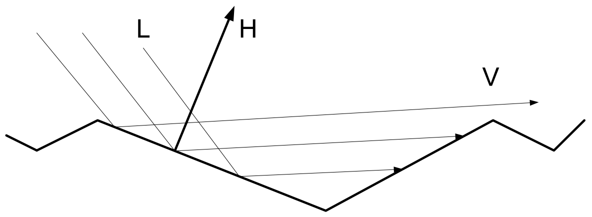
Furthermore, the Cook-Torrance model takes into account the fact that facets can cast shadows on each other (see Figure 12-9 below). For example, even if a microfacet is facing a direction that will cause it to reflect light (Case A), it's possible that not all of the light reflected by the microfacet will reach the viewer (Case B). Likewise, the opposite situation is also possible, namely that some of the light may not reach the microfacet (Case C). These possibilities are included in the model by adding another factor, called the *geometry factor*.

Equation 12-14: Basic Cook-Torrance Model with Geometry Factor

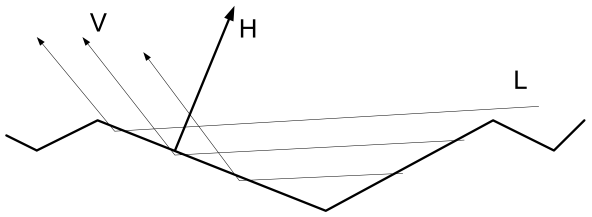
$$reflectance \sim F(V \cdot H, \eta) \times D(N \cdot H) \times geometry\_factor$$

Figure 12-9: Details of the Geometry Factor





Case B



Case C

To represent Equation 12-14 above more accurately, it is necessary to multiply it by a parameter that takes the angle into account. See the *viewing geometry factor* $(= 1/N \cdot V)$[3] for more details. In DMP fragment lighting, this parameter and the geometry factor are grouped together as a single factor, represented by the symbol G. This is called the *combined angular and geometry factor*. This yields the Cook-Torrance models shown below.

Equation 12-15: Cook-Torrance Model for Dielectric Materials

$$Cook\_Torrance = F(V \cdot H, \eta) \times D(N \cdot H) \times G$$

Equation 12-16: Cook-Torrance Model for Metals

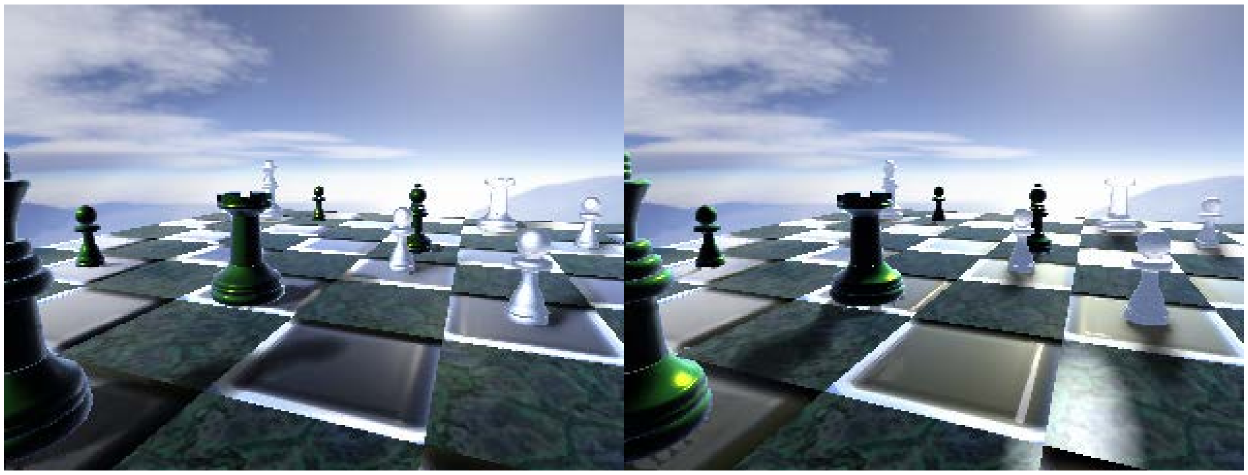
$$Cook\_Torrance = F(V \cdot H, \eta, \kappa) \times D(N \cdot H) \times G$$

From this point onward, the geometry factors are referred to collectively and simply as G. (This refers to the geometry factor defined by Cook-Torrance, multiplied by the viewing geometry factor. G is further transformed using a proprietary method and is implemented by an approximation formula. For details, see the *DMPGL Specifications*.)

G approaches infinity as the angle formed by the view vector and the light vector approaches 180°. This means that Cook-Torrance highlight values become very large if the incident light is coming from straight ahead.

[3] Watt, Alan. 3D Computer Graphics, 3rd Edition, Addison-Wesley Publishing Ltd, Addison-Wesley Publishing Company Inc., 2000, pp. 21

Figure 12-10: Effect of the Geometry Factor



As seen in Figure 12-10 above, the greater the angle between the view vector and the light vector, the brighter the highlights.

The Cook-Torrance model uses the Beckmann function as its distribution function.

Equation 12-17: Standard Beckmann Function

$$D = \frac{1}{m^2(N \cdot H)^4} e^{\frac{((N \cdot H)^2 - 1)}{m^2 (N \cdot H)^2}}$$

Here, if m is small (Beckmann distributions are usually such that $m < 1$), m has the following significance:

"The proportion of facets whose slope is m is $1/e$ times the proportion of facets with a slope of zero."

When $s = \frac{2}{m^2} = 2(\frac{1}{m_B} + 2)$, three different distribution functions yield very similar values in the proximity of $N \cdot H = 1$ and will yield very similar values: the power function, the Gaussian function, and the Beckmann function.

The Beckmann function is normalized so that the value of its integral is one. However, when m is small, the values of this function in the proximity of $N \cdot H = 1$ become very large. Because DMP fragment lighting implements all functions that contain distribution functions as lookup tables, their values can range only from 0.0 to 1.0. As a result, it is best to use a distribution function shaped such that its maximum value is one. DMP fragment lighting uses the Beckmann distribution shown below, without the m^2 in the denominator.

Equation 12-18: Modified Beckmann Function Used by DMP Fragment Lighting

$$D = \frac{1}{(N \cdot H)^4} e^{\frac{((N \cdot H)^2 - 1)}{m^2 (N \cdot H)^2}}$$

The refractive index η and absorption coefficient κ of the Fresnel equations depend on the color. If $F(V \cdot H, \eta)$ is expressed as three components (red, green, and blue), Equation 12-15 becomes the following:

Equation 12-19: DMP Cook-Torrance Model Expressed Using Color Components

$$Cook\_Torrance = Frgb(V \cdot H) \times D(N \cdot H) \times G$$

Here, $Frgb(V \cdot H)$ consists of three components: $Fr(V \cdot H)$, $Fg(V \cdot H)$, and $Fb(V \cdot H)$.

To find $Frgb(V \cdot H)$ easily, use the η and κ values that correspond to the wavelengths of the red, green, and blue components to calculate F .[4] For example, the color gold would be represented by the following values for η, κ, Fr, Fg, and Fb.

$$\eta = 0.183521, \kappa = 2.959155 \quad 630nm$$
$$\eta = 0.516924, \kappa = 2.276178 \quad 525nm$$
$$\eta = 1.464924, \kappa = 1.1860113 \quad 455nm$$

$$Fr = F(V \cdot H, 0.183521, 2.959155)$$
$$Fg = F(V \cdot H, 0.516924, 2.276178)$$
$$Fb = F(V \cdot H, 1.464924, 1.1860113)$$

As a consequence, if $V \cdot H$ is close to 1, the inequality $Fr > Fg > Fb$ will apply, and this causes gold to appear yellow.

With metals, the Fresnel reflection has almost no dependence on the angle. As a result, the Fresnel factor can sometimes be treated as a constant. This type of Cook-Torrance model is known as a *simplified Cook-Torrance model*.

Equation 12-20: Simplified Cook-Torrance Model

$$Simplified\_Cook\_Torrance = Rrgb \times D(N \cdot H) \times G$$

$Rrgb$ consists of three constants: Rr, Rg, and Rb. Variants of the Cook-Torrance model that use angle-dependent Fresnel factors are known as *full-scope Cook-Torrance models*. See *Glassner* for a list of the refractive indices and absorption coefficients for silver, copper, aluminum, and other metals.

12.3.3.1 Implementation Using DMP Fragment Lighting

Full-scope Cook-Torrance models can be implemented in DMP fragment lighting using layer configurations 4, 5, and 7. The following example has been implemented using layer configuration 4. First, since we're only using the second term of the lighting equation, we set the material's `specular0` component to zero.

Code 12-4: Setting the Material's specular0 Component to Zero

```
GLfloat ms0[] = {0.f, 0.f, 0.f, 0.f};
glUniform4fv(LOC("dmp_FragmentMaterial.specular0"), 1, ms0);
```

We then set the color of the light source's `specular1` component to (1.0,1.0,1.0). This is because the color is obtained from the Fresnel factor.

[4] Glassner, Andrew S. Principles of Digital Image Synthesis, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1995

Code 12-5: Setting the Light Source's specular1 Component

```
GLfloat ls1[] = {1.f, 1.f, 1.f, 1.f};
glUniform4fv(LOC("dmp_FragmentLightSource[0].specular1"), 1, ls1);
glUniform1i(LOC("dmp_LightEnv.lutEnabledRefl"), GL_TRUE);
```

Next, we prepare four lookup tables. Three of these are used as Fresnel factors and take $V \cdot H$ as an argument, and the other one is used as a distribution function and takes $N \cdot H$ as an argument.

Code 12-6: Preparing Four Lookup Tables

```
glUniform1i(LOC("dmp_LightEnv.config"),GL_LIGHT_ENV_LAYER_CONFIG4_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputRR"), GL_LIGHT_ENV_VH_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputRG"), GL_LIGHT_ENV_VH_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputRB"), GL_LIGHT_ENV_VH_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputD1"), GL_LIGHT_ENV_NH_DMP);
glUniform1i(LOC("dmp_LightEnv.absLutInputRR"), GL_FALSE);
glUniform1i(LOC("dmp_LightEnv.absLutInputRG"), GL_FALSE);
glUniform1i(LOC("dmp_LightEnv.absLutInputRB"), GL_FALSE);
glUniform1i(LOC("dmp_LightEnv.absLutInputD1"), GL_FALSE);
```

We then enable the combined angular and geometry factor.

Code 12-7: Enabling the Combined Angular and Geometry Factor

```
glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor1"), GL_TRUE);
```

Next, we configure three of the lookup tables using ?? and ?? values that represent gold.

© 2009–2012 Nintendo
CONFIDENTIAL

Code 12-8: Configuring Three Lookup Tables to Express Gold

```
  for (j = 0; j < 128; j++)
      lut[j] = nk_fresnel((float)j/128.f, 0.183521f, 2.959155f);
  for (j = 0; j < 127; j++)
      lut[j + 256] = lut[j+1] - lut[j];
  lut[127 + 256] = nk_fresnel(1.f, 0.183521f, 2.959155f) - lut[127];
  glBindTexture(GL_LUT_TEXTURE0_DMP, lutids[0]);
  glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerRR"), 0);

  for (j = 0; j < 128; j++)
      lut[j] = nk_fresnel((float)j/128.f, 0.516924f, 2.276178f);
  for (j = 0; j < 127; j++)
      lut[j + 256] = lut[j+1] - lut[j];
  lut[127 + 256] = nk_fresnel(1.f, 0.516924f, 2.276178f) - lut[127];
  glBindTexture(GL_LUT_TEXTURE1_DMP, lutids[1]);
  glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerRG"), 1);

  for (j = 0; j < 128; j++)
      lut[j] = nk_fresnel((float)j/128.f, 1.464924f, 1.860113f);
  for (j = 0; j < 127; j++)
      lut[j + 256] = lut[j+1] - lut[j];
  lut[127 + 256] = nk_fresnel(1.f, 1.464924f, 1.860113f) - lut[127];
  glBindTexture(GL_LUT_TEXTURE2_DMP, lutids[2]);
  glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerRB"), 2);
```

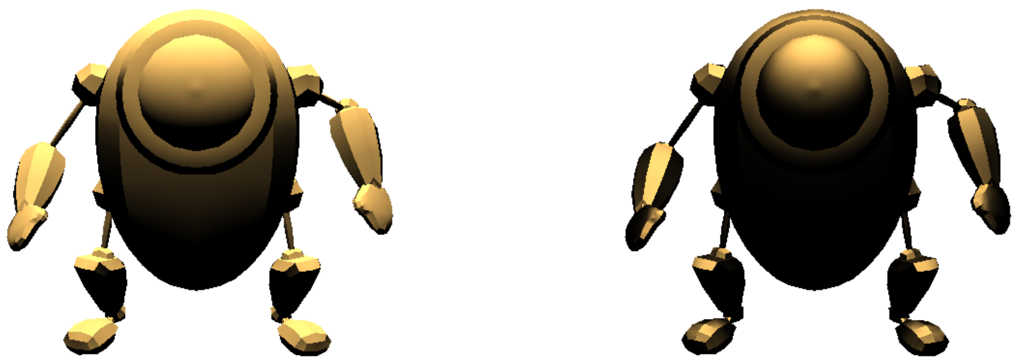
Finally, we configure the Beckmann distribution function.

Code 12-9: The Beckmann Distribution Function

```
  for (j = 1; j < 128; j++)
      lut[j] = beckmann((float)j/128.f, 1.f);
  for (j = 0; j < 127; j++)
      lut[j + 256] = lut[j+1] - lut[j];
  lut[127 + 256] = beckman(1.f, 1.f) - lut[127];
  glBindTexture(GL_LUT_TEXTURE3_DMP, lutids[3]);
  glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerD1"), 3);
```

Figure 12-11 includes the result of simulating a gold appearance using a Beckmann function with $m = 1$.

Figure 12-11: A Cook-Torrance Model for Gold



For these images, the Beckmann distribution used the values $m = 1.0$ (left) and $m = 5.5$ (right).

12.3.4 Schlick Anisotropic Model

The models described so far have assumed that the surfaces have the same lighting characteristics in all directions—in other words, these were all isotropic lighting models. Under this assumption, reflections are always symmetrical with respect to the normal. The opposite case, in which surfaces have characteristics with a directional bias, is referred to as *anisotropic*. Consider the case of brushed metal as an example; the direction of the brushed pattern constitutes the directional bias. Whereas for cloth, the threads that make up the surface have a certain directionality to them.

The light-scattering properties of an anisotropic surface vary as a function of what direction along the surface you are considering. To consider reflections on these types of surfaces, we introduce another unit vector called the *tangent vector* (see Figure 12-12 below).

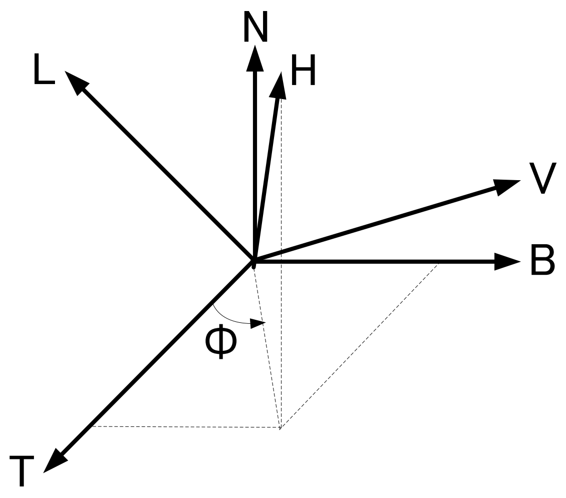
© 2009–2012 Nintendo
CONFIDENTIAL

Schlick[5] proposed that in order to represent anisotropic reflection, the distribution function should be made up of two functions. One of these functions depends on $N \cdot H$, and the other depends on $\cos(\varphi)$. (Here, φ represents the angle formed by the tangent vector T and the projection of the vector H on the tangent plane.) The distribution function D can therefore be represented by the following equation.

Equation 12-21: Distribution Function for Schlick Anisotropic Model

$$D = Z(N \cdot H) \times A(\cos(\varphi))$$

Figure 12-12: Relationships Between the Angles and Vectors Used in Anisotropic Reflection Models



Schlick proposed the following for these two functions Z and A.

Equation 12-22: Schlick's Proposed Zenith and Azimuth Functions

$$Z(t) = \frac{r}{(1 + rt^2 - t^2)^2}$$

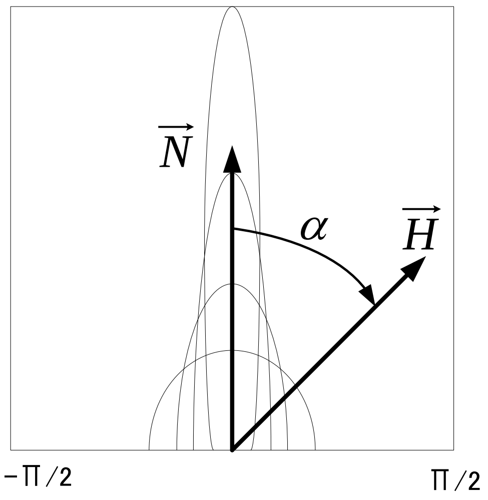
$$A(\omega) = \sqrt{\frac{p}{p^2 - p^2\omega^2 + \omega^2}}$$

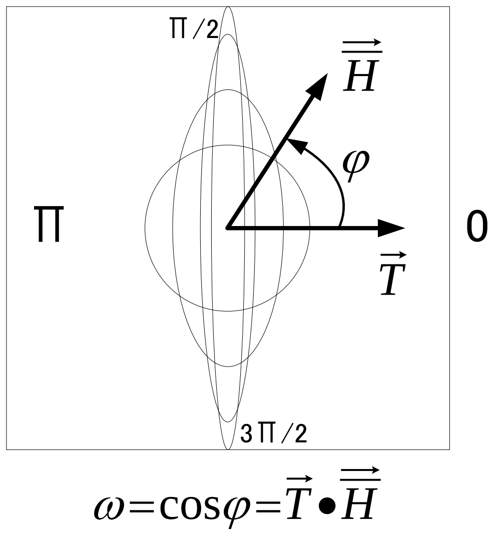
The r parameter characterizes the roughness of the surface. When $r = 1$, Z is a constant, which indicates that the surface has a pure diffuse reflection. As r approaches 0, the reflection elongates further in the direction of the specular light. When $r = 0$, the surface exhibits total specular reflection.

The p parameter characterizes the anisotropy of the surface. When $p = 1$, A is a constant, which indicates that the surface has no anisotropy whatsoever. As p approaches 0, the anisotropy increases. When $p = 0$, the surface exhibits complete anisotropy.

[5] Schlick, Christophe. An Inexpensive BRDF Model for Physically-Based Rendering, Computer Graphics Forum, 13(3), pp. 233-246 (1994)

Figure 12-13: Relationship Between the Zenith-Angle and Z Function Value (Left) and the Azimuth-Angle and A Function Value (Right) in the Polar Coordinate System





The figure on the left shows the relationship between the zenith angle and the Z function value for four types of r where α ranges between $-\pi/2$ and $\pi/2$. The figure on the right shows the relationship between the azimuth angle and the A function value for four types of p where φ ranges between 0 and 2π .

The geometry factor designated by Schlick is different from the one used in the Cook-Torrance model. However, its behavior is similar to the geometry factor of the Cook-Torrance model if we assume that the angular term and the geometry factor have been combined into a single variable, as was described earlier in section 12.3.3 Cook-Torrance Model. In other words, highlights become extremely bright as the angle formed by the view vector and the normal approaches 90°. This phenomenon occurs when the incident light is coming from straight ahead and r is not very large ($r < 1$). In the Cook-Torrance model, the G factor of surfaces with low degrees of roughness becomes very large as the angle formed by the view vector and the light vector approaches 180°.

Based on these facts, the same combined angular and geometry factor that was used for the Cook-Torrance model can also be used for the Schlick model. The equation for the Schlick model in DMP fragment lighting is shown below.

Equation 12-23: Schlick Model Used in DMP Fragment Lighting

$$Schlick = Frgb(V \cdot H) \times Z(N \cdot H) \times A(\cos(\varphi)) \times G$$

The terms $Frgb$ and G are the same as the corresponding terms in section 12.3.3 Cook-Torrance Model.

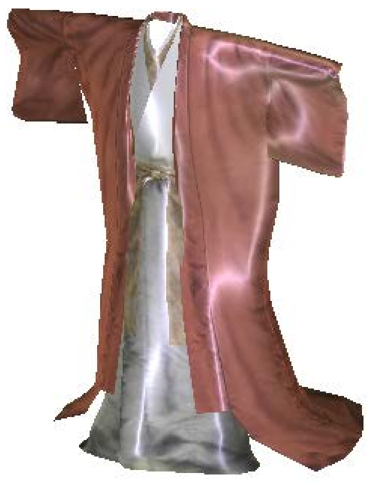
12.3.4.1 Implementation Using DMP Fragment Lighting

The sample code below is an implementation of a Schlick model using DMP fragment lighting.

Code 12-10: Sample Implementation of the Schlick Model

```
glUniform1i(LOC("dmp_LightEnv.absLutInputD1"), GL_TRUE);
glUniform1i(LOC("dmp_LightEnv.absLutInputRB"), GL_TRUE);
glUniform1i(LOC("dmp_LightEnv.absLutInputRG"), GL_TRUE);
glUniform1i(LOC("dmp_LightEnv.absLutInputRR"), GL_TRUE);

glUniform1i(LOC("dmp_LightEnv.lutInputD1"), GL_LIGHT_ENV_CP_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputRB"), GL_LIGHT_ENV_NH_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputRG"), GL_LIGHT_ENV_NH_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputRR"), GL_LIGHT_ENV_NH_DMP);

glUniform1i(LOC("dmp_LightEnv.lutEnabledRefl"), GL_TRUE);

glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor0"), GL_FALSE);
glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor1"), GL_FALSE);

glUniform1i(LOC("dmp_LightEnv.config"), GL_LIGHT_ENV_LAYER_CONFIG7_DMP);
glUniform1i(LOC("dmp_LightEnv.bumpMode"),GL_LIGHT_ENV_BUMP_AS_BUMP_DMP);
glUniform1i(LOC("dmp_LightEnv.bumpSelector"), GL_TEXTURE1);
```

Figure 12-14: Sample Rendering Lit Using a Schlick Model Implemented Using DMP Fragment Lighting



12.3.5 Subsurface-Scattering Model

When light enters an object made of a translucent material like skin, wax, or marble, the light is scattered internally in many directions, then emitted from the object. In other words, such materials are characterized by strong subsurface scattering. In such materials, the position from which a ray of light is reflected is different from the position of incidence where it entered the material. As a result, it is not possible to apply one of the BRDF models described in section 12.1 Specular and Diffuse Reflections to this type of material.

Jensen, et al. proposed a model that is based on the diffusion theory. This model takes the integral of the incident light and the diffuse reflection function over a certain region of the surface. The size of the integral region is roughly on the same order as the length of the mean free path. For typical materials, this is between one and several millimeters.

Let's consider an object that is made of a translucent material: for example, a hand. If only a single light source illuminates the object in a dark environment, the convex areas appear to have the highest degree of transparency. The same is true under complicated lighting environments. In other words, the rendering of the convex areas is most important with subsurface scattering.

With that in mind, if we consider the convex areas to be areas on the surfaces of spheres with radius S, we can take the integral of those areas using the formula proposed by Jensen et al. (The radius S is much longer than the length of the mean free path.) If we also assume that the surface is rough, the reflection model can be expressed using Equation 12-24 below.

Equation 12-24: Subsurface Scattering Equation

$$Subsurface\_scattering = Rrgb(L \cdot N) \times T(V \cdot N)$$

$Rrgb(L \cdot N)$ is composed of two terms. These terms are the *diffuse* (or *Lambertian*) *term* and the *wrapping term*. The wrapping term represents the penetration of light into surface regions that are not lit directly.

Equation 12-25: Definitions of Diffuse and Wrapping Terms for Subsurface Scattering

$$Rrgb(L \cdot N) = r(L \cdot N) + W(L \cdot N)$$

The terms are defined as follows:

$$r = 0.5\alpha'\left(e^{-\beta} + e^{-\beta'}\right)$$

$$\beta = \sqrt{3(1-\alpha')}$$

$$\beta' = \beta B$$

$$B = 1 + \frac{4}{3}\frac{(1+Fdr)}{(1-Fdr)}$$

$$W(L \cdot N) = \frac{\alpha' Y i_0}{4\pi(1 + i_1 h)}$$

$$Y = l\frac{\sqrt{1-(L \cdot N)^2}}{S}$$

$$h = \frac{L \cdot N}{Y}$$

$$i_0 = \sqrt{2\pi}\left(e^{-\beta}\beta^{-0.5} + Be^{-\beta'}\beta'^{-0.5}\right)$$

$$i_i = \frac{\pi(e^{-\beta} + e^{-\beta'})}{i_0}$$

See Jensen et al.[6] for more information about the terms α' (albedo) and Fdr.

12.3.5.1 Implementation Using DMP Fragment Lighting

The sample code below shows a subsurface scattering implementation based on DMP fragment lighting.

[6] Jensen, H. W., Marschner, S., Levoy, M., and Hanrahan, P. A practical model for subsurface light transport, SIGGRAPH 2001 Proceedings, E. Fiume, Ed., Annual Conference Series, pp. 511–518

Code 12-11: Sample Implementation of Subsurface Scattering

```
  glUniform1i(LOC("dmp_LightEnv.absLutInputD0"), GL_FALSE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputD1"), GL_TRUE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputSP"), GL_FALSE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputFR"), GL_TRUE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputRB"), GL_FALSE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputRG"), GL_FALSE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputRR"), GL_FALSE);

  glUniform1i(LOC("dmp_LightEnv.lutInputRB"), GL_LIGHT_ENV_LN_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputRG"), GL_LIGHT_ENV_LN_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputRR"), GL_LIGHT_ENV_LN_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputD1"), GL_LIGHT_ENV_NV_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputD0"), GL_LIGHT_ENV_NH_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputFR"), GL_LIGHT_ENV_NV_DMP);

  glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor0"), GL_FALSE);
  glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor1"), GL_FALSE);

  glUniform1f(LOC("dmp_LightEnv.lutScaleRR"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleRG"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleRB"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleD0"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleD1"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleSP"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleFR"), 2.f);

  glUniform1i(LOC("dmp_LightEnv.lutEnabledRefl"), GL_TRUE);

  glUniform1i(LOC("dmp_LightEnv.config"), GL_LIGHT_ENV_LAYER_CONFIG7_DMP);
  glUniform1i(LOC("dmp_LightEnv.fresnelSelector" ),
GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP);
  glUniform1i(LOC("dmp_LightEnv.clampHighlights"), GL_FALSE);

  glUniform1i(LOC("dmp_LightEnv.lutEnabledD0"), GL_TRUE);
  glUniform1i(LOC("dmp_LightEnv.lutEnabledD1"), GL_TRUE);

  GLfloat qlut[3][512], lut[512];
  int j, co;
  for (co = 0; co < 3; co++)
      memset(qlut[co], 0, sizeof(qlut[0]));
  memset(lut, 0, sizeof(lut));
  for (j = 0 ; j < 128; j++) {
      LN = (float)j/128.f;
      kappa = 1.0f - LN * LN;
      for (co = 0; co < 3; co++) {
          if (LN > 0.0)
          lut[co][j] = mat.dif_refl[co] * LN;
          gamma = mat.zr[co] * sqrt(kappa);
          h = fabsf(LN) / gamma;
          qlut[co][j] += mat.i0[co] * (1.0f / (1.0f + mat.i1[co] * h)) *
gamma * mat.albedo[co] * 0.25f * REV_PI;
          qlut[co][j] = pow(qlut[co][j], display_gamma);
      }
  }
  for (j = 128 ; j < 256; j++) {
      LN = (float)(j - 256) /128.f;
      kappa = 1.0f - LN * LN;
      for (co = 0; co < 3; co++){
          if (LN > 0.0)
              lut[co][j] = mat.dif_refl[co] * LN;
          gamma = mat.zr[co] * sqrt(kappa);
          h = fabsf(LN) / gamma;
          qlut[co][j] += mat.i0[co] * (1.0f / (1.0f + mat.i1[co] * h)) *
          gamma * mat.albedo[co] * 0.25f * REV_PI;
          qlut[co][j] = pow(qlut[co][j], display_gamma);
      }
  }
  for (j = 0 ; j < 127; j++)
      for (co = 0; co < 3; co++)
          qlut[co][j + 256] = qlut[co][j+1] - qlut[co][j];
  for (co = 0; co < 3; co++)
      qlut[co][127 + 256] = pow(mat.dif_refl[co], display_gamma) -
qlut[co][127];
  for (j = 128 ; j < 255 ; j++)
      for (co = 0; co < 3; co++)
          qlut[co][j + 256] = qlut[co][j+1] - qlut[co][j];
  for (co = 0; co < 3; co++)
      qlut[co][255 + 256] = qlut[co][0] - qlut[co][255];
  glBindTexture(GL_LUT_TEXTURE0_DMP, lutids[0]);
  glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512,
0,GL_LUMINANCEF_DMP, GL_FLOAT, qlut[0]);
  glBindTexture(GL_LUT_TEXTURE1_DMP, lutids[1]);
  glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512,
0,GL_LUMINANCEF_DMP, GL_FLOAT, qlut[1]);
```

```
  glBindTexture(GL_LUT_TEXTURE2_DMP, lutids[2]);
  glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512,
0,GL_LUMINANCEF_DMP, GL_FLOAT, qlut[2]);

  for (j = 0 ; j < 256; j++) {
      lut[j] = 1.f - r_fresnel((float)j/256.f, 1.7f, 0.36f, 0.f);
      lut[j] = pow(lut[j], display_gamma);
  }
  for (j = 0 ; j < 255; j++)
      lut[j + 256] = lut[j+1] - lut[j];
  lut[255 + 256] = pow((1.f - r_fresnel(1.f, 1.7f, 0.36f, 0.f)),
display_gamma) - lut[255];
  glBindTexture(GL_LUT_TEXTURE3_DMP, lutids[3]);
  glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
  GL_LUMINANCEF_DMP, GL_FLOAT, lut);

  for (j = 0 ; j < 128; j++)
      lut[j] = beckmann( (float)j/128.f, 0.5f);
  for ( j = 0 ; j < 127 ; j++ )
      lut[j + 256] = lut[j+1] - lut[j];
  lut[127 + 256] = 1.f - lut[127];
  glBindTexture(GL_LUT_TEXTURE4_DMP, lutids[4]);
  glTexImage1D(GL_LUT_TEXTURE4_DMP, 0, GL_LUMINANCEF_DMP, 512,
0,GL_LUMINANCEF_DMP, GL_FLOAT, lut);

  for (j = 0 ; j < 256; j++)
      lut[j] = r_fresnel((float)j/256.f, 2.f, 0.25f, 0.f);
  for (j = 0 ; j < 255; j++)
      lut[j + 256] = lut[j+1] - lut[j];
  lut[255 + 256] = r_fresnel(1.f, 2.f, 0.25f, 0.f) - lut[255];
  glBindTexture(GL_LUT_TEXTURE5_DMP, lutids[5]);
  glTexImage1D(GL_LUT_TEXTURE5_DMP, 0, GL_LUMINANCEF_DMP, 512,
0,GL_LUMINANCEF_DMP, GL_FLOAT, lut);

  glUniform1i(LOC("dmp_FragmentMaterial.samplerRR"), 0);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerRG"), 1);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerRB"), 2);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerD1"), 3);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerD0"), 4);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerFR"), 5);
```

The sample code below shows how to configure the texture combiners. The example uses the combiner function `GL_MULT_ADD_DMP`, which multiplies `Arg0` and `Arg1` and then adds `Arg2` to the result.

Code 12-12: Sample Texture Combiner Settings

```
glUniform1i(LOC("dmp_Texture[0].samplerType"), GL_TEXTURE_CUBE_MAP);
glUniform1i(LOC("dmp_TexEnv[0].combineRgb"), GL_ADD);
glUniform1i(LOC("dmp_TexEnv[0].combineAlpha"), GL_REPLACE);
glUniform3i(LOC("dmp_TexEnv[0].operandRgb"),
GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(LOC("dmp_TexEnv[0].operandAlpha"),
GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(LOC("dmp_TexEnv[0].srcRgb"), GL_FRAGMENT_PRIMARY_COLOR_DMP,
GL_FRAGMENT_SECONDARY_COLOR_DMP, GL_PRIMARY_COLOR);
glUniform3i(LOC("dmp_TexEnv[0].srcAlpha"),
 GL_PRIMARY_COLOR, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);

glUniform1i(LOC("dmp_TexEnv[1].combineRgb"), GL_MULT_ADD_DMP);
glUniform1i(LOC("dmp_TexEnv[1].combineAlpha"), GL_REPLACE);
glUniform3i(LOC("dmp_TexEnv[1].operandRgb"),
GL_SRC_COLOR, GL_SRC_ALPHA, GL_SRC_COLOR);
glUniform3i(LOC("dmp_TexEnv[1].operandAlpha"),
 GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(LOC("dmp_TexEnv[1].srcRgb"),
GL_TEXTURE0, GL_FRAGMENT_PRIMARY_COLOR_DMP, GL_PREVIOUS);
glUniform3i(LOC("dmp_TexEnv[1].srcAlpha"),
 GL_PREVIOUS, GL_PREVIOUS, GL_PREVIOUS);
```

Figure 12-15: Sample Rendering of a Material That Causes Subsurface Scattering Using DMP Fragment Lighting

12.3.6 Toon Shading

Toon shading is a technique that only uses a few bands for shading (typically it uses either two or three bands). Each of these bands is shaded using a solid color. To perform toon shading, we prepare a dot product value D (either $N \cdot H$ or)$L \cdot N$), as well as a shading function $f(D)$ that satisfies the requirements listed below.

Consider the sequence $\delta_0, \delta_1, \delta_2, \ldots, \delta_m$, comprising all the values within the range of possible dot product values. The shading function is such that it yields a constant value over each segment δ_{j-1}, δ_j within that sequence.

The values of the function for each segment are multiplied by a constant color.

12.3.6.1 Implementation Using DMP Fragment Lighting

The code sample below implements toon shading using layer configuration 6.

Code 12-13: Sample Toon Shading Implementation Using Layer Configuration 6

```
glUniform1i(LOC("dmp_LightEnv.absLutInputD0"), GL_TRUE);
glUniform1i(LOC("dmp_LightEnv.absLutInputD1"), GL_TRUE);
glUniform1i(LOC("dmp_LightEnv.absLutInputFR"), GL_TRUE);
glUniform1i(LOC("dmp_LightEnv.absLutInputRR"), GL_FALSE);

glUniform1i(LOC("dmp_LightEnv.lutInputRR"), GL_LIGHT_ENV_LN_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputD0"), GL_LIGHT_ENV_LN_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputD1"), GL_LIGHT_ENV_NV_DMP);
glUniform1i(LOC("dmp_LightEnv.lutInputFR"), GL_LIGHT_ENV_NV_DMP);

glUniform1i(LOC("dmp_LightEnv.lutEnabledRefl"), GL_TRUE);

glUniform1i(LOC("dmp_LightEnv.config"), GL_LIGHT_ENV_LAYER_CONFIG6_DMP);

glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor0"), GL_FALSE);
glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor1"), GL_FALSE);
glUniform1i(LOC("dmp_FragmentLightSource[0].spotEnabled"), GL_FALSE);
glUniform1i(LOC("dmp_LightEnv.clampHighlights"), GL_FALSE);
```

Let $f(D) = D$, and use $L \cdot N$ as the dot product. Although different parameters and dot product values can be used for the diffuse and highlight portions, we've used $L \cdot N$ as the dot product for both in our sample code.

The configuration of the diffuse component is shown below.

Code 12-14: Sample Diffuse Configuration

```
 float delta[] = {1.f, 0.7f, 0.5f, -1.f};
 for (j = 127; j >= 0; j--) {
     LN = (float)j/128.f ;
     if (LN > delta[i])
         lut[j] = previous;
     else
     {
         lut[j] = LN;
         previous = lut[j];
         i++;
     }
 }
 for (j = 0 ; j < 127; j++)
     lut[j + 256] = lut[j+1] - lut[j];
 lut[127 + 256] = 0.f;
 for (j = 255; j >= 128; j--) {
     LN = (float)(j - 256) /128.f;
     if (LN > delta[i])
         lut[j] = previous;
     else {
         lut[j] = LN;
         previous = lut[j];
         i++;
     }
 }
 for (j = 128; j < 255; j++)
     lut[j + 256] = lut[j+1] - lut[j];
 lut[255 + 256] = lut[0] - lut[255];
 glBindTexture(GL_LUT_TEXTURE0_DMP, lutids[0]);
 glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut);
 glUniform1i(LOC("dmp_FragmentMaterial.samplerRR"), 0);
```

If the `delta` array is changed, the shaded portion will change as well.

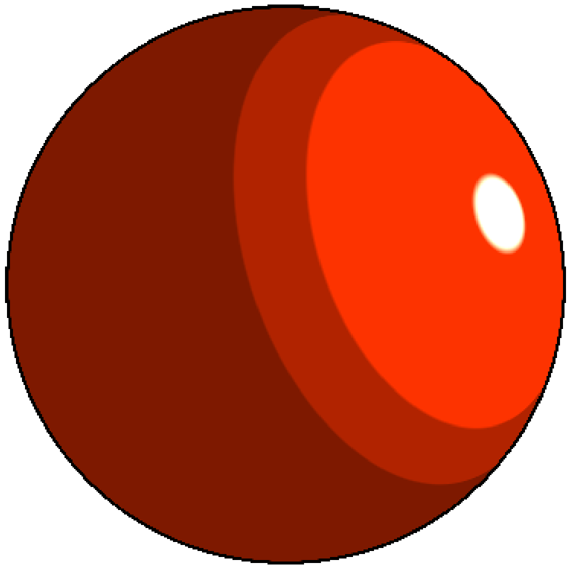
The following code configures the highlights.

Code 12-15: Sample Highlight Configuration

```
  float highlight_eps = 0.01f ;
  for (j = 0; j < 256; j++)
      if ((float)j/256.f <= 1.f - highlight_eps)
          lut[j] = 0.f;
      else
          lut[j] = 1.f;
  for (j = 0; j < 255; j++)
      lut[j + 256] = lut[j+1] - lut[j];
  lut[255 + 256] = 0.f;
  glBindTexture(GL_LUT_TEXTURE1_DMP, lutids[1]);
  glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, lut);
  glUniform1i(LOC("dmp_FragmentMaterial.samplerD0"), 1);
```

Figure 12-16 shows the rendered result.

Figure 12-16: Toon Shading Sample with $L \cdot N$ as the Dot Product and $f(D) = D$ as the Shading Function



Code 12-16 below is another example, this time of toon shading using layer configuration 7. In this code, we've used $Rrgb(L \cdot N)$ as $f(D)$. (See section 12.3.5 Subsurface-Scattering Model for a prior example of using $Rrgb(L \cdot N)$.)

Code 12-16: Sample Toon Shading Implementation Using Layer Configuration 7

```
  glUniform1i(LOC("dmp_LightEnv.absLutInputD0"), GL_FALSE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputD1"), GL_TRUE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputSP"), GL_FALSE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputFR"), GL_TRUE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputRB"), GL_FALSE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputRG"), GL_FALSE);
  glUniform1i(LOC("dmp_LightEnv.absLutInputRR"), GL_FALSE);

  glUniform1i(LOC("dmp_LightEnv.lutInputRB"), GL_LIGHT_ENV_LN_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputRG"), GL_LIGHT_ENV_LN_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputRR"), GL_LIGHT_ENV_LN_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputD1"), GL_LIGHT_ENV_NV_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputD0"), GL_LIGHT_ENV_NH_DMP);
  glUniform1i(LOC("dmp_LightEnv.lutInputFR"), GL_LIGHT_ENV_NV_DMP);

  glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor0"), GL_FALSE);
  glUniform1i(LOC("dmp_FragmentLightSource[0].geomFactor1"), GL_FALSE);

  glUniform1f(LOC("dmp_LightEnv.lutScaleRR"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleRG"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleRB"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleD0"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleD1"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleSP"), 2.f);
  glUniform1f(LOC("dmp_LightEnv.lutScaleFR"), 2.f);

  glUniform1i(LOC("dmp_LightEnv.lutEnabledRefl"), GL_TRUE);

  glUniform1i(LOC("dmp_LightEnv.config"), GL_LIGHT_ENV_LAYER_CONFIG7_DMP);
  glUniform1i(LOC("dmp_LightEnv.fresnelSelector"),
GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP);
  glUniform1i(LOC("dmp_LightEnv.clampHighlights"), GL_FALSE);

  glUniform1i(LOC("dmp_LightEnv.lutEnabledD0"), GL_TRUE);
  glUniform1i(LOC("dmp_LightEnv.lutEnabledD1"), GL_TRUE);
```

Next we use the $Rrgb(L \cdot N)$ expression (see section 12.3.5 Subsurface-Scattering Model for a prior example) to configure the lookup table. When `TOON` is not defined, we use the original $Rrgb(L \cdot N)$ expression.

Code 12-17: Configuring the Lookup Table Using $Rrgb(\boldsymbol{L} \cdot \boldsymbol{N})$

```
  for (j = 127; j >= 0; j--) {
      LN = (float)j/128.f;
  // When TOON is defined, the skin shading function is
  // transformed into steps acc. to the delta[] array
  #ifdef TOON
      if (LN > delta[i]){
          for (co = 0; co < 3; co++)
              qlut[co][j] = previous[co];
          continue;
      }
  #endif
      kappa = 1.0f - LN * LN;
      for (co = 0; co < 3; co++) {
          if (LN > 0.0)
              qlut[co][j] = mat.dif_refl[co] * LN;
          gamma = mat.zr[co] * sqrt(kappa);
          h = fabsf(LN);
          qlut[co][j] += mat.i0[co] * (gamma / (gamma + mat.i1[co] * h)) *
gamma * mat.albedo[co] * 0.25f * REV_PI;
          qlut[co][j] = pow(qlut[co][j], display_gamma);
      }
  // When TOON is defined, the skin shading function is
  // transformed into steps according to the delta[] array
  #ifdef TOON
      for (co = 0; co < 3; co++)
          previous[co] = qlut[co][j];
      i++;
  #endif
  }
  for (j = 255; j >= 128; j--) {
      LN = (float)(j - 256) /128.f;
  // When TOON is defined, the skin shading function is
  // transformed into steps according to the delta[] array
  // In other words, f(LN) = f_sss(LN) is used as the initial shading
function.
  // See Chapter 12 Illumination Models in DMP Fragment Lighting for details.
  #ifdef TOON
      if (LN > delta[i]){
          for (co = 0; co < 3; co++)
              qlut[co][j] = previous[co];
          continue;
```

```
        }
    #endif
        kappa = 1.0f - LN * LN;
        for (co = 0; co < 3; co++) {
            if (LN > 0.0)
                qlut[co][j] = mat.dif_refl[co] * LN;
            gamma = mat.zr[co] * sqrt(kappa);
            h = fabsf(LN);
            qlut[co][j] += mat.i0[co] * (gamma / (gamma + mat.i1[co] * h)) *
                gamma * mat.albedo[co] * 0.25f * REV_PI;
            qlut[co][j] = pow(qlut[co][j], display_gamma);
        }
    // When TOON is defined, the skin shading function is
    // transformed into steps according to the delta[] array
    #ifdef TOON
        for (co = 0; co < 3; co++)
            previous[co] = qlut[co][j];
        i++;
    #endif
    }
    for (j = 0; j < 127; j++)
        for (co = 0; co < 3; co++)
            qlut[co][j + 256] = qlut[co][j+1] - qlut[co][j];
    for (co = 0; co < 3; co++)
        qlut[co][127 + 256] = pow(mat.dif_refl[co], display_gamma) -
qlut[co][127];
    for (j = 128; j < 255; j++)
        for (co = 0; co < 3; co++)
            qlut[co][j + 256] = qlut[co][j+1] - qlut[co][j];
    for (co = 0; co < 3; co++)
        qlut[co][255 + 256] = qlut[co][0] - qlut[co][255];
    glBindTexture(GL_LUT_TEXTURE0_DMP, lutids[0]);
    glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, qlut[0]);
    glBindTexture(GL_LUT_TEXTURE1_DMP, lutids[1]);
    glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, qlut[1]);
    glBindTexture(GL_LUT_TEXTURE2_DMP, lutids[2]);
    glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
GL_LUMINANCEF_DMP, GL_FLOAT, qlut[2]);
    glUniform1i(LOC("dmp_FragmentMaterial.samplerRR"), 0);
    glUniform1i(LOC("dmp_FragmentMaterial.samplerRG"), 1);
    glUniform1i(LOC("dmp_FragmentMaterial.samplerRB"), 2);
```

Figure 12-17: Using a Nonlinear Function as the Shading Function $f(D)$ as the Nonlinear Function

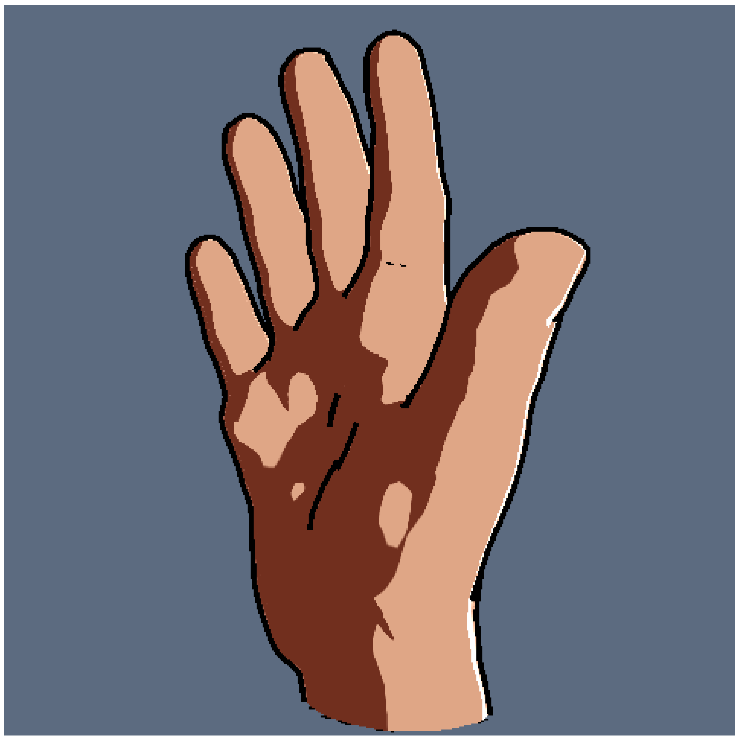


Figure 12-17 above is an example of rendered results with the subsurface scattering function $Rrgb(L \cdot N)$ used for toon shading.

13 DMPGL 2.0 Bump Mapping

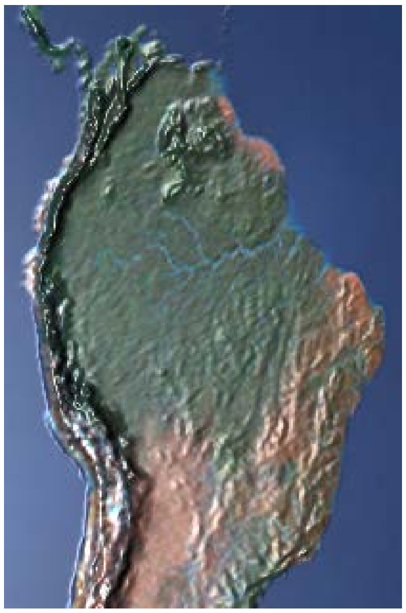
With DMP fragment lighting, bump mapping can be implemented using normal maps. This chapter explains *local-surface space* (a very important concept in implementing bump mapping) as well as the specific bump-mapping-related features and operations of DMPGL 2.0.

13.1 Overview

Bump mapping is a technique for *perturbing* (that is, providing fluctuation to) the normal vectors of objects that are used in the lighting equation. Bump mapping perturbs these normals at each pixel to produce the appearance of bumps and depressions in the surfaces of the objects. Bump mapping makes it possible to use simple geometry with low polygon counts but achieve rendered results that appear to have complicated shapes.

DMP fragment lighting supports bump mapping that uses maps (called *normal maps*) that store perturbed normals that were calculated in advance based on height information.

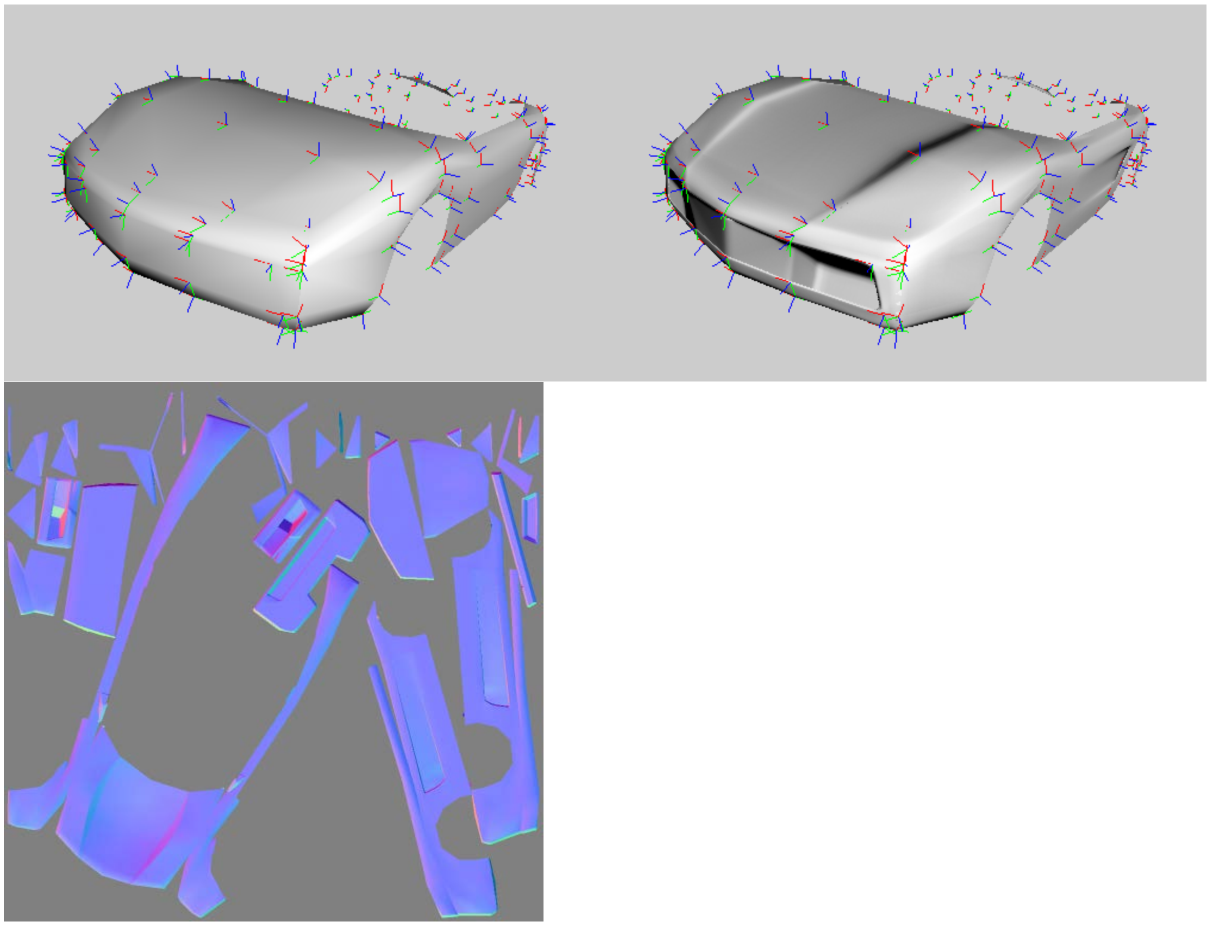
Figure 13-1: Example of Bump Mapping



Although the model used to generate the image in Figure 13-1 above is made from only two triangles, the use of bump mapping allows shading on a per-pixel basis.

© 2009–2012 Nintendo
CONFIDENTIAL

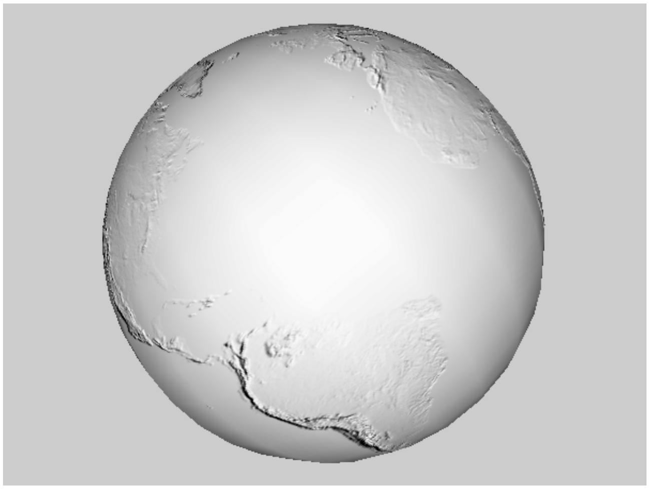
Figure 13-2: Bump Mapping to Make a Low-Polygon Count Model Appear to Have Many Polygons



The image at the upper left in Figure 13-2 is the original polygon model. The image at the upper right is the result of applying a normal map to the original model. The image below is the normal map. Recent modeling tools support features for creating normal maps that can give models with low polygon counts the type of highly detailed appearance seen in this example.

To use bump mapping, you must calculate lighting on a per-fragment basis. OpenGL 2.0 and prior versions do not support fragment lighting as a standard feature. Starting with OpenGL 1.3, a texture combiner function called `GL_DOT3_RGB` was added. It provided support for calculating dot products on a per-fragment basis using the two inputs to the texture combiner. By combining this dot product calculation with cube mapping (used as normalized cube maps), it is possible to calculate the diffuse or specular components on a per-fragment basis. Bump mapping can also be implemented using normal maps.

Figure 13-3: Bump Mapping Using GL_RGB_DOT3



The light vector is converted to the surface-local coordinate system and is then output as a texture coordinate that references a cube map. Because this light vector loses its normalization during the interpolation process, the cube map is used to normalize it. The dot product of the normalized light vector and the perturbed normal is calculated by the texture combiner. (In the example in Figure 13-3, the boundary lines between polygons are visible because the light vectors have not been normalized by a cube map.)

DMP fragment lighting supports per-fragment calculation of the dot products listed below in the same way that it supports the `GL_DOT3_RGB` texture combiner function described earlier. The following are just examples of the dot products supported by DMP fragment lighting. For more details, see the chapter on DMP Fragment Lighting in the *DMPGL 2.0 Specifications*.

- `VH`: Dot product of the view vector and the half-angle vector
- `NH`: Dot product of the normal vector and the half-angle vector
- `LN`: Dot product of the normal vector and the light vector
- `VN`: Dot product of the normal vector and the view vector

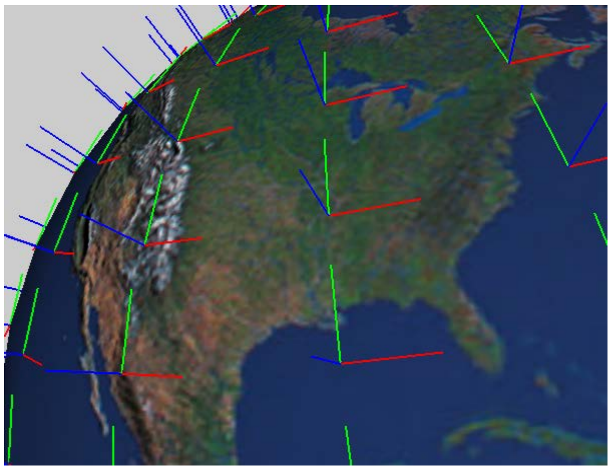
The vectors indicated here are generated by interpolating at each fragment. However, DMP fragment lighting also allows you to replace these normals generated at each fragment with R, G, and B components that are looked up from a texture (these components are taken to be the *x*, *y*, and *z* component of the normal, respectively). Using the lookup values from the texture as the perturbed normals makes bump mapping possible.

13.2 Surface-Local Space and Tangents

As stated in the previous section, the use of values referenced from textures as perturbed normals usually involves storing the normals within the textures in a coordinate system called *surface-local space*. This is true regardless of whether bump mapping is implemented using OpenGL's `GL_DOT3_RGB` texture combiner function or by using DMP fragment lighting.

The surface-local coordinate system assumes that each vertex is at the origin (0.0, 0.0, 0.0) and that the unperturbed surface normal (the original normal of the surface) at each vertex is (0.0, 0.0, 1.0). In other words, normal vectors point toward the positive Z-axis in this coordinate system.

Figure 13-4: Example Surface-Local Coordinate System



The lighting calculation requires that the individual vectors are all defined in the same coordinate system. In other words, one of two approaches is required: (1) converting all vectors used in the calculation except the normals to the surface-local coordinate system, or (2) conversely, converting the perturbed normals defined in the surface-local coordinate system to the coordinate system in which the other vectors are defined.

In either case, the basis vectors that construct the surface-local space are required for the conversion. The normal is one of the basis vectors. Because the basis vectors are orthogonal to each other, defining one additional basis vector also determines the last one. As Figure 13-4 above shows, the basis vectors that make up the surface-local space commonly have different values at each vertex. Here we define a new vertex attribute that, along with the surface normal, is one of the basis vectors. This new vertex attribute is known as either the *tangent vector* or the *tangent*.

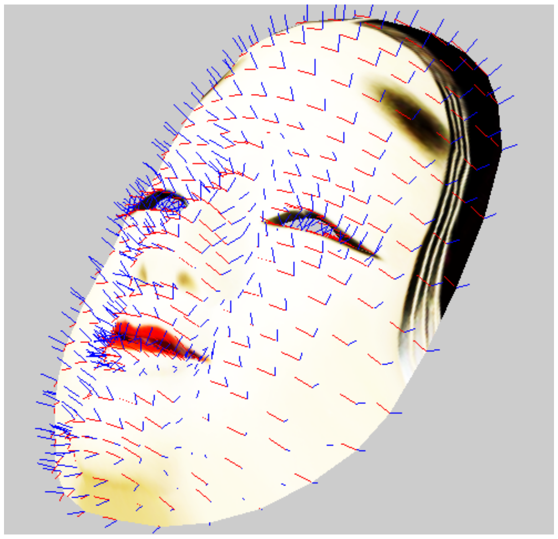
Equation 13-1: Basis Vectors That Define Surface-Local Space

$$rotation\_matrix = \begin{bmatrix} Tx & Ty & Tz \\ Bx & By & Bz \\ Nx & Ny & Nz \end{bmatrix}$$

The matrix in Equation 13-1 above is a rotation matrix made from the basis vectors that form the surface-local space at each vertex. This rotation matrix transforms from object space to the surface-local coordinate system. In this equation, N represents the surface normal, T represents the tangent vector, and B represents the binormal vector.

If we take this approach and use vectors defined in the surface-local coordinate system, representing the surface-local coordinate system defined at each vertex requires that we define an additional vertex attribute for the last of the basis vectors that make up this coordinate system.

Figure 13-5: Example of Surface-Local Space Defined at Each Vertex of an Object

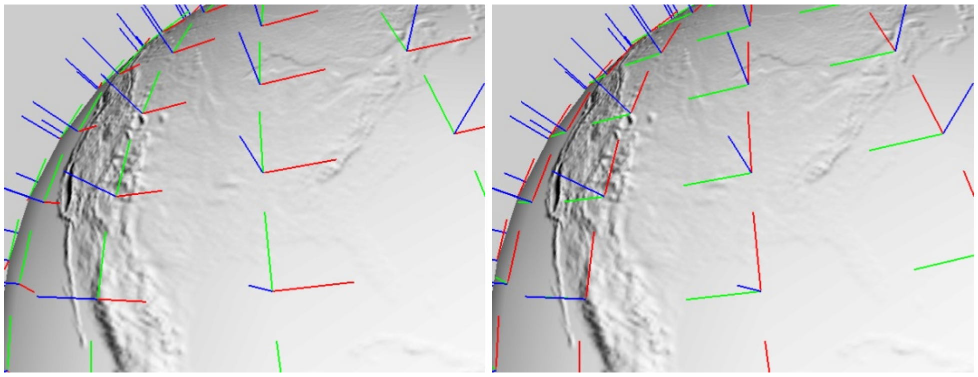


The blue lines superimposed on the Noh mask indicate the normal vectors, and the red lines indicate the tangent vectors. The coordinate system formed by these two vectors and the binormal vector is defined at each vertex.

13.3 Normals Stored in Textures and the Arbitrariness of Surface-Local Space

As stated in the previous section, surface normals are one of the basis vectors that form the surface-local coordinate system. The values of the other two basis vectors are *arbitrary*, and there are multiple ways of determining them.

Figure 13-6: Example of an Arbitrary Coordinate System



Although one of the basis vectors that make up a surface-local coordinate system is defined as the normal, the other two vectors are arbitrary, because they are nondeterministic: they do not resolve to any specific unique values. In the figure above, blue lines indicate normals, red lines indicate tangent vectors, and green lines indicate binormal vectors. The tangent and binormal vectors can be rotated about the normal in an infinite number of directions.

However, let's assume we create a perturbation normal that satisfies the following requirements:

> "Let the U-axis of the texture space be the direction of the tangent vector in surface-local space, and let the V-axis of the texture space be the direction of the binormal vector in surface-local space. The *x*, *y*, and *z* components of the perturbation normal, expressed in the surface-local coordinate system, are all stored in the texture."

Under this arrangement, the directions that maximize ∂u and ∂v (the partial differentials of the texture coordinates at each vertex) are the U-axis and V-axis of the texture space. Furthermore, ∂u and ∂v are themselves the respective tangent and binormal vectors, in relation to the perturbed normal stored in the texture. In other words, ∂u and ∂v at each vertex are the remaining two basis vectors that form the surface-local coordinate system at that vector, and as a result, with this approach the surface-local spaces at each vertex are *deterministic* (that is to say, a single normal will define only a single surface-local space).

Based on these assumptions, the tangent vector is then calculated for each vertex as follows.

Equation 13-2: Per-Vertex Tangent Vector Calculation

$$\vec{T} = \frac{(v_3 - v_1)(\vec{p}_2 - \vec{p}_1) - (v_2 - v_1)(\vec{p}_3 - \vec{p}_1)}{(u_2 - u_1)(v_3 - v_1) - (v_2 - v_1)(u_3 - u_1)}$$

In this equation, $\vec{T}$ represents the tangent vector at the vertex $\vec{p}_1$ within the triangle formed by the vectors $\vec{p}_1, \vec{p}_2, \vec{p}_3$., which themselves represent the vertex coordinates. The variables u and v represent the texture coordinates at each vertex. For details about the derivation of this equation, click here[7].

13.4 Bump Mapping with DMPGL 2.0 Fragment Lighting

As we explained in the previous section, shading calculations for per-fragment bump mapping require two conditions: (1) the perturbed normals must be stored in a texture in the surface-local coordinate system, and (2) the surface normal and the tangent vector—two of the basis vectors that form the surface-local coordinate system—must be defined as vertex attributes for each vertex.

13.4.1 Bump Mapping Operations

When bump mapping is enabled in DMP fragment lighting, dot products are calculated as follows.

1. The normals and tangent vectors that are defined as vertex attributes are transformed to eye coordinates during vertex operations (that is, by a vertex shader).
2. The per-fragment normals, tangent vectors, and binormals are generated based on the per-vertex normals and tangent vectors defined as described above. All three of these vectors are defined in the eye coordinate system.
3. These three vectors are used to generate a transformation matrix that converts to the eye coordinate system from the surface-local coordinate system. This matrix is used to transform the perturbed normals referenced from the texture from the surface-local coordinate system to the eye coordinate system.

[7] http://jerome.jouvie.free.fr/OpenGl/Lessons/Lesson8.php

4. Once the perturbed normals have been transformed to the eye coordinate system, they replace the unperturbed normals, and the dot products `LN`, `VN`, and `NH` are calculated.

To enable bump mapping, set the reserved uniform `dmp_LightEnv.bumpMode` to `GL_LIGHT_ENV_BUMP_AS_BUMP_DMP`. You must also use the reserved uniform `dmp_LightEnv.bumpSelector` to select the texture unit where the perturbed normals are stored.

13.4.2 Quaternion Transformation

Note that the rasterization process does not generate per-fragment normals, tangent vectors, and binormals directly from the normals, tangent vectors, and binormals that are specified as vertex attributes (the binormals are actually calculated from the normals and tangent vectors). This means that the vertex shaders don't output normals or tangent vectors directly.

The 3x3 transformation matrix shown below is made up of a normal, a tangent vector, and a binormal. It is used to convert from the surface-local coordinate system to the eye coordinate system.

Equation 13-3: Surface-Local to Eye Coordinate Transformation Matrix

$$\begin{bmatrix} Ex \\ Ey \\ Ez \end{bmatrix} = \begin{bmatrix} Tx & Bx & Nx \\ Ty & By & Ny \\ Tz & Bz & Nz \end{bmatrix} \begin{bmatrix} Sx \\ Sy \\ Sz \end{bmatrix}$$

Here, N indicates the normal, T indicates the tangent vector, B indicates the binormal, S indicates the coordinate values in the surface-local coordinate system, and E indicates the coordinate values in the eye coordinate system. This rotation matrix can be converted into a (per-vertex) quaternion, which is used to generate per-fragment quaternions in the rasterization process. These quaternions are converted into the original rotation matrix during fragment lighting.

As a result, when handling perturbed normals using DMP fragment lighting (to be more precise, if there are any vectors that must be converted from the surface-local coordinate system to the eye coordinate system), the vertex shaders must use the normal and tangent vector at each vertex to generate a rotation matrix that will convert from the eye coordinate system to the surface-local coordinate system, and convert the resulting rotation matrices into quaternions. Furthermore, these quaternions must be output by the vertex shaders.

The quaternion (Qx, Qy, Qz, Qw) that is generated from the rotation matrix in Equation 13-3 above is calculated as follows.

Equation 13-4: Conversion of Rotation Matrix to Quaternions

$$Qw = \frac{1}{2}\sqrt{1 + Tx + By + Nz}$$

$$Qx = \frac{1}{4Qw}(Bz - Ny)$$

$$Qy = \frac{1}{4Qw}(Nx - Tz)$$

$$Qz = \frac{1}{4Qw}(Ty - Bx)$$

Some ingenuity is required to create a mathematically stable routine for calculating these components. For details, see Akeine-Moller and Haines[8]. For more details about the actual vertex shader code used to generate the quaternions, refer to the sample code in the file `Common.asm` in the CTR-SDK.

13.5 Format and Type of Normal Maps

DMPGL 2.0 allows all supported 2D textures to be used for storing perturbed normals. Normally, a texture format that has RGB components is used for this purpose, in which case the R, G, and B components are used to represent the *x*, *y*, and *z* components, respectively, of the perturbed normals in the surface-local coordinate system. The values of the *x*, *y*, and *z* components range from -1.0 to 1.0. The value -1.0 is interpreted as the minimum intensity of the component, and the value 1.0 is interpreted as the maximum intensity of the component.

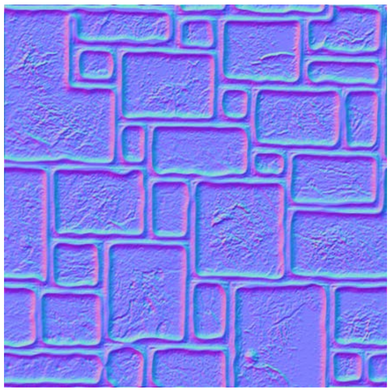
Equation 13-5 below expresses the mapping between the intensities and the vector components in a format that uses 8 bits to store each of the R, G, and G components.

Equation 13-5: Relationship Between Intensity and the Vector Components for 8-Bit RGB

$$value\_of\_x, y, z = \left(\frac{value\_of\_r, g, b}{2^8 - 1}\right) \times 2.0 - 1.0$$

As an example, if the format is `GL_RGB` and the type is `GL_UNSIGNED_BYTE`, a value of -1.0 indicates an intensity of zero for a given component, whereas a value of 1.0 indicates an intensity of 255.

Figure 13-7: Sample Normal Map



For flat (unperturbed) areas, the normals will have a value of (0.0, 0.0, 1.0), which in the normal map will be indicated by the RGB values (128, 128, 255). These areas will appear bluish.

To enable the generation of the *z*-component of the perturbed normals, it is also possible to use the `GL_HILO8_DMP` format, which lacks a B component.

If the generation of the z-component of the perturbed normals is disabled, then perturbed normals fetched from the texture will be used as is, without normalization. You should thus store the normalized values in the texture.

[8] Akenine-Moller, Tomas; Haines, Eric. Real-Time Rendering, 2nd Edition (Japanese translation). Tokyo, Born Digital, Inc., 2006, p. 38 - p. 40

If the texture filter mode of perturbed normals is not point sampling (GL_NEAREST), then due to sample-value filtering, perturbed normals may be fetched with non-normalized values. If this happens, enable the generation of the *z*-component of the perturbed normals.

13.6 Tangent Mapping

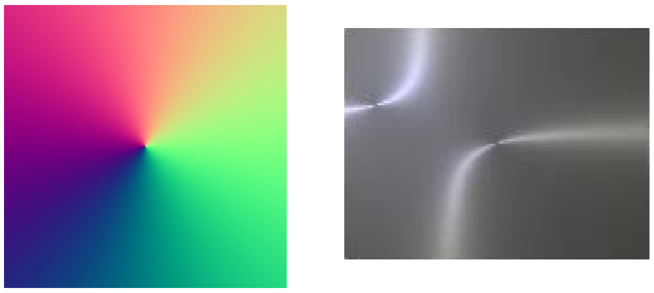
In DMP fragment shading, the lookup values from textures have other applications beyond their use as perturbed normals. If you specify `GL_LIGHT_ENV_CP_DMP` to one of the reserved uniforms `dmp_LightEnv.lutInput{D0,D1,SP}`, a value with the tangent applied will be used as the input to the lookup table. This type of input value is normally required for lighting that expresses anisotropic reflection. If you use tangent mapping, the tangent defined at each vertex can be replaced by the perturbed normal, which provides variation in the shading for per-fragment anisotropic reflection.

To enable bump mapping, set the reserved uniform `dmp_LightEnv.bumpMode` to `GL_LIGHT_ENV_BUMP_AS_TANG_DMP`. You must also use the reserved uniform `dmp_LightEnv.bumpSelector` to select the texture unit where the perturbed tangents are stored.

To reiterate, tangent mapping replaces the individual tangents defined at each vertex with perturbed tangents.

When using tangent mapping, we recommend that you don't use features that generate the *z*-component. DMPGL 2.0 tangent maps assume that the perturbed tangents don't have a *z*-component. If you use a feature to generate the *z*-component, a perturbed normal may be generated that has a non-zero z-component in the tangential coordinate system, giving rise to an unintended perturbed normal.

Figure 13-8: Perturbed Tangents Used for Tangent Mapping (Left) and the Results of Applying Them (Right)



In Figure 13-8 the original tangents assigned to the object on the right all faced in the same direction, but by using the map shown to the left, the tangents are perturbed on a per-pixel basis.

13.7 Related Parameters

The DMP fragment lighting settings related to bump mapping are shown below.

© 2009–2012 Nintendo
CONFIDENTIAL

Table 13-1: Reserved Uniforms Related to Bump Mapping

Reserved Uniforms	Default Value	Description
`dmp_LightEnv.bumpSelector`	`GL_TEXTURE0`	Specifies the texture unit to reference for the perturbed normals.
`dmp_LightEnv.bumpMode`	`GL_LIGHT_ENV_BUMP_NOT_USED_DMP`	Enables/disables bump mapping. When enabled, this uniform is used to specify whether the perturbation vectors referenced from the texture unit should be used as normals or tangents.
`dmp_LightEnv.bumpRenorm`	`GL_FALSE`	Specifies whether to recalculate the z-component of the perturbation vectors referenced from the texture unit.

If the reserved uniform `dmp_LightEnv.bumpRenorm` is set to `GL_TRUE`, the *z*-component of the perturbed normals referenced from the texture is not used; instead, the *z*-component is recalculated based on the *x*- and *y*-components. In most cases, recalculating the *z*-component will yield a better result than if the *z*-component stored in the texture were used as the perturbation vector without modifying it at all. This feature must be enabled if using textures in the `GL_HILO8_DMP` format.

14 DMP Shadows

DMP shadows use a two-pass shadow algorithm. The first pass obtains scene depth information using the light source as the starting point, and the second pass uses that depth information to evaluate whether a given pixel is lit or in shadow. In addition to the depth information from the first pass, the DMP shadow feature also collects shadow intensity information. This extension makes it possible to have soft shadows.

14.1 Shadow Generation Overview

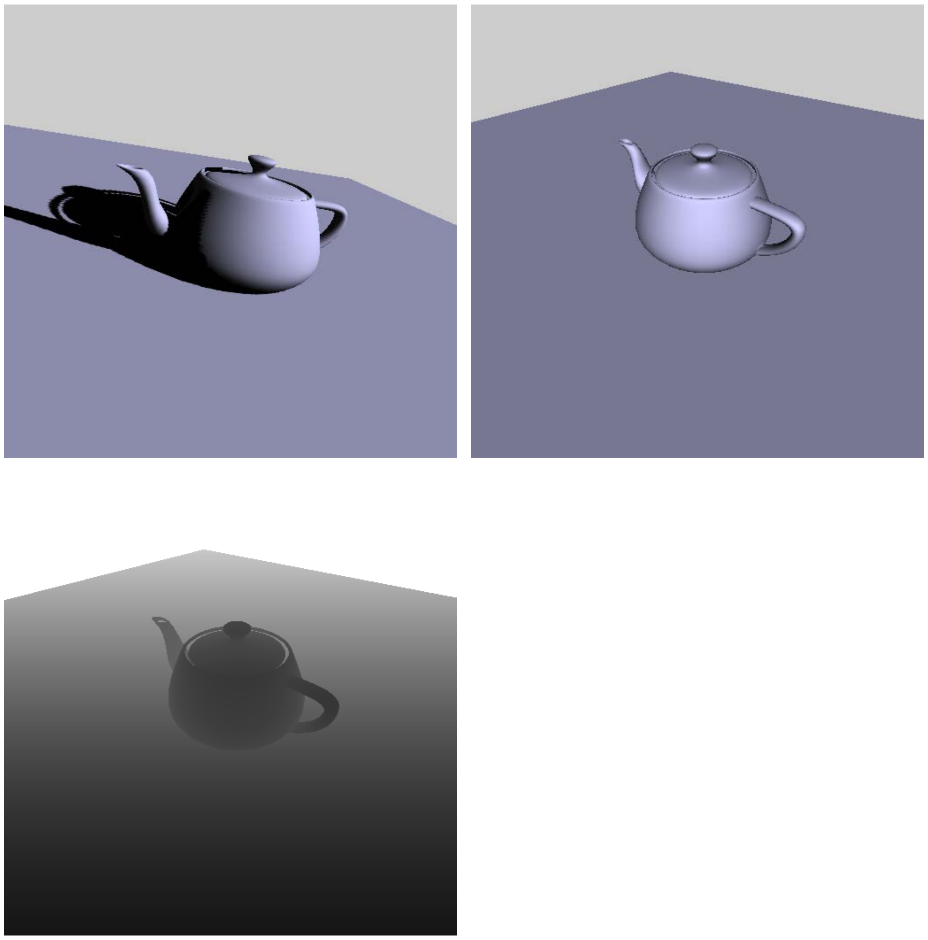
The basic concept for DMP shadow generation is the same as the OpenGL two-pass z-buffer shadow algorithm that uses depth textures.

The scene is rendered as seen from the light source (called the first pass), and depth information is stored in a buffer. This way, the distance from the light source to the first object that the light encounters is calculated and recorded in the depth buffer. This is the distance that the light reaches from the light source. It indicates the range of the lit areas, and all regions beyond this distance are in shadow. Actual rendering (called the second pass) determines what areas are in shadow by calculating the distance from the light source to each pixel and comparing that to the distance that the light reaches, as recorded in the first pass. If the fragment is determined to be in shadow, it is rendered as such.

In OpenGL, the depth information that is stored in the z-buffer in the first pass is applied as a texture in the second pass. Then, the texture unit's comparison feature determines whether each given fragment is affected by an object blocking its light. To support this method, the *depth texture* and *depth texture comparison* features were added to OpenGL starting from version 1.4.

© 2009–2012 Nintendo
CONFIDENTIAL

Figure 14-1: Shadow Generation Example



The top left image in Figure 14-1 is an example of shadow generation using the OpenGL depth texture and depth comparison features. The top right image is a scene as seen from the light source. The bottom image shows its depth information.

For details on the thought process related to the above shadow technique, and implementation using OpenGL, see the available documentation of shadows that use depth textures.

In the following we explain the differences between DMP shadows and shadows in OpenGL, including the features unique to DMP shadows, and give implementation examples.

14.2 Shadow Textures

DMP shadows use shadow textures. Shadow textures use the `GL_SHADOW_DMP` format and contain shadow intensities as well as depth values.

To create a shadow texture, create a framebuffer object, and specify the shadow texture as the render target. Note that shadow textures are attached to color buffer attachment points; they are not attached to the depth buffer. Next, set the DMPGL reserved fragment shader's reserved uniform `dmp_FragOperation.mode` to `GL_FRAGOP_MODE_SHADOW_DMP` to switch the pipeline mode and features to a mode that stores shadow information (depth values and shadow intensities) in shadow textures.

The following code sample creates a 512x512 shadow texture.

Code 14-1: Example of Shadow Texture Generation

```
glGenTextures(1, &sdwtex_name);
glGenFrameBufferOES(1, &fbo_name);
glBindTexture(GL_TEXTURE_2D, sdwtex_name);
glTexImage2D(GL_TEXTURE_2D, 0, GL_SHADOW_DMP, 512, 512,0,
GL_SHADOW_DMP, GL_UNSIGNED_INT, 0);
glBindFrameBuffer(GL_FRAMEBUFFER, fbo_name);
glFrameBufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0_EXT,GL_TEXTURE_2D, sdwtex_name, 0);
glUniform1i(LOC(“dmp_FragOperation.mode”), GL_FRAGOP_MODE_SHADOW_DMP);
```

14.2.1 Shadow Texture Support

Note that not all texture units support shadow textures. In DMPGL 2.0, only texture unit 0 can handle shadow textures.

14.2.2 The `dmp_FragOperation.mode` Reserved Uniform

DMPGL’s reserved fragment shader extends per-fragment operations (also called PFO) to include not only fog processing but subsequent steps as well. When the reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`, the per-fragment operations provide such OpenGL-specification operations as alpha tests, stencil tests, depth tests, and blending.

Setting this reserved uniform to `GL_FRAGOP_MODE_SHADOW_DMP` results in an operation that writes the render objects' depth values and shadow intensities to the color buffer (to which there is a shadow texture attached). In this mode, the per-fragment operations do not include the standard processes such as alpha tests or stencil tests. For fragments, the shadow depth information stored in the shadow texture is accessed and the depth information is tested using a process similar to the `GL_LESS` function. For soft shadows, the shadow strength information is tested instead. (For details, see section 6.4 DMP Shadows of the *DMPGL 2.0 Specifications*.)

14.3 Shadow Texture Depth

As described above, a shadow texture contains depth values and shadow intensity information. However, note that the depth values of shadow textures are linearly related to eye-space Z-values (z_e) in viewpoint space. This stands in sharp contrast to OpenGL depth textures, which in most cases (such as when perspective projection has been used for the projection transformation) store depth values that are not linearly related to the Z-values in eye space. The notations such as z_w and z_e below comply with section 2.15 Coordinate Systems in the *DMPGL 2.0 Specifications*. Note that z_c, the Z-values in clip coordinates, diverge from the OpenGL Specifications.

Equation 14-1 below shows an example of the depth values (Z-values in window coordinates) stored in a shadow texture.

Equation 14-1: Depth Values in a Shadow Texture

$$z_w = -\frac{1}{f-n} z_e - \frac{n}{f-n}$$

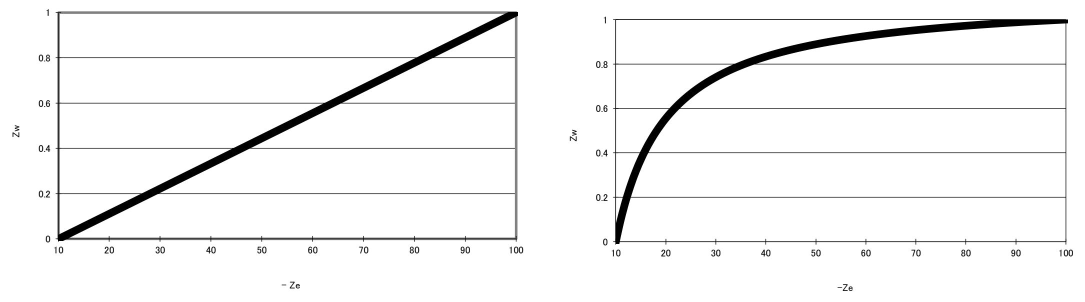
Here, z_e is the depth in the eye coordinate system, n and f indicate the values of the near and far clipping planes, and z_w indicates the depth in window coordinates. The depth value z_e is 0.0 at the near clipping plane and 1.0 at the far clipping plane. However, note that z_w has a first-order correlation with z_e. In OpenGL, it is common (such as when perspective projection has been used for the projection transformation) for z_w to be expressed using Equation 14-2 below, which yields a non-linear relationship between z_w and z_e.

Equation 14-2: Depth Values in OpenGL

$$z_w = \frac{fn}{f-n} \frac{1}{z_e} + \frac{f}{f-n}$$

Figure 14-2 below shows the relationship between depth values before (z_e) and after (z_w) projection transformation. The post-transformation depth values (z_w) are stored in shadow textures in DMP, and in depth textures in OpenGL.

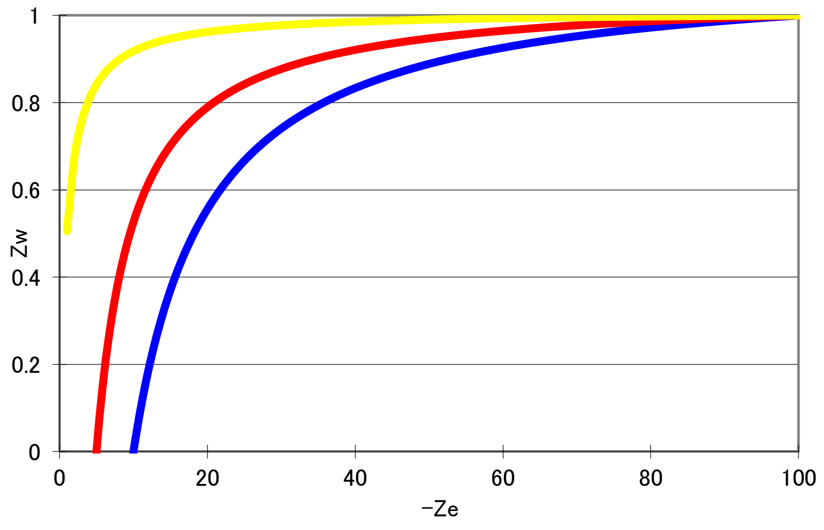
Figure 14-2: Relationship Between Depth Values z_w and z_e in a DMP Shadow Texture (Left) and an OpenGL Depth Texture (Right)



Note that in OpenGL, the z_w depth values stored in depth textures do not have a linear correlation with the z_e. In OpenGL, the effective precision of z_w decreases near the far clipping plane. (For both images in Figure 14-2, $-z_e$ is shown on the X-axis, z_w is shown on the Y-axis, and the values of the near and far clipping planes are set to 10.0 and 100.0, respectively.)

The application of Equation 14-1 guarantees that the precision of the post-transformation depth values (z_w) is always consistent over the entire range of the depth values (z_e). In contrast, Figure 14-3 below shows how the OpenGL Z-depth in Equation 14-2 varies with n. The smaller the value of the near clipping plane, the lesser the effective precision of the depth values approaching the far clipping plane.

Figure 14-3: Relationship Between z_w and z_e for Different Near Values in OpenGL



In Figure 14-3 the blue line has a near value of 10.0, the red line has a near value of 5.0, and the yellow line has a near value of 1.0. The horizontal axis indicates $-z_e$, and the vertical axis indicates z_w. The far value was set to 100.0 for all three lines.

14.3.1 The `dmp_FragOperation.wScale` Reserved Uniform

The shader's reserved uniform `dmp_FragOperation.wScale` is set to 0.0. The variables z_w, z_c, and w_c have the relationship shown in Equation 14-3. (The variables z_c and w_c are in clip coordinates.) (The z_c definition diverges from the OpenGL ES 1.1 Specification. See section 2.15 Coordinate Systems in the *DMPGL 2.0 Specifications*.)

Equation 14-3: Relationship Between z_w, z_c, and w_c with `dmp_FragOperation.wScale` Set to 0.0

$$z_w w_c = -z_c$$

As a result, the following is true:

Equation 14-4: Solving Equation 14-3 for z_w

$$z_w = -\frac{z_c}{w_c}$$

For perspective projection, this is no different Equation 14-2. In other words, this mode calculates the depth based on the OpenGL specifications and is also the default value. If the reserved uniform is set with an argument other than 0.0 (let us call this nonzero argument a), then z_w, and z_c, have the relationship shown in Equation 14-5.

Equation 14-5: Relationship Between z_w and z_c with `dmp_FragOperation.wScale` Set to a Nonzero Value

$$z_w = -az_c$$

For perspective projection, if we take $a = 1.0/f$, Equation 14-5 is equivalent to Equation 14-1. In other words, this mode obtains depth values that have a linear correlation to z_e. Under most circumstances, set a to $1.0/f$ if perspective projection was used for the projection transformation, and set a to 1.0 if orthographic projection was used. These settings yield a linear correlation with z_e and also calculate z_w such that $z_w = 0.0$ at the near clipping plane and $z_w = 1.0$ at the far clipping plane. Rendering is done in this mode during the first pass of shadow texture creation.

14.3.2 Shadow Texture Comparison

DMP shadow textures only have two possible interpretations:

- $R > D$ Fragment is in shadow
- $R \leq D$ Fragment is not in shadow

Here, R indicates the R component of the texture coordinate when accessing the shadow texture, and D indicates the depth value of the shadow texture. Note that there is only one comparison mode with DMPGL, unlike the multiple comparison modes that OpenGL has.

14.3.3 Texture Coordinates and Texture Transformation Matrices During Shadow Texture Access

During the second pass, the depth information that was created during the first pass is accessed as a texture. This is true of both OpenGL shadows and DMP shadows. The texture unit compares the content of the shadow texture with the value at the texture coordinates. If the wrong texture coordinates are specified here, the comparison is not performed correctly.

Note that with DMP shadows, the texture coordinates ($^s/_r$, $^t/_r$, $r - bias$) must be used to access shadow textures. This is in contrast to OpenGL, which uses the texture coordinates ($^s/_q$, $^t/_q$, $^r/_q$) to access depth textures. Here, (s, t, r, q) indicate the texture coordinates after applying the texture transformation matrix, and $bias$ indicates the floating-point value that is provided to the reserved uniform shown below.

```
glUniformi(LOC(“dmp_Texture[0].shadowZBias“), bias);
```

If the projection matrix is set to perform perspective projection in the first pass and this projection transformation is equivalent to calling `glFrustum(-r, r, -t, t, n, f)` in OpenGL ES 1.1, the window coordinates are as follows when shadow information is rendered to the shadow texture.

Equation 14-6: X-Window Coordinate for Shadow Texture

$$x_w = \left(\frac{-n}{2r}\frac{x_e}{z_e} + \frac{1}{2}\right) width$$

Equation 14-7: Y-Window Coordinate for Shadow Texture

$$y_w = \left(\frac{-n}{2t}\frac{y_e}{z_e} + \frac{1}{2}\right) height$$

Equation 14-8: Z-Window Coordinate for Shadow Texture

$$z_w = -\frac{1}{f-n}z_e - \frac{n}{f-n}$$

Here, $width$ and $height$ specify the size of the shadow texture. If we consider these window coordinates to be texture coordinates, that texture coordinate would be as follows.

Equation 14-9: Texture Coordinates of Shadow Texture

$$\left(\frac{-n}{2r}\frac{x_e}{z_e}+\frac{1}{2}, \frac{-n}{2r}\frac{y_e}{z_e}+\frac{1}{2}, -\frac{1}{f-n}z_e-\frac{n}{f-n}\right)$$

As a result, the texture transformation matrix and $bias$ must be set during the second pass so that ($^s/_r$, $^t/_r$, $r - bias$) matches the texture coordinates in Equation 14-9. The following texture transformation matrix and bias accomplish that.

Equation 14-10: Texture Transformation Matrix for Second Pass of Shadow Generation

$$texture\_matrix = \begin{bmatrix} \frac{-1}{f-n}\frac{-n}{2r} & 0 & \frac{-1}{f-n}\frac{1}{2} & 0 \\ 0 & \frac{-1}{f-n}\frac{-n}{2t} & \frac{-1}{f-n}\frac{1}{2} & 0 \\ 0 & 0 & \frac{-1}{f-n} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Equation 14-11: Bias for Second Pass of Shadow Generation

$$bias = \frac{n}{f-n}$$

To achieve the correct comparison, you must implement both of these. Implement the $bias$ value by setting it in the reserved uniform `dmp_Texture[0].shadowZBias`, as mentioned earlier, and use a vertex shader to implement the texture transformation matrix in Equation 14-10. However, when the texture coordinate r is outside the range of [0.0, 1.0], when calculating the value for comparison with the depth information for the shadow buffer (r – bias), r is clamped in the [0.0, 1.0] range and subtraction is performed thereafter according to the bias, after which it is again clamped in the [0.0, 1.0] range. To perform a correct comparison, specify 0 in bias for objects placed at the back of the far plane in the light source coordinate system for the first pass.

The code below shows how to implement Equation 14-10's texture transformation matrix using the OpenGL ES 1.1 API.

© 2009–2012 Nintendo
CONFIDENTIAL

```
  glMatrixMode(GL_TEXTURE);
  glLoadIdenity();
  glMultMatrixf(xxx);  (Here, “xxx” is the texture transformation matrix
shown above)
  Alternatively:
  glMatrixMode(GL_TEXTURE);
  glLoadIdentity();
  glFrustumf(r/n, -3r / n, t/n, -3t / n, 1.f, 0.f);
  glScalef(-1.f/(f-n), -1.f/(f-n), -1.f/(f-n));
```

In the next sample implementation, an orthographic projection is used in the first pass. If this projection transformation is equivalent to calling `glOrtho(-r, r, -t, t, n, f)` in OpenGL ES 1.1, a texture transformation matrix for shadow texture lookup is implemented using the OpenGL ES 1.1 API as follows.

```
  glMatrixMode(GL_TEXTURE);
  glLoadIdentity();
  glOrthof(-3r, r, -3t, t, 2n-f, f);
```

The equations below show the texture transformation matrix and bias for the above.

Equation 14-12: Texture Transformation Matrix for Orthographic Projection

$$texture\_matrix = \begin{bmatrix} \frac{1}{2r} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2t} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{-1}{f-n} & \frac{-n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 14-13: Bias for Orthographic Projection

$$bias = 0$$

14.4 Rendering Shadow Objects

Recall that shadow textures are attached to color buffers. Clear values for shadow textures are set using **`glClearColor`**, and shadow textures are cleared by specifying `GL_COLOR_BUFFER_BIT` as an argument to the **`glClear`** function.

Use (1.0, 1.0, 1.0, 1.0) as the clear value for shadow textures. Objects that generate soft shadows must be rendered after first rendering objects that generate hard shadows. Note that, if hard shadow regions and soft shadow regions are rendered alternately, the result is not guaranteed. Hard shadows are rendered as shadow objects with the color g-component set to 0.0. The typical technique involves implementing a vertex shader that outputs the vertex color g-component as 0.0, then disables textures.

The use of any value other than 0.0 for the color g-component is considered as rendering soft shadows.

Rendering of soft shadows does not use the color r-, b-, or a-components. Only the g-component is used as the shadow intensity. However, the clear value for the r-, g-, b-, and a-components must be set to 1.0. This is required because both the depth information and the intensity information are cleared with 1.0.

The example below illustrates clearing the shadow texture and the render state when rendering shadow objects.

Code 14-2: Example for Rendering Shadow Objects

```
//clear
glClearColor(1.f, 1.f, 1.f, 1.f);
glClear(GL_COLOR_BUFFER_BIT);
…
//render
glUniform1i(LOC("dmp_Texture[0].samplerType"), GL_FALSE);
glDrawArrays() or glDrawElements()
```

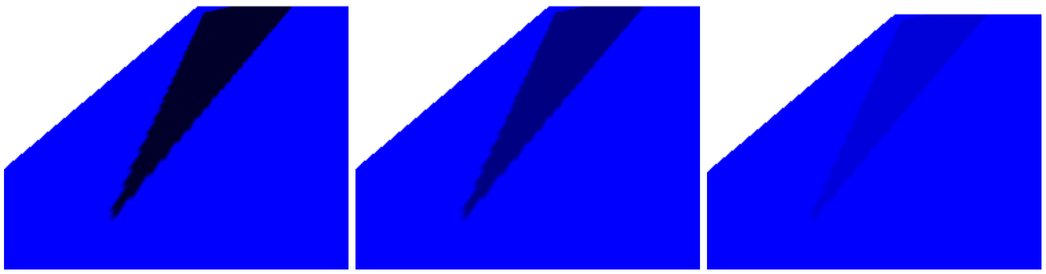
Code 14-2 uses a program object to which a vertex shader that outputs a vertex color with a g-component of 0.0 attached. Textures are disabled.

14.4.1 Shadow Intensity

As stated earlier, shadow textures include both depth values and shadow intensities. Soft shadows are possible by setting the shadow intensity to an appropriate value. The texture unit to which the shadow texture has been applied outputs black (0.0, 0.0, 0.0, 0.0) if it determines that the area in question is within a shadowed region. Otherwise, it outputs the shadow intensity (a value between 0.0 and 1.0) that is stored in the shadow texture.

The figure below shows how differences in the shadow intensities of a shadow texture will cause variation in the shadows generated during the second pass.

Figure 14-4: Differences in Shadow Intensity

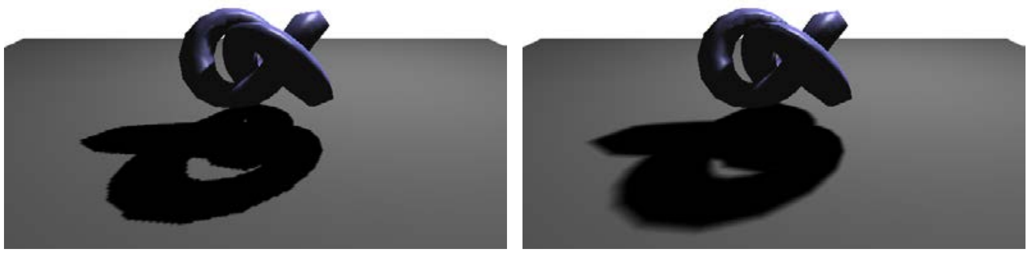


14.4.2 Soft Shadows Using Silhouette Primitives

As the shadow intensity example in Figure 14-4 shows, rendering the shadow's silhouette lines with the appropriate shadow intensity makes it possible to express the silhouette lines as a penumbra (partial shadow). Silhouette lines are rendered using silhouette primitives. The object itself must be rendered in black. For the silhouette portion, the g-component of the vertex color for two of the vertices of the silhouette quad (the two that don't form the edge) must be 1.0. The vertex colors of the silhouette are interpolated within the silhouette quad, and the shadow intensity is also interpolated within the silhouette quad to match the vertex color.

The example below shows soft shadows that use silhouette primitives.

Figure 14-5: Soft Shadow Sample Using Silhouette Primitives



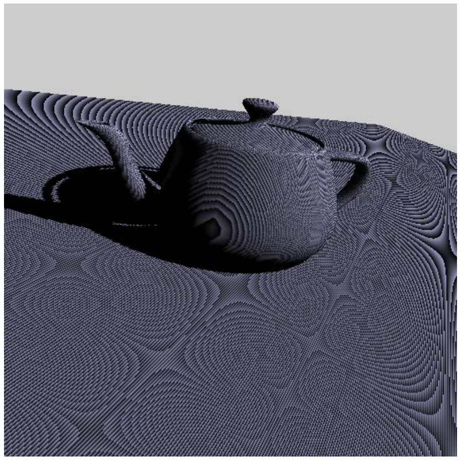
(The image on the left in Figure 14-5 has silhouettes disabled; the image on the right has silhouettes enabled.)

14.5 Shadow-Related Artifacts

14.5.1 Self-Shadow Aliasing

One of the problems that can arise during shadow generation that uses this type of two-pass approach is known as *self-shadow aliasing*. This occurs when fragments mistakenly cast shadows on themselves. When the depth values created in the first pass are compared to the depths as seen from the light source in the second pass, if the depth values from the first pass are just marginally smaller, an incorrect determination is sometimes made that ($R > D$) and that the fragment is in shadow. This incorrect determination is the cause of self-shadow aliasing. Figure 14-6 below illustrates the problem.

Figure 14-6: Example of Self-Shadow Aliasing



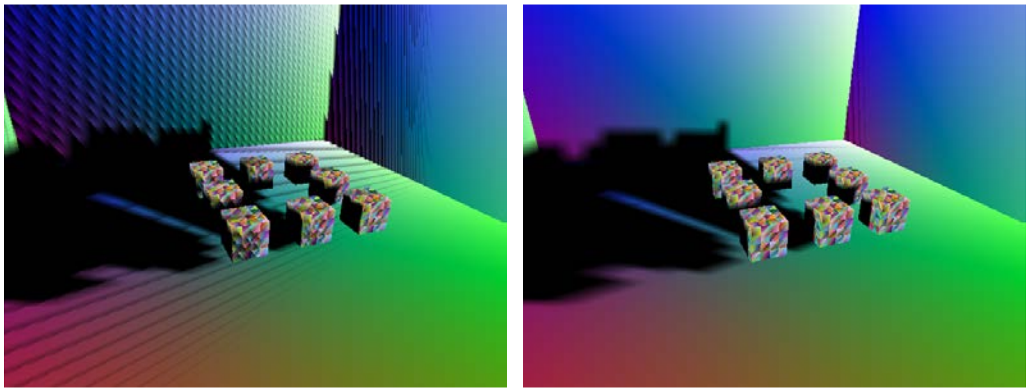
In Figure 14-6 the fragments are mistakenly casting shadows on themselves, which causes a Moiré effect on the whole object.

One method for avoiding this problem is to suppress aliasing by applying a negative bias to the depth values in the second pass. The code below is an example of configuring this $bias$ value to suppress aliasing.

```
glUniform1f(LOC("dmp_Texture[0].shadowZBias"), 1.2f*n/(f-n));
```

With these settings for $bias$, the third component of the texture coordinate becomes $-\frac{1}{f-n}z_e - \frac{1.2n}{f-n}$ when accessing the shadow texture.

Figure 14-7: Self-Shadow Aliasing on the Floor and Walls (Left) and Suppressed by a Bias (Right)



14.5.2 Silhouette Shadow Artifacts

Artifacts generated by silhouette shadows cannot be mitigated by adding a bias as described in section 14.5.1. The DMP shadow feature outputs the shadow intensity if the shadow-determination process finds that a given fragment is not in a shadow region. Normally, non-shadowed areas have a shadow intensity of 1.0, so they are not affected by shadow attenuation. (That is, the brightness of these fragments does not change when it is multiplied by the shadow intensity.) Soft shadow regions generated by silhouettes, on the other hand, are affected by shadow attenuation, even though they are not deemed to be part of a shadow region in the shadow-determination process. (Since their shadow intensity is not 1.0, their brightness decreases when multiplied by the shadow intensity.) This phenomenon causes artifacts to occur with some objects.

That said, it is possible to suppress this type of artifact by configuring the shadow-related settings in DMP fragment lighting, along with the texture combiner settings. This type of artifact normally becomes a problem when shadowing is applied to a surface that is parallel to the light source, so one suppression method is to define the shadow texture output value (the attenuation term) as shown in Equation 14-14.

Equation 14-14: Modification of Shadow Attenuation Term to Prevent Silhouette Shadow Artifacts

$$1.0 - f(1.0 - Sdw)$$

Sdw is the original shadow attenuation term, and f is an arbitrary function whereby $f \approx 0.0$ when the surface's normal is perpendicular to the light source and $f \approx 1.0$ when the surface's normal is parallel to the light source. With the shadow attenuation term defined in this way, the shadow attenuation is approximately 1.0 when the surface's normal is perpendicular to the light source. In other words, there will be zero attenuation due to shadows (the shadows will have no effect). Conversely, if the surface's normal is nearly parallel with the light source, the attenuation is approximately Sdw.

An example that implements the shadow attenuation term as $1.0 - f(1.0 - Sdw)$ is shown below in Code 14-3. Note that the primary and secondary colors are not directly multiplied by the shadow (Sdw); instead the alpha components are multiplied by it. The actual multiplication is done in the texture combiner. Also note that the output from the `FR` table is output as the alpha component. As a result, the layer configuration must be set to a mode in which `FR` can be used.

Code 14-3: Implementation of the Shadow Attenuation Term $\mathbf{1.0 - f(1.0 - Sdw)}$ Using DMP Fragment Lighting

```
  glUniform1i(LOC(“dmp_FragmentLighting.enabled”), GL_TRUE);

  ..other code..
  glUniform1i(LOC(“dmp_LightEnv.lutInputFR”), GL_LIGHT_ENV_LN_DMP);
  glUniform1i(LOC(“dmp_LightEnv.config”), GL_LIGHT_ENV_LAYER_CONFIG1_DMP);
  glUniform1i(LOC(“dmp_LightEnv.fresnelSelector”),
GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP);
  glUniform1i(LOC(“dmp_LightEnv.shadowAlpha”), GL_TRUE);
  glUniform1i(LOC(“dmp_LightEnv.invertShadow”), GL_TRUE);

  GLuint luts[2];
  GLfloat lut[512];
  int j;
  ..other code..
  memset(lut, 0, sizeof(lut));
  for (j = 1; j < 128; j++)
  {
      lut[j] = powf((float)j/127.f, 2.0f);
      lut[j+255] = lut[j] - lut[j-1];
  }
  glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
  GL_LUMINANCEF_DMP, GL_FLOAT, lut);
  glUniform1i(LOC(“dmp_FragmentMaterial.samplerFR”), 0);
```

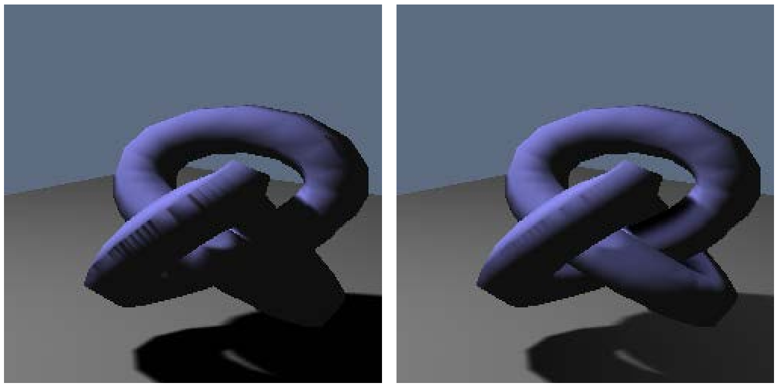
In Code 14-3, $f(1.0 - Sdw)$ is output as the alpha component of the fragment’s primary color and secondary color. The reserved uniform `dmp_LightEnv.shadowAlpha` is set to `GL_TRUE` and only the alpha component is multiplied by Sdw. Also note that the reserved uniform `dmp_LightEnv.invertShadow` is set to `GL_TRUE` so that Sdw is being inverted $(1.0 - Sdw)$. The values in the table set for the reserved uniform `dmp_FragmentMaterial.samplerFR` are used for the alpha component to implement the function $f(1.0 - Sdw)$. The table f is set to x^2, and the lookup is done using the dot product of the light vector and the normal (`LN`). This means that this table will output 0.0 when L and N are perpendicular, and will output 1.0 when the two vectors are parallel.

Code 14-4: Using the Texture Combiners to Multiply the Shadow Attenuation Term $1.0 - f(1.0 - Sdw)$ by the Primary Color

```
glUniform1i(LOC("dmp_TexEnv[0].combineRgb"), GL_MODULATE);
glUniform1i(LOC("dmp_TexEnv[0].combineAlpha"), GL_REPLACE);
glUniform3i(LOC("dmp_TexEnv[0].operandRgb"),
         GL_SRC_COLOR, GL_ONE_MINUS_SRC_ALPHA, GL_SRC_COLOR);
glUniform3i(LOC("dmp_TexEnv[0].operandAlpha"),
         GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(LOC("dmp_TexEnv[0].srcRgb"), GL_FRAGMENT_PRIMARY_COLOR_DMP,
         GL_FRAGMENT_PRIMARY_COLOR_DMP, GL_PRIMARY_COLOR);
glUniform3i(LOC("dmp_TexEnv[0].srcAlpha"),
         GL_PRIMARY_COLOR, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
```

Since the settings in Code 14-3 set the alpha component of the primary color to $f(1.0 - Sdw)$, the function $1.0 - f(1.0 - Sdw)$ can be implemented by setting the second element of the reserved uniform `dmp_TexEnv[0].operandRgb` to `GL_ONE_MINUS_SRC_ALPHA`, as shown above in Code 14-4. The RGB components of the primary color are then multiplied by this function.

Figure 14-8: Example of Silhouette Shadow Artifacts (Left) and Artifact Suppression (Right)



14.6 Other Related Parameters

Parameters related to DMP shadows are listed below.

Table 14-1: Shadow-Related Parameters

Reserved Uniforms	Default Value	Description
`dmp_Texture[0].shadowZBias`	`0.0`	Bias value to subtract from the calculated distances between each fragment and the light source
`dmp_Texture[0].perspectiveShadow`	`GL_TRUE`	Set to `GL_TRUE` to divide the texture coordinates *s* and *t* by the *r* coordinate when accessing the shadow texture
`dmp_FragOperation.penumbraScale`	`0.0`	Scaling factor used when calculating penumbra hardness
`dmp_FragOperation.penumbraBias`	`1.0`	Bias value when calculating penumbra hardness

14.7 How to Check Shadow Texture Content

To check the image rendered to a shadow texture, read texel data by calling **`glReadPixels`** with *`format`* set to `GL_RGBA` and *`type`* set to `GL_UNSIGNED_BYTE` while the shadow texture is attached to the current color buffer.

A single texel is represented by 32 bits; the actual hardware and the PicaOnDesktop environments use different data components. In the actual hardware environment, the shadow intensity takes 8 bits and depth information takes 24 bits. The R component represents the shadow intensity; the G, B, and A components each represent 8 bits of depth information, holding the lower 8 bits, middle 8 bits, and upper 8 bits respectively. On the PicaOnDesktop environment, the shadow intensity takes 8 bits and depth information takes 16 bits. The R component indicates the shadow intensity; the B and A components each represent 8 bits of depth information, holding the lower 8 bits and upper 8 bits respectively. On the PicaOnDesktop environment the G component is undefined.

© 2009–2012 Nintendo
CONFIDENTIAL

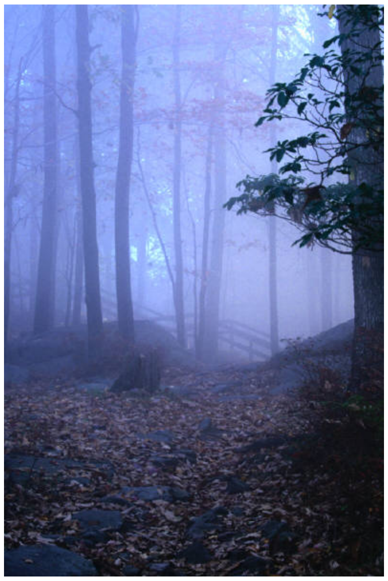
15 Fog

This chapter describes how to use the fog feature.

15.1 Overview

The fog feature is used to represent environmental effects such as mist or steam. Objects melt away into the color of the fog as they move further away from the viewpoint.

Figure 15-1: Image Showing Fog



15.2 Fog in OpenGL ES 1.1

OpenGL ES 1.1 defines the effect of fog using the equation below.

Equation 15-1: OpenGL ES 1.1 Fog Equation

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

Here, C' indicates the post-fog fragment color, and C indicates the pre-fog fragment color. Cf indicates the fog color, which is set using `GL_FOG_COLOR`. f indicates the fog coefficient, which is defined based on the mode, as shown by Table 15-1 below.

Table 15-1: Fog Coefficients in OpenGL ES 1.1

Fog Mode	Formula Used to Calculate the Fog Coefficient
GL_LINEAR	$f = \frac{end - c}{end - start}$
GL_EXP	$f = e^{-(density \times c)}$
GL_EXP2	$f = e^{-(density \times c)^2}$

$start$	:	Set using GL_FOG_START
end	:	Set using GL_FOG_END
$density$	:	Set using GL_DENSITY
c	:	Distance from the origin to the fragment in the eye coordinate system

15.3 Fog in OpenGL ES 2.0

Fog is not defined in OpenGL ES 2.0; it is implemented instead using pixel shaders.

15.4 Fog in DMPGL 2.0

Just as in OpenGL ES 1.1, the calculation of fog in DMPGL 2.0 is defined by Equation 15-1. However, DMPGL 2.0 uses the output of the fog lookup table for the fog coefficient f.

15.4.1 Enabling and Disabling Fog

Set the reserved uniform dmp_Fog.mode to GL_FOG to enable fog.

```
glUniform1i(LOC(dmp_Fog.mode), GL_FOG);
```

To disable fog, set the same uniform to GL_FALSE.

```
glUniform1i(LOC(dmp_Fog.mode), GL_FALSE);
```

15.4.2 Setting the Fog Color

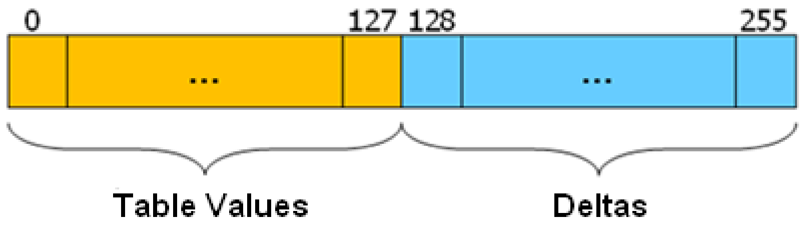
The fog color is set using the reserved uniform dmp_Fog.color.

```
GLfloat fog_color[3] = {0.3f, 0.3f, 0.5f};
glUniform3fv(LOC(dmp_Fog.color), 1, fog_color);
```

15.4.3 Fog Coefficient

The fog coefficient f is the output from the fog lookup table. The fog lookup table is accessed using z_w, the fragment's z-value in window coordinates. The fog lookup table has 256 entries.

Figure 15-2: Fog Lookup Table



The first 128 entries of the fog lookup table store values in the range [0.0, 1.0]. The last 128 entries store values in the range [-1.0, 1.0]. The fog lookup table input value z_w is in the range [0.0, 1.0]. Its minimum value corresponds to the near clipping plane, and its maximum value corresponds to the far clipping plane. The output values of the lookup table are defined using the following equations.

Equation 15-2: Output Values of Fog Lookup Table

$$\begin{aligned} i &= (\text{int})floor(z_w \times 128) && (\text{Integer portion}) \\ fr &= fract(z_w \times 128) && (\text{Fractional portion}) \\ f &= table[i] + table[i+128] \times fr && table[]\text{: Lookup table} \end{aligned}$$

If the function $F(z_w)$ is set for the fog lookup table, the entries in the table will be determined as follows:

Equation 15-3: Calculation of Fog Lookup Table Entries $(\mathbf{0 \le \mathit{i} \le 127})$

$$table[i] = F\left(\frac{i}{128}\right)$$
$$table[i+128] = F\left(\frac{i+1}{128}\right) - F\left(\frac{i}{128}\right)$$

To configure the fog lookup table, prepare an array of `GLfloat` values that will store its values.

```
GLfloat fog_lut[256];
// Set up fog table contents
for (int i=0; i<128; i++)
{
    Fog_lut[i] = F(i/128.0f);
    Fog_lut[i+128] = F((i+1)/128.0f) – F(i/128.0f);
}
```

Next, load the content of the array into the lookup table object.

```
glBindTexture(GL_LUT_TEXTURE1_DMP, id_FogLut);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 256, 0,
                                GL_LUMINANCEF_DMP, GL_FLOAT, fog_lut);
```

To use the lookup table with fog, set the reserved uniform `dmp_Fog.sampler` to the lookup table number to which the lookup table object is bound.

```
glUniform1i(LOC(”dmp_Fog.sampler"), 1);
```

15.5 Implementing OpenGL ES 1.1 Fog with DMPGL 2.0

To implement OpenGL ES 1.1 fog with DMPGL 2.0, use a fog lookup table to implement the fog coefficient used by OpenGL ES 1.1 (refer to Table 15-1). The fog coefficient is a function of c (the distance from the origin to the fragment in eye coordinates), so we use the notation $f(c)$. c approximates $-z_e/w_e$, the distance between the fragment and the XY plane in eye coordinates. If we use the equation $-z_e/w_e = g(z_w)$ to represent the relationship between $-z_e/w_e$ and z_w, we can express the fog coefficient $f(c)$ in terms of z_w, as shown below.

Equation 15-4: Fog Coefficient in Terms of z_wDistance Between Fragment and XY Plane

$$f(c) \fallingdotseq f(-z_e/W_e) = f(g(z_w))$$

Configuring the lookup table using Equation 15-4 allows us to implement the fog functionality of OpenGL ES 1.1. The chart below summarizes the z-coordinates of the near and far clipping planes in each of the coordinate systems.

Table 15-2: Z-Values of the Near and Far Clipping Planes in Each Coordinate System

Coordinate System	Near	Far
Eye Coordinates	-near	-far
Normalized Device Coordinates	0.0	-far/far
Window Coordinates	0.0	1.0

This shows that the range [0.0, 1.0] for z_w maps to [0.0, -1.0] in normalized device coordinates. This leads to the following equation.

Equation 15-5: Z-Value Equivalencies

$$z_d = -z_w$$

Here, z_d indicates the z-value in normalized device coordinates. This lets us find the coordinate values for z_w in eye coordinates using Equation 15-6 below:

Equation 15-6: Transforming z_w to Eye Coordinates

$$(X_e \quad y_e \quad z_e \quad w_e) = (0 \quad 0 \quad -z_w \quad 1.0){M_{proj}}^{-1}$$

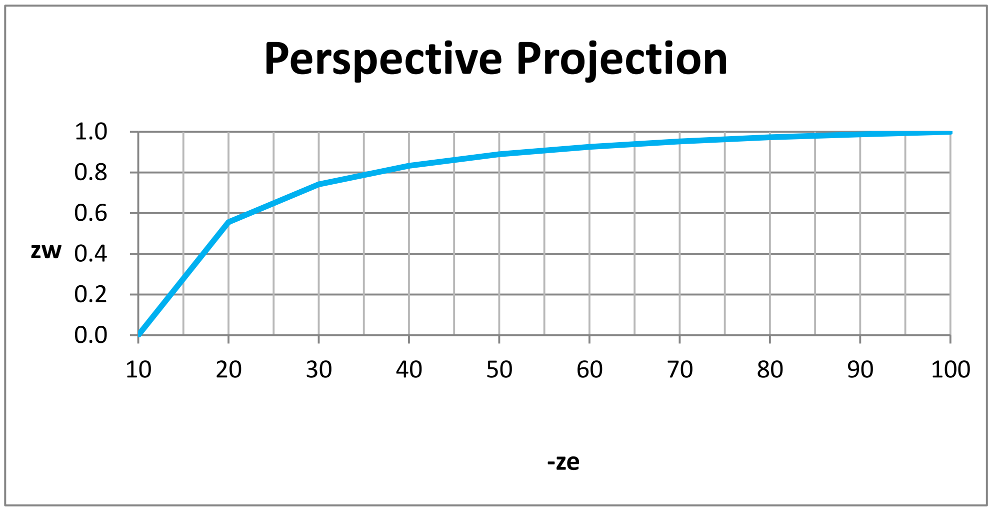
M_{proj}: Projection matrix

An example of how to configure the fog table is shown below. We use the fog table input value z_w to find the distance c from the origin to the fragment in eye coordinates. We then use the fog coefficient function $f(c)$ to configure the contents of the lookup table.

```
GLfloat Fog_LUT[256];                 // Fog lut contents
GLfloat c[128+1];                     // distance in eye coordinate
GLfloat zw;                     // depth value
vec4_t p_clip(0.0, 0.0f, 0.0f, 1.0f);     // position in clip coordinate
vec4_t p_eye;                         // position in eye coordinate
mat4_t invMproj = {…};                // inverted projection matrix

// Get distance in eye coordinate from zw
for (int i=0; i<=128; i++)
{
    zw = ((GLfloat)i/128);            // zw : [0.0, 1.0]
    p_clip.z = -zw;                   // p_clip.z = {0.0, 0.0, -zw, 1.0};
    p_eye = invMproj * p_clip;              // c = -ze/we = g(zw);
    c[i] = -p_eye.z/p_eye.w;          //
}
// Set fog table contents
for (int i=0; i<128; i++)
{
    Fog_LUT[i] = f(c[i]);                   // Fog lut value
    Fog_LUT[i+128] = f(c[i+1]) – f(c[i]);// Fog lut delta value
}
```

Note that in OpenGL ES 1.1, the distance c from the origin in eye coordinates is used to calculate the fog coefficient. In DMPGL, however, the fog lookup table input value z_w is the z-value of the fragment in window coordinates. Figure 15-3 below illustrates the relationship between $-z_e$ and z_w for a perspective projection in which the near clipping plane is at 10.0 and the far clipping plane is at 100.0.

Figure 15-3: Relationship Between Z-Values in the Eye and Window Coordinate Systems



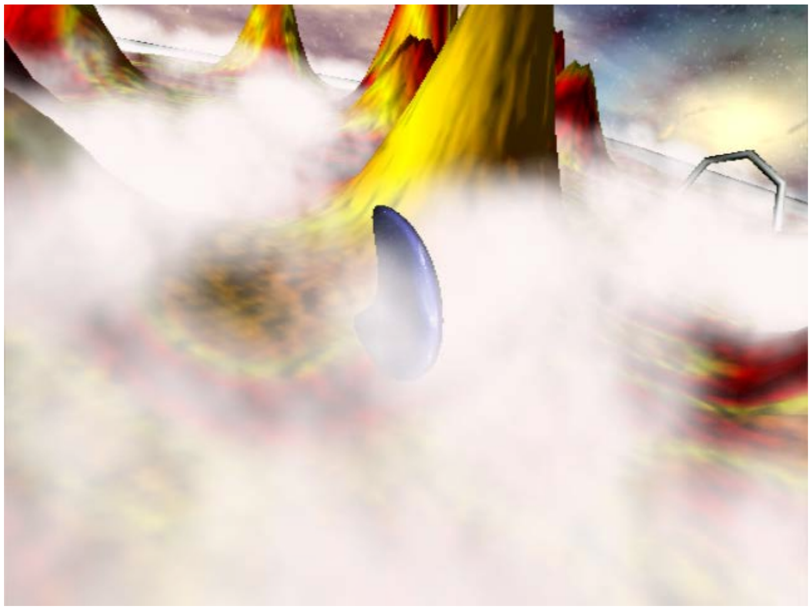
With perspective projections, z_w (the depth value) does not vary linearly with z_e (the z-value in eye coordinates). The distribution of depth values is non-uniform. The closer an object gets to the near clipping plane, the greater its resolution will be; the closer an object gets to the far clipping plane, the lower its resolution will be.

16 Gas

16.1 Overview

DMPGL 2.0 provides a gas feature for rendering gaseous objects. Gaseous objects are rendered using this feature by generating a gas texture that contains the density values that were accumulated during the density-rendering pass. In a subsequent shading pass, the gaseous objects are shaded by referencing this gas texture. Gas textures are generated during the density-rendering pass by determining the areas where the gas intersects with polygonal models, so gaseous objects are rendered by determining the *foreground/background relationship* of the gas versus the polygonal model; that is, determining at each pixel whether the gas or the polygonal model should appear in front.

Figure 16-1: Rendering of a Gaseous Object Using the Gas Feature



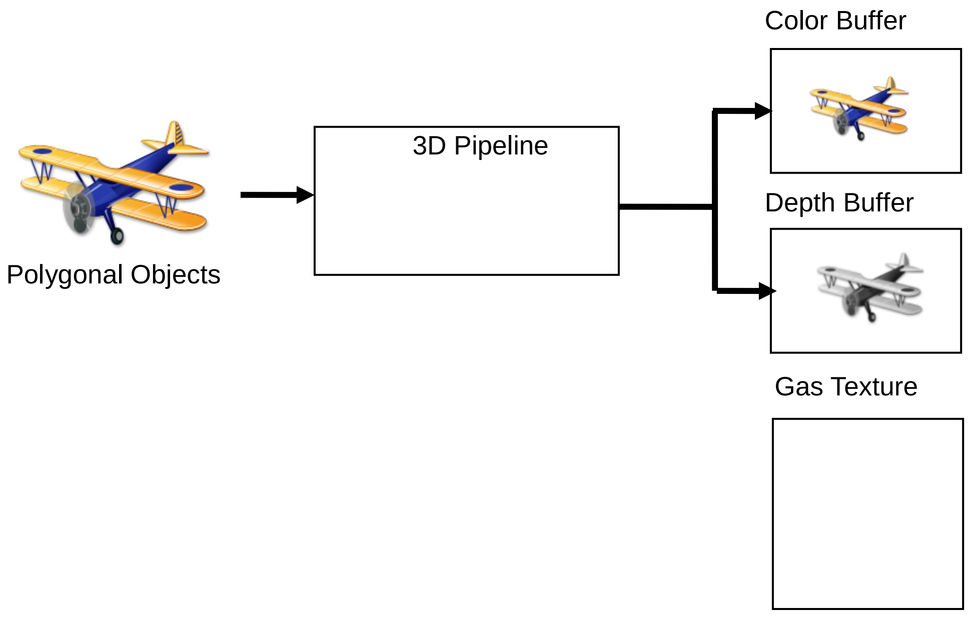
16.1.1 Rendering Procedure for Gaseous Objects

Gaseous objects are rendered using the following procedure.

- Perform a render pass on the polygonal objects (polygonal object rendering pass)
- Perform a render pass on the density values (density-rendering pass)
- Perform a shading pass

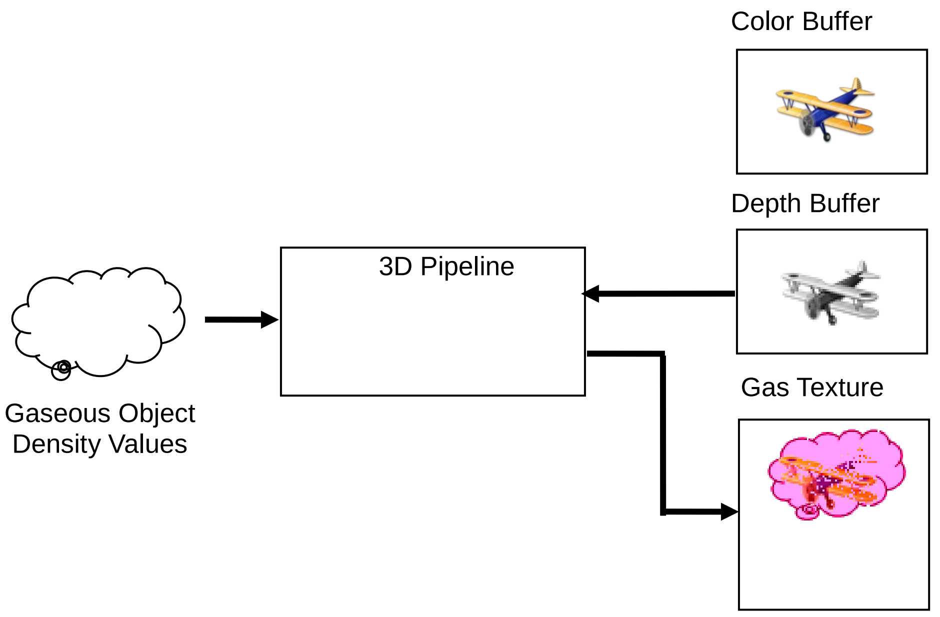
When the gas feature of DMPGL 2.0 renders gaseous objects, it takes into consideration the areas where the gas intersects with polygonal objects in the scene. For this reason, the polygonal objects must be rendered using standard methods and the depth values for the polygonal objects must be stored in the depth buffer before the gaseous object can be rendered (in other words, before the density-rendering pass or the shading pass can be run).

Figure 16-2: Polygonal Object Rendering Pass



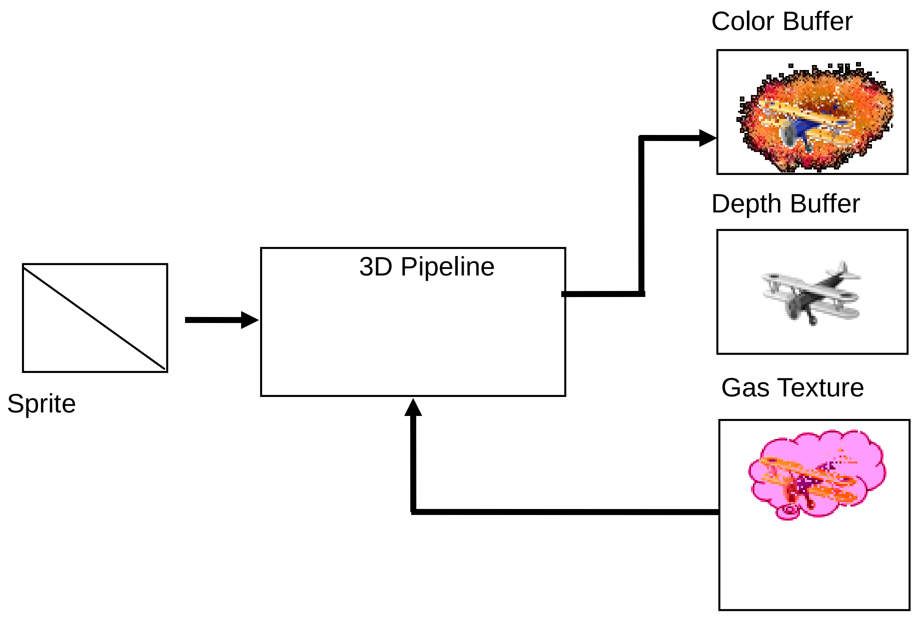
During the *density-rendering pass*, the depth values of the polygonal objects that were rendered in the *polygonal object rendering pass* are referenced in order to render the density values of the gaseous object to the gas texture.

Figure 16-3: Density-Rendering Pass



During the shading pass, the density values stored in the gas texture are referenced to shade the gaseous object. The shading results are rendered to the color buffer and then blended with the polygonal objects that were already rendered during the polygonal rendering pass.

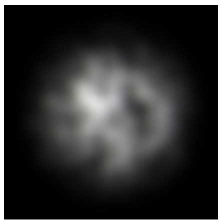
Figure 16-4: Shading Pass



16.2 Gas Particles

Gas particles comprise the smallest constituent parts of gaseous objects. Although gas particles can be of any geometric form, they are typically defined as either sprites (billboarded quads) or point sprites. The texture defined by the density pattern is applied to the gas particles. This texture is called the *density pattern texture*.

Figure 16-5: Sample Density Pattern Texture



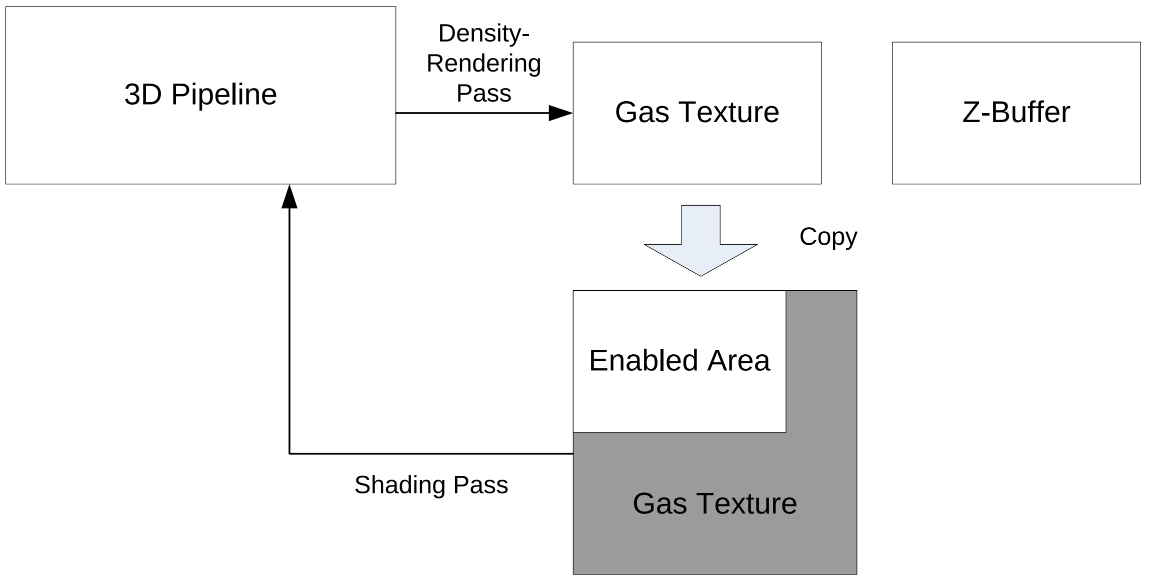
16.3 Gas Textures

Gaseous objects are defined as collections of gas particles. Gas textures store the cumulative density values for gaseous objects (these *cumulative density values* represent the net density of all the individual gas particles). Each texel of the gas texture contains both the simple cumulative net density value of the individual gas particle fragments (called $D1$) and the cumulative value that takes into account the intersections with polygonal models (called `D2`). For more details, see section 16.4 Density-Rendering Pass. The gas texture's contents are rendered during the density-rendering pass, and referenced during the shading pass. The buffer used as the render target for the density-rendering pass and the texture buffer used during the gas shading pass have the following size restrictions:

- The buffer used as the render target during the density-rendering pass must be of the same size as the Z-buffer.
- The size of the texture buffer referenced during the shading pass must be a power of two.

Due to these two restrictions, if the size of the final render buffer (the size of the Z-buffer) is not a power of two, the result from the density-rendering pass will be copied to part of a texture whose size is a power of two. This copy will then be used during the shading pass.

Figure 16-6: Use of Gas Textures



The OpenGL extension `GL_OES_framebuffer_object` is used to render gas textures. Create a framebuffer object, and specify a render buffer to which to render the density values.

When calling **`glRenderbufferStorage`** to render the density values, specify `GL_GAS_DMP` as the *`internalformat`* argument. Then, create the texture into which the gas texture will be copied. At this time, specify `GL_GAS_DMP` as the *`internalformat`* and *`format`* arguments of the **`glTexImage2D`** function. The *`type`* argument must be `GL_UNSIGNED_SHORT`. Specify `GL_NEAREST` as the minification and magnification filters, since filters will not work on gas textures. Code 16-1 below shows how to use the gas texture-related buffers when the final rendering size is 640×480. In this example, we render the gas texture to a 640x480 render buffer during the density-rendering pass, copy it to a 1024x512 texture, and use that copy.

Code 16-1: Implementation of Gas Texture-Related Buffers for Final Render Size of 640x480

```
  // generate framebuffer and renderbuffer
  glGenFramebuffers(1, (GLuint*)&gastexfb);
  glGenRenderbuffers(1, (GLuint*)&gas_acc);
  // initialize renderbuffer for density accumulation
  glBindRenderbuffer(GL_RENDERBUFFER, gas_acc);
  glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA8_OES, 640, 480);
  // attach renderbuffer to framebuffer
  glBindFramebuffer(GL_FRAMEBUFFER, gastexfb);
  glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                       GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER, gas_acc);

  // initialize texture for shading
  glGenTextures(1, (GLuint*)&gastex_shading);
  glBindTexture(GL_TEXTURE_2D, gastex_shading);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_GAS_DMP, 1024, 512, 0,
                                          GL_GAS_DMP, GL_UNSIGNED_SHORT,
0);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);

  // gas density accumulation pass
     . . . . . .

  // copy generated gas texture
  glUniform1i(LOC("dmp_Texture[0].samplerType"), GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D, gastex_shading);
  glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, 640, 480);
```

16.4 Density-Rendering Pass

During the density-rendering pass, the density values of a gaseous object are rendered by calculating the cumulative density value of all gas particles and rendering the result to a gas texture. To do this, specify the framebuffer that was created to render the gas texture as the render target for the density values.

```
  glBindFramebuffer(GL_FRAMEBUFFER, gastexfb);
```

Per-fragment operations must be set to *gas mode* in order to render the density values. Once fragment operations enter gas mode, the portion of the pipeline starting with the alpha test is replaced by a pipeline specific to gas-related processing.

Switching per-fragment operations to gas mode is done using the following code.

```
glUniform1i(LOC("dmp_FragOperation.mode"), GL_FRAGOP_MODE_GAS_ACC_DMP);
```

You can revert to the ordinary fragment pipeline mode using the following code.

```
glUniform1i(LOC("dmp_FragOperation.mode"), GL_FRAGOP_MODE_GL_DMP);
```

When the fragment pipeline mode is set to gas mode, the simple cumulative density value of each gas particle fragment (called $D1$) and the cumulative value that takes into account the intersections with polygonal models (called $D2$) are both written to the color buffer. If the R component of the post-fog fragment color (R, G, B, A) is taken to represent the fragment density D_f, $D1$ is updated to the cumulative value $D1'$ as shown in Equation 16-1 below.

Equation 16-1: Simple Cumulative Density from Each Gas Particle Fragment

$$D1' = D1 + Df$$

Likewise, $D2$ is updated to $D2'$ as shown below.

Equation 16-2: Cumulative Density Taking Polygonal Intersections into Account

$$DZ = (Zb - Zf < 0.0)?\,0.0\!:(Zb - Zf) \times EZ$$
$$ATT = (DZ > 1.0)?\,1.0\!:DZ$$
$$D2' = D2 + Df \times ATT$$

Zb indicates the depth value stored in the depth buffer, and Zf indicates the depth value of the fragment. (The depth values of the polygonal models must be written to the depth buffer ahead of time). EZ is a floating-point value that is set using the reserved uniform `dmp_Gas.deltaZ`. It represents the ratio of the attenuation of the density values in the depth direction.

Section 16.5 Shading Pass describes how the $D1$ and $D2$ values accumulated in the gas texture are used in more detail.

Be sure to disable the depth test, depth mask, and blend settings during the density-rendering pass.

```
glDisable(GL_DEPTH_TEST);
glDepthMask(GL_FALSE);
glDisable(GL_BLEND);
```

16.5 Shading Pass

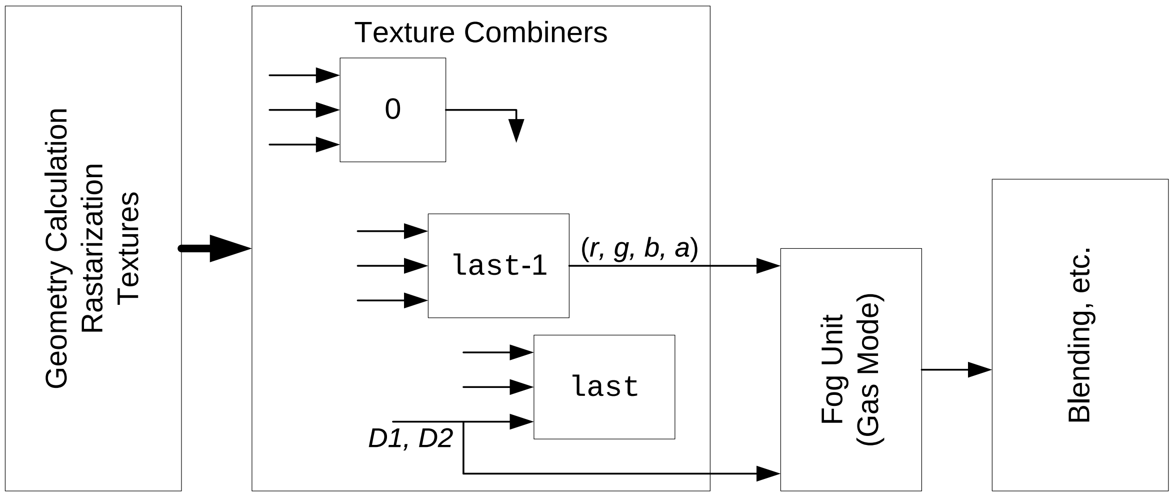
During the shading pass, gaseous objects are shaded based on their cumulative density values stored in the gas texture. Shading is done by applying the gas texture to sprites of the same size as the gas texture, and then using the fog unit (set to gas mode) to perform shading. The result of this shading is then blended with the image of the rendered polygonal models that is stored in the framebuffer.

16.5.1 Overview of Fog Operations in Gas Mode

The fog unit performs special gas shading operations when is set to *gas mode*. Set the reserved uniform `dmp_Fog.mode` to `GL_GAS_DMP` to switch the fog unit to gas mode. To restore fog operations to *fog mode* or disable fog operations altogether, set `dmp_Fog.mode` to `GL_FOG` or `GL_FALSE`, respectively.

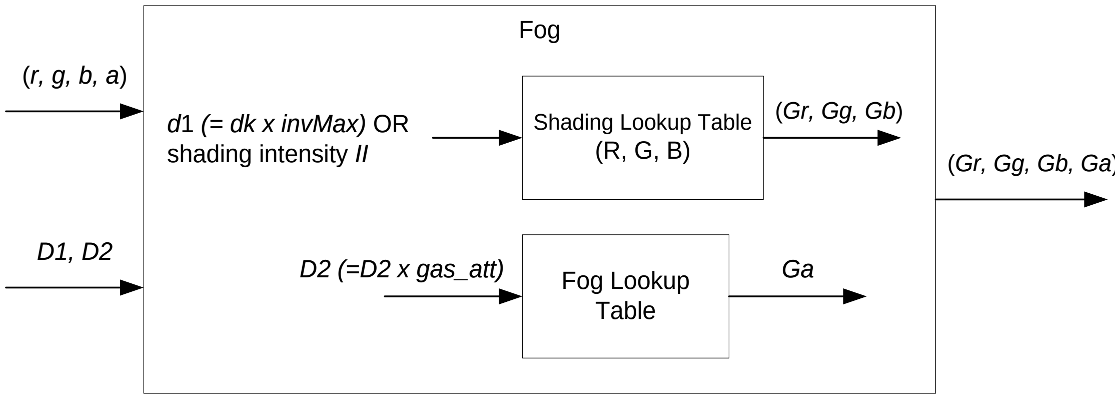
When the fog unit is set to gas mode, it takes three inputs: the output (r, g, b, a) of the next-to-last texture combiner (`dmp_TexEnv[last-1]`), the final texture combiner `dmp_TexEnv[last].srcRgb`, and the third argument to `dmp_TexEnv[last].srcAlpha`.. (If there are six texture combiners implemented in the pipeline, for example, `last` is texture combiner `5`.) Specify the gas texture that was generated during the density-rendering pass both as `dmp_TexEnv[last].srcRgb` and the third argument of `dmp_TexEnv[last].srcAlpha`. The texels of the gas texture sent as input to the fog unit are used as both $D1$ and $D2$.

Figure 16-7: Input to the Fog Unit in Gas Mode



Internally, the fog unit uses shading lookup tables and a fog lookup table to calculate the RGB and alpha values that serve as the shaded result.

Figure 16-8: Overview of Fog Operations in Gas Mode



16.5.2 RGB Values from Shading

The density $d1$ is used to calculate the RGB values during shading. This density is calculated using either the $D1$ or $D2$ input to the fog unit. The formula used to calculate $d1$ is shown below.

Equation 16-3: Calculation of Density During Shading

$$d1 = dk \times invMax$$

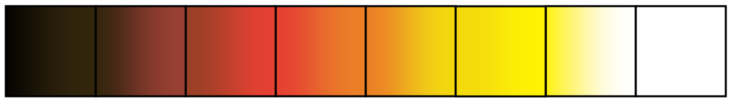
Here, either $D1$ or $D2$ can be chosen for the dk term by setting the reserved uniform `dmp_Gas.shadingDensitySrc` to either `GL_GAS_PLAIN_DENSITY_DMP` or `GL_GAS_DEPTH_DENSITY_DMP`, respectively. The $invMax$ term is the scaling value used to restrict $d1$ to the range [0.0, 1.0]. If the reserved uniform `dmp_Gas.autoAcc` has been set to `GL_TRUE`, the inverse of the maximum value of $D1$ from the density-rendering pass is used as $invMax$. When `dmp_Gas.autoAcc` has been set to `GL_FALSE`, the floating-point value set for the reserved uniform `dmp_Gas.accMax` is used as $invMax$. The resulting density $d1$ can be used both to calculate the shading intensity II (described in more detail in section 16.5.2.4), and as the input value for the shading lookup tables. The output of the shading lookup tables is output from the fog unit as the RGB values (Gr, Gg, Gb) of the shading result.

16.5.2.1 Configuring the Shading Lookup Tables

The shading lookup tables are 16-entry tables that store the R, G, or B components separately. The first eight entries in each table are used to store the values for the color component in question. The last eight entries are used to store the deltas between adjacent entries in the first half of the table. Figure 16-9 shows the contents of a sample shading lookup table.

Figure 16-9: Sample Shading Lookup Tables

R	0.00	0.20	0.60	0.90	0.92	0.95	1.00	1.00	1.00
G	0.00	0.15	0.25	0.25	0.60	0.85	0.95	1.00	1.00
B	0.00	0.05	0.15	0.20	0.15	0.05	0.00	1.00	1.00



To configure this type of shading lookup table, for each color component prepare an array with 16 elements as shown in Code 16-2 below, using the names `shading_table{R,G,B}`.

Code 16-2: Configuring Shading Lookup Tables

```
// define shading color table contents: note that RGB8 is used
// only for calculating deltas.
const float color_define[3*9] =
{
    0.00f,  0.00f,  0.00f,  /* RGB0 */
    0.20f,  0.15f,  0.05f,  /* RGB1 */
    0.60f,  0.25f,  0.15f,  /* RGB2 */
    0.90f,  0.35f,  0.20f,  /* RGB3 */
    0.92f,  0.60f,  0.15f,  /* RGB4 */
    0.95f,  0.85f,  0.05f,  /* RGB5 */
    1.00f,  0.95f,  0.00f,  /* RGB6 */
    1.00f,  1.00f,  1.00f,  /* RGB7 */
    1.00f,  1.00f,  1.00f   /* RGB8 */
};
// Arrays for shading color table
float shading_tableR[16], shading_tableG[16], shading_tableB[16];

// set up shading color table contents
for (int i=0; i<8; i++)
{
    // setting shading color value entry
    shading_tableR[i] = color_define[3*i+0];
    shading_tableG[i] = color_define[3*i+1];
    shading_tableB[i] = color_define[3*i+2];

    // setting deltas of shading color value entries
    shading_tableR[8+i] = color_define[3*(i+1)+0] - color_define[3*i+0];
    shading_tableG[8+i] = color_define[3*(i+1)+1] - color_define[3*i+1];
    shading_tableB[8+i] = color_define[3*(i+1)+2] - color_define[3*i+2];
};
```

The content of the arrays are loaded into lookup table objects.

```
  glBindTexture(GL_LUT_TEXTURE1_DMP, id_R);
  glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 16, 0,
                               GL_LUMINANCEF_DMP, GL_FLOAT,
shading_tableR);

  glBindTexture(GL_LUT_TEXTURE2_DMP, id_G);
  glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 16, 0,
                               GL_LUMINANCEF_DMP, GL_FLOAT,
shading_tableG);

  glBindTexture(GL_LUT_TEXTURE3_DMP, id_B);
  glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 16, 0,
                               GL_LUMINANCEF_DMP, GL_FLOAT,
shading_tableB);
```

To use the lookup tables during the shading pass, the reserved uniforms `dmp_Gas.samplerT{R,G,B}` must be set to the lookup table numbers to which the lookup table objects are bound.

```
  glUniform1i(LOC(”dmp_Gas.samplerTR"), 1);
  glUniform1i(LOC("dmp_Gas.samplerTG"), 2);
  glUniform1i(LOC(”dmp_Gas.samplerTB"), 3);
```

16.5.2.2 Input Values to the Shading Lookup Tables

You can choose whether to use the density $d1$ or the shading intensity as the input to the shading lookup tables. This input choice is set using the reserved uniform `dmp_Gas.colorLutInput`. To configure the lookup table to use the density as input, use the following code:

```
  glUniform1i(LOC(”dmp_Gas.colorLutInput"), GL_GAS_DENSITY_DMP);
```

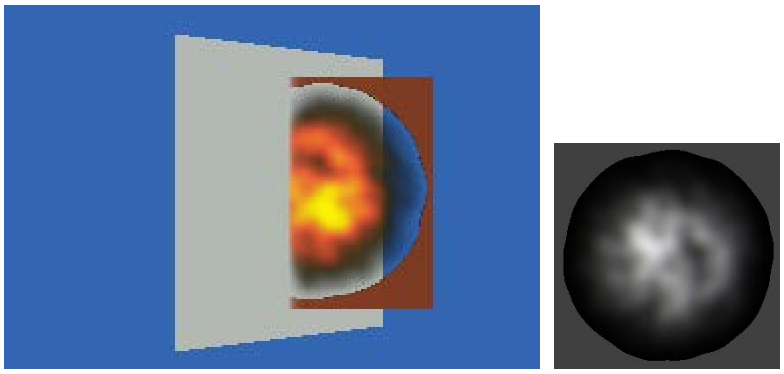
To configure the lookup table to use the shading intensity as input, use the following code:

```
  glUniform1i(LOC(”dmp_Gas.colorLutInput"), GL_GAS_LIGHT_FACTOR_DMP);
```

16.5.2.3 Referencing the Shading Lookup Tables Using Density as Input

If $d1$ is used as the input value to the shading lookup tables, the density distribution of the gaseous object is applied directly to the RGB values during shading. Let’s consider a scene like the one shown in Figure 16-10 below, in which a gas particle defined as a single sprite has been mapped with a density pattern texture, and in which this gas particle intersects with a polygon.

Figure 16-10: Sample Scene (Left) and Sample Density Pattern Texture (Right)



To make the boundaries of the sprite visible, the density pattern texture in Figure 16-10 deliberately has edges of nonzero density. The choice of dk values when calculating $d1$ will determine whether or not the shaded RGB values are affected by the areas where gas particles intersect with polygonal objects.

Figure 16-11: Shaded RGB Values Resulting from Using Density as Input

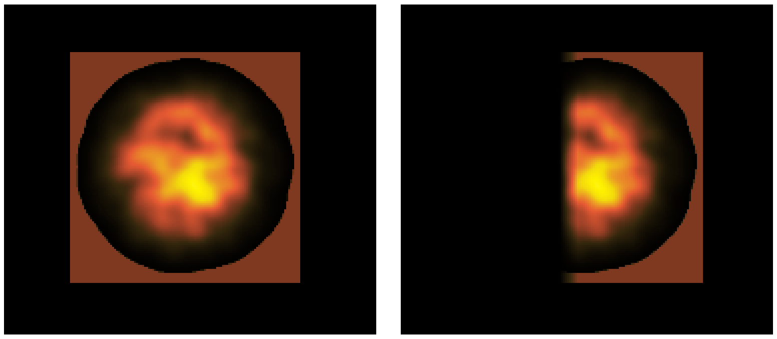


Figure 16-11 shows the results of `GL_GAS_PLAIN_DENSITY_DMP` (Left) and `GL_GAS_DEPTH_DENSITY_DMP` (Right).

If dk is set to $D1$ (`GL_GAS_PLAIN_DENSITY_DMP`), the intersections with polygonal objects will not affect the shaded RGB values. Conversely, if $D2$ is chosen (`GL_GAS_DEPTH_DENSITY_DMP`), the shading lookup tables are accessed using density values that take into account the intersections with polygonal objects. The alpha value of the shading pass is always calculated in a way that takes into account the intersections with polygonal objects (for details, see section 16.5.3 Alpha Values from Shading). Consequently, even if you use `GL_GAS_PLAIN_DENSITY_DMP`, it is still possible to use alpha blending to render gaseous objects in a way that takes into account the intersections with polygonal objects.

16.5.2.4 Referencing the Shading Lookup Tables Using Shading Intensity as Input

The shading lookup tables can be accessed using the shading intensity II as input.

Equation 16-4: Shading Intensity (II)

$$II = IG + IS$$

Here, IG indicates the planar shading intensity, and IS indicates the view shading intensity. The planar shading intensity is calculated using the following formulas.

Equation 16-5: Planar Shading Intensity (IG)

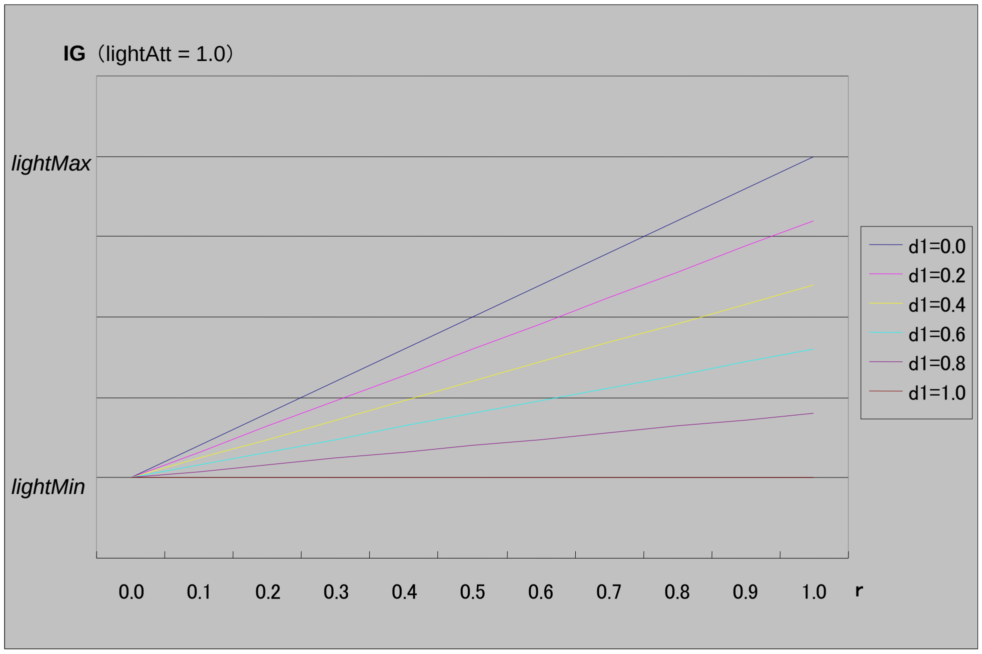
$$Perturbation = (1.0 - lightAtt \times d1)$$
$$ig = r \times Perturbation$$
$$IG = (1.0 - ig) \times lightMin + ig \times lightMax$$

Here, r is the R component of the input color to the fog unit. $lightMin$, $lightMax$, and $lightAtt$ refer respectively to the minimum intensity, maximum intensity, and attenuation due to density, all of which control planar shading. These are all set using the reserved uniform `dmp_Gas.lightXY`.

```
GLfloat lightXY[3] = {lightMin, lightMax, lightAtt};
glUniform3fv(LOC(“dmp_Gas.lightXY"), 1, lightXY);
```

Figure 16-12 through Figure 16-14 below use graphs to show the relationships between $d1$, r, and IG.

Figure 16-12: Planar Shading Intensity ($lightAtt = 1.0$)



© 2009–2012 Nintendo
CONFIDENTIAL

Figure 16-13: Planar Shading Intensity ($lightAtt = 0.6$)

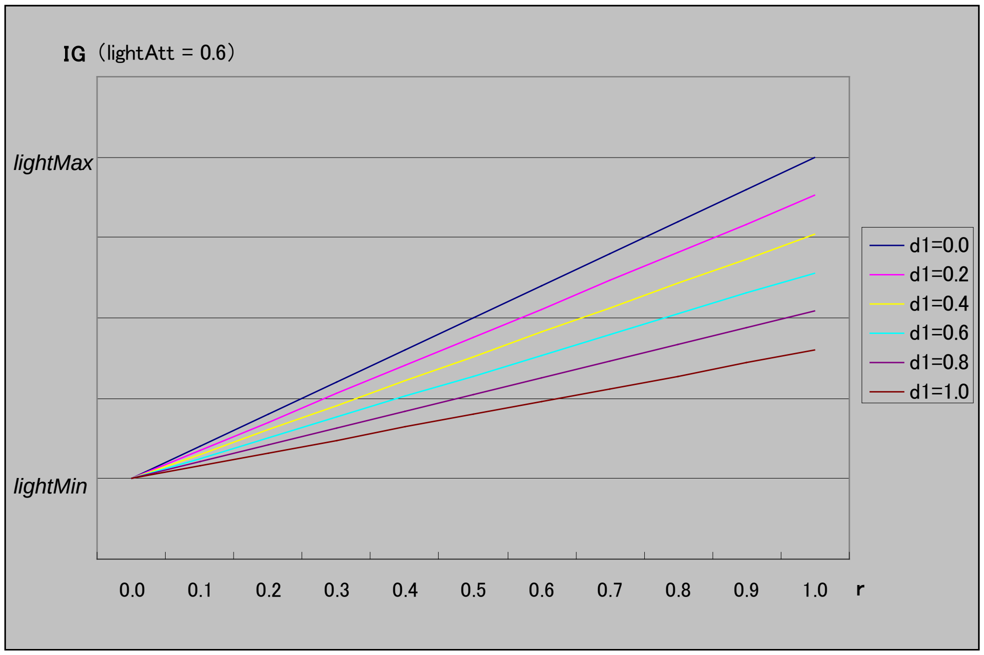
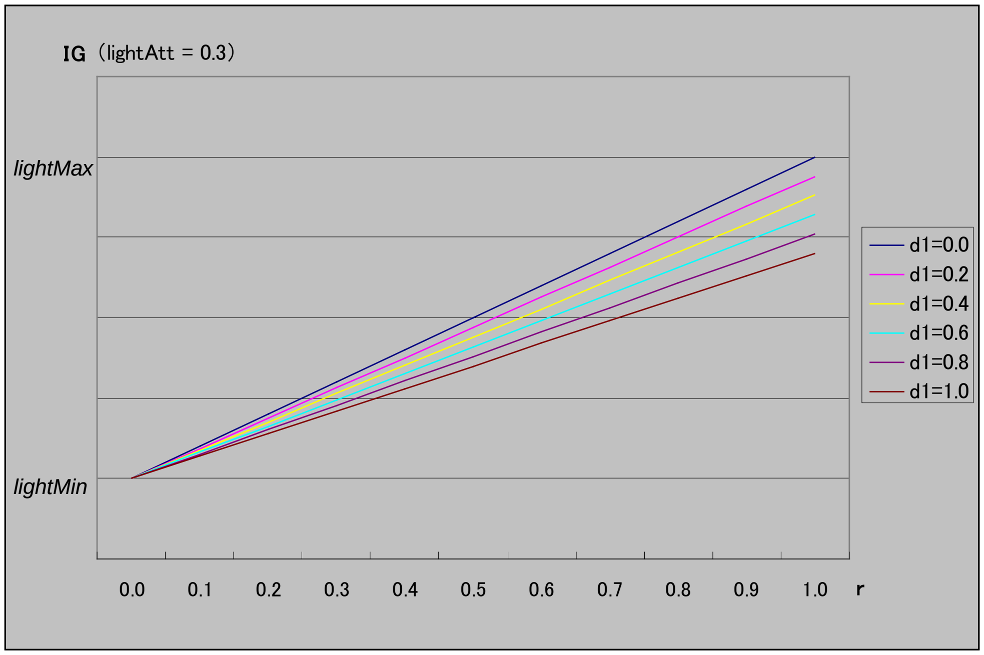
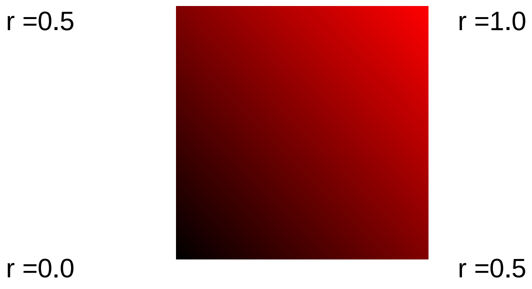


Figure 16-14: Planar Shading Intensity ($lightAtt = 0.3$)



These figures show that the planar shading intensity IG is proportional to both r and $(1 - d1)$. Also, comparing the three graphs shows that the effect of $d1$ increases proportionally to the attenuation of $lightAtt$. Figure 16-15 below is a visual representation of the effect of shading, given uniform values for $d1$ and a view shading intensity IS of zero. The shading lookup tables have been set to values that yield a transition between black and red. You can see that the color approaches red as r increases; it approaches black as r decreases.

Figure 16-15: Representation of the Effect of Planar Shading, Using Lookup Tables That Transition from Red to Black



The view shading intensity IS is calculated as follows.

Equation 16-6: View Shading Intensity (IS)

$$Perturbation = (1.0 - scattAtt \times d1)$$
$$is = LZ \times Perturbation$$
$$IS = (1.0 - is) \times scattMin + is \times scattMax$$

Here, the terms $scattMin$, $scattMax$, $scattAtt$, and LZ refer respectively to the minimum intensity, maximum intensity, attenuation corresponding to cumulative density, and the effect in the direction of the line of sight (view direction). These are all set using the reserved uniform `dmp_Gas.lightZ`.

```
GLfloat lightZ[4] = {scattMin, scattMax, scattAtt, LZ};
glUniform4fv(LOC(“dmp_Gas.lightZ"), 1, lightZ);
```

Figure 16-16 below shows the relationship between the input $d1$), the view shading intensity IS), and the effect in the direction of the line of sight LZ).

Figure 16-16: View Shading Intensity

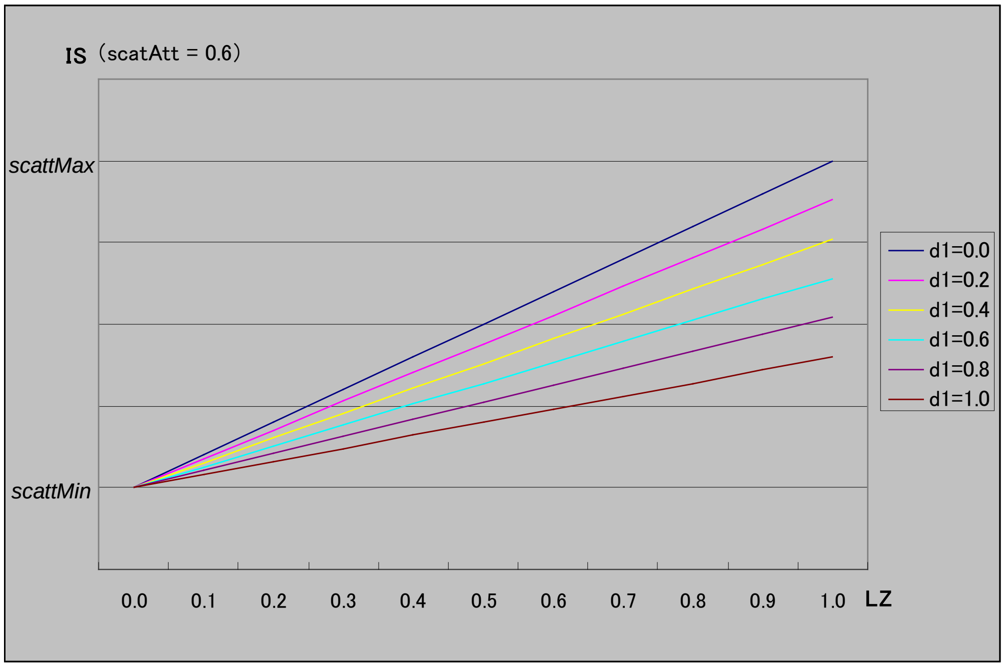
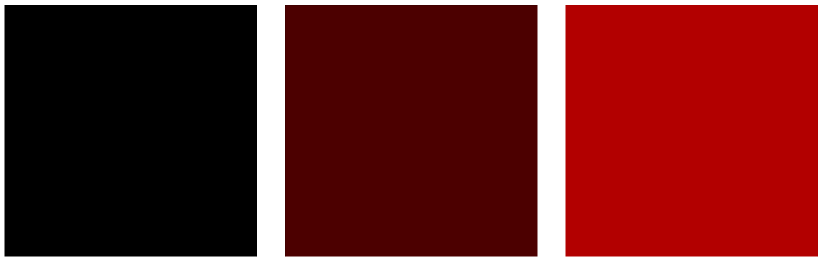


Figure 16-16 shows that the view shading intensity IS is proportional to both LZ and $(1 - d1)$. Furthermore, the effect of $d1$ increases proportionally to the attenuation $scattAtt$ for the view shading intensity, as was the case with the planar shading intensity. Figure 16-17 below is a visual representation of the effect of shading, given uniform values for $d1$ and a planar shading intensity IG of zero. You can see that the resulting color approaches red (the input to the shading lookup tables approaches 1.0) as LZ increases, and that the resulting color approaches black (the input to the shading lookup tables approaches 0.0) as LZ decreases.

Figure 16-17: Effect of IS



16.5.3 Alpha Values from Shading

The density $d2$ is used to calculate the alpha value Ga during shading. This density is based on the $D2$ input to the fog unit and is calculated using Equation 16-7 shown below:

Equation 16-7: Calculation of $d2$ Density

$$d2 = D2 \times gas\_att$$

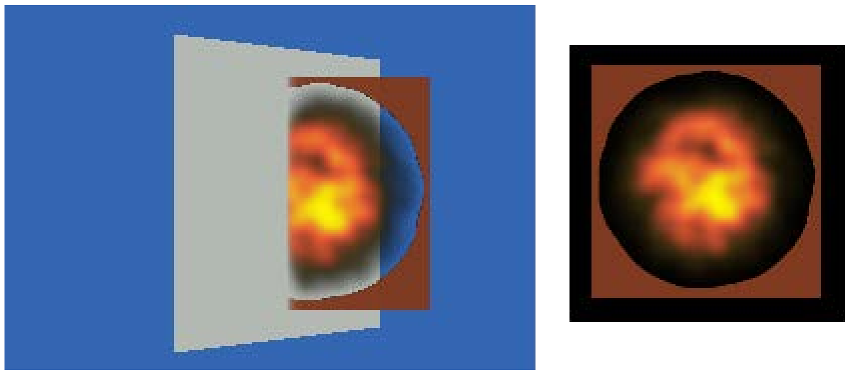
The gas_att term is the density attenuation that was set using the reserved uniform `dmp_Gas.attenuation`. $d2$ is used as the input value to the fog lookup table. Ga is the output of the fog table.

Equation 16-8: Calculation of Alpha Component Using the Fog Lookup Table

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

The output alpha component Ga of the fragments is affected by the density value $D2$ (which takes into account intersections with polygonal models). As a result, by using Ga to blend the shaded output with the rendered results from the polygonal object rendering pass, we can render gaseous objects and polygonal objects so that they have the correct foreground/background relationships.

Figure 16-18: Using the Alpha Value from Shading to Blend Gaseous Objects with Polygonal Objects



In the Figure 16-18 example above, we were able to represent the proper foreground/background relationship. This was accomplished by using Ga to blend the gas particle with the shaded RGB values shown to the right with a polygonal object.

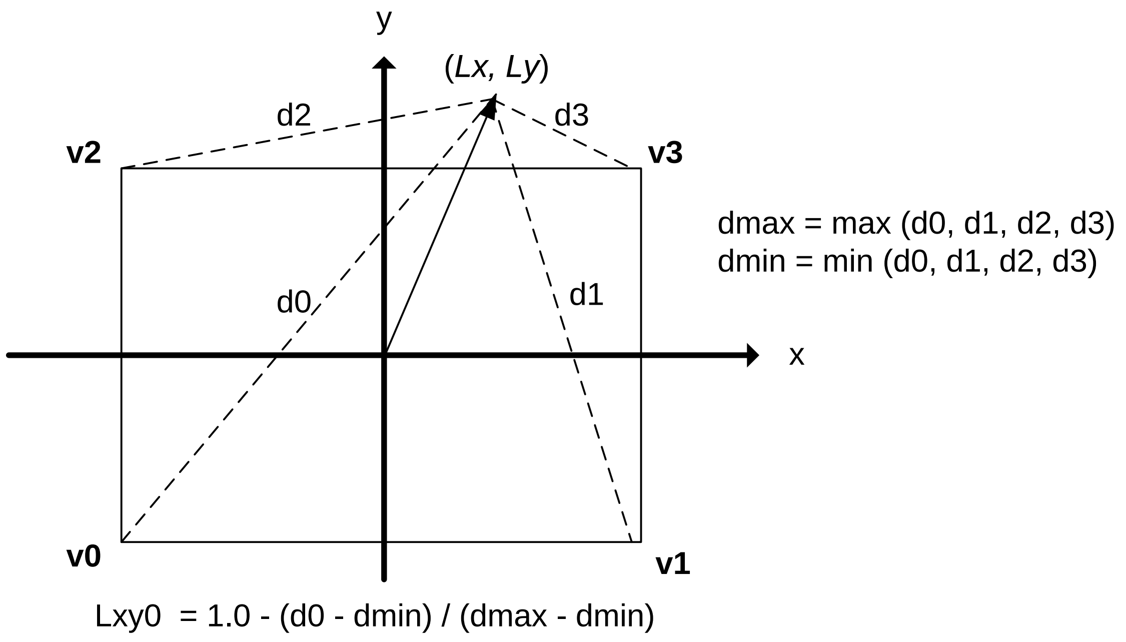
16.5.3.1 Light Definitions and Shading

In this section, we introduce one example for how the shading intensity II can be used, namely as a method for defining the light direction and allowing the light direction to affect shading. We will use the planar shading intensity IG as the effect in the light's XY directions, and use the view shading intensity IS as the effect in the light's Z direction. For more details about the shading intensity, see section 16.5.2.4 Referencing the Shading Lookup Tables Using Shading Intensity as Input.

We define the light direction (Lx, Ly, Lz) in the eye coordinate system. We then apply the effects of Lx and Ly to the planar shading, and apply the effects of Lz to the view shading. Lx and Ly can be of any value, but Lz must be a number in the range [0.0, 1.0].

First, we can control the planar shading using r (the R component of the color that was input to the fog unit).

Figure 16-19: Method for Controlling Light Direction and Planar Shading


y
(*Lx, Ly*)
d2
d3
v2
v3
dmax = max (d0, d1, d2, d3)
dmin = min (d0, d1, d2, d3)
d0
d1
x
v0
v1
Lxy0 = 1.0 - (d0 - dmin) / (dmax - dmin)

Let's assume that the four vertices v0, v1, v2, and v3 define a sprite to which a gas texture is applied during the shading pass. Here, (Lx, Ly) indicates the light vector, and dn indicates the distance between vertex vn and a representative point (Lx, Ly) in the direction of the light vector. The effect of light on each vertex, $Lxyn$, is defined by the following equation.

Equation 16-9: Effect of Light on Gas Textures

$$Lxyn = 1.0 - (dn - d_{min})/(d_{max} - d_{min})$$

Here, d_{min} and d_{max} indicate the minimum and maximum distances between that representative point in the direction of the light vector and the various vertices. Each $Lxyn$ value we find is specified as the R component of the color at the vertex in question.

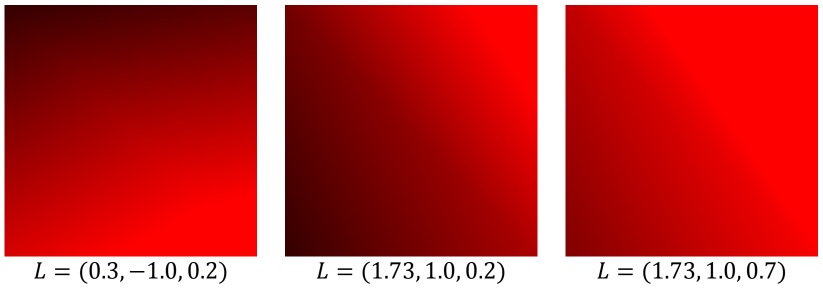
```
GLfloat color[4*4] = {0.f};
GLfloat d[4];
GLfloat lightXY = {LIGHT_X, LIGHT_Y};
GLfloat dmax = 0.f;
GLfloat dmin = 100000.f;
for (int i=0; i<4; i++) {
    d[i] = distance(lightXY, position[i]);
}
dmax = max(d);
dmin = min(d);
color[4*0+0] = 1.f - (d[0] - dmin)/(dmax - dmin);
color[4*1+0] = 1.f - (d[1] - dmin)/(dmax - dmin);
color[4*2+0] = 1.f - (d[2] - dmin)/(dmax - dmin);
color[4*3+0] = 1.f - (d[3] - dmin)/(dmax - dmin);
```

Be sure to configure the texture combiners appropriately and also make sure vertex color is sent directly as input to the fog unit.

With view shading, we can use the effect in the direction of the line of sight (LZ) to control shading. By using LZ as the Lz coordinate in the light definition, we can apply the effect of the light during shading. We have already explained that the view shading intensity IS increases as LZ increases. This is consistent with the fact that the greater Lz (the z-component of the light vector) becomes, the more closely the light direction approaches the Z-axis (which is the line of sight), which will in turn cause the effect of the light to become stronger.

Figure 16-20 below shows visual representations of how shading is affected by the settings used to define lights.

Figure 16-20: Effect of Light Definitions on Shading



In Figure 16-20, the shading tables used for all three cases were defined using a black-to-red gradation. The left and center images share the same value for Lz but have different values for Lx and Ly. You can see that the effect of the red (the color with the maximum intensity) is most prominent in the direction of (Lx, Ly). Moreover, if you compare the center and right images, you'll see they share the same values for Lx and Ly but have different Lz values. You can see that the image at the right, in which the direction of light is closer to the direction of the Z-axis, has more pronounced red tones due to the effect of the view shading intensity.

17 Clipping

This chapter describes how to program clipping operations.

17.1 Overview

The clipping functionality provided by DMPGL 2.0 is quite similar to the clipping functionality defined by the OpenGL ES 1.1 specification. There are two main aspects of clipping under DMPGL 2.0 that differ from the OpenGL ES specification: the way in which viewing volumes are defined, and the methods for specifying clipping planes.

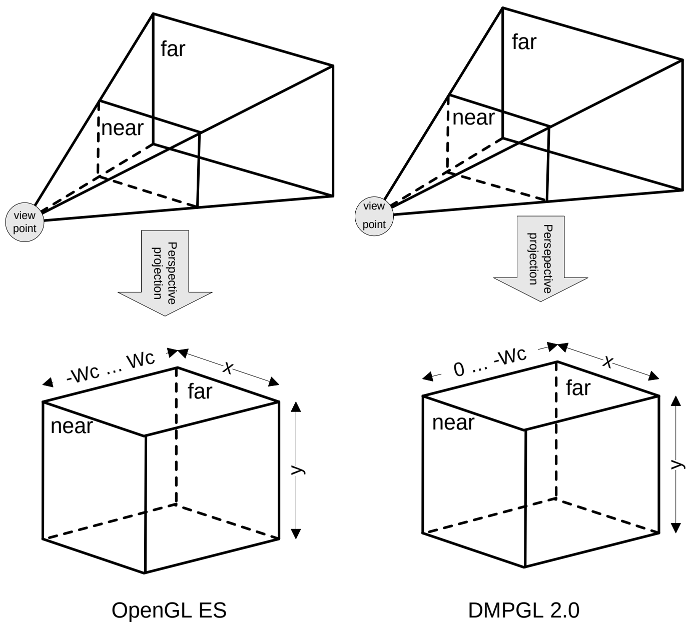
17.2 Specifying Clipping Planes

In DMPGL 2.0, the four coefficients of a clipping plane set using the reserved uniform `dmp_FragOperation.clippingPlane` must be defined in the clip coordinate system. Specify a standard OpenGL ES clipping plane to which modelview and perspective projection transformations have been applied.

17.3 Defining the Viewing Volume

The z-coordinates of viewing volumes in DMPGL 2.0 and OpenGL ES are not defined in the same way. With the OpenGL ES standard, the z-component is clipped to the range $[-w_{c,}w_c]$, but with DMPGL 2.0, it is clipped to the range $[0, -w_c]$ (note that the sign is inverted). Clipping planes must be specified with this definition in mind.

Figure 17-1: Comparison of Viewing Volumes in DMPGL 2.0 and OpenGL ES (Example of a Perspective Projection Transform)



17.3.1 Defining the Projection Matrix

As stated above, the range of the z-coordinates of the viewing volume differs between DMPGL 2.0 and OpenGL ES. As a result, the definition of the projection matrix in DMPGL 2.0 is also different from OpenGL ES. The projection matrix used for perspective transformation is defined using Equation 17-1 below. In this equation, the coordinates $(l \quad b \quad -n)^T$ represent the lower-left corner of the near clipping plane, the coordinates $(r \quad t \quad -n)^T$ represent the upper-right corner of the near clipping plane, and f represents the distance from the camera to the far clipping plane.

Equation 17-1: Perspective Projection Matrix

$$M_{frustum} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f}{f-n} & \frac{fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

The orthographic projection transformation is defined as follows.

Equation 17-2: Orthographic Projection Matrix

$$M_{ortho} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & \frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

17.3.2 Using OpenGL ES-Compatible Projection Matrices

To use OpenGL ES-compatible projection matrices in the DMPGL 2.0 environment, you must convert the projection matrix either in your application or in a vertex shader. This conversion of the projection matrix is defined by Equation 17-3.

Equation 17-3: Conversion of Projection Matrix for OpenGL ES Compatibility

$$M_{proj\_dmpgl20} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.5 & -0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} M_{proj\_oes}$$

17.3.3 Clipping-Related Precautions

Because PICA uses 24-bit floating-point numbers to convert coordinates for vertex processing, clipping is sometimes not possible to perform correctly close to the far clipping plane if the ratio of the near clipping plane to the far clipping plane is large. Either avoid using values that are larger than necessary for the near and far clipping planes or, if the ratio of the near clipping plane to the far clipping plane must be large, avoid placing polygons close to the far clipping plane.

18 Early Depth Tests

This chapter describes how to program early depth tests.

18.1 Overview

This chapter describes how to program early depth tests. The early depth test functionality in DMPGL 2.0 has the following characteristics.

- Because this depth test is performed at an early stage in the pipeline, it allows a lot of wasteful calculations to be removed from certain scenes.
- The precision of the calculation is lower than that of the standard depth test, so early depth tests must be done in conjunction with standard depth tests.
- Because the order in which the fragments are generated will change with early depth tests, the render buffer must be set to block-32 mode.
- Early depth tests cannot be run if textures are attached to the framebuffer. Moreover, even if **`glCopyTexImage2D`** or **`glCopyTexSubImage2D`** are called, rendered results will not be transferred to textures.

18.2 Structure

Early depth tests are run using a buffer called the early depth buffer. The early depth buffer has the following structure.

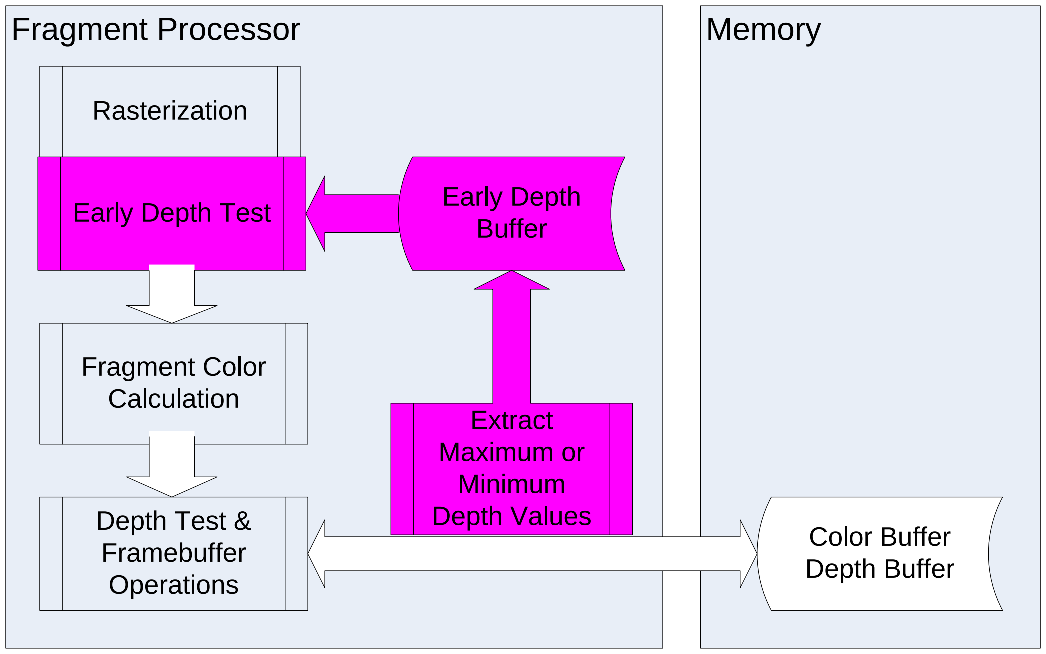
The early depth buffer contains depth values that each represent a block covering 32×32 pixels. These representative values are stored with a precision of 12 bits. When depth values are written to the standard depth buffer, the early depth values of the corresponding blocks are updated in the early depth buffer. Figure 18-1 shows the flow for updating the early depth buffer.

If early depth tests are enabled, then when values are written to the standard depth buffer, the corresponding values in the early depth buffer are updated to either the maximum or minimum depth value present within their block. The maxima are used when the early depth test comparison function is `LESS` or `LEQUAL`, and the minima are used when the comparison function is `GREATER` or `GEQUAL`.

The early depth buffer and the registers are both within the rendering pipeline, and thus there is no need to allocate memory. The early depth buffer supports depth buffer widths and heights up to 1024×1024 pixels.

The contents of the early depth buffer cannot be directly obtained or set from outside. It is only possible to initialize the buffer by setting a single initial value, and to update it based on the results of writing to the standard depth buffer. Clearing the early depth buffer clears only its contents in accordance with the early depth buffer clear value.

Figure 18-1: Early Depth Buffer Updating


Fragment Processor
Memory
Rasterization
Early Depth Test
Early Depth Buffer
Fragment Color Calculation
Extract Maximum or Minimum Depth Values
Depth Test & Framebuffer Operations
Color Buffer
Depth Buffer

The early depth test is run at the same time as rasterization. The representative depth value for each polygon is calculated at the time of rasterization and compared against the corresponding depth value extracted from the early depth buffer. If the early depth test comparison function is `LESS` or `LEQUAL`, the minimum depth value among the vertices of the polygon being rendered is taken as that polygon's representative depth value; if the comparison function is `GREATER` or `GEQUAL`, the polygon's maximum depth value is taken as its representative value. If a block passes the early depth test, all the fragments in that block proceed to subsequent fragment processing. If a block fails the early depth test, all the fragments in that block fail and are discarded. Fragments that have passed the early depth test may either pass or fail the standard depth test.

The early depth test comparison function and clear value are configured independently from the comparison function and clear value for the standard depth test. Consequently, make sure the configurations are not contradictory, or rendering may not produce the expected results.

18.3 False Passes and False Failures

The early depth test is run not on a per-fragment basis, but rather on a per-block basis. As a result, it may yield different results for the fragments in a block than the standard depth test. The term *false pass* refers to a fragment that passes the early depth test, even though it should have failed. The term *false failure* refers to a fragment that fails the early depth test, even though it should have passed.

If the early depth test results in a false pass for a given fragment, there will be no effect on the rendered result, since the fragment will fail the standard depth test. If a lot of false passes occur, the rendering speed will drop to a level that makes the early depth test ineffective, but the rendered results will not be corrupted.

If a false failure occurs, however, the fragment (which is required for the correct rendered result) is discarded immediately. Be careful not to let false failures occur as a result of your settings.

18.4 Configuring the Block Mode

DMPGL 2.0 allows the render buffer to be set to one of two modes: *block-8 mode* and *block-32 mode*. You must use block-32 mode if using early depth tests.

Block-8 mode is the default and is set whenever early depth tests are not used.

Block-32 mode is set when early depth tests are used. In this mode, the relationship between the fragments being rendered and their addresses is different from 8-block mode. The width and height (in pixels) of the render buffer acting as the render target must be a multiple of 32. In addition, the functions **`glCopyTexImage2D`** and **`glCopyTexSubImage2D`** will not copy the rendered results properly when handling color buffers that were rendered in block-32 mode.

The rendered results are not guaranteed if the render buffer's block mode is changed during rendering. Within each single scene, be sure to use the same settings for the functions **`glDrawElements`**, **`glDrawArrays`**, and **`glReadPixels`**.

When we say that the block mode cannot be changed during rendering, we mean that rendering even a single object with early depth testing enabled requires that all other all other objects in that scene also be rendered in block-32 mode. In that situation, **`glReadPixels`** must also be called in block-32 mode.

```
glRenderBlockModeDMP(GL_RENDER_BLOCK8_MODE_DMP); // Set block-8 mode
glRenderBlockModeDMP(GL_RENDER_BLOCK32_MODE_DMP); // Set block-32 mode
```

You can switch between block-8 mode and block-32 mode by using the code snippets above.

18.5 Enabling and Disabling Early Depth Tests

If standard depth tests are disabled, disable early depth tests as well. If early depth tests are enabled while standard depth tests are left disabled, false failures may occur, which will cause incorrect rendered results.

```
glEnable(GL_EARLY_DEPTH_TEST_DMP);  // Enable early depth tests
glDisable(GL_EARLY_DEPTH_TEST_DMP); // Disable early depth tests
```

The code snippets above can be used to enable and disable early depth tests.

18.6 Comparison Functions for Early Depth Tests

The four modes that can be used with the early depth test comparison functions are: `LESS`, `LEQUAL`, `GREATER`, and `GEQUAL`. Although the API allows the early depth test comparison functions to be set independently of the standard depth test comparison function, setting different comparison functions for the standard and early depth tests may cause false failures.

One use case that doesn't use only the `LESS` comparison mode involves inverting the depth direction once per frame and alternating between the use of the `LESS` and `GREATER` modes. A more concrete example would be to call **`glDepthRangef`** for even-numbered frames and set the *`zNear`* value to 0.0, the *`zFar`* value to 0.5, and the depth test comparison mode to `LESS`, then call **`glDepthRangef`** for odd-numbered frames and set the *`zNear`* value to 1.0, the *`zFar`* value to 0.5, and the depth test comparison mode to `GREATER`, and then render. This results in less-precise depth values, but avoiding clear operations on the depth buffer for each frame increases the speed. (Rendering this way assumes that all pixels are rendered for each frame. Moreover, you must clear the depth buffer with 0.5 for just the first frame.) However, when using early depth tests, you still need to clear the early depth buffer at the start of each frame, even when using them in this way. This is because the early depth buffer representative value is not updated afterwards for written pixels until the buffer is cleared. See 18.8 Depth Buffer Update Flag for details.

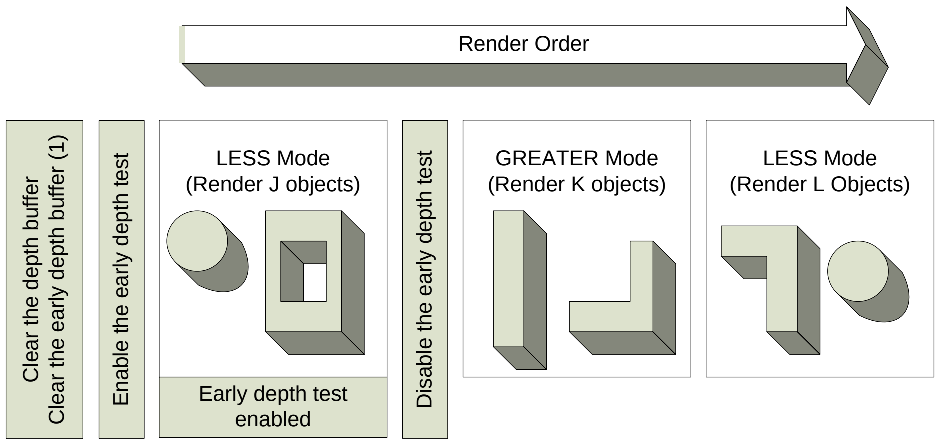
Regardless of whether you change the comparison function within a single frame, the following restrictions apply.

- **Restriction 1:** The comparison function for the early depth test must be the same as the comparison function for the standard depth test.
- **Restriction 2:** Once the comparison function for the early depth test is set, it cannot be changed until the early depth buffer is cleared.
- **Restriction 3:** Once the depth test comparison function is changed for a given render buffer and rendering is performed, early depth tests cannot be enabled until the early depth buffer is cleared.

If any of these three restrictions are violated, false fails may occur, which will cause the rendered results to be incorrect. An example of this is described below.

Example: Let's assume that in a scene in which a (J+K+L) number of objects are being rendered, you want J objects rendered in `LESS` mode, K objects rendered in `GREATER` mode, and L objects rendered in `LESS` mode.

Figure 18-2: Example of Early Depth Test Use



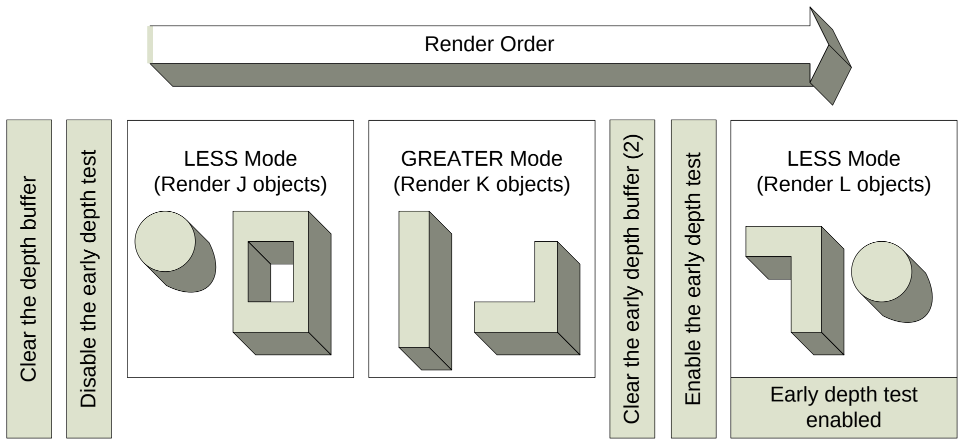
In the Figure 18-2 example above, the early depth buffer and the depth buffer are cleared simultaneously. Once early depth tests are enabled for the first J objects, early depth tests cannot be used for objects in `GREATER` mode. This is clear based on the first and second restrictions. Due to the third restriction, early depth tests cannot be enabled afterward either, even though the remaining L objects are rendered by returning to `LESS` mode. As a result, unless the early depth buffer is cleared, only the first J objects can be rendered with early depth tests enabled.

If the early depth buffer and the standard depth buffer are cleared simultaneously, the early depth clear value can be set to the same value as the standard depth clear value.

If the performance of the final L objects is especially important, and you absolutely need to use early depth tests on the final L objects, you might consider using one of the workarounds described below.

18.6.1 Workaround 1

Figure 18-3: Workaround 1

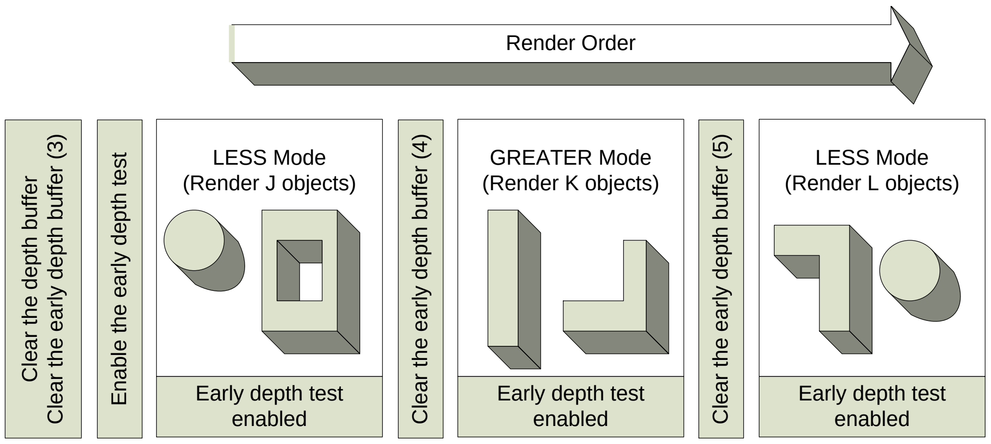

Render Order
Clear the depth buffer
Disable the early depth test
LESS Mode (Render J objects)
GREATER Mode (Render K objects)
Clear the early depth buffer (2)
Enable the early depth test
LESS Mode (Render L objects)
Early depth test enabled

When rendering the first J+K objects, disable early depth tests and then enable early depth tests for the first time when rendering the final L objects. This way the early depth buffer is cleared right before the final L objects are rendered.

When using this approach (namely, clearing the early depth buffer and the standard depth buffer at different times), using either the maximum or minimum depth values as the clear values will ensure that no false failures will occur. In the case of workaround 1 as shown in Figure 18-3, an early depth clear value of `0xffffff` is the most appropriate in the stage labeled "*Clear the early depth buffer (2)*."

18.6.2 Workaround 2

Figure 18-4: Workaround 2



Enable early depth tests, and render J objects. Then, clear the early depth buffer before rendering the next K objects, and also clear the early depth buffer before rendering the final L objects. This approach to rendering allows you to change the early depth comparison function. However, clearing the early depth buffer very often will increase the incidence of false passes.

For workaround 2 as shown in Figure 18-4, the most appropriate clear value for the stage labeled "*Clear the early depth buffer (3)*" is the value calculated based on the standard depth buffer clear value. In the same way, `0x000000` is an appropriate clear value for the stage labeled "*Clear the early depth buffer (4)*," and `0xffffff` is an appropriate clear value for the stage labeled "*Clear the early depth buffer (5)*."

If you use a comparison function for standard depth tests that is not available for early depth tests (for example, `EQUAL` or `NOTEQUAL`), you must disable early depth tests.

```
glEarlyDepthFuncDMP(GL_LESS); // Set early depth tests to LESS mode
```

The code snippet above sets early depth tests to `LESS` mode.

18.7 Clearing the Early Depth Buffer

The API allows the early depth buffer to be cleared independently from the standard depth buffer. However, if using the early depth buffer, be sure to clear the early depth buffer whenever you clear the standard depth buffer.

You cannot easily change the comparison function for the early depth test. If you need to change the depth test comparison function while rendering a certain scene, then you cannot use early depth tests in that scene. However, in this situation you can avoid the early depth test restrictions by clearing only the early depth test buffer and not the standard depth test buffer. (For details, see section 18.6 Comparison Functions for Early Depth Tests.)

Use an early depth buffer clear value that is functionally the same as the clear value for the standard depth buffer, as explained below.

The clear value for the standard depth test is set to a `float`-type value. The clear value for the early depth test, on the other hand, must be a non-negative integer. The equation below gives a rough guide for how to calculate the early depth test clear value (written as $Depth_{Early}$).

Equation 18-1: Calculating the Early Depth Buffer Clear Value

$$Depth_{Early} = Depth \times \mathrm{0xffffff} + offset$$

In the equation above, the value chosen for $offset$ must take into account the fact that the early depth test is calculated with lower precision than the standard depth test, and must be chosen to prevent the occurrence of false failures. For `LESS`, a positive number of `0x1000` or greater is recommended. For `GREATER` mode, a number of `-0x1000` or less is recommended. If the result after adding $offset$ is outside of the range `0x000000-0xffffff`, take steps to force the value of $Depth_{Early}$ to be within that range.

```
glClearEarlyDepthDMP(DepthEarly);         // Set the clear value
glClear(GL_EARLY_DEPTH_BUFFER_BIT_DMP);   // Run the clear operation
```

18.8 Depth Buffer Update Flag

The early depth buffer saves early depth values as the representative values for each 32×32 pixel block, as well as a 1-bit depth buffer update flag for each 4×4 pixel block.

The depth buffer update flag is zeroed out at the same time that the early depth buffer is cleared. Standard depth test results cause the standard depth buffer to be updated and also set the depth buffer update flags corresponding to those pixels to 1. When the depth buffer is updated for even just one pixel in a 4×4 block, the depth buffer update flag is set to 1. When this flag is set to1, it remains set to 1, no matter how many times the standard depth buffer is updated thereafter, until the early depth buffer is cleared.

We have already described how the early depth value in the early depth buffer for a pixel block is updated when the depth data is written to the standard depth buffer, but when the standard depth buffer is updated, the early depth value is only updated if the depth buffer update flag corresponding to those pixels is 0.

In other words, after the depth buffer has been updated once for a pixel and the 4×4 block including that pixel, the early depth value for the corresponding 32×32 pixel block will not be updated no matter how many times the depth buffer is updated. (Of pixels belonging to the same block, the early depth value can only be updated for those pixels for which the depth buffer update flag is set to 1.)

This means that the effects of the early depth test are very dependent on the order in which the model is rendered.

For instance, when the comparison function is set to the `LESS` mode, rendering the model starting from the back means that the early depth buffer would not be updated when later rendering the model in the front, resulting in an increase in the number of fragments that pass the early depth test, causing the test to have less of an effect. Conversely, rendering a model in the front first increases the number of fragments that fail the early depth test, increasing the effect of the test.

To increase the effect of the early depth test when the comparison function is set to the `LESS` or `LEQUAL` modes, Nintendo recommends rendering models starting from the front and proceeding towards the back. When the comparison function is set to the `GREATER` or `GEQUAL` modes, Nintendo recommends rendering models starting from the back and proceeding towards the front. (Nintendo also recommends the same rendering order when using just the standard depth test, but the effect is greater for the early depth test.)

Rendering in some other order, such as rendering starting from the background when the comparison function mode is set to `LESS`, means that all of the early depth buffer values are updated with the background's depth values, even when rendering with the early depth test enabled, such that all fragments rendered in front of the background pass the early depth test, with the early depth test having no effect. In such cases, you should disable the early depth test just when rendering the background, and then enable it again when rendering the other models to ensure that the early depth test has some effect, albeit limited.

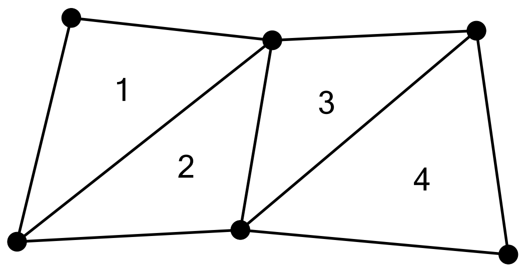
19 Performance-Enhancement Techniques

This section describes techniques for improving the performance of your graphics code.

19.1 Creating Vertex Indices

With DMPGL 2.0, when the *mode* argument of the **glDrawElements** function is set to GL_TRIANGLES, vertex data will be processed most efficiently if you create your vertex indices such that adjacent triangles share the same vertex indices. Figure 19-1 below explains this visually.

Figure 19-1: How to Create Efficient Vertex Indices



In the figure above, specifying the triangles in the order 1, 2, 3, 4 (or the reverse order) will result in the most efficient processing. With indices like these, vertex operations will be more efficient if you use GL_TRIANGLES than if you use GL_TRIANGLE_STRIP.

19.2 Existence of Vertex Buffers

The performance will drop considerably if **glBindBuffer** and/or **glBufferData** are not used to set up the vertex buffer. Using a vertex buffer causes data to be loaded into PICA's geometry pipeline. Failing to use a vertex buffer, however, will cause the CPU to re-sort the vertex arrays to match the vertex index arrays, convert all vertex data to 24-bit floating-point values, and pack the data into the command buffer. This process not only significantly increases the load on the CPU, it also reduces PICA's efficiency in loading data into the geometry pipeline and requires the command buffer to be larger. If packing vertex data into the command buffer, the required buffer size will be the number of vertices multiplied by the number of vertex attributes, multiplied by 12 bytes. (All four components (xyzw) of all vertex data is converted into 24-bit floating-point values, so each attribute will require 4x24/8 = 12 bytes.)

Placing the vertex buffer in VRAM will be faster than placing it in FCRAM.

If the vertex buffer is placed partially in VRAM and partially in FCRAM, its speed will be limited to that of FCRAM.

19.3 Data Structure of Vertex Arrays

There are two structures that can be used for vertex arrays: (1) interleaved arrays in which vertex data is stored as an array of structures, each of which includes multiple vertex attributes and (2) independent arrays that store vertex data as an array of single vertex attributes.

If a vertex buffer is used, interleaved vertex arrays are more efficient than independent arrays at loading vertex data. The actual time spent loading vertex data is often obscured by operations like the vertex shader and rasterization that take place later in the pipeline. However, if the vertex data is stored in FCRAM, using the more efficient approach at loading data will reduce the burden of accessing FCRAM, and can therefore sometimes contribute to higher-speed CPU operations.

19.4 Implementing Vertex Shaders

This section describes a few methods related to implementing shader assembly code that can yield improved performance.

19.4.1 Optimization Using Instruction Dependencies

Dependencies between instructions that are issued in shader assembly language can sometimes cause stalls to occur. You can improve the throughput of your program by reordering instructions to avoid these stalls. Below is one specific example.

```
add  r0,   r1,   r2
mul  r4,   r0,   r3
```

This code stores the sum of the `add` instruction in register r0, and immediately thereafter specifies r0 as a source register of a `mul` instruction. As a result, when the `mul` instruction is issued, it must wait for the `add` instruction to finish, causing a stall. In such cases, insert the required number of independent instructions (that is, instructions that don’t use any registers with dependencies on the surrounding code) between the `add` and `mul` instructions. This will ensure that the value of r0 is already known when the `mul` instruction runs. Table 19-1 below shows for various instructions the required number of independent instructions to insert to avoid dependency stalls. The latency (in cycles) of each instruction is one greater than the number of independent instructions indicated.

Table 19-1: Assembly Language Instructions and Number of Independent Instructions to Insert

Instruction	Number of Independent Instructions to Insert
`add`	2+
`dp3`	4+
`dp4`	4+
`dph`	4+
`dst`	2+
`exp`	3+
`litp`	1+
`log`	3+
`mad`	3+
`max`	1+
`min`	1+
`mov`	1+
`mul`	2+
`rcp`	3+
`rsq`	3+
`sge`	1+
`slt`	1+
`flr`	1+
`cmp` (Insert the independent instructions between the conditional branch instructions like `ifc` and `callc`)	3+

In addition to considering the number of instructions to insert, you can increase the chances of avoiding stalls by taking into account the latencies of the independent instructions you insert. The latency of each inserted instruction should be greater than or equal to that of the dependent instruction it is inserted after. Here is an example:

```
exp        r0.x,       r1.x  // The latency is 4, so (at least) three
                                   // instructions should be inserted
mov        r2,         r3
mov        r4,         r5
mov        r6,         r7
mul        r8,         r0,   r0
```

The fact that the `exp` instruction has a latency of four means that we should insert three or more instructions between the `exp` instruction and the `mul` instruction that uses the result of the `exp` instruction. However, rather than inserting three `mov` instructions (each of which has a latency of two), the design of the hardware makes it more efficient to change the instruction order by inserting three instructions (like `mad` or `dp3`) that each have a latency of four or higher.

In addition, having multiple registers of the same type and index will result in a dependency even if the registers' components are different, and this can cause a stall. See the section Stalls Due to Instruction Dependencies in the *Vertex Shader Reference Manual* for details.

19.4.2 Optimization by Avoiding Unconditional Stall Instructions

Instructions that change the program counter in a non-sequential fashion will cause an unconditional stall of three clock cycles. Examples of such instructions include branch control (`if`-type and `call`-type instructions), `loop` instructions, and jump instructions. Such stalls will not occur if no branching takes place and therefore the program counter changes sequentially. The `mova` instruction will also cause an unconditional stall of three clock cycles. The performance of your applications can be increased by reducing the number of times these instructions are used.

19.5 Data Types, Data Size of Vertex Attributes, and the Relationship with Transfer Speed

If you are using a vertex buffer, the way in which data type (type for `glVertexAttribPointer`) and data size (size for `glVertexAttribPointer`) of the vertex attribute are used together will affect the transfer speed of the vertex data.

The vertex attribute data stored in the vertex buffer is loaded into the GPU as a single vertex attribute or a group of several vertex attributes. The unit in which the vertex attributes are loaded is described as a *load array* in the *DMPGL 2.0 System API Specifications*. For the DMPGL 2.0 interface, when data for one vertex attribute is placed without gaps (an independent array), then one vertex attribute becomes one load array. When data for multiple vertex attributes become elements of structures and the vertex attribute data is placed as an array of those structures (an interleaved array), the data for the multiple vertex attributes that makes up those structures become one load array.

When the GPU transfers each load array, the combination of the data type and data size for the vertex attribute(s) that makes up the load array will determine whether a pre-read transfer is used. When a pre-read transfer is used, the transfer speed is faster compared to when a pre-read transfer is not used.

When the following conditional equation is met, a pre-read transfer is used. When it is not met, a pre-read transfer is not used.

([*The number of attributes with a type other than `GL_FLOAT`*] + [*The number of attributes with a data size of 1*]) <= ([*The number of attributes that are type `GL_FLOAT` and with a data size of 4*] + [*The number of attributes that are type `GL_FLOAT` and with a data size of 3*] / 2)

For [*the number of attributes with a type other than `GL_FLOAT`*], the data size does not matter. For [*the number of attributes with a data size of 1*], the data type does not matter. For vertex attributes that meet multiple conditions, count them for each one. For example, when the data size is 1 and the vertex attribute is of the `GL_BYTE` type, count it for both [*the number of attributes with a type other than `GL_FLOAT`*] and [*the number of attributes with a data size of 1*].

When the conditions are the same for whether pre-read transfer is to be used, the transfer speed is dependent on the amount of data in the load array. When there is a small amount of data, the transfer speed will faster than if there is a larger amount of data. When the amount of vertex data is the same, the transfer speed will depend on the number of attributes included in the load array. When there are fewer attributes in the load array, the transfer speed will be faster than when there are more attributes.

19.6 Vertex Array Address Alignment

If you are using a vertex buffer for rendering, the transfer of vertex array data may proceed more efficiently if the address of the vertex array is aligned to 32 bytes. The address of the vertex array is the value calculated by adding the offset specified by `glVertexAttribPointer` (the value specified by `ptr`) to the address of the vertex buffer.

The extent of any increase in speed achieved by aligning the address to 32 bytes will depend on the vertex attribute type, the size, the location where the vertex array is stored, and the content of the vertex index. There may not, necessarily, be any improvement. Merely improving the transfer speed performance will not lead to any overall improvement in system performance, if vertex array transfers are not a performance bottleneck.

19.7 Improving Vertex Cache Performance

The vertex pipeline caches up to 32 data entries on which vertex operations were performed, using the vertex indices as keys.

If the same vertex indices are used repeatedly, by positioning them as close as possible cache hits occur and vertex processing is not performed again, thereby improving the vertex processing performance.

The functionality of the vertex-caching algorithm is close to the functionality of LRU, but it is actually implemented by an independent algorithm. If the same vertex indices are used repeatedly, and are within the 32 entries, the likelihood of cache hits increases.

The efficiency of the vertex cache is affected by factors other than the order of the indices, such as the usage state of the memory where the index array is stored and the length of the shader running in the index shader. The optimal index depends on the content and might not be uniquely determined.

19.8 Vertex Shader Output Attributes

Output attributes defined by the vertex shaders (using `#pragma output_map`) are not actually defined if those vertex attributes are not used by the application. Because defined output attributes must be written to output registers, unnecessary output attributes entail unnecessary shader assembly instructions. Also, the existence of an output attribute definition causes clock control to be run in some hardware circuits, so unnecessary output attributes can lead to unnecessary power consumption.

19.9 Configuring Textures

Texture settings affect performance. Some performance-related trends are listed below.

- ETC compressed textures are the fastest texture format. The next-fastest textures are the ones that use the fewest bytes per pixel.
- The smaller the texture size, the faster the performance.
- The greater the number of textures used simultaneously, the slower the performance. This is due to conflicts during memory access.
- Given a texture with a single mip level, the minification filter mode does not affect performance. In other words, `GL_NEAREST` and `GL_LINEAR` have the same performance. Likewise, `NEAREST_MIPMAP_NEAREST` and `GL_LINEAR_MIPMAP_NEAREST` have the same performance, as do `GL_NEAREST_MIPMAP_LINEAR` and `GL_LINEAR_MIPMAP_LINEAR`. However, `GL_NEAREST(_XXX)` and `GL_LINEAR(_XXX)` fetch a different number of texels around a single pixel: 1 and 4, respectively. It can therefore be said that `GL_NEAREST(_XXX)` uses less memory.
- When mapping large textures to small areas, performance is faster if the texture is mipmapped. However, the processing load will change based on the filter mode, even if the textures are mipmapped. `GL_XXX_MIPMAP_LINEAR` will sometimes have roughly double the processing load of `GL_XXX_MIPMAP_NEAREST`.
- Although the `GL_NEAREST` and `GL_LINEAR` magnification filter modes have nearly the same performance, `GL_NEAREST` is sometimes slightly faster.
- Mipmapping cannot be used with gas textures and shadow textures, so these textures have slower performance than ordinary textures.
- Variations in the configuration conditions of procedural textures do not cause any degradation in performance. They run faster than typical 2D textures.
- When you use multiple textures, it is faster to place them all together in either VRAMA or VRAMB than to place them separately in VRAMA and VRAMB.
- When rendering a scene using certain textures, rendering speeds sometimes differ depending on whether a texture is oriented horizontally or vertically in memory, even when the rendered results in both cases are identical. For instance, when you render a textured rectangular polygon, rendering is

sometimes faster when the texture is stored in memory with its upward direction matching the framebuffer's upward direction than when the texture is stored in memory at a 90° rotation. This is because storing a texture in memory with its upward direction matching the framebuffer's means that the direction along which the fragments are generated matches the direction along which the texture is loaded, resulting in improved texture cache hit rates. (Textures are read in 8x4 texel blocks, and fragments are generated along the horizontal axis in 8x8 pixel blocks.) There is no change in rendering speed when textures are rotated 180°. Also, note that the CTR's LCD screens are oriented with their shorter sides forming the top and bottom edges.

19.10 Texture Caches

The Level-1 (L1) texture cache size is 256 bytes, and the Level-2 (L2) cache is 8 KB. Within the caches, only ETC-format data is handled as is; all other data formats are converted to 32-bit formats. However, ETC-format textures that contain alpha data are not compressed within the cache, and are converted to 32-bit format. Each texture unit has one L1 cache; the L2 cache is shared between all texture units.

For each texel, a miss in the L1 cache requiring that data be fetched from the L2 cache results in a processing penalty of approximately five cycles. An L2 cache miss resulting in a fetch from VRAM results in a penalty of another 30 cycles. However, the hardware is implemented to conceal this penalty by pre-fetching texel data.

19.11 Configuring Fragment Lighting

The processing speed will change depending on the layer configuration type of fragment lighting. The fastest processing will result if the reserved uniform `dmp_LightEnv.config` is set to one of `GL_LIGHT_ENV_LAYER_CONFIG0_DMP` - `GL_LIGHT_ENV_LAYER_CONFIG3_DMP`. The next-fastest configurations are `GL_LIGHT_ENV_LAYER_CONFIG4_DMP` - `GL_LIGHT_ENV_LAYER_CONFIG6_DMP`. The slowest configuration is `GL_LIGHT_ENV_LAYER_CONFIG7_DMP`. Also, the greater of number of lights, the lower the performance will be. Settings to access bump or shadow textures will also reduce the performance.

The configurations `GL_LIGHT_ENV_LAYER_CONFIG0_DMP` – `GL_LIGHT_ENV_LAYER_CONFIG3_DMP` take one processing cycle per pixel, the configurations `GL_LIGHT_ENV_LAYER_CONFIG4_DMP` – `GL_LIGHT_ENV_LAYER_CONFIG6_DMP` take two processing cycles per pixel, and `GL_LIGHT_ENV_LAYER_CONFIG7_DMP` takes four processing cycles per pixel.

As a result of hardware properties, however, if the color buffer is configured to have write-only access it takes three cycles per pixel to write to the color buffer with the configurations `GL_LIGHT_ENV_LAYER_CONFIG4_DMP` – `GL_LIGHT_ENV_LAYER_CONFIG6_DMP`. This behavior is not exhibited for `GL_LIGHT_ENV_LAYER_CONFIG0_DMP` – `GL_LIGHT_ENV_LAYER_CONFIG3_DMP` or `GL_LIGHT_ENV_LAYER_CONFIG7_DMP`. Pixels that fail the depth test or stencil test do not require this processing time.

© 2009–2012 Nintendo
CONFIDENTIAL

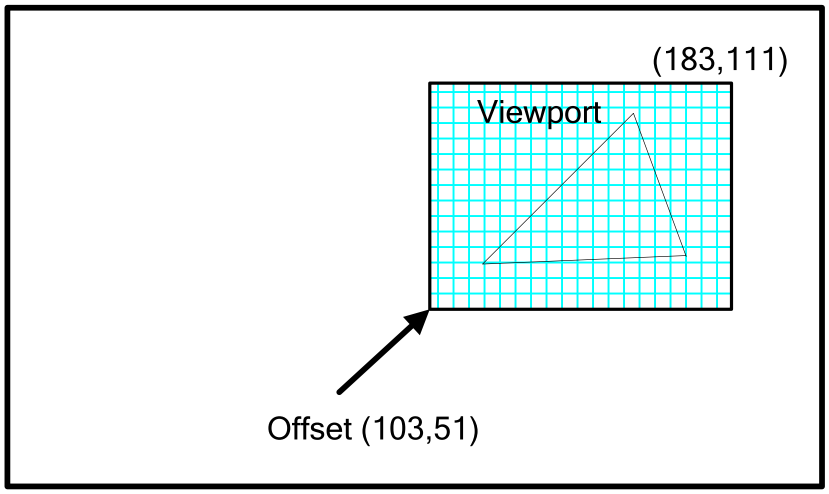
See section 19.14 Configuring Access Control to the Framebuffer for more information on the conditions that cause write-only access to the color buffer.

19.12 Configuring the Viewport

If the offsets specified to **`glViewport`** are not multiples of four, the performance will degrade. If the specified offsets contain multiples of two that aren't multiples of four, the performance will drop to about half of the ideal value. If the specified offsets contain odd numbers, the performance will drop to about one-third of the ideal value. If you need to specify viewport offsets that are not multiples of four, work around these restrictions by extending the actual viewport settings to a power of four, then adjusting the frustum during perspective projection transformation and using scissoring to remove the unnecessary portions.

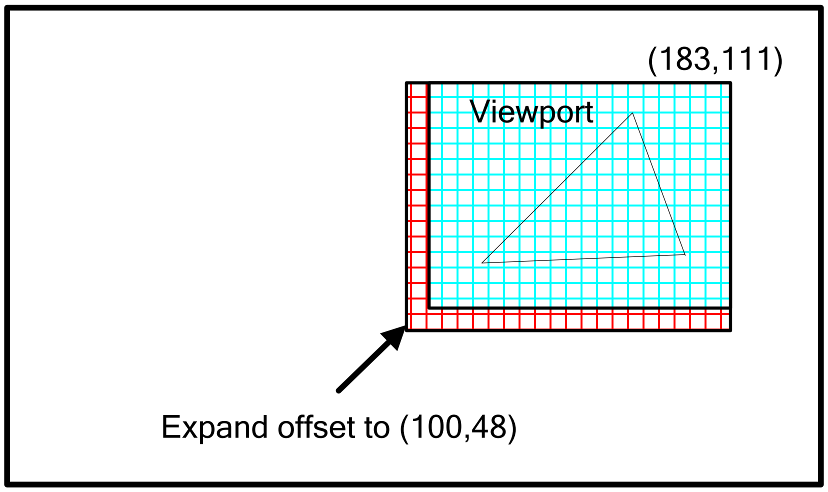
To give a concrete example, let's consider the case shown in Figure 19-2 below, which was created by calling **`glViewport(103, 51, 80, 60)`**.

Figure 19-2: Viewport with Offsets That Are Not Multiples of Four



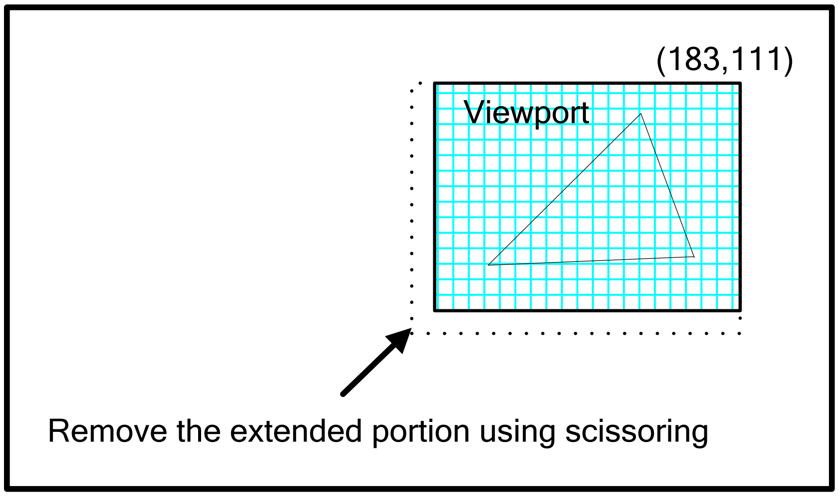
To make the viewport's offsets multiples of four, we extend the viewport region. In this case, we use the code **`glViewport(100, 48, 83, 63)`**. We then adjust the frustum so that the rendered result within the viewport will be the same size as the original viewport.

Figure 19-3: Extending the Offsets to Be Multiples of Four



We then use scissoring to prevent the extended area (the area indicated with a red fill) from being rendered.

Figure 19-4: Using Scissoring to Prevent the Extended Area from Being Rendered



19.13 Generating Shadows

When using light sources that won't move, you can reduce the processing load of the pass that generates the shadow texture by rendering only the stationary objects to the shadow texture in advance. It is also effective to simplify the geometry used during the shadow texture generation pass in order to reduce the polygon count.

19.14 Configuring Access Control to the Framebuffer

When you aren't using the color buffer, z-buffer, or stencil buffer, explicitly disabling this functionality will reduce unnecessary operations. That said, PICA uses the same physical buffer for both the z-buffer and the stencil buffer. As a result, configurations that can access either the z-buffer or the stencil buffer (but not both) have the same performance as configurations that can access both. The conditions for accessing the various buffers are listed below. To prevent unnecessary access, make sure your code doesn't bring about any of these conditions.

19.14.1 Write Access to the Color Buffer

Any of the components are set to `GL_TRUE` using **`glColorMask`**.

19.14.2 Read Access to the Color Buffer

When write access is granted and any of the following conditions are satisfied:

- `GL_BLEND` is enabled using **`glEnable`**.
- The **`glColorMask`** settings don't use the same values for all components.
- `GL_COLOR_LOGIC_OP` is enabled using **`glEnable`**.

19.14.3 Write Access to the Z-Buffer

`GL_DEPTH_TEST` is enabled using **`glEnable`** and **`glDepthMask`** is set to `GL_TRUE`.

19.14.4 Read Access to the Z-Buffer

`GL_DEPTH_TEST` is enabled using **`glEnable`**.

19.14.5 Write Access to the Stencil Buffer

`GL_STENCIL_TEST` is enabled using **`glEnable`** and **`glStencilMask`** is set to a non-zero value.

19.14.6 Read Access to the Stencil Buffer

`GL_STENCIL_TEST` is enabled using **`glEnable`**.

19.15 Note About Silhouette Rendering

DMPGL 2.0 provides two geometry shaders that can be used to render silhouettes, `DMP_silhouetteTriangle.obj` and `DMP_silhouetteStrip.obj`. They differ in terms of how vertex indices are specified. The performance of these shaders differs even when the exact same model is being rendered. With `DMP_silhouetteTriangle.obj`, each TWN is specified using six vertices. With `DMP_silhouetteStrip.obj`, after the initial TWN is specified, only two vertices are required to specify each additional TWN. You can expect `DMP_silhouetteStrip.obj` to have more than doubled the performance of `DMP_silhouetteTriangle.obj`.

19.16 CPU Performance

When using the DMPGL 2.0 API, you may be able to improve the processing speed of the DMPGL 2.0 driver by paying attention to the following points while you implement your application.

- You can link multiple shader objects into a single shader binary. We recommend linking as many vertex shader objects together as possible. There is a higher processing cost associated with switching between shader objects that are in different shader binaries than with switching between shader objects that are linked in a single shader binary.
- The locations of uniforms are fixed after **`glLinkProgram`** is called and do not change until **`glLinkProgram`** is called again. Applications should keep the location values obtained with **`glGetUniformLocation`** and use them repeatedly.
- The **`nngxSplitDrawCmdlist`** function generates a split command each time it is called. Do not call this function unnecessarily. Because the **`nngxTransferRenderImage`** function also generates a split command, it does not need to be immediately followed by a call to **`nngxSplitDrawCmdlist`**.
- Vertex data accumulates in the 3D command buffer when the vertex buffer is not used; thus, it entails a considerable increase in CPU processing over when the vertex buffer is used.
- You can decrease the cost of function calls by using texture collections and vertex state collections to bind multiple textures or set multiple vertex arrays all at once.
- All uniform values are saved for each program object. It is sometimes cheaper to switch between multiple program objects that have different uniform values set and the same shader object attached than to switch multiple uniform settings in a single program object.
- Each time data is loaded with **`glTexImage1D`** for a lookup table object, the lookup table data is converted into an internal hardware format when it is used. We recommend that you do not delete or regenerate lookup table objects wastefully.

19.17 Load Sizes for Each Data Type

This section describes the amount of data loaded when PICA loads each data type.

19.17.1 Vertex Buffers

When loading a vertex buffer, the unit size depends on the vertex index array order. The vertex index is first loaded in groups of 16 elements, and the index is sorted. Vertex arrays are then loaded in the sorted index order. Vertex arrays with sequential indices are loaded sequentially. Arrays are loaded individually for non-sequential indices.

Vertex attributes are loaded for vertex arrays that have attributes enabled by **`glEnableVertexAttribArray`**. If the DMPGL driver determines from information specified in a call to **`glVertexAttribPointer`**, such as the *`ptr`* and *`stride`* arguments, that multiple vertex attributes form an interleaved array, PICA settings are configured to load in units of interleaved arrays. In other words, multiple vertex attributes are loaded at once. (Loading in units of interleaved arrays means loading by load array. See the *DMPGL 2.0 System API Specifications* for details.)

© 2009–2012 Nintendo
CONFIDENTIAL

Sequential vertex array data with a maximum burst length greater than 256 bytes is split at 256 bytes, with the remaining data then loaded again. Vertex arrays are loaded in minimum units of 16 bytes even when non-sequential and loaded individually.

When using **`glDrawArrays`**, the same loading process is used when using a sequential index array starting from 0.

19.17.2 Textures

The transfer size is different for each texture format. If the required texel data is not found in the L1 or L2 caches, the data size read from memory is as follows.

Table 19-2: Texture Data Transfer Sizes

Texture Format	Texture Type	Unit Size for Transfers (in bytes)
GL_RGBA	GL_UNSIGNED_BYTE	128
GL_RGB	GL_UNSIGNED_BYTE	96
GL_RGBA	GL_UNSIGNED_SHORT_5_5_5_1	64
GL_RGB	GL_UNSIGNED_SHORT_5_6_5	64
GL_RGBA	GL_UNSIGNED_SHORT_4_4_4_4	64
GL_LUMINANCE_ALPHA	GL_UNSIGNED_BYTE	64
GL_HILO8_DMP	GL_UNSIGNED_BYTE	64
GL_LUMINANCE	GL_UNSIGNED_BYTE	32
GL_ALPHA	GL_UNSIGNED_BYTE	32
GL_LUMINANCE_ALPHA	GL_UNSIGNED_BYTE_4_4_DMP	32
GL_LUMINANCE	GL_UNSIGNED_4BITS_DMP	16
GL_ALPHA	GL_UNSIGNED_4BITS_DMP	16
GL_ETC1_RGB8_NATIVE_DMP	-	128
GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP	-	32

Note: If the texture format is PICA native format, the data size is the same as for the corresponding source format.

19.17.3 Command Buffers

Command buffers are command lists for setting PICA registers. (See the *DMPGL 2.0 System API Specifications* for details on command buffers.) Command buffers are loaded in units of 128 bytes.

19.18 Optimization Through Command Buffer Subroutines

You may be able to improve performance by executing command buffer subroutines.

19.18.1 Overview

Command buffer subroutines represent a method for executing the command buffer using command buffer execution registers. Rather than executing each command in a continuous command buffer, a command buffer address jump is used to execute commands stored outside of the command buffer. This is called “executing a command buffer subroutine” because execution jumps to a particular command buffer address, runs the command buffer there, and then jumps back.

For more details on how to execute a command buffer subroutine, see sections 5.4.20 Adding Subroutine Commands and 5.8.43 Command Buffer Execution Registers in the *DMPGL 2.0 System API Specifications*.

19.18.2 Effect on Subroutine Behavior

Command buffer subroutines have the following advantages.

- The CPU process required to copy commands can be reduced because a command to jump to another command just needs to be stored in the command buffer, rather than a command is copied to a command buffer. Command buffer subroutines are thus effective at executing commands that set a large amount of data, such as lookup table data and shader programs, frequently.
- When commands are made into a subroutine, they can be directly accessed by the GPU without being copied to a command buffer and can thus reduce the overall size of the command buffer.
- When commands are made into a subroutine in VRAM, the GPU is able to access the command buffer more quickly than command buffers in FCRAM. If command buffer memory accesses are a system bottleneck, you can expect an overall improvement in speed.

On the other hand, command buffer subroutines also have the following disadvantages.

- There is memory access overhead associated with switching the command buffer address during a subroutine jump. The GPU may thus access commands more slowly if you turn them into small subroutines that are called frequently.

Any gains or losses in processing speed that you can achieve through subroutines are affected by memory access conflicts and other factors; they are therefore highly dependent on how your application is actually implemented.

19.18.3 Where to Place Command Subroutines

We recommend placing command subroutines in VRAM because it is faster to access command buffers in VRAM than FCRAM.

One disadvantage of executing a command subroutine—as mentioned earlier—is that it has an associated memory access overhead. You can reduce this overhead by placing command subroutines in VRAM.

To store commands in VRAM, you first need to create them in FCRAM and then use a DMA to transfer them to VRAM. Call **`nngxAddVramDmaCommand`** to transfer commands. For more details, see the *DMPGL 2.0 System API Specifications*.

19.18.4 Balancing Command Execution and Access

Performance bottlenecks may move between command access and execution, depending on the content of a command subroutine.

If you make a subroutine out of commands that write to registers in the rasterization and later modules, the amount of processing used to execute the commands will increase because it will take two cycles to process each command. If there are burst commands, the processing for command execution will increase even more relative to the processing for command access. In these cases, command execution becomes the bottleneck and you may not even notice the increased cost of subroutines' memory access.

If you make a subroutine out of commands that write to registers *before* the rasterization module, the amount of processing used to execute the commands will be smaller than the example given above because it will take one cycle to process each command. This makes it easier for command access to be the bottleneck and thus easier for the increased cost of subroutines' memory access to affect overall performance.

For more details on the relationship between the various modules, see Chapter 2 DMPGL 2.0 Pipeline in the *DMPGL 2.0 Specifications*.

19.19 Impact of Command Buffer Address and Size on Load Speed

The command buffer address and size can affect the load speed of the command buffer during execution. "Command buffer address and size" refers to either of the following two items.

- The address and size of the command buffer executed using a render command request
- The address and size of the command buffer executed using command buffer execution registers

The first definition refers to the command buffer address that immediately follows the address where the previous flush occurred (by a call of a command flush function, such as **`nngxFlush3DCommand)`**and the number of bytes from that point to the address of the next flush. The address of the command buffer currently accumulating commands can be obtained by specifying `NN_GX_CMDLIST_CURRENT_BUFADDR` for the *`pname`* parameter of **`nngxGetCmdlistParameteri`**. The second definition refers to the address and size of the command buffer set in the command buffer execution registers, including the addresses and sizes of command buffers added by calls of **`nngxAddJumpCommand`** or **`nngxAddSubroutineCommand`**. (For **`nngxAddSubroutineCommand`**, the addresses and sizes are for command buffers executed as subroutines as well as the addresses and sizes of command buffers executed to return control from the subroutine to the caller.) See the *DMPGL 2.0 System API Specifications* for details on the above functions and the command buffer execution registers.

When the command buffer address is 128-byte aligned, command buffer transfer speed may be improved by setting command buffer sizes to multiples of 256 bytes (256 bytes, 512 bytes, etc.).

If the command buffer address is not 128-byte aligned, speed may improve if the number of bytes from the 128-byte-aligned address immediately before the command buffer address to the end of the command buffer is aligned to a multiple of 256 bytes. For example if the command buffer address is

`0x20000010` and the command buffer size is `0x1f0`, the 128-byte aligned address immediately before the command buffer address is `0x20000000`, just `0x10` bytes before. Accordingly, from that address to the end of the command buffer is `0x1f0+0x10 bytes`, or `0x200` bytes, so the command buffer size can be considered aligned to a multiple of 256 bytes.

Although the above guidelines always apply to command buffer addresses and sizes, the actual improvement you see will vary because of how the GPU is implemented. Other factors will have an effect, such as the location where the command buffer is stored, the command details, and contention with other modules for memory access or other resources.

20 Troubleshooting

This chapter explains how to troubleshoot DMPGL applications.

20.1 Lines Are Unexpectedly Rendered Onscreen

When you render extremely small polygons whose right edges are close to `x=0` in window coordinates, you may unintentionally cause other lines to be rendered as well. This phenomenon occurs when calculation errors cause a polygon's pixels to be generated with negative x-coordinates, which "wrap around" to extremely large values. This results in extremely large x-coordinates and thus stretches the rendered polygons in the positive X-direction.

When this phenomenon occurs, there is a chance that memory outside of the rendering area might be corrupted. The wrapped-around x-coordinate becomes 1023, and the system generates pixels extending from `(0, y)` to `(1023, y)`. Even when the rendering area dimensions are set to a width smaller than 1024 (such as an area of 256×256), the system generates pixels for the full range of x-coordinates from 0 through 1023. The pixel address to access when accessing the framebuffer is calculated from the pixel's xy coordinates and from the rendering area's width, and even when the pixel's x-coordinate is outside the rendering area's width, it is still used as is in calculating the address. Consequently, depending on the y-coordinate, the framebuffer write might extend through the rendering area and into the memory regions beyond it, and overwrite the data in those regions with pixel color data.

This phenomenon only depends on a polygon's window coordinates and will occur consistently for the same window coordinates.

Because this problem manifests itself when geometry is clipped by the view volume or user clip volume and polygons are generated, it can even occur for polygons that were originally large but that protruded into the edge of the screen at window coordinates `x=0`, leaving only an extremely small area within the view volume.

You can adjust x-coordinates in the vertex shader to work around this problem. The vertex shader calculates x clip coordinates that are clipped between -w and w; when `x` is close to -w, the vertex is close to the edge of the screen at window coordinates `x=0`. By taking vertices like this, which are close to the edge of the screen at `x=0`, and moving them onto the edge of the screen, you can prevent lines from being drawn unexpectedly. This shifts vertex coordinates by only one pixel or less and therefore has a negligible effect on the rendered results.

The vertex shader processes the x-value after the projection transformation (written to the output register as the vertex's x-coordinate) as follows.

```
if ( -w < x && x < -w * (1–epsilon) )
     x = -w ;
```

`x` and `w` are the vertex's x- and w-coordinates after the projection transformation. `epsilon` is a variable used to make adjustments and is set to an appropriate value for the scene to be rendered.

The following are sample vertex shader implementations. The first one is a normal implementation that does not handle the phenomenon discussed in this section.

```
// v0    : position attribute
// o0    : output for position
// c0-c3 : modelview matrix
// c4-c7 : projection matrix
m4x4   r0, v0, c0   // modelview transformation
m4x4   o0, r0, c4   // projection transformation and output
```

The next implementation, on the other hand, handles this problem.

```
// v0    : position attribute
// o0    : output for position
// c0-c3 : modelview matrix
// c4-c7 : projection matrix
// c8    : (1 - epsilon, 1, any, any)
m4x4   r0, v0, c0   // modelview transformation
m4x4   r1, r0, c4   // projection transformation
mul    r2.xy, -r1.w, c8.xy    // r2.x = -w * (1-epsilon), r2.y = -w
cmp    2, 4, r1.xx, r2.xy     //
ifc  1, 1, 1                  // if ((x < -w * (1-epsilon)) &&  (x > -w))
    mov r1.x, -r1.w           //     x = -w ;
endif
mov    o0, r1
```

20.2 Rendered Textures Are Distorted

Texture images are sometimes rendered unevenly when the texture filter mode is `GL_NEAREST`. For more details, see section 7.3 Precautions When the Filter Mode Is `GL_NEAREST`.

20.3 Boundaries Between Texture Mipmap Levels Appear Conspicuous

The boundary lines between texture mipmap levels are sometimes pronounced when the texture filter mode is `GL_XXX_MIPMAP_LINEAR`. For more details, see section 7.4 Precautions When the Filter Mode Is `GL_XXX_MIPMAP_LINEAR`.

20.4 Polygons Are Not Clipped Properly

Polygons are sometimes not clipped properly when they are rendered close to the far plane of the view volume. For details, see section 17.3.3 Clipping-Related Precautions.

20.5 Rendering Results for Polygons with the Same Vertex Coordinate Do Not Perfectly Match

Even when polygons with the exact same vertex coordinates are rendered, there are cases when interpolated attribute values for each fragment, such as depth values or texture coordinates, do not match perfectly. This phenomenon occurs due to differences in the interpolation calculation for fragment values when the order of vertices input into the PICA rasterization module is different. In other words, the fragment attribute values for polygons rendered with a vertex index order of 0, 1, and 2 and polygons rendered with an order of 1, 2, and 0 may not match perfectly.

When the vertex input order is exactly the same, this phenomenon does not occur. However, in the `GL_TRIANGLES` mode of **`glDrawElements`**, depending on the relationship with the vertex indices of the immediately prior polygon, the vertex input order may be changed internally. In this case, to render polygons that have perfectly matched fragment values multiple times, be sure to use the same vertex indices including the relationship with the prior and subsequent polygons when rendering.

This phenomenon is likely to happen for all fragment attributes. In other words, when the vertex input order is different, the depth value, texture mapping result, primary color, lighting result, and other values may not match exactly for polygon fragments that have the same vertex attribute values, such as vertex coordinates.

20.6 Block-Shaped Noise Is Rendered on Certain Pixels

4x4 pixels of block-shaped noise sometimes appear in specific positions of the rendered results. This phenomenon is caused by pixels in specific positions that are mistakenly determined to hit the framebuffer cache even though they actually did not. This section describes the problem in detail.

20.6.1 Overview

Pixel data in the framebuffer is processed in 4x4 pixel blocks, and the address of a 4x4 pixel block is called a *block address*. The framebuffer cache manages pixel data in blocks; when the cache's tag information is cleared, tags are initialized to the default value of `0x3fff` (the cache's tag information is cleared by a command that writes 1 (one) to register `0x110`). If pixels with a block address of `0x3fff` are rendered immediately after the cache tags have been cleared, the pixels are mistakenly determined to have hit the cache because they have the same block address as the default tag value. As a result, the wrong color is applied to the pixels.

Block addresses are assigned consecutively, beginning at 0, in 16-pixel units from the starting address of the framebuffer (the color buffer, depth buffer, and stencil buffer). Because addresses are assigned to data in a PICA rendering format, the relationship between block addresses and pixel locations in a rendered image differs between the 8x8 block format and the 32x32 block format.

Because this phenomenon only occurs for pixels that have a block address of `0x3fff`, it does not occur when the total number of blocks in the framebuffer is less than or equal to `0x3fff`—in other words, when there are no more than `0x3fff`×16 (262,128) pixels in the framebuffer.

This phenomenon also does not occur when there are no read accesses to the color buffer, depth buffer, or stencil buffer. To determine which conditions result in a read access to each buffer, see section 19.14 Configuring Access Control to the Framebuffer.

20.6.2 Relationship Between Pixels and Block Addresses

Block addresses begin at 0 and are assigned in ascending order, 16 pixels at a time, from the starting address of the color buffer and depth/stencil buffer when they are in a rendering format.

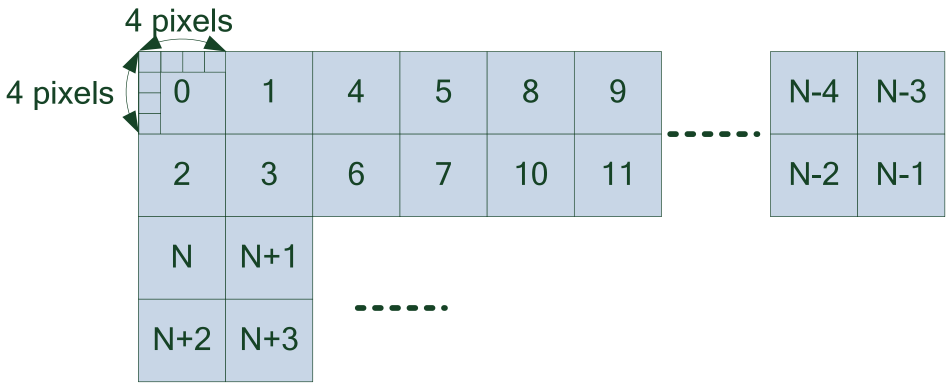
In the PICA rendering format, the pixel data for the upper-left corner of a rendered image is placed at the first buffer address. Note that this differs from the origin for **`glViewport`** and that the horizontal direction on the rendered image corresponds to the shorter side of the LCDs on the CTR system.

Because the 8x8 block format and the 32x32 block format use different addressing schemes, when an image is rendered in different block formats its pixels will correspond to different block addresses in the cache.

20.6.2.1 8x8 Block Format

The following figure shows how block addresses correspond to pixels in an image rendered with the 8x8 block format. Addresses are assigned from the upper-left corner of the image.

Figure 20-1 Block Addresses for Each Pixel in the 8x8 Block Format

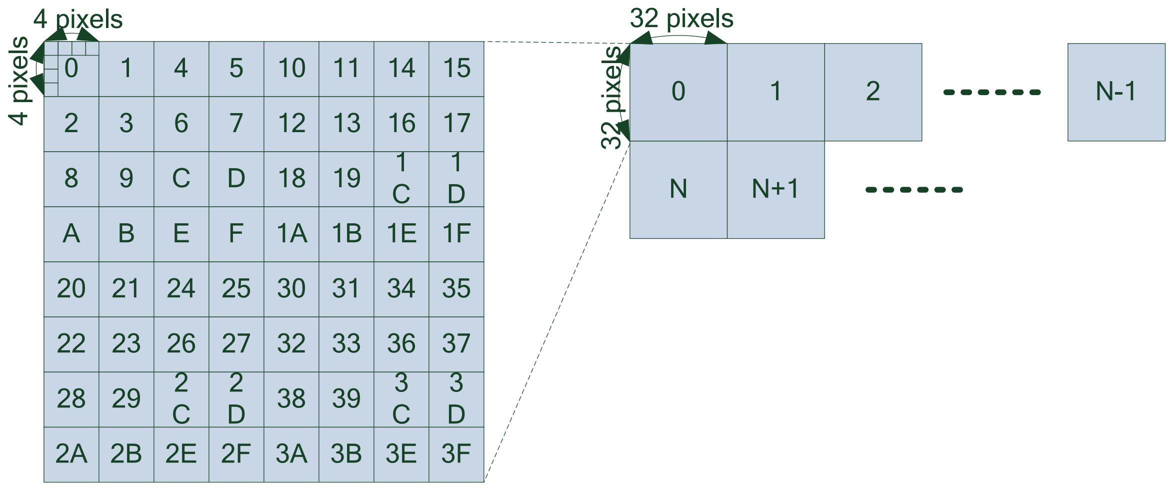


Given that the framebuffer is `W` pixels wide, the number `N` in the figure is equal to `W÷4×2`. The 4x4 pixel block at the upper-left corner of the rendered image corresponds to block address 0, the next 4x4 pixel block immediately to the right corresponds to block address 1, the 4x4 pixel block immediately below block address 0 corresponds to block address 3, and the 4x4 pixel block immediately below block address 1 corresponds to block address 4. Block addresses continue to increase to the right 8x8 pixels at a time until they reach the edge of the image, at which point they continue on the next row at the left edge of the image.

20.6.2.2 32x32 Block Format

The following figure shows how block addresses correspond to pixels in an image rendered with the 32x32 block format. Addresses are assigned from the upper-left corner of the image.

Figure 20-2 Block Addresses for Each Pixel in the 32x32 Block Format



The figure shows 32x32 pixel regions called *metablocks*. The block addresses of the pixels in a metablock are shown on the left (in hexadecimal). The metablock addresses for the entire image are shown on the right.

The 32x32 pixel region at the upper-left corner of the rendered image is located at metablock address 0. The next 32x32 pixel region immediately to the right is located at metablock address 1. Metablock addresses continue to increase to the right until they reach the edge of the image, at which point they continue with the 32x32 pixel region on the next line at the left edge of the image.

As shown in the figure on the left, the block addresses of pixels in each metablock follow a zigzag pattern starting with the 4x4 pixels at the upper-left corner. The block address of a pixel as seen from the entire rendered image is `0x40` times that pixel's metablock address plus its block address within the metablock.

20.6.3 Workaround #1

The first workaround is to not use a framebuffer that is larger than necessary. This problem does not occur when the product of the framebuffer's width and height is less than or equal to 262,128 pixels.

If your framebuffer is the same size as the LCDs on the CTR system, this problem does not occur because each screen has fewer than 262,128 pixels: the 240x400 LCD has a total of 96,000 pixels and the 240x320 LCD has a total of 76,800 pixels.

The **`glRenderbufferStorage`** function determines the framebuffer size. If you allocate a framebuffer that is unnecessarily large but actually use only 240x400 pixels of it, you can restrict the allocated framebuffer region to the minimum required size to work around this problem. (Bits [10:0] and [21:12] of the `0x6e` and `0x11e` registers determine the framebuffer size.)

20.6.4 Workaround #2

The second workaround is to adjust the size of the framebuffer so that pixels at the problematic block address `0x3fff` are outside of the rendering region.

For example, if you set the framebuffer size equal to 480x800 when you want to render to a 480x800 region, the pixel coordinates (124, 548) correspond to the upper-left corner of the 4x4 pixel square at block address `0x3fff`, assuming that the rendered image is in the 8x8 block format and its upper-left corner has coordinates (0, 0). If you were to widen the framebuffer by 32 pixels to be 512x800 pixels in size, pixel coordinates (508, 508) would correspond to the upper-left corner of the 4x4 pixel square at block address `0x3fff`. You can then avoid the problematic pixels by configuring the viewport to only use the leftmost 480x800 pixels of the 512x800 pixels in the framebuffer.

This workaround has the drawback of wasting VRAM by requiring you to allocate a framebuffer that is larger than necessary, but it is a simple method that only involves adjusting the framebuffer size.

For more information on which pixels correspond to block address `0x3fff`, see section 20.6.2 Relationship Between Pixels and Block Addresses.

20.6.5 Workaround #3

The third workaround is to avoid rendering to the pixels at block address `0x3fff` immediately after cache tags have been cleared. By rendering several pixels whose block address is not `0x3fff`, you change the content of the cache tags and prevent cache hits from being mistakenly registered.

20.6.5.1 Details

To work around this problem, you must render four pixels at a particular block address. If settings cause both the color buffer and the depth/stencil buffer to be read, these four pixels must each have a different block address for which the lower *three* bits are all `1` (`0x7`). If settings cause only the color buffer or the depth/stencil buffer to be read, these four pixels must each have a different block address for which the lower *four* bits are all `1` (`0xf`).

Consider the case in which pixels at the following block addresses are rendered immediately after tags are cleared: `0`, `1`, `0x0f`, `2`, `0x1f`, `3`, `0x0f`, `0x2f`, `0x3f`, and so on. Pixels at block addresses like `0` and `1` are not counted because the lower four bits of the block address are not `0xf`. Pixels at block address `0x0f` show up twice, but they are only counted once because they have the same block address. The problem is resolved once pixels at block addresses `0x0f`, `0x1f`, `0x2f`, and `0x3f` have been input. If a pixel at block address `0x3fff` is rendered before the pixel at block address `0x3f`, the problem occurs.

To work around this problem, render a dummy polygon of pixels that meet these conditions immediately after cache tags are cleared.

Commands to clear the cache are accumulated in the following instances:

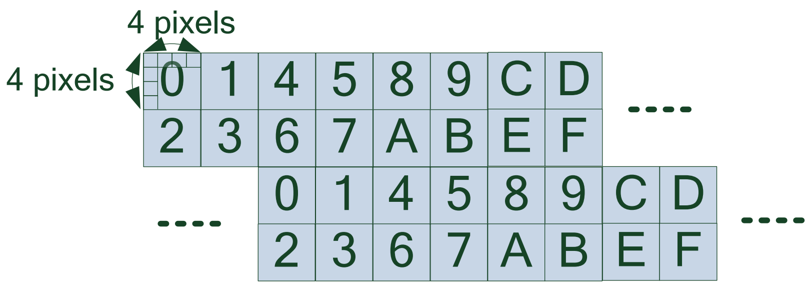
- **`glFlush`**, **`glFinish`**, and **`glClear`** are called
- The current framebuffer has changed
- The framebuffer's access control settings have changed (see section 19.14 Configuring Access Control to the Framebuffer)
- A function that flushes 3D commands, such as **`nngxFlush3DCommand`**, has been called

For more details, see section 5.8.41 Clearing the Framebuffer Cache in the *DMPGL 2.0 System API Specifications*.

20.6.5.2 Block Addresses of Rendered Dummy Pixels

This workaround requires you to render dummy pixels at block addresses whose lower three or four bits are all 1. If you look at only the lower four bits of the block address in the 8x8 block format, you will see the same 32x8 pixel pattern repeated horizontally. However, depending on the framebuffer width, this pattern may be shifted horizontally by 8 pixels for every 8 pixels vertically.

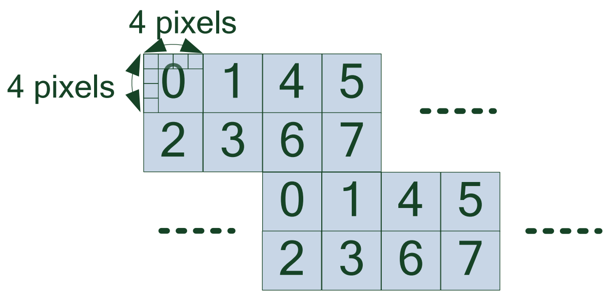
Figure 20-3 Lower 4 Bits of the Block Address for Pixels in the 8x8 Block Format



The hexadecimal numbers in Figure 20-3 represent the lower four bits of the block address.

If you look at only the lower three bits of the block address in the 8x8 block format, you will see the same 16x8 pixel pattern repeated horizontally. However, depending on the framebuffer width, this pattern may be shifted horizontally by 8 pixels for every 8 pixels vertically.

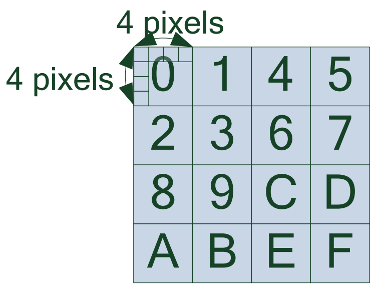
Figure 20-4 Lower 3 Bits of the Block Address for Pixels in the 8x8 Block Format



The hexadecimal numbers in Figure 20-4 represent the lower three bits of the block address.

If you look at only the lower four bits of the block address in the 32x32 block format, you will see the same 16x16 pixel pattern repeated horizontally and vertically.

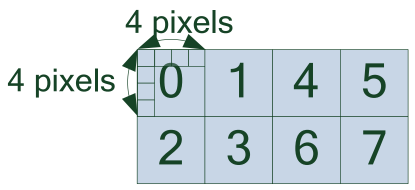
Figure 20-5 Lower Four Bits of the Block Address for Pixels in the 32x32 Block Format



The hexadecimal numbers in Figure 20-5 represent the lower four bits of the block address.

If you look at only the lower three bits of the block address in the 32x32 block format, you will see the same 16x8 pixel pattern repeated horizontally and vertically.

Figure 20-6 Lower Three Bits of the Block Address for Pixels in the 32x32 Block Format



The hexadecimal numbers in Figure 20-6 represent the lower three bits of the block address.

20.6.5.3 Concrete Dummy Rendering Example (Lower 4 Bits Must Be 0xf in the 8x8 Block Format)

Here we will show concrete examples of dummy polygons that can be used as valid workarounds when the 8x8 block format is used and the lower four bits of the block address must be `0xf`. These examples render a single rectangle as a dummy polygon. The figures show the lower four bits of the block address.

Figure 20-7 Rendering 4 Pixels at Block Addresses Whose Lower 4 Bits Are 0xf in the 8x8 Block Format: First Example

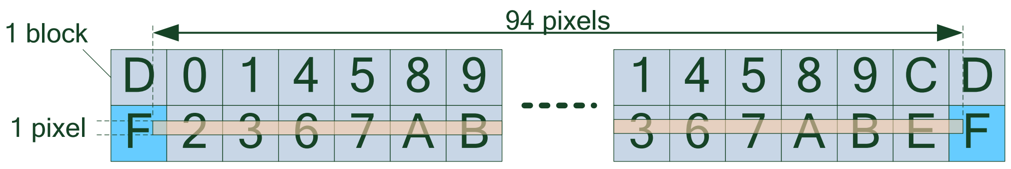


Figure 20-7 shows a rectangular dummy polygon of 94x1 pixels. This example minimizes the area to be rendered. The polygon must be placed so that the pixels at both ends have block addresses whose lower four bits are `0xf`. For example, assuming that the upper-left corner of the rendered image has the pixel coordinates (0, 0), you could use a polygon that covers pixel coordinates (31, 5)–(124, 5).

Figure 20-8 Rendering 4 Pixels at Block Addresses Whose Lower 4 Bits Are 0xf in the 8x8 Block Format: Second Example

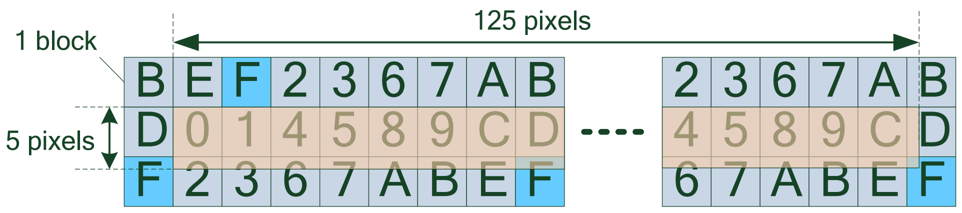


Figure 20-8 shows a rectangular dummy polygon of 125x5 pixels. A polygon of this size can be placed anywhere and will always include four pixels at block addresses whose lower four bits are `0xf`.

© 2009–2012 Nintendo
CONFIDENTIAL

Figure 20-9 Rendering 4 Pixels at Block Addresses Whose Lower 4 Bits Are 0xf in the 8x8 Block Format: Third Example

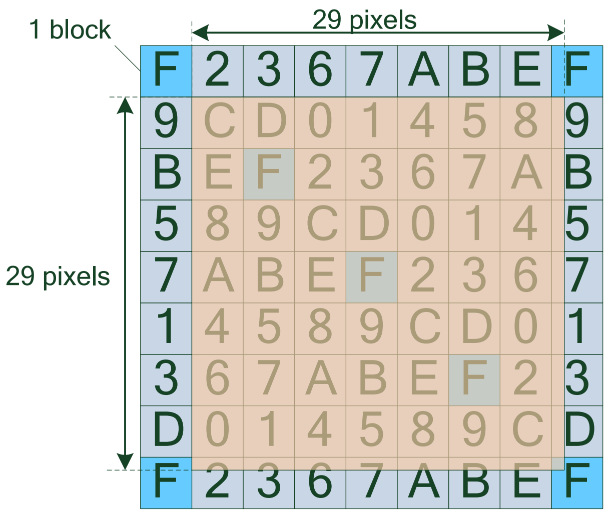


Figure 20-9 shows a rectangular dummy polygon of 29x29 pixels. A polygon of this size can be placed anywhere and will always include four pixels at block addresses whose lower four bits are `0xf`.

20.6.5.4 Concrete Dummy Rendering Example (Lower 3 Bits Must Be 0x7 in the 8x8 Block Format)

Here we will show concrete examples of dummy polygons that can be used as valid workarounds when the 8x8 block format is used and the lower three bits of the block address must be `0x7`. These examples render a single rectangle as a dummy polygon. The figures show the lower three bits of the block address.

Figure 20-10 Rendering 4 Pixels at Block Addresses Whose Lower 3 Bits Are 0x7 in the 8x8 Block Format: First Example

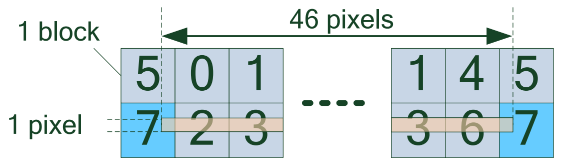


Figure 20-10 shows a rectangular dummy polygon of 46x1 pixels. This example minimizes the area to be rendered. The polygon must be placed so that the pixels at both ends have block addresses whose lower three bits are `0x7`. For example, assuming that the upper-left corner of the rendered image has the pixel coordinates (0, 0), you could use a polygon that covers pixel coordinates (15, 5)–(60, 5).

Figure 20-11 Rendering 4 Pixels at Block Addresses Whose Lower 3 Bits Are 0x7 in the 8x8 Block Format: Second Example

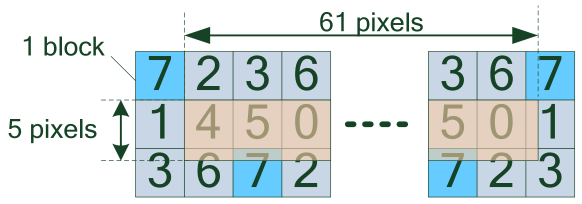


Figure 20-11 shows a rectangular dummy polygon of 61x5 pixels. A polygon of this size can be placed anywhere and will always include four pixels at block addresses whose lower three bits are `0x7`.

Figure 20-12 Rendering 4 Pixels at Block Addresses Whose Lower 3 Bits Are 0x7 in the 8x8 Block Format: Third Example

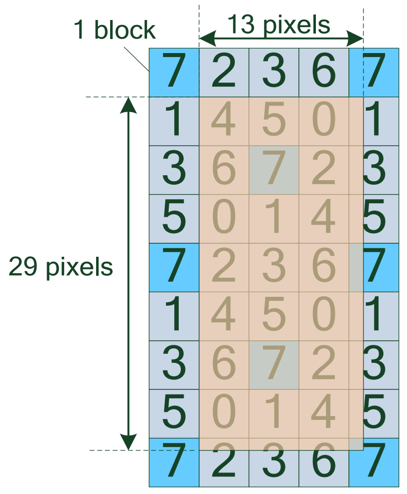
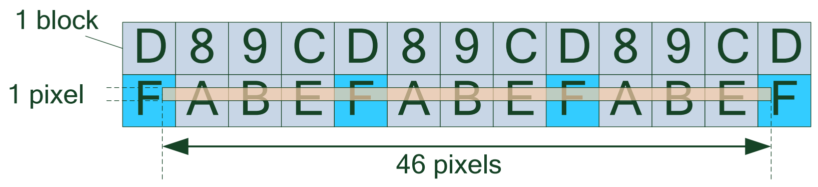


Figure 20-12 shows a rectangular dummy polygon of 13x29 pixels. A polygon of this size can be placed anywhere and will always include four pixels at block addresses whose lower three bits are `0x7`.

20.6.5.5 Concrete Dummy Rendering Example (Lower 4 Bits Must Be 0xf in the 32x32 Block Format)

Here we will show concrete examples of dummy polygons that can be used as valid workarounds when the 32x32 block format is used and the lower four bits of the block address must be `0xf`. These examples render a single rectangle as a dummy polygon. The figures show the lower four bits of the block address.

Figure 20-13 Rendering 4 Pixels at Block Addresses Whose Lower 4 Bits Are 0xf in the 32x32 Block Format: First Example



© 2009–2012 Nintendo
CONFIDENTIAL

Figure 20-13 shows a rectangular dummy polygon of 46x1 pixels. This example minimizes the area to be rendered. The polygon must be placed so that the pixels at both ends have block addresses whose lower four bits are `0xf`. For example, assuming that the upper-left corner of the rendered image has the pixel coordinates (0, 0), you could use a polygon that covers pixel coordinates (15, 13)–(60, 13).

Figure 20-14 Rendering 4 Pixels at Block Addresses Whose Lower 4 Bits Are 0xf in the 32x32 Block Format: Second Example

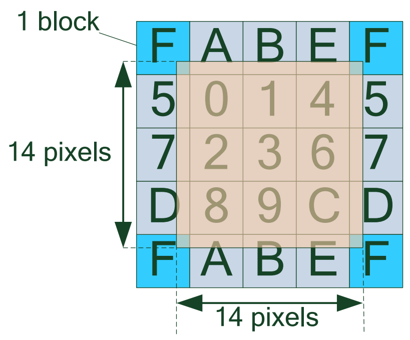


Figure 20-14 shows a rectangular dummy polygon of 14x14 pixels. The rectangle must be placed so that the pixels in each of the four corners have block addresses whose lower four bits are `0xf`. For example, assuming that the upper-left corner of the rendered image has the pixel coordinates (0, 0), you could use a polygon that covers pixel coordinates (15, 15), (28, 15), (15, 28), and (28, 28).

Figure 20-15 Rendering 4 Pixels at Block Addresses Whose Lower 4 Bits Are 0xf in the 32x32 Block Format: Third Example

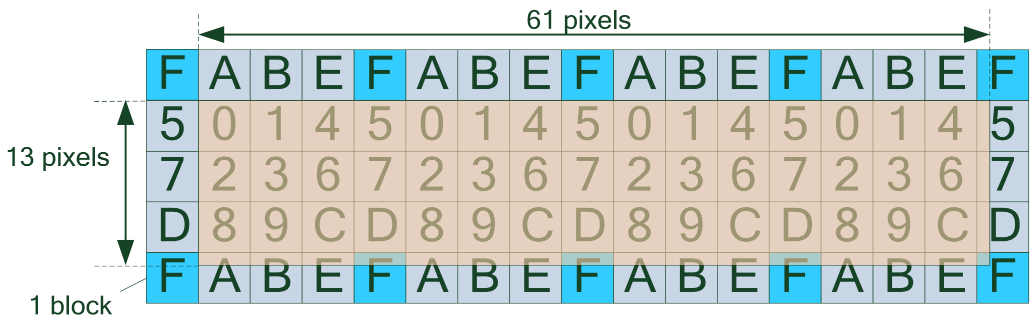

61 pixels
13 pixels
1 block
F A B E F A B E F A B E F A B E F
5 0 1 4 5 0 1 4 5 0 1 4 5 0 1 4 5
7 2 3 6 7 2 3 6 7 2 3 6 7 2 3 6 7
D 8 9 C D 8 9 C D 8 9 C D 8 9 C D
F A B E F A B E F A B E F A B E F

Figure 20-15 shows a rectangular dummy polygon of 61x13 pixels. A polygon of this size can be placed anywhere and will always include four pixels at block addresses whose lower four bits are `0xf`.

Figure 20-16 Rendering 4 Pixels at Block Addresses Whose Lower 4 Bits Are 0xf in the 32x32 Block Format: Fourth Example

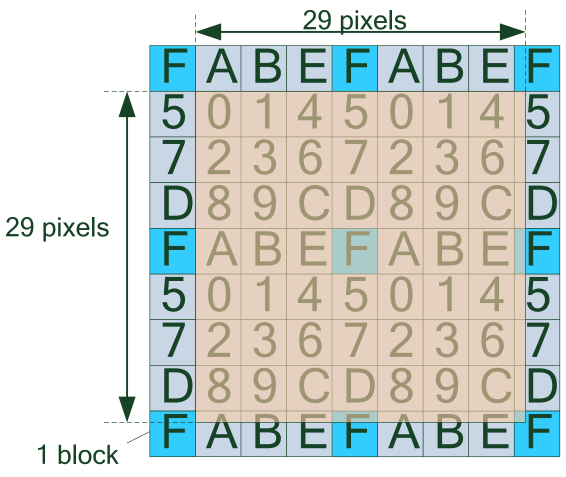


Figure 20-16 shows a rectangular dummy polygon of 29x29 pixels. A polygon with these dimensions can be placed anywhere and will always include four pixels at block addresses whose lower four bits are `0xf`.

20.6.5.6 Concrete Dummy Rendering Example (Lower 3 Bits Must Be 0x7 in the 32x32 Block Format)

Here we will show concrete examples of dummy polygons that can be used as valid workarounds when the 32x32 block format is used and the lower three bits of the block address must be `0x7`. These examples render a single rectangle as a dummy polygon. The figures show the lower three bits of the block address.

Figure 20-17 Rendering 4 Pixels at Block Addresses Whose Lower 3 Bits Are 0x7 in the 32x32 Block Format: First Example

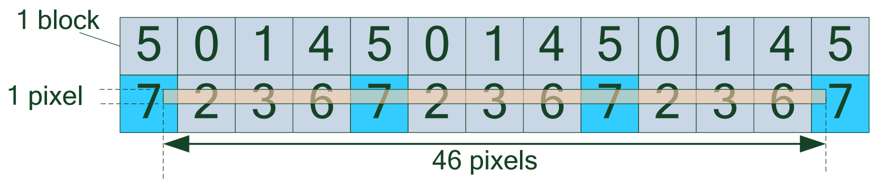


Figure 20-17 shows a rectangular dummy polygon of 46x1 pixels. This example minimizes the area to be rendered. The polygon must be placed so that the pixels at both ends have block addresses whose lower three bits are `0x7`. For example, assuming that the upper-left corner of the rendered image has the pixel coordinates (0, 0), you could use a polygon that covers pixel coordinates (15, 5)–(60, 5).

Figure 20-18 Rendering 4 Pixels at Block Addresses Whose Lower 3 Bits Are 0x7 in the 32x32 Block Format: Second Example

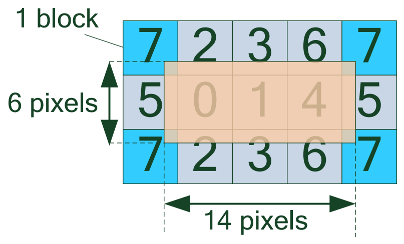


Figure 20-18 shows a rectangular dummy polygon of 14x6 pixels. The rectangle must be placed so that the pixels in each of the four corners have block addresses whose lower three bits are `0x7`. For example, assuming that the upper-left corner of the rendered image has the pixel coordinates (0, 0), you could use a polygon that covers pixel coordinates (15, 7), (28, 7), (15, 12), and (28, 12).

Figure 20-19 Rendering 4 Pixels at Block Addresses Whose Lower 3 Bits Are 0x7 in the 32x32 Block Format: Third Example

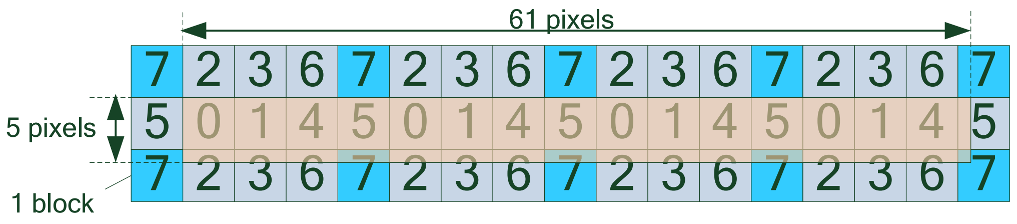


Figure 20-19 shows a rectangular dummy polygon of 61x5 pixels. A polygon of this size can be placed anywhere and will always include four pixels at block addresses whose lower three bits are `0x7`.

Figure 20-20 Rendering 4 Pixels at Block Addresses Whose Lower 3 Bits Are 0x7 in the 32x32 Block Format: Fourth Example

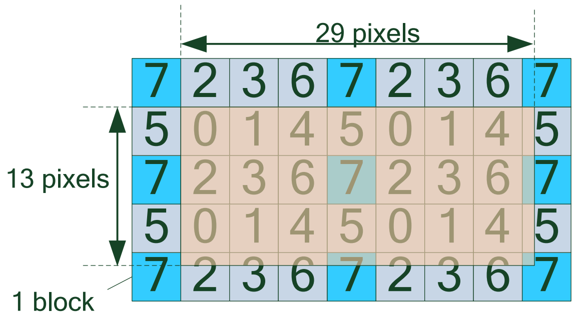


Figure 20-20 shows a rectangular dummy polygon of 29x13 pixels. A polygon of this size can be placed anywhere and will always include four pixels at block addresses whose lower three bits are `0x7`.

20.6.5.7 Notes for Dummy Rendering

There are a number of issues to be aware of when you render dummy polygons as a workaround.

For example, the following are valid dummy pixels.

- Pixels that fail the depth test, stencil test, or alpha test

Note: If you use settings that cause dummy pixels to always fail these tests—for example, by specifying `GL_NEVER` for the depth test function—be sure to restore the original depth test function when you resume normal rendering. Note that the cache flush command (a command that writes to register `0x111`) would be required at this time. For more details, see section 5.8.41 Clearing the Framebuffer Cache in the *DMPGL 2.0 System API Specifications*.

- Pixels that do not affect the color buffer when they are rendered because of alpha blend settings

The following are *not* valid dummy pixels.

- Pixels that are clipped by the view volume or user-defined clipping planes
- Pixels that are dropped by the scissor test
- Pixels that are dropped by the early depth test

20.7 Cannot Render Correctly When Viewport Size Exceeds 1023 x 1016

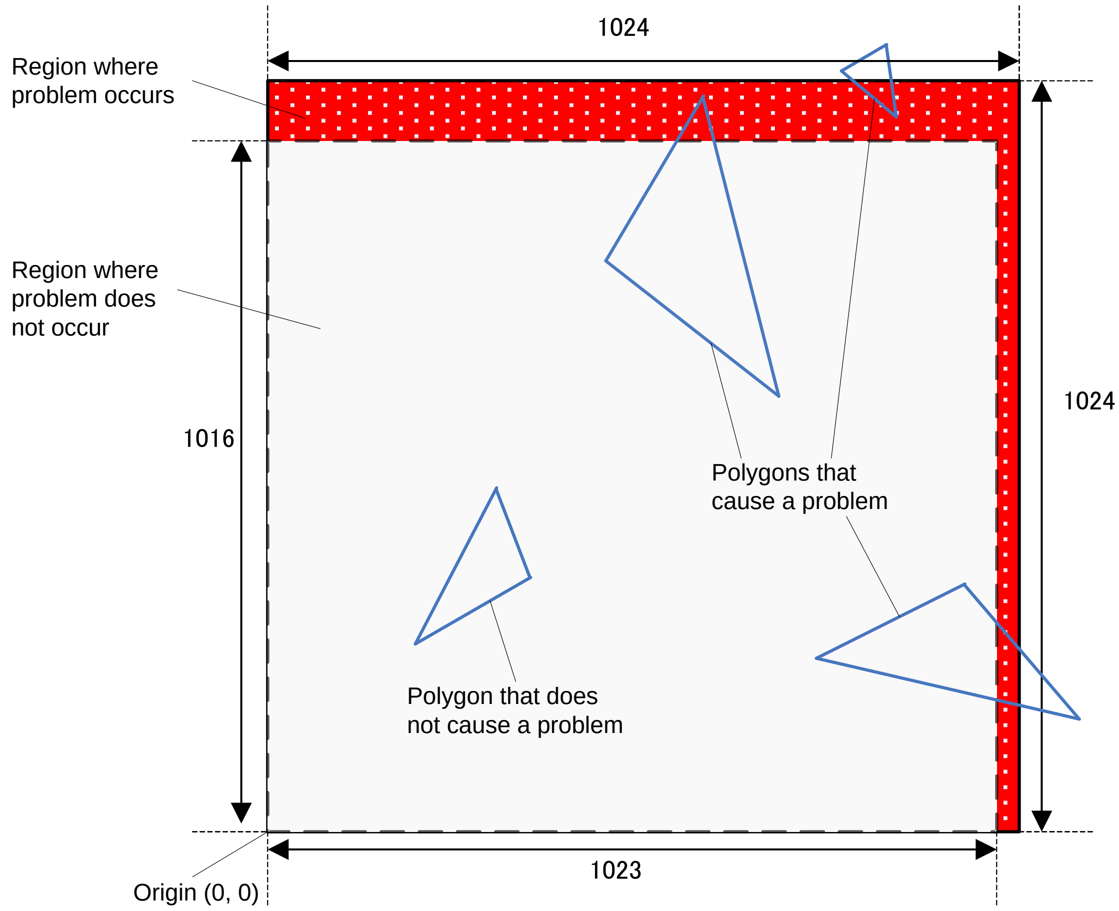
Note the following restrictions for the `glViewport` width and height settings (x and y respectively).

- If a polygon pixel's x-coordinate is located at a point such that the required display width would be greater than 1023, then the entire polygon is not rendered.
- If a polygon pixel's y-coordinate is located at a point such that the required display height would be greater than 1016, then the GPU hangs.

These restrictions, which are a result of the hardware design, are shown in the following figure.

© 2009–2012 Nintendo
CONFIDENTIAL

Figure 20-21 Regions Where Problems Occur with Polygon Rendering


1024
Region where problem occurs
Region where problem does not occur
1016
1024
Polygons that cause a problem
Polygon that does not cause a problem
Origin (0, 0)
1023

The example in Figure 20-21 uses a frame buffer size of 1024 x 1024. The region shown in red is where the problem occurs when the viewport offset is set to (0, 0). If a polygon crosses into the 1-pixel region on the right of the rendering screen, then that polygon is not rendered. If the polygon crosses into the top 8-pixel region of the rendering screen, then the GPU hangs. To render in the region shown in red, the viewport offset must be adjusted. For example, if the viewport offset is set to (`1, 8`) (x is set to `1` and y is set to `8` with **`glViewport`**), then the polygon not extending beyond the red region can be rendered with no problem.

By setting the viewport width to 1023 or less and the height to 1016 or less, the pixels associated with coordinates that would cause problems are not rendered. When rendering a 1024 x 1024 image that will be used as a texture, therefore, adjust the coordinates so that the effective texture region is 1023 x 1016.

If you need to render the entire 1024 x 1024 area, divide the image into regions that do not exceed the restrictions and change the viewport offset accordingly to render each portion. For example, use the following `glViewport` coordinates for problem-free rendering of 4 equal areas: (0, 0, 512, 512), (512, 0, 512, 512), (0, 512, 512, 512), and (512, 512, 512, 512).

Note also that scissoring cannot be used to prevent the pixels in a problem area from being drawn.

20.8 Early Depth Tests Are Not Performed Correctly When Using Viewport Offsets

When early depth testing is enabled, it is not performed correctly if the viewport offset — **`glViewport`** `(x, y)` — is not (`0, 0`).

A hardware malfunction fails to apply the viewport offset to the coordinates read from the early depth buffer. Updates to the early depth buffer incorrectly compare the depth value for pixels that have the viewport offset applied with pixels that do not have the offset applied. Pixels that should pass the early depth test fail and are therefore not rendered. For a false fail, an 8x8 pixel area is erroneously discarded.

The following two methods circumvent this problem.

- When setting the viewport offset `(x, y)` to something other than (`0, 0`), disable early depth testing. If early depth testing is toggled from enabled to disabled when rendering a single scene, the rendering result is correct because normal depth testing results in the appropriate pixels being dropped.
- Set the viewport offset `(x, y)` to (`0, 0`), use a modelview transformation to shift the rendering region, and cut the unnecessary region with scissoring.

20.9 GPU Hangs when Using Multitextures

The GPU sometimes hangs when using multiple textures at the same time.

The GPU will sometimes hang in situations that meet both of the following conditions.

- The application is using multiple textures.
- There is a sizable performance disparity between the different texture units.

Procedural textures are not a trigger for this issue. This issue does not occur when using, for example, a single regular texture with procedural textures.

You can avoid this issue using by using any of the following methods.

- Only use one texture.
- Set the `GL_TEXTURE_MIN_FILTER` texture parameter to `GL_XXX_MIPMAP_LINEAR` for all textures used. (This uses the trilinear filter.)

You can mitigate this issue by using any of the following methods.

- Set the `GL_TEXTURE_MIN_FILTER` texture parameter to `GL_XXX_MIPMAP_LINEAR` for some of the textures used. (This uses the trilinear filter.) You can completely avoid this issue by using the trilinear filter for all textures used, but using this filter for just some of your textures should mitigate the symptoms.
- Place all the textures used at the same time in the same VRAM.
- Reduce the number of textures used.

This issue may or may not occur depending on the timing of events. Accordingly, changing the following texture settings can avoid hangs, depending on the kind of content being processed. On the other hand, in some cases changing these settings might instead make the problem re-occur more frequently.

- Change the texture size.
- Change the texture format.
- Change the texture filter mode.
- Change where textures are stored. (Try switching between VRAM-A, VRAM-B, and FCRAM.)

Should your GPU hang, check the GPU busy state to find out whether the hang is due to this issue. Call **`nngxGetCmdlistParameteri`** and pass a value of `NN_GX_CMDLIST_HW_STATE` as the *`pname`* parameter. Bit [2] in the value returned by *`param`* indicates the texture unit busy state. A value of `1` for this bit indicates that the hang was very likely due to this issue. See section 3.3.14 Getting the Parameters of Command List Objects and 3.4 `NN_GX_CMDLIST_HW_STATE` in the *DMPGL 2.0 System API Specifications* for details.

However, note that the above method does not definitively indicate that this problem caused the GPU to hang. If 4-bit textures are not stored in their permitted locations, the GPU can hang in the same busy state. See section 5.1.3 Texture Image Specifications in the *DMPGL 2.0 Specifications* for details.

20.10 Calculating `GL_INTERPOLATE` for the Texture Combiner

The `GL_INTERPOLATE` calculation for the texture combiner is `src0 * src2 + src1 * (1 – src2 )`. If `src2` is 1 or 0, the result is the value of either `src0` or `src1`. However, because of how the hardware is implemented, in cases where `src2` is 0 and `src0` < `src1`, although one would expect `src1` to be output as is, the value outputs with a brightness that is 1 less than `src1`. To work around this problem, you must calculate the combiner in two stages using a combination of `GL_MODULATE` and `GL_MULT_ADD_DMP`. The number of fragments where this problem occurs can also be reduced by changing the `src2` operand to `GL_ONE_MINUS_*` and swapping `src0` and `src1`.

Revision History

Version	Revision Date	Category	Description
3.5	2012/11/01	Added	• Added section 20.10 Calculating `GL_INTERPOLATE` for the Texture Combiner.
		Changed	• Revised the clamp mode `GL_CLAMP_TO_ZERO_DMP` interval for procedural textures. • Revised the description in section 19.7 Improving Vertex Cache Performance.
3.4	2012/02/03	Changed	• Revised the copyright notice.
3.3	2011/11/30	Added	• Added the new section19.6 Vertex Array Address Alignment.
3.2	2011/10/27	Changed	• Changed 19.5 Data Types of Vertex Attributes to Data Types, Data Size of Vertex Attributes and the Relationship with Transfer Speed and revised content. • Revised copyright notation.
3.1	2011/10/05	Added	• 20.9 GPU Hangs when Using Multitextures
3.0	2011/09/16	Added	• 19.18 Impact of Command Buffer Address and Size
2.9	2011/08/30	Added	• 19.17 Optimization Through Command Buffer Subroutines
2.8	2011/07/20	Added	• 18.8 Depth Buffer Update Flag
		Changed	• General Fixed typos. • 14.3.3 Texture Coordinates and Texture Transformation Matrices During Shadow Texture Access Added texture transformation matrix for orthographic projection. • 18.6 Comparison Functions for Early Depth Tests Revised explanations.
2.7	2011/06/06	Changed	• General Removed `dmp_Texture[0].shadowZScale` from specifications. • Table 11-4: Non-Boolean Lighting Environment Parameters and Their Default Values Fixed mistake in `dmp_LightEnv.lutScale` setting range. • 19.4 Implementing Vertex Shaders Added explanatory text. • 19.8 Configuring Textures Added explanatory text. • 20.1 Lines Are Unexpectedly Rendered Onscreen Added explanatory text.
2.6	2011/03/17	Changed	• Fixed typos. • 13.5 Format and Type of Normal Maps Added supplementary information about normal maps. • 14.3.3 Texture Coordinates and Texture Transformation Matrices During Shadow Texture Access

© 2009–2012 Nintendo
CONFIDENTIAL

Version	Revision Date	Category	Description
			Revised description of r coordinate in shadow maps.
2.5	2011/02/07	Added	• 20.6 Block-Shaped Noise Is Rendered on Certain Pixels • 20.7 Cannot Render Correctly When Viewport Size Exceeds 1023 x 1016 • 20.8 Early Depth Tests Are Not Performed Correctly When Using Viewport Offsets
		Changed	• Changed the default value of `dmp_LightEnv.lutEnableRefl` from `GL_TRUE` to `GL_FALSE`. • 14.3 Shadow Texture Depth Replaced r with t in equations 14.7 and 14.9. • 10.8 Configuring the Noise Fixed typos related to noise settings for procedural textures and added a supplementary explanation.
2.4	2010/11/09	Changed	• Made corrections in sections 14.3.3, 14.4, and 14.4.2 regarding the rendering color of the shadow accumulation pass.
2.3	2010/11/05	Added	• 20.5 Rendering Results for Polygons with the Same Vertex Coordinate Do Not Perfectly Match
2.2	2010/10/08	Changed	• 10.6 Configuring the Color Lookup Tables Corrected errors in sample code. • 14.6 Other Related Parameters Corrected descriptions of `dmp_FragOperation.penumbraScale` and `dmp_FragOperation.penumbraBias` parameters, which were reversed.
2.1	2010/09/30	Changed	• 7.4 Precautions When the Filter Mode Is `GL_XXX_MIPMAP_LINEAR` Added supplementary information.
2.0	2010/09/14	Changed	• Added a description of texture coordinate conversion during shadow texture lookup and fixed typos.
1.9	2010/08/20	Added	• 20 Troubleshooting
1.8	2010/07/30	Added	• 19.16 Load Sizes for Each Data Type.
		Changed	• Revised information about shadows. • Added notes about trilinear filtering mode. • Added supplemental information about the Schlick anisotropic model and replaced the related figure. • Revised the description about generation of the z-component in tangent mapping. • Replaced some of the figures to prevent corruption that occurred upon PDF generation. • Added supplemental information about texture caches. • Fixed typos.
1.7	2010/07/07	Added	• 7.3 Precautions When the Filter Mode is `GL_NEAREST`
		Changed	• Revised the input values to the fragment lighting lookup table.

Version	Revision Date	Category	Description
			• Added information about performance changes related to vertex buffers.
1.6	2010/06/04	Added	• 17.3.3 Clipping-Related Precautions • 19.14 CPU Performance
		Changed	• 19.9 Configuring Fragment Lighting Described differences in efficiency caused by varying layer configurations.
1.5	2010/05/11	Changed	• Revised the number of cycles for `LIGHT_ENV_LAYER_CONFIG7_DMP`. • Removed `LIGHT_ENV_LAYER_CONFIG8_DMP`, `LIGHT_ENV_LAYER_CONFIG9_DMP`, and `LIGHT_ENV_LAYER_CONFIG10_DMP` from the specifications. • 19.7 Configuring Textures Added information on filter modes.
1.4	2010/04/23	Added	• 19.6 Vertex Shader Output Attributes
		Changed	• 19.7 Configuring Textures Added information on texture placement. • Fixed typos. • Revised descriptions related to `LIGHT_ENV_SP_DMP`.
1.3	2010/04/02	Changed	• Added text to section 19.7 Texture Cache Sizes.
1.2	2010/03/19	—	• Initial English version. (This document was branched from version 1.8 of the *DMPGL 2.0 Programming Guide*, which was included in CTR-SDK 0.8.3.)

DMP and PICA are registered trademarks of Digital Media Professionals Inc.
All other company and product names in this document are the trademarks or registered trademarks of their respective companies.

© 2009–2012 Digital Media Professionals Inc. All rights reserved.

This documentation is the confidential and proprietary property of Digital Media Professionals Inc. The possession or use of this documentation and its content requires a written license from Digital Media Professionals Inc.

© 2009–2012 Nintendo

The contents of this document cannot be duplicated, copied, reprinted, transferred, distributed, or loaned in whole or in part without the prior approval of Nintendo.