Graphics Engine User's Manual

Table of Contents

1. Overview

The Graphics Engine enables three-dimensional geometry calculations, rendering, surface processing and vertex deformation to be performed in an integrated fashion. Acting as bus master of the host bus, it can directly read drawing packets in main memory. The Graphics Engine also has a built-in VRAM that can hold buffers such as the frame buffer and depth buffer.
The following are the main features of the Graphics Engine.

- High-speed geometry processing and rendering
- Surface processing of Bezier and spline curves
- Built-in VRAM
- Vertex blending
- Environment mapping
- Fogging
- Texture mapping
- Alpha blending
- Antialiasing

2. System Configuration

Figure 2-1 shows the configuration of the Graphics Engine in the PSP™ system.
The frame buffer, depth buffer, texture buffer and all data for the display list, model, etc. can be stored in local memory. Any data can be stored in host memory other than the frame buffer and the depth buffer. In addition, the Graphics Engine has an internal, local AHB which enables it to access local memory without having to go through the external system AHB.

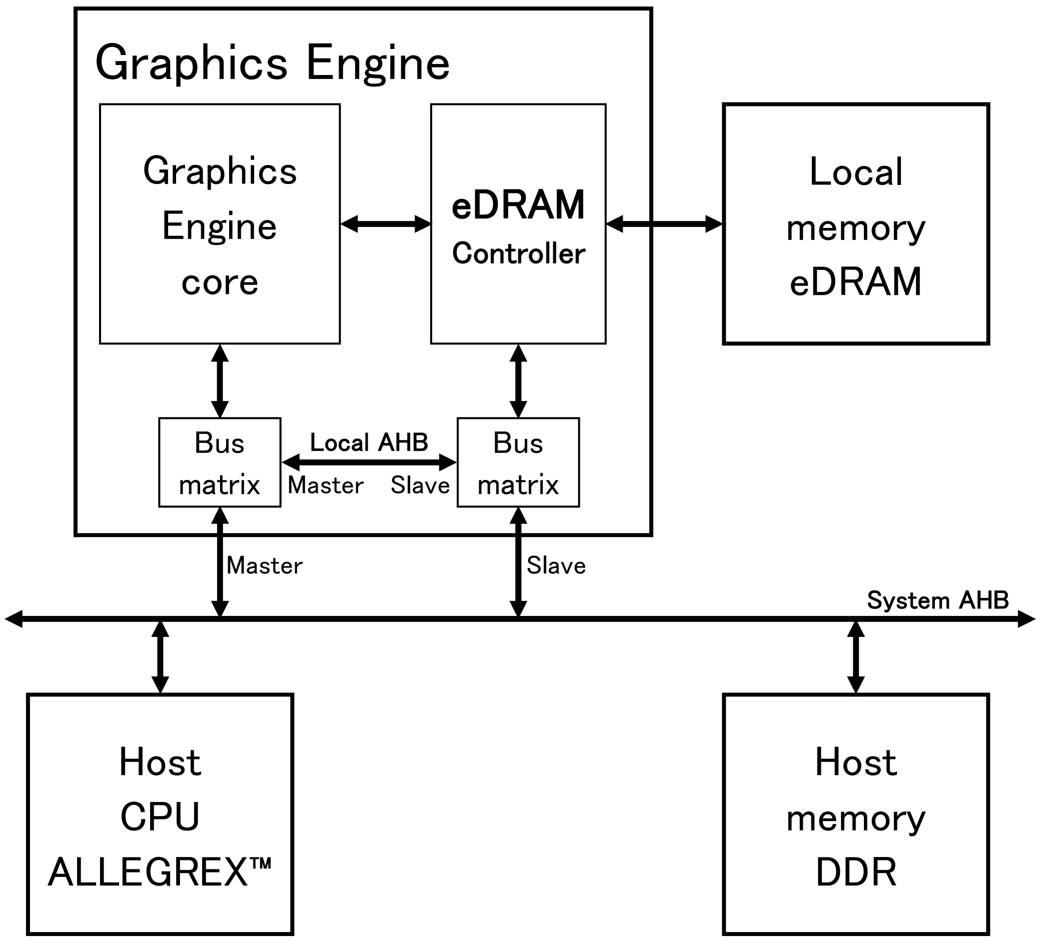


Figure 2-1 System Configuration

3. Coordinate Systems

The Graphics Engine has six kinds of coordinate systems for drawing primitives. Coordinate values defined in the Model Coordinate System will be eventually transformed to values in the Drawing Coordinate System as shown below. When Through Mode is active, which can be set using the Vertex Type command (CMD_VTYPE), it is also possible to draw primitives directly in the Drawing Coordinate System.

Model Coordinate System
↓
World Coordinate System
↓
View Coordinate System
↓
Clip Coordinate System
↓
Screen Coordinate System
↓
Drawing Coordinate System

The Graphics Engine also has two kinds of coordinate systems for defining textures. Coordinate values defined in the Texture Coordinate System are transformed to values in the Texel Coordinate System as shown below. When Through Mode is active, which is a coordinate-defined mode as in the coordinate system for primitives, it is possible to execute texture mapping directly in the Texel Coordinate System.

Texture Coordinate System
↓
Texel Coordinate System

3.1. Model Coordinate System

The Model Coordinate System is used for defining three-dimensional model data. Each of the coordinate values (Xm, Ym, Zm) can be represented with an 8-bit signed fixed-point number, a 16-bit signed fixed-point number, or a 32-bit single-precision floating-point number. The Model Coordinate System is transformed to the World Coordinate System by applying the World Matrix in Equation 3-1.

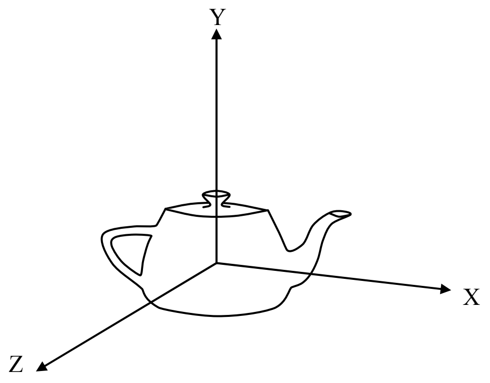


Figure 3-1 Model Coordinate System

The World Matrix is represented by a 3x3 matrix and a translation vector. The World Matrix can be set using the World Matrix Number command (CMD_WORLDN) and the World Matrix Data command (CMD_WORLDD).

$$\begin{bmatrix} Xw \\ Yw \\ Zw \end{bmatrix} = \begin{bmatrix} Mw00 & Mw03 & Mw06 \\ Mw01 & Mw04 & Mw07 \\ Mw02 & Mw05 & Mw08 \end{bmatrix} \begin{bmatrix} Xm \\ Ym \\ Zm \end{bmatrix} + \begin{bmatrix} Mw09 \\ Mw10 \\ Mw11 \end{bmatrix}$$

Equation 3-1 Model to World Coordinate Transformation

To perform lighting properly, the matrix of Mw00-Mw08 should consist of only rotation, translation, uniform magnification (reduction) and inversion.

3.2. World Coordinate System

The World Coordinate System defines all the positions in the three-dimensional world and is used for setting a view point and light source. The coordinate values (Xw, Yw, Zw) are transformed to the View Coordinate System by applying the View Matrix in Equation 3-2.

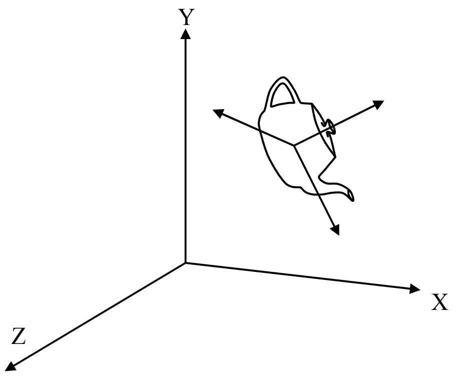


Figure 3-2 World Coordinate System

Like the World Matrix, the View Matrix is also represented with a 3x3 matrix and translation vector. The View Matrix can be set with the View Matrix Number command (CMD_VIEWN) and the View Matrix Data command (CMD_VIEWD).

$$\begin{bmatrix} Xv \\ Yv \\ Zv \end{bmatrix} = \begin{bmatrix} Mv00 & Mv03 & Mv06 \\ Mv01 & Mv04 & Mv07 \\ Mv02 & Mv05 & Mv08 \end{bmatrix} \begin{bmatrix} Xw \\ Yw \\ Zw \end{bmatrix} + \begin{bmatrix} Mv09 \\ Mv10 \\ Mv11 \end{bmatrix}$$

Equation 3-2 World to View Coordinate Transformation

To perform lighting properly, the matrix of Mv00-Mv08 should consist of only rotation, translation, uniform magnification (reduction) and inversion.

3.3. View Coordinate System

The View Coordinate System is the coordinate system of three-dimensional space as seen from a view point. The fog parameters for calculating fog coefficients are set in this coordinate system. The view point position to calculate a specular light source is also defined internally in this coordinate system. The coordinate values (Xv, Yv, Zv) are transformed to the Clip Coordinate System by applying the Perspective Transformation Matrix in Equation 3-3.

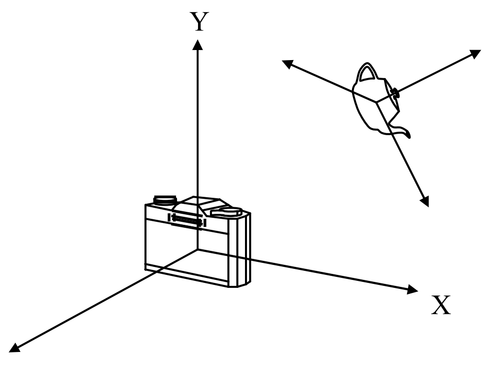


Figure 3-3 View Coordinate System

The Perspective Transformation Matrix is represented by a 4x4 matrix. It can be set with the Perspective Transformation Matrix Number command (CMD_PROJN) and the Perspective Transformation Matrix Data command (CMD_PROJD).

$$\begin{bmatrix} Xc \\ Yc \\ Zc \\ Wc \end{bmatrix} = \begin{bmatrix} Mp00 & Mp04 & Mp08 & Mp12 \\ Mp01 & Mp05 & Mp09 & Mp13 \\ Mp02 & Mp06 & Mp10 & Mp14 \\ Mp03 & Mp07 & Mp11 & Mp15 \end{bmatrix} \begin{bmatrix} Xv \\ Yv \\ Zv \\ 1.0 \end{bmatrix}$$

Equation 3-3 View to Clip Coordinate Transformation

3.4. Clip Coordinate System

The Clip Coordinate System is a homogeneous coordinate system. As shown below, each of its coordinate values (Xc, Yc, Zc) lies inside of the view volume and are in the range of -Wc ~ +Wc. This coordinate system is used for clipping primitives that are outside of the view volume.

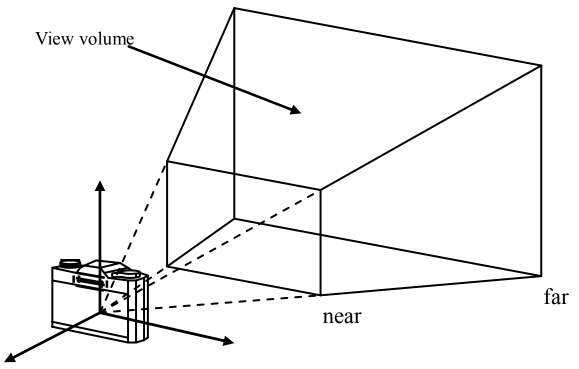


Figure 3-4 Clip Coordinate System

Primitives are transformed to the Screen Coordinate System by using the viewport transformation of Equation 3-4. The parameters required for the viewport transformation are set by the Viewport Command.

$$Xs = Sx \times Xc \div Wc + Tx$$
$$Ys = Sy \times Yc \div Wc + Ty$$
$$Zs = Sz \times Zc \div Wc + Tz$$

Equation 3-4 Viewport Transformation

3.5. Screen Coordinate System

The Screen Coordinate System is a coordinate system as shown in Figure 3-5. Each of (Xs, Ys) is a 16-bit unsigned fixed-point number in the range 0.0 ~ 4095.9375 with the low-order four bits representing a decimal fraction. Zs is a 16-bit unsigned integer and is stored in the depth buffer.

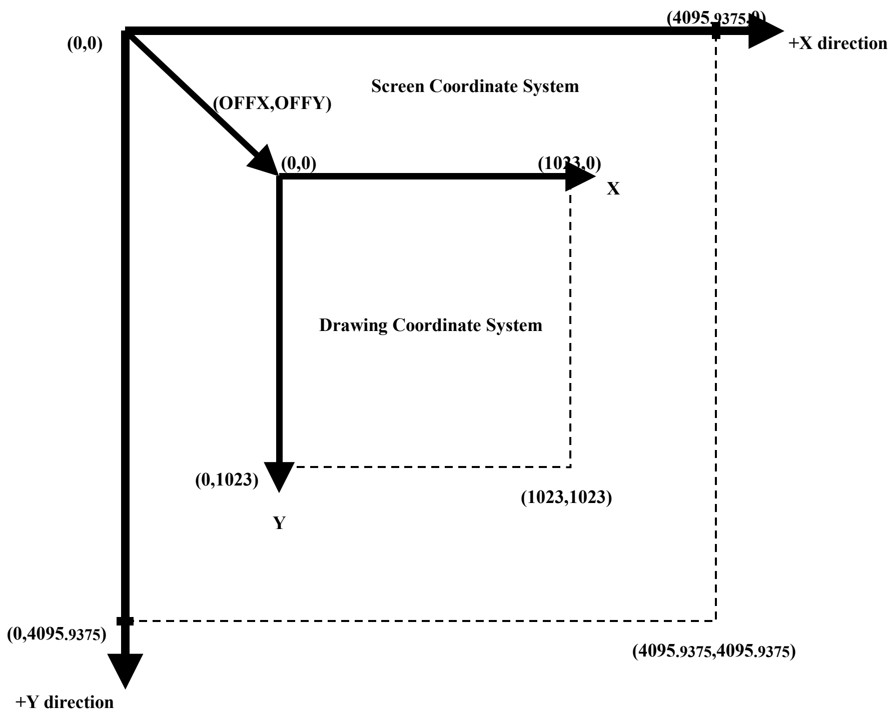


Figure 3-5 Screen and Drawing Coordinate Systems

The Screen Coordinate System is transformed to the Drawing Coordinate System by subtracting the offset values (OFFX, OFFY) as shown in Equation 3-5. The offset values can be set using the Screen Offset commands (CMD_OFFSETX, CMD_OFFSETY).

$$Xd = Xs - OFFX$$
$$Yd = Ys - OFFY$$
$$Zd = Zs$$

Equation 3-5 Screen to Drawing Coordinate Transformation

3.6. Drawing Coordinate System

The Drawing Coordinate System is a coordinate system as shown in Figure 3-5. Each of (Xd, Yd) is a 10-bit unsigned integer in the range 0 to 1023. This coordinate system is used for setting up regions for scissoring and for performing transfers between host and local memories.

The pixel center is positioned at 0.5 as shown in Figure 3-6.

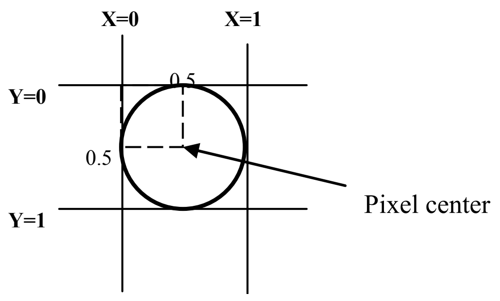


Figure 3-6 Pixel Center

3.7. Texture Coordinate System

The Texture Coordinate System is a coordinate system in which the width and height of a two-dimensional texture are normalized to 1.0 each, as shown in Figure 3-7. Each of the coordinate values (S, T) is specified by a 32-bit single-precision floating-point number. The Texture Coordinate System is transformed to the Texel Coordinate System by calculating (S, T) with texture width (TW) and height (TH) as shown in Equation 3-6.

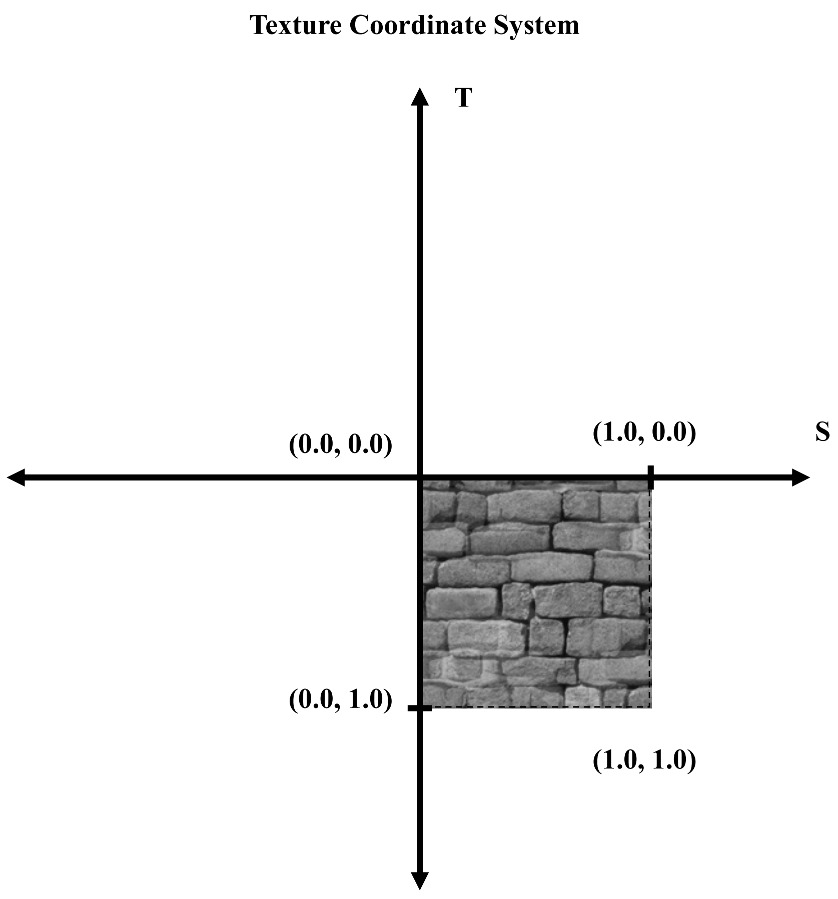


Figure 3-7 Texture Coordinate System

$$U = S \times TW$$
$$V = T \times TH$$

Equation 3-6 Texture to Texel Coordinate Transformation

3.8. Texel Coordinate System

The Texel Coordinate System is a coordinate system for specifying the position of a two-dimensional texture image. Each of the coordinate values (U, V) is a 9-bit unsigned integer in the range 0 to 511. The texel center is positioned at 0.5 as shown in Figure 3-8.

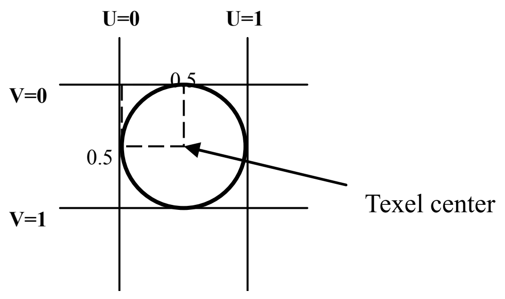


Figure 3-8 Texel Center

4. Basic Primitives

The Graphics Engine can draw four types of basic primitives, which are Points, Lines, Triangles, and Rectangles. These primitives are specified in the Model Coordinate System, the aforementioned coordinate transformation is executed, and the primitives are ultimately transformed to the Drawing Coordinate System.

4.1. Points

A Point is a point in three-dimensional space as shown in Figure 4-1 and is drawn to the nearest pixel in drawing coordinates as shown in Figure 4-2. You can execute texture mapping and fogging when drawing Points, but you cannot specify the pixel size.

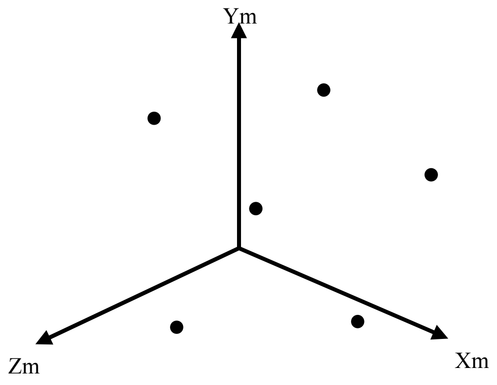


Figure 4-1 Points

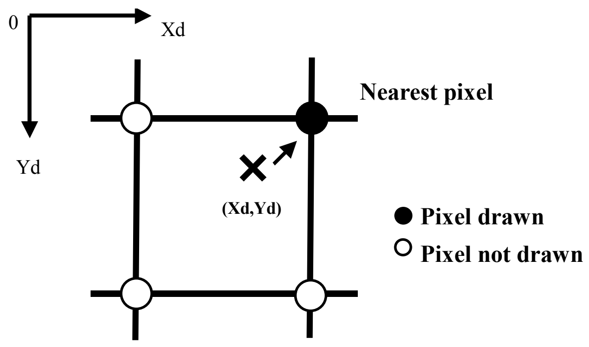


Figure 4-2 Drawing Rule for Points

If you draw with two or more vertices, you can also render point sequences. Points are drawn with the Draw Primitive command (CMD_PRIM, PRIM=POINTS). A point sequence can also be drawn if two or more vertices are set when drawing. The draw command can also be used with an index vertex, which is set with the Index Type bit (IT) of the Vertex Type command (CMD_VTYPE).

4.2. Lines

A Line is a line segment in three-dimensional space. Figure 4-3 shows a sample figure created with Lines. As shown in Figure 4-4, pixels are drawn where a line segment which connects two specified vertices passes through an area of diamonds. You can perform shading, texture mapping and fogging when drawing Lines.

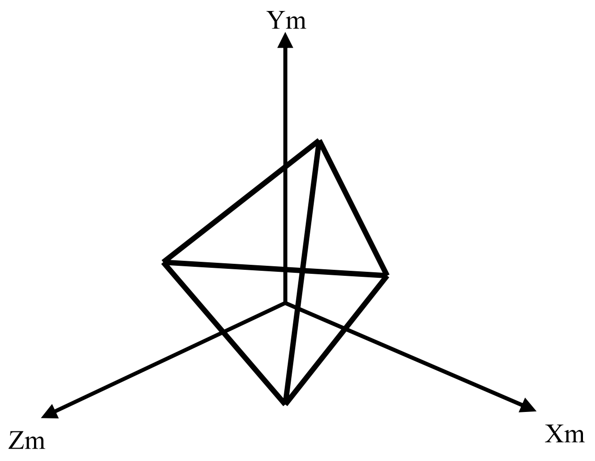


Figure 4-3 Lines

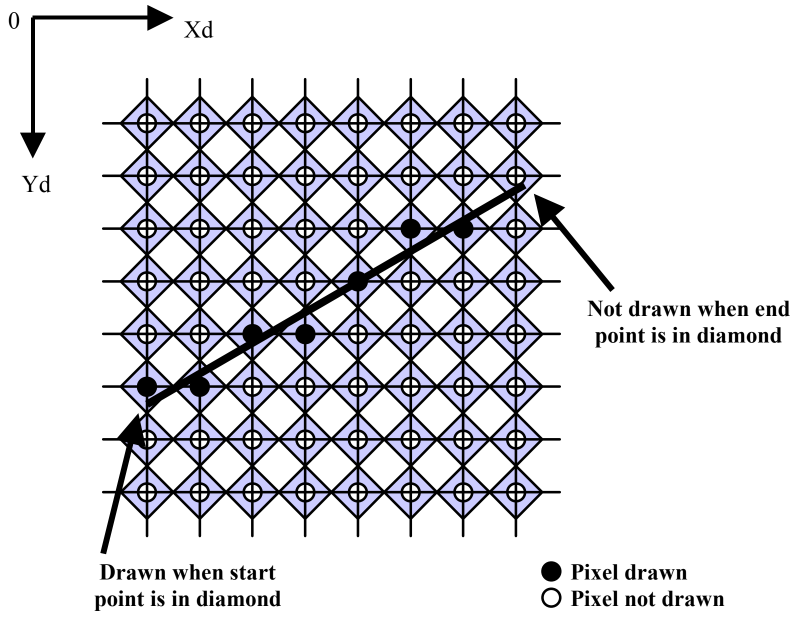

0
Xd
Yd
Not drawn when end point is in diamond
Drawn when start point is in diamond
● Pixel drawn
○ Pixel not drawn

Figure 4-4 Drawing Rule for Lines

Lines can be either independent line segments (LINES) as shown in Figure 4-5, or sequences of connected line segments (LINE_STRIP) as shown in Figure 4-6. Lines can be drawn using the Draw Primitive command (CMD_PRIM, PRIM=LINES or PRIM=LINE_STRIP). When drawing independent line segments it is possible to draw several sequences of line segments at a time by making the number of vertices a multiple of two. These can also be drawn using an index array, which is set using the Index Type bit (IT) of the Vertex Type command (CMD_VTYPE).

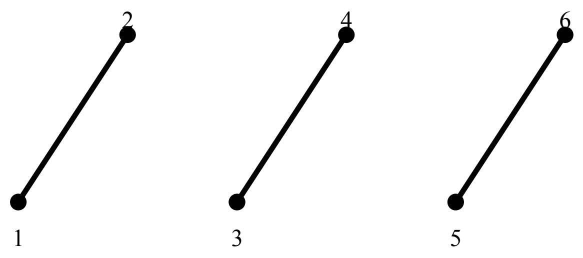


Figure 4-5 LINES

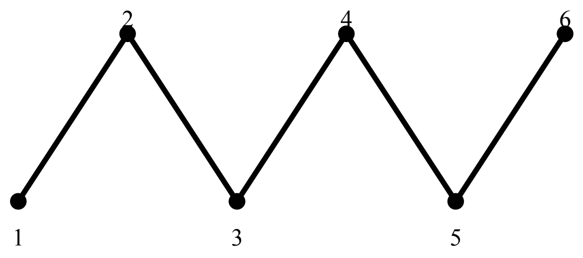


Figure 4-6 LINE_STRIP

4.3. Triangles

A Triangle is a triangle in three-dimensional space. Figure 4-7 shows a sample figure created with Triangles. As shown in Figure 4-8 and Figure 4-9, pixels are drawn inside the three sides that are constructed from three specified vertices. You can perform shading, texture mapping and fogging when drawing Triangles.

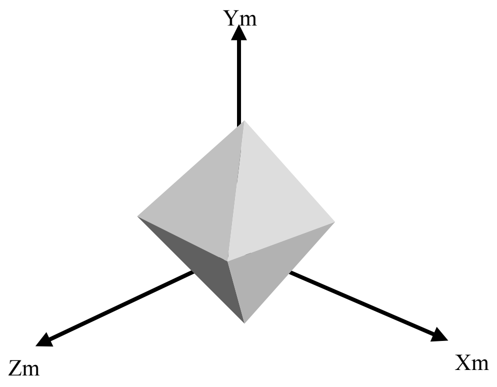


Figure 4-7 Triangle

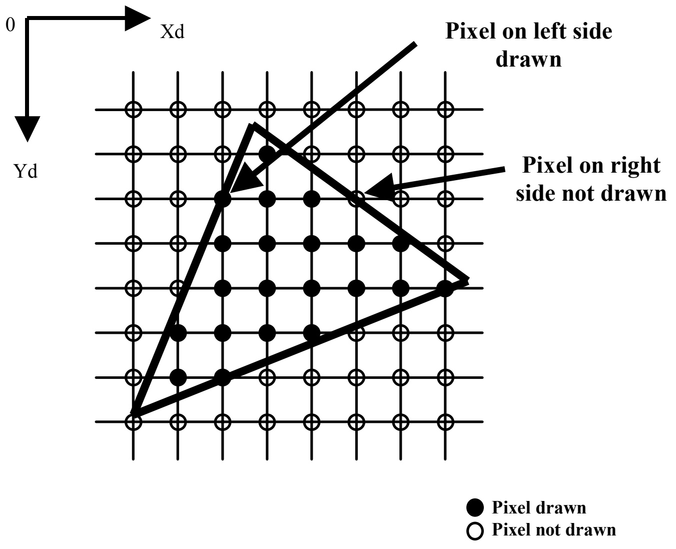


Figure 4-8 Drawing Rule 1 for Triangles

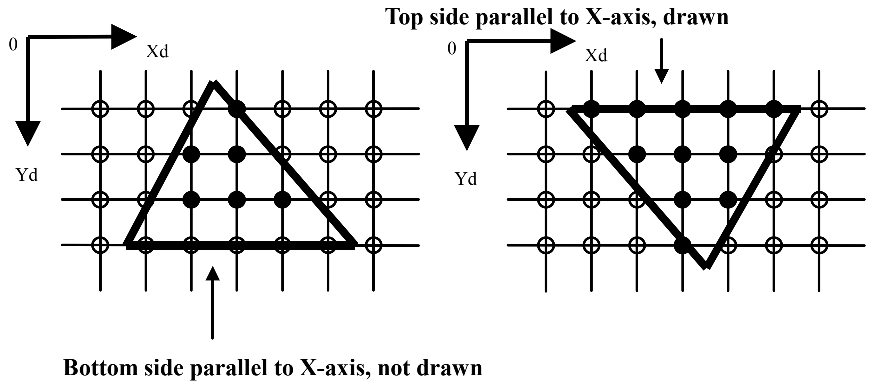

0
Xd
Yd
Top side parallel to X-axis, drawn
Bottom side parallel to X-axis, not drawn

Figure 4-9 Drawing Rule 2 for Triangles

As shown in Figure 4-10, a Triangle can be an independent triangle (TRIANGLES), a linked strip of Triangles (TRIANGLE_STRIP) as shown in Figure 4-11 and a linked fan of Triangles (TRIANGLE_FAN) as shown in Figure 4-12. Triangles can be drawn using the Draw Primitive command (CMD_PRIM, PRIM=TRIANGLES, PRIM=TRIANGLE_STRIP or PRIM=TRIANGLE_FAN).
When drawing independent Triangles it is possible to draw several Triangles at a time by making the number of vertices a multiple of three. These can also be drawn using an index array, which is set using the Index Type bit (IT) of the Vertex Type command (CMD_VTYPE).

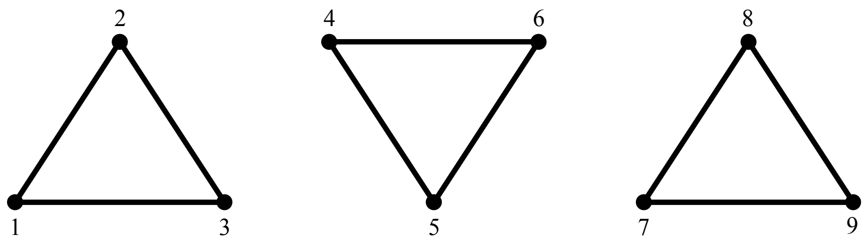


Figure 4-10 TRIANGLES

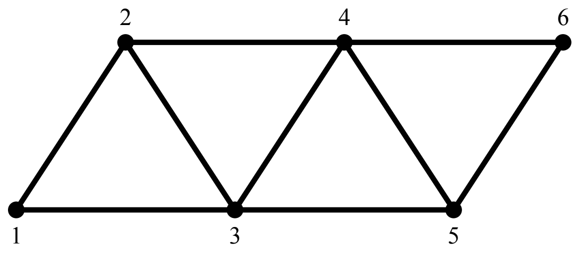


Figure 4-11 TRIANGLE_STRIP

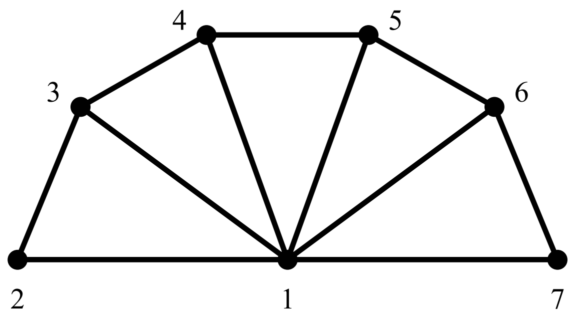


Figure 4-12 TRIANGLE_FAN

4.4. Rectangles

A Rectangle is rendered by drawing a rectangle whose two specified vertices are at opposite angles as shown in Figure 4-13. Since rectangles here are calculated in the Drawing Coordinate System, you cannot draw rectangles that are sloped or rotated.
You can perform texture mapping and fogging when drawing Rectangles. During drawing, the color and depth values for the second vertex are used for all the pixels of the Rectangle. Rectangles can be drawn using the Draw Primitive command (CMD_PRIM, PRIM=RECTANGLES). In addition, it is possible to rotate and map textures depending on

the positions of the first and second vertices as shown in Figure 4-14. When drawing Rectangles, it is possible to draw several at a time by making the number of vertices a multiple of two. These can also be drawn using an index array, which is set by using the Index Type bit (IT) of the Vertex Type command (CMD_VTYPE).

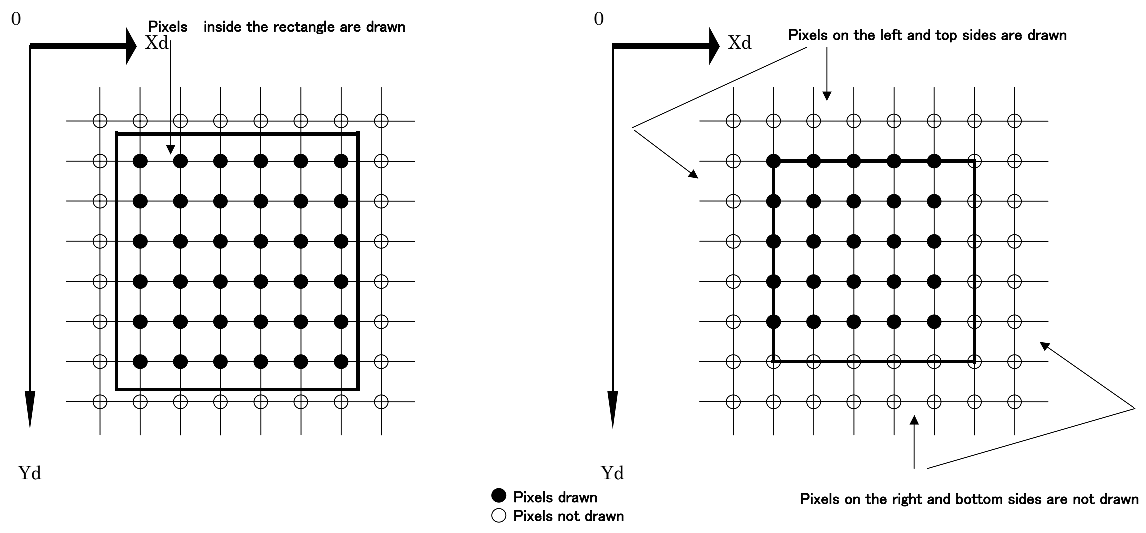

0
Xd
Yd
Pixels inside the rectangle are drawn
0
Xd
Yd
Pixels on the left and top sides are drawn
Pixels on the right and bottom sides are not drawn
● Pixels drawn
○ Pixels not drawn

Figure 4-13 Drawing Rule for Rectangles

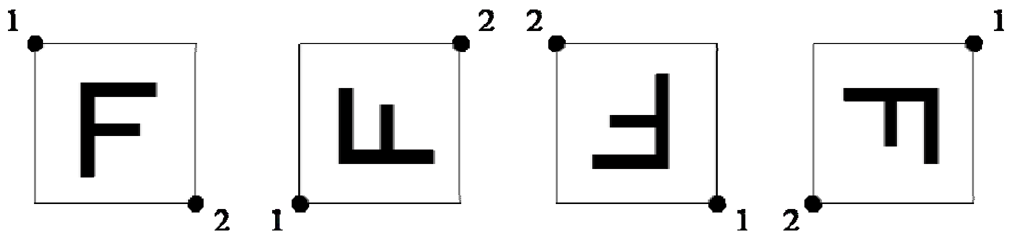


Figure 4-14 Texture Mapping Direction

5. Curved Surface Primitives

The Graphics Engine can draw two types of curved surface primitives, namely Bezier surfaces and spline surfaces. These primitives consist of small surfaces known as "patches" which are the smallest units common to both Bezier surfaces and spline surfaces. A patch is defined with 16 control points that are placed in order from the upper left corner as shown in Figure 5-1.

The curved surface primitives are specified in the Model Coordinate System and are divided into basic primitives, which are Points, Lines or Triangles. The process after division is the same as for basic primitives.

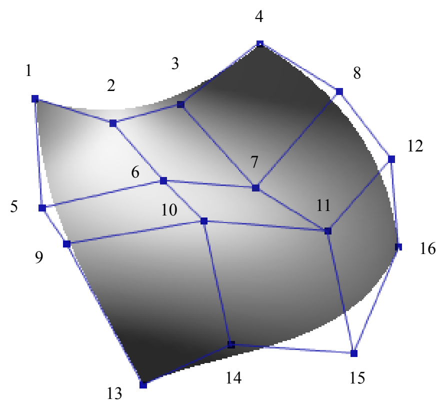


Figure 5-1 Patch

Figure 5-2 shows a sample figure created with a Bezier surface.

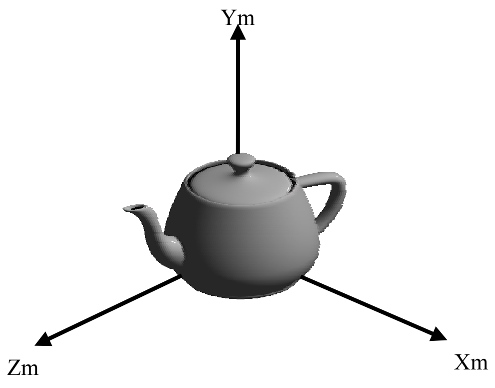


Figure 5-2 Curved Surface Primitive (Bezier Surface)

5.1. Bezier Surfaces

With Bezier surfaces, control points at the edge of a patch are shared with an adjacent patch. The number of control points are 3N + 1 (N = 1, 2, 3,...) for each direction of U and V. Their values can be set with the Bezier Surface command (CMD_BEZIER). Figure 5-3 shows a sample Bezier surface whose number of control points is seven for each U and V direction. In this case, the total number of patches is four.

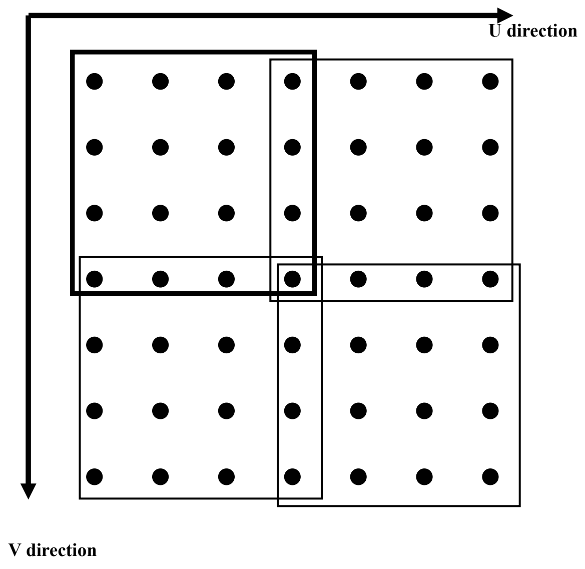


Figure 5-3 Bezier Surface 1

Figure 5-4 shows a sample of a solid figure for the Bezier surface that consists of two patches.

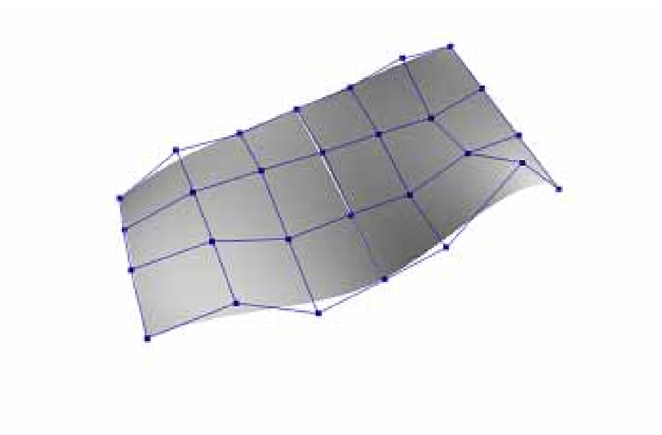


Figure 5-4 Bezier Surface 2

The Graphics Engine supports polygon division for Bi-Cubic Bezier surfaces. Point S on a Bi-Cubic Bezier surface can be expressed by the following equation using the Bernstein basis functions B.

$$S(u,v) = (x(u,v), y(u,v), z(u,v)) = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i(u) B_j(v) P_{i,j}$$

provided $0 \le u, v \le 1$

Here, the Bernstein basis functions determine the proportions for blending each control point, and the u-direction blending functions are defined as follows.

$$B_0 = (1-u)^3, B_1 = 3u(1-u)^2, B_2 = 3u^2(1-u), B_3 = u^3$$

The blending functions are similar in the v-direction.

5.2. Spline Surfaces

With a spline surface, a value of four or more can be assigned to the number of control points for each direction U and V. The number of control points can be set using the Draw Spline command (CMD_SPLINE). Figure 5-5 shows a sample spline surface. In this case, the number of control points in the U direction is six and the number in the V direction is five. The total number of patches is six.

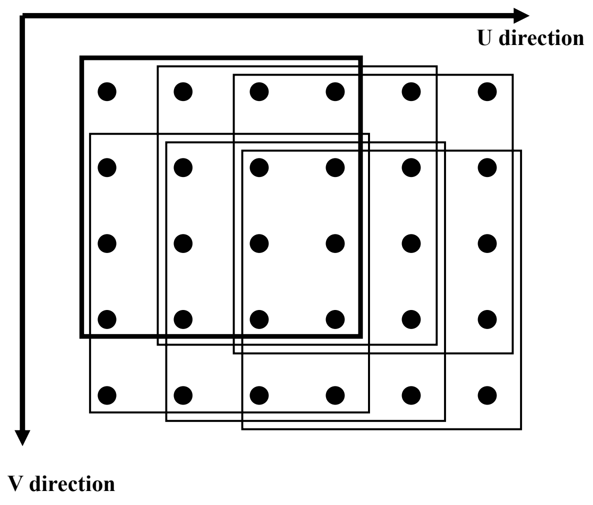


Figure 5-5 Spline Surface 1

Figure 5-6 shows a sample of a solid figure of a spline surface that consists of two patches.

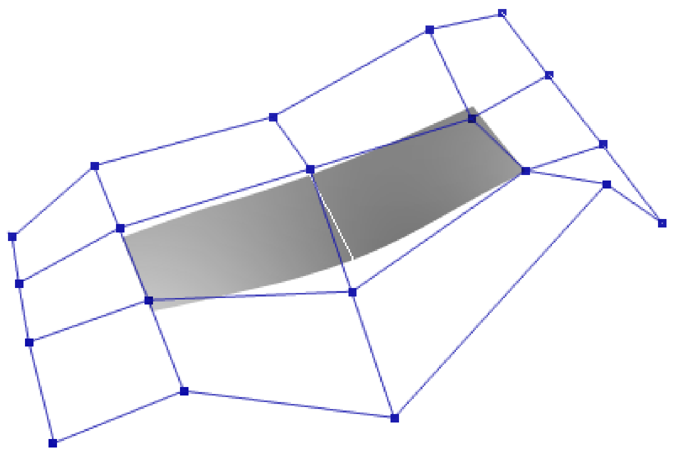


Figure 5-6 Spline Surface 2

5.2.1. Connectivity of spline surfaces

Besides the control points, the nature and shape of spline surfaces also depend on knots at the start and end points. Two types of knots are available, open and closed, and you can choose either one for a start point or an end point in each of the U and V directions. The knot types can be set with the Draw Spline command (CMD_SPLINE).

Open edges of a spline surface can be connected with other open edges, or a Bezier surface,

to form a continuous surface by sharing control points at the edges. Close edges can be connected smoothly with other close edges that share a U or V axis by sharing three-row control points at the connecting edges. In this close case, it is preferable that edges of U axes (or edges of V axes) are connected to each other. We do not recommend connecting an edge of a U axis to an edge of a V axis since the connectivity is not guaranteed.
Examples of knot types are shown in Figure 5-7, Figure 5-8 and Figure 5-9. These examples show cases of all points closed, only the start point in the U direction being open, and only the start point in the U direction and the end point in the V direction being open, respectively.

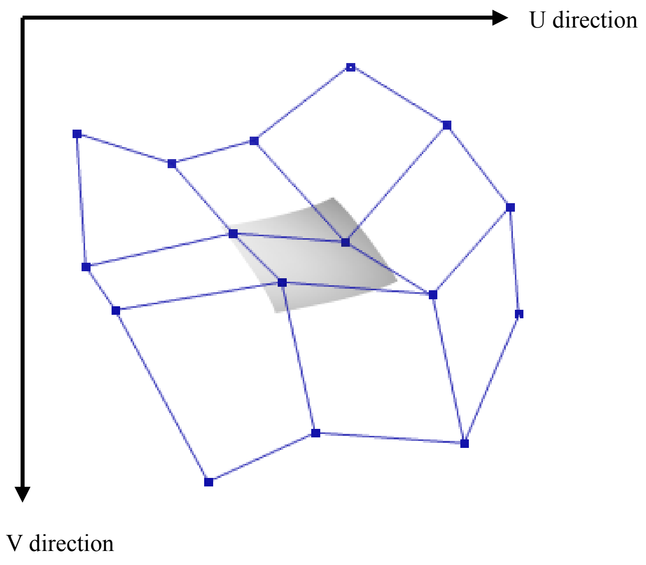


Figure 5-7 Spline Surface: All points are closed

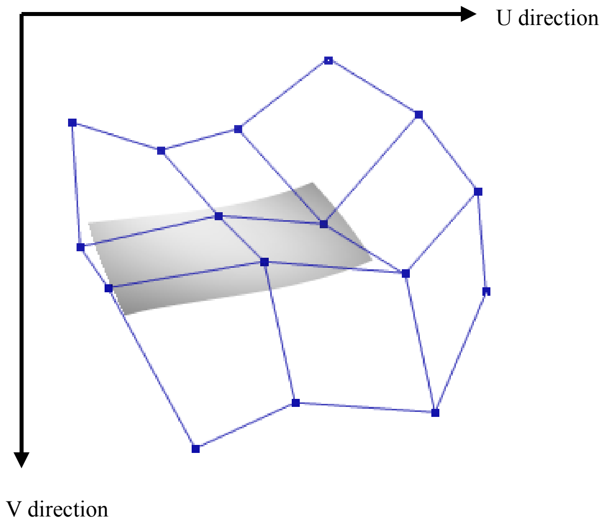


Figure 5-8 Spline Surface: Only the start point in the U direction is open

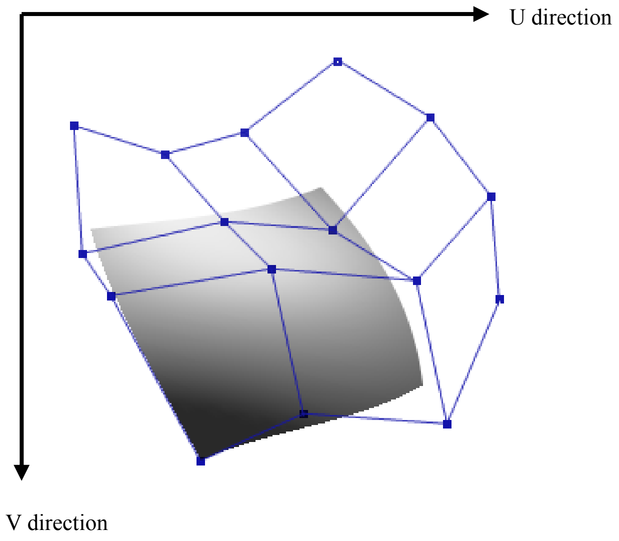


Figure 5-9 Spline Surface: Only the start point in the U direction and the end point in the V direction are open

If all knots of a spline surface are open with one patch, the shape will be the same as the Bezier surface shown in Figure 5-10.

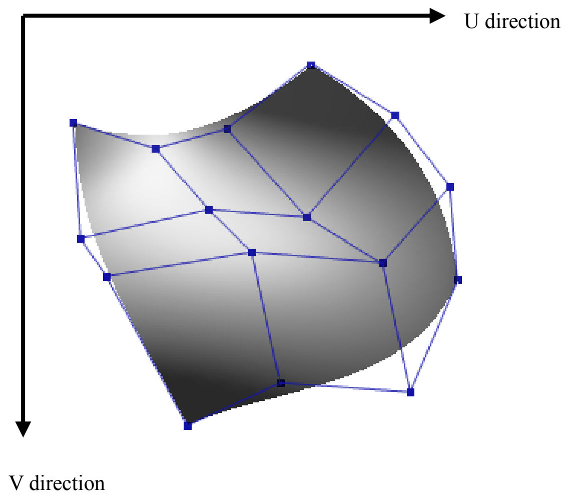


Figure 5-10 Spline Surface: All points are open

The Graphics Engine supports polygon division for Bi-Cubic Uniform B-Spline and some Non-Uniform B-Spline surfaces. A point S on a Bi-Cubic Non-Uniform B-Spline surface is defined by the following equation using the B-Spline basis function N.

$$S(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,3}(u) N_{j,3}(v) P_{i,j}$$

provided $0 \le u \le (n-2), 0 \le v \le (m-2)$

n and m represent values that are one less than the number of control points in the u- and v-directions, respectively. Here, the B-Spline basis functions determine the proportions for blending the control points of the spline surface, and the u-direction blending functions are defined by the following recursive expressions using the knot vector U.

$$N_{i,0}(u) = \begin{cases} 1 (\text{if } u_i \le u < u_{i+1}) \\ 0 (\text{all other cases}) \end{cases}$$

$$N_{i,j}(u) = \frac{u - u_i}{u_{i+j} - u_i} N_{i,j-1}(u) + \frac{u_{i+j+1} - u}{u_{i+j+1} - u_{i+1}} N_{i+1,j-1}(u)$$

$$U = \{u_0, u_1, u_2, u_3, \ldots, u_{n+1}, u_{n+2}, u_{n+3}, u_{n+4}\}$$

Here, ui is called a knot. A knot determines the affected range of a control point and a knot vector is an ordered sequence of knots. A u-direction knot vector U and v-direction knot

vector V are needed to define a spline curve.
The Graphics Engine enables the following four types of knot vectors to be defined in the u- and v-directions, respectively.

Knot Vector Type	Knot Vector
CLOSE/CLOSE	{-3,-2,-1,0,…,n-2,n-1,n,n+1}
OPEN/CLOSE	{0,0,0,0,…,n-2,n-1,n,n+1}
CLOSE/OPEN	{-3,-2,-1,0,…,n-2,n-2,n-2,n-2}
OPEN/OPEN	{0,0,0,0,…,n-2,n-2,n-2,n-2}

5.3. Patch Division

The division level for a patch can be set from 1 division through 64 divisions in each direction of U and V. The higher a division level is, the smoother is the surface that can be drawn. However, the drawing performance will be lower since the number of polygons increases. The division level can be set using the Patch Division Count command (CMD_DIVIDE). Figure 5-11 shows sample divisions of 4x4 and 16x16.

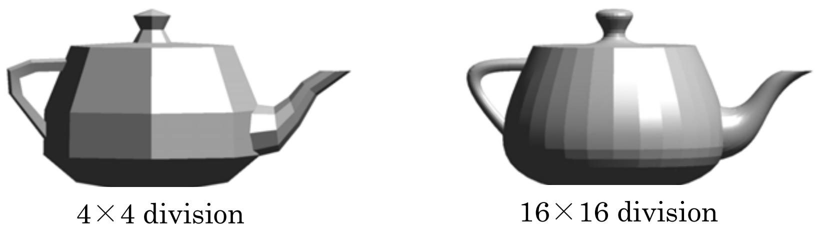


Figure 5-11 Example of Patch Division

When drawing, patches are transformed to primitives corresponding to a preset division level. The primitives here should be Points, Lines or Triangles that are basic primitives. That is, both Bezier surfaces and spline surfaces are made up of basic primitives. The basic primitives used for patch division can be set using the Patch Primitive command (CMD_PPM).
Figure 5-12 shows how the image of a pot is divided into the basic primitives of Points, Lines and Triangles.

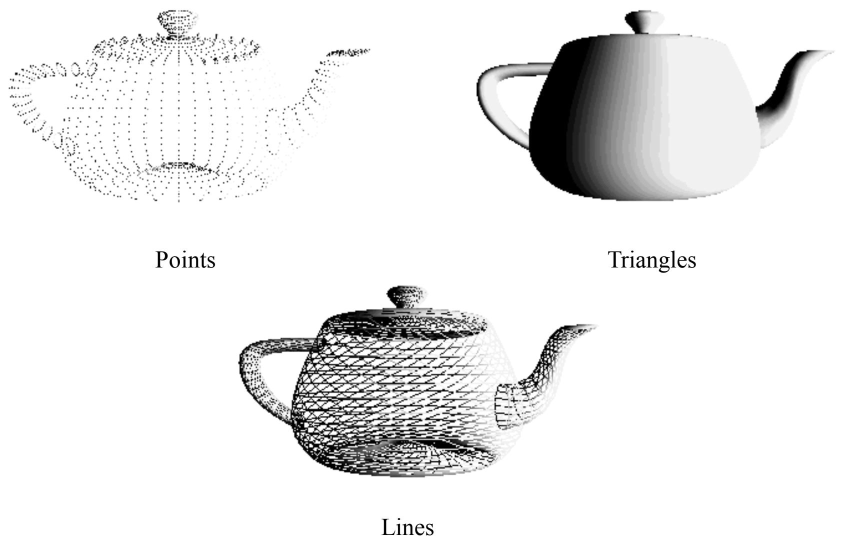


Figure 5-12 Example Showing Division into Basic Primitives

Figure 5-13 shows an example of a patch whose U and V directions are divided by four each. In this case, Triangles are specified as the basic primitive to divide the patch.

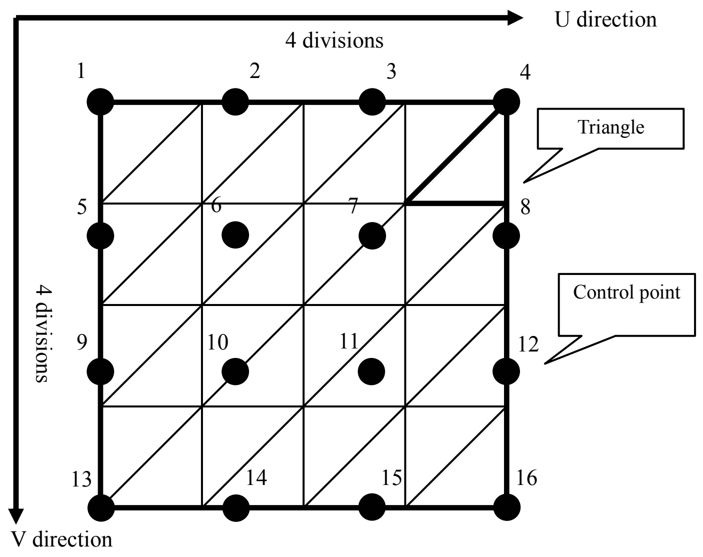


Figure 5-13 Example of 4x4 Division Level (Basic Primitive: Triangle)

Only when Triangles are specified as the basic primitive, patches are divided with either a "Clockwise" or "Counterclockwise" direction. Figure 5-14 shows an example of patches that are transformed to Triangles with a division level of 4x4. There are several ways to perform the transformation depending on the control point at the start of drawing. This example shows two types of orders. In clockwise order, drawing starts from the upper-right-corner point and proceeds to the left. In counterclockwise order, drawing starts from the upper-left-corner point and proceeds to the right. In addition, the normal generated at each vertex after division is such that the clockwise direction faces the front. To select the face, use the Patch Face command (CMD_PFACE).

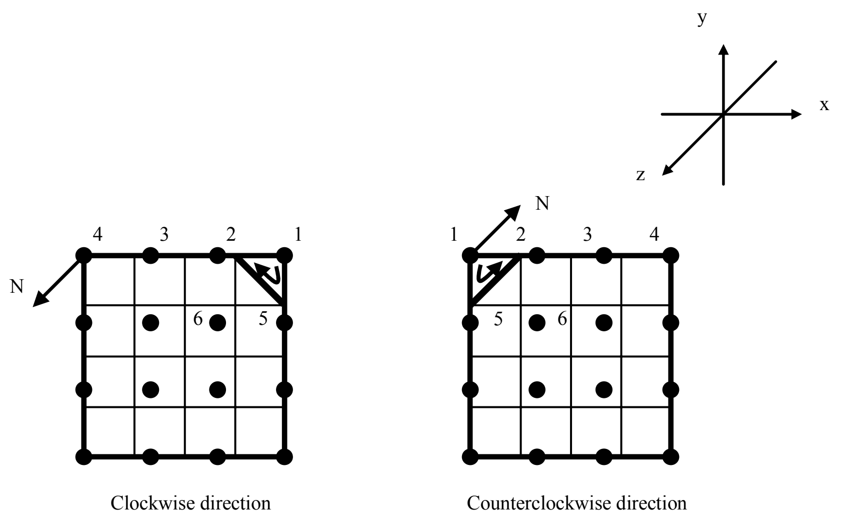


Figure 5-14 Example: Order of Control Points

6. Vertex Formats

With the Graphics Engine, you can use a wide variety of vertex data formats for drawing. Vertex formats can be set using the Vertex Type command (CMD_VTYPE). Vertex formats are mainly divided into two modes, Normal Mode and Through Mode. Which mode is active depends on the coordinate mode setting of the Vertex Type command (CMD_VTYPE). In general, Normal Mode is used for three-dimensional drawing, and Through Mode is used for two-dimensional drawing. In addition, both direct and index transfers are supported for transferring vertex data.

6.1. Vertex Format in Normal Mode

The vertex format in Normal Mode consists of five parameters as shown below. By combining these parameters one vertex format can be defined.

- Weight (Wa, Wb, Wc, Wd, We, Wf, Wg, Wh)
- Texture coordinate (S, T)
- Color (C)
- Normal vector (Nx, Ny, Nz)
- Model coordinate (Xm, Ym, Zm)

The model coordinate is the only parameter that is required and the others can be selected if necessary. However when the parameters are stored, they must be stored in the order shown above.
Except for color, the parameters can be stored in any of the following representations: as an 8-bit fixed-point number, a 16-bit fixed-point number, or a 32-bit floating-point number. Values can be either signed or unsigned depending on the parameters. With respect to color, the 16-bit 5:6:5:0 color format, 16-bit 5:5:5:1 color format, 16-bit 4:4:4:4 color format, or 32-bit 8:8:8:8 color format can be selected. In addition, up to eight weight parameters can be selected. See the table below for details.

Parameter	Data Type
Weight	8-bit unsigned fixed-point number 16-bit unsigned fixed-point number 32-bit floating-point number

Parameter	Data Type
Texture coordinate	8-bit unsigned fixed-point number 16-bit unsigned fixed-point number 32-bit floating-point number
Color	16-bit 5:6:5:0 color format 16-bit 5:5:5:1 color format 16-bit 4:4:4:4 color format 32-bit 8:8:8:8 color format
Normal vector	8-bit signed fixed-point number 16-bit signed fixed-point number 32-bit floating-point number
Model coordinate	8-bit signed fixed-point number 16-bit signed fixed-point number 32-bit floating-point number

6.2. Vertex Format in Through Mode

The vertex format in Through Mode consists of three parameters as shown below. By combining these parameters one vertex format can be defined. Note that the weight and normal vector, which are definable in Normal Mode, cannot be defined in Through Mode.

- Texel coordinate (U,V)
- Color (C)
- Drawing coordinate (Xd,Yd,Zd)

The drawing coordinate is the only parameter that is required and the others can be selected if necessary, in the same way as for Normal Mode. However, when the parameters are stored, they must be stored in the order shown above.
Except for color, parameters must be represented as 16-bit integers. With respect to color, the 16-bit 5:6:5:0 color format, 16-bit 5:5:5:1 color format, 16-bit 4:4:4:4 color format, or 32-bit 8:8:8:8 color format can be selected as in Normal Mode. See the table below for details.

Parameter	Data Type
Texel coordinate	16-bit unsigned integer
Color	16-bit 5:6:5:0 color format 16-bit 5:5:5:1 color format 16-bit 4:4:4:4 color format 32-bit 8:8:8:8 color format
Drawing coordinate	16-bit signed integer (Only Z is unsigned) 32-bit floating-point number

Note that the following operations do not function in Through Mode even if they are enabled.

- Vertex blending
- Lighting
- Texture mapping mode (always performed as UV mapping)
- Clipping
- Fogging
- Depth range test
- Object culling
- Patch culling

6.3. Vertex Addresses

When a direct transfer is performed, vertex data is transferred from a vertex data address. When an index transfer is performed, an index value is read from an index data address, then the vertex data is transferred using a base address.
The vertex data register can be set using the Vertex Data command (CMD_VADR), and the index data register can be set using the Index Data command (CMD_IADR). See Section 20.3, "Display List Address Generation" for a description of how the address is set for each command.

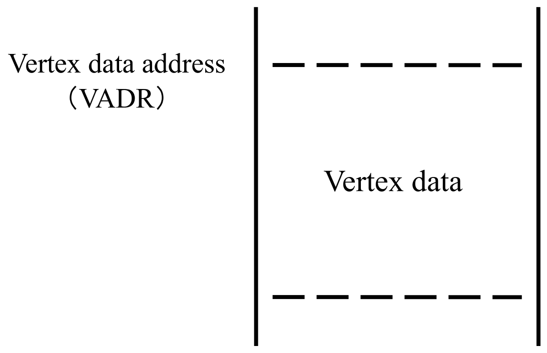


Figure 6-1 Direct Transfer

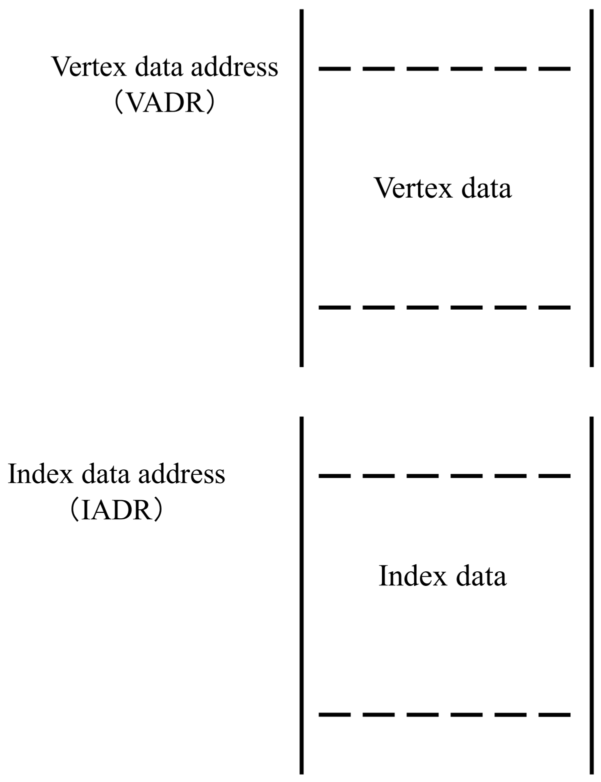


Figure 6-2 Index Transfer

6.4. Storing Vertex Data

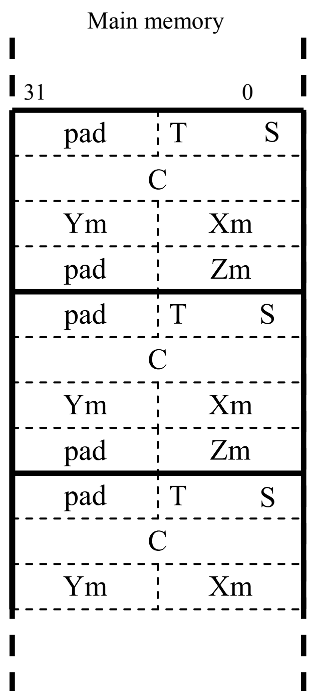
Some alignment rules apply to the storing of vertex data. One rule is that 16-bit data is always aligned on a half word boundary. Likewise, 32-bit data is always aligned on a word boundary. In addition, the last piece of vertex data is aligned on the boundary appropriate for the largest parameter. For example if one of the parameters is 32 bits, the last piece of vertex data should be aligned on a word boundary.
These are the same rules that apply when a structure is defined in the C language. Therefore, with the Graphics Engine it is possible to define vertex data by using C language structures.

6.4.1. Storing Vertex Data Example 1 (Normal Mode)

Parameter	Data Type
Weight	None
Texture coordinate	8-bit unsigned fixed-point number
Color	32-bit 8:8:8:8 color format

Parameter	Data Type
Normal vector	None
Model coordinate	16-bit signed fixed-point number

In this example, the weight and normal vector are not selected so that six parameters (S, T, C, Xm, Ym and Zm) are used. Because C is 32 bits, 16 bits of padding are added after S and T as shown in Figure 6-3. The end of the vertex data is also aligned on a word boundary since C (32-bit) is the largest parameter.



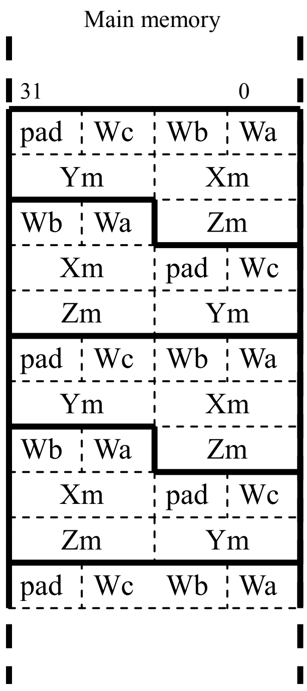
```
struct example1
{
   unsigned char S, T;
   unsigned int C __attribute__((aligned(4)));
   signed short Xm, Ym, Zm;
} __attribute__((packed));
```

Figure 6-3 Storing Vertex Data Example 1

6.4.2. Storing Vertex Data Example 2 (Normal Mode)

Parameter	Data Type
Weight	8-bit unsigned fixed-point number (Wa, Wb, Wc)
Texture coordinate	None
Color	None
Normal vector	None
Model coordinate	16-bit signed fixed-point number

In this example, the texture coordinate, color and normal vector are not selected so that six parameters (Xm, Ym, Zm, Wa, Wb and Wc) are used. Because Xm, Ym and Zm are each 16 bits, 8 bits of padding are inserted after Wc as shown in Figure 6-4. Since the end of the vertex data is already 16-bit aligned, no extra padding is inserted.



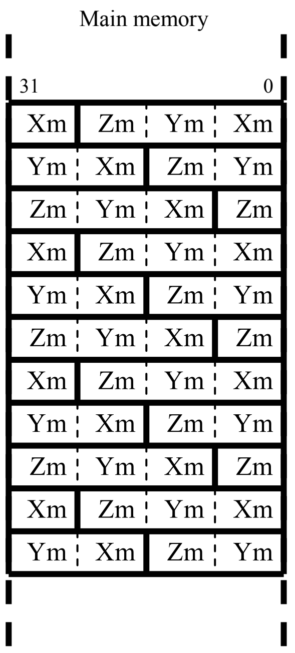
```
struct example2
{
   unsigned char Wa, Wb, Wc;
   signed short Xm __attribute__((aligned(2)));
   signed short Ym, Zm;
} __attribute__((packed));
```

Figure 6-4 Storing Vertex Data Example 2

6.4.3. Storing Vertex Data Example 3 (Normal Mode)

Parameter	Data Type
Weight	None
Texture coordinate	None
Color	None
Normal vector	None
Model coordinate	8-bit signed fixed-point number

In this case, no parameter is selected except for the model coordinate. Therefore, the parameters used here are Xm, Ym and Zm. All of the parameters are 8 bits so no padding is needed.



```
struct example3
{
   signed char Xm, Ym, Zm;
} __attribute__((packed));
```

Figure 6-5 Storing Vertex Data Example 3

6.4.4. Storing Vertex Data Example 4 (Through Mode)

Parameter	Data Type
Texel coordinate	16-bit unsigned integer
Color	16-bit 5:5:5:1 color format
Drawing coordinate	16-bit signed integer

In this case, the parameters used are U, V, C, Xd, Yd and Zd. All of the parameters are 16 bits so no padding is needed.

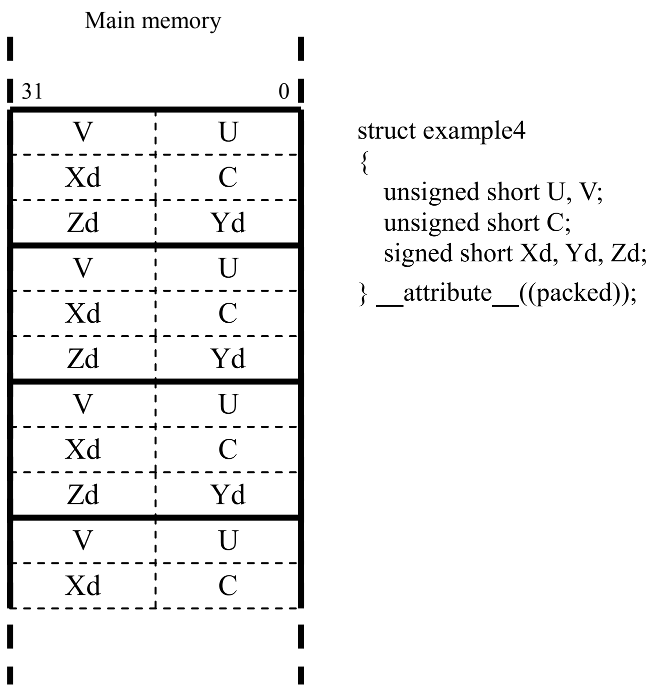


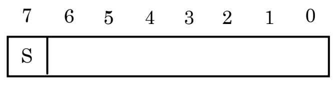
Figure 6-6 Storing Vertex Data Example 4

6.5. Data Formats

The Graphics Engine also uses special data formats other than normal integers for vertex data parameters. The details of the various data formats are shown below.
Note that the color format notation a[b:c] represents bit (c) to bit (b) of color component (a) before expansion.

8-bit signed fixed-point number

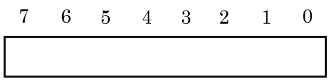
An 8-bit signed fixed-point number is a fixed-point number whose decimal fraction is 7 bits wide and includes a sign bit in the high-order bit. A negative number is indicated with a two's complement representation.



* The decimal point is located between bits 6 and 7.

8-bit unsigned fixed-point number

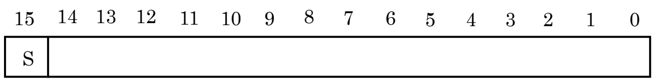
An 8-bit unsigned fixed-point number is a fixed-point number whose decimal fraction is 7 bits wide.



* The decimal point is located between bits 6 and 7.

16-bit signed fixed-point number

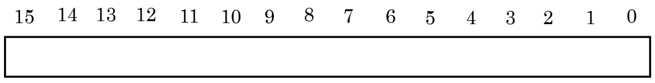
A 16-bit signed fixed-point number is a fixed-point number whose decimal fraction is 15 bits wide and includes a sign bit in the high-order bit. A negative number is indicated with a two's complement representation.



* The decimal point is located between bits 14 and 15.

16-bit unsigned fixed-point number

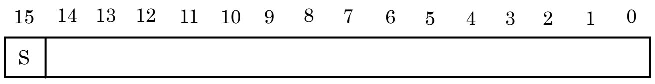
A 16-bit unsigned fixed-point number is a fixed-point number whose decimal fraction is 15 bits.



* The decimal point is located between bits 14 and 15.

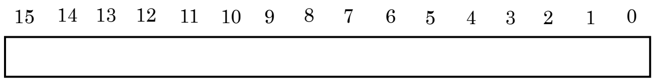
16-bit signed integer

A 16-bit signed integer is a 15-bit integer with a sign bit in the high-order bit. A negative number is indicated with a two's complement representation.



16-bit unsigned integer

A 16-bit unsigned integer is a 16-bit integer.



32-bit floating-point number

A 32-bit floating-point number is based on the IEEE 754 single-precision floating-point format and has the following structure. A 32-bit floating-point number does not support

NaN (E = 255, F!=0), infinity (E = 255, F = 0) or denormalized numbers (E = 0, F!=0). Equation 6-1 shows how values are represented in this format.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E								F																						

$$(-1)^{s} \times 1.F \times 2^{(E-127)}$$

Equation 6-1 32-bit Floating-point Number

16-bit 5:6:5:0 color format

In the 16-bit 5:6:5:0 color format, R and B are each 5 bits, and G is 6 bits. The Graphics Engine further expands this format to 32-bit 8:8:8:8 when data is input as shown below. 65536 colors can be represented in this format and the alpha value is considered to be 255.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B					G						R				

↓

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	B					B[4:2]			G						G[5:4]		R					R[4:2]		

16-bit 5:5:5:1 color format

In the 16-bit 5:5:5:1 color format, R, G and B are each 5 bits, and A is 1 bit. The Graphics Engine further expands this format into 32-bit 8:8:8:8 when data is input as shown below. 32768 colors can be represented in this format, but the alpha value can only represent two gradations of color.

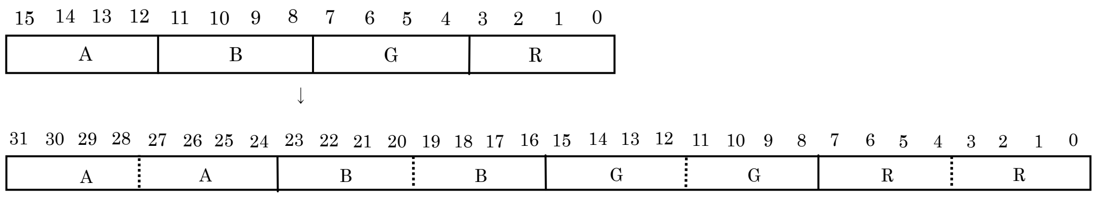
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A	B					G					R				

↓

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A	A	A	A	A	A	A	A	B					B[4:2]			G					G[4:2]			R					R[4:2]		

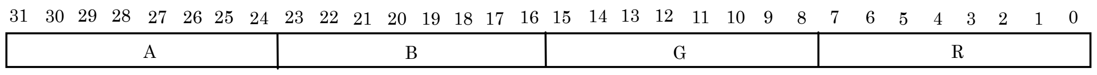
16-bit 4:4:4:4 color format

In the 16-bit 4:4:4:4 color format, R, G, B and A are each 4 bits. The Graphics Engine further expands this format into 32-bit 8:8:8:8 when data is input as shown below. Only 4096 colors can be represented in this format, but the alpha value can represent 16 gradations of color.



32-bit 8:8:8:8 color format

In the 32-bit 8:8:8:8 color format, R, G, B and A are each 8 bits. Full color can be represented, and the alpha value can also represent 256 gradations of color.



6.6. Index Format

The Graphics Engine has a function that can process an indexed vertex array. This function makes it unnecessary to replicate the same vertex data in main memory, enabling the total amount of data to be reduced.

Either 8-bit unsigned integers or 16-bit unsigned integers can be used as the index value. Using the 8-bit format enables a vertex array of up to 256 vertices to be addressed, whereas the 16-bit format supports a vertex array with up to 65536 vertices.

To perform index drawing, use the Index Data command (CMD_IADR) to set the starting address of the index array in advance and specify the index type when drawing.

For instance, suppose you want to use an index array of 8-bit unsigned integers when drawing a quadrilateral enclosed by lines as shown in Figure 6-7. Since the data for P0 is repeated, an index array and vertex array can be set up as shown in the figure.

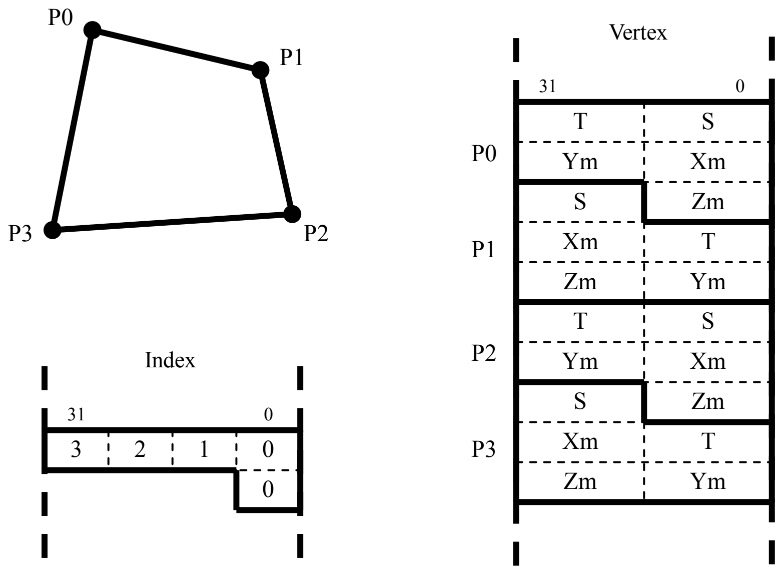


Figure 6-7 Example Data for Index Drawing

7. Vertex Blending

7.1. Morphing

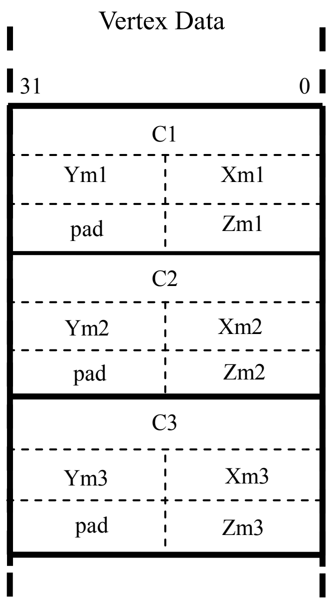
The Graphics Engine has a function which uses weights to blend between vertices, as shown in Equation 7-1. This function is used to smoothly transform from one shape to another (morphing). Up to eight vertices can be blended using morphing weights (Mi). Morphing is executed for all parameters that have been set up by the vertex format.
In addition, the number of vertices for morphing is set with the blending vertex count parameter (MC) of the Vertex Type command (CMD_VTYPE), and the morphing weight (Mi) is set with the Vertex Weight commands (CMD_WEIGHT0-7).
Note that morphing is not executed if the coordinate defined mode is Through Mode.

$$P' = \sum_{i=0}^{\text{No. of vertices}(MC)} MiPi$$

*P=[S, T, C, Nx, Ny, Nz, Xm, Ym, Zm, Wa, Wb, Wc, Wd, We, Wf, Wg, Wh]

Equation 7-1 Morphing

When morphing is executed, vertex data for a set of blending vertices (MC) is packed into one structure. As an example, when morphing is executed for three vertices, the structure will be as follows (Figure 7-1). When index transfers are performed, the index values for a set of MC must be packed into one structure.



```
struct blend_vertex
{
    // Vertex 1
    unsigned int C1 __attribute__((aligned(4)));
    signed short Xm1, Ym1, Zm1;

    // Vertex 2
    unsigned int C2 __attribute__((aligned(4)));
    signed short Xm2, Ym2, Zm2;

    // Vertex 3
    unsigned int C3 __attribute__((aligned(4)));
    signed short Xm3, Ym3, Zm3;
} __attribute__((packed));
```

Figure 7-1 Example Data for Morphing

Figure 7-2 shows an example of morphing. In this example, a leopard model is morphed into a dinosaur model.

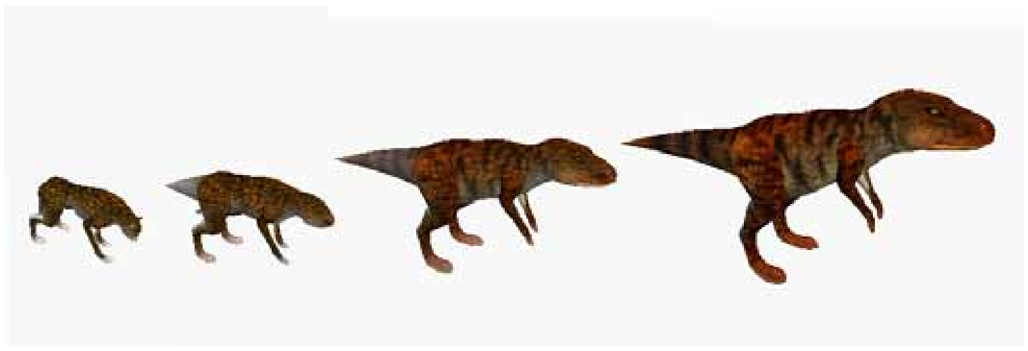


Figure 7-2 Morphing

7.2. Skinning

The Graphics Engine has a skinning (bone blending) function, which executes blending using a bone matrix with respect to the normal vector of a vertex and model coordinate as shown in Equation 7-2. This function is mainly used to animate vertices, which are dependent on bone movements. Up to eight bone matrices can be used for skinning. Parameters, except for the normal vector of the vertex and model coordinate, are not changed when skinning is performed.

The skinning weight parameters (Wa, Wb, Wc, Wd, We, Wf, Wg, Wh) are set for each vertex.

The number of weights is set with the weight count parameter (WC) of the Vertex Type command (CMD_VTYPE), and the bone matrix is set with the Bone Matrix Number command (CMD_BONEN) and the Bone Matrix Data command (CMD_BONED).

Note that skinning is not executed if the coordinate defined mode is Through Mode.

$$\begin{vmatrix} Xm' \\ Ym' \\ Zm' \end{vmatrix} = \sum_{i=a,b,c,d,e,f,g,h} Wi \left(\begin{vmatrix} Bi00 & Bi03 & Bi06 \\ Bi01 & Bi04 & Bi07 \\ Bi02 & Bi05 & Bi08 \end{vmatrix} \begin{vmatrix} Xm \\ Ym \\ Zm \end{vmatrix} + \begin{vmatrix} Bi09 \\ Bi10 \\ Bi11 \end{vmatrix} \right)$$

$$\begin{vmatrix} Nx' \\ Ny' \\ Nz' \end{vmatrix} = \sum_{i=a,b,c,d,e,f,g,h} Wi \begin{vmatrix} Bi00 & Bi03 & Bi06 \\ Bi01 & Bi04 & Bi07 \\ Bi02 & Bi05 & Bi08 \end{vmatrix} \begin{vmatrix} Nx \\ Ny \\ Nz \end{vmatrix}$$

Equation 7-2 Skinning

Figure 7-3 and Figure 7-4 show examples of skinning. It is possible to change the positions of vertices depending on the movement of bones as shown in the examples.

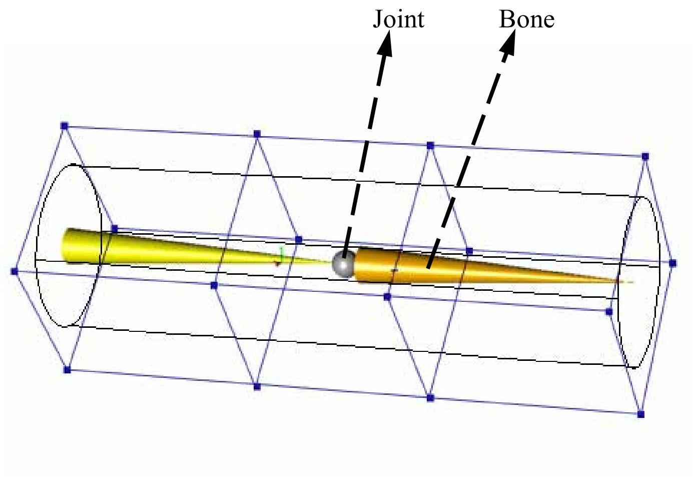


Figure 7-3 Skinning 1

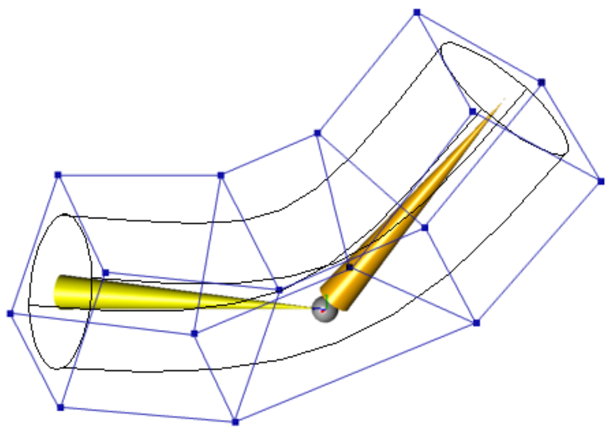


Figure 7-4 Skinning 2

8. Shading

With the Graphics Engine, it is possible to select and specify the shading methods shown below when drawing Lines or Triangles. For Bezier and spline drawing it is also possible to specify the shading method since both drawings are subdivided into basic primitives.

- Flat shading
- Gouraud shading

For flat shading, all pixels are drawn by using the color value of an end point for a Line drawing, or the color value of the third vertex for a Triangle drawing.
For Gouraud shading, color values at each vertex are interpolated linearly at each pixel when both Lines and Triangles are drawn.
Figure 8-1 shows the drawing of an object with both flat and Gouraud shadings. The sphere on the left is drawn with flat shading and the one on the right is drawn with Gouraud shading. As you can see, the surface of the object can be represented smoothly when Gouraud shading is applied.
With Rectangles, all pixels are drawn using the color value of the second diagonal point regardless of the shading method.
To change the shading method, use the Shading Command (CMD_SHADE).

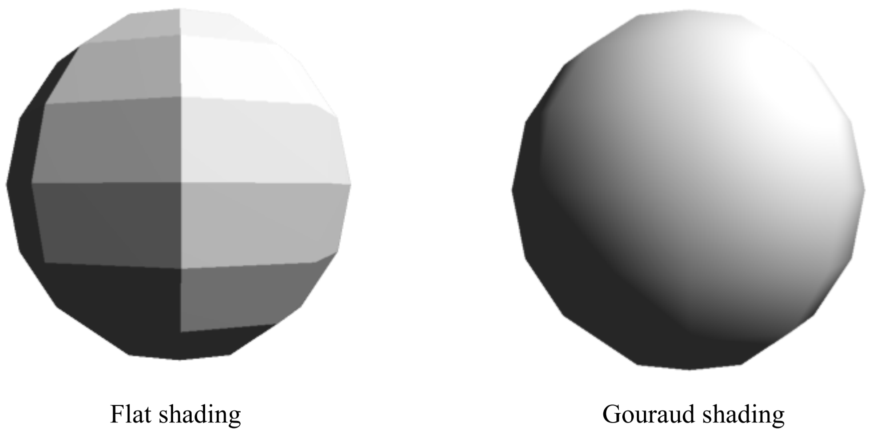


Figure 8-1 Effect of Shading

9. Lighting

The Graphics Engine has four built-in hardware light sources. Settings are provided to switch lighting on and off using the Lighting Enable command (CMD_LTE), and to enable and disable each light source using the Light Enable commands (CMD_LE0-3). The surface that is lit can be swapped using the Normal Reverse command (CMD_NREV).

9.1. Types of Light Sources

The Graphics Engine supports three types of light sources as shown below. The light source type can be selected using the Light Type command (CMD_LTYPE0-3). Each type can be assigned to any of the four built-in light sources.

- Directional light source
- Point light source
- Spotlight

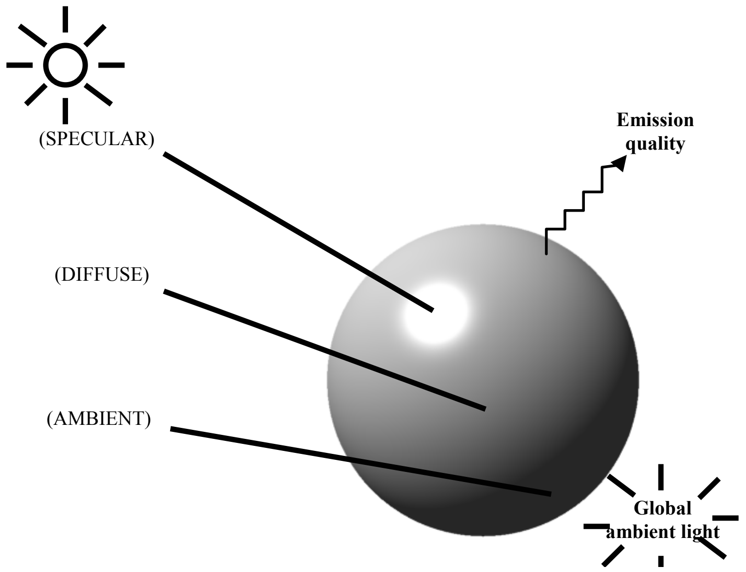


Figure 9-1 Light Source Types

9.2. Light Source Calculation

The Graphics Engine performs light source calculations for vertices that are made up of objects, and from that determines the color of each vertex. As shown in Figure 9-1, the color of each vertex is calculated from the emission quality, the global ambient light, and from the ambient, diffuse and specular lighting affected by the light source.

Vertex color = Emission quality
+ Global ambient light
+ Effect from light source (ambient, diffuse, specular lighting)

9.2.1. Emission quality

The color which an object emits (MEC) can be specified using the Model Color command (CMD_MEC).

Emission quality = MEC

9.2.2. Global ambient light

You can set up a global ambient light which is not affected by a specific light source. The global ambient light is calculated by multiplying the ambient color (AC) by the model color (MAC).
AC can be set using the Ambient Color command (CMD_AC) and MAC can be set using the Vertex Color or Model Color command (CMD_MAC).

$$\text{Global Ambient Light} = AC \times MAC$$

Equation 9-1 Global Ambient Light

9.2.3. Effect of light source

The attenuation and spotlight effect of each light source varies according to the distance from the light source and these effects are added together.

$$\text{Effect of light source} = \sum \text{(Attenuation} \times \text{Spotlight effect} \times \text{(Ambient light + Diffuse light + Specular light))}$$

Equation 9-2 Effect of Light Source

Attenuation (att) is given by the following equation.

$$att = clamp[0,1]\left(\frac{1}{LKA + LKB \times d + LKC \times d^2}\right)$$

Equation 9-3 Attenuation

Here, d is the distance from each light source to the vertex, and LKA, LKB and LKC are light distance attenuation factors. These can be set using the CMD_LKA0-3, CMD_LKB0-3 and CMD_LKC0-3 commands.
However, when directional is set for the light source type, the attenuation is always 1.0.
The spotlight effect (spot) is given by the following equation.

$$\begin{aligned} &when\ \frac{\mathbf{V} \bullet \mathbf{LD}}{|\mathbf{V}||\mathbf{LD}|} > \mathbf{LKO} \\ &\mathbf{spot} = \left(\frac{\mathbf{V} \bullet \mathbf{LD}}{|\mathbf{V}||\mathbf{LD}|}\right)^{\mathbf{LKS}} \end{aligned}$$

$$\begin{aligned} &when\ \frac{\mathbf{V} \bullet \mathbf{LD}}{|\mathbf{V}||\mathbf{LD}|} \leq \mathbf{LKO} \\ &\mathbf{spot} = 0 \end{aligned}$$

Equation 9-4 Spotlight Effect

Here, V is the vector from the light source to the vertex, LD is the direction of the spotlight, set by the Light Direction commands (CMD_LDX0-3, CMD_LDY0-3, and CMD_LDZ0-3), LKS is the light convergence factor, set by the Light Convergence Factor commands (CMD_LKS0-3), LKO is the light cut-off dot product coefficient which limits the spread of the spotlight and which is set by the Light Cut-off Dot Product Coefficient commands (CMD_LKO0-3). It is not necessary to normalize LD to the unit vector in advance.

Note that when the light source type is not a spotlight, the spotlight effect will always be 1.0.

Ambient light

The following equation shows the effect of light color and model color on the ambient light.

$$\text{Ambient light} = \text{LAC} \times \text{MAC}$$

Equation 9-5 Ambient light

Here, LAC is the light color, and MAC is the vertex color or model color. LAC can be set using the Light Color commands (CMD_LAC0-3). MAC can be set using the Vertex Color or Model Color command (CMD_MAC).

Diffuse light

Diffuse light is affected by the light color, model color, vertex normal and light position as seen in the following equation.

$$\text{Diffuse light} = \text{LDC} \times \text{MDC} \times \left(\frac{\max(\text{L} \bullet \text{N}, 0)}{|\text{L}||\text{N}|} \right)$$

Equation 9-6 Diffuse Light

Here, LDC is the light color, MDC is the vertex color or model color, L is the light vector from the vertex to the light source, and N is the vertex normal. LDC can be set using the Light Color commands (CMD_LDC0-3) and MDC can be set using the Vertex Color or Model Color command (CMD_MDC).
The normal is either included in vertex data, or, for Bezier and spline drawing, is automatically generated internally by the Graphics Engine.
When a directional light source is used, the values set by the Light Position commands (CMD_LX0-3, CMD_LY0-3, and CMD_LZ0-3) are used for the light vector. For other types of light sources, the light source is assumed to be at the position set by the Light Position commands, and the values used for the light vector are automatically generated by the Graphics Engine. It is not necessary to normalize the normal or light vector to the unit vector in advance.

If the Light Type commands (CMD_LTYPE0-3) are used to select powered diffuse light as a special case of diffuse light, diffuse light is found with the following equation. K is the specular coefficient of the model which is set with the Model Specular command (CMD_MK).

$$\text{Diffuse light} = LDC \times MDC \times \left(\frac{\max(\mathrm{L} \bullet \mathrm{N}, 0)}{|\mathrm{L}||\mathrm{N}|} \right)^{K}$$

Equation 9-7 Powered Diffuse Light

Specular light

Specular light is affected by the light color, model color, vertex normal, light position, view point position and specular coefficient, as seen in the equation below.
However, when L・N <= 0, specular light is zero so that the back surface will be illuminated.

$$\mathrm{H} = \frac{\mathrm{E}}{|\mathrm{E}|} + \frac{\mathrm{L}}{|\mathrm{L}|}$$

$$\text{Specular light} = \mathrm{LSC} \times \mathrm{MSC} \times \left(\frac{\max(\mathrm{H} \bullet \mathrm{N}, 0)}{|\mathrm{H}|\,|\mathrm{N}|} \right)^{\mathrm{K}}$$

Equation 9-8 Specular Light

Here, E is the line of sight vector from the vertex to the view point, L is the light vector from the vertex to the light source, and the H vector is obtained by normalizing each of these to the unit vector and summing them.
Note that when the Graphics Engine performs light source calculations, because it places the view point at infinity, it uses a fixed vector (0, 0, 1) in the View Coordinate System for the view point vector E.
In addition, LSC is the light color set by the Light Color commands (CMD_LSC0-3), MSC is the vertex color or model color set by the Model Color command (CMD_MSC), K is the specular coefficient of the model set by the Model Specular command (CMD_MK), and N is the vertex normal. The normal and light vector are generated in the same manner as with diffuse light.
Specular light can be switched on and off with the Light Type commands (CMD_LTYPE0-3).

9.2.4. Color calculation

The light source calculation will ultimately compute RGB values for the primary color

(Cp) and secondary color (Cs), along with one alpha (A) value.
Either a single color or separate colors can be used for the primary and secondary colors. The selection can be made with the Light Mode command (CMD_LMODE).
When separate colors are used, the secondary color is added to the texture-mapped color using color addition described later (see 15 Color Addition).
Equation 9-9 shows the color calculation for a single color, and Equation 9-10 shows the color calculation for separate colors. Color is calculated from the sum of each material's emission, global ambient light, ambient light, diffuse light and specular light. Note that specular light can be switched on and off with the Light Type commands (CMD_LTYPE0-3). In addition, when powered diffuse light is specified, it replaces the diffuse light term.

Single color

$$\begin{aligned}
Cp &= MEC \\
&\quad + AC \times MAC \\
&\quad + \sum ((spot)i \times (att)i \times (LACi \times MAC \\
&\qquad + LDCi \times MDC \times (\frac{\max(Li \bullet N, 0)}{| Li || N |}) \\
&\qquad + LSCi \times MSC \times (\frac{\max(Hi \bullet N, 0)}{| Hi || N |})^{K})) \\
Cs &= 0 \\
A &= AA \times MAA
\end{aligned}$$

Equation 9-9 Single Color

Separate colors

$$\begin{aligned}
Cp &= MEC \\
&\quad + AC \times MAC \\
&\quad + \sum ((spot)i \times (att)i \times (LACi \times MAC \\
&\qquad + LDCi \times MDC \times (\frac{\max(Li \bullet N, 0)}{| Li || N |}))) \\
Cs &= \sum ((spot)i \times (att)i \times (LSCi \times MSC \times (\frac{\max(Hi \bullet N, 0)}{| Hi || N |})^{K})) \\
A &= AA \times MAA
\end{aligned}$$

Equation 9-10 Separate Colors

Here, the model colors which are used for calculating the color of each vertex (MAC, MDC, MSC, MAA) use either the color value in the vertex data, or the model colors set by the Model Color commands (CMD_MAC, CMD_MDC, CMD_MSC). Which values are

used for the vertex colors depends on the setting of the update flag in the Material command (CMD_MATERIAL). If the update flag is set, then the color values in the vertex data are used. If the update flag is not set, then the model colors set by the Model Color commands are used.
In addition, if color is not included in the vertex data, the model colors set by the Model Color commands will be used for the vertex colors. Also, AA is the A value of the ambient light set by the Ambient Light Color command.

If lighting is disabled and color is specified in the vertex data, that color value will be used for the vertex color. If lighting is disabled and color is not specified in the vertex data, the color value set by the Model Color commands (CMD_MAC, CMD_MAA) will be used for the vertex color, as shown in the following equations.

$Cp = MAC$

$Cs = 0$

$A = MAA$

10. Clipping

The Graphics Engine can clip a drawing primitive (Point, Line, Triangle) that is outside of a view volume. Clipping can be enabled and disabled using the Clipping Enable command (CMD_CLE). When all vertices are outside of the view volume, the basic operation of the Graphics Engine is to discard those primitives whether or not clipping is enabled. (see Example 2 of Figure 10-1)

It is important to note that in reality, even if there are primitives outside of the view volume, there are cases when they may not be discarded.
Example 5 of Figure 10-1 shows the case when clipping is enabled and primitives are clipped in the near clip surface.

Clipping disabled

In the following cases, primitives are discarded.

- All of the vertices lie outside of one clipping surface (Ex. 2 of Figure 10-1)
- Any vertex exceeds one of the ranges below, after viewport transformation (Ex. 3 of Figure 10-1)
  0 <= Xs, Ys < 4096, or Zs < 65536
- Any vertex lies on the side behind the viewpoint (Ex. 4 of Figure 10-1)

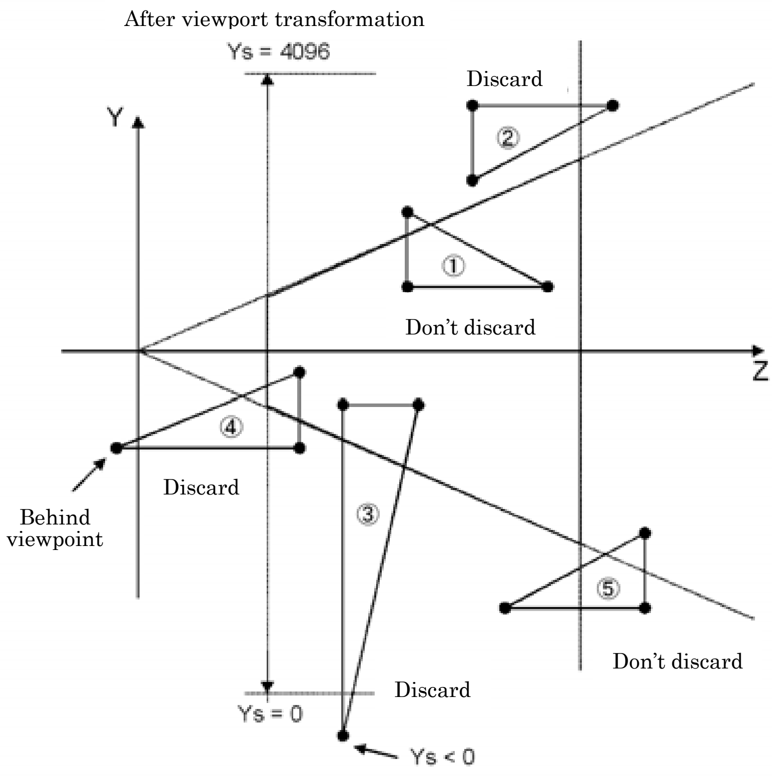


Figure 10-1 Clipping Disabled

Clipping enabled

In the following cases, primitives will be discarded.

- All the vertices lie outside of one clipping surface. (Ex. 2 of Figure 10-2)
- Any vertex exceeds one of the ranges below, after viewport transformation (Ex. 3 of Figure 10-2)
  0 <= Xs, Ys < 4096

In the following case, primitives are clipped then divided.

- One vertex lies outside of the near clip surface. (Ex. 4 of Figure 10-2)

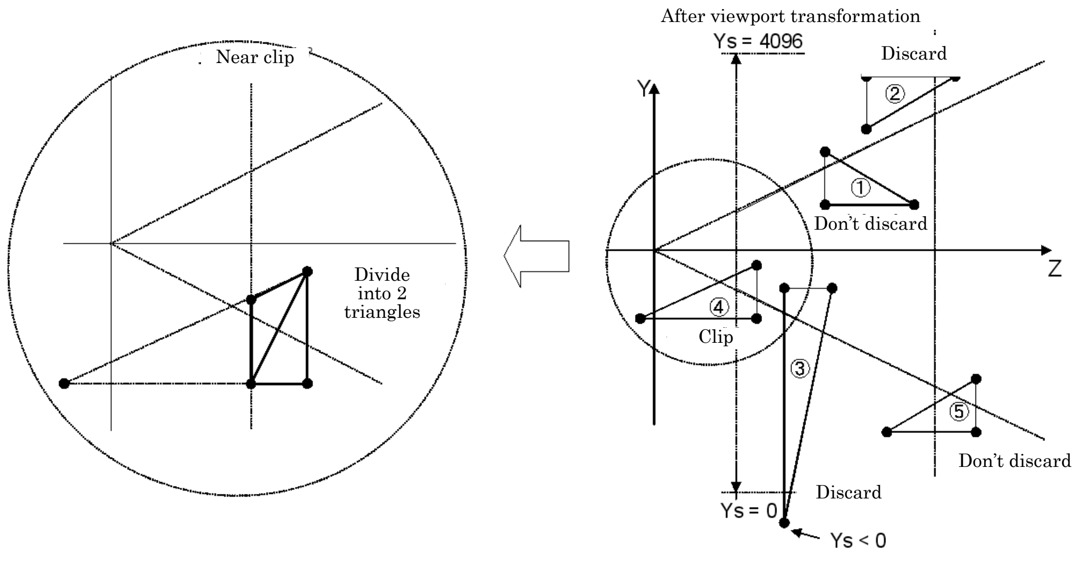


Figure 10-2 Clipping Enabled

Note that during clipping, if the given range in the X or Y directions is exceeded, primitives will end up getting discarded because processing will only be done for the near clip surface. For this case, you must divide the primitives in advance. In addition, the Z value is clamped in the range 0 <= Z <= 65535 after viewport transformation, and if the value of Z is outside of this range, errors will tend to accumulate in the Z value of the primitive.

11. Culling

The Graphics Engine provides three culling functions as follows.

- Object culling
- Patch culling
- Back face culling

These are described in detail below.

11.1. Object Culling

With the Graphics Engine it is possible to draw objects only in a drawing region and to cull objects outside the drawing region, so that fewer memory accesses are needed. An object is specified in a box called a bounding box whose number of vertices is a multiple of eight. The number of vertices can be set using the Bounding Box command (CMD_BBOX). The drawing regions can be set using the Drawing Region commands (CMD_REGION1, CMD_REGION2).
The bounding box can be set using the Bounding Box command (CMD_BBOX) and the Graphics Engine checks whether that bounding box is in a specified drawing region. If part of the bounding box is in the specified drawing region (BFLG=1), the drawing commands for the object in the bounding box are executed without being skipped. On the other hand, if the bounding box is completely outside the drawing region (BFLG=0), the object is culled and execution jumps to the next object using the Conditional Jump command (CMD_BJUMP). Figure 11-1 shows the display list for object culling.

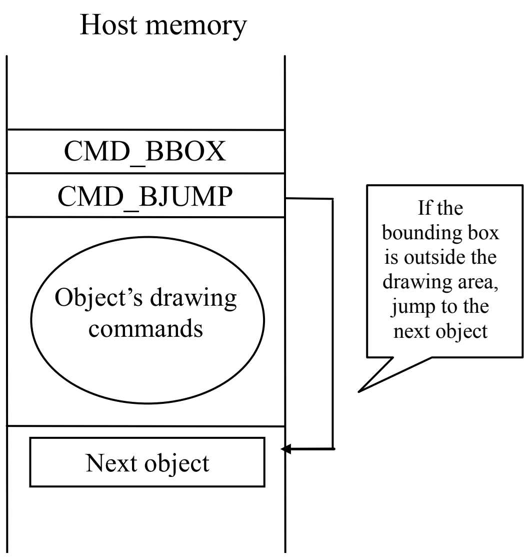

Host memory
CMD_BBOX
CMD_BJUMP
Object's drawing commands
Next object
If the bounding box is outside the drawing area, jump to the next object

Figure 11-1 Display List for Object Culling

Object culling can also be performed hierarchically. Figure 11-2 and Figure 11-3 show display lists that have hierarchical structures. Object 1 is specified with BBOX1, Object 2 with BBOX2 and Object 3 with BBOX3. BBOX2 and BBOX3 are both inside of BBOX1.
In Figure 11-2, Object 1 is drawn since BBOX1 is in the drawing region. BBOX2 is also in the drawing region so Object 2 is drawn. However, Object 3 is culled since BBOX3 is outside the drawing region and execution will jump to the next object.

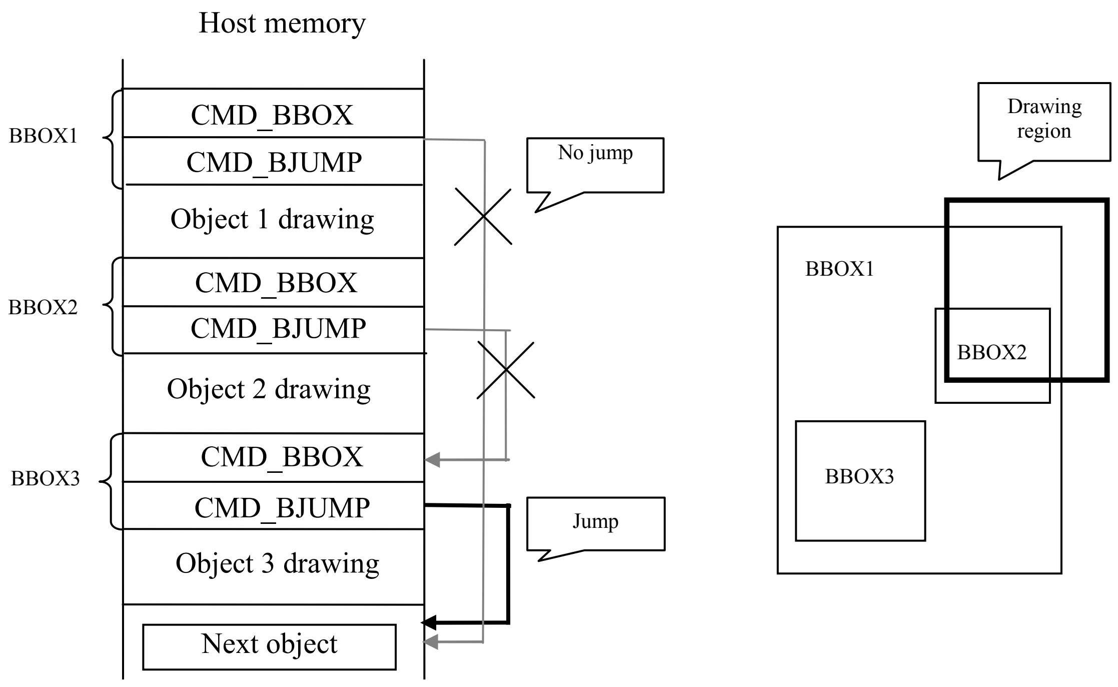


Figure 11-2 Display List Hierarchy Example 1

In Figure 11-3, Object 1 is not drawn since BBOX1 is outside the drawing region. And in this hierarchical structure, the objects of BBOX2 and BBOX3 are culled automatically. That is, a jump to the next object is executed after checking BBOX1.

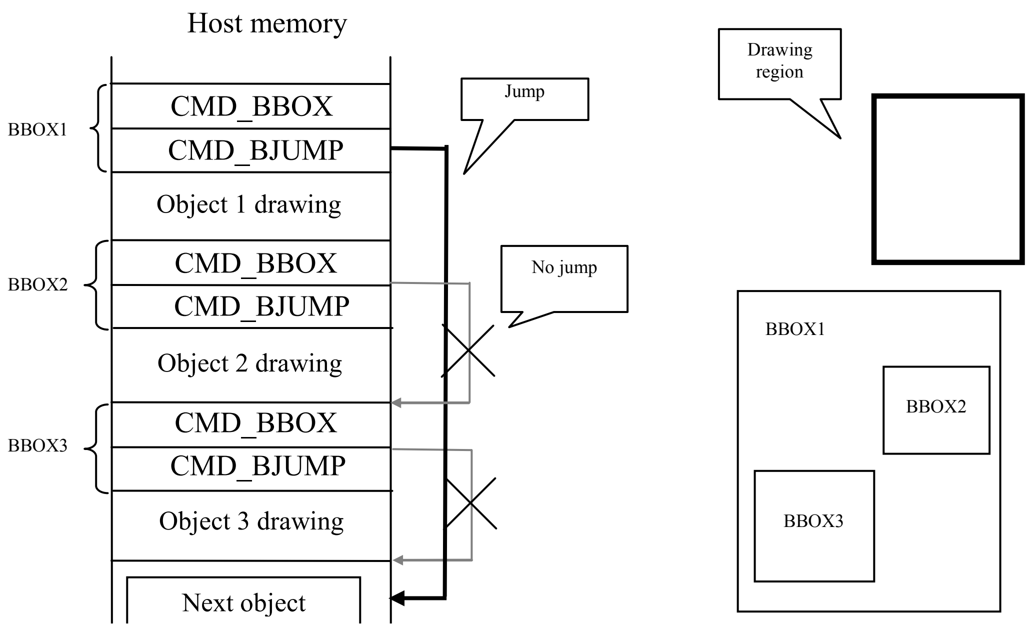


Figure 11-3 Display List Hierarchy Example 2

Figure 11-4 shows an example of object culling. In this case, a drawing region has three objects specified in bounding boxes, which are a body, a left hand and a left foot. The objects are drawn and the objects outside the drawing region are culled.

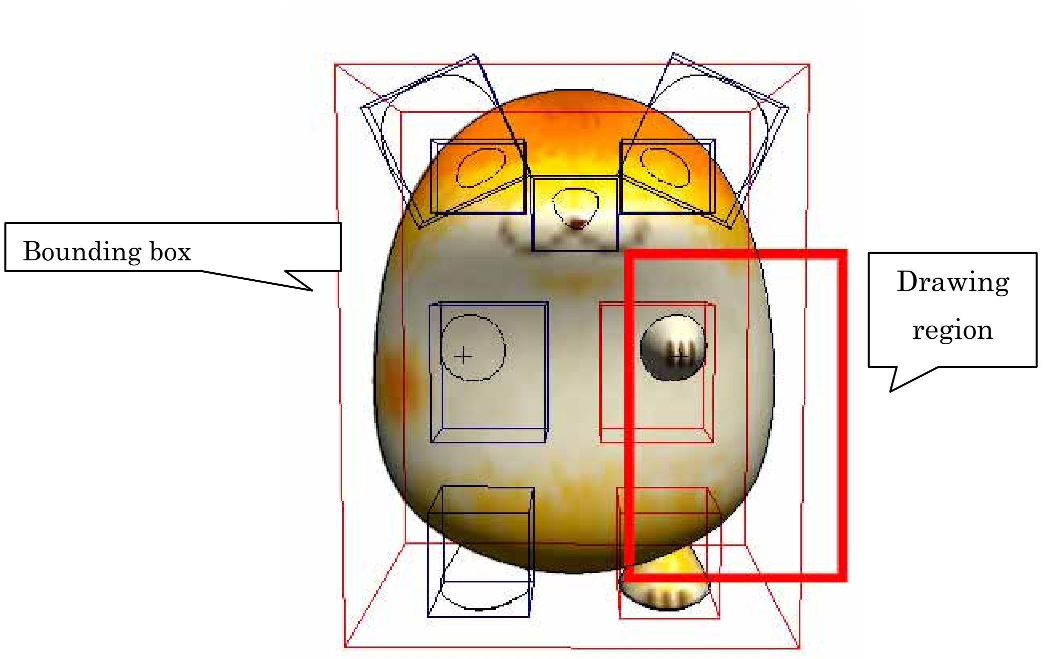


Figure 11-4 Object Culling

11.2. Patch Culling

With the Graphics Engine it is possible to draw patches in a drawing region and to cull patches outside the drawing region. This enables you to omit processing after dividing a curved surface. Patch culling can be enabled and disabled using the Patch Culling Enable command (CMD_PCE). To specify the drawing region, use the Drawing Region commands (CMD_REGION1, CMD_REGION2) just like with object culling.

Figure 11-5 shows an example of patch culling. The patches outside the drawing region are culled.

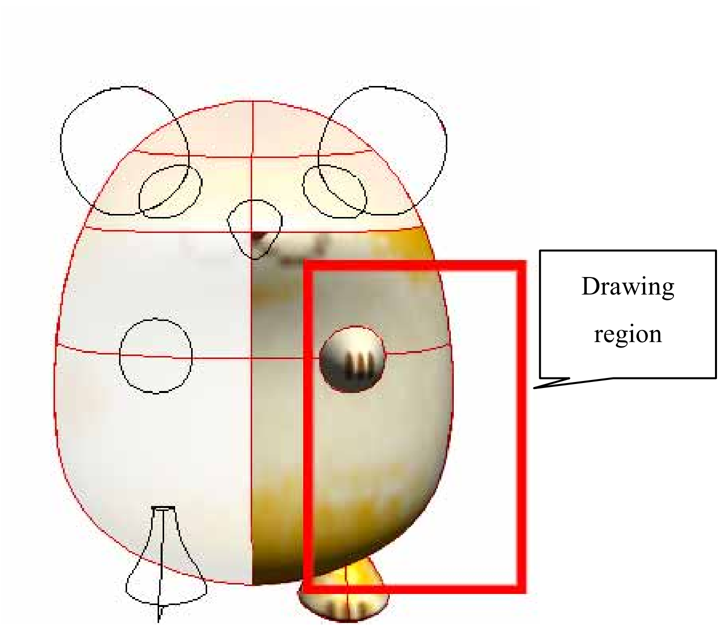


Figure 11-5 Patch Culling

11.3. Back Face Culling

The Graphics Engine allows you to omit the drawing of Triangles that are facing back. This is a technique known as back face culling and it saves you from having to do unnecessary drawing. Back face culling can be enabled and disabled using the Culling Enable command (CMD_BCE). In addition, using the Culling Surface command (CMD_CULL), you can select primitives to be culled whose vertices are arranged either clockwise or counterclockwise with respect to the view point.
Note that Points, Lines and Rectangles are always drawn regardless of whether back face culling is enabled or disabled.

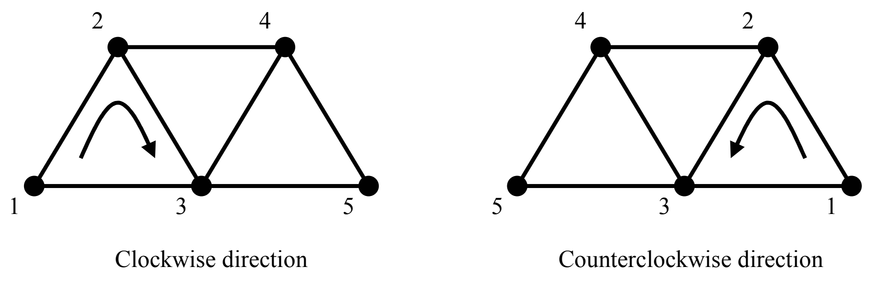


Figure 11-6 Culling of Triangles

12. Pixel Processing Flow

Figure 12-1 shows the pixel processing flow in the Graphics Engine. The figure shows how the generated X, Y, Z coordinates, primary color (Cp), secondary color (Cs), and texture coordinates (S,T,Q) are processed. This is described in more detail in the following sections.

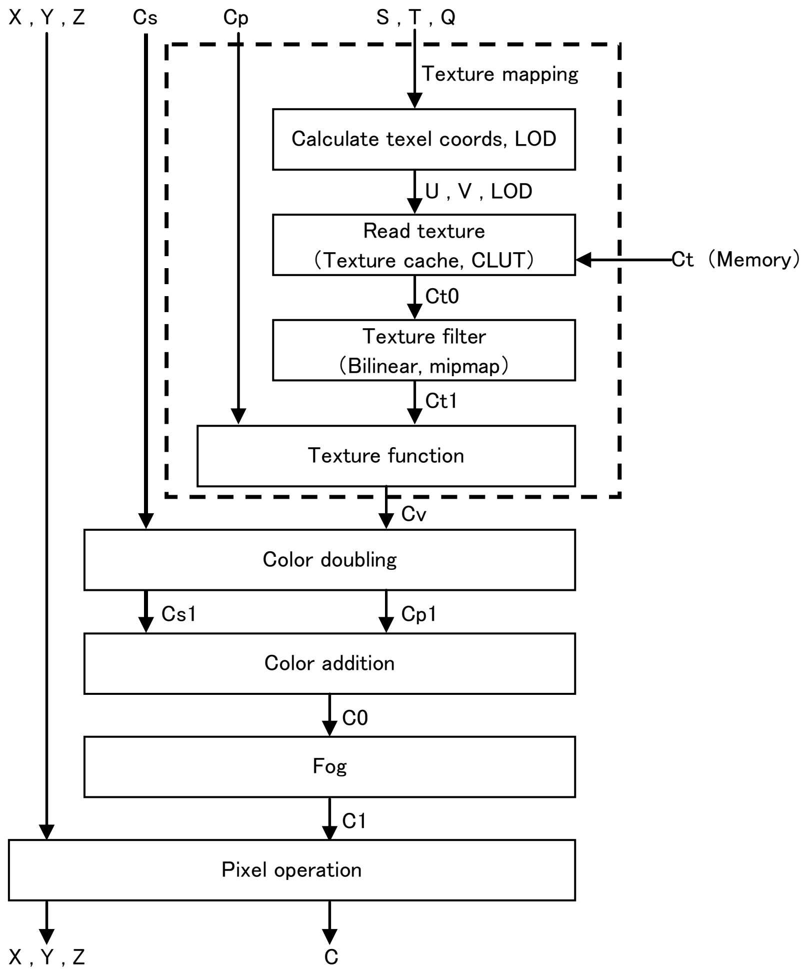


Figure 12-1 Pixel Processing Flow

13. Texture Mapping

The Graphics Engine can perform texture mapping with perspective correction by using texture data that is stored in main memory or VRAM.

13.1. Texture Mapping Modes

The Graphics Engine enables you to select one of the following three mapping modes as a method of applying a texture image when performing texture mapping.

13.1.1. UV mapping

With UV mapping, you can calculate a texel coordinate (U, V) by using a texture coordinate (S, T, 1) in vertex data. If a drawing primitive is Bezier or spline, it is possible to generate a texture coordinate (S, T) automatically, even if the texture coordinate (S, T) is not included in vertex data.
As shown in Figure 13-1, S and T are generated automatically in accordance with the number of patches for each U direction and V direction. If there are multiple patches, the values of S and T might each be beyond the range of 0.0 ~ 1.0. In this case, in order to map exactly one image, it would be necessary to perform texture scaling using the Texture Scale commands (CMD_SU, CMD_SV).

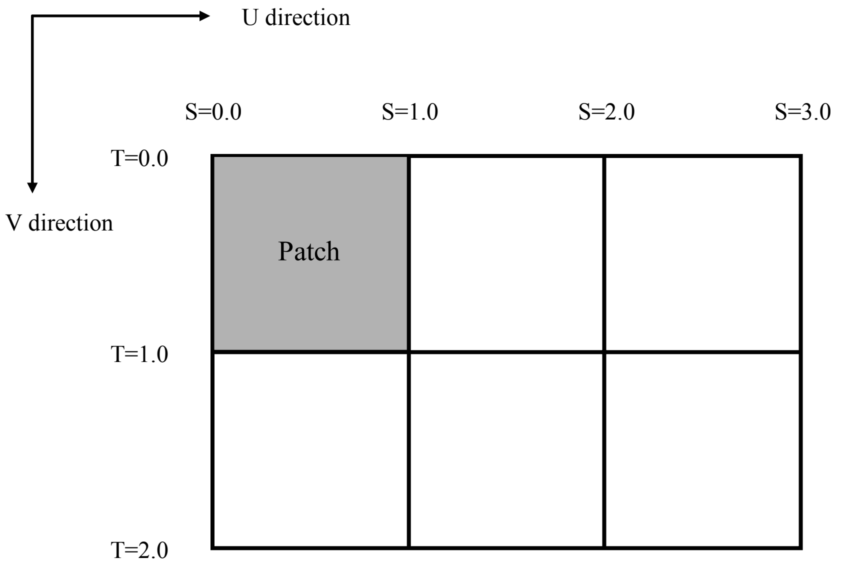


Figure 13-1 Automatic Generation of S and T for a Patch

13.1.2. Projection mapping

The Graphics Engine can generate texture coordinates (S,T,Q) from model coordinates by applying the texture generation matrix (Mt00-Mt11) as shown in the equation below. Note that if the texture coordinate (S,T,1) is included in vertex data, its value will be ignored. This function implements projection mapping in which a texture is projected onto an object.
The texture generation matrix can be specified using the Texture Generation Matrix Number command (CMD_TGENN) and the Texture Generation Matrix Data command (CMD_TGEND).

$$\begin{bmatrix} S \\ T \\ Q \end{bmatrix} = \begin{bmatrix} Mt00 & Mt03 & Mt06 \\ Mt01 & Mt04 & Mt07 \\ Mt02 & Mt05 & Mt08 \end{bmatrix} \begin{bmatrix} Xm \\ Ym \\ Zm \end{bmatrix} + \begin{bmatrix} Mt09 \\ Mt10 \\ Mt11 \end{bmatrix}$$

Equation 13-1 Projection Mapping

In addition, the Texture Map Mode command (CMD_TMAP) enables matrix calculations to be performed using the following values instead of vertex coordinates.

- Texture coordinates (U, V, 0)
- Normalized normal coordinates (Nx/|N|, Ny/|N|, Nz/|N|)
- Non-normalized normal coordinates (Nx, Ny, Nz)

The above coordinates are generated after patch division for spline surfaces and Bezier surfaces. They will become the coordinates of vertices for points, lines, triangles, and rectangles. Although the above coordinates are always generated during patch division for curved surfaces, for non-curved surfaces, there are cases when the coordinates may not be those of vertices and for those cases, results are not guaranteed.

13.1.3. Shade mapping

The Graphics Engine has a built-in shade mapping function which performs texture mapping by using values of light source calculations. When shade mapping is selected, texture coordinates (S, T) are determined from the calculated light source value of each selected light source set with the Shade Mapping command (CMD_TSHADE). The values of the texture coordinate can be obtained from the equation below, with Pu and Pv being the normalized dot product values for normal vector "N" and light source vector "L" (or "H" in case of specular light). Even if the texture coordinate (S, T) is included in vertex data, its value will be ignored.

$$\boldsymbol{S} = (\boldsymbol{Pu} + 1)/2$$
$$\boldsymbol{T} = (\boldsymbol{Pv} + 1)/2$$

P: Intensity of light source determined from lighting calculation

u, v: Numbers of the two light sources set with the Shade Mapping command (CMD_TSHADE)

For diffuse light

$$\boldsymbol{Pi} = \frac{\boldsymbol{L} \bullet \boldsymbol{N}}{|\boldsymbol{L}||\boldsymbol{N}|}$$

For specular light

$$\boldsymbol{H} = \frac{\boldsymbol{E}}{|\boldsymbol{E}|} + \frac{\boldsymbol{L}}{|\boldsymbol{L}|}$$

$$\boldsymbol{Pi} = \left(\frac{\boldsymbol{H} \bullet \boldsymbol{N}}{|\boldsymbol{H}||\boldsymbol{N}|}\right)^{\boldsymbol{K}}$$

(*) i is the light number set for u,v (See 9. "Lighting")

Equation 13-2 Shade Mapping

By using this function, shading can be indicated clearly for a light when a texture is attached as shown in Figure 13-2 (Toon Shading).

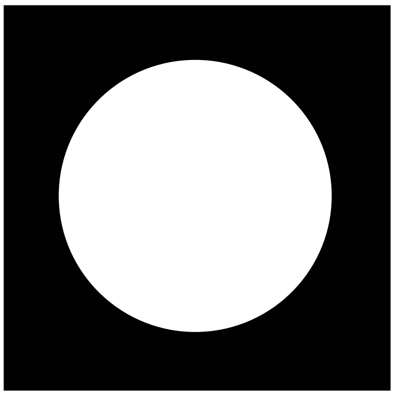


Figure 13-2 Texture 1 for Shade Mapping

In addition, if you set a light source vector for shade mapping along a coordinate axis such as (1.0, 0.0, 0.0) and (0.0, 1.0, 0.0) and apply a texture where a fish-eye lens is used as shown in Figure 13-3, you can produce a reflection effect like a mirror (Environment Mapping).

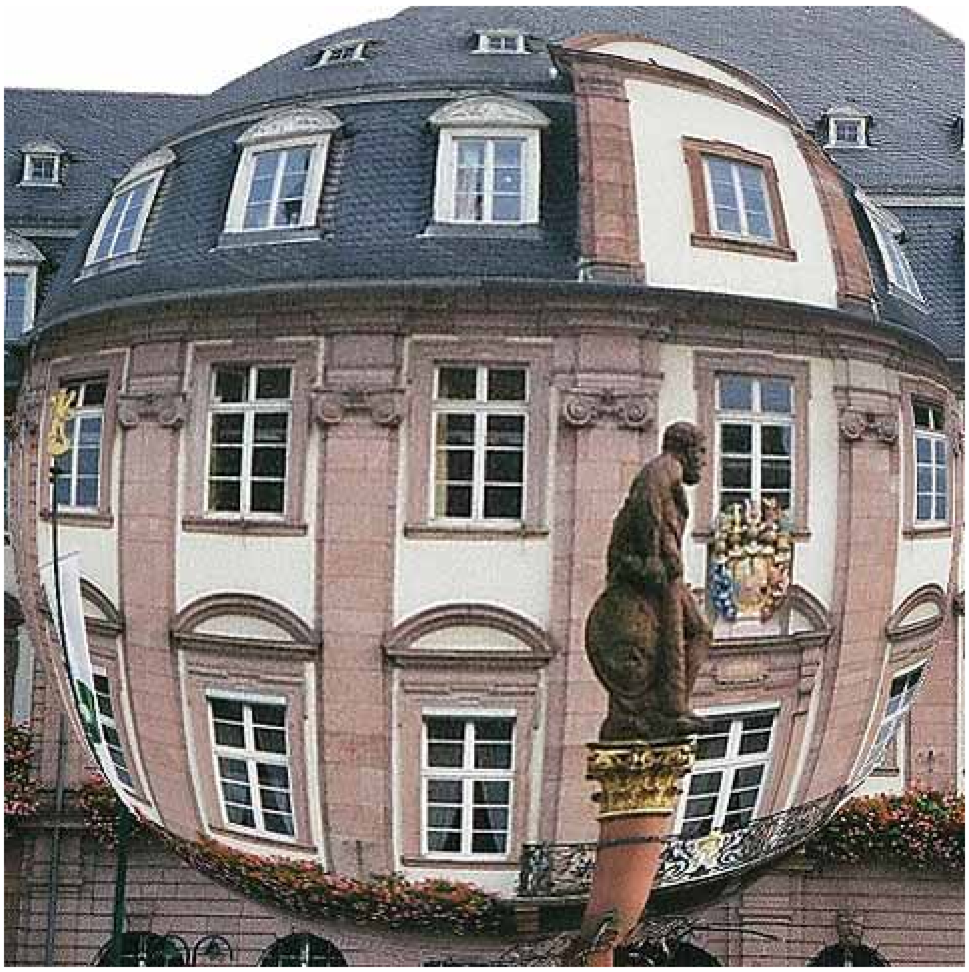


Figure 13-3 Texture 2 for Shade Mapping

13.2. Texture Scale / Offset

A texture coordinate (S, T) determined by the texture mapping mode is scaled or offset with values set by the Texture Scale commands (CMD_SU, CMD_SV) and the Texture Offset commands (CMD_TU, CMD_TV) as shown in Equation 13-3.

$$S' = SU \times S + TU$$
$$T' = SV \times T + TV$$

Equation 13-3 Texture Scale / Offset

Note that this process is only performed in UV mapping mode.

13.3. Texture Data

The Graphics Engine uses texture data stored in either host memory or local memory to perform texture mapping after perspective correction.

13.3.1. Pixel formats for textures

The Graphics Engine supports 8 different pixel formats for textures as shown below. The pixel format can be set using the Texture Pixel Format command (CMD_TPF).

- 16-bit 5:6:5:0 color format
- 16-bit 5:5:5:1 color format
- 16-bit 4:4:4:4 color format
- 32-bit 8:8:8:8 color format
- 4-bit index color format
- 8-bit index color format
- 16-bit index color format
- 32-bit index color format
- DXT1
- DXT3
- DXT5

The details of each color format are shown below.

16-bit 5:6:5:0 color format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B					G						R				

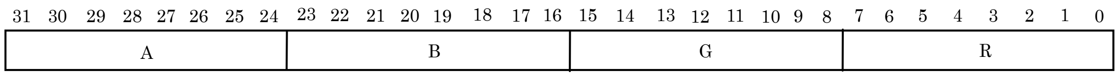
16-bit 5:5:5:1 color format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A	B					G					R				

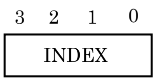
16-bit 4:4:4:4 color format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A				B				G				R			

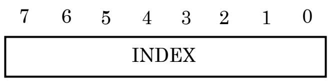
32-bit 8:8:8:8 color format



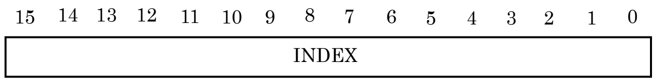
4-bit index color format



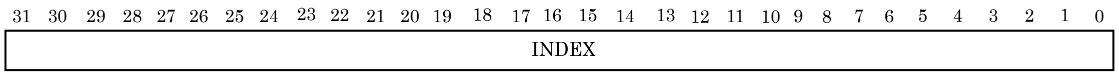
8-bit index color format



16-bit index color format



32-bit index color format

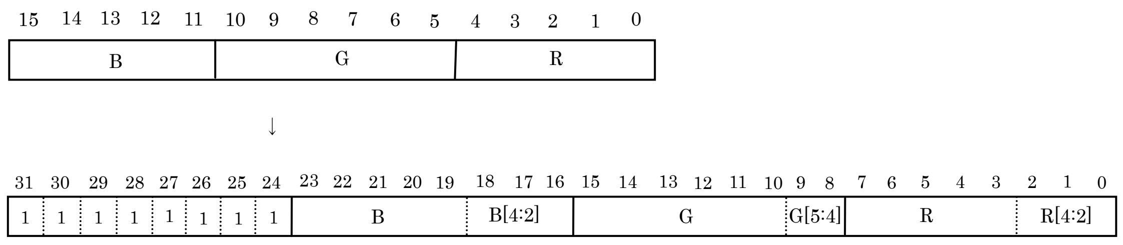


DXT1 / DXT3 / DXT5

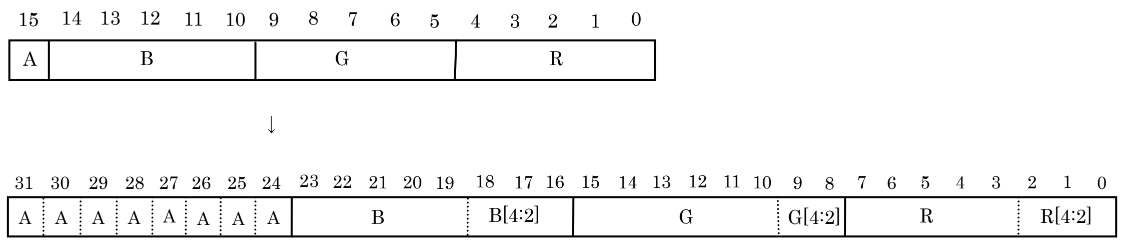
These are the formats when using texture compression. Details are presented in 13.3.2, “Texture Compression Formats.”

For index color, the CLUT (Color Lookup Table) which is described later, is used to convert to 32-bit (8:8:8:8) color or 16-bit (5:6:5:0, 5:5:5:1, 4:4:4:4) color formats. Furthermore, 16-bit color is expanded to 32-bit (8:8:8:8) color as shown below, and regardless of which color format is used, texture mapping is always performed at the end using 32-bit color.

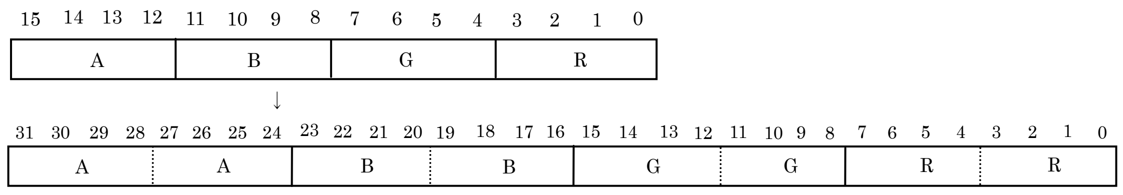
16-bit 5:6:5:0 color format



16-bit 5:5:5:1 color format



16-bit 4:4:4:4 color format



13.3.2. Texture Compression Formats

If DXT1, DXT3, and DXT5 are selected as the pixel format of a texture, the texture is compressed and stored in the buffer. Figure 13-4 Texture Compression

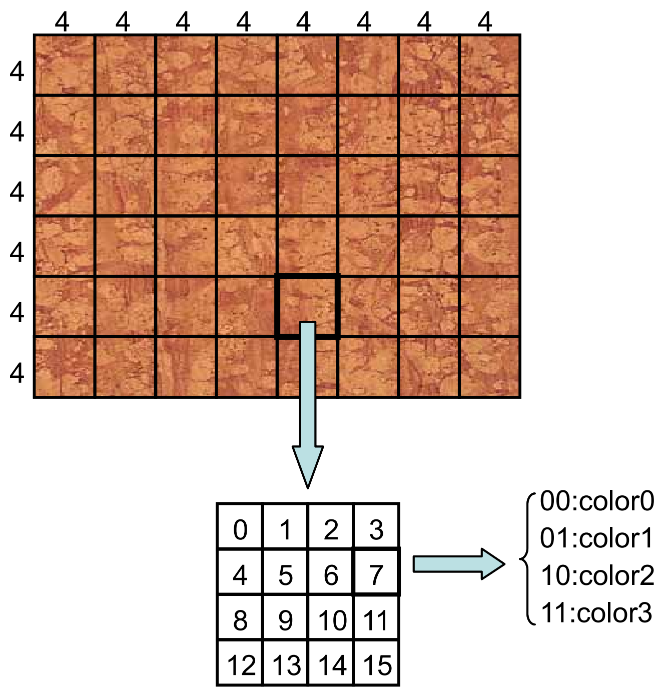


Figure 13-4 Texture Compression

DXT1 Format

The DXT1 format is shown below.

63 — 48	47 — 32	31 — 0
COLOR1	COLOR0	TEX15-0

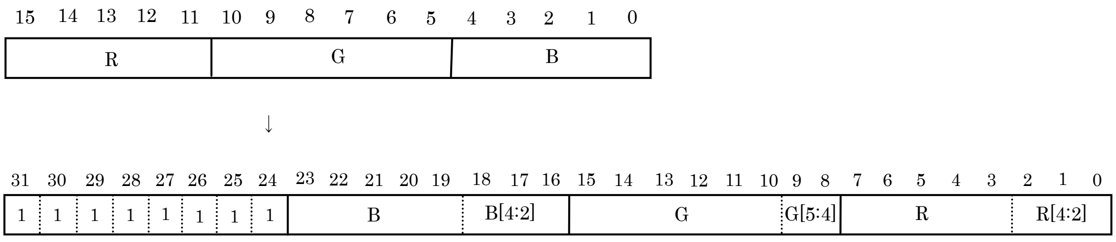
Here, TEX15-0 are the respective 2-bit indexes of texels 0 to 15 in the 4x4 texel block in Figure 13-4 Texture Compression

and correspond to the following.

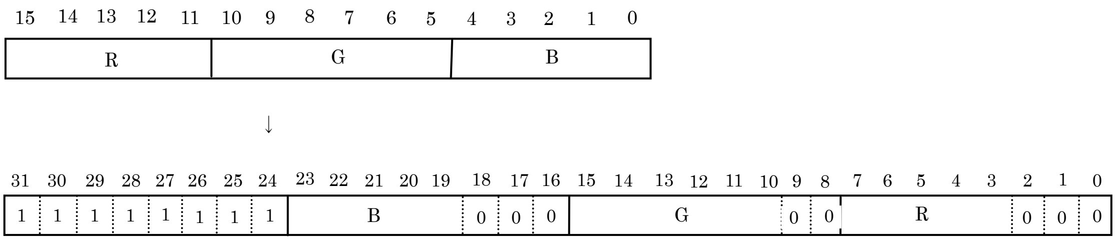
Bits 1 – 0:	TEX0
Bits 3 – 2:	TEX1
Bits 5 – 4:	TEX2
Bits 7 – 6:	TEX3
	:
Bits 29 – 28:	TEX14
Bits 31 – 30:	TEX15

COLOR0 and COLOR1 are the 2-bit indexes shown in Figure 13-4 Texture Compression and are color values corresponding to 00 and 11. The pixel format is set to be 16-bit 5:6:5:0 format, and is expanded to 32-bit 8:8:8:8 format prior to calculating COLOR2 and COLOR3, described below. Using the texture pixel format command (CMD_TPF), the expansion method can be selected from among the following two methods.

- With color expansion



- Without color expansion



COLOR2 and COLOR3 in Figure 13-4 are found by interpolating COLOR0 and COLOR1, and are calculated using the following two modes via transparent color processing.

- Non-transparent mode
  Non-transparent mode is when COLOR0 > COLOR1. In this case, COLOR2 and COLOR3 both become non-transparent colors and, as shown in Equation 13-4 each color value is calculated in RGB units (C0, C1, C2, C3), and the result is the respective RGB 8-bit color. The A values will be 255 for both A2 and A3.

$$
\begin{aligned}
C2 &= (2 \times C0 + C1)/3 \\
C3 &= (C0 + 2 \times C1)/3 \\
A2 &= 255 \\
A3 &= 255
\end{aligned}
$$

Equation 13-4 DXT1 Non-Transparent Mode

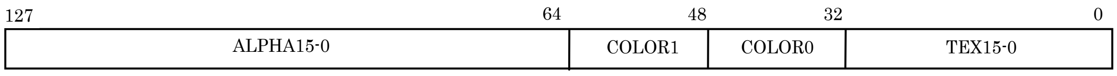
- Transparent mode
  Transparent mode is when COLOR0 ≤ COLOR1. In this case, COLOR3 becomes a transparent color and, as shown in Equation 13-5 each color and the A value are calculated and the result is the respective RGBA 8-bit color.

$$C2 = (C0 + C1)/2$$
$$C3 = 0$$
$$A2 = 255$$
$$A3 = 0$$

Equation 13-5 DXT1 Transparent Mode

DXT3 Format

The DXT3 format is shown below.

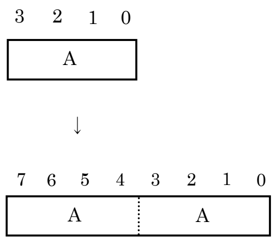


DXT3 handles semi-transparent A values. A values are 4 bits, and the correspondence with texels 0 to 15 is shown below.

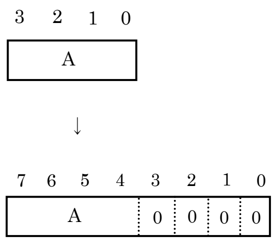
Bits 67 - 64:	ALPHA0
Bits 71 - 68:	ALPHA1
Bits 75 - 72:	ALPHA2
Bits 79 - 76:	ALPHA3
	:
Bits 123 – 120:	ALPHA14
Bits 127 – 124:	ALPHA15

A values are expanded by the texture pixel format command (CMD_TPF) as follows.

- With A expansion



- Without A expansion



For color expansion, calculation is performed in the same manner as for DXT1, using bits 63–0.

DXT5 Format

The DXT5 format is shown below.

127	120	112	64	48	32 0
A1	A0	ALPHA15-0	COLOR1	COLOR0	TEX15-0

In DXT5, data (ALPHA15-0) is held which indexes the A values which have been generated by interpolating from two A values (A0, A1) in the same manner as for color values. Indexes are 3 bits, and the correspondence with texels 0 to 15 is shown below.

Bits 66 – 64:	ALPHA0
Bits 69 – 67:	ALPHA1
Bits 72 – 70:	ALPHA2
Bits 75 – 73:	ALPHA3
	:
Bits 108 – 106:	ALPHA14
Bits 111 – 109:	ALPHA15

There are two A value generation modes, as follows.

- Uniform interpolation mode
  Uniform interpolation mode is when ALPHA0 > ALPHA1. In this method, six A values are generated by uniformly interpolating from A0 and A1. Equation 13-6 shows how the values are generated.

$$
\begin{aligned}
A2 &= (6\times A0+1\times A1)/7\\
A3 &= (5\times A0+2\times A1)/7\\
A4 &= (4\times A0+3\times A1)/7\\
A5 &= (3\times A0+4\times A1)/7\\
A6 &= (2\times A0+5\times A1)/7\\
A7 &= (1\times A0+6\times A1)/7
\end{aligned}
$$

Equation 13-6 DXT5 Uniform Interpolation Mode

- 0 / 255 mode
  0 / 255 mode is when ALPHA0 ≤ ALPHA1. In this method, completely transparent (A=0) and non-transparent (A=255) are provided, and the remaining four values are found by interpolating from A0 and A1. Equation 13-7 shows how the values are generated.

$$
\begin{aligned}
A2 &= (4\times A0 + 1\times A1)/5\\
A3 &= (3\times A0 + 2\times A1)/5\\
A4 &= (2\times A0 + 3\times A1)/5\\
A5 &= (1\times A0 + 4\times A1)/5\\
A6 &= 0\\
A7 &= 255
\end{aligned}
$$

Equation 13-7 DXT5 0 / 255 Mode

For color values, calculation is performed in the same manner as for DXT1, using bits 63–0.

13.3.3. Texture Buffer

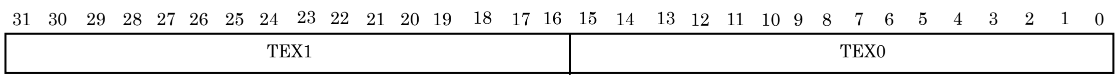
The texture buffer is the buffer which stores textures. Settings are specified using the texture buffer commands (CMD_TBP0-7 and CMD_TBW0-7). Eight texture buffers can be specified which correspond to the texture levels described below. The base point and width of each of the buffers is specified and must be in units of 16 bytes. As a result, the 32-bit (8:8:8:8, indexed) color format can be specified as a multiple of 4. With color formats of less than 32 bits, storage to the buffer is performed as follows, and the buffer width is also constrained to multiples of 8, 16, and 32.

32-Bit Color Format

The buffer width must be a multiple of 4.

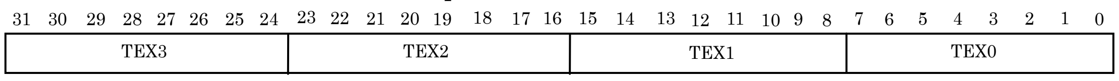
16-Bit Color Format

The buffer width must be a multiple of 8.



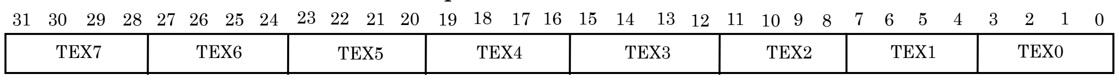
8-Bit Color Format : Multiple of 16

The buffer width must be a multiple of 16.



4-Bit Color Format : Multiple of 32

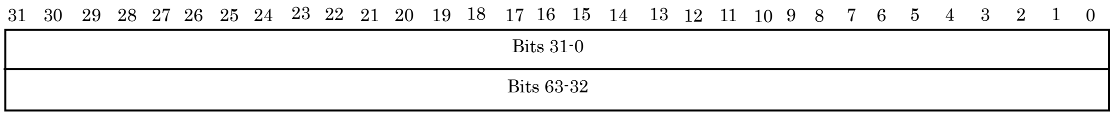
The buffer width must be a multiple of 32.



DXT1 Format :　Multiple of 8

The buffer width must be a multiple of 8.

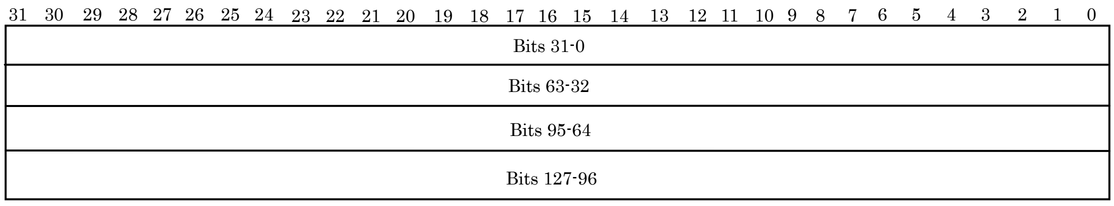
64-bit data is stored as bits 31-0 and 63–32, starting with the smallest address.



DXT3 Format :　Multiple of 4

The buffer width must be a multiple of 4.

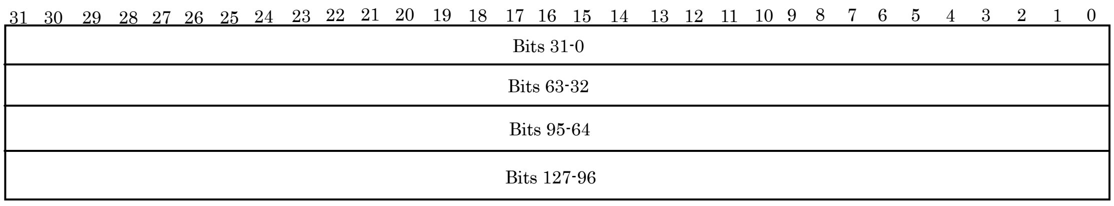
128-bit data is stored as bits 31-0, 63-32, 95-64 and 127-96, starting with the smallest address.



DXT5 Format :　Multiple of 4

The buffer width must be a multiple of 4.

128-bit data is stored as bits 31-0, 63-32, 95-64 and 127-96, starting with the smallest address.



13.4. Texture Size

The smallest texture that can be handled by the Graphics Engine is 1x1 and the largest is 512x512. In addition, the width and height of a texture must be specified as values of 2 to the nth power. The texture size can be set with the Texture Size commands (CMD_TSIZE0-7).

13.5. Texture Cache

The Graphics Engine, which has an internal texture cache for accelerating texture mapping, can handle texture sizes of up to 512x512.

13.5.1. Texture cache structure

The texture cache is an 8 KB, 4-way set associative cache. Replacement is performed in rectangular units 16 bytes (X) by 8 pixels (Y). The replacement algorithm is LRU (Least Recently Used).
For each pixel mode, Figure 13-5 shows examples of texture sizes that fit in the texture cache without causing cache misses. Since the cache is 4-way set associative, other combinations of width and height from those shown in Figure 13-5 can also be selected.
In addition, when the Texture Mode command (CMD_TMODE) is used to set the maximum texture level (MXL) to a value of 1 or greater, half the cache will be allocated to the even texture levels (LOD) and half to the odd texture levels. As a result, the size of the texture that can fit in the texture cache will be reduced.
Table 13-1 shows combinations of texture sizes that fit in the cache.

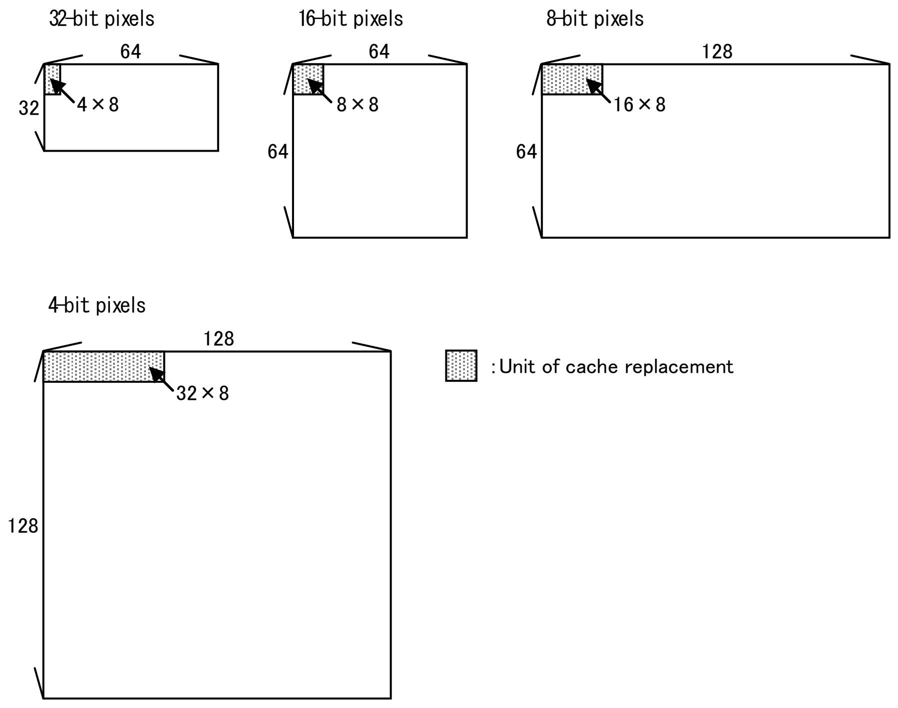


Figure 13-5 Examples of On-Cache Texture Sizes

	MXL = 0	MXL > 0
32-bit pixels	128×16 64×32 32×64	64×16 32×32 16×64
16-bit pixels	128×32 64×64 32×128	128×16 64×32 32×64
8-bit pixels	256×32 128×64 64×128	128×32 64×64 32×128
4-bit pixels	256×64 128×128 64×256	256×32 128×64 64×128

Table 13-1 Combinations of On-Cache Texture Sizes

13.5.2. Texture storage modes

The following two modes are available for storing textures in the texture buffer. The choice of storage mode affects the replacement performance in the texture cache.
The Texture Mode command (CMD_TMODE) is used to set the storage mode.

Normal mode

In Normal mode, texture data is stored linearly in the texture buffer.
For example,
Figure 13-6 shows how 32-bit color textures are stored that have a width of 16 pixels and a height of 16 pixels.
For color formats less than 32 bits, please see the "Texture Buffer" section which describes how multiple pixels are stored in one 32-bit color pixel. In addition, for texture compression formats, 4x4-unit textures which are represented by 64 bits or 128 bits are stored in a linear texture buffer.

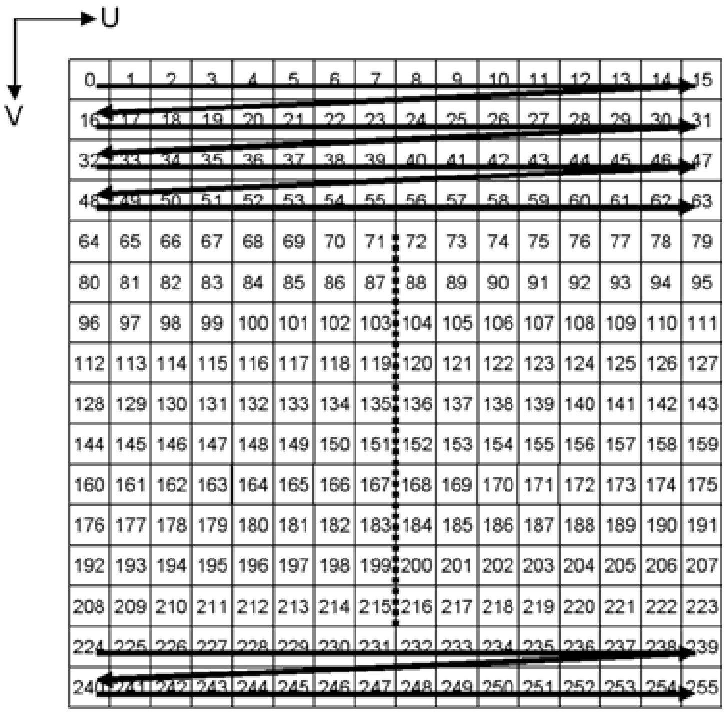


Figure 13-6 Normal Mode

Fast mode

In Fast mode, texture data is stored in a way that is best suited to the texture cache. Specifically, texture data with a width of 16 bytes and a height of 8 pixels is stored in the texture buffer, which is the unit of replacement for the texture cache.
Figure 13-7 shows an example of storing 16-pixel wide, 16-pixel high, 32-bit color textures.
For color formats less than 32 bits, please see the "Texture Compression Formats" section which describes how multiple pixels are stored in one 32-bit color pixel. Moreover, for texture compression formats, this mode is not supported.
When using Fast mode, normal operation is not possible when the texture width and texture buffer width are different. When Fast mode is used, make sure the texture width and texture buffer width are the same.

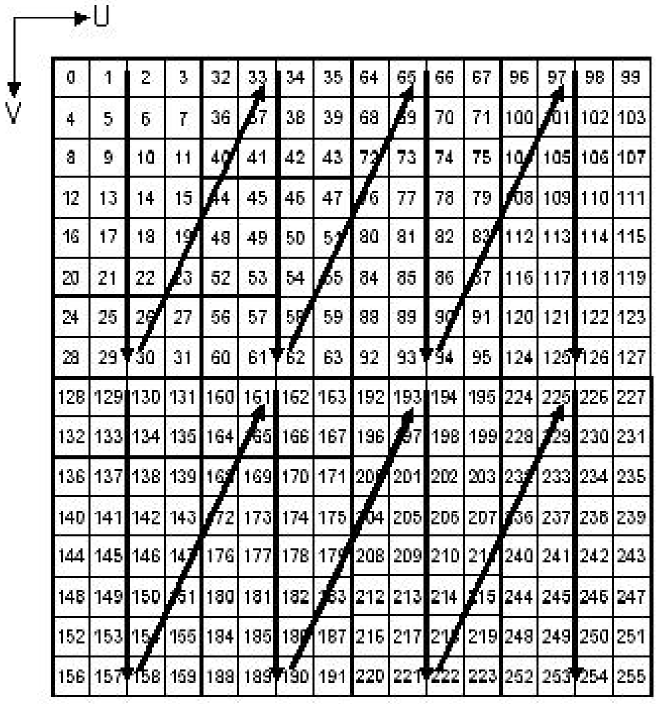


Figure 13-7 Fast Mode

13.6. CLUT (Color Lookup Table)

When 4, 8, 16 and 32-bit index color formats are used as the pixel format in the Graphics Engine, the index color is converted into 16-bit or 32-bit RGBA using the CLUT.

13.6.1. CLUT pixel formats

Index color is converted into the color formats listed below using the CLUT. The color format can be selected using the CLUT command (CMD_CLUT).

- 16-bit 5:6:5:0 color format
- 16-bit 5:5:5:1 color format
- 16-bit 4:4:4:4 color format
- 32-bit 8:8:8:8 color format

The number of palettes that can be used for color conversion depends on the target color format. For the 16-bit color formats (5:6:5:0, 5:5:5:1, 4:4:4:4), 512 palettes are available, and for the 32-bit 8:8:8:8 color format, 256 palettes are available.

13.6.2. CLUT buffer

When the pixel format is index color, a CLUT buffer located in host memory is needed to perform texture mapping. The CLUT buffer contains a color conversion table, and index values that are stored in the texture buffer are used by the CLUT to convert colors to 16-bit or 32-bit color format. The CLUT buffer base point is set by the CLUT Buffer commands (CMD_CBP, CMD_CBW).
The CLUT size is normally 256 palettes when the texture pixel format is 8-bit index color, and 16 palettes when the texture pixel format is 4-bit index color. The CLUT is stored linearly in memory.
However, when multi-CLUT mode is set by the Texture Mode command (CMD_TMODE), CLUTs are stored with sizes corresponding to the required texture levels.
Before texture mapped drawing can be performed, the CLUT must be loaded in advance using the CLUT Load command.

13.6.3. CLUT loading

To use the CLUT, CLUT data must first be loaded from the CLUT buffer to the palettes. The starting position (CSA) and the number of palettes (NP) can be specified. NP is specified using the CLUT Load command (CMD_CLOAD).
For 16-bit color, loading is done in units of 16 palettes, and for 32-bit color, units of 8 palettes.

16-bit (5:6:5:0, 5:5:5:1, 4:4:4:4) color

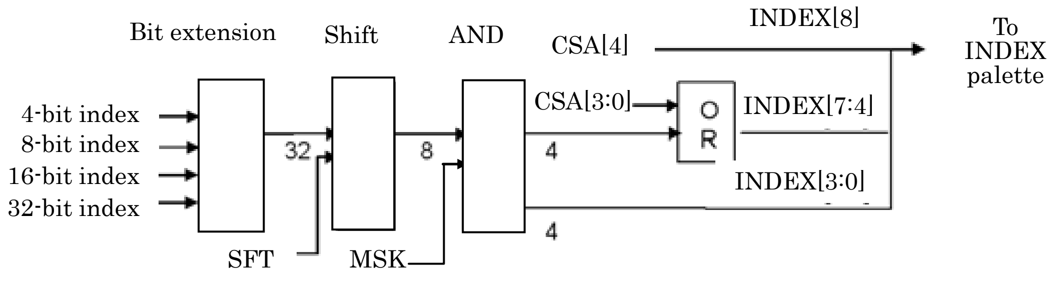
Number of loads = 16 palettes × NP

32-bit 8:8:8:8 color

Number of loads = 8 palettes × NP

13.6.4. CLUT index generation

Figure 13-8 shows index generation during CLUT loading. To generate an index, the values for SFT, MSK and CSA must be set using the CLUT command (CMD_CLUT).



Bit extension	: Zero-extend through the MSB to make 32 bits
Shift	: Shift right SFT, output the least significant 8 bits
AND	: Perform logical AND with MSK
OR	: Perform logical OR of CSA[3:0] with bits[7:4] of the AND output

Figure 13-8 Index Generation

The CSA is used in the same way as when the CLUT is loaded, and the starting position for reading is specified in 16-palette units.

Starting read position (palette number) = 16 × CSA

Note that to set the starting position in the CSA, only a logical OR is done with the AND output without performing an addition. Consequently, the MSK must be set to prevent bits set to 1 from overlapping with those in the CSA.

13.6.5. Multiple CLUTs

Multiple CLUTs can be used that correspond to different texture levels LOD. Multiple CLUTs can be set using the Texture Mode command (CMD_TMODE). The value set in the CSA will be used as the palette for level 0.

4-bit index color

For 4-bit index color, eight levels of CLUTs (Level 0~7) can be used as follows.

```
Level 0:   ( ((CSA + 0) × 16) ~ ((CSA + 0) × 16 + 15) )
Level 1:   ( ((CSA + 1) × 16) ~ ((CSA + 1) × 16 + 15) )
                          :
                          :
Level n:   ( ((CSA + n) × 16) ~ ((CSA + n) × 16 + 15) )
                          :
```

Level 7: (((CSA + 7) × 16) ~ ((CSA + 7) × 16 + 15))

8-bit index color

For 8-bit index color, two levels of CLUTs (Level 0~1) can be used as shown below. Note that since 512 palettes are used for the two levels of CLUTs, a 16-bit color format (5:6:5:0, 5:5:5:1, 4:4:4:4) must be used for the CLUT pixel format.

Level 0: (((CSA + 0) × 16) ~ ((CSA + 0) × 16 + 255))
Level 1: (((CSA + 16) × 16) ~ (CSA + 16) × 16 + 255))

13.7. Texture Flush

After changing and switching texture data, it is necessary to flush any texture data residing in the cache using the Texture Flush command (CMD_TFLUSH).
If you do not flush the cache, texture mapping might get executed improperly by using previous texture data in the cache.

13.8. Texture Synchronization

With the Graphics Engine, you can perform recursive drawing and use a drawing result as a texture. However, you must synchronize with the completion of drawing, otherwise texture mapping might get executed improperly using data that has not yet been rendered. When performing recursive drawing, you must synchronize with the completion of drawing. This can be done using the Texture Sync command (CMD_TSYNC).

13.9. Texture Size

The texture size that the Graphics Engine can handle is 1x1 at the minimum to 512x512 at the maximum. The width and height of a texture must be defined as values of 2 to the nth power.

13.10. Texture Wrap Mode

Each texture coordinate (S, T) is stored within the range [0,1] depending on the texture wrap mode. Texture wrap mode is controlled using the Texture Wrap Mode command

(CMD_TWRAP).

REPEAT

Ignore the integer part of a texture coordinate.

For example, a texture coordinate will become 0.5 if it is 1.5.

CLAMP

Clamp a texture coordinate to [0,1].

For example, a texture coordinate will become 1.0 if it is 1.5.

13.11. Calculation of Texture Level

To calculate texture level, an auto mode and a constant mode are provided. The following equation is used to calculate the texture level, and the level is eventually clamped within the range of 0 to the maximum. The calculation mode, maximum level and level offset can be set using the Texture Level Mode (CMD_TLEVEL), Texture Slope (CMD_TSLOPE) and Texture Mode commands (CMD_TMODE).

Variable mode1

$$\text{Level} = \log_2\left(\frac{\max\left(TW_0\left|\frac{dS}{dXs}\right|, TW_0\left|\frac{dS}{dYs}\right|, TH_0\left|\frac{dT}{dXs}\right|, TH_0\left|\frac{dT}{dYs}\right|\right)}{|Q|}\right) + \text{OFFL}$$

Variable mode2

$$\text{Level} = \log_2\left(\frac{|TSLOPE|}{|Q|}\right) + \text{OFFL}$$

Fixed mode

Level＝OFFL

＊TW_0	: Width of the texture at zero level
＊TH_0	: Height of the texture at zero level
＊OFFL	: Texture level offset

13.12. Texture Filtering

For texture filtering, two types of filters are provided to magnify and reduce. You can switch between filters depending on the level. If the texture level is zero, select a magnifying filter. If the texture level is greater than zero, select a reducing filter. Each filter has several modes to be selected as shown below. Note that the filter status may be switched at a level of 0.5 if the mode of the magnifying filter is LINEAR and the mode of the reducing filter is either NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR (the reducing filter is selected at a level of 0.5). Texture filtering can be set using the Texture Filter command (CMD_TFILTER).

Magnifying filter	NEAREST LINEAR
Reducing filter	NEAREST LINEAR NEAREST_MIPMAP_NEAREST LINEAR_MIPMAP_NEAREST NEAREST_MIPMAP_LINEAR LINEAR_MIPMAP_LINEAR

13.13. Texture Functions

For texture mapping, it is possible to execute texture functions as shown in the table below between a calculated texture color and a primary color.
The texture function operation is a calculation as shown in the table below and depends on the color components and function mode that are set. The texture function can be set using the Texture Function command (CMD_TFUNC). The subscripts "p", "t", "e" and "v" indicate the primary color, texture color, texture environment color and value calculated using the texture function, respectively. The texture environment color can be set using the Texture Environment Color command (CMD_TEC). The color value after calculation is clamped to the range of 0~255.
Note that the operator ◎ in the table means X◎Y＝ X×Y÷255.

Color Component	Function	Equation

RGB	MODULAT	Rv=Rt◎Rp
		Gv=Gt◎Gp
		Bv=Bt◎Bp
		Av=Ap

	DECAL	Rv=Rt
		Gv=Gt
		Bv=Bt
		Av=Ap

	BLEND	Rv=(255 - Rt)◎Rp + Rt◎Re
		Gv=(255 - Gt)◎Gp + Gt◎Ge
		Bv=(255 - Bt)◎Bp + Bt◎Be
		Av=Ap

	REPLACE	Rv=Rt
		Gv=Gt
		Bv=Bt
		Av=Ap

	ADD	Rv=Rt + Rp
		Gv=Gt + Gp
		Bv=Bt + Bp
		Av=Ap

RGBA	MODULAT	Rv=Rt◎Rp
		Gv=Gt◎Gp
		Bv=Bt◎Bp
		Av=At◎Ap

	DECAL	Rv=(255 - At)◎Rp + At◎Rt
		Gv=(255 - At)◎Gp + At◎Gt
		Bv=(255 - At)◎Bp + At◎Bt
		Av=Ap

	BLEND	Rv=(255 - Rt)◎Rp + Rt◎Re
		Gv=(255 - Gt)◎Gp + Gt◎Ge
		Bv=(255 - Bt)◎Bp + Bt◎Be
		Av=At◎Ap

	REPLACE	Rv=Rt
		Gv=Gt
		Bv=Bt
		Av=At

	ADD	Rv=Rt + Rp
		Gv=Gt + Gp
		Bv=Bt + Bp
		Av=At◎Ap

14. Color Doubling

The Graphics Engine has a primary color (Rp, Gp, Bp, Ap) and a secondary color (Rs, Gs, Bs). When texture mapping is enabled, a texture function is used to calculate (Rv, Gv, Bv, Av) for the primary color. When texture mapping is disabled, the primary color (Rp, Gp, Bp, Ap) is used directly for (Rv, Gv, Bv, Av).

Equation 14-1 and Equation 14-2 show the two sets of color components that are generated by color doubling.

$$\begin{aligned} Rp\_new &= Rv \\ Gp\_new &= Gv \\ Bp\_new &= Bv \\ Ap\_new &= Av \\ Rs\_new &= Rs \\ Gs\_new &= Gs \\ Bs\_new &= Bs \\ As\_new &= As \end{aligned}$$

Equation 14-1 Color Doubling (Disabled)

$$\begin{aligned} Rp\_new &= 2 \times Rv \\ Gp\_new &= 2 \times Gv \\ Bp\_new &= 2 \times Bv \\ Ap\_new &= Av \\ Rs\_new &= 2 \times Rs \\ Gs\_new &= 2 \times Gs \\ Bs\_new &= 2 \times Bs \\ As\_new &= As \end{aligned}$$

Equation 14-2 Color Doubling (Enabled)

15. Color Addition

A fragment has two colors: a primary color (Rp, Gp, Bp, Ap) and a secondary color (Rs, Gs, Bs). By adding these two color components, one RGBA color (R0, G0, B0, A0) can be produced. The color value produced is clamped to the range of 0~255. See Equation 15-1. Note that the subscript "v" indicates a value calculated using a texture function.
For your reference:
If texture mapping is enabled, (Rv, Gv, Bv, Av) is calculated with a texture function by using a primary color, a texture color, and a texture environment color. If texture mapping is disabled, then (Rv, Gv, Bv, Av) will be used as the primary color.

$$\begin{aligned} R_0 &= Rv + Rs \\ G_0 &= Gv + Gs \\ B_0 &= Bv + Bs \\ A_0 &= Av \end{aligned}$$

Equation 15-1 Color Addition

16. Fogging

The Graphics Engine has a fogging function that makes it possible to place a haze over a primitive that is positioned far from a view point.

When fogging is enabled using the Fog Enable command (CMD_FGE), a fog coefficient is calculated depending on the distance from a pixel to the view point. By using the fog coefficient and by blending with the equation below, fogging can be implemented.

Note that the operator ◎ below means X◎Y＝ X×Y÷255.

$$R = Ffrag \circledcirc Rfrag + (255 - Ffrag) \circledcirc Rfog$$
$$G = Ffrag \circledcirc Gfrag + (255 - Ffrag) \circledcirc Gfog$$
$$B = Ffrag \circledcirc Bfrag + (255 - Ffrag) \circledcirc Bfog$$

Equation 16-1 Blending with Fog Color

The fog coefficient is altered linearly between a fog start Z coordinate (Zstart) and fog end Z coordinate (Zend) as shown in Figure 16-1. These values can be set using the Fog Parameter commands (CMD_FOG1, CMD_FOG2). The values that are set must be calculated in advance using Zstart and Zend, as shown below.

$$FOG1 = -Zend$$
$$FOG2 = -\frac{1}{(Zend - Zstart)}$$

Equation 16-2 Calculation of Fog Parameters

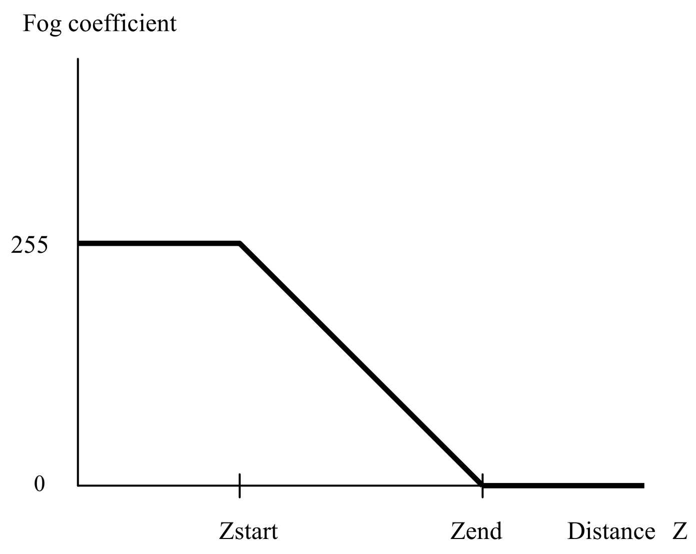


Figure 16-1 Relationship Between Fog Coefficient and Distance

17. Pixel Operations

In the Graphics Engine, pixel operations for texture-mapped pixel data are performed as needed in the following order.

Scissoring
↓
Depth Range Test
↓
Color Test
↓
Alpha Test
↓
Stencil Test
↓
Depth Test
↓
Alpha Blending
↓
Dithering
↓
Color Clamp
↓
Logical Operation
↓
Masking

17.1. Scissoring

A scissoring area should be set up as a given rectangular area in drawing coordinates. As shown in Figure 17-1, the area is specified with two points that are in the upper left and lower right corners, and no drawing is performed for pixels that are outside of this area. Note that pixels on the border are drawn. Since scissoring always takes place in the Graphics Engine, it is necessary to set a scissoring area before starting drawing. The scissoring area can be set using the Scissoring Area commands (CMD_SCISSOR1, CMD_SCISSOR2).

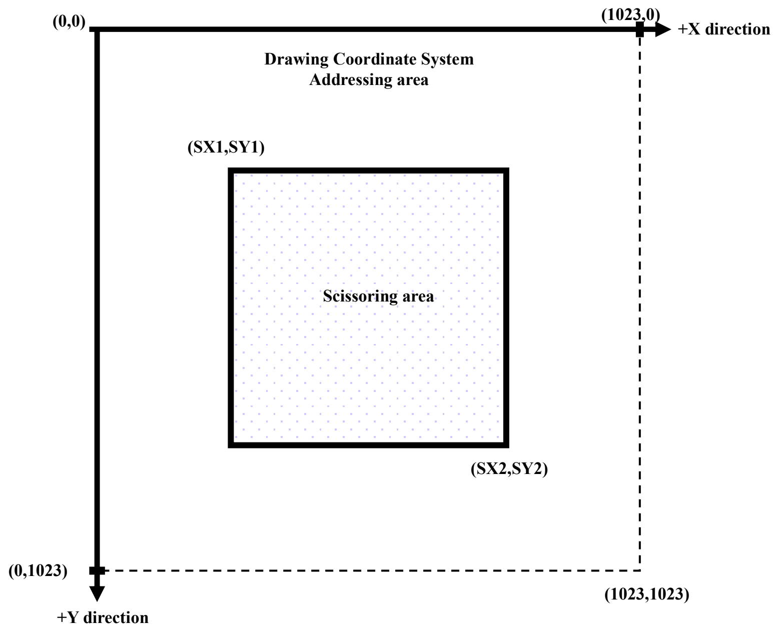


Figure 17-1 Scissoring

17.2. Depth Range Test

The depth range test sets minimum and maximum values for a depth value. No drawing is performed for pixels that are outside of this range. Since the depth range test is always performed in the Graphics Engine, it is necessary to set a depth range before starting drawing. However, when Through Mode is set, the depth range test is disabled. The depth range can be set using the Depth Range commands (CMD_MINZ, CMD_MAXZ).

17.3. Color Test

The color test allows drawing to be performed based on the result of comparing the color value of a pixel to a reference color value set with the Color Test Reference command (CMD_CREF). Both the color value and the reference value are ANDed with a mask before they are compared. The mask is set with the Color Test Mask command (CMD_CMSK). The color test is enabled using the Color Test Enable command (CMD_CTE). The color test parameter is set with the Color Test Parameter command (CMD_CTEST).

Each action of the Color Test Function (CTF) is shown in the following table.

CTF	Action
NEVER	Never draw any pixels
ALWAYS	Always draw all pixels
EQUAL	Draw pixels if pixel color = fixed value
NOTEQUAL	Draw pixels if pixel color != fixed value

17.4. Alpha Test

The alpha test allows drawing to be performed based on the result of comparing the alpha value of a pixel to a reference alpha value set by the Alpha Test Parameter command (CMD_ATEST). Both the pixel value and the reference value are ANDed with a mask before they are compared. The mask is set with the Alpha Test Parameter command (CMD_ATEST).
The alpha test is enabled using the Alpha Test Enable command (CMD_ATE). The alpha test parameter is set with the Alpha Test Parameter command (CMD_ATEST).
Each action of the Alpha Test Function (ATF) is shown in the following table.

ATF	Action
NEVER	Never draw any pixels
ALWAYS	Always draw all pixels
EQUAL	Draw pixels if pixel alpha value = fixed value
NOTEQUAL	Draw pixels if pixel alpha value != fixed value
LESS	Draw pixels if pixel alpha value < fixed value
LEQUAL	Draw pixels if pixel alpha value <= fixed value
GREATER	Draw pixels if pixel alpha value > fixed value
GEQUAL	Draw pixels if pixel alpha value >= fixed value

17.5. Stencil Test

If the stencil test is enabled, the decision to draw or not draw is based on the result of comparing a stencil value in the frame buffer to a preset value. If disabled, the stencil test always passes regardless of the stencil value in the frame buffer and control is passed to the next process.
The stencil test is enabled using the Stencil Test Enable command (CMD_STE). The stencil test parameter is set using the Stencil Test Parameter command (CMD_STEST). The stencil value is internally processed as 8 bits.
When the 5:6:5:0 color format is used, the stencil value in the frame buffer is assumed to be zero and it will not be updated. When the 5:5:5:1 color format is used, the stencil value is processed as 1 bit. After extending the value to 8 bits, only MSB bit 1 is valid. When the 4:4:4:4 color format is used, only MSB bit 4 is valid.

When the stencil test is performed, you can specify an 8-bit mask value to AND on both the preset value and the stencil value. This can be set using the Stencil Parameter command (CMD_STEST). When the 5:5:5:1 and 4:4:4:4 color formats are used, invalid stencil bits must be masked.

The actions of the Stencil Test Function (STF) are shown in the following table.

STF	Action
NEVER	Never draw any pixels
ALWAYS	Always draw all pixels
EQUAL	Draw pixels if stencil value = preset value
NOTEQUAL	Draw pixels if stencil value != preset value
LESS	Draw pixels if stencil value < preset value
LEQUAL	Draw pixels if stencil value <= preset value
GREATER	Draw pixels if stencil value > preset value
GEQUAL	Draw pixels if stencil value >= preset value

It is possible to execute the operations shown in the table below in the following situations: (1) if the stencil test fails; (2) if the stencil test passes and the depth test fails; and (3) if both the stencil and depth tests pass. The stencil test operation can be set using the Stencil Operation command (CMD_SOP).

Operation	Action
KEEP	Do nothing
ZERO	Change the stencil value to zero
REPLACE	Change the stencil value to the preset value
INVERT	Invert the stencil value
INCR	Increment the stencil value by one
DECR	Decrement the stencil value by one

17.6. Depth Test

The depth test allows drawing to be performed based on the result of comparing the depth value of a pixel to a depth value in the depth buffer. If the depth test is disabled, it always passes, but the depth buffer value is not updated. The depth test is enabled using the Depth Test Enable command (CMD_ZTE). The depth test parameter is set using the Depth Test Parameter command (CMD_ZTEST).

The actions of the Depth Test Function (DTF) are shown in the following table.

DTF	Action
NEVER	Never draw any pixels
ALWAYS	Always draw all pixels
EQUAL	Draw pixels if pixel depth value = depth value in depth buffer
NOTEQUAL	Draw pixels if pixel depth value != depth value in depth buffer
LESS	Draw pixels if pixel depth value < depth value in depth buffer

DTF	Action
LEQUAL	Draw pixels if pixel depth value <= depth value in depth buffer
GREATER	Draw pixels if pixel depth value > depth value in depth buffer
GEQUAL	Draw pixels if pixel depth value >= depth value in depth buffer

17.7. Alpha Blending

Alpha blending enables the drawing color value to be determined using the calculations shown below. The parameters of A and B can be combined freely as shown in the table below. FIXA and FIXB indicate values set with the Fixed Color A command (CMD_FIXA) and Fixed Color B command (CMD_FIXB), respectively. Note that the result is passed to dithering without clamping.

Alpha blending can be enabled and disabled by using the Alpha Blending Enable command (CMD_ABE). Alpha blending parameters can be set with the Alpha Blending Parameter command (CMD_BLEND).

The operator ◎ in the following equations means X◎Y＝X×Y÷255.

Mode	Equation
ADD	Cs◎A＋Cd◎B
SUBTRACT	Cs◎A－Cd◎B
REVERSE_SUBTRACT	Cd◎B－Cs◎A
MIN	min(Cs,Cd)
MAX	max(Cs,Cd)
ABS	abs(Cs,Cd)

Parameter	Options
A	FIXA, Cd, 255 - Cd, As, 255 - As, Ad, 255 – Ad, 2×As, 255 - 2×As, 2×Ad, 255 - 2×Ad
B	FIXB, Cs, 255 - Cs, As, 255 - As, Ad, 255 - Ad, 2×As, 255 - 2×As, 2×Ad, 255 - 2×Ad

As	: Alpha value of input pixel (source)
Cs	: Color value of input pixel (source)
Ad	: Alpha value in the frame buffer (destination)
Cd	: Color value in the frame buffer (destination)
FIXA,FIXB	: Fixed color A and B

17.8. Dithering

With the Graphics Engine, dithering can take place when drawing is done in the frame buffer. If dithering is enabled, a value, which is calculated from a dither matrix is added to the color value. (No addition is performed with the alpha value.) The dither matrix can be set using the Dither Coefficient commands (CMD_DTTH1-4). Dithering can be enabled using the Dithering Enable command (CMD_DTE).

R = R + MATRIX[dY][dX]
G = G + MATRIX[dY][dX]
B = B + MATRIX[dY][dX]

The dither matrix used in the Graphics Engine is shown below. You can set each value of the dithering coefficients (DM00-15) within the range of -8 to +7.

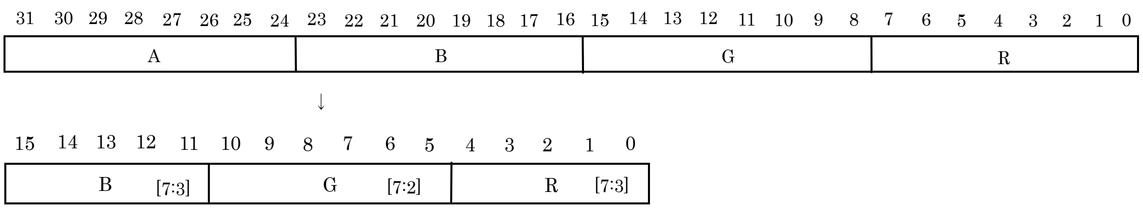
DM 00	DM 04	DM 08	DM 12
DM 01	DM 05	DM 09	DM 13
DM 02	DM 06	DM 10	DM 14
DM 03	DM 07	DM 11	DM 15

The coordinate (dX, dY) which is used to access the dither matrix uses the low-order two bits of the drawing coordinate (Xd, Yd).

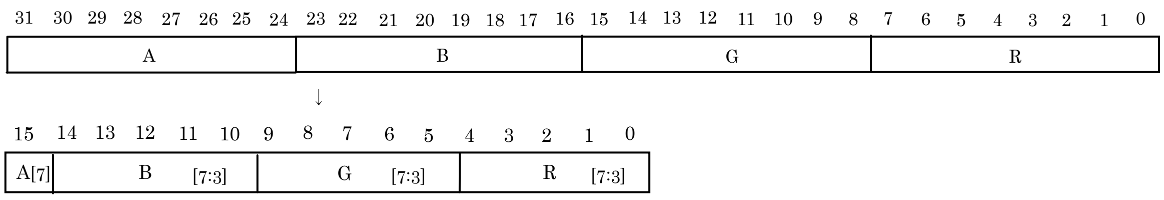
17.9. Color Clamp

Before drawing to the frame buffer, the Graphics Engine clamps color and alpha values to each assigned bit length starting from the high-order bit, to support the 16-bit color format set with the Frame Pixel Format command (CMD_FPF).

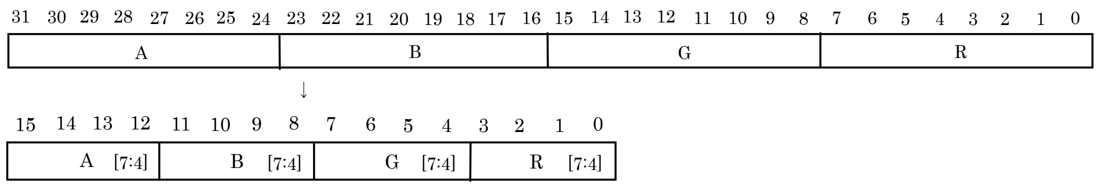
16-bit 5:6:5:0 color format



16-bit 5:5:5:1 color format



16-bit 4:4:4:4 color format



17.10. Logical Operations

The Graphics Engine can perform logical operations between the color value of a pixel and a color value in the frame buffer, and between the alpha value of a pixel and an alpha value in the frame buffer. Logical operations can be enabled using the Logical Operation Enable command (CMD_LOE). The logical operation to be performed can be set using the Logical Operation Parameter command (CMD_LOP). Note that logical operations do not affect the stencil bit.

Logical Operation (LOP) actions are shown in the following table.

LOP	Action
CLEAR	0
AND	Cs&Cd
AND_REVERSE	Cs&~Cd
COPY	Cs
AND_INVERTED	~Cs&Cd
NOOP	Cd
XOR	Cs^Cd
OR	Cs|Cd
NOR	~(Cs|Cd)
EQUIV	~(Cs^Cd)
INVERT	~Cd
OR_REVERSE	Cs|~Cd
COPY_INVERTED	~Cs
OR_INVERTED	~Cs|Cd

LOP	Action
NAND	~(Cs&Cd)
SET	1

Cs	: Color and alpha values of the input pixel (source)
Cd	: Color and alpha values in the frame buffer (destination)

17.11. Masking

The Graphics Engine can perform a mask operation on a depth value. The mask can be set using the Depth Mask command (CMD_ZMSK).

It can also perform masking on a color value and the stencil bit. You can also set a 32-bit plain mask value. When using 16-bit color format, the bits of the mask are applied to the 32-bit color before clamping. This can be set using the Plain Mask commands (CMD_PMSK1-2).

18. Antialiasing

The Graphics Engine enables you to perform antialiasing by drawing an antialias line along a jagged part after rendering.
The antialias line is drawn with a normal line primitive after antialiasing is enabled.
Antialising can be enabled using the Antialiasing Enable command (CMD_AAE).
As shown in Figure 18-1, the antialiasing lines blend by replacing a pixel coverage value with an alpha value and by alpha-blending with the closest pixels. At this time, you need to set data again that has been drawn once as a texture. To draw properly, you have to set textures, filter modes, functions and alpha blending correctly.

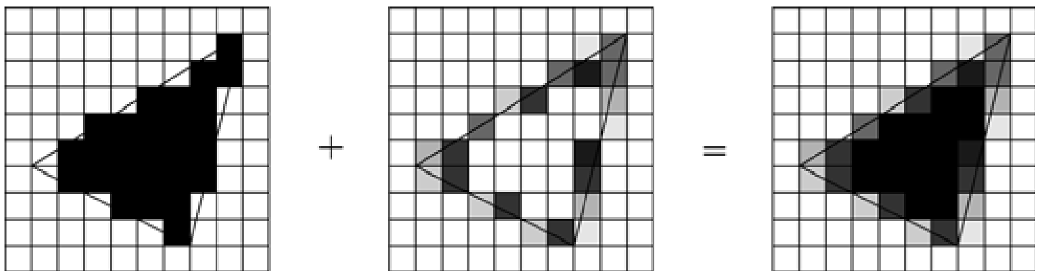


Figure 18-1 Antialiasing

The correct settings are shown below.

Setting	Contents
Texture mapping mode	UV mapping
Texture scale/offset	(SU,SV)=(1.0f, 1.0f) (TU,TV)=(0.0f,0.0f)
Texture wrap mode	CLAMP
Texture filtering	NEAREST
Texture function	REPLACE
Alpha blending	ADD Parameters(A,B)=(As,255-As)
Fogging	Disabled

19. Buffer Configuration

The Graphics Engine has a built-in VRAM, which is addressed linearly as shown in Figure 19-1. The VRAM can store the frame buffer, depth buffer, and texture buffer. Each buffer is defined by a specified base point and buffer width.
The base point must be specified in units of 8K bytes. When the frame buffer mode is 16 bits, the buffer width must be specified in 128-byte units. When the frame buffer mode is 32 bits, the buffer width must be specified in 256-byte units. For example, when the frame buffer mode is 16 bits and the buffer width is set to 480 pixels, 1024 bytes would be specified.

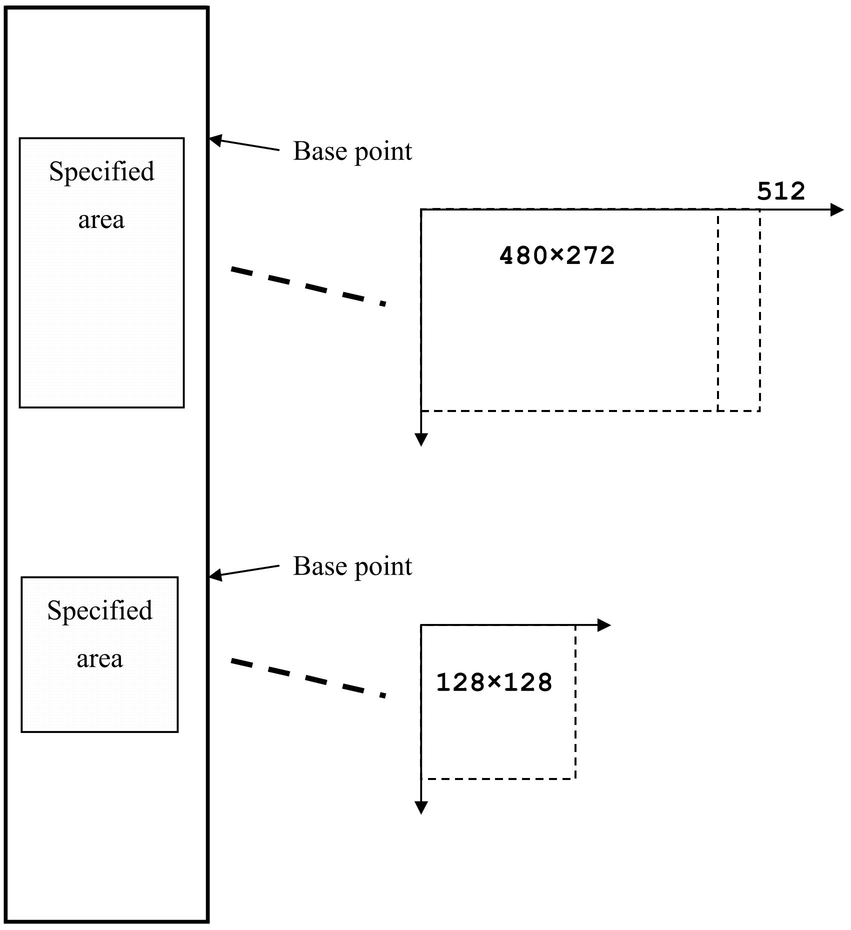


Figure 19-1 VRAM

19.1. Frame Buffer

The frame buffer is used for storing color values at the coordinates defined in the Drawing Coordinate System. A pixel in the frame buffer can be stored in any of the following color formats: 16-bit 5:6:5:0, 16-bit 5:5:5:1, 16-bit 4:4:4:4 and 32-bit 8:8:8:8. The highest order "A" bit of the 16-bit 5:5:5:1, 16-bit 4:4:4:4 and 32-bit 8:8:8:8 color formats also functions as a stencil bit during the stencil test.
The base point and width of the frame buffer can be set using the Frame Buffer commands (CMD_FBP, CMD_FBW).

16-bit 5:6:5:0 color format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B					G						R				

16-bit 5:5:5:1 color format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A	B					G					R				

16-bit 4:4:4:4 color format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A				B				G				R			

32-bit 8:8:8:8 color format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A								B								G								R							

19.2. Depth Buffer

The depth buffer is used for storing depth values at the coordinates defined in the Drawing Coordinate System. A pixel in the depth buffer is stored as a 16-bit unsigned integer. The base point and width of the depth buffer can be set using the Depth Buffer commands (CMD_ZBP, CMD_ZBW).

19.3. Texture Buffer

The Graphics Engine uses texture buffers stored in host memory or local memory to perform texture mapping. For details about texture buffers, see Section 13.3.3 “Texture Buffer.“

19.4. CLUT Buffer

When the texture mode is index format, CLUT buffers stored in host memory or local memory are used. For details about the CLUT buffer, see Section 13.6.2, “CLUT buffer.“

19.5. Inter-buffer Transfers

The Graphics Engine has a built-in inter-buffer transfer function for exchanging data between host and local memories. To perform an inter-buffer transfer, first use the Source Transfer Buffer commands (CMD_XBP1, CMD_XBW1) to define the source memory buffer. Next, use the Destination Transfer Buffer commands (CMD_XBP2, CMD_XBW2) to define the destination memory buffer. Then use the Buffer Transfer Position commands (CMD_XPOS1, CMD_XPOS2) to specify the transfer starting positions in the buffers, and use the Transfer Size command (CMD_XSIZE) to set the width and height, as shown in Figure 19-2. Finally, start the transfer with the Inter-buffer Transfer Start command (CMD_XSTART). This command can also be used to specify the pixel format of the data to be transferred.

The source buffer can be located in either host or local memory. The destination buffer is located in the other memory.

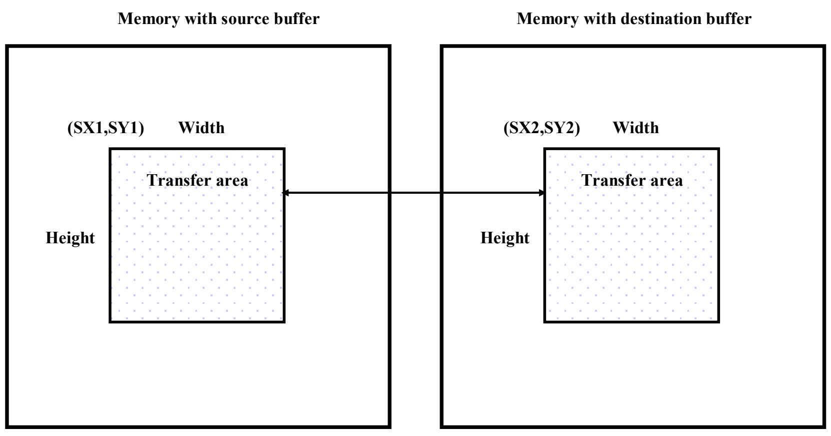


Figure 19-2 Inter-buffer Transfers

19.6. Buffer Clearing

The Graphics Engine provides built-in functions for clearing a buffer. Before clearing a buffer, you need to set the clear mode using the Clear Mode command (CMD_CMODE). When a clear mode is set, the Graphics Engine switches contexts to one used for clearing. The contexts used for clearing are shown below.

Function	Setting Value
Back face culling	DISABLE
Texture mapping	DISABLE
Fogging	DISABLE
Antialiasing	DISABLE
Alpha test	ALWAYS
Stencil test	DISABLE
Depth test	ALWAYS
Color test	DISABLE
Alpha blending	DISABLE
Logical operation	DISABLE

The clearing operation is implemented by setting the context to clear mode, then drawing. There are three enable bits for the Clear Mode command (CMD_CMODE) which are shown below, and it is possible to set whether to draw each of color, alpha and depth values.

Bit	Meaning
CEN	Color value enable
AEN	Alpha value enable
ZEN	Depth value enable

In a normal clearing operation, the buffer is completely cleared by setting the clear mode, then performing drawing with a given color value, alpha value and depth value. At the same time, Through Mode is used to set a coordinate in the Drawing Coordinate System. If the drawing were done in Normal Mode, the effect of vertex deformation, lighting or coordinate transformation would be too great so that it might not be possible to clear with the selected coordinate value, color value and depth value. Note that dithering, scissoring, depth mask and plain mask operations are performed in either mode.
You do not need to save the current state beforehand since the clearing context has a function to restore the original settings when it returns to Normal Mode, based on the setting of the Clear Mode command (CMD_CMODE).

20. Display List

The Graphics Engine provides a display list, which is a list for collecting together commands. Stall control is provided so that you can continue to draw even while the display list is being generated. If you want to use the created display list repeatedly, it is possible to nest a subroutine call up to two times.

20.1. Start of Display List

When a display list in main memory is to be transferred, set the start address of the display list in the Memory Address Register (MADR) and set the address following the last address in the Stall Address Register (SADR), then start the transfer. After the transfer starts, the memory address will be updated automatically. The Graphics Engine will stall when the memory address matches the stall address. After that, the stall address will be updated and reading will be restarted.

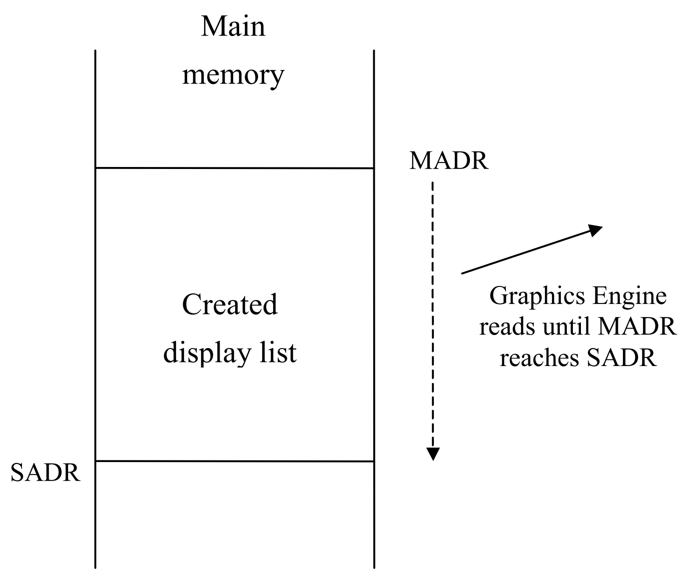


Figure 20-1 Start of Display List

20.2. Adding to the Display List

New commands can be added to the end of the created display list. After that, the stall address will be updated. With this procedure it is possible to add commands continuously without knowing how far the Graphics Engine has read.

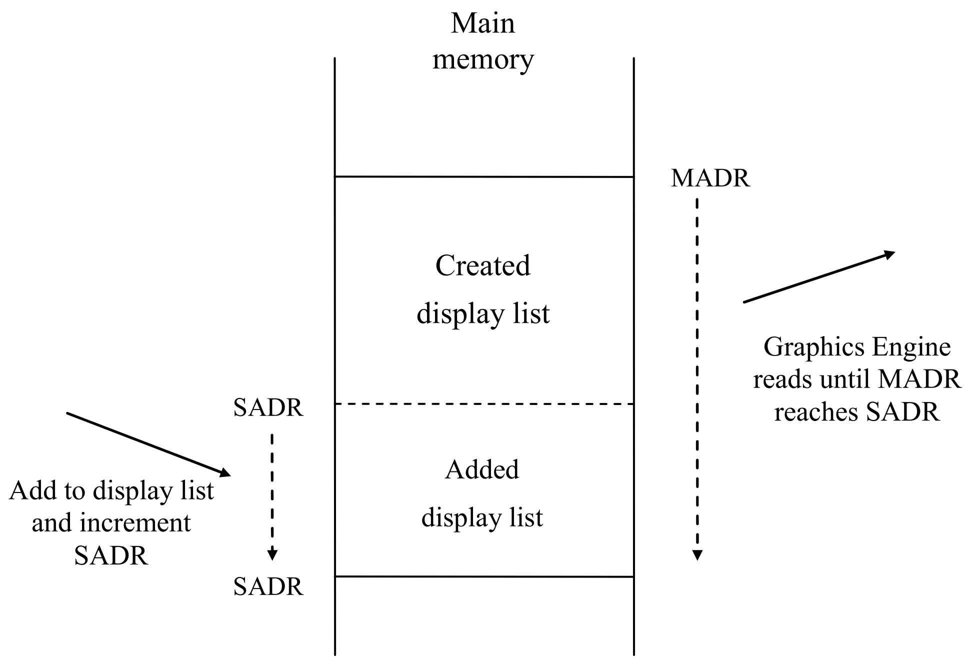

Main memory
MADR
Created display list
Graphics Engine reads until MADR reaches SADR
SADR
Added display list
Add to display list and increment SADR
SADR

Figure 20-2 Adding to the Display List

20.3. Display List Address Generation

When the Graphics Engine reads the following commands from the display list, it generates an address (ADDRnew) for internally controlling execution as shown in Equation 20-1.

- Jump command (CMD_JUMP)
- Conditional Jump command (CMD_BJUMP)
- List Call command (CMD_CALL)
- Vertex Data command (CMD_VADR)
- Index Data command (CMD_IADR)

In Equation 20-1, ADDR is the low-order 24-bit address set by the above commands, BASE is the high-order 8-bit address set by the Address Base command (CMD_BASE).

$$ADDRnew = (BASE \mid ADDR) + OFFSET$$

Equation 20-1 Address Calculation for Commands

In addition to directly setting the contents of the OFFSET, the register is automatically updated with the address set by the Offset command (CMD_OFFSET), when that command is read from the display list, or automatically updated with the address of the location of the Origin command (CMD_ORIGIN), when that command is read from the display list.
The address generated by the Jump command, Conditional Jump command, and List Call command is used by a subsequent branch instruction, and the address generated by the Vertex Data command and the Index Data command is used to set vertex data. (See Section 6.3 "Vertex Addresses")

20.4. Display List Jump

The Graphics Engine provides Jump and Conditional Jump commands to support branching. If a Jump command (CMD_JUMP) or Conditional Jump command (CMD_BJUMP) is read from the display list, the Graphics Engine can jump to the address generated by the command (see Section 20.3 "Display List Address Generation"). The Jump command causes an unconditional jump to occur. The Conditional Jump command jumps by finding the region outside the drawing area (see Section 11.1 "Object Culling").
Figure 20-3 shows the operation of jumps and conditional jumps.

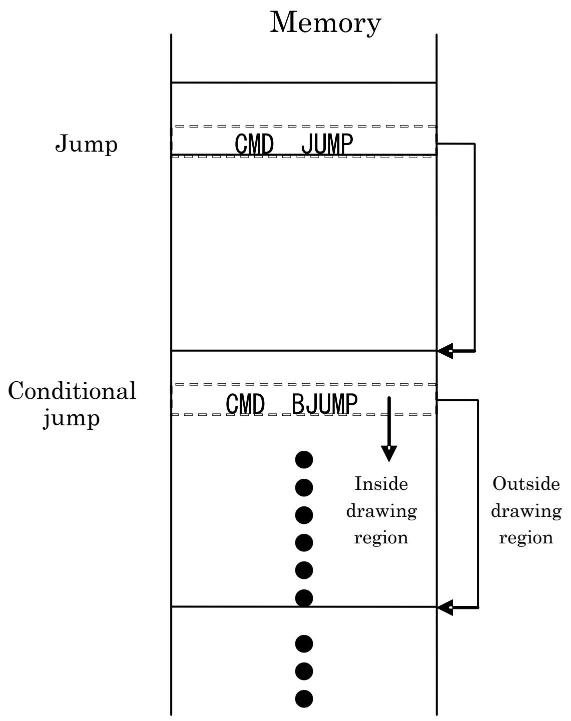


Figure 20-3 Display List Jumps

20.5. Calling a Display List

The Graphics Engine supports a call function for a display list to reuse the created display list. To execute a call, a display list to be used specifically for the call should be created in a memory area separate from the main display list currently being read. After creating the called display list, add the List Call command (CMD_CALL) which specifies the starting address of the called display list to the main display list and execute the call.
Calls can be nested up to a maximum of two times.
In addition, a List Return command (CMD_RET) should be added to the end of the called display list to return to the main display list. When the Graphics Engine reads the return command, it writes the return address corresponding to the nesting level back to the memory address, then continues reading the display list.

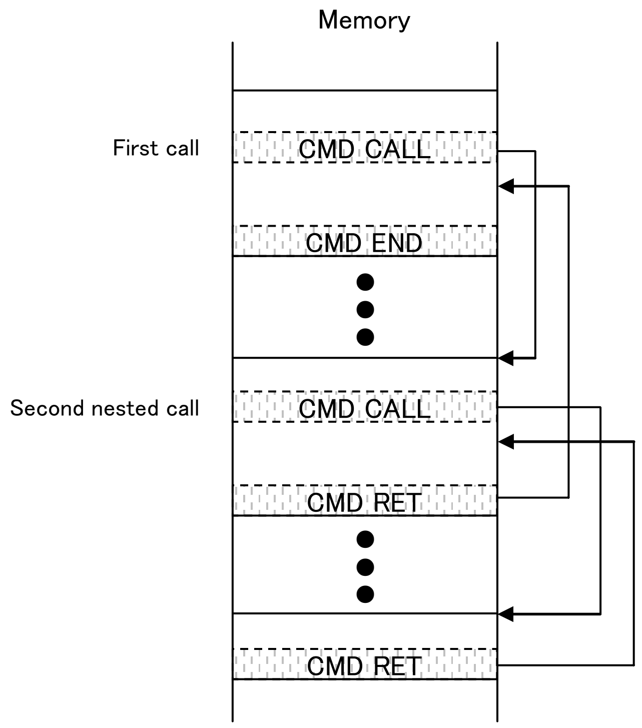


Figure 20-4 Calling a Display List

20.6. Ending a Display List

To end a display list, add an End Reading command (CMD_END) to the end of the display list. When the Graphics Engine reads this command, it resets the start bit to 0 and stops reading the display list. At the same time, a Reading Ended interrupt will also be generated for the host processor. To read the display list again, the start bit must again be set to 1.

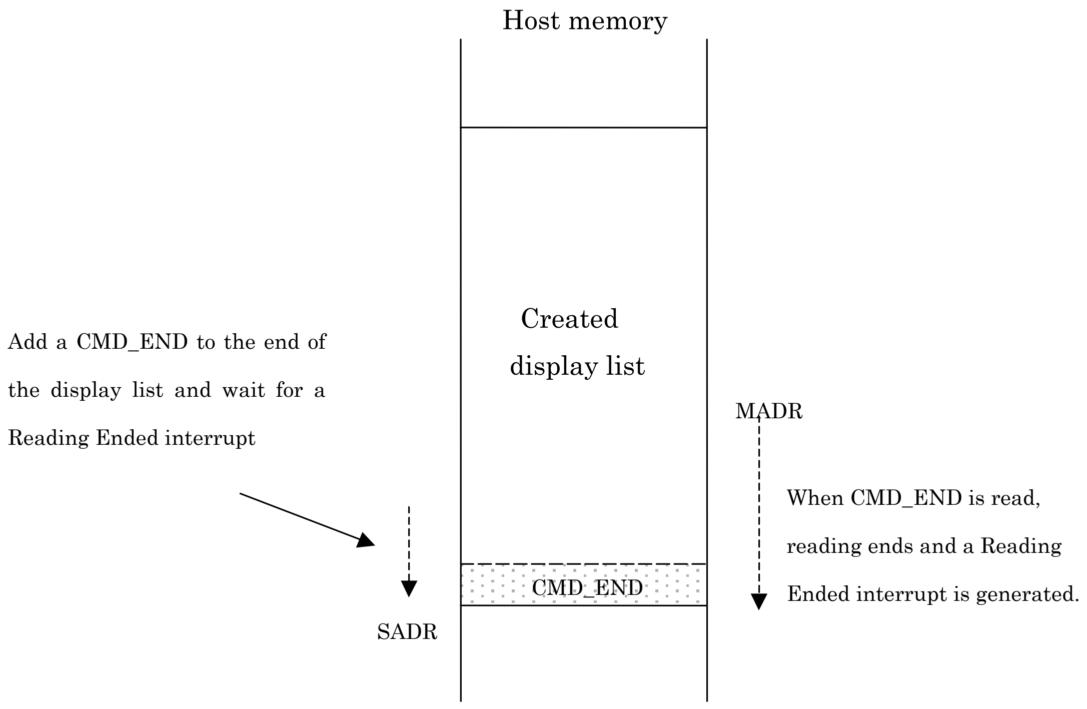

Host memory
Created display list
Add a CMD_END to the end of the display list and wait for a Reading Ended interrupt
CMD_END
SADR
MADR
When CMD_END is read, reading ends and a Reading Ended interrupt is generated.

Figure 20-5 Ending a Display List

20.7. Signal Operation

The host processor can use the Signal Interrupt command (CMD_SIGNAL) to find out how far a display list has been read. When the Graphics Engine reads the Signal Interrupt command (CMD_SIGNAL), a Signal interrupt is generated. At this time, the Graphics Engine will continue to read the next command without stopping the read operation.

20.8. Completion of Drawing

Even though reading of display lists has completed, this does not mean that all drawing operations have completed. The Finish Drawing command (CMD_FINISH) should be used to determine whether or not all drawing operations have completed. When a Finish Drawing command (CMD_FINISH) is executed, a Drawing Finished interrupt is generated after all previous commands have completed.

21. Interrupts

The Graphics Engine supports the four types of interrupts shown below. Each interrupt request signal has a user-programmable mask which is applied before an interrupt request is asserted to the host.

Interrupt Request Signal	Interrupt Cause
INTSIG	Signal interrupt
INTEND	Reading Ended interrupt
INTFIN	Drawing Finished interrupt
INTERR	Error interrupt

22. eDRAM Control

The Graphics Engine's eDRAM controller controls the 2MB dedicated eDRAM that is on the PSP™ system LSI chip. The eDRAM consists of two 1024-page × 32-column × 128-bit × 2-bank memory macros. The eDRAM macros can perform 128-bit reads and 128-bit writes simultaneously. The eDRAM controller has one 32-bit AHB port for setting, one 512-bit port for drawing, and one 128-bit AHB port for data. Figure 22-1 shows the ports of the eDRAM controller.

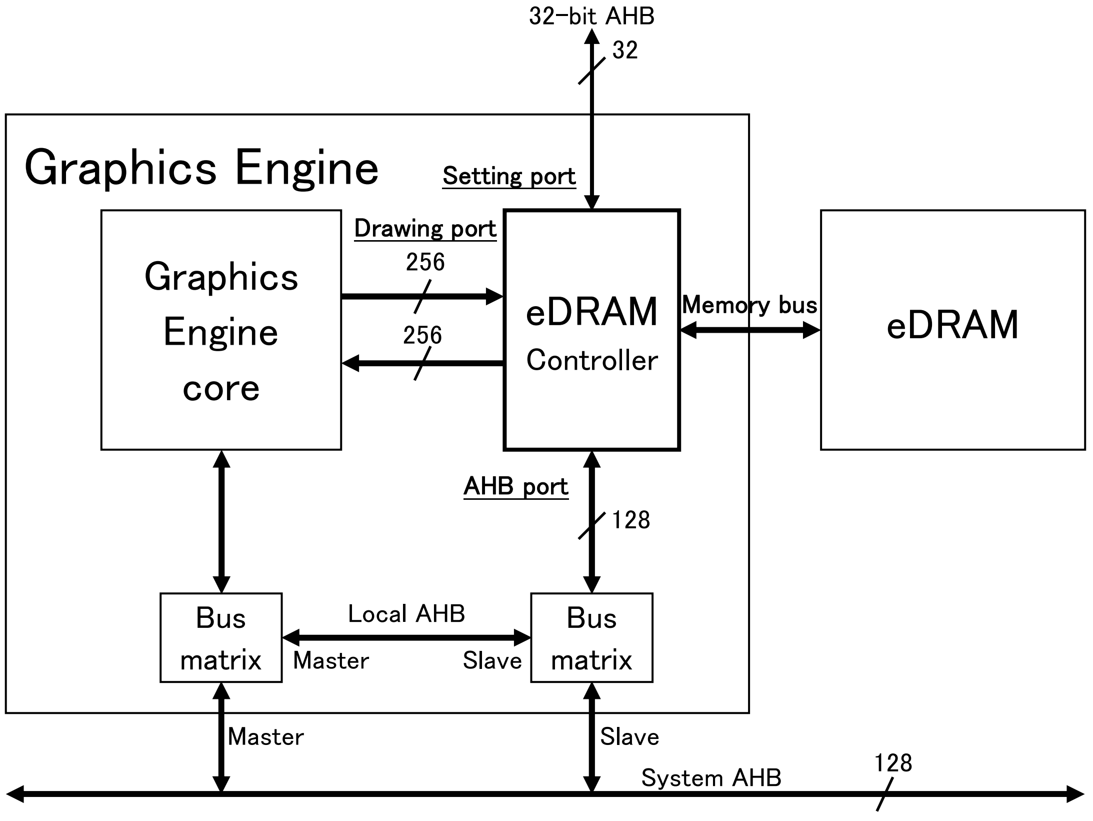


Figure 22-1 eDRAM Controller Ports

22.1. Ports Used for Settings

For all settings registers used for control, only 32-bit word access is possible. If access with a size other than 32 bits is attempted, an error response is returned.

22.2. Drawing Ports

In terms of drawing ports, there is a 256-bit read data bus, as well as a 256-bit write data bus, and simultaneous reading and writing of color values and depth values for up to four pixels is possible.

22.3. Addressing Modes

For the eDRAM controller, up to 2MB of eDRAM can be accessed. There are two types of addressing: a linear mode in which sequential access performance is emphasized, and graphics mode in which graphics performance is emphasized. After system reset, graphics mode goes into effect.

22.3.1. Linear Mode

In linear mode, the low-order addresses are allocated to one page of eDRAM as is, so this mode is appropriate for access to sequential addresses. Figure 22-2 shows the address conversion in linear mode.

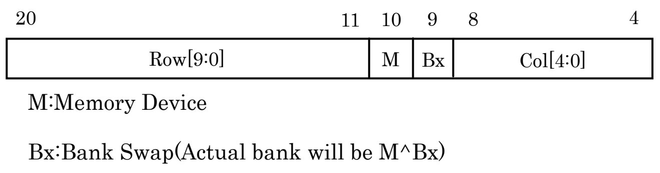


Figure 22-2 Linear Mode Address Conversion

22.3.2. Graphics Mode

In graphics mode, rectangular regions in the frame buffer are allocated to one page of eDRAM, so this mode is appropriate for the drawing of graphics in which there are many localized accesses. In graphics mode, as shown in Figure 22-3, addresses are arranged such that they fold back based on memory width. Drawing will be most efficient if the frame buffer width and the memory width match. The memory width can be set to one of 512 bytes, 1024 bytes, 2048 bytes, and 4096 bytes. Figure 22-4 shows the address conversion in graphics mode.
In addition, the addresses which are stored internally in the depth buffer will change based on the pixel format of the frame buffer. For that reason, the depth buffer cannot be correctly accessed directly from the AHB port. For information regarding AHB access to the depth buffer, see Section 22-4 Memory Map.

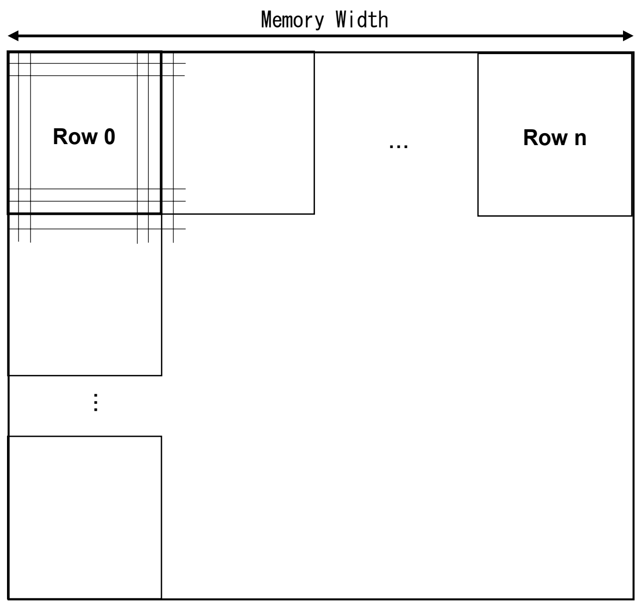


Figure 22-3 Graphics Mode

20 … 14	13	12 … 10	9 … 7	6	5 … 4
Row[9:3]	Bx	Col[4:2]	Row[2:0]	M	Col[1:0]

M:Memory Device

Bx:Bank Swap(Actual bank will be M^Bx)

Figure 22-4 Graphics Mode Address Conversion

22.4. Memory Map

Figure 22-5 shows the memory map used for accesses from the AHB port. Although the frame buffer and depth buffer base points are specified as addresses within a 2 MB space, the way the depth buffer is stored in memory depends on the pixel format of the frame buffer. When accessing the actual memory area of the depth buffer, be sure to use an address with a 2 MB offset (for 16-bit pixel format) or an address with a 4 MB offset (for 32-bit pixel format).

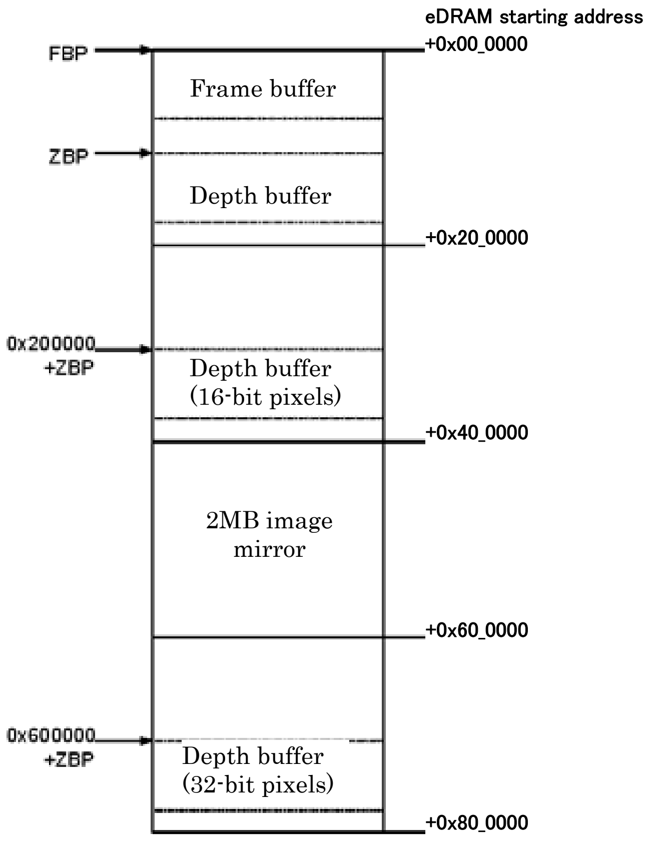


Figure 22-5 Memory Map

22.5. Arbitration

The drawing port and the AHB port both access the eDRAM, so arbitration needs to be performed internally. The priority order for arbitration is: AHB port (system AHB > local AHB) > drawing port. If the AHB port makes an access request while a transfer is in progress for the drawing port, reading is immediately stopped for the drawing port, and the AHB access is blocked until pixel data that has already been read is written back to memory (RMW).