Revolution

Graphics Library (Advance Rendering)

Version 1.00

Graphics Library (Advanced Rendering)

SDK Version 1.0.0

Contents

Revision History vii
1 Overview 1
2 Texture scaling, rotation, and translation 3
3 Texture projection 5
3.1 Texture projection principles 5
3.2 The lighting frustum 6
3.3 Logical overview 6
3.3.1 Texture image generation 7
3.3.2 Lit triangle set determination 7
3.3.3 Texture coordinate generation 8
3.3.4 Rendering 8
3.4 What is *q* for? 8
3.4.1 The math of geometry projection 8
3.4.2 The math of texture projection 9
3.4.3 Texture-perspective correction 9
3.4.4 Projected-texture correction 9
3.5 Complications when *q* is greater than zero 9
3.6 Complications with backfacing triangles 10
3.7 Complications with cross-shadowing 11
3.8 Texture projection vs. vertex lighting 11
3.8.1 Texture projection strengths 11
3.8.2 Vertex lighting strengths 11
3.8.3 Advantage of texture projection for shadows 11
3.9 Texture projection and vertex lighting combined 12
4 Reflection mapping 13
4.1 Simple planar reflections 13
4.1.1 Generating planar reflection maps 14
4.2 Spherical reflection (environment) mapping 16
4.2.1 Generating sphere-maps from cubic environment maps 17
4.2.2 Limitations 17
4.3 Dual-paraboloid environment mapping 18
4.3.1 Generating parabolic reflection maps from cubic environment maps 19
4.3.2 Limitations 19
5 Bump mapping 21
5.1 Bump mapping mathematics 21
5.2 Forward differencing 22
5.3 Bump mapping using indirect textures 23
5.3.1 Make the bump texture and light maps 23
5.3.2 Supply normal, binormal and tangent vectors for each vertex 23
5.3.3 Generate texture coordinates from the normals 26
5.3.4 Configure indirect texture stages 27
5.3.5 Configure regular TEV stages 28
5.3.6 Configure TEV processing 28
5.3.7 Binormal and tangent calculations for the polygon model 29

5.4 Diffuse bump mapping (embossing technique) 32
5.4.1 Generating offset texture coordinates 34
5.4.2 Configuring lighting channel 35
5.4.3 Texture Settings 36
5.4.4 Configuring TEV 36
5.4.5 Limitations 37
5.5 Specular bump mapping (environment map technique) 37
5.5.1 Creating Bump Maps and Environment Maps 37
5.5.2 Providing Normal Vectors for Each Peak 38
5.5.3 Create Texture Coordinates from the Normal Vectors 38
5.5.4 Indirect Stage Settings 38
5.5.5 Regular TEV Stage Settings 38
6 Applying Indirect Textures 41
6.1 Image warping 41
6.1.1 Settings for Warping 41
6.2 Tile Texture 43
6.2.1 Creating a Tile Map 43
6.2.2 Texture Settings 44
6.2.3 Texture Coordinate Settings 44
6.2.4 Indirect Stage Settings 44
6.2.5 Tile Settings 44
6.2.6 TEV Stage Settings 47
6.2.7 Drawing Polygons 47
6.3 Pseudo-3D textures 47
6.3.1 Pseudo-3D Textures 48
6.3.2 Tile Textures 49
6.3.3 Creating Indirect Textures 49
7 Full-scene shadow mapping 51
7.1 Depth-based full-scene shadow mapping 51
7.1.1 Overview of the Algorithm 51
7.1.2 Shadow Depth Map Generation 51
7.1.3 Ramp Texture Look Ups 52
7.1.4 TEV Settings for Drawing Shadows 52
7.1.5 Advantages and Disadvantages of this Algorithm 54
7.2 ID-based full-scene shadow mapping 54
7.2.1 Overview of the Algorithm 54
7.2.2 Generating a Shadow ID Map 54
7.2.3 TEV Settings for Drawing Shadows 54
7.2.4 Advantages and Disadvantages of this Algorithm 55
8 Gloss maps 57
9 Cartoon lighting 59
10 Outlining 61
11 Geometric decals 63
12 Z textures 65
12.1 Creating Z-Textures 65
12.1.1 Formatting 65
12.1.2 Z Values 65
12.2 Drawing Mode Settings 66
12.3 Texture Object Initialization 66
12.3.1 Texture Object Initialization 66
12.4 Drawing 66
12.5 Reference 67
13 Stitching 69
14 Antialiasing algorithm 71

14.1 Antialiasing impact on performance and quality 71
14.2 Non-antialiasing 72
14.3 Deflicker algorithm 72
15 Multi-resolution geometry 73
16 CPU direct access to the on-chip frame buffer 75

Code Examples

Code 1 - Sample scale, rotate, and translate 3
Code 2 - Splayed normals array 13
Code 3 - MTX functions 13
Code 4 - Scale and translate functions 14
Code 5 - Loading texture matrix and generating texture coordinates 14
Code 6 - Scale and translate functions 16
Code 7 - Forward differencing pseudocode 22
Code 8 - GXSetVtxDesc() 24
Code 9 - GXSetVtxAttrFmt() 24
Code 10 - Vertex Function 24
Code 11 - GXSetTevBumpST Source 26
Code 12 - TEV Stage 0 Settings 28
Code 13 - TEV Stage 1 Settings 28
Code 14 - TEV Stage 2 Settings 29
Code 15 - TEV Stage 3 Settings 29
Code 16 - Texture Coordinate Generation Number Settings 34
Code 17 - Texture Coordinate Generation 0 Settings 35
Code 18 - Texture Coordinate Generation 1 Settings 35
Code 19 - TEV Stage Number Settings 36
Code 20 - First Stage TEV Settings 36
Code 21 - Second Stage TEV Settings 36
Code 22 - GXSetTevIndBumpXYZ Source 38
Code 23 - TEV Settings when Using an Environment Map 39
Code 24 - TexObj Initialization 41
Code 25 - Indirect Texture Look Up Settings 41
Code 26 - Indirect Texture Process Settings 41
Code 27 - GXSetTevIndWarp Source 42
Code 28 - TEV Settings 42
Code 29 - GXSetTexCoordScaleManually 44
Code 30 - GXSetIndTexCoordScale 44
Code 31 - Indirect Stage Settings 44
Code 32 - Tile Settings 45
Code 33 - GXXSetTevIndTile Source 46
Code 34 - Assigning Scale Values 49
Code 35 - TEV Stage 2 50
Code 36 - TEV Stage 1 52
Code 37 - TEV Stage 2 53
Code 38 - TEV Stage 3 53
Code 39 - TEV Stage 1 (tg-shadow2.c) 54
Code 40 - TEV Stage 2 (tg-shadow2.c) 55
Code 41 - Gloss Map: TEV Stage 1 57
Code 42 - Gloss Map: TEV Stage 2 57
Code 43 - Z Buffer Activation 66
Code 44 - Comparison Process Sites 66
Code 45 - Clearing the Z Buffer 66
Code 46 - Texture Object Initialization 66

Code 47 - Filter Mode Settings 66
Code 48 - Final TEV Stage 67
Code 49 - GXSetCopyFilter 71

Equations

Equation 1 - Bump Map 21
Equation 2 - Perturbed normal 22
Equation 3 - Definition of the tangent and binormal vectors 22
Equation 4 - Simplified perturbed normal equation 22
Equation 5 - Forward differencing, example A 22
Equation 6 - Forward differencing, example B 22
Equation 7 - Calculation of the Binormal Vector (1) 31
Equation 8 - Calculation of the Binormal Vector (2) 31
Equation 9 - Calculation of the Binormal Vector (3) 31
Equation 10 - Calculation of the Tangent Vector 31
Equation 11 - Diffuse bump mapping 32
Equation 12 - Diffuse intensity 33
Equation 13 - Light vector transform matrix 33
Equation 14 - Light Vector Calculation (1) 33
Equation 15 - Light Vector Calculations (2) 34
Equation 16 - Light Vector Calculation (3) 34

Figures

Figure 1 - Example of projected lights and shadow 5
Figure 2 - A lighting frustum 6
Figure 3 - Data flow through the four logical stages of texture projection 7
Figure 4 - Splayed normals 13
Figure 5 - Desired texture coordinates 14
Figure 6 - Generating a planar reflection texture 15
Figure 7 - Creating a sphere map 16
Figure 8 - Mapping normals to sphere map texture coordinates 16
Figure 9 - Generating a sphere map by warping a cube map 17
Figure 10 - Sphere map texture coordinate interpolation problem 17
Figure 11 - Dual-paraboloid mapping 18
Figure 12 - Generating a dual-paraboloid map by warping a cube map 19
Figure 13 - Height field 21
Figure 14 - Relationship between V1, V2, binormal and tangent vectors. 30
Figure 15 - Diffuse bump mapping 33
Figure 16 - Tile Texture 43
Figure 17 - Pseudo-3D Texture 48
Figure 18 - Creating Indirect Textures 49
Figure 19 - Ramp Textures 52
Figure 20 - Gloss Map Image Diagram 57
Figure 21 - Antialiasing 71
Figure 22 - Non-antialiasing blending 72

Tables

Table 1 - GX_NEAR filtering 59
Table 2 - GX_LINEAR filtering 60

Revision History

Revision No.	Date Revised	Items (Chapter)	Description	Revised By
21-Aug-2006	8/25/2006	-	First release by Nintendo of America.	-

1 Overview

Note: The Revolution Graphics Programming Manual is still being written. This Nintendo GameCube Graphics Programming Manual is being provided on a temporary basis for reference. Please refer to the graphics library function reference manual until the Programming Manual is available.

This document explains how to achieve certain graphics effects, such as environment mapping and bump mapping. We highlight the basics of each technique and explain how each can be achieved using the hardware features available on Nintendo GameCube.

In addition, this manual is written with the assumption that the user has read the *Revolution Graphics Library (GX)*, included with this SDK.

2 Texture scaling, rotation, and translation

One of the simplest, but most useful, forms of texture coordinate generation is a simple 2x4 transform of an input texture coordinate. The following code fragment creates three output texture coordinates from one input texture coordinate. Texture coordinates 1 and 2 can be scaled, rotated, and/or translated versions of texture coordinate 0. This method may be used to compress the amount of data needed to represent an object.

Code 1 - Sample scale, rotate, and translate

```
Mtx tm1, tm2;

MTXScale(tm1, 0.7, 1.1, 0.4);
MTXRotDeg(tm2, 'z', 45.0);

GXSetVtxDesc(GX_VA_POS, GX_DIRECT);
GXSetVtxDesc(GX_VA_TEX0, GX_DIRECT);
// set vertex format, not shown

GXLoadTexMtxImm(tm1, GX_TEXMTX0, GX_MTX_2x4);
GXLoadTexMtxImm(tm2, GX_TEXMTX1, GX_MTX_2x4);

GXSetTexCoordGen(GX_TEXCOORD0, GX_TG_MTX2x4, GX_TG_TEX0, GX_IDENTITY);
GXSetTexCoordGen(GX_TEXCOORD1, GX_TG_MTX2x4, GX_TG_TEX0, GX_TEXMTX0);
GXSetTexCoordGen(GX_TEXCOORD2, GX_TG_MTX2x4, GX_TG_TEX0, GX_TEXMTX1);

GXSetNumTexGens(3);
```

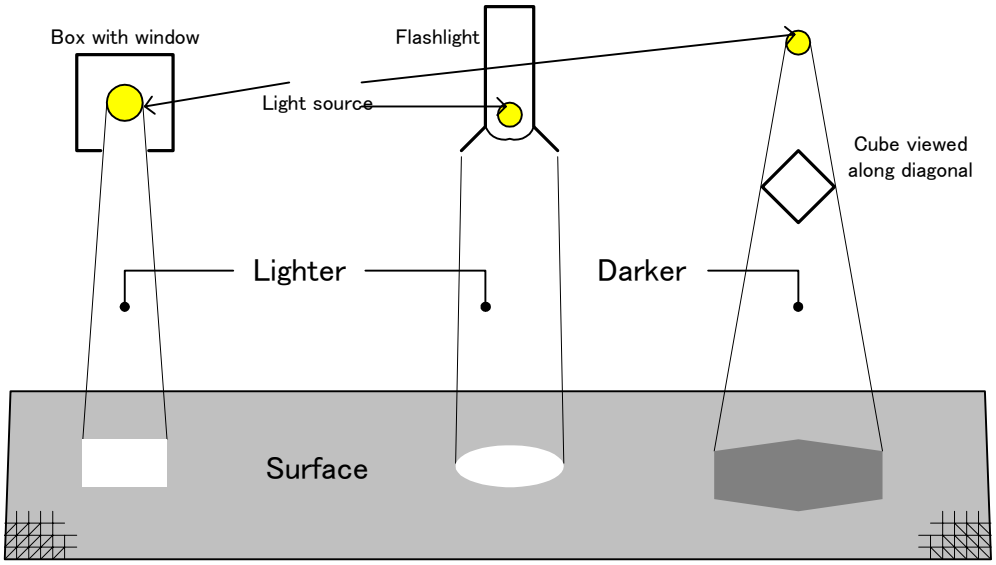
3 Texture projection

With the technique of texture projection (also known as texture lighting), you can create a very wide range of light and shadow effects that simulate a light source shining on a surface with an intervening object creating a shadow. The effects can be seen as rough-but-convincing approximations to light and shadow effects normally associated with ray-tracing. In contrast, texture projection is much more efficient, but less general and therefore more troublesome.

3.1 Texture projection principles

Painting a texture onto a local group of triangles can render the area either lighter or darker. You can paint either a light patch (i.e., a light shining through an opening or shutter), or a dark patch (i.e., the effect of a shadow cast from an object). Examples of a projected light texture include a lantern, a spotlight, a flashlight, a slide projector, and a searchlight projecting onto the sky. Examples of a projected shadow texture include the shadows cast by characters or objects onto the background.

Figure 1 - Example of projected lights and shadow

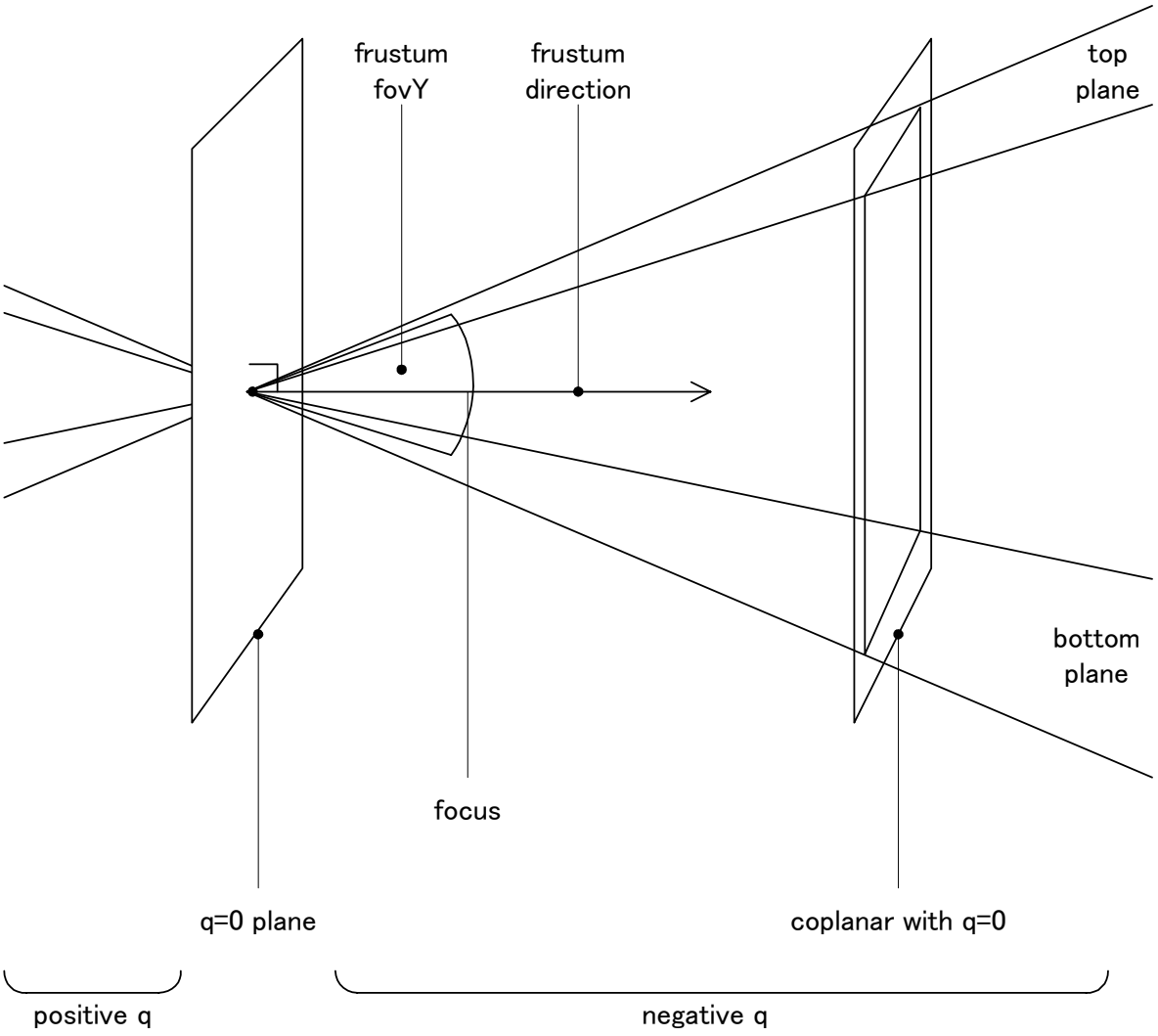


3.2 The lighting frustum

Texture projection has similarities to geometry projection, and can apply the concept of a frustum. A lighting frustum is a region of space that is bounded by four planes (left, right, top, and bottom) that meet at a focal point and is defined relative to a direction vector. The direction vector can be seen as the predominant or median direction of the light source. The focal point is at the location of the light source—this is the point from which the projected rays radiate.

A lighting frustum is pyramidal in shape, describes an (infinite) volume of 3D space that is to be lit (or shadowed), and orients the texture within that space. Because the light frustum has no near plane or far plane, the space extends from the focus point to infinity—in contrast to geometry projection. This is reflected in the Matrix-Vector library functions for creating lighting frustums (see “Matrix-Vector Library (MTX)” in the *Dolphin Reference Manual*).

Figure 2 - A lighting frustum



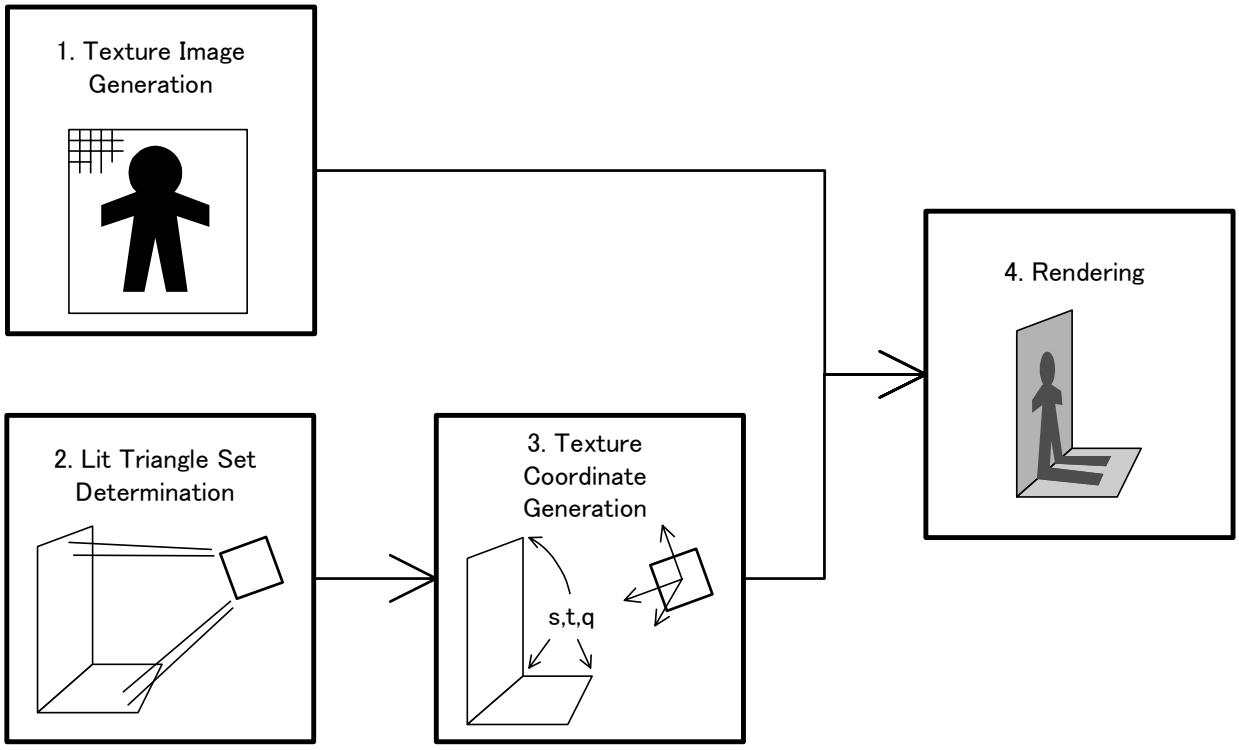
3.3 Logical overview

Logically, the texture projection process has four stages:

1. **Texture generation**: A texture image is made which represents the cast light (or shadow).
2. **Lit triangle set determination**: The set of triangles to be lit (or shadowed) is selected.

3. **Texture coordinate generation**: For every lit vertex, (*x, y, z*) is projected into (*s, t, q*).
4. **Rendering**: The lit triangles are rendered using the texture and the generated texture coordinates.

Figure 3 - Data flow through the four logical stages of texture projection



3.3.1 Texture image generation

You must generate a texture image which represents the light (or shadow) cast on a surface. Black texels (i.e., texels with value 0) represent areas of the surface unaffected by the light (or shadow). An intensity texture will often suffice. For a spotlight effect, for example, you might use a texture that was a white solid circle on a black background. Or to simulate a shadow from sunlight, you could take the character from the viewpoint of the sun (possibly with a reduced geometry LOD and probably with a reduced viewport size) and render it into a buffer every frame, and then use the buffer as the texture image.

3.3.2 Lit triangle set determination

The set of triangles to be lit or shadowed must be selected with the application. Although you could select all triangles in the scene, in most cases you will want to find a much smaller lit set. The entire problem of lit set determination is similar to the visible triangle determination problem common in graphics rendering. You can borrow a combination of culling, clipping and scissoring techniques, with familiar trade-offs resulting between coding cost, runtime load, and resulting flexibility:

- Selecting all triangles—simple, but inefficient.
- Culling objects/characters/space/sub-units using bounding-solids, and testing against the light frustum's *q*=0 plane.
- Testing against the full light frustum (top, bottom, left, and right).
- Culling triangles by testing vertices against the full light frustum (or just the *q*=0 plane).
- Clipping triangles against the full light frustum (or just the *q*=0 plane).

Implementing the last two per-triangle solutions is time-consuming, makes many other graphics features and optimizations more difficult to implement or modify, and has a high runtime load. However, these techniques also make the lights very mobile.

Any or all of these solutions can be combined with static offline decisions. Controlling and restricting the background, location of the objects, and location of the lights is the simplest way to gain maximum efficiency and avoid problems. For example, if the light and background never move, and you neglect objects, the lit triangle set can be determined completely statically.

3.3.3 Texture coordinate generation

In this stage, for every vertex of every triangle in the lit set, the per-vertex (*x, y, z*) geometry coordinates are projected using a lighting frustum into per-vertex (*s, t, q*) texture coordinates.

The Nintendo GameCube performs the following during this stage:

- Texture generation/projection by a 3x4 texture matrix from (*x, y, z*) to (*s, t, q*) per-vertex.
- Rasterization of (*s, t, q*) and division by *q* per-texel.

3.3.4 Rendering

In this stage, the triangles together with the texture image are rendered using the calculated vertex (*s, t, q*) texture coordinates.

The Nintendo GameCube allows the following to be used with rendering:

- Multiple texture coordinates (up to eight coordinate triples).
- Multiple texture lookup (up to eight textures).
- Multiple blend stages in the TEV (up to 16 stages).

Assuming your triangles have one (static) texture already, you can add up to seven projected textures on top of that one texture on each triangle. The eight textures can be rendered *in a single pass*. This means you do not need to transform all the triangles in an object two or more times (i.e., once to draw the static base texture, and once to draw each projected shadow on the object).

Multiple pass techniques are also possible, if it is more convenient to draw the object twice or if you need more than eight textures per triangle.

3.4 What is *q* for?

3.4.1 The math of geometry projection

When projecting geometry, there are two stages:

1. Multiply each (*x, y, z*) triple by a 4x4 projection matrix to make a new vector (*x, y, z, w*). In this way the projection matrix generates a new, fourth parameter *w* which measures the perpendicular distance from the camera (in the *z* direction). The projection matrix also scales (*x, y, z*) to a convenient range.
2. Scale (*x, y, z*) by *w*, and perform the actual projection effect.

As a result of perspective projection from world space to screen space, the parameter can be regarded as representing the scale factor needed by every vertex (actually 1/scale-factor). For instance, if vertex P1 has (*w* = -1), meaning no scaling, then vertex P2, twice as far from the camera, would have (*w* = -2), meaning “shrink to half size on the screen.” Vertex P0, precisely coincident with the camera, would have (*w* = 0), meaning “enlarge infinitely.” Vertices like P0 are a problem, and near-clipping exists to eliminate them.

3.4.2 The math of texture projection

In the same way, texture projection has the same two stages:

1. Multiply each (*x, y, z*) by a 3x4 projection matrix to make a new vector (*s, t, q*). In this way the projection matrix generates a new, third parameter *q* which measures the perpendicular distance from the light focus point (in the light *z* direction). The projection matrix also scales (*s, t*).
2. Scale (*s, t*) by *q*, and perform the actual texture projection.

As a result of the texture projection the parameter *q* can be regarded as representing the scale factor needed by every vertex on the model (actually 1/scale-factor). For example, if vertex P1 has (*q* = -1), then vertex P2, twice as distant from the light focus, would have (*q* = -2). Vertex P0, precisely coincident with the light focus, would have (*q* = 0), meaning “enlarge infinitely.” Vertices like P0 are a problem.

3.4.3 Texture-perspective correction

As a result of the fact that most hardware calculates *w for every vertex*, it is possible to simulate perspective (also called *foreshortening*), the effect where objects in the distance appear smaller on the screen.

Note, however, that this *w* calculation is later performed *for every pixel* on the model, and it is used to scale (*s, t*) *for every texel* on the model. As a result, each texture will distort properly, even *inside* triangles. This feature, known as texture-perspective correction, prevents “folding” of textures on big triangles.

3.4.4 Projected-texture correction

Because Nintendo GameCube hardware calculates *q* for every texture coordinate set *for every vertex*, it is possible to simulate widening or narrowing shadows, where one end of the projected texture is enlarged in comparison to the other.

Again, this *q* calculation is later performed *for every pixel* on the model, and it is used to scale (*s, t*) *for every texel* on the model. As a result, each texture will distort properly, even *inside* triangles. This feature, which could be called projected-texture correction, prevents “folding” of projected textures on big triangles, and it supports lights not-at-infinity on big triangles.

In fact, because both *q* and *w* are calculated and combined for every texel, both these distortions or corrections are combined and can interact without any problem or complication.

Note: This vector has an implicit (1) as an extra, last element. For example, Nintendo GameCube hardware represents (*x, y, z*) as (*x, y, z, 1.0*) internally.

3.5 Complications when *q* is greater than zero

In the case of geometry projection, vertices that get “too close” to the camera plane (*z* = 0) have a *w* value which is very tiny. Dividing (*x, y, z*) by these small values often causes precision problems and overflow, and when (*w* = 0), the resulting values are infinite. Clipping geometry to a near-plane avoids these problems—each triangle with offending vertices is cut into two pieces by the near-plane, and the half that is “too close” is thrown away.

Triangles with vertices “behind” the camera (*z* > 0) are also clipped to this near-plane. If this were not done, large triangles with one vertex behind and one vertex in front of the camera could not be drawn properly by the triangle-steppers (DDAs). After clipping, all vertices have legitimate values for *w*. It is for this reason that near-clipping is supported.

Nintendo GameCube hardware does not support clipping for (*s, t, q*) and projected textures. As a result:

1. Even if only *one* of the three vertices on a triangle has (*q* > 0) (i.e., is behind the light frustum), the entire triangle will be adversely affected—in practice, multiple aberrant sheared copies of the texture scintillate and strobe over the triangle in a highly inappropriate fashion.
2. If the entire triangle is behind the light frustum, the same thing will happen.

This means that the application should control events so that triangles with projected texture(s):

- Do not cross the (q = 0) plane, or if they do, the texture is attenuated to be invisible.
- Do not sit on the wrong side of the (q = 0) plane, or if they do, the texture is attenuated to be invisible.

Therefore, the following are possible solutions:

1. If all the lit triangles are fairly small, the desired lighting frustum is not too wide, and the light cannot get too close to any lit triangles, it should be possible to use vertex lighting spotlight attenuation to ensure that the projected texture is invisible on the entire surface of these problem triangles.
2. It may also be helpful to use distance attenuation to render invisible triangles which are too close to the (q = 0) plane.
3. Alternatively, objects (or triangles) entirely inside (or outside) of the frustum (or (q > 0) half-space) can be culled by the application (i.e., not rendered with the projected texture).
4. Use color generation.
5. Clip all lit triangles against a near-plane; e.g., the plane (q = -1).
6. Ensure that the light focus point is always outside any objects that may be lit, that the light always points towards the objects, and that the objects can never intersect the (q = 0) plane.
7. Place the light focus at infinity; i.e., (q = 1) for all vertices.

In fact, because of limited (s, t, q) precision, it is possible that the following problems will be experienced:

- **Small *q*** – Fixed-point precision of (s, t) after dividing by q is the problem here. Vertices too near to the (q = 0) plane and too far from the center of the projected texture (as controlled by the light frustum direction vector) will overflow.
- **Large *q*** – The maximum q fixed-point precision is unknown at this time.
- **Large *s* and/or *t*** – Before and after dividing by q, there are maximum fixed-point precisions.

Exceeding any of these limits on any vertex will cause the texture on the *entire* triangle to wrap and move around.

3.6 Complications with backfacing triangles

Projected texture lights and shadows will normally “shine” straight through any object, appearing on both the front (as desired) and on the back of the object. (For objects without extreme projections, the texture could easily be visible on the surface far more than two times.) This results in a bizarre and unrealistic X-ray type of effect. Solutions include:

- Combining a projected texture light with a vertex light, using the vertex lighting angle attenuation feature to render the projected texture invisible on backfacing triangles.
- Ensuring that the light is constrained and the object is constructed such that either it only has one side (with respect to the light), or that the “back” is never visible to the viewer. This is often possible with landscapes, which are often flat enough (and the light sufficiently high) that the light cannot shine “through.” It is also possible if the light position is always near the viewer position.
- Dynamically culling every triangle which faces away from the light. This is equivalent to a backfacing test for every triangle, and must be performed by the CPU.
- Statically determining which triangles are backfacing, and restricting or fixing the light so that they remain backfacing at all times.

3.7 Complications with cross-shadowing

Texture projection is not ray-tracing, and cannot easily be extended to general support of all self-shadowing and object cross-shadowing effects. However, if necessary, you can perform effective and workable approximations to cross-shadowing by the following methods:

- Testing lighting frustums against object bounding-solids.
- Sorting objects according to distance *q* for each light.
- Calculating a shadow texture for each object from the point-of-view of each light.

Nearby (and intersecting) objects, as well as objects with extreme projections, will not shadow each other correctly.

3.8 Texture projection vs. vertex lighting

Texture projection and vertex lighting (see the *Revolution Graphics Library (GX)* have different effects and different strengths.

3.8.1 Texture projection strengths

First, texture projection can handle complex and interesting shapes and patterns of light or dark, whereas vertex lighting is limited to a few ambient, specular, and spotlight effects generated by simple mathematical equations.

Second, texture projection can cast sharper-edged, higher-frequency shadows or light areas on triangles with a far more convincing effect, because it can provide a completely different color per texel, and is limited only by the texture resolution. By comparison, vertex lighting is restricted to a different color per vertex, and therefore specular and spotlight effects can easily appear unnatural when the underlying triangle geometry is revealed. This becomes a greater problem as triangles get larger—if all triangles are about one screen pixel in size, it will not be an issue, but generally triangles are much larger.

3.8.2 Vertex lighting strengths

Vertex lighting can take vertex normals into account, and can therefore attenuate lighting depending on the incident angle of the light on the surface. For example, the sides of a sphere will be lit less than the front, and the back will not be lit at all.

Texture projection does not recognize vertex normals, and will affect the front, sides, and back equally. Moreover, texture projection will “shine” straight through a solid object, possibly generating two textures, one on the front, and one on the back of the object.

Vertex lighting can be local, and it can attenuate the lighting depending on the distance of the vertex from the light, so that farther triangles are lit less. Texture projection will not attenuate with distance.

Vertex lighting can simulate a local light which shines outwards in all directions, such as a naked candle or light bulb. Texture projection cannot—it is limited to project through a frustum, which has a field-of-view that cannot extend wider than 180° in either the *s* or *t* direction (i.e., it cannot diverge more than 90° from the projection direction).

3.8.3 Advantage of texture projection for shadows

Vertex lighting is not much good for simulating and painting shadows. It is ideal for omnidirectional, partly directional local, or infinite lights, which do not change much. The weaknesses of texture lights are generally less of a problem when painting a shadow cast by one object onto another object. The fact that projected textures will shine “through” a solid object is often not a problem if you are painting object shadows on the ground or landscape.

3.9 Texture projection and vertex lighting combined

You can use texture projection and vertex lighting together for more sophisticated effects and to combine complex high-frequency patterns with distance attenuation and incident angle attenuation. With careful manipulation of the four lighting channels (see the *Revolution Graphics Library (GX)*, four independently positioned lights can be realized in a single pass. However, the problem of limited field-of-view for texture projection still remains.

4 Reflection mapping

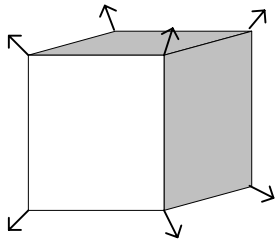
Reflection mapping is a technique used to create mirror-like surfaces. The following sections outline several methods to implement reflection mapping.

4.1 Simple planar reflections

A simple way to create the effect of a shiny surface (like chrome) is to generate texture coordinates from the vertex normal. The generated texture coordinates are used to look up a texture map that contains the reflected image.

As an example, look at how a cube can be reflection-mapped. Since the normals will be used to create texture coordinates, it is important that all the normals for a polygon are not pointing the same way (or only one texel will be applied to the whole polygon). In our case, the cube will be modeled with the normals pointing away from the center through each corner.

Figure 4 - Splayed normals



Code 2 - Splayed normals array

```
float FloatNorm[] =
{
   -1,  1, -1,
   -1,  1,  1,
   -1, -1,  1,
   -1, -1, -1,
    1,  1, -1,
    1, -1, -1,
    1, -1,  1,
    1,  1,  1
};
```

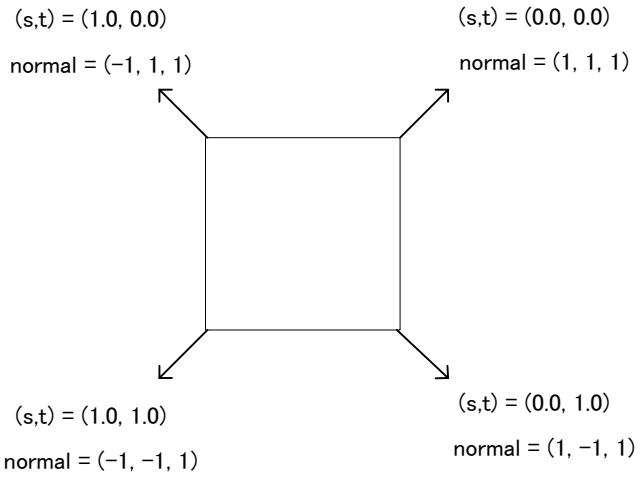
To construct the matrix that will convert the normals to texture coordinates, first rotate the model normals into view space:

Code 3 - MTX functions

```
       MTXInverse(mv, mv);
       MTXTranspose(mv, mv);
```

Now we must convert the three-dimensional normal coordinate to a two-dimensional texture coordinate. We will assume that most of the reflection effect should occur when the mirrored surface is facing the viewer. Therefore, we map the (*x, y*) coordinates of the normal in view space to the (*s, t*) texture coordinates. Since the texture coordinates should range from 0.0 to 1.0 in order to address the entire reflection map, we need to scale and translate the matrix as shown in the figure below. This figure shows the cube when it is directly in front of the viewer:

Figure 5 - Desired texture coordinates



The following code will scale and translate the view space normals to convert them to the desired texture coordinates:

Code 4 - Scale and translate functions

```
MTXScale(s, 0.5F, -0.5F, 0.0F);
MTXTrans(t, 0.5F, 0.5F, 1.0F);
MTXConcat(s, mv, mv);
MTXConcat(t, mv, mv);
```

Finally, we load the matrix and enable texture coordinate generation to use it:

Code 5 - Loading texture matrix and generating texture coordinates

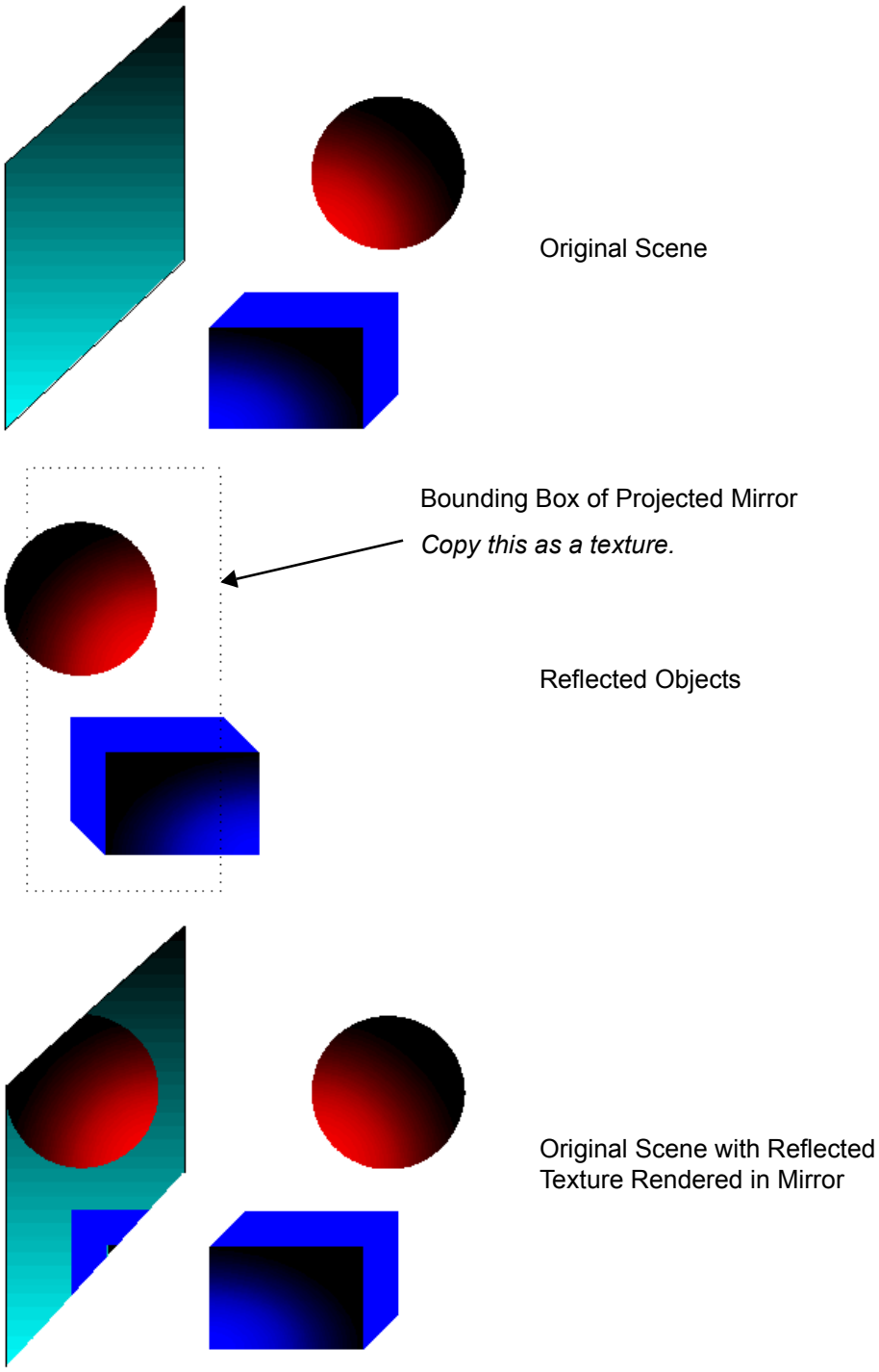
```
GXLoadTexMtxImm(mv, GX_TEXMTX0, GX_MTX3x4);
GXSetTexCoordGen(GX_TEXCOORD0, GX_TG_MTX3x4, GX_TG_NRM, GX_TEXMTX0);
```

Note: When a polygon has view space normals that vary only in Z, the texture coordinates will not be very different. This causes a few texels to be stretched across the polygon.

4.1.1 Generating planar reflection maps

You can create planar reflection maps by reflecting the viewpoint about the plane of the reflector, rendering the scene as a texture, and then rendering the texture onto the planar reflector from the original viewpoint. The function `GXProject` can be used to determine the screen location of a projected vertex position. By finding the bounding box of the planar reflector in screen space, you can reduce the size of the viewport and scissor box when rendering the reflected objects, and thus the size of the texture that needs to be copied.

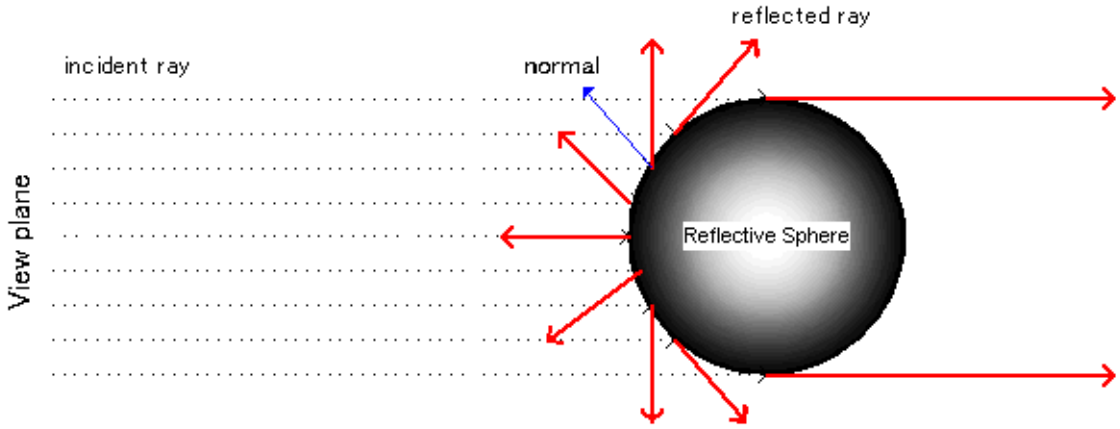
Figure 6 - Generating a planar reflection texture



4.2 Spherical reflection (environment) mapping

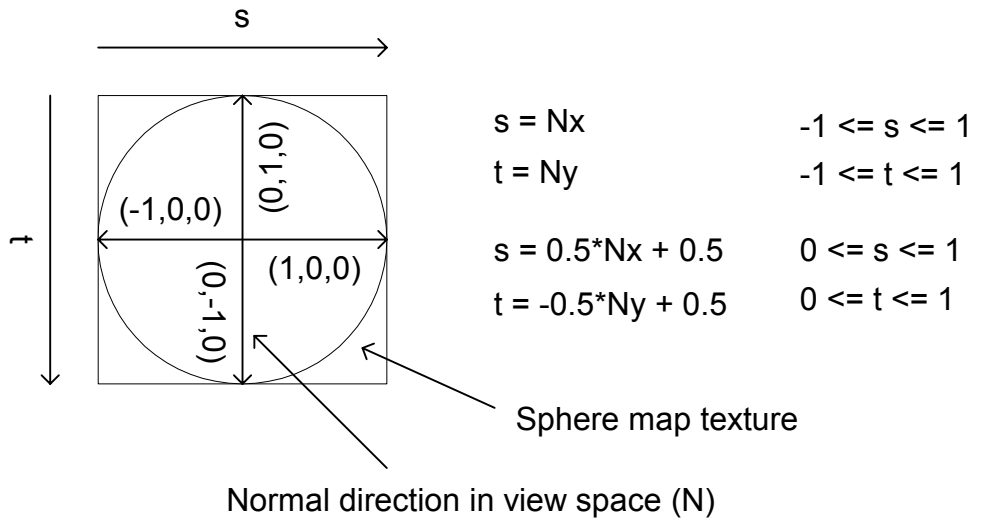
A sphere map is the orthographically-projected image of a perfectly reflecting sphere in an environment where the viewer is infinitely far away. The environment behind the viewer maps to a circle in the center of the texture, and the environment in front of the viewer maps to the edges of the texture surrounding the circle.

Figure 7 - Creating a sphere map



Nintendo GameCube has no special hardware for generating texture coordinates for sphere maps. The following indirect texture diagram shows how to map normals into texture coordinates that can be used to index a sphere map.

Figure 8 - Mapping normals to sphere map texture coordinates



As shown in the code below, you can generate a texture matrix to scale and translate the normal to generate a texture coordinate. In this example, the matrix *mv* is the inverse transpose of the modelview matrix.

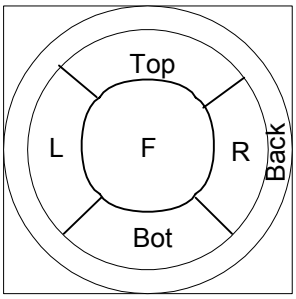
Code 6 - Scale and translate functions

```
MTXScale(s, 0.5F, -0.5F, 0.0F);
MTXTrans(t, 0.5F, 0.5F, 1.0F);
MTXConcat(s, mv, mv);
MTXConcat(t, mv, mv);
```

4.2.1 Generating sphere-maps from cubic environment maps

Cube maps describe the environment as six sides of a cube where each side represents a different view: up, down, left, right, top, or bottom. Cube maps may be created offline or at runtime. A sphere map may be created at runtime by warping the texture of a cube map using geometry. Each face of the cube map is warped (using a geometric mesh) into a corresponding area of the sphere map, as shown in the figure below:

Figure 9 - Generating a sphere map by warping a cube map



The geometry and texture coordinates of the mesh are static for a given tessellation. See demo `tg-spheremap` as an example.

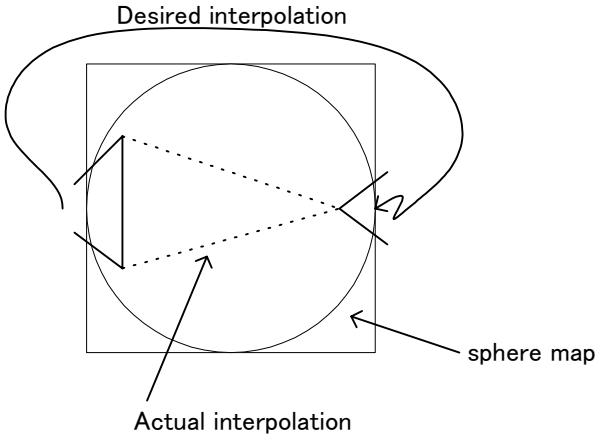
4.2.2 Limitations

The sphere map is view-dependent. When the viewpoint moves, the sphere map will need to be regenerated.

The texture coordinates generated for sphere maps assume that the view vector is constant for all vertices. When the view remains constant and the object translates; the reflection does not change.

Texture coordinate interpolation at the silhouette edges of objects may look up texels from the interior of the sphere map, rather than at the outer edges as desired. This can cause “sparkles” on the silhouette as a reflection-mapped object moves in the scene.

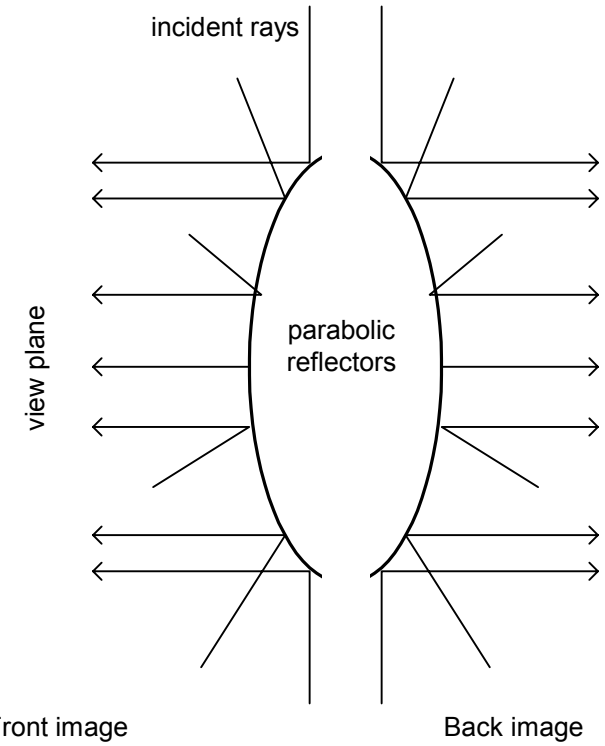
Figure 10 - Sphere map texture coordinate interpolation problem



4.3 Dual-paraboloid environment mapping

Another approach, called dual-paraboloid environment mapping, is view-independent and avoids the sampling problem at silhouette edges. In this approach, two parabolic reflectors capture *front* and *back* images of the environment by using a parabolic mapping function:

Figure 11 - Dual-paraboloid mapping



Note: *Front* and *back* are independent of the viewer's notion of front and back. Each parabolic map contains some of the information in the other one, and neither map has a complete image of the environment. The area in the center of each map, called the *sweet-circle*, contains information that is better-sampled (if it is contained at all) than the other map. This circle is defined by an alpha value of 1.0 in each texture.

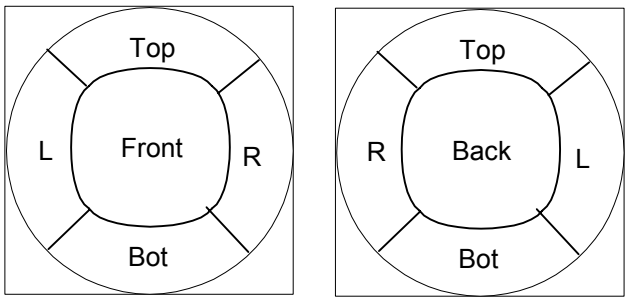
Texture coordinates are generated using the following code. Two texture coordinates are generated: one for the front map, and one for the back map.

The texture pipeline is configured to look up both textures, and the TEV is programmed to interpolate between the textures using the alpha channel of the second texture.

4.3.1 Generating parabolic reflection maps from cubic environment maps

You can generate dual-paraboloid environment maps by warping cubic environment maps. This is similar to the process used to generate sphere maps, but the mapping of cube faces to the mesh is different. See demo `tg-parabolicmap` for an example.

Figure 12 - Generating a dual-paraboloid map by warping a cube map



Front Texture Back Texture

Alpha = 1.0 inside the circle and 0.0 outside

4.3.2 Limitations

If the environment is changing dynamically, the dual-paraboloid maps will need to be re-generated. This requires generating (warping) two textures and running two TEV stages, instead of the one texture and one TEV stage needed for sphere mapping.

5 Bump mapping

Bump mapping allows you to add realism to an image without using a lot of geometry. Bump mapping modifies the shading of a polygon by effectively perturbing the surface normal on a per-pixel basis. The shading makes the surface appear bumpy, even though the underlying geometry is relatively flat. This section describes the basic mathematics of bump mapping, and how these equations are implemented for Nintendo GameCube.

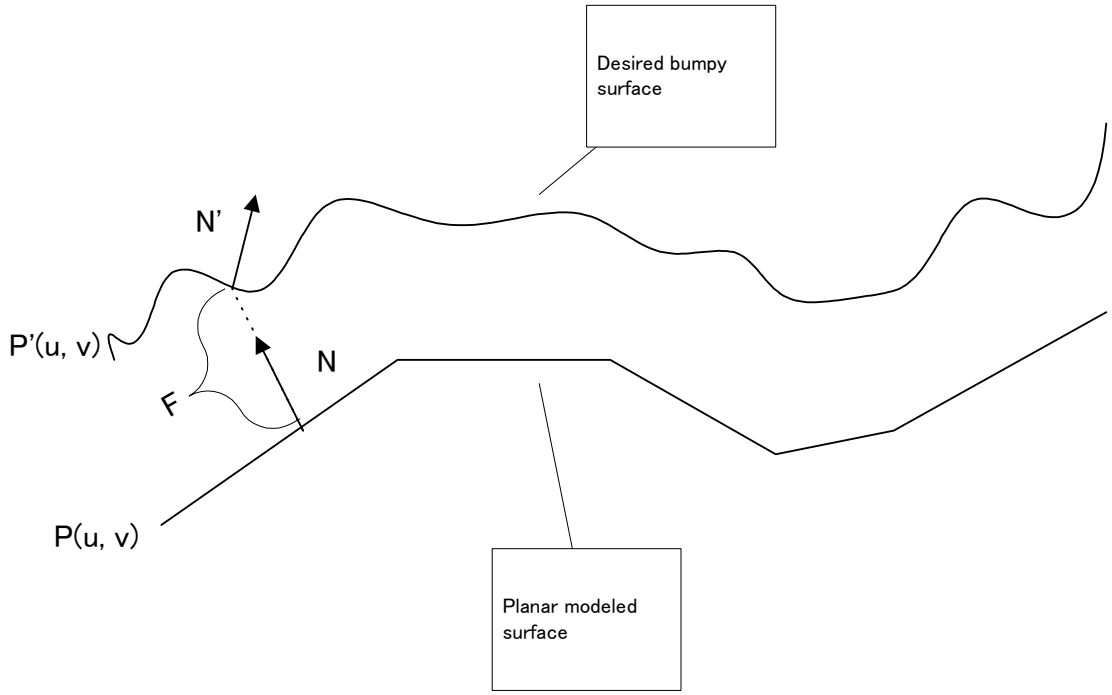
5.1 Bump mapping mathematics

The bumpiness of a surface can be described by a 2D height field, also called a *bump map*. The bump map is defined by the scalar difference *F(u, v)* between the flat surface *P(u, v)* and the desired bumpy surface *P'(u, v)* along the normal *N* at each point (*u, v*). The notation F_u indicates the partial derivative of *F* with respect to direction *v*.

Equation 1 - Bump Map

$$P'(u,v) = P(u,v) + F(u,v)\hat{N}$$

Figure 13 - Height field



Normally, *P* is modeled using polygons, and *F,* the bump height field, is modeled as a 2D image using a paint program or other tool.

"Equation 1 - Bump Map" on page 21 can be solved for *N'* to become the following equation:

Equation 2 - Perturbed normal

$$\vec{N}' = \hat{N} + F_u\left(\hat{N} \times P_u\right) + F_v\left(\hat{N} \times P_v\right)$$

Using *T* for the tangent vector and *B* for the binormal vector:

Equation 3 - Definition of the tangent and binormal vectors

$$\vec{T} = \hat{N} \times P_u, \vec{B} = \hat{N} \times P_v$$

Finally, we get:

Equation 4 - Simplified perturbed normal equation

$$\vec{N}' = \hat{N} + F_u\vec{T} + F_v\vec{B}$$

The values F_u and F_V can be computed beforehand through forward differencing of the 2D bump map. P_u and P_V can be computed either directly from the surface definition, or from forward differencing applied to the surface parameterization. As we will show in the following sections, the normal, binormal, and tangent vectors are defined for each vertex and interpolated across the polygon as texture coordinates. The indirect texture hardware solves equation 4 for each pixel, generating another texture coordinate that can be used to look up an environment or light map directly.

5.2 Forward differencing

Forward differencing is a method used to compute the first derivative of the height field (bump map) in a given direction. Assume a one-dimensional bump map represented as the height function *F(s)*. The forward difference is given by:

Equation 5 - Forward differencing, example A

$$F_s = F(s) - F(s + \Delta s)$$

If the delta *s* is one texel, and the width of the bump map is *w*, then, assuming *s* is normalized to map coordinates, the forward difference is:

Equation 6 - Forward differencing, example B

$$F_s = F(s) - F(s + \frac{1}{w})$$

This equation would approximate the first derivative of *F*, if *F* were continuous.

For a 2D bump map, the forward differencing operation is similar to the following pseudo-code:

Code 7 - Forward differencing pseudocode

```
for (t = 0; t < h; t++) {
    for( s = 0; s < w; s++) {
        tex00 = bump_img(s,t);
        tex01 = bump_img(s+1,t);
        tex10 = bump_img(s, t+1);
        Fs = tex00 - tex01;
        Ft = tex00 - tex10;
    }
}
```

The forward differencing equation can produce negative values.

5.3 Bump mapping using indirect textures

You can implement bump mapping, as defined by "Equation 4 - Simplified perturbed normal equation" on page 22, by using a combination of texture coordinate generation and indirect texturing. The basic steps are outlined below (with further details in later sections):

1. With a tool prepared in advance, compute an indirect texture that is the forward difference F_u and F_v of the bump map.
2. With a tool or in real time, create a light map for referencing by indirect textures.
3. Supply the normal, binormal, and tangent vectors per-vertex for the model being bump mapped.
4. Rotate the normals in view space and convert them to texture coordinates using `GXSetTexCoordGen`. The texture coordinates (normals) are iterated across the polygon and supplied to the indirect hardware for each pixel.
5. The regular texture coordinates are used to look up the indirect texture, F_u and F_v. During the indirect stage the binormal- and tangent-generated texture coordinates are scaled by F_u and F_v, respectively, and then added to the coordinate generated by the normal. This texture coordinate is the perturbed normal, *N'*. The perturbed normal can be used to look up a light or an environment map.
6. The TEV is configured to combine the lighting effects created by the light map referenced by the indirect texture feature with the material texture pasted into the model as normal texture.

5.3.1 Make the bump texture and light maps

The bump map hardware can add a bias of -128 to each difference map texel. Therefore, a range of -128 to +127 is possible for 8-bit-per-component difference maps. When computing the difference image, scale the difference so it is in the range -128 to +127, then add a 128 bias to each texel.

Bump maps use 8-bit difference maps in the S and T directions, so the texture format uses IA8. The S-direction difference is stored in the alpha component and the T direction difference is stored in the intensity component.

The light maps are employed so that the diffuse and specular effects can be used with indirect textures. Light vectors cannot be input with indirect textures, so the lighting effect is achieved by applying simple normal maps. For this reason, it is necessary to prepare textures for which lighting effects in the camera space have been rendered in advance for use with normal maps.

We will describe the way that light maps are made using a sample demo as an example. In the sample demo, the normal map of a 32 x 32 grid on a hemispherical mesh was rendered using diffuse and specular light. This was then copied as a 64 x 64 IN ADVANCE texture using `GXCopyTex`. Refer to the `drawHemisphere` function for the sample `ind-bump-st.c`. For the settings used for this mesh, a specular light object was created and this was enabled on two channels. The diffuse light of the first channel served as the color component while the alpha component of the second channel was used as specular light. These were enabled and rendered using a TEV. This makes use of the fact that the direction of the light set up using `GXInitSpecularDir` can be used as diffuse light of an unlimited directivity.

When the positional relationships of the light and camera will not be changed, this light map may be created offline, but we feel that there are times when the positional relationships between the light and camera are constantly changing. In such a case, a light map will have to be generated for each frame.

5.3.2 Supply normal, binormal and tangent vectors for each vertex

To use this form of bump mapping, you must supply a normal, binormal, and tangent vector per vertex. In Nintendo GameCube hardware, the three vectors must be indexed using an independent index. Independent indexing allows for the use of three smaller tables that can be shared for all objects.

To enable normal, binormal, and tangent per vertex, use:

Code 8 - GXSetVtxDesc()

```
GXSetVtxDesc(GX_VA_NBT, GX_DIRECT); // for using direct mode
GXSetVtxDesc(GX_VA_NBT, GX_INDEX8); // for using index mode
```

Note: All three normals will share the same vertex attribute format:

Code 9 - GXSetVtxAttrFmt()

```
// Using a Common Index for Normal, Binormal and Tangent Vectors
// or direct mode
GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_NBT, GX_NRM_NBT, GX_F32, 0);

// Using 3 indepedent indices for normal, binormal, and tangent vectors
GXSetVtxAttrFmt(GX_VTXFMT0, GX_VA_NBT, GX_NRM_NBT3, GX_F32, 0);
```

When drawing a triangle with attribute `GX_VA_NBT` enabled, you must send the three vectors, directly after the position attribute:

Code 10 - Vertex Function

```
// Using direct mode
GXBegin(GX_VTXFMT0, GX_TRIANGLES, 3);
    GXPosition3f32(0.0, 1.0, 0.0); // vertex 0
    GXNormal3f32(0.0, 0.0, -1.0);
    GXNormal3f32(0.0, 1.0,  0.0);
    GXNormal3f32(1.0, 0.0,  0.0);

    GXPosition3f32(1.0, 0.0, 0.0); // vertex 1
//  ...
GXEnd();

// Using a Common Index for Normal, Binormal and Tangent Vectors
GXBegin(GX_VTXFMT0, GX_TRIANGLES, 3);
    GXPosition1x8(0); // vertex 0
    GXNormal1x8(0);

    GXPosition1x8(1); // vertex 1
//  ...
GXEnd();
// Using 3 indepedent indices for normal, binormal, and tangent vectors
f32 Nrm_f32[] ATTRIBUTE_ALIGN(32) =
{
//         x,             y,             z
NRM_00_X,NRM_00_Y,NRM_00_Z, // index 0
        NRM_01_X,NRM_01_Y,NRM_01_Z, // index 1
        NRM_02_X,NRM_02_Y,NRM_02_Z, // index 2
NRM_03_X,NRM_03_Y,NRM_03_Z, // index 3
        NRM_04_X,NRM_04_Y,NRM_04_Z, // index 4
        NRM_05_X,NRM_05_Y,NRM_05_Z, // index 5
NRM_06_X,NRM_06_Y,NRM_06_Z, // index 6
//  ...
};

.....
```

```
GXBegin(GX_VTXFMT0, GX_TRIANGLES, 3);
    GXPosition1x8(0); // vertex 0

   GXNormal1x8(3);    // index = 3 ( NRM_03_X, NRM_03_Y, NRM_03_Z )
   GXNormal1x8(3);    // index = 3+1 ( NRM_04_X, NRM_04_Y, NRM_04_Z )
   GXNormal1x8(3);    // index = 3+2 ( NRM_05_X, NRM_05_Y, NRM_05_Z )
// the normal data is addressed as follows:
//
// address = array_base + index * array_stride
//  + (3 * NBT_offset + XYZ_component) * component_size
//
//      NBT_offset: N = 0, B = 1, T = 2
//      XYZ_component: X = 0, Y = 1, Z = 2

   GXPosition1x8(1); // vertex 1

//  ...
GXEnd();
```

5.3.3 Generate texture coordinates from the normals

When using an ST direction difference map, the displacement amounts for the normal, binormal and tangent vectors are calculated and added together using a TEV stage, so three TEV stages are necessary.

For this, `GXSetTevIndBumpST`, the TEV indirect feature-setting function, is used. The internal operation of `GxsetTevIndBumpST` is as follows:

Code 11 - GXSetTevBumpST Source

```
/*---------------------------------------------------------------------------*/
//
// Name:    GXSetTevIndBumpST
//
// Desc:    Set up TEV stages for environment-mapped bump-mapped texture
//          lookup.  The bump map is in ST space.
//
// Args:
//
/*---------------------------------------------------------------------------*/

void GXSetTevIndBumpST (GXTevStageID tev_stage, GXIndTexStageID ind_stage,
                            GXIndTexMtxID matrix_sel)
{
    GXIndTexMtxID sm, tm;

    switch(matrix_sel)
    {
      case GX_ITM_0:
        sm = GX_ITM_S0;
        tm = GX_ITM_T0;
        break;
      case GX_ITM_1:
        sm = GX_ITM_S1;
        tm = GX_ITM_T1;
        break;
      case GX_ITM_2:
        sm = GX_ITM_S2;
        tm = GX_ITM_T2;
        break;
      default:
        ASSERTMSG(0, "GXSetTevIndBumpST: Invalid matrix selection");
    }

    GXSetTevIndirect(tev_stage,         // tev stage
                     ind_stage,         // indirect stage
                     GX_ITF_8,          // format
                     GX_ITB_ST,         // bias
                     sm,                // matrix select
                     GX_ITW_0,          // wrap direct S
                     GX_ITW_0,          // wrap direct T
                     FALSE,             // add prev stage output?
                     FALSE,             // use unmodified TC for LOD?
                     GX_ITBA_OFF        // bump alpha select
        );
    GXSetTevIndirect((GXTevStageID) (tev_stage+1),       // tev stage
                     ind_stage,         // indirect stage
                     GX_ITF_8,          // format
                     GX_ITB_ST,         // bias
                     tm,                // matrix select
                     GX_ITW_0,          // wrap direct S
                     GX_ITW_0,          // wrap direct T
                     TRUE,              // add prev stage output?
                     FALSE,             // use unmodified TC for LOD?
```

```
                    GX_ITBA_OFF        // bump alpha select
        );
    GXSetTevIndirect((GXTevStageID) (tev_stage+2),       // tev stage
                    ind_stage,         // indirect stage
                    GX_ITF_8,          // format
                    GX_ITB_NONE,       // bias
                    GX_ITM_OFF,        // matrix select
                    GX_ITW_OFF,        // wrap direct S
                    GX_ITW_OFF,        // wrap direct T
                    TRUE,              // add prev stage output?
                    FALSE,             // use unmodified TC for LOD?
                    GX_ITBA_OFF        // bump alpha select
        );
}
```

Here is a simple description of the internal operation above:

Three TEV stages are used and include the TEV stage that was specified using the `GXSetTevIndBumpST` argument **tev_stage**.

With the indirect function of the first TEV stage, the vector made by rotating the binormal mapping vector over the camera space is taken as regular texture coordinates. This is used as the S direction dynamic matrix. The amount of displacement of the S direction texture coordinates can be calculated by multiplying this by the S direction difference texture.

With the indirect function of the second TEV stage, the vector made by rotating the tangent mapping vector over the camera space is taken as regular texture coordinates. This is used as the T direction dynamic matrix. The amount of displacement of the T direction texture coordinates can be calculated by multiplying this by the T direction difference texture.

For the third TEV stage, the normal vector is rotated above the camera space and the vector converted over the texture space is input as the regular texture coordinates. To this are added the S and T direction texture coordinates that were calculated using the first and second stages.

By doing it this way, the final output of the third indirect stage can be used as the texture coordinates for both the S direction and the T direction, corrected using the bump maps with a regular TEV stage texture reference.

The source of the sample demo showing how to use this function is in `ind-bump-st.c`.

5.3.4 Configure indirect texture stages

During the indirect stage, the texture coordinates used to apply a difference map to an object and a texture ID, which has loaded the difference map to be referenced, are given. `GXSetIndTexOrder` is used for this purpose.

If the size of the texture maps to be used with the indirect stage and the regular TEV stage are not identical, use `GXSetIndTexCoordScale` to scale them in the ST direction so that the texture coordinates can be shared. However, these scale coefficients are restricted to a geometric series of halves from 1 through 256, so anything other than that cannot share texture coordinates.

The indirect matrix settings are determined with `GXSetIndTexMtx`. Only the dynamic matrix is used for bump maps. No static matrix is used, so it does not really matter what the static matrix given to the `GXSetIndTexMtx` argument is set to. This function is used to set the scale value that you wish to multiply the dynamic scale by.

`GXSetTevIndBumpST` (described above) is used for the TEV indirect function settings. `GXSetTevIndBumpST` uses the TEV function in three steps, so the corresponding TEV stages must be made active.

The content that should be set using GXSetTevIndBumpST is the first stage ID of the TEV to be used with this function and the indirect stage ID. Then the indirect scale value that the offset is multiplied by is set. The indirect scale value is the scale value multiplied by the dynamic matrix, so the static matrix ID that stores the scale value used for it is set. In this way, the appropriate combination of dynamic matrix and scale value can be selected within the function.

5.3.5 Configure regular TEV stages

At the regular TEV stage, the reference to the light map is based on the results of adding the displacement in the S direction and the T direction generated by the indirect stage and the texture coordinate values for the normal map to the regular texture coordinate values.

It will be necessary to set the GX_TEXMAP_DISABLE to bit OR with GXSetTevOrder so that the first two stages of the three continuous TEV stages corresponding to GXSetTevIndBumpST will not perform a texture look up. The texture look up of the light map will be based on the results of the third stage texture coordinate calculations.

5.3.6 Configure TEV processing

In the sample ind-bump-st.c, a four-stage TEV stage is used.

In the first stage, the normal color texture is referenced and the result is stored in the PREV register. Here no indirect textures are used, so the indirect function is made inactive using GXSetTevDirect.

In the next three stages, texture coordinates are generated using the indirect texture. The light map is referenced based on the texture coordinates in the last of those stages and the texture color is used as diffuse and the texture alpha is used as specular. By combining this with the material texture color in the PREV register, the bump mapped results can be obtained.

The settings for TEV stage 0 are as follows:

Code 12 - TEV Stage 0 Settings

```
GXSetTevDirect(GX_TEVSTAGE0);
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD0, GX_TEXMAP0, GX_COLOR_NULL);
GXSetTevOp(GX_TEVSTAGE0, GX_REPLACE);
```

This is simply the application of color texture to model data and storage of the color and alpha in corresponding PREV registers.

The settings for TEV stage 1 are as follows:

Code 13 - TEV Stage 1 Settings

```
GXSetTevIndBumpST(GX_TEVSTAGE1, GX_INDTEXSTAGE0, GX_ITM_0);
GXSetTevOrder( GX_TEVSTAGE1, GX_TEXCOORD1,
(GXTexMapID)(GX_TEXMAP1 | GX_TEX_DISABLE), GX_COLOR_NULL);
GXSetTevOp(GX_TEVSTAGE1, GX_PASSCLR);
```

The three stages from GX_TEVSTAGE1 are used for indirect processing for bump map use with GXSetTevIndirectBumpST. The bump map texture ID and texture coordinates are stored in the indirect stage 0, so GX_INDTEXSTAGE0 is specified. The scale value for direct matrix uses the scale value for static matrix 0.

GXSetTevOrder is used to set the texture coordinate ID that determines the vector that rotates the binormal vector in the camera space as texture coordinates. The texture map that loads the light map is specified in the texture map ID, but at this stage, the actual texture will not be referenced, so GX_TEX_DISABLE is set to bit OR.

The first stage PREV register colors and alphas pass through unchanged as the TEV output colors.

The settings for TEV stage 2 are as follows:

Code 14 - TEV Stage 2 Settings

```
GXSetTevOrder( GX_TEVSTAGE2, GX_TEXCOORD2,
(GXTexMapID)(GX_TEXMAP1 | GX_TEX_DISABLE), GX_COLOR_NULL);
GXSetTevOp(GX_TEVSTAGE2, GX_PASSCLR);
```

The texture coordinate ID that specifies the vector that rotates the tangent map in the camera space is specified as the texture coordinates using `GXSetTevOrder`. The other settings are the same as for TEV stage 1.

The settings for TEV stage 3 are as follows:

Code 15 - TEV Stage 3 Settings

```
GXSetTevOrder(GX_TEVSTAGE3, GX_TEXCOORD3, GX_TEXMAP1, GX_COLOR_NULL);
GXSetTevColorIn(GX_TEVSTAGE3, GX_CC_ZERO, GX_CC_CPREV, GX_CC_TEXC, GX_CC_TEXA);
GXSetTevAlphaIn(GX_TEVSTAGE3, GX_CA_ZERO, GX_CA_ZERO, GX_CA_ZERO, GX_CA_PREV);
GXSetTevColorOp(GX_TEVSTAGE3,
GX_TEV_ADD, GX_TB_ZERO, GX_CS_SCALE_1, GX_TRUE, GX_TEVPREV);
GXSetTevAlphaOp(GX_TEVSTAGE3,
GX_TEV_ADD, GX_TB_ZERO, GX_CS_SCALE_1, GX_TRUE, GX_TEVPREV);
```

Texture coordinates specified using `GXSetTevOrder` will be those mapping the normal vector to the texture space. These texture coordinates result from adding the displacement amounts generated by TEV stage 1 and TEV stage 2. They reference the texture maps. Here the texture maps specify the light maps, so ultimately, the color values of the light maps are given as the texture color input and the alpha values are given as the texture alpha input. Diffuse was stored in the light map color and specular in the light map alpha in the first place, so the bump results are reflected by synthesizing the normal texture color that was stored in the PREV register at TEV stage 0 using `GXSetTevColorOp`.

(Texture Alpha) + ((1-(PREV Color)) * 0 + (PREV Color) * (Texture Color))

The alpha value outputs the PREV color of TEV stage 0 without change, so the normal texture alpha will be used without change.

5.3.7 Binormal and tangent calculations for the polygon model

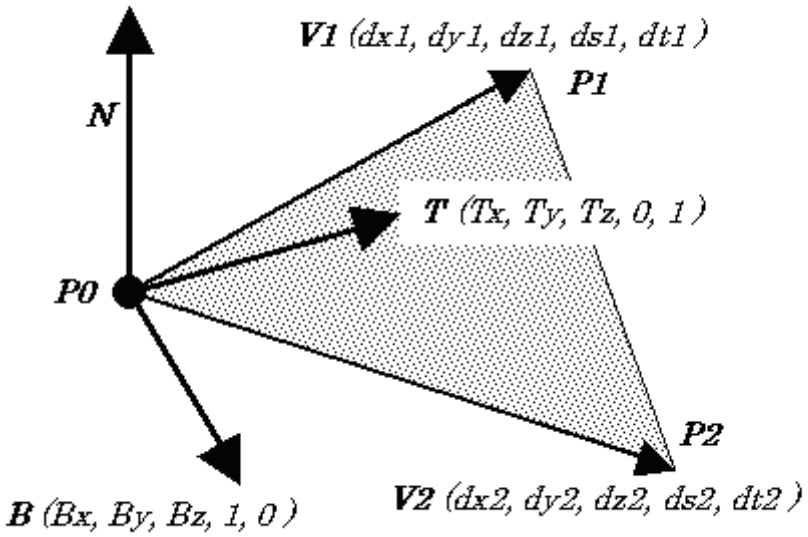
The side vector of the triangular polygon P0-P1-P2 is defined as:

V1 = P1 – P0

V2 = P2 – P0

Normal geometric coordinate values (x, y and z) and texture coordinate values (s and t) are added and the relationships between V1 and V2. The binormal vector (B) and the tangent vector (T) will look like the figure in the diagram below:

Figure 14 - Relationship between V1, V2, binormal and tangent vectors.



Because the binormal and the tangent vectors in bump mapping are given as S and T direction vectors of their respective textures, the elements of S and T become (1, 0) and (0, 1).

B forms a direct line with the normal vector (N), so when the starting point is P0 as in the diagram above, it will be in the same plane as the triangular polygon P0-P1-P2. This gives rise to the following equation.

Equation 7 - Calculation of the Binormal Vector (1)

$$\boldsymbol{B} = a\ \boldsymbol{V1} + b\ \boldsymbol{V2}$$

Substituting the coordinate values from the diagram above into this equation yields the following:

Equation 8 - Calculation of the Binormal Vector (2)

$$a \times dx1 + b \times dx2 = Bx$$
$$a \times dy1 + b \times dy2 = By$$
$$a \times dz1 + b \times dz2 = Bz$$
$$a \times ds1 + b \times ds2 = 1$$
$$a \times dt1 + b \times dt2 = 0$$

Solving for Bx, By and Bx yields the following.

Equation 9 - Calculation of the Binormal Vector (3)

$$Bx = \frac{dx1 \times dt2 - dx2 \times dt1}{ds1 \times dt2 - ds2 \times dt1}$$

$$By = \frac{dy1 \times dt2 - dy2 \times dt1}{ds1 \times dt2 - ds2 \times dt1}$$

$$Bz = \frac{dz1 \times dt2 - dz2 \times dt1}{ds1 \times dt2 - ds2 \times dt1}$$

This yields a normalized B.

$$\boldsymbol{B} = Normalized\ (Bx, By, Bz)$$

Solving for T in the same way:

Equation 10 - Calculation of the Tangent Vector

$$Tx = \frac{ds1 \times dx2 - ds2 \times dx1}{ds1 \times dt2 - ds2 \times dt1}$$

$$Ty = \frac{ds1 \times dy2 - ds2 \times dy1}{ds1 \times dt2 - ds2 \times dt1}$$

$$Tz = \frac{ds1 \times dz2 - ds2 \times dz1}{ds1 \times dt2 - ds2 \times dt1}$$

$$\boldsymbol{T} = Normalized\ (Tx, Ty, Tz)$$

The denominator (ds1 x dt2 – ds2 x dt1) is zero when (ds1, dt1) and (ds2, dt2) are parallel. In this case, the texture coordinates affixed to the polygon do not form a flat surface, so the B and T values are not unique. There are many cases in which the normal vertices are shared by several polygons. In many such cases, B and T will use their average values, so it is often the case that the problem of undefined values can be resolved by removing the values of polygons without defined B and T vectors from the averaging operation

If there is no sharing by several polygons, then it will be necessary to come up with another method. (It is believed that the use of a zero vector for B and T will cause no problems in most cases.)

5.4 Diffuse bump mapping (embossing technique)

Assume the surface *P* is coincident with the *X*-*Y* plane, changes in *u* correspond to changes in the *X-direction*, and changes in the *Y-direction* correspond to changes in *v*. Then *F* can be substituted for *P'*, and the equation for *N'* is:

Equation 11 - Diffuse bump mapping

$$\vec{N}' = \begin{bmatrix} \frac{-\partial F}{\partial u} \\ \frac{-\partial F}{\partial v} \\ 1 \end{bmatrix}$$

Note: *N'* should be normalized, but if the displacements in the bump map are small, the error caused by not normalizing can be ignored.

The diffuse intensity can be found by:

Equation 12 - Diffuse intensity

$$N' \bullet \vec{L} = \frac{\partial F}{\partial u} L_x + \frac{\partial F}{\partial v} L_y + L_z$$

Since most objects will have arbitrary orientations in space, the light direction *L* is transformed into tangent space. Three orthogonal vectors define tangent space: *N*, *B*, and *T*. *B* and *T* lie in the plane of the surface. *T* should be oriented in the direction of the *s*-axis. *T* can then be computed as the cross product of *N* and *B*. Normal *N* is perpendicular to the surface. The light vector must transformed into tangent space using:

Equation 13 - Light vector transform matrix

$$\begin{bmatrix} L_x \\ L_y \\ L_z \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix} \begin{bmatrix} L_x \\ L_y \\ L_z \end{bmatrix}$$

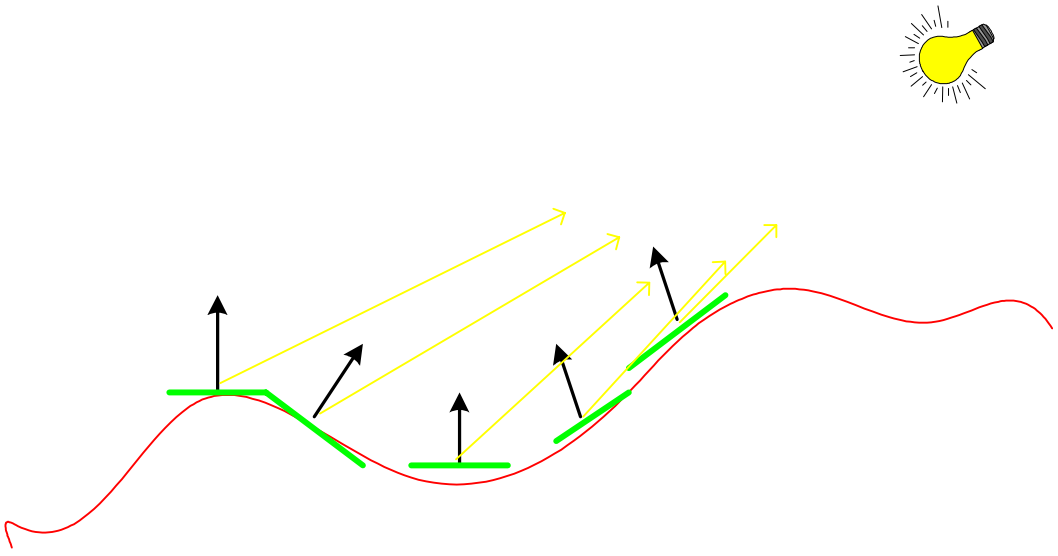
Equation 14 - Light Vector Calculation (1)

From Equations 11 and 12

$$\vec{N}' \bullet \vec{L} = -\frac{\partial F}{\partial u} L_x - \frac{\partial F}{\partial v} L_y + L_z$$

If we assume that “F” is locally flat around the peak in Equation 14, then F(u, v) could be expressed by a primary equation such as in Equation 15.

Figure 15 - Diffuse bump mapping



Equation 15 - Light Vector Calculations (2)

$$F(u,v) = Au + Bv + C \quad \text{((u,v) is near (s,t))}$$

$$\downarrow$$

$$\frac{\partial F}{\partial u} = A$$

$$\frac{\partial F}{\partial v} = B$$

When this is applied to the Equation 14, the following is true.

$$\vec{N}' \bullet \vec{L} = -AL_x - BL_y + L_z$$

$$= (As + Bt + C) - (A(s + L_x) + B(t + L_y) + C) + L_z$$

If (s + Lx, t + Ly) are near (s, t), then the following is true.

Equation 16 - Light Vector Calculation (3)

$$\vec{N}' \bullet \vec{L} = F(s,t) - F(s + L_x, t + L_y) + L_z$$

That is, by taking the difference of the height map with only Lx, and Ly shifted, embossing near the vertices can be expressed.

Note: The L_z component of "Equation 13 - Light vector transform matrix" on page 33 is the dot product of the light direction *L* and the normal *N*. The L_z component can be computed using the regular per-vertex diffuse lighting function.

The steps below outline the emboss bump mapping technique:

1. Beforehand, generate the bump map using a paint tool.
2. Define the normal, binormal, and tangent vectors for each vertex.
3. Generate the bump offsets (*s'*, *t'*) and add them to bump texture coordinates (*s*, *t*) using `GXSetTexCoordGen`. Both (*s*, *t*) and (*s* + *s'*, *t* + *t'*) are iterated across the polygon.
4. Look up the bump image using both (*s, t)* and (*s* + *s'*, *t* + *t'*), then subtract the values per pixel.
5. Add the result of Step 4 to the result of per-vertex local diffuse lighting.
6. Combine the result of Step 5 with material or other textures.

5.4.1 Generating offset texture coordinates

The following discussion assumes that normal, binormal, and tangent vectors are assigned to each vertex. Here the sample demo `tq-emboss` will be described as an example.

The texture coordinate generation feature uses two stages. Enable the two-stage texture coordinate generation feature as described below:

Code 16 - Texture Coordinate Generation Number Settings

```
GXSetNumTexGens(2);
```

The settings are made in the same way as using the normal texture mapping in the first stage. The vertex ST value will be used and the texture will be laid over it without change, so set it up as described below:

Code 17 - Texture Coordinate Generation 0 Settings

```
GXSetTexCoordGen(GX_TEXCOORD0, GX_TG_MTX2x4, GX_TG_TEX0, GX_IDENTITY);
```

This shows that the generated texture coordinates use the texture coordinates that are input as vertex data without changing them.

At the second stage, the results of adding the texture offsets calculated from the specified light, normal vector, binormal vector, and tangent vector to the texture coordinates generated at the first stage are created.

The specific settings would be as follows:

Code 18 - Texture Coordinate Generation 1 Settings

```
GXSetTexCoordGen(GX_TEXCOORD1, GX_TG_BUMP0, GX_TG_TEXCOORD0, GX_IDENTITY);
```

GX_TG_BUMP0, specified as the second argument, indicates that the offset texture coordinates are calculated based on the results of the light computations of the light loaded as **GX_LIGHT0**. Up to eight lights (**GX_LIGHT0** to **GX_LIGHT7**) can be loaded on the Nintendo GameCube, so offset coordinate calculations (**GX_TG_BUMP0** to **GX_TG_BUMP7**) corresponding to those ID's can be used.

GX_TG_TEXCOORD0, specified as the third argument, indicates that the desired texture offset has been added to the texture coordinates calculated with **GX_TEXCOORD0**.

When using **GX_TG_BUMP***, the texture matrix cannot be used, so set **GX_IDENTITY** as the last argument.

This way, the texture coordinates, to which the offset texture coordinates have been added as a result of **GX_TEXCOORD1**, can be used with TEV.

Something that you must be aware of are the sizes of s' and t'. They will correspond to the dot product of the tangent and binormal vectors with the light vectors. However, the normal tangent and binormal vectors are normalized and assigned to the vertex. Also, the light vectors are normalized in the Nintendo GameCube. When dot products have been calculated, a significant displacement will result for s' and t', so it will be necessary to use suitable scaling to produce a natural shift. Normally, suitable application of the normal vector matrix will adjust this displacement amount. The scaled normal vector will be normalized again before being used in the light calculation, so this will cause no variation in the results of the light calculations.

5.4.2 Configuring lighting channel

The diffuse light settings are set up on the lighting channel.

The diffuse light is registered using `GXSetChanCtrl`. The ID of this diffuse light must agree with the ID of the bump map function from the texture coordinate generation.

The diffuse light is registered by loading the light object that was initialized using `GXInitLight*` onto the hardware using `GXLoadLightObj*`. Next, the light calculations in the Nintendo GameCube are made active by setting the flag (light_mask) corresponding to the Light ID that was registered using `GXSetChanCtrl`.

5.4.3 Texture Settings

Put simply, the way the emboss map works is just to subtract the misalignment of the texture coordinates using the light calculations and show the difference with the misaligned texture image as shadow. It applies the embossing effect generally used on 2D tools and elsewhere as a texture.

For this reason, textures are expressed using grey scale. They are made so that the areas closest to white are the higher parts and the areas close to black are the lower parts.

In the sample, the texture was made with grey scale using the I8 format.

5.4.4 Configuring TEV

In TEV, normal texture color is added to the diffuse color and then texture color referenced by the offset texture coordinates is subtracted from that. This leaves the effect of shaded areas and lighted areas.

The following TEV settings are used in the sample.

The number of TEV stages needed is two. The two TEV stages are enabled as follows below:

Code 19 - TEV Stage Number Settings

```
GXSetNumTevStages(2)
```

Next, the grey scale image referenced by the normal texture coordinates is read in during the first TEV stage. Also, the results of the diffuse light calculations are also read in.

Code 20 - First Stage TEV Settings

```
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD0, GX_TEXMAP0, GX_COLOR0A0);
GXSetTevColorIn(GX_TEVSTAGE0, GX_CC_ZERO, GX_CC_TEXC, GX_CC_A0, GX_CC_RASC);
GXSetTevColorOp(GX_TEVSTAGE0, GX_TEV_ADD, GX_TB_ZERO, GX_CS_SCALE_1, GX_DISABLE, GX_TEVPREV);
```

Here, the following processes are carried out on the normal texture and light results.

(Texture Color) * (Bump Scale) + (Rasterizer Color)

The alpha value of the TEV register 0 is used for the bump scale, but this is for the purpose of adjusting the concentration of the shaded area generated using the bump map. In the sample, this is set up so that a number of the stages can be modified, so you should check to make sure that these are correct.

Next, the following types of processes are carried out at the second TEV stage.

Code 21 - Second Stage TEV Settings

```
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD1, GX_TEXMAP0, GX_COLOR_NULL);
GXSetTevColorIn(GX_TEVSTAGE1, GX_CC_ZERO, GX_CC_TEXC, GX_CC_A0, GX_CC_CPREV);
GXSetTevColorOp(GX_TEVSTAGE1, GX_TEV_SUB, GX_TB_ZERO, GX_CS_SCALE_1, GX_ENABLE, GX_TEVPREV);
```

Here the bump texture (same as the previous stage) is referenced using the texture coordinates for which bump calculations were performed. No lighting calculations are performed, so GX_COLOR_NULL is specified. The following types of processes take place as a result.

(Previous Stage TEV Stage Results) – (Texture Color) x (Bump Scale)

That is, the embossing process is carried out by subtracting the color that was shifted by the amount of the bump calculations from the texture color of the previous stage. As a result, positive values and negative values (areas that are struck by light and areas that are shaded) can be obtained. By scaling these using the bump scale values, the positive and negative concentrations of light can be adjusted and this is the result of shading using bump. An embossed bump process can be expressed by adding this to the rasterizer color.

5.4.5 Limitations

It is necessary to generate the texture coordinates in the following sort of functionality order. The order is as follows:

1. All of the conversion processes (`GX_TG_MTX2x4` or `GX_TG_MTX3x4`).
2. The bump mapping (`GX_TG_BUMP0-7`).
3. The texture coordinates (`GX_TG_SRTG`) are generated from the lighting.

Follow this rule even when using each function many times.

Also, it will be necessary to use sequential texture coordinate numbers starting with GX_TEXCOORD0 when generating the texture coordinates.

5.5 Specular bump mapping (environment map technique)

As environment mapping, this method is basically the same as the spherical environment method described in "4.2 Spherical reflection (environment) mapping" on page 16. As bump calculations using indirect texture, instead of using the ST difference texture described above, indirect texture using texture from normal vector perturbation in the XYZ directions are explained.

Note: In the current sample, there was no specular bump mapping performed, but `ind-bump-xyz` serves as an indirect texture sample using the normal vector perturbation texture, so we will provide supplementary descriptions of the necessary areas based on this.

5.5.1 Creating Bump Maps and Environment Maps

The bump map hardware adds -128 to each element of the texel of the normal vector perturbation map as a bias. In this way, the 8-bit normal vector perturbation map of each element can take a range from –128 to +127.

The normal vector perturbation map uses the normal vector perturbation in each of the X, Y and Z directions as the 8-bit color value of A (alpha), B (blue) and G (green). This means that the RGBA8 format is used. R (red) is not used.

Care must be taken in that it is necessary to calculate this normal perturbation vector with an object coordinate system. For this reason, when attempting to express the same bump on each surface of an object like a cube, for example, the texture image to be applied as a bump map will be different for each surface. This makes it necessary to have images corresponding to the texture for each surface at the same time. Moreover, applying this to a more complex geometric shape will require tools for processing texture images in advance. This method has the advantage of using up just one TEV stage on the bump calculation, but the downside is that texture images can become very large or require the use of tools.

Environment maps use spherical reflection maps. Refer to "4.2 Spherical reflection (environment) mapping" on page 16 regarding spherical reflection maps. It will be necessary to generate the environment map dynamically if the camera is to be moving. Refer to "4.1.1 Generating planar reflection maps" on page 14 concerning the dynamic generation of spherical reflection maps.

5.5.2 Providing Normal Vectors for Each Peak

With this type of bump map, there is no need to send out binormal or tangent vectors. Normal vectors are needed for environment map look ups.

5.5.3 Create Texture Coordinates from the Normal Vectors

Normal perturbations that are mapped to texels are simply added to the texture coordinates for the spherical reflection maps that are input as regular texture coordinates. For this reason, only one TEV stage is used.

5.5.4 Indirect Stage Settings

In the indirect stage, a texture ID that loaded the normal perturbation map for reference is given and texture coordinates for applying the normal perturbation map to the object are also given. Use `GXSetIndTexOrder`.

The indirect matrix settings are made with `GXSetIndTexMtx`. For this sort of bump map, a matrix for converting the perturbation of the normal vector to a texture space is created and used as a static matrix. In the indirect feature of the TEV, `GXSetTevIndBumpXYZ` (described above) is used. `GXSetTevIndBumpXYZ` generally uses TEV, so the corresponding TEV stage must be made active.

The internal operation of `GXSetTevIndBumpXYZ` is as shown below:

Code 22 - GXSetTevIndBumpXYZ Source

```
/*---------------------------------------------------------------------------*/
//
// Name:    GXSetTevIndBumpXYZ
//
// Desc:    Set up TEV stages for environment-mapped bump-mapped texture
//          lookup.  The bump map is in 3D object space.
//
// Args:
//
/*---------------------------------------------------------------------------*/

void GXSetTevIndBumpXYZ (GXTevStageID tev_stage, GXIndTexStageID ind_stage,
                             GXIndTexMtxID matrix_sel)
{
    GXSetTevIndirect(tev_stage,         // tev stage
                     ind_stage,         // indirect stage
                     GX_ITF_8,          // format
                     GX_ITB_STU,        // bias
                     matrix_sel,        // matrix select
                     GX_ITW_OFF,        // wrap direct S
                     GX_ITW_OFF,        // wrap direct T
                     FALSE,             // add prev stage output?
                     FALSE,             // use unmodified TC for LOD?
                     GX_ITBA_OFF        // bump alpha select
        );
}
```

5.5.5 Regular TEV Stage Settings

At the regular TEV stage, the offsets generated at the indirect stage are added to the texture coordinates for use with the spherical reflection map given as the regular texture coordinates. That result is used to reference the environment map.

Two TEV stages are used in the sample `ind-bump-xyz`.

During the first stage, the normal color textures are referenced and the results are stored in the PREV register. Here, no indirect textures are used, so the indirect function is disabled using `GXSetTevDirect`.

In the sample, texture coordinates are generated using indirect textures in the second stage of the TEV. The light map is referenced using that result and the results of the bump mapping is obtained by combining it with the material color in the PREV register.

Next, the sample will be modified slightly and the use of an environment map considered.

Basically, referencing the light map also uses the spherical reflection map method, so the operation follows the same procedure up to the texture reference. The only difference is the part involving synthesis in the TEV. The reflectance ratio of the normal color texture and the environment map is 0.0 to 1.0. For example, when this reflectance value is stored in the alpha value of TEV register 0, an environment map can be synthesized by using the following TEV settings:

Code 23 - TEV Settings when Using an Environment Map

```
GXSetTevColorOp( GX_TEVSTAGE1, GX_CC_CPREV, GX_CC_TEXC, GX_CC_A0, GX_CC_ZERO);
```

Moreover, another TEV stage will be used when synthesizing both an environment map and a light map,.

When `GXSetTevIndRepeat` is used and the type of environment map described above is applied, the light map can be referenced with the following TEV stage by using the same indirect texture coordinates a second time. Synthesizing this light map as was done in the sample makes it possible to express bump mapping that applies environment mapping and light mapping.

6 Applying Indirect Textures

6.1 Image warping

There are descriptions of the warping function using indirect textures in the *Revolution Graphics Library (GX)*. Here, we will describe specific methods of use based on the demo, `ind-warp`.

The indirect textures used here have the offset values for the regular look up ST values as texel values. That is, dS and dT are assigned to the normal S and T. When dS and dT are changed, the textures looked up using S + dS and T + dT will have a different shape than when looked up using the original ST.

In the demo, dS and dT are changed when the indirect texture texel values for each frame are written over or when the indirect texture matrices are written over. Also, the dS and dT patterns assigned to the indirect texture as well as the parameters for making modifications in the demo can be changed. There is just one example of how dS and dT can be assigned in the demo, so we will omit any description of what values were assigned. Look at the actual source and the demo.

6.1.1 Settings for Warping

TexObj of the regular textures and the indirect textures are loaded.

At this point, the indirect textures do not need to have any content, but TexObj must be correctly initialized.

Code 24 - TexObj Initialization

```
GXLoadTexObj((Regular Texture TexObj) , GX_TEXMAP0);
GXLoadTexObj((Indirect Texture TexObj) , GX_TEXMAP1);
```

Warping requires one TEV stage and one indirect stage.

The texture coordinate generation settings are exactly the same as for normal textures.

The settings for the indirect texture look up are made as follows:

Code 25 - Indirect Texture Look Up Settings

```
GXSetIndTexOrder(GX_INDTEXSTAGE0, GX_TEXCOORD0, GX_TEXMAP1);
GXSetIndTexCoordScale(GX_INDTEXSTAGE0, GX_ITS_4, GX_ITS_4);
```

In the demo, the indirect map is 64 x 64, in contrast with the image map, which is 256 x 256. For this reason, the texture coordinates S and T were scaled down to 1/4. However, if they are to be the same size, then there is no need for `GXSetIndTexCoordScale`.

The indirect texture process is set up as follows:

Code 26 - Indirect Texture Process Settings

```
GXSetTevIndWarp(GX_TEVSTAGE0,       // tev stage
                GX_INDTEXSTAGE0,    // indirect stage
                GX_TRUE,            // signed offsets?
                GX_FALSE,           // replace mode?
                GX_ITM_0);          // ind matrix select
```

The internal operation of this function is as shown below. With this function, the indirect matrix ID is specified, but because the content of the matrix is not specified, it must be set up using `GXSetIndTexMtx`. In the sample demo, the input ST values were changed by setting the matrices for each frame.

Code 27 - GXSetTevIndWarp Source

```
/*---------------------------------------------------------------------------*/
//
// Name:    GXSetTevIndWarp
//
// Desc:    Used for simple indirect texture warping.
//
// Args:
//
/*---------------------------------------------------------------------------*/

void GXSetTevIndWarp (GXTevStageID tev_stage, GXIndTexStageID ind_stage,
                          GXBool signed_offset, GXBool replace_mode,
                          GXIndTexMtxID matrix_sel)
{
    GXIndTexWrap wrap = (replace_mode) ? GX_ITW_0 : GX_ITW_OFF;

    GXSetTevIndirect(tev_stage,         // tev stage
                     ind_stage,         // indirect stage
                     GX_ITF_8,          // format
                     (signed_offset) ? GX_ITB_STU : GX_ITB_NONE, // bias
                     matrix_sel,        // matrix select
                     wrap,              // wrap direct S
                     wrap,              // wrap direct T
                     FALSE,             // add prev stage output?
                     FALSE,             // use unmodified TC for LOD?
                     GX_ITBA_OFF        // bump alpha select
        );
}
----------------------------------------------------------------
```

The argument `replace_mode` is a parameter related to the ST that is output. For `GX_TRUE`, the regular ST = 0 and an ST calculated using the indirect circuit will be output. For `GX_FALSE`, the ST calculated for the regular ST using the indirect value will be added as an offset before being output. In the demo, only `GX_FALSE` was used.

For `GX_TRUE`, the signed_offset adds an offset of –128, which will make the input ST values of (0, 256), (-128, +127). In the demo, `GX_TRUE` was used in order to assign +/- changes on the regular ST values.

Next, the TEV settings will be set up.

Code 28 - TEV Settings

```
  GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD0, GX_TEXMAP0, GX_COLOR_NULL);
  GXSetTevOp(GX_TEVSTAGE0, GX_REPLACE);
```

There are no particular settings. GX_TEXCOORD0, which is specified here, specifies the texture coordinates that reflect the results of the indirect texture process.

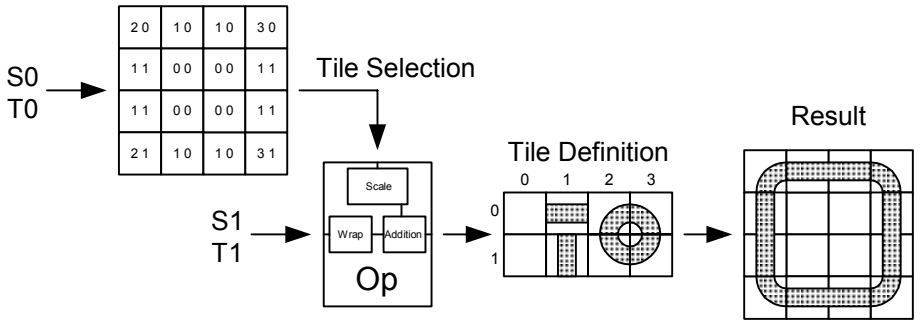
This concludes the settings and the actual modifications that are carried out by varying the texel values of the indirect textures and the values of the indirect matrices.

6.2 Tile Texture

There are descriptions of the tile function using indirect textures in the *Revolution Graphics Library (GX)*. Here we will describe specific methods of use based on the demo `ind-tile-test`.

We will start by defining the terms.

Figure 16 - Tile Texture



The diagram above is the same as the *Revolution Graphics Library (GX)*. The square on the far left is called a tile map and the individual squares in the area divided up into 2 squares by 4 squares labeled “Tile Definition,” are called tiles.

Tile: Think of this part as being a fixed-size square having several types of designs. These tiles are lined up horizontally and vertically to create the “result” shown on the far right.

Tile Definition: All of the types of tiles used are lined up to form a single surface texture. These textures are called “Tile Definitions.”

Tile Map: These are textures that have data telling how to line up each tile as texel data. Tiles are specified using ST values for the tile definitions. These ST values are not texture ST values. The upper left tile in the tile definition is (0, 0) and they assign the “(s, t)” of the m^{th} tile from the left and the n^{th} tile from the top as (m-1, n-1).

Here, the tile and texture mapping method that uses tile maps will be called a “tile texture.”

6.2.1 Creating a Tile Map

The tile definition and tile map are created from the entire image that will actually be drawn. The tile definition becomes the normal texture and the tile map becomes the indirect texture. As textures that also create indirect textures, these are basically the same a normal textures. The formatting is also the same and wrapping and mirroring are possible.

In the demo, each tile is 32 x 32 texels in RGBA8 format and the tile map is in 128 x 64 IA4 format.

Each texel in the tile map needs to have the same number of bits per texel as the ST data to select the tile. However, with indirect textures, “T” is stored in the Blue channel and “S” in the Alpha channel of the normal texture. Red and Green are not used in tile textures. The texture format to be used is selected taking the required number of bits for the ST into consideration. In the demo, 4-bit is sufficient for both S and T, so the IA4 format is used.

The tile map takes the Nintendo GameCube format, so the tile data cannot be created by arranging it without modification in a 2D configuration. Create a texture image following the format. See the *Revolution Graphics Library (GX)*.

In the demo, the types of tiles are assigned to ASCII characters, which are arranged in a 2D configuration to define the tile map. The IA4 texture is created based on this.

6.2.2 Texture Settings

The settings for the texture of the tile map and the tile definition are made using `GXInitTexObj` and `GXInitTexObjLOD`. Precautions include the inability to use MIP maps and the use of only GX_NEAR for filtering in tile maps.

After initializing, the texture of the tile map and the tile definitions are each loaded. In the demo, the tile definition is loaded into GX_TXMPAP0 and the tile map is loaded into GX_TEXMAP1.

6.2.3 Texture Coordinate Settings

Texture coordinates can be shared in tile definitions and tile maps. The texture coordinates ST of the indirect texture can be used by scaling them down using `GXSetIndTexCoordScale`, a geometric series of halves (from 1/1 - 1/256), for the texture coordinates in the TEV stage.

The TEV stage texture coordinates must be given not as tile definition textures, but as the texture coordinates of the map to actually be drawn. For this reason, scaling is done using `GXSetTexCoordScaleManually` so that the tile size x tile map will result.

When `GXSetTexCoordScaleManually` is not used, GX_TEXCOORDx, linked with `GXSetTevOrder` will be automatically scaled according to GX_TEXMAPx. However, when it is used, the scaling can be set so that it is fixed.

In the sample demo, the tile size is 32 x 32 and the tile map is 128 x 64, so the settings are made as follows:

Code 29 - GXSetTexCoordScaleManually

```
GXSetTexCoordScaleManually( , GX_TRUE, 128*32, 64*32);
```

The argument **GX_TRUE** enables the feature that assigns the scale manually, and keeps scaling from taking place automatically. The indirect S and T are both set to 1/32 and sized to 128 x 64 as shown below:

Code 30 - GXSetIndTexCoordScale

```
GXSetIndTexCoordScale( , GX_ITS_32, GX_ITS_32);
```

6.2.4 Indirect Stage Settings

In the sample demo, the indirect stage settings are as follows:

Code 31 - Indirect Stage Settings

```
GXSetNumIndStages(1);
GXSetIndTexOrder(GX_INDTEXSTAGE0, GX_TEXCOORD0, GX_TEXMAP1);
```

As with TEV, the stage count is set and then the stage number, the texture coordinates, and the map ID are set.

6.2.5 Tile Settings

Here, the TEV stage and indirect stage are linked and the tile size, spacing, indirect texture formatting, and indirect matrix ID are specified.

In the demo, these are specified as follows:

Code 32 - Tile Settings

```
    GXSetTevIndTile(GX_TEVSTAGE0,       // TEV Stage
                    GX_INDTEXSTAGE0,    // Indirect Stage
                    32,                 // S Direction Tile Size
                    32,                 // T Direction Tile Size
                    32,                 // S Direction Tile Spacing
                    32,                 // T Direction Tile Spacing
                    GX_ITF_4,           // Indirect Texture Format
                    GX_ITM_0,           // Matrix Selection
                    GX_ITB_NONE,        // Not Used
                    GX_ITBA_OFF);       // OFF
```

`GXSetTevIndTile` sets up the indirect matrices using `GXSetIndTexMtx` based on these arguments, and calls `GXSetTevIndirect`. **`GX_ITB_NONE`** and **`GX_ITBA_OFF`** are always set. The internal operation of these functions is shown below:

Code 33 - GXXSetTevIndTile Source

```
/*---------------------------------------------------------------------------*/
// Name:    GXSetTevIndTile
// Desc:    Used for indirect texture tiling and pseudo-3D texturing.
/*---------------------------------------------------------------------------*/
#define IND_TEX_MTX_EXP     10
#define IND_TEX_MTX_SCALE   (f32)(1 << IND_TEX_MTX_EXP)

void GXSetTevIndTile (GXTevStageID tev_stage, GXIndTexStageID ind_stage,
                          u16 tilesize_s, u16 tilesize_t,
                          u16 tilespacing_s, u16 tilespacing_t,
                          GXIndTexFormat format, GXIndTexMtxID matrix_sel,
                          GXIndTexBiasSel bias_sel, GXIndTexAlphaSel alpha_sel)
{
    GXIndTexWrap        wrap_s, wrap_t;
    f32                 mtx[2][3];

    ASSERTMSG(tev_stage < GX_MAX_TEVSTAGE, "GXSetTevIndTile: Invalid tev stage id");
    ASSERTMSG(ind_stage < GX_MAX_INDTEXSTAGE, "GXSetTevIndTile: Invalid indirect stage id");
    // Compute wrap parameters from tilesize
    switch (tilesize_s) {
        case 256:       wrap_s = GX_ITW_256;    break;
        case 128:       wrap_s = GX_ITW_128;    break;
        case 64:        wrap_s = GX_ITW_64;     break;
        case 32:        wrap_s = GX_ITW_32;     break;
        case 16:        wrap_s = GX_ITW_16;     break;
        default:
            ASSERTMSG(0, "GXSetTevIndTile: Invalid tilesize for S coordinate");
            wrap_s = GX_ITW_OFF;
            break;
    }

    switch (tilesize_t) {
        case 256:       wrap_t = GX_ITW_256;    break;
        case 128:       wrap_t = GX_ITW_128;    break;
        case 64:        wrap_t = GX_ITW_64;     break;
        case 32:        wrap_t = GX_ITW_32;     break;
        case 16:        wrap_t = GX_ITW_16;     break;
        default:
            ASSERTMSG(0, "GXSetTevIndTile: Invalid tilesize for T coordinate");
            wrap_t = GX_ITW_OFF;
            break;
    }

    // compute the matrix using tilespacing values.
    mtx[0][0] = ((f32) tilespacing_s / IND_TEX_MTX_SCALE);
    mtx[0][1] = mtx[0][2] = 0.0f;
    mtx[1][1] = ((f32) tilespacing_t / IND_TEX_MTX_SCALE);
    mtx[1][0] = mtx[1][2] = 0.0f;
    GXSetIndTexMtx(matrix_sel, mtx, IND_TEX_MTX_EXP);

    GXSetTevIndirect(tev_stage,         // tev stage
                     ind_stage,         // indirect stage
                     format,            // format
                     bias_sel,          // bias select
                     matrix_sel,        // matrix select
                     wrap_s,            // wrap direct S
                     wrap_t,            // wrap direct T
                     FALSE,             // add prev stage output?
```

```
                    TRUE,               // use unmodified TC for LOD?
                    alpha_sel);         // bump alpha select
}
```

Here, the indirect texture format of the arguments will specify how many of the upper texel bits will be used. The specified bits will be shifted and used. For example, if 3 bits are specified, xxx00000 will become 00000xxx, so caution is required when creating tile map texel data.

In the demo, **GX_ITF_4** is set up to use the upper 4 bits. The tile map is IA4, so texel data will go into the upper 4 bits (Alpha and Blue). (The upper 4 bits are copied into the lower 4 bits.) This is how these 4 bits are specified for use. This means that the 4-bit texel value will be used without modification. For the indirect matrix, the scale matrix for converting from the tile map to the ST values will be set up according to the tile spacing. In the demo, the scale was $32/2^{10}$ (2^{10} is equivalent to ST 1.0) for both S and T.

6.2.6 TEV Stage Settings

There is nothing in particular you need to be careful of here. The settings will use the texture coordinates set at first.

6.2.7 Drawing Polygons

The entire image to actually be drawn should be set up as a single rectangle. In the sample demo, (S, T) were made (-1, 1) – (2, 2) and the tile map was applied using a 3 x 3 wrap.

6.3 Pseudo-3D textures

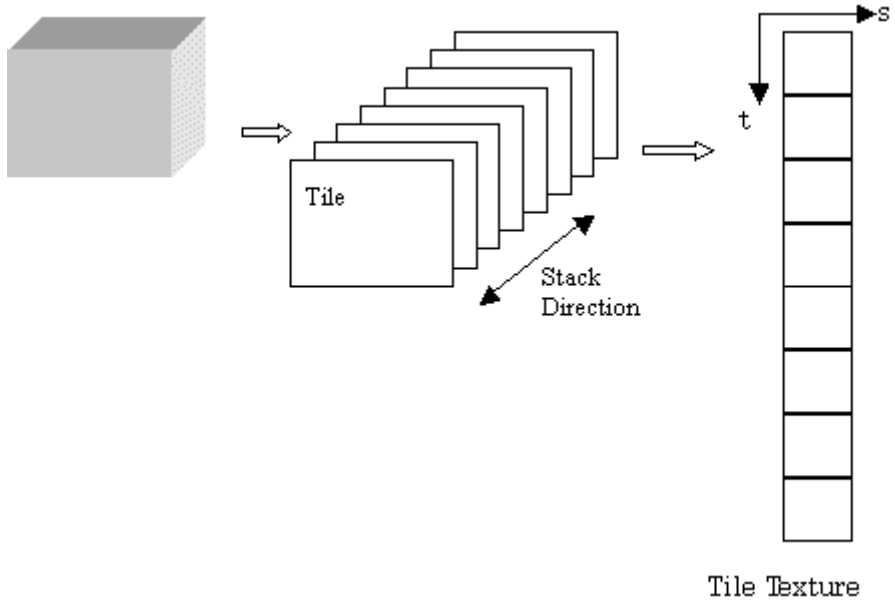
There is a description of the pseudo-3D texture function using indirect textures at the beginning of the *Revolution Graphics Library (GX)*. Here, we will describe specific methods of use based on the demo, `ind-pseudo-3d`.

6.3.1 Pseudo-3D Textures

In contrast with 2D textures, which define textures as rectangles, 3D textures, also called solid textures, define textures using cuboids. Here, the pseudo-3D textures to be described do not define the texture across the entire range of the cuboid. Instead, they slice the cuboid at certain intervals and define the slices as a 2D texture. The area between the slices is expressed as a fill-in (blend) of 2D textures. Expressing colors as changing discontinuously is impossible, but when qualities change continuously, a similar sort of expression is possible.

Here, the 2D textures corresponding to the slices are stacked in a direction perpendicular to the tiles and slices. That direction is called the stack or the stack direction. The tiles are lined up in a single row on a flat surface, forming a single texture surface.

Figure 17 - Pseudo-3D Texture



In the demo, a model similar to mountain terrain is drawn. The stack is taken in the direction of the mountain's height. For the coordinates of this model, a 40 x 40 mesh was made on the xy plane and each mesh point was given height data using the z coordinates. Each tile is 64 x 64.

The texture data for the stack direction is given by indirect textures. The selection of tiles from the coordinates of the object, to which the 3D texture will actually be applied, is used in the conversion to the blend coefficients of the two tiles. Indirect textures only require one-dimensional data in the stack direction. Two pieces of data, the stack direction position and the blend coefficients, should be stored in the texel, so it should be possible to use the IA format. In the demo, IA4 is used. Since it is one-dimensional data, a width of one texture will be sufficient, but single-texture widths cannot be used with Nintendo GameCube. Therefore, the narrowest texture corresponding to the texture format is used. IA4 was used in the demo, so a 4-texel width is used.

6.3.2 Tile Textures

In the demo, eight 64 x 64 tiles are used with all eight lined up in the T direction forming a single surface of texture.

The texture coordinates are not scaled for this entire texture. Instead, the scale values are assigned manually so that scaling will take place on an individual tile.

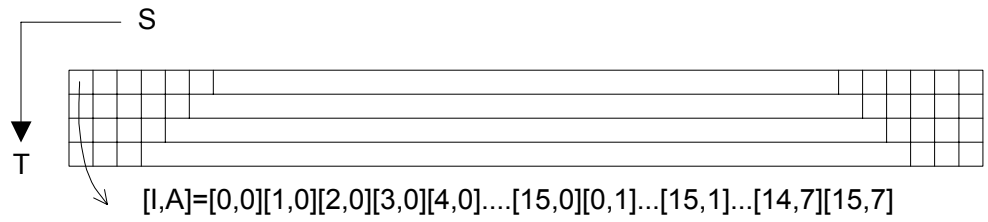
Code 34 - Assigning Scale Values

```
GXSetTexCoordScaleManually(GX_TEXCOORD0, GX_ENABLE, TIW-1, TIH-1);
```

6.3.3 Creating Indirect Textures

In the sample demo, a 128 x 4 IA4 texture is made. The tile selection data is held in the I channel, while the A channel holds the blend coefficients. If the 6th and 7th textures were to be blended, the tile selection data would include as data, a 6 and the values 0 through 8 for the S direction as the stack direction. The blend coefficients are held in alpha. The values 0 through 15 are held repeating in the S direction. These values are used as bump alphas in indirect texture processing. Bump alphas are selected using the indirect texture settings and the spaces between textures is blended by specifying the bump alphas as the TEV input color C.

Figure 18 - Creating Indirect Textures



Indirect Texture

The process flow in TEV is shown below:

(1) TEV Stage 1

1. Texture coordinate generation is used to convert from the Z coordinates of the model to S coordinates for indirect stage use. The T coordinate is 0. The matrix used for this conversion is assigned by **updateMatrices**.
2. The texel I is obtained using an indirect texture look up.
3. I is entered into the B and G channels. At the indirect stage, it becomes S and T without any changes. The indirect matrix conversion changes these to S = 0 and T = (the coordinates corresponding to the first blended tile of the tile texture).
4. This ST value is added to the regular texture coordinates and the color of the tile texture is looked up.

(2) TEV Stage 2

1. Even at the second stage, the texel I, which is obtained from the indirect texture look up, is used and the color look up of the tile texture is performed in exactly the same way. However, GX_ITB_T has been specified in the bias selection, so the T direction will undergo a +1 bias. Based on this, the tile to follow the first stage in the stack direction will be looked up.
2. The texel A, obtained with the indirect texture look up, is output as 8-bit data after undergoing a 4-bit left shift due to the bump alpha selection (see "Code 35 - TEV Stage 2" on page 50). With the settings described in Code 35, bump alpha will become the TEV input color C.
3. The following types of calculations are performed in the TEV, completing the tile blend.

   (Stage 1 Output Color) x (1 – Bump Alpha) + (Texture Color) x (Bump Alpha)
4. When the argument **GX_ALPHA_BUMP** is set to **GX_ALPHA_BUMPN**, the bump alpha output will not be entered into the TEV as is. It will be multiplied by 255/248, that is it will be normalized to the range of 0 - 255 before being entered.

Code 35 - TEV Stage 2

```
GXSetTevOrder(,,, GX_ALPHA_BUMP);
GXSetTevColorIn(,,, GX_RASC, );
```

Like the tile textures, the function used at the first and second stage indirect texture settings is `GXSetTevIndTile`. The usage differs as follows:

1. Based on the tile definition, either S or T will be set to 0 in the tile spacing.
2. For the second stage settings, **GX_ITB_T** (or **GX_ITB_S**) is used for the bias selection and assigned +1.
3. In order to assign the above bias, the texture formatting can only be 3, 4 or 5-bit formatting.
4. Bump alpha is specified with the second stage settings.

All other cautions are the same as for the tile textures.

(3) TEV Stage 3

In the sample demo, the light GX_COLOR0A0 has been applied here.

Lastly, we will summarize how the 3D textures are assigned to the polygon.

The color in a certain position on a tile that corresponds to the polygon is controlled using the ST value, which is assigned to the vertex of the polygon. The stack position corresponding to the polygon can be controlled with the texture matrix coefficient from the conversion of the indirect stage S coordinates from the position coordinates and the value held in the indirect texture texel I.

7 Full-scene shadow mapping

In this chapter, we describe ways to realize full-scene shadow mapping using two methods based on the sample demo. In order to understand this chapter, you need to understand texture projection and how TEV works. Refer to "3 Texture projection" on page 5 in this section and the *Revolution Graphics Library (GX)*.

7.1 Depth-based full-scene shadow mapping

This method makes a determination by comparing the distance between light objects and the Z value. This method will be described based on the sample demo `tq-shadow3`. In this sample, it is possible to switch between two types of precision shadow depth maps for the sake of comparison, but our description here will concern 8-bit precision.

7.1.1 Overview of the Algorithm

First, a scene is rendered from the light perspective. The Z value is saved as a texture. (This is called a “shadow depth map.”) A “ramp texture” is also created statically, for the purpose of calculating the distance in advance.

Next, a suitable look up is performed based on the ramp texture for calculating the distance from the light to the peak of the object. The shadow depth map is then applied to the scene using texture projection from the light. The shade determination can be made by comparing these two texels.

7.1.2 Shadow Depth Map Generation

For the shadow depth map, the Z buffer area is used from a scene rendered from the perspective of the light with the color update OFF and the Z update ON. The point to bear in mind here is running the front face culling when the scene is being rendered. This creates shade along the back side of the object as viewed from the light. This method is also sometimes called “second depth shadow mapping.” The effect of this method depends on the precision of the Z value and associated errors. When the Z value of the back side of the object (as seen from the light) is obtained, a trace shadow is sometimes generated on the back side due to precision-related matters. In order to avoid this, a shadow depth map is created for the back side of the object (as seen from the light). Doing so will prevent shadows from appearing on the front surface and, even if those sorts of shadows appear on the back side, they will not stand out because they become the shadow of the lighting.

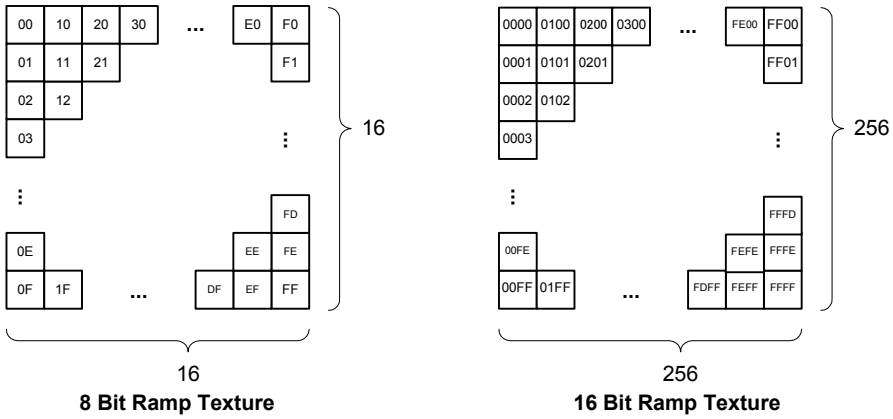
Note: This is conditional upon the object being completely closed and not extremely thin.

In this way, the Z buffer of the rendered scene is copied into the texture using GX_TF_Z8. In actual use, it is loaded using **GX_TF_I8** and applied as a shadow depth map.

7.1.3 Ramp Texture Look Ups

Ramp texture is a grey scale gradation texture that is created statically.

Figure 19 - Ramp Textures



Using a suitable matrix, a look up is performed.

Perspective Projection (Point Light Source type of Effect)

s = (1 + Near/z) * Far / (Far - Near)

t = s * tscale

Orthographic Projection (Parallel Light Source type of Effect)

s = - (z + Near) / (Far - Near)

t = s * tscale

The ramp texture width goes into the tscale. This is multiplied by the model view matrix of the perspective from the light to generate the texture coordinates. This makes it possible to look up suitable ramp textures.

7.1.4 TEV Settings for Drawing Shadows

In order to make the shadow areas visible from the projected shadow depth map, you need three TEV stages: “Ramp Texture Look Up,” “Comparison” and “Color Selection and Mixing.”

7.1.4.1 TEV Stage 1

Code 36 - TEV Stage 1

```
// TEV Stage 0  ( Loads a depth value from ramp texture )
// REGPREV(C) = TEX(I)
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD0,
              GX_TEXMAP0, GX_COLOR_NULL);
GXSetTevColorIn(GX_TEVSTAGE0, GX_CC_ZERO, GX_CC_ZERO,
                GX_CC_ZERO, GX_CC_TEXC);
GXSetTevColorOp(GX_TEVSTAGE0, GX_TEV_ADD, GX_TB_ZERO,
                GX_CS_SCALE_1, GX_TRUE, GX_TEVPREV);
```

First of all, the distance between the light and object is calculated. The ramp texture is used to calculate the distance. The ramp texture is looked up using the appropriate matrix in this first stage, which is then passed without change to the second stage.

7.1.4.2 TEV Stage 2

Code 37 - TEV Stage 2

```
// TEV Stage 1  ( Compare with shadow map texture )
// REGPREV = ( REGPREV > shadow map texture(R) ? 255 : 0 )
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD1,
              GX_TEXMAP1, GX_COLOR_NULL);
GXSetTevColorIn(GX_TEVSTAGE1, GX_CC_CPREV, GX_CC_TEXC,
                GX_CC_ONE, GX_CC_ZERO);
GXSetTevColorOp(GX_TEVSTAGE1, GX_TEV_COMP_R8_GT, GX_TB_ZERO,
                GX_CS_SCALE_1, GX_FALSE, GX_TEVPREV);
```

The results of the ramp texture look up in the previous stage and the projected shadow depth map are compared. The shadow depth map is applied using the texture coordinates generated for projecting and mixed with the output from the previous stage. The comparison is made as follows:

> Comparison Results = [Output of Previous Stage (Distance of Light and Object) > [Z Value Projected Shadow Depth Map] ? 255 : 0

A Z value for the shadow depth map that is too small based on the distance calculated using the ramp texture, will be taken as an obstruction in the way and cause a shadow.

The color used here is 8-bit grey scale and in the TEV, the same thing goes into the RGB. Here, only the R component was used in the comparison (GX_TEV_COMP_R8_GT.) (For 16-bit comparison, I in IA8 is broadcast to R and A is broadcast to G. Then GX_TEV_COMP_R8_GT is used.) The results of this comparison are passed along to the next TEV stage.

7.1.4.3 TEV Stage 3

Code 38 - TEV Stage 3

```
// TEV Stage 2  ( Select shadow/lit color )
// output = ( REGPREV == 0 ? rasterized color : shadow color )
// Register 0 is supporsed to hold shadow color
GXSetTevOrder(GX_TEVSTAGE2, GX_TEXCOORD_NULL,
              GX_TEXMAP_NULL, GX_COLOR0A0);
GXSetTevColorIn(GX_TEVSTAGE2, GX_CC_RASC, GX_CC_C0,
                GX_CC_CPREV, GX_CC_ZERO);
GXSetTevColorOp(GX_TEVSTAGE2, GX_TEV_ADD, GX_TB_ZERO,
                GX_CS_SCALE_1, GX_TRUE, GX_TEVPREV);
```

At the third stage, the drawing settings for the scene that will actually be viewed are set up based on the second stage shadow determinations. The only item to be specified in `GXSetTevOrder` is the color channel that enables the lighting.

Input operands include the color of the shadow set up in the register using `GXSetTevColor` when the model is drawn, rasterizer color that is lighted, and the output of the previous TEV stage.

Here, the normal TEV calculations are performed as follows using GX_TEV_ADD.

> Results of Output = (0 + ((1.0 - results of previous stage) * shadow color + results of previous stage * rasterizer color) + 0) * 1

That is, if the result of the previous stage was 1, and the rasterizer color (area without shadow) is 0, then the shadow color (shadow area) will be applied.

7.1.5 Advantages and Disadvantages of this Algorithm

The advantage of this algorithm is that it makes self-shadowing (shadowing on the object itself) possible, because it performs the measurements using the actual distances.

Disadvantages include the generation of unnatural aliases due to the accuracy of the shadow depth map, and that three TEV stages are used.

7.2 ID-based full-scene shadow mapping

In this method, a shadow determination is made using the ID assigned to each of the objects. This method is described based on the sample demo `tq-shadow2`.

7.2.1 Overview of the Algorithm

First, a scene is rendered from the perspective of the light, where each of the drawn objects has been filled in with an ID number, and stored as a texture. (This is called a “shadow ID map”.)

Next, the texture projection from the light is applied to the shadow ID map and the scene is rendered from the camera's perspective. Here, the ID of the object and the shadow ID map projected onto that object are compared in the TEV. If the two ID values are different, the beam of light from the light will be obstructed by a polygon with a different ID, forming a “shadow”.

7.2.2 Generating a Shadow ID Map

A shadow ID map is created in the first rendering pass, but at this point, the object is drawn with the ID as a color. Each time an object is drawn, the ID is incremented and that is considered a grey scale. A rendering of this grey scale scene from the perspective of the light becomes the shadow ID map.

7.2.3 TEV Settings for Drawing Shadows

In order to make the shaded area visible from the projected shadow ID map, a two-stage TEV (ID Comparison) and (Color Selection and Mixing) is necessary.

7.2.3.1 TEV Stage 1

Code 39 - TEV Stage 1 (tg-shadow2.c)

```
// TEV Stage 0  ( Color0(R) == Texture0(R) ? 255 : 0 )
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD0, GX_TEXMAP0, GX_COLOR0A0);
GXSetTevColorIn(GX_TEVSTAGE0, GX_CC_TEXC, GX_CC_RASC, GX_CC_ONE, GX_CC_ZERO);
GXSetTevColorOp(
    GX_TEVSTAGE0,
    GX_TEV_COMP_R8_EQ,  // R8 equal compare mode
    GX_TB_ZERO,         // actually N/A
    GX_CS_SCALE_1,      // actually N/A
    GX_TRUE,
    GX_TEVPREV );
```

First of all, in the first stage, the shadow ID map projection and the ID comparison processes are run. Refer to the *Revolution Graphics Library (GX)* regarding the TEV Comparison Process.

In `GXSetTevOrder`, the texture coordinates for projecting the shadow ID map textures, the shadow ID map and the color channels for overwriting the object with the ID are specified. The effect of the light is eliminated for these channels in advance using the `GXSetChanCtrl` light mask.

Next, the TEV input operand is specified using `GXSetTevColorIn`. Here the texture color and rasterizer color (that is the color that indicates its ID) are specified.

Finally, the input operands for each process method are specified. The equation here would be as follows:

Results of Comparison = 0 + (([Shadow ID Map Texture Color] = [ID of filled in object] ? 255 0)

The color used here is 8-bit grey scale. In the TEV, the same colors are used in the RGB so the comparison is actually made using just the 8-bit R component (GX_TE_COMP_R8_EQ). The results of this comparison are passed along to the next TEV stage.

7.2.3.2 TEV Stage 2

Code 40 - TEV Stage 2 (tg-shadow2.c)

```
// TEV Stage 1  ( REGPREV == 0 ? Reg0 : Color1 )
// Register 0 is supporsed to hold shadow color
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR1A1);
GXSetTevColorIn(GX_TEVSTAGE1, GX_CC_C0, GX_CC_RASC, GX_CC_CPREV, GX_CC_ZERO);
GXSetTevColorOp(
    GX_TEVSTAGE1,
    GX_TEV_ADD,
    GX_TB_ZERO,
    GX_CS_SCALE_1,
    GX_TRUE,
    GX_TEVPREV );
```

At the second stage, the drawing settings for the scene that will be actually viewed are established based on the shadow determination in the first stage. The color channel that enables the lighting is all that is specified using `GXSetTevOrder`.

The four input arguments include the shadow color that is set up in the register using `GXSetTevColor` when the model is drawn, the lighted rasterizing color, and the output from the previous TEV stage.

The normal TEV calculation GX_TEV_ADD here would be calculated as follows:

Results of Output = (0 + ((1 - results of previous stage) * shadow color + results of previous stage * rasterizer color) + 0) * 1

That is, if the result of the previous stage is 1, and the rasterizer color (the area without shade) is 0, the shadow color (the shaded area) will be applied.

7.2.4 Advantages and Disadvantages of this Algorithm

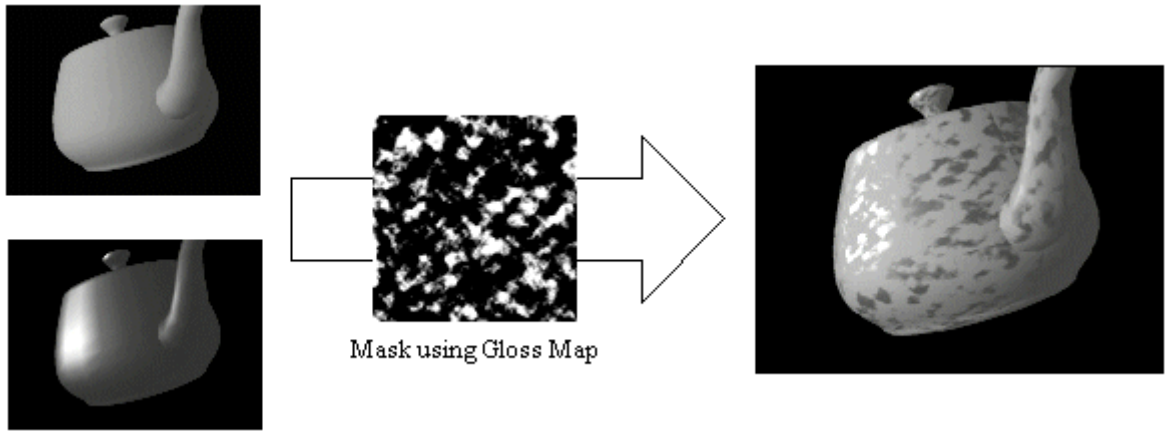
This algorithm manages each object separately, so shadowing onto the object itself (self-shadowing) is not possible.

However, compared with the depth method, a digital comparison object is used as ID, making a simple comparison process possible. Moreover, no unnatural aliasing is caused by the Z value precision. Furthermore, only two TEV stages are used.

8 Gloss maps

Gloss maps allow for surfaces that are non-uniformly specular (e.g., wet or waxy surfaces). Typically, gloss maps are encoded in the alpha channel of a texture and used to modulate the result of specular lighting.

Figure 20 - Gloss Map Image Diagram



We have prepared textures for use in grey scale gloss maps as a specific process. (This could also be the alpha area of a color texture.) What this does is to separate the areas to be masked or not based on the specular light. The effect is that the masked area is struck by diffuse light, while the unmasked area seems to be hit by specular light. The following would be how these are expressed as TEV settings:

Code 41 - Gloss Map: TEV Stage 1

```
GXSetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD_NULL, GX_TEXMAP_NULL, GX_COLOR0A0);
GXSetTevColorIn(GX_TEVSTAGE0, GX_CC_ZERO, GX_CC_ZERO, GX_CC_ZERO, GX_CC_RASC);
GXSetTevColorOp(GX_TEVSTAGE0, GX_TEV_ADD, GX_TB_ZERO, GX_CS_SCALE_1, GX_FALSE, GX_TEVPREV);
```

Code 42 - Gloss Map: TEV Stage 2

```
GXSetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD0, GX_TEXMAP0, GX_COLOR1A1);
GXSetTevColorIn(GX_TEVSTAGE1, GX_CC_CPREV, GX_CC_RASC, GX_CC_TEXC, GX_CC_CPREV);
GXSetTevColorOp(GX_TEVSTAGE1, GX_TEV_ADD, GX_TB_ZERO, GX_CS_SCALE_1, GX_FALSE, GX_TEVPREV);
```

Diffuse light goes into color 0 and specular light goes into color 1. At the first stage, the results of the rasterization are applied using the diffuse light of color 0 and passed along unchanged to the second stage. (Here, since this is the description of just the gloss map, it is passed along unchanged, but actually, this is where you would apply the normal textures or mix colors, as necessary.)

At the second stage, the gloss map GX_TEXMAP0 and the results of rasterizing the specular light from color 1 are applied. This gloss map is a grey scale texture in an I8 or similar format. The results of the diffuse lighting from the previous stage of this texture and the results of the specular lighting are blended as follows:

> Output Results = (Diffuse Light Results + ((1.0 - Gloss Map Value) x Diffuse Light Results + Gloss Map Value x Specular Light Results) + 0) x 1

In this example, the specular portion and the diffuse portion take up two lights. If the color portion and the alpha portion of the same light were used, it would be possible to have just a single light. Naturally, in this case, the positions of the diffuse light and the specular light would be shared. Also, no light attenuation can be used.

Because the term “gloss map” refers to the area made shiny by the texture, there are other conceivable applications. For example, it would be possible to express such things as portions of the reflections of the lights of a city at night in the puddles on a road's surface or a partly rusted iron tool by running an environment map with the specular areas above.

9 Cartoon lighting

The Graphics Processor allows texture coordinates to be generated based upon the red (R) and green (G) components from a given lighting channel. This allows you to post-process the lighting computations with a texture map to create interesting effects. One such effect is cartoon lighting, wherein the color palette is limited, and shading is limited to sharp transitions between colors in the palette. As an example, the trunk of a tree may be shaded using only three colors: light brown, dark brown, and black.

To achieve cartoon lighting, you set up a local diffuse light that operates only on the R component. The G component may then be used to select a given material color. The texture map then represents a series of 1D functions that convert an intensity (from the R component or *s*-index) into a given output color, with a different function for each material (from the G component or *t*-index).

Passing the color components correctly requires that the light be R and that the G component be incorporated through the ambient part of a channel's lighting equation. Thus, the material may be sent down through the vertex-color G component, or through the given channel's ambient color register G component.

You must be careful about the filtering mode chosen when using cartoon lighting. Problems may result due to the fact that the *s*- and *t*-axes are used for independent factors, but the filtering mode applies to both axes simultaneously (i.e., you cannot choose one filtering mode for *s* and a different one for *t*). If you use `GX_NEAR` for the *min_filt* and *mag_filt* modes, then such problems may be avoided; however, a certain amount of aliasing may result. Using `GX_LINEAR` with a single LOD may help slightly, but it cuts in half the number of materials that are available. You must duplicate each entry along the *t*-axis in order to ensure that linear interpolation between adjacent *t* values has no effect on the final output. Thus, the one-dimensional filters for adjacent values of *t* will be identical (see details in table below). Mipmap filtering with multiple LODs is possible, but this also reduces the number of materials that are available. The least-detailed LOD will determine the number of materials that will be available.

Another complication results from the method of converting the color value into a texture coordinate. The color value produced from the rasterizer will be an 8-bit integer in the range 0-255. This is converted into a floating point number by dividing it by **255**. However, the value is converted into a texture coordinate by multiplying it by the texture **size**. Therefore, you must pay careful attention to the texture coordinate for choosing the one-dimensional table. Assuming a texture size of 256 and `GX_NEAR` filtering, this will map values in the following manner:

Table 1 - GX_NEAR filtering

Color Value	Converted Texture Coordinate	Nearest Value
0	0	0
64	64.251	64
127	127.498	127
128	128.502	129
192	192.753	193
254	254.996	255
255	256	255

Due to the conversion process, coordinate value 128 is skipped, and color values 254 and 255 map to the same coordinate value. If we assume a texture size of 256 and `GX_LINEAR` filtering, we have the following mappings:

Table 2 - GX_LINEAR filtering

Color Value	Converted Texture Coordinate	Coordinates Looked Up
0	-0.5	0, 0
1	0.504	0, 1
2	1.508	1, 2
3	2.512	2, 3
4	3.516	3, 4
126	125.994	125, 126
127	126.998	126, 127
128	128.002	128, 129
253	253.492	253, 254
254	254.496	254, 255
255	255.5	255, 255

It is safe to use *N* * 2 to convert a table ID into a color value; however, the tables must be stored in a non-straightforward manner within the texture. The table for *N* = 0 must be stored at coordinate 0 only; tables for *N* = 1 to *N* = 63 must be stored at *N* * 2 - 1 and *N* * 2, and tables for *N* = 64 to *N* = 127 must be stored at *N* * 2 and *N* * 2 + 1. Coordinate 127 may be left empty (it will not normally be accessed).

10 Outlining

Outlining involves drawing dark lines around the silhouettes of objects. It is another common cartoon effect that is useful for making objects more distinct when shading is kept to a minimum. There are various ways to achieve outlining. One way is to simply draw dark lines along silhouette edges. Of course, this requires software to detect which edges are silhouette edges and which are interior ones.

Another method involves mapping the surface normal into a texture map that darkens the underlying surfaces for normals that are nearly perpendicular to the view direction. This method is demonstrated in `tg-light-fx`. The disadvantage to this method is that large surfaces that are close to edge-on will appear darkened, instead of outlined as desired.

A more effective way to produce outlines involves drawing the surfaces using object IDs, then post-processing the ID image to detect differences which indicate silhouette edges, as follows:

1. Set up the frame buffer to include an alpha channel.
2. Draw the scene normally, except include object IDs in the alpha channel.
3. Write the alpha image to a texture.
4. Set up the Graphics Processor to write black pixels based upon alpha texture differences.
5. Draw a screen-sized quad using the alpha texture to write horizontal outlines.
6. Draw the quad again to write vertical outlines.

Some attention to detail is necessary when assigning IDs to surfaces. Since the alpha channel is limited to 8 bits, IDs may need to be reused. In fact, only seven bits of alpha can actually be used, since the eighth bit is needed as a sign bit for the threshold operation (described later). This is okay as long as different objects with identical IDs do not overlap.

In addition, if you want silhouettes on overlapping sections of concave objects, a more complex ID and threshold system is required. One way to do this is to assign IDs that vary by 1 to different parts of the same object, with a total difference of at least 2 for possibly overlapping sections. The threshold function would then only create silhouettes for differences of at least 2.

When writing to a texture buffer, there are synchronization problems that may occur. Before utilizing the texture to draw the outlines, you must be sure the texture has been copied completely. `GXPixModeSync` can perform this synchronization. In addition, once the frame buffer has been copied to texture memory, the Graphics Processor's texture cache will need to be flushed.

Finding the differences in the ID images requires computing $abs(A_0 - A_1)$, then comparing this value with a given threshold to determine whether or not to write black pixels. The A_0 value is the alpha value for a given pixel, while the A_1 value is the alpha value for a neighboring pixel. Both of these values are looked up from the alpha texture using one set of texture coordinates and two different texture matrices. One matrix is set to the identity, while the other is set to identity plus a translation.

In order to perform the difference computation, the TEV is set up to subtract one alpha value from the other and write the unclamped result to the output register. The alpha compare unit is then set up to detect alpha values that exceed the threshold.

Note: Unclamped negative results from the TEV turn into alpha values with the eighth bit set (for example, -1 becomes 255). Thus for a threshold of 2, you must write only pixels with an alpha difference greater than 2 and smaller than (256-2). The color value to be written is set to come from a register, *not* the texture.

In the first pass (writing horizontal outlines), the texture matrices should be set up to shift the alpha texture image one pixel (texel) vertically. In the second pass (writing vertical outlines), the texture matrices should be modified to shift the alpha image one pixel (texel) horizontally. Thicker outlines can be achieved by making the shifts larger than one pixel. Do not make the outlines too thick though, or visual anomalies will result.

11 Geometric decals

The Graphics Processor hardware has special provisions for drawing co-planar geometry without Z buffer problems. This means that it is not necessary to raise decal geometry from the underlying surface to avoid drop-outs, nor is it necessary to render decals using the same polygons as the underlying surface. The problem is solved in the GP by allowing polygons to share the same Z-plane coefficients. This guarantees that all decals will be *exactly* co-planar with the underlying surface. Ordered drawing is still required to make sure that decals properly overwrite the underlying surfaces.

The procedure for drawing decals is as follows:

1. Make sure the Z-compare mode is `GX_LEQUAL`.
2. Draw a reference polygon.
3. Call `GXSetCoplanar(GX_TRUE)`. The Graphics Processor locks in its Z-plane coefficients.
4. Draw the decal polygons from bottom to top. Each decal shares the exact same plane as the reference polygon.
5. Call GXSetCoplanar(GX_FALSE).

Some care must be taken in choosing the reference polygon. Avoid choosing a polygon that may become trivially rejected, in which case its Z-plane coefficients would not be calculated. A good choice is a polygon that encompasses all of the co-planar polygons; it will not be rejected unless all the relevant polygons are off-screen.

The reference polygon does not have to be a visible polygon. In fact, you can make the reference polygon invisible by setting the polygon-culling mode to `GX_CULL_ALL` prior to rendering it. The use of invisible reference polygons allows more flexibility in the drawing order of different decals. This is important, since texture-caching issues suggest that rendering should be done in texture-sorted order for best performance. Therefore, you could modify the procedure above to loop first over the different types of decals, then have an inner loop for the surfaces where the decals are applied, using an invisible reference polygon for each surface.

12 Z textures

Z-textures are those which have Z-values in the texture texels, not colors and not alphas.

It is possible to apply the use of 2D sprites having Z values and pre-rendered images in 3D spaces in image-based rendering. This will not be covered here, so refer to the various reference materials.

The procedure and precautions for using Z-textures are summarized here.

Procedure for Using Z-Textures

1. Creation of Z-Textures
2. Settings
3. Initialization and Loading of Textures
4. Drawing

12.1 Creating Z-Textures

The Z values of Z-textures are values that are compared directly with the value in the Z buffer. For this reason, when copying from the Z buffer in the graphics process to create Z-textures, the content may be used without change, but when creating them offline using a CG or other tool, it will be necessary to calculate a suitable Z-value. Creation using a CPU is also conceivable, but the cautions are the same as the latter.

`GXCopyTex` is used when copying from the Z buffer. GX_TF_Z24X8 or other GX_TF_Z* and GX_CTF_Z4 or other GX_CTF_Z* are specified using `GXSetTexCopyDst`. When these have been specified, the Z buffer will be copied as textures. In this case, the Z value can only be read out as 24-bit, so Z-textures can only be created properly using a 24-bit Z buffer, that is, a non-antialiasing frame buffer mode.

12.1.1 Formatting

Textures are handled as I8 for 8-bit, IA8 for 16-bit, and RGBA8 for 24-bit. When creating them externally, they are made in one of the texture formats I8, IA8, or RGBA8. For formatting details, refer to the *Revolution Graphics Library (GX)*.

12.1.2 Z Values

Z values written to the Z buffer are the Z values following projective transformation. Therefore, when creating offline, the Z values have to be calculated with this transformation in mind.

With 24-bit, the value written to the Z buffer is the same scale, so for perspective projection, the Z values have to be calculated by converting the position coordinates corresponding to each texel from a camera space to a screen space. For orthographic projection, the same sort of calculations can be made, but the Z coordinates can also be defined without any relation to the three-dimensional coordinates of the camera space. For example, in the `tev-ztex` demo, the Z values are defined as being the Z coordinates from 0 to 2^{24} without modification. When this is done, the Z values of the Z-textures can be completed just by considering the Z coordinates.

The same is true for 16-bit and 8-bit, but in this case, the Z values will be relative to the polygon using the Z-textures and, depending upon the application, there will probably be times when a suitable amount of depth information can be had without conducting a stringent projection transformation. Naturally, the drawing itself will take place using coordinates that have undergone a projection transformation, so be careful that there is not a linear correspondence between the Z value and the Z coordinates of the model space or camera space.

12.2 Drawing Mode Settings

The frame buffer mode can use everything. In modes where the Z buffer is 16-bit, it is just like the Z value of the polygon. After undergoing 24-bit calculations, it will be converted to 16-bit when written to the Z buffer, so the Z values that are treated like Z-textures can always be thought of as being 24-bit linear irrespective of the mode.

Be sure to set the Z buffer to active.

Code 43 - Z Buffer Activation

```
GXSetZMode(GX_ENABLE, *, *);
```

Think of updating the Z buffer or not in the same way as polygon drawing. The Z-textures are only used in the final TEV stages, so the Z comparison is normally set to take place after the texture process.

Code 44 - Comparison Process Sites

```
GXSetZCompLoc(GX_FALSE);
```

Unless you have something particular in mind, it should be sufficient to clear the Z buffer with the maximum value.

Code 45 - Clearing the Z Buffer

```
GXSetCopyClear(*, 0xffffff);
GXCopyDisp(*, GX_TRUE);
```

The frame buffer is cleared using `GXCopyDisp`.

12.3 Texture Object Initialization

This is about the same as normal textures. The differences between this and normal textures are: The formats are GX_TF_Z8, GX_TF_Z16 and GX_TF_Z24x8, and no texture filter can be used.

12.3.1 Texture Object Initialization

Specify Z-texture as the format and turn the MIP map to OFF.

Code 46 - Texture Object Initialization

```
GXInitTexObj(*, *, *, *, GX_TF_Z*, *, *, GX_FALSE);
```

... GX_NEAR is specified for filter mode.

Code 47 - Filter Mode Settings

```
GXInitTexObjLOD(*, GX_NEAR, GX_NEAR, 0, 0, 0, GX_FALSE, GX_FALSE, GX_ANISO_1);
```

12.4 Drawing

This draws a polygon covered with a Z-texture. The differences between this and normal drawing of polygons are: The Z-texture settings and the Z-textures are input at the final TEV stage.

GXLoadTexObj(*, *); // Load Z-Textures

GXSetZTexture (Processing Mode, Z-Texture Format, Bias)

...Processing Mode

GX_ZT_DISABLE: Polygon Z Value

GX_ZT_ADD: Add Z value of Z-Texture to Z Value of Polygon

GX_ZT_REPLACE: Replace Z Value of Polygon with Z Value of Z-Texture

GX_ZT_ADD is a mode for assigning the Z value, relative to a polygon, using 8-bit or 16-bit Z-textures. GX_ZT_REPLACE becomes the Z value of the Z-texture's Z value in relation to the scene.

The Z-texture format specifies the format that is specified by the texture object initialization.

For GX_ZT_ADD, adding the Z value of the Z-texture to the Z value, yields a Z value that is further back than the polygon, but by using bias, it is possible to put the Z value of a Z-texture in front of the polygon or to have it straddle the polygon.

The final TEV stage is as follows:

Code 48 - Final TEV Stage

```
GXSetTevOrder(GX_TEVSTAGE*, GX_TEXCOORD0, Map ID with Z-Texture Loaded, *);
```

When calculating colors and alphas, no texture colors or texture alphas can be used, but other input may be used. Set up the following in an appropriate manner: `GXSetTevOrder`, `GXSetTevColorIn`, `GXSetTevAlphaIn`, `GXSetTevColorOp`, `GXSetTevAlphaOp`, etc.

12.5 Reference

In the `tev-ztek` demo, according to the initial settings, the tree texture settings were made for –128 to +127 around the 8-bit, Z texture polygon, while the ground surface used 24-bit Z texture. The surface of the water was drawn using 3D square shapes.

By switching the Z texture processing mode, the bias effect and the differences between GX_ZT_ADD and GX_ZT_REPLACE can be seen.

Refer to the *Revolution Graphics Library (GX)*, as it contains descriptions of Z textures.

13 Stitching

Skinning is the process of joining together portions of surface geometry which are associated with different transformation matrices. It is typically used with skeletal animation systems to produce continuous “skin” over the skeleton transformation hierarchy. The ideal support for skinning involves having multiple different matrices associated with each vertex of the skin polygons, weighted by how close a vertex is to a given “bone.” The Graphics Processor (GP) supports a subset of this functionality called *stitching*. Only one matrix may be used to transform a given vertex, but different matrices can be used for different vertices.

Stitching is based on the idea that it is faster to generate intermediate bone matrices directly from interpolated animation data than it is to multiply every vertex by multiple matrices and weights (effectively interpolating bones).

Using one matrix per vertex allows boundary polygons to stretch to cover joints; however, the stretching is not necessarily smooth. The joint skin can be smoothed by adding more matrices (bones) to the joint, and perhaps by adding more polygons to transition among the various matrices.

While it is true that stitching usually requires more matrices to achieve a smooth joint, the GP only needs to perform a single matrix multiply for each position, and the method of interpolation is not required to be linear (unlike typical skinning hardware).

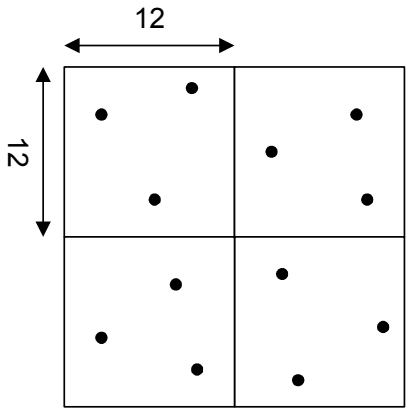
14 Antialiasing algorithm

Nintendo GameCube uses super sampling to perform antialiasing. The super sampling method computes *n* pixel samples per pixel and computes a weighted average to make the final color for this pixel.

Nintendo GameCube super sampling uses the super sampling EFB format and has three samples per pixel. The sample locations are programmable. The weight coefficients for blending these samples to form the final pixel are also programmable.

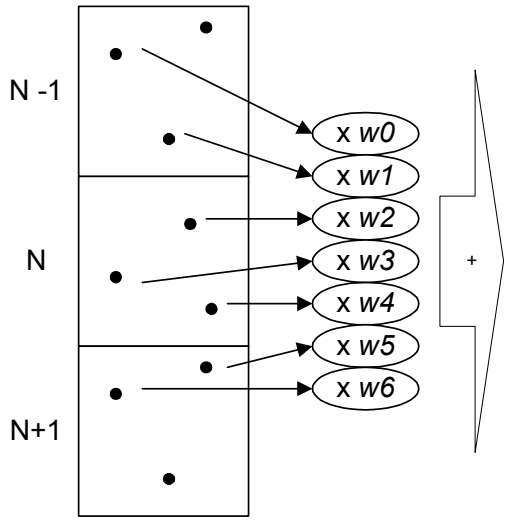
Figure 21 - Antialiasing

Sample Pattern



- Define 3 samples per pixel.
- Define patterns for 2x2 pixel group.

Vertical Filter Blending



- Define 3 weights from current pixel, 2 above, 2 below.

The following function sets the sample patterns and blending coefficients for this copy operation.

Code 49 - GXSetCopyFilter

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

14.1 Antialiasing impact on performance and quality

Although antialiasing increases visual quality on the polygon edges and intersections, it does cost performance and Z quality.

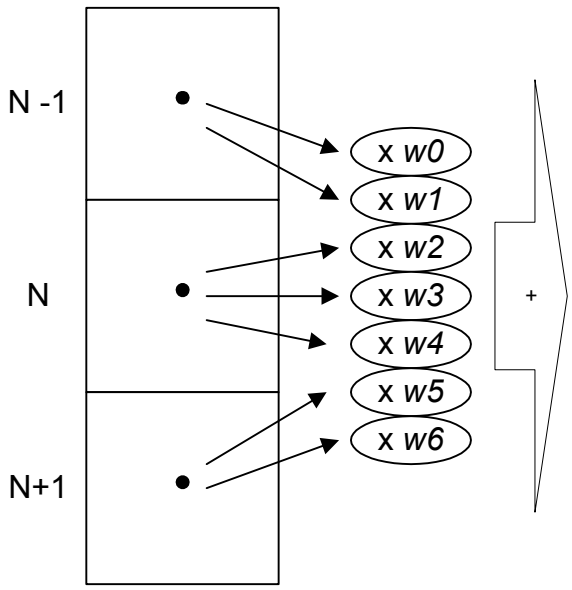
Antialiasing requires a 96-bit super sampling EFB format, which reduces Z buffer precision to 16 bits, rather than 24 bits as in other formats.

Antialiasing also reduces peak fill rate from 648 million pixels/second to 324 million pixels/second. However, if you are using more than one TEV stage, this reduction is hidden. Two TEV stages also reduce the pixel fill rate to 324 million pixels/second.

14.2 Non-antialiasing

For non-antialiasing, the sample patterns are not programmable. The hardware uses only the center of the pixel; the values of `GXSetCopyFilter(sample_pattern)` are ignored. The weights are programmable. They are set as follows:

Figure 22 - Non-antialiasing blending



As shown in the figure above, (*w0*, *w1*) should be set to the same value, (*w2*, *w3*, *w4*) set to the same value, and (*w5*, *w6*) set to the same value.

14.3 Deflicker algorithm

Deflickering is typically used to solve two problems:

- To eliminate flickering of thin (one-pixel tall) horizontal lines for interlaced video display. A one-pixel tall horizontal line will flicker at 30Hz as the TV interlaced video shows this line every other field.
- To perform simple antialiasing by rendering 480 lines at 60Hz in the frame buffer and deflickering to 240 lines for interlaced display. This is basically 2-sample super sampling.

The vertical filter hardware is used for deflickering purposes, so antialiasing and deflickering are similar concepts.

15 Multi-resolution geometry

To achieve optimum performance, you must apply level of detail (LOD) methods to scene geometry. Often, multiple versions of a model at various resolutions are kept on hand, and the proper model to display is selected based on some decision criteria or *LOD metric*, like range from the viewpoint.

This method of LOD has several problems. Since there are multiple versions of each model, more data memory is required. At the instant that you replace one LOD with another one, the player can detect a momentary “pop.” If you use some method of blending between the two LODs to avoid this pop, you temporarily increase the load on the Graphics Processor, and the point of LOD is to lower the load. Also, there is no fine control of the LOD, since each LOD requires more memory.

Another method of LOD is called *multi-resolution geometry* or *continuous LOD*. In this method, a single vertex list that represents the highest resolution required is stored in memory. A separate list of connectivity information is stored in another array. The connectivity list is modified at runtime to merge small triangles into larger triangles (thus reducing the level of detail in the model), or to split large triangles into smaller ones (increasing the level of detail). It is the responsibility of an offline modeling program (and the modeler) to make sure that the transitions at each step are smooth and continuous. The runtime program is responsible for determining the current LOD metric and editing the connectivity list as required.

Continuous LOD has the advantage of usually requiring less overall data than separately-modeled LODs do. In addition, the transitions between LODs can be very smooth, since they take place one polygon at a time (at a minimum). On the other hand, this type of LOD requires more runtime processing, as well as careful consideration of other attributes like color and texture coordinates.

The Graphics Processor supports continuous LOD in several ways. First, there is native support for indexed geometry. In addition, you can “disable” a primitive in a display list by making *all* the indices of the primitive equal to 0xff (8-bit index) or 0xffff (16-bit index).

Note: There is no way to disable a single vertex. Failure to disable all the indices of a primitive will result in undefined behavior.

16 CPU direct access to the on-chip frame buffer

The embedded frame buffer (EFB) is mapped to memory; however, CPU writes to the EFB go through parts of the pixel pipeline. This means that the alpha-compare, Z-compare, blending, and dithering units can condition such writes. Consequently, there are function calls to set how each of these units will behave for CPU writes to the EFB.

Writing to the EFB is complicated by cache and bus transaction issues. Uncached writes can be used to write color or Z values, but not both together (for the same pixel address). Therefore, when using uncached writes, you cannot conditionally write a color based upon an associated Z value, nor can you conditionally write Z based upon an alpha value. Writing color and Z requires at least a 64-bit bus transaction, which can only occur through the CPU cache.

Using the regular cache to access the EFB is very tricky due to coherency issues. Since values can be modified as they are written to the EFB, any reads which come from the cache may not necessarily reflect what is in the EFB. We recommend using the locked-down cache mechanism when writing bulk data to the EFB.