DOT3 Normal Mapping on the PS2

Morten Mikkelsen
IO Interactive
mm@ioi.dk

November 4, 2004

Abstract

This paper describes a method for doing PC-style normal mapping on the Playstation 2 by taking advantage of the GS and VU1 units. Two variations are described: A cheap two-pass solution without per-pixel normalization and a per-pixel normalized alternative which requires four passes.

1 Introduction

Bump mapping is a technique originally developed by Blinn in 1978 [Blinn78], where the surface normal is perturbed by information stored in a two dimensional bump map. While bump mapping perturbs the existing normal of a model, normal mapping [Cohen98] replaces the normal entirely by doing a look-up into a normal map, which usually is a texture with tangent-space normals stored as RGB. Both are inexpensive ways to fake geometric shading detail on low resolution models.

While normal mapping today is standard on X-Box and PC hardware platforms [Kilgard00], it has not yet been done satisfactorily on the PS2. The aim of this paper is to show how to achieve normal mapping on the PS2 with a visual quality comparable to the PC and X-Box.

The following section describes a related method and discusses its limitations. Section 3 describes the proposed approach which is split in two parts: One without per-pixel normalization (see section 3.5) and one with per-pixel normalization (see section 3.6). Section 4 discusses the results. Finally, section 5 draws the conclusion. The reader is expected to be familiar with the PS2 architecture and the general concepts of normal mapping as known from the PC.

2 Previous work

In 2002 Mark Breugelmans [Breugelmans02] suggested a method for achieving bump mapping on the PS2. The clever part of this method is how it computes the sum $R + G + B$ on the GS. The weakness is how the signed multiplications are handled. Mark splits the normals into a positive and a negative side by using two palettes. The components in either palette are clamped to zero if they are of the opposite sign. The vectors towards the light source (transformed into tangent space) are delivered in the vertex colors TL. They are also split into a positive and a negative side. With NM being the normal map the equation to obtain the signed multiplication becomes four passes:

$$NM_{pos} \cdot TL_{pos} + NM_{neg} \cdot TL_{neg} - NM_{neg} \cdot TL_{pos} - NM_{pos} \cdot TL_{neg}$$

One problem is that TL is clamped to zero per vertex and not per pixel which may result in incorrect dot products. Figure 1 illustrates the interpolation problem.

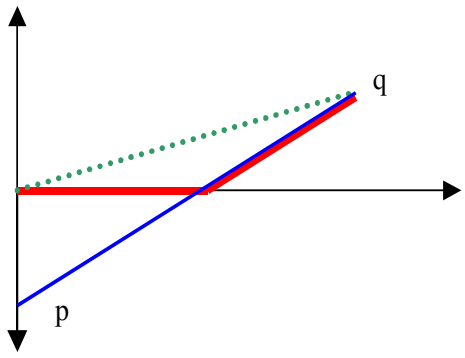


Figure 1: The blue line represents the interpolation from signed to unsigned values $y1 = t \cdot q + (1-t) \cdot p$. The thick red line is what we would like to have, which is the blue line clamped to zero per pixel $y2 = Max(t \cdot q + (1-t) \cdot p, 0)$. The green dotted line is what we actually get using per vertex clamping $y3 = t \cdot Max(q, 0) + (1-t) \cdot Max(p, 0)$.

This approximation results in artifacts for a wide range of 3D models since it is assumed that every triangle has the same tangent space assigned

to all three vertices and that the light is a directional light. This means it usually works well for flat models but fails for smooth shaded models.

3 The Approach

This paper proposes a different way to achieve signed multiplications on the GS which improves the speed and quality compared to [Breugelmans02]. Two methods are presented: One that does the signed multiplication in two passes without the per vertex clamping problem. And an alternative method which does it in four passes and also normalizes the *tolight* vectors per pixel using a sphere map look-up. The first method uses a positive/negative side strategy. The second method uses a traditional normal map as known from the PC (centered at 128) but with a reorganized palette.

Both methods work with directional, spot, and point light sources.

3.1 Overview

The two normal mapping methods share the same general rendering steps. The only difference between the two is how they achieve signed multiplications. In the general case, two buffers are needed: A 32 bit light accumulation buffer (LAB) and a 32 bit dot product buffer (DPB). The rendering steps (1-4) for rendering normal mapped objects are as follows:

1. Render all the visible normal mapped objects to fill the z-buffer and to set the ambient color in the LAB.
2. For every light hitting the objects:
   - 2.1 Clear the DPB to 0x00808080. This pass is free for every light after the first one (as will be explained in section 3.4).
   - 2.2 Disable color clamp.
   - 2.3 Render all objects that are hit by the current light so that signed multiplications are delivered to red, green, and blue.

2.4 Apply a 2D post filter pass over the DPB to achieve $R + G + B$, resulting in the final dot product.

2.5 Enable color clamp.

2.6 Add the lighting contribution of the DPB to the LAB. To do this the DPB is read as an 8 bit texture with the dot products as texels. We use an intensity look-up palette (ILP) to add the dot product to all 3 channels of the LAB. The ILP entries of the negative dot products are set to zero. The look-up is finally multiplied by the color of the light and added to the LAB.

3. We are done with the DPB. Clear it to zero and render all objects unlit with its diffuse texture.

4. Multiply the buffer (unsigned) of the diffuse layer with the LAB: $(red_{diff} \cdot red_{light}, green_{diff} \cdot green_{light}, blue_{diff} \cdot blue_{light})$

5. Render all non-normal mapped geometry, as one generally would, into the same buffer as the one containing the result of the multiplication in step 4.

Even if there is not enough VRAM for two draw buffers, it is still possible to use the techniques in this paper. Instead, use a single light and save the dot product layer in a free alpha channel like the one in the display buffer. The intensities may be applied once it seems convenient during the rendering pipeline. For a single light source, pre-filling the z-buffer (step 1) is not necessary.

Alternatively, 2 to 3 lights can be used by storing their dot product layers in unused alpha pixels in the VRAM. Then use the frame buffer as the LAB. Once the LAB is done, copy the red, green, and blue to the free alphas and render the unlit diffuse buffer. Afterwards, at step 4, apply the LAB by using these alphas. Note, this is <u>not</u> necessary if more than one draw buffer is available.

3.2 Achieving signed multiplication with unsigned input without the per vertex clamping problem

Assume we have two signed values a and b and we wish to calculate the product $a \cdot b$ (implicit signed shift right by 7). Since the input is 8 bit, we assume $a \in \{-127, -126, ..., 128\}$ and $b \in \{-128, -127, ..., 127\}$ so that $a \cdot b \in \{-128, -127, ..., 127\}$. We create two intermediate values a_2 and b_2 by the equations (1) and (2):

$$\begin{aligned} a_2 &= 128 - a && (1) \\ b_2 &= b + 128 && (2) \\ a \cdot b &= (128 - a_2) \cdot (b_2 - 128) \\ &= 128b_2 - a_2b_2 + 128a_2 - 128^2 && (3) \end{aligned}$$

We can rearrange a little and take advantage of the fact that every multiplication has an implicit shift to the right by 7:

$$a \cdot b = (128 - a_2)\, b_2 + a_2 - 128 \quad (4)$$

Now (4) is quite close to an equation that can be computed using the GS blend mode function. Furthermore, a_2 and b_2 are both 8 bit unsigned inputs.

Alternatively, the signed product can be expressed as (5):

$$a \cdot b = Max(a, 0) \cdot b_2 - Max(-a, 0) \cdot b_2 + (Max(-a, 0) - Max(a, 0)) \quad (5)$$

The last term, however, is still signed. In order to fix that, we create another intermediate value c_2 and rewrite (5) to:

$$\begin{aligned} c_2 &= 128 + (Max(-a, 0) - Max(a, 0)) && (6) \\ a \cdot b &= Max(a, 0) \cdot b_2 - Max(-a, 0) \cdot b_2 + c_2 - 128 && (7) \end{aligned}$$

On the GS clamping to 8 bit is performed once after the texture function operation and then clamping/wrapping is performed at the end of the blend mode operation. We can complete the dot product with full 8 bit precision by taking advantage of simple modulo tricks. These are explained in section 3.3.

3.3 Adding R+G+B.

Leaving out the subtraction by 128 in equation (4) yields:

$$a \cdot b + 128 = (128 - a_2)\, b_2 + a_2 \tag{8}$$

Applying (8) on the channels of the DPB to do the signed multiplications of the per pixel dot product, the final dot product may be obtained by using equation (9)[1]:

$$\begin{aligned} dotprod &= (r + g + b) - 128 - 128 - 128 \\ &= (r + g + b) - 128 \end{aligned} \tag{9}$$

Similar calculations can be made for (7), by again leaving out the subtraction by 128:

$$a \cdot b + 128 = Max(a, 0) \cdot b_2 - Max(-a, 0) \cdot b_2 + c_2 \tag{10}$$

Using (10) the final result can again be obtained by (9).

As mentioned in step 2.6, an ILP is used to add the dot products to the LAB. Assuming the pixels we look-up in the DPB are the final dot products, the ILP must contain 1, 2, 3, ..., 128 in red, green, and blue of the first 128 entries. For free clamping, we keep zero in the last 128 entries.

It is possible to simplify equation (9) to the sum of $r + g + b$. Since the result is used as a palette look-up, we can simply compensate by reordering the palette, so the new palette is a simple permutation of the original palette.

Leaving out the subtraction by 128 in (9) just offsets the entries of the ILP by 128.

The main principle of fetching any channel in a 24/32 bit frame buffer on the PS2 is using the buffer as an 8 bit texture twice the width and twice the height. Looking at the tables in section 8.3 of the GS users manual [Sony02] makes it clear that real-time swizzling is needed to get this to work.

[1] Note that all calculations are made using modulo by 256 on the GS with color clamping disabled.

The good news is that this can be done using a selection of pretesselated sprites and the region repeat mode. The details of how to fetch channels in a 24/32 bit buffer on the PS2 are available on the playstation2-linux website [Breugelmans01].

Unlike [Breugelmans01] which keeps the GS primitives in a DMA chain, this paper's implementation uses a VU1 program which tessellates the primitives required to do the operation. The same primitives are then reused for every page. This is done by delivering the settings of the FRAME, TEX0 and ZBUF registers for every page. The unpacking is done using difference mode and STCYCL(3,1) on the VIF1. For every sequence, the ROW register is set to the value of the first register and then only the VRAM pointer differences are unpacked to keep the chain small. This is done double buffered and 40 page settings are delivered per source buffer. In addition, the mask of the FRAME register is set in the COLUMN register so the mask may be set independently from the rest of the chain.

3.4 Clearing the DPB

As mentioned in step 2.1 of the overview, it is necessary to initialize the DPB to 0x00808080 before rendering the signed multiplications. The reason is that pixels that are not rendered to should have an intensity level of zero after look-up into the ILP.

When using multiple lights, one trick is to clear the DPB by setting the z-buffer (24 bit) to point to the DPB when adding the contribution of the current light to the LAB (step 2.6). The depth is set to 0x00808080 and the TEST register is set to all pixels pass. This involves reading from the DPB as an 8 bit texture and writing to it as a z-buffer. There are two ways to make this work without getting texels overwritten before they are read:

- Set the XYZ2s of the sprites according to the Z24/Z32 layout by setting them so the four 32x16 regions in the pages (a page is 64x32) are swapped along the diagonals.
- Alternatively, render to the LAB in PSMZ32.

They both give the same result. It works because the z-buffer is forced to write to pixels inside the 32x16 region that is currently being read as a texture (and is already cached). This eliminates the deleting of pixels in 32x16 regions that have not yet been read. Of course this also means the contribution gets added to the LAB in PSMZ32 layout. This can be fixed by rendering the signed multiplications in the DPB in PSMZ32 layout as well, which will take us back to PSMCT32 layout in the LAB. Alternatively, one could also just unswizzle the LAB on the GS at the end, once all lights have been processed (before step 3).

3.5 Two-pass DOT3 solution on the GS

Step 2.3 can be achieved in two passes. In order to do so, the *tolight* vector must be packed and passed per vertex stored as vertex colors. The packed *tolight* vector l_x, l_y, l_z is the normalized direction towards the light source T_x, T_y, T_z (transformed into tangent space), scaled and then decentralized using equation (2):

$$(l_x, l_y, l_z) = (128, 128, 128) + ((char)(s \cdot T_x), (char)(s \cdot T_y), (char)(s \cdot T_z))$$

The value s is an empirical scale factor and is given later in this section.

The surface normal n_x, n_y, n_z of length 128 in tangent space is packed by splitting it into a postive and a negative side. This is done similar to [Breugelmans02] but in addition an alpha term is computed. Two palettes are used, one for each side. An additional difference is that the *tolight* vectors are not divided into positive/negative sides but simply offset by 128.

In order to do the per-pixel shading, we need to compute the following (according to equation (10)):

$$\begin{aligned}
mul_x &= Max(n_x, 0) \cdot L_x - Max(-n_x, 0) \cdot L_x + \\
&\quad (128 + Max(-n_x, 0) - Max(n_x, 0)) \\
mul_y &= Max(n_y, 0) \cdot L_y - Max(-n_y, 0) \cdot L_y + \\
&\quad (128 + Max(-n_y, 0) - Max(n_y, 0)) \\
mul_z &= Max(n_z, 0) \cdot L_z - Max(-n_z, 0) \cdot L_z + \\
&\quad (128 + Max(-n_z, 0) - Max(n_z, 0))
\end{aligned}$$

The vector L_x, L_y, L_z is the barycentrically weighted result of the surrounding per-vertex packed *tolight* vectors. The final dot product is computed as in equation (9) (without subtraction by 128), which yields:

$$\begin{aligned} mul_x + mul_y + mul_z &= Max(n_x, 0) \cdot L_x - Max(-n_x, 0) \cdot L_x + \\ & Max(n_y, 0) \cdot L_y - Max(-n_y, 0) \cdot L_y + \\ & Max(n_z, 0) \cdot L_z - Max(-n_z, 0) \cdot L_z + \\ & (128 + Max(-n_x, 0) - Max(n_x, 0)) + \\ & (128 + Max(-n_y, 0) - Max(n_y, 0)) + \\ & (128 + Max(-n_z, 0) - Max(n_z, 0)) \end{aligned}$$

This means that we can precompute the three last terms and store the result in alpha of the normal map.

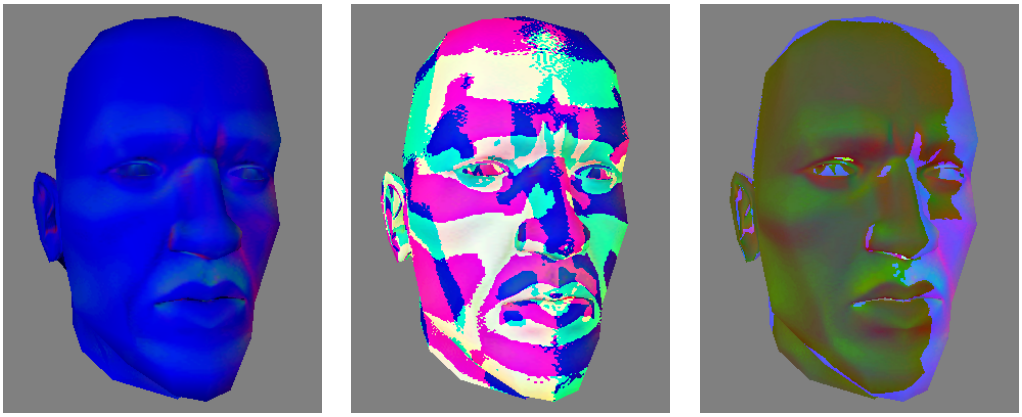
$$\begin{aligned} n_a &= (128 + Max(-n_x, 0) - Max(n_x, 0)) + \\ & (128 + Max(-n_y, 0) - Max(n_y, 0)) + \\ & (128 + Max(-n_z, 0) - Max(n_z, 0)) \end{aligned}$$

Since n_a is potentially larger than 255, visible wrapping errors are likely to occur, i.e. the bilinearly interpolated values will be wrong. To solve this we accept to lose a bit of precision and round n_a to the nearest multiple of three and divide by three $n'_a = \frac{n_a+1}{3}$. During the post filter pass (step 2.4), we can then add n'_a to red, green, and blue of the DPB.

To completely avoid the precision loss, one could also do a three-pass solution by adding the terms as a separate additive pass, but it is probably not worth it.

To summarize, the procedure is as follows:

- **Pass 1:** Render triangles using the positive palette with alpha blending disabled and the texture function set to MODULATE.
- **Pass 2:** Render the triangles again but use the negative palette and enable alpha blending to do a subtraction of the source from the frame buffer. Set vertex alpha to 128 so the texture alpha n'_a may be passed to the frame buffer alpha (set TCC=RGBA and use MODULATE).



(a) Pass 1 (b) Pass 2 (c) alpha added to rgb

Figure 2: (a) A low resolution model rendered using MODULATE and the positive palette (first pass). (b) Second pass uses the negative palette and subtracts the source from the frame buffer. Furthermore, the alpha of the normal map is passed to alpha of the frame buffer. (c) Third shot shows the result after adding the alpha. This is not a part of step 2.3, but done at step 2.4, i.e. purely 2D without using the geometry of the model. Adding $r+g+b$ yields the final dot products (see section 3.3)

In the two-pass case, we have to modify the post filter of the DPB (step 2.4) to get the final dot product result: $r + g + b + a + a + a$. This is done by using one 2D pass reading the DPB in PSMT8H and adding the contents to red, green, and blue (see figure 2(c)). The result at this point is not the signed multiplications since we have just added the last term n_a by adding an equally large slice (one third) of it to each channel. The final dot products are obtained by completing the post filter adding red, green, and blue.

Per vertex attenuation of any kind may be achieved by scaling down the vectors towards the light source T_x, T_y, T_z.

Through trial and error, good results (i.e., without wrapping errors) have been achieved using the factor $s = 122$. The code for packing the normals is shown in appendix 2. It is possible that larger factors may be used for s depending on how rounding was performed during normal map creation.

3.6 Four-pass and per pixel normalization solution

When performing a linear interpolation of the *tolight* vectors across a triangle in the two-pass solution, the result is not a unit vector since it is not renormalized. This causes lighting to decrease in intensity across the triangle and results in edge highlighting (mach-band artifacts). The solution is of course to renormalize the *tolight* vector per pixel.

Again we only consider step 2.3. One way to achieve renormalization on the GS is to do look-up into a sphere map containing packed unit vectors[2] . Since the sphere map is a texture and since reading from another texture (the normal map) is required during blending, the *tolight* vectors will have to be delivered to the DPB during the first pass. Thus, the first pass simply renders geometry with the sphere map applied, i.e., the interpolated tangent-space *tolight* vectors are used as look-ups into the sphere map, using the regular sphere map look-up equations:

$$\begin{aligned} m &= 2 \cdot \sqrt{T_x^2 + T_y^2 + (T_z + 1)^2} \\ &= 2 \cdot \sqrt{2 \cdot T_z + 2} \\ s &= \frac{T_x}{m} + 0.5 \\ t &= \frac{T_y}{m} + 0.5 \end{aligned}$$

So using this method, the *tolight* vectors are delivered via the texture coordinates and not the vertex colors.

After the first pass, the DPB contents is rasterized triangles with their normalized *tolight* vectors. To execute the signed dot product multiplications on the GS, we take advantage of equation (8). So, for the subsequent 3 passes the following GS blendmode is used:

$$(normalmap_{rgb} - framebuffer_{rgb}) \cdot normalmap_a + framebuffer_{rgb}$$

This blend mode implies that X, Y and Z of the normal map must be passed through via alpha, so 3 palettes have to be used: One for each X, Y

[2]Code for precomputing the sphere map is shown in Appendix 1.

0				1				...	255			
r	g	b	a	r	g	b	a		r	g	b	a
X_1	Y_1	Z_1	0	X_2	Y_2	Z_2	0		X_{256}	Y_{256}	Z_{256}	0

(a) Ordinary palette

0				1				...	255			
r	g	b	a	r	g	b	a		r	g	b	a
128	128	128	X_1	128	128	128	X_2		128	128	128	X_{256}

(b) Reordered Palette

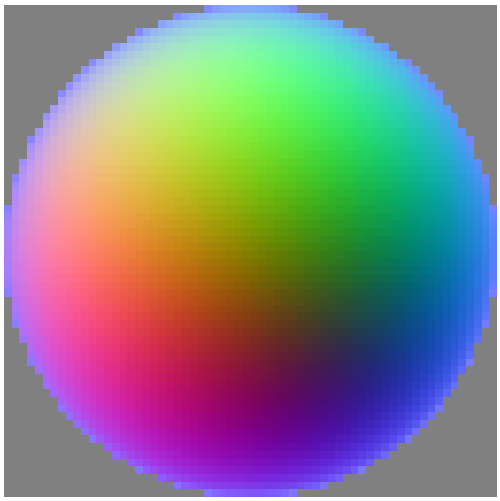
Figure 3: (a) The structure of a palette of an 8 bit normal map. (b) The reordered palette used to represent X during the signed multiplication pass. Similar ones are made for Y and Z.

and Z of the normal map. A quantized 8 bit normal map can be used with three palettes or alternatively three 8 bit normal maps, which will give the same quality as using a standard 24 bit normal map. The RGB of the 3 palettes should be set to 128 (see figure 3(b)) so *tolight* will be subtracted from 128 during blending (the first part of equation (8)).

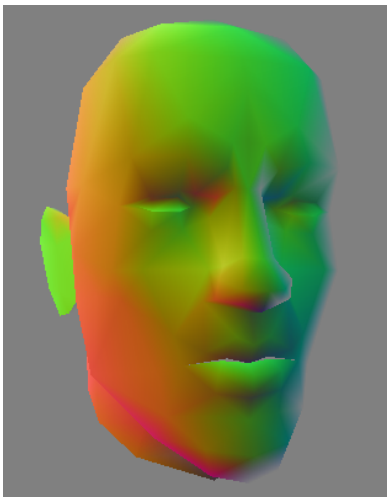
The resulting signed multiplications do not have rounding errors, and are identical with an ordinary char multiplication $a \cdot b$ with a signed shift right by 7 (offset by 128).

To summarize, the procedure is as follows:

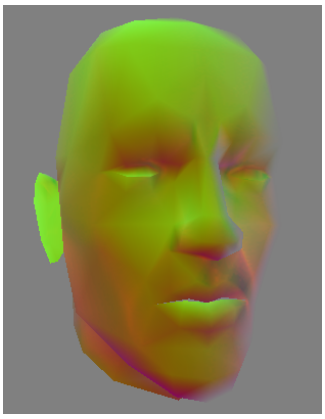
- **Pass 1:** Render triangles with the sphere map applied into the DPB.
- **Pass 2:** Render the triangles again but use palette for X, set mask to affect red only and use the blend mode above, use the normal map as a texture.
- **Pass 3:** Same as pass 2 but use palette for Y and affect green only.
- **Pass 4:** Same as pass 2 but use palette for Z and affect blue only.



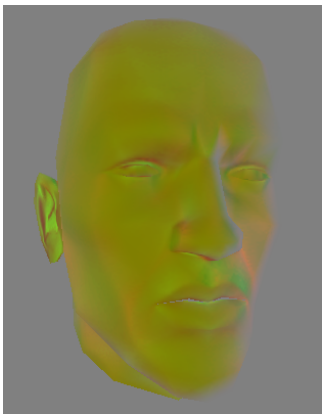
(a) Spheremap



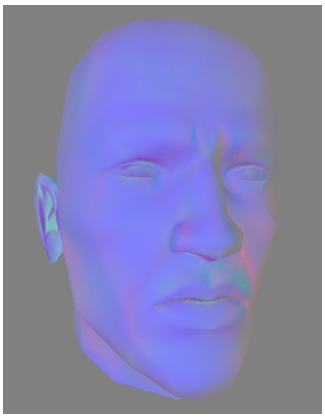
(b) Pass 1: Apply sphere map



(c) Pass 2



(d) Pass 3



(e) Pass 4

Figure 4: (a) The sphere map normalization table/texture. (b) A low resolution model rendered with the sphere map applied (first pass). (c-e) The 2nd, 3rd and 4th pass. Each pass updates a single channel in the framebuffer (red, green and blue respectively). Thus, (e) shows the final signed multiplication offset by 128. Adding these yields the final dot products (see section 3.3)

The sphere map forms *tolight* vectors which are packed according to equation (1) and hence are in the set $\{0, 1, ...255\}$ (see the code in appendix 1). As mentioned in section 3, a traditional normal map is used, centered at 128 and also in the set $\{0, 1, ...255\}$ (equation (2)).

In general, it is common to fade off the results of the normal mapping near the silhouette of the models (as seen from the light source) since the illusion fails there. On PS2, this attenuation must be applied per vertex, in the following way.

$$fade\_att = clamp(6 \cdot (n \cdot l), 0, 1)$$

The term $n \cdot l$ is the dot product between the unit length vertex normal and the unit length direction towards the light in object space. This is actually fortunate for this four-pass method since it fixes a well-known sphere mapping artifact, i.e., that projecting onto a sphere map per vertex and not per pixel goes wrong once the look-ups reach the far back of the sphere. Applying this factor will make sure triangles facing away from the light remain unlit.

The question is how the attenuation factor should be applied now that the light vectors are in a sphere map and not in the vertex colors. Applying any kind of attenuation factor can be done by scaling down the vectors in the sphere map during the first pass. The attenuation cannot be put directly into the vertex colors of this pass since the sphere map is stored as vectors subtracted from 128. A solution is to use any of the HIGHLIGHT texture functions to get the correct result, by undoing equation (1), applying the scale, and applying equation (1) again:

$$128_{rgb} - ((128_{rgb} - sphmap_{rgb}) \cdot scale) = sphmap_{rgb} \cdot scale + ((1 - scale) \cdot 128)_{rgb}$$

So by setting red, green, and blue of the vertex color to $scale \cdot 128$ and then $(1 - scale) \cdot 128$ in the vertex color alpha and by using a HIGHLIGHT texture function during first pass, per vertex attenuation is possible.

For a spot light it is possible to apply a projective texture of intensities to the DPB after the first pass and have the attenuation applied per pixel.

This is done by using the GS blend mode to apply the attenuation in the same way as when using the HIGHLIGHT texture function. Alternatively, this can also be applied after all four passes are complete.

3.7 Normal Mapped Specular Highlights

By using half-vectors instead of directions towards the light, it is possible to do normal mapped specular highlights. A half-vector is computed by adding the direction from the vertex to the eye and the direction from the vertex to the light source. The resulting half vector must still be transformed into tangent space and normalized.

In addition, the ILP should have its entries raised by an appropriate power term to match the properties of the material (only for specular, not diffuse).

To apply any form of attenuation such as distance attenuation it is necessary to compensate for the power term, otherwise the result will be $(att \cdot dot)^n$ and not $att \cdot dot^n$ as it should be. The obvious way to fix it is by using $(att^{\frac{1}{n}} \cdot dot)^n$. There are different ways to take a fast Nth root on the VU1 of values between 0 and 1, which are covered on the pro news groups and the playstation2-linux developers forum.

3.8 Using Level of Detail

As it was noted in step 5 of the overview, it is still possible to render non-normal mapped geometry in a single pass. They are just rendered into the frame buffer using standard Gouraud shading. An additional nice property is that it is possible to fade at distance continuously to the Gouraud shaded lighting. Since attenuation is possible, two lights of the same type can be put in the same position: One is rendered using Gouraud shading and the other using the fancy per pixel lighting. If one is rendering at say 70% then the other renders at 30% and so on. Once 0% intensity on the expensive version of the light is reached, it is removed from the rendering process.

Models using 4/8 bit normal maps can switch from four-pass to two-pass (e.g., based on distance) using the same normal map but with a modified palette. This is due to the fact that every index represents a unique normal so only the palette will be different when switching between the two methods.

4 Results

The methods have been implemented and tested on the PS2. The results for a low resolution model rendered using the four-pass method running in real-time on the PS2 are shown in figure 6. The model was lit by two point light sources, a green and a blue. The frame buffer resolution was 512 x 448 and the normal map (figure 5) is 256 x 256 in 8 bit. A comparison of the two methods is shown in figure 7.

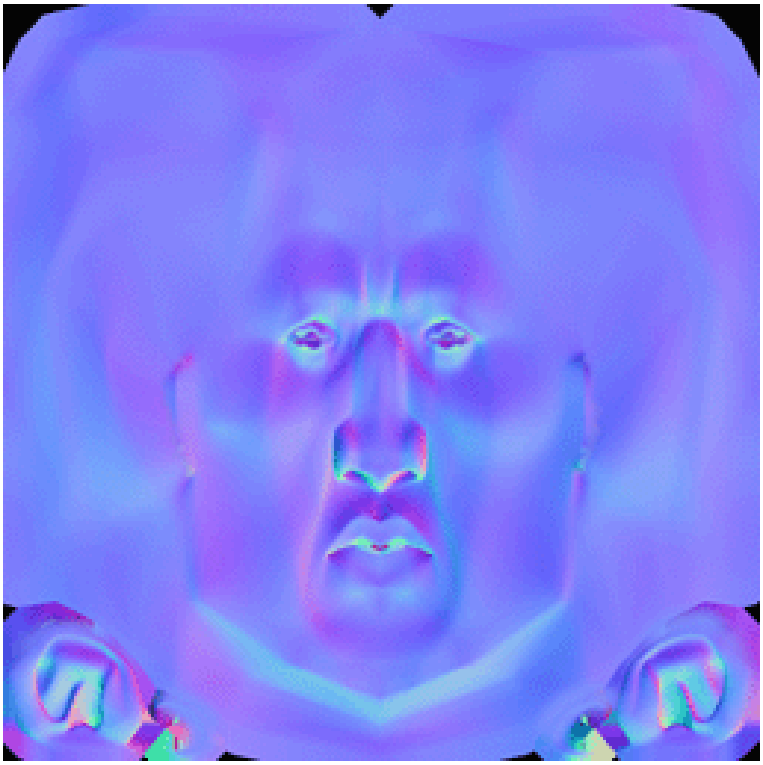
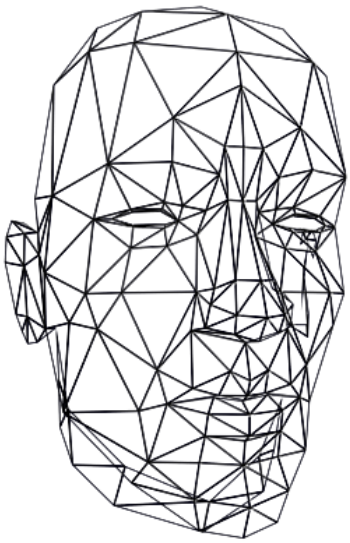


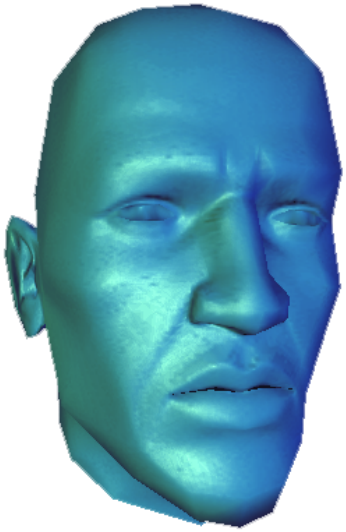
Figure 5: The normal map.



(a) Wireframe

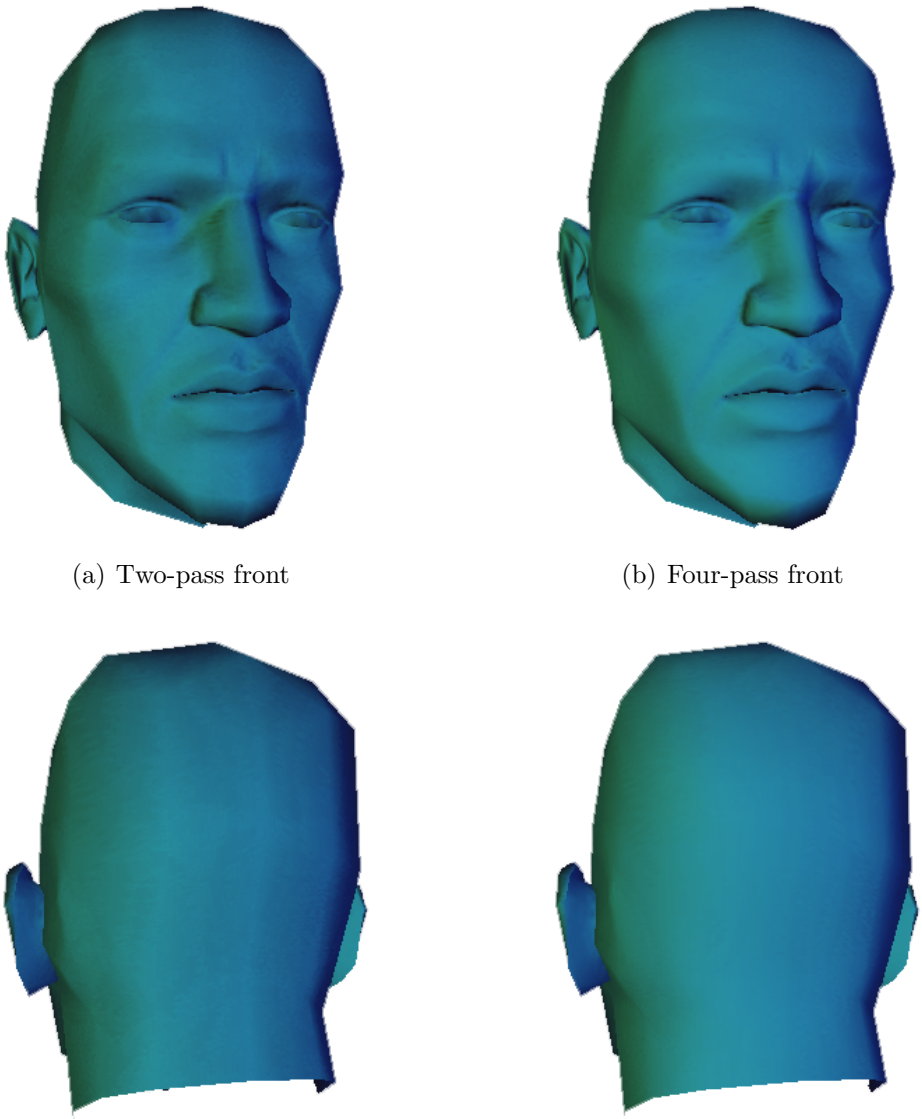
(b) Gouraud shading

(c) DOT3 diffuse



(d) DOT3 diffuse + specular

Figure 6: (a) Wireframe model (412 triangles), to give an idea of the amount of actual detail in the model. (b) Traditional Gouraud shading. (c) The model with normal mapping applied. (d) Normal mapping with specular highlights.



(a) Two-pass front

(b) Four-pass front

(c) Two-pass back

(d) Four-pass back

Figure 7: Shown to the left (a) and (c) is the front and the back of the head rendered using the two-pass DOT3 solution. To the right (b) and (d) shows the same shots using the four-pass method. (b) is the same picture as seen in 6(c). The edge highlighting is especially noticable on (c).

5 Conclusion

We have shown that it is possible to do DOT3 normal mapping very similar in quality to PC-style normal mapping. Two methods have been presented: A cheap two-pass solution, without per-pixel normalization, and a more expensive four-pass alternative with per-pixel normalization. Both take advantage of the hardware available on the PS2, more specifically the GS and the VU1. This paper has also described how it is possible to integrate the normal mapping with ordinary shading of non-normal mapped objects without any additional effect on performance.

6 Acknowledgements

The author would like to thank Kasper Høy Nielsen for his help restructuring this paper and for his many suggetions that improved its readability. Thanks also to Mircea Marghidanu, Steven Osman, Lionel Lemarie and Trine Mikkelsen for additional proof reading and their insightful comments. Finally, thanks to IO Interactive and to Eidos for letting me publish this paper.

References

[Blinn78] Blinn, J.F.: "Simulation of wrinkled surfaces", Proceedings of the 5th annual conference on Computer graphics and interactive techniques, ACM Press, pp. 286–292, 1978.

[Breugelmans01] Breugelmans, M.: "32bit Colour Channel Shifting using 8bit and 16bit texture formats", Sony Computer Entertainment Europe, `www.playstation2-linux.com/files/p2lsd/32bit_colour_channel_shifting.pdf`, October, 2001.

[Breugelmans02] Breugelmans, M.: "PS2 Bump mapping", Sony Computer Entertainment Europe, Published on PlayStation 2 professional developer's resources, January, 2002.

[Cohen98] Cohen J., Olano, M., Manocha, D.: "Appearance-Preserving Simplification", Computer Graphics, SIGGRAPH Procedings, July, 1998.

[Kilgard00] Kilgard, M. J.: "A practical and robust bump-mapping technique for today's GPU's", GDC 2000: Advanced OpenGL Game Development, 2000.

[Sony02] Playstation 2: GS User's Manual, 6th Edition, Sony Computer Entertainment Inc., 2002.

Appendix 1: Code to generate the sphere map texture

```
int nearest(const float x)
{
   if(x<0)}
      return -nearest(-x);
   else
      return (int) (x+0.5f);
}


// I use 64x64 in 24bit myself
void GenerateSphereMap(void * mem, int iWidth, int iHeight)
{
   // if lx^2+ly^2+lz^2=1
   // m = 2*sqrt(lx^2+ly^2+(lz+1)^2) = 2*sqrt(2*lz+2)
   // s = lx/m + 0.5
   // t = ly/m + 0.5

   // we setup a second degree polynomial
   // for lz by using lx^2+ly^2+lz^2-1=0
   // We then find the roots.
   // A = 1
   // B = ((2*s-1)^2 + (2*t-1)^2)*2
   // C = B - 1

   // det = B^2 - 4AC
   // lz = (-B+-sqrt(det))/2A
   // this can be reduced since det = (2-B)^2
   // lz = (-B+-(2-B))/2A
   // so we have two roots
   // lz_1 = (-B+(2-B))/2 which is 1-B (usable)
   // lz_2 = (-B-(2-B))/2 which is -1 (not usable)
   // so this means lz = 1-B
   // once we have the lz component we
   // can calculate lx and ly aswell


   // 24bit sphere map
   for(int y=0; y<iHeight; y++)
   {
      for(int x=0; x<iWidth; x++)
      {
         // reversed GS remapping (section 3.4.8)
         const float s = (x+0.5f)/((float) iWidth);
         const float t = (y+0.5f)/((float) iHeight);
         const float s2 = 2*s-1;
         const float t2 = 2*t-1;
```

```
            float Lz = 1 - (s2*s2 + t2*t2)*2;

            // zero vector
            int r = 128;
            int g = 128;
            int b = 128;

            if(Lz >= -1)
            {
               const float m = 2*sqrt(2*Lz+2);

               float Lx = (s-0.5)*m;
               float Ly = (t-0.5)*m;

               // normalize for accuracy
               const float div = sqrt(Lx*Lx+Ly*Ly+Lz*Lz);
               Lx /= div;
               Ly /= div;
               Lz /= div;

               // must remember to subtract
               // the vector from 128
               r = 128 - nearest( Lx*127 );
               g = 128 - nearest( Ly*127 );
               b = 128 - nearest( Lz*127 );

               assert(r>=0 && r<256);
               assert(g>=0 && g<256);
               assert(b>=0 && b<256);
            }

            // write
            const int vect = (int) ((b<<16)|(g<<8)|(r<<0));
            ((int *) mem)[y*iWidth+x] = vect;
         }
      }
   }
```

Appendix 2: Packing normals for two-pass

```
// packing the normals (for two-pass solution)
// IN: (nx, ny, nz) is the unit length normal in tangent space
// OUT: (R_pos, G_pos, B_pos, %), (R_neg, G_neg, B_neg, alpha)
const float scale = 127.9f;

// scale to range
int iX = (int) (nx * scale);
int iY = (int) (ny * scale);

// clamp to [-127; 128]
iX = (iX<(-127))?(-127):((iX>128)?128:iX);
iY = (iY<(-127))?(-127):((iY>128)?128:iY);

// Choose iZ so length(N) <= 128
int iZ = (int) sqrt(128*128-(iX*iX+iY*iY));
assert((iX*iX+iY*iY+iZ*iZ)<=128*128);
assert((iX*iX+iY*iY+(iZ+1)*(iZ+1))>128*128);

// positive side
const int R_pos = Max(iX, 0);
const int G_pos = Max(iY, 0);
const int B_pos = Max(iZ, 0);

// negative side
const int R_neg = Max(-iX, 0);
const int G_neg = Max(-iY, 0);
const int B_neg = Max(-iZ, 0);   // B_neg should always be zero
const int delta = (R_neg-R_pos)+(G_neg-G_pos)+(B_neg-B_pos);
int alpha = (3*128 + delta + 1) / 3;
alpha = (alpha<0)?0:((iX>255)?255:alpha);
```