IMEM: An Intelligent Memory for Bump- and Reflection-Mapping

Anders Kugler

University of Tübingen[†]
Computer Graphics Laboratory
Germany

Abstract

Data path simplification in the context of reflection- and bump-mapping hardware opens new solutions in the design of rendering and shading circuits. We are proposing a novel approach to rendering bump- and reflection-mapped surfaces, where the local geometry defining bump-maps is transformed on-the-fly prior to surface shading. Applying angular encoding to normal vectors results in narrower data paths and permits hardware integration of look-up tables of acceptable size. A special-purpose logic-embedded memory architecture is presented, where bump- and reflection-mapping of textured surfaces are executed by an intelligent memory device. High-performance surface shading is achieved by making use of precomputed shading- and reflection-map coordinate generation tables, and considering cache coherence of pixel-to-pixel normal vectors. Such a dedicated memory chip can easily be interfaced to a standard rasterizer, in place of texture memory to offer bump-, texture- and reflection-mapping hardware support.

CR Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles - Associative and Cache Memories; I.3.1 [Computer Graphics]: Hardware Architecture - Graphics Processors; I.3.3 [Computer Graphics]: Picture/Image Generation - Display Algorithms.

Additional Keywords: reflection- and bump-mapping, logic-embedded memory architectures.

1 INTRODUCTION

The appearance of computer-generated objects looks much better when surfaces are properly shaded and textured. Letting a human observer experience the surrounding environment as if he were physically present in real space is one scope of realistic rendering. When the viewer turns about a reflecting or metallic object he must get the feeling that the environment reflected by the object matches a real space in appearance.

[†] Universität Tübingen, WSI/GRIS, Auf der Morgenstelle 10-C9, D-72076 Tübingen, Germany.
email: kugler@gris.uni-tuebingen.de
http://www.gris.uni-tuebingen.de/~kugler

1998 Workshop on Graphics Hardware Lisbon Portugal

Reflection-mapping was introduced by Blinn and Newell [4] to simulate reflections from mirror-like surfaces. Bump-mapping [3] is a simple technique for simulating the roughness or grain of a wrinkled surface without having to model the surface profile geometrically. More surface detail is added by bevelling the surface with micro structures, or scribing the surface with engraving patterns. Effects like engraving text on a marble texture, mortar grooves on a brick wall or drops of water on a mirror surface, such as a metallic object reflecting light can add more realism to computer-generated objects.

One speed limiting factor in hardware supported texturing is the required bandwidth between the raster pipeline and the external memories, as explained by Knittel and Schilling [14][22]. Texture-mapping [10], environment-mapping [7] and realistic shading all require on a per-pixel basis data retrieval from external memories, containing the texture or a precomputed shading environment-map. The second limitation lies in the implementation of surface shading algorithms, often presenting a high computational complexity [1]. To overcome the performance limits of external memories, we designed a specialized memory integrating little arithmetic, performing different types of shading and texturing on the pixels generated by a raster pipeline.

The design of a special-purpose intelligent memory chip is motivated by the fact that texture-mapping, bump- and reflection-mapping or Phong shading all require parallel accesses on a per-pixel basis to external memories to fetch either texture data, the bump deflection vector, the projection in an environment of a reflected pixel, or specular and diffuse shading information.

In this paper, we are presenting a novel and elegant method for rendering bump- or reflection-mapped objects that is implemented in a special-purpose memory chip, adding shading in the sense of Phong Shading [20] to render objects part of a computer-generated scene with more realism. Such a dedicated memory chip can be easily integrated with existing rasterizer chips, in place of texture memory. The memory architecture presented in this paper assumes the existence of a standard graphics rasterization pipeline that interpolates a normal vector **N** and a texture coordinate (u, v, w) across a polygon. An object is retrieved from the object database, its associated texture- and bump-map, precomputed shading tables and environment-maps are downloaded to the memory chip, and then the object is rendered into the frame buffer.

The algorithms and hardware architecture presented in this paper make use of:

- geometry and symmetries for the bump-mapping process;
- caching: pixel to pixel cache coherence;
- shading and environment-map coordinate generation tables: the precomputed tables are indexed by the interpolated surface normal directly;
- consecutive pixels may share a common entry: making use of precomputed tables lowers the cost of computing the shading for each textured pixel.

2 BUMP- AND REFLECTION-MAPPING

2.1 Bump-Mapping

Bump-mapping is a technique that was invented by Blinn [3] to add roughness or wrinkles to a smooth surface. It does not change the underlying geometry of the model, but fools the shading to produce an interesting surface. The normal vector **N** to the surface at a point P is perturbed by a perturbation vector **B** dependent on a perturbation function F(u, v) of the surface parameters, stored as a two-dimensional table indexed by the texture coordinates u and v. Normal vector shading [2] can be applied to the deviated normals and small deviations of surface normals cause luminance variations on a smooth surface, responsible of the wrinkled surface appearance.

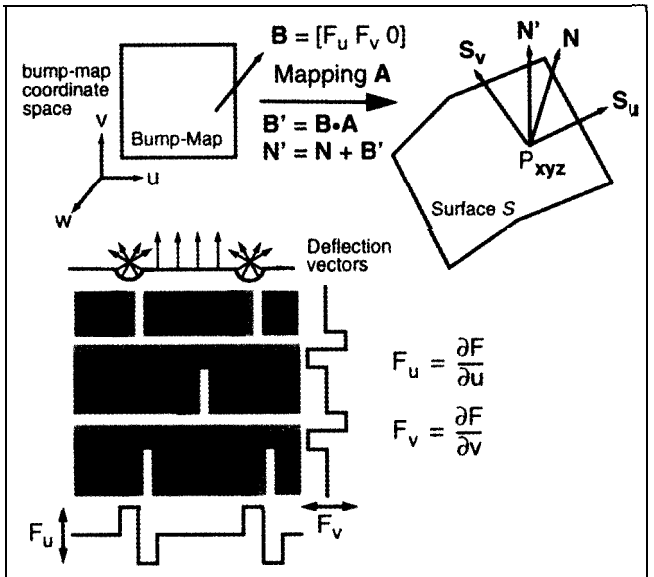

Figure 1: Bump-mapping.

The displaced surface is formed by moving a point P of the parameterized surface S(u, v) an amount F(u, v) in the direction of the surface normal **N**. Given a point P on a surface S(u, v), the normal vector **N** at that point is expressed as:

$$N_P = S_u \times S_v = \frac{\partial S}{\partial u} \times \frac{\partial S}{\partial v} \tag{1}$$

where S_u and S_v are the partial derivatives in the parameter directions u, v. The new normal to the perturbed surface is given by:

$$N' = N + \frac{F_u(N \times S_v)}{|N|} + \frac{F_v(S_u \times N)}{|N|} \qquad B' \tag{2}$$

2.2 Reflection-Mapping

Reflection-mapping calculates the reflection direction **R** of a ray from the viewer to the current sample point being shaded:

$$R = 2(N \cdot E) \cdot N - E \tag{3}$$

where **N** is the unnormalized surface normal and **E** points toward the viewer. If the reflection texture is a photograph of the environment taken from the object center, the texture-mapped surface of the object appears to be reflecting an image of its surrounding environment located infinitely far away from the object center. **R** is used to index a texture in a spherical or cubic environment-map, as described by Greene, Voorhies and Foran [7][24].

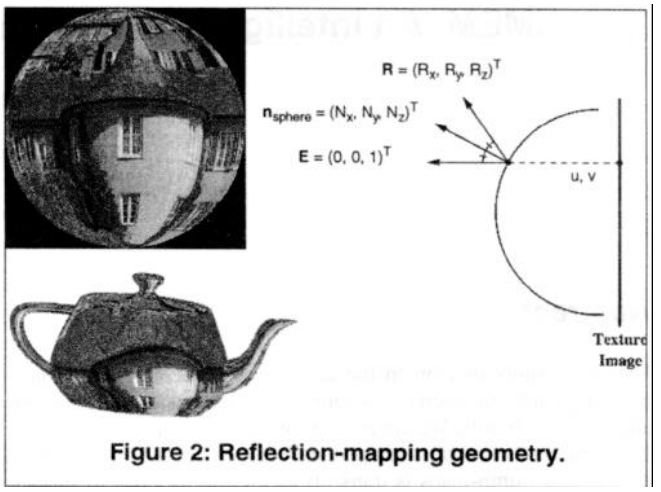

Figure 2: Reflection-mapping geometry.

The second approach to reflection-mapping is to generate an environment-map from a texture image of a perfectly reflective sphere rendered in orthographic projection, as presented by Haeberli and Segal [8] and shown in Figure 2. For any reflection vector **R** from the object surface, the corresponding normal $\mathbf{n}_{sphere}$ of the reflective sphere, where **R** hits the environment-map is calculated.

When the viewing direction is constant, i.e. we are in orthographic projection and the viewing rays going from the eye to the object surface are parallel, a spherical environment-map can be precomputed and indexed by the interpolated object surface normal $\mathbf{n}_{obj}$. We can fill up the texture image of the reflective sphere by precomputing **R** for all object normals, determine the normal $\mathbf{n}_{sphere}$, where **R** hits the reflective sphere, and index the image with the current object surface normal $\mathbf{n}_{obj}$:

$$R = 2(n_{obj} \cdot E)\, n_{obj} - E = 2(n_{obj} \cdot v_{dir}) \cdot n_{obj} - v_{dir} \tag{4}$$

$$u_N = n_{obj} \cdot v_{side} \qquad v_N = n_{obj} \cdot v_{up} \tag{5}$$

To generate coordinate (u_R, v_R) indexing into the environment-map, we use the view-up $\mathbf{v}_{up}$ and view-side $\mathbf{v}_{side}$ vectors of the current viewing direction $\mathbf{v}_{dir}$:

$$v_{dir} = \frac{v_{side} \times v_{up}}{|v_{side} \times v_{up}|} \tag{6}$$

The environment-map is addressed with a (u,v)-texture coordinate in the same way a surface texture is indexed.

$$u_R = R\, v_{side} = 2(n_{obj}\, v_{dir})\, (n_{obj}\, v_{side}) = 2(n_{obj}\, v_{dir})\, u_N \tag{7}$$

$$v_R = R \cdot v_{up} = 2(n_{obj}\, v_{dir})\, (n_{obj} \cdot v_{up}) = 2(n_{obj}\, v_{dir})\, v_N \tag{8}$$

This method has the disadvantage that a new texture image must be recomputed whenever the viewing direction changes, but requires only a single texture image. We will explain in section 5, how this method can be efficiently used to render reflective surfaces in combination with bump-mapping without having to compute the reflected ray vector **R** for each pixel.

2.3 Shading

Several existing graphics hardware accelerators evaluate the Phong lighting model at polygon vertices, before interpolating shaded color values across the polygon. However, to correctly shade a bump-mapped surface, the shading equation must be re-evaluated for each interior polygon pixel, because surface normals are likely to change between neighbouring polygon vertices.

In the Phong illumination model, reflectivity is split into a diffuse and specular component: For distant lights the light vector **L** is independent of the surface location and the shading equation becomes a function of the surface normal **N** and reflected ray vector **R**:

$$I(P) = Ambient + Diff(N) + Spec(R) \tag{9}$$

$$I(P) = k_{amb} \cdot I_{amb} + k_{diff} \cdot (L \cdot N) + k_{spec} \cdot (R \cdot E)^{n_s} \tag{10}$$

k_{amb}: ambient light reflection coefficient
k_{diff}: amount of energy reflected diffusely
k_{spec}: amount of energy reflected specularly
n_s: specular reflection exponent of surface material

The diffuse term Diff(**N**) and specular term Spec(**R**) can be thought as two separate environment-maps, containing the diffuse and specular highlights for a surface, indexed by the surface normal **N** and reflected ray vector **R**. Lighting calculations can be performed as a pre-process to incorporate highlights into a precomputed environment-map, as presented in [1][12][24]. The precomputed map Spec(**R**) can be accessed with the surface normal **N** in analogous way as a spherical environment-map.

3 RELATED AND PREVIOUS WORK

All architectures discussed so far have in common that the underlying circuitry for bump- and reflection-mapping follows a straightforward implementation of formulas and produces costly designs, in the sense of necessary computer arithmetic.

Environment-maps were first discussed by Blinn and Newell [4], further developed by Greene [7] for approximating surface illumination from reflected rays. Voorhies and Foran [24] presented a hardware architecture for reflection vector shading. Their architecture indexes an unnormalized reflection vector in an environment-map cube. It is a straightforward implementation of equation (3) for the reflected ray computation: the hardware implementation can be assembled with adders and multipliers which can be broken down in a 10-stage pipeline, if one full addition is carried out per pipeline stage.

A bump-mapping circuit doing per pixel a full matrix multiplication to transform the deflection vector from texture space to screen space was designed and is implemented by Ernst, Wittig, Rüsseler and Jackèl [1][6][12]. Their architecture requires extra square root and division units to normalize the interpolated surface normal. The straightforward method for bump-mapping presented in [6] consists of interpolating the cartesian coordinates of surface normals and applying the perturbation in a local coordinate system $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{n}\}$ tangent to the surface and defined by the normalized surface normal vector **n** and two vectors perpendicular to **n**. This method requires a matrix transformation **A** of the perturbation vector **B** before deviating the normal vector **N** for each pixel, and therefore is computationally expensive. Rotation matrix **A** must be redefined at each pixel for the current normal.

An orthonormal coordinate system, as shown by Schilling [23], can be built from the interpolated surface normal **N** and a constant main direction **m**, such as the polar axis, if a spherical texture coordinate parameterization is used:

$$e_3 = \frac{N}{\|N\|} = n \quad e_1 = \frac{m \times N}{\|m \times N\|} \quad e_2 = e_3 \times e_1 \tag{11}$$

$$A = \left[e_1\ e_2\ e_3\right]^T$$

This operation, carried out on a per-pixel basis, requires:
- 2 vector normalizations (equivalent to 2 square roots, followed by 2 reciprocals and 6 floating-point multiplies);
- 2 cross products $\mathbf{n} \times \mathbf{e}_1$ and $\mathbf{m} \times \mathbf{n}$; the second cross product is simplified if $\mathbf{m} = (0\ 1\ 0)^T$;
- 9 floating-point multiplies and 6 adds for the matrix multiplication in $\mathbf{N'} := \mathbf{N} + \mathbf{A} \bullet \mathbf{B}$;
- if vector interpolation is done in object space, the deflected normal must be converted back to world coordinates prior to shading or reflection-mapping, which necessitates a second matrix multiplication of the deflected normal vector by a constant matrix.

When matrix **A** is generated on a per-triangle basis [6] and its components are interpolated across the current triangle, the computational cost is lowered, but the bump-mapping deflection becomes inaccurate between adjacent triangles in surface regions with extreme curvature.

OpenGL's approach [15] to bump-mapping uses texture-maps to generate bump-mapping effects without requiring a custom renderer. The technique is just one possible implementation that uses more fundamental primitives in OpenGL. The lighting computation is transformed into tangent space at polygon vertices and does not require any significant new shading hardware beyond Phong shading. As shading is a function of the dot product between the perturbed surface normal vector and other vectors, such as the light or halfangle vector, all shading vectors are transformed to tangent space, where bump-mapping happens by evaluating the shading from the bump deflection vector defined in tangent space and the transformed shading vectors.

When set to environment-mapping mode [18], OpenGL is using a method similar to the technique presented by Haeberli and Segal to generate its texture coordinate [8]. At each polygon vertex the reflected ray vector is evaluated analytically from equation (3), before the (u, v) coordinate is computed and interpolated across the polygon.

Recently, Peercy et al. [19] presented a minimal architecture for bump-mapping hardware support, where the perturbed normal is precomputed for a surface tangent space. Shading and perturbed normal vectors must still go through normalization pipelines for the shading to work properly. This architecture was specially trimmed to switch easily between bump-mapping and standard shading. The deflection of the normal vector happens in tangent space at polygon vertices, where the illumination is directly computed. Recovering the distorted normal vector to further use it for reflection-mapping is no longer possible.

Ikedo and Ma [11] present a graphics chip with bump-mapping and Phong shading support. The algorithms implemented in their circuit follow the known straightforward implementation with a series of matrix multiplications. Their bump-mapping support is dubious in the sense that it interpolates angles between light and normal vectors, and may produce satisfactory results only when rendering very small sized polygons.

4 THE ALGORITHM

The object surface normal vector **N** is interpolated in cartesian object coordinates. Interpolating angles in a spherical coordinate system would be wrong: it does not follow a great arc between two normals, but involves great swings. For the purpose of our bump-mapping algorithm, **N** is transformed to spherical coordinates (φ_N, θ_N) after interpolation.

4.1 Normal Vector Representation

Any normal vector $\mathbf{N}=(N_x, N_y, N_z)^T$ is expressed with a horizontal angle φ_N and a vertical angle θ_N in spherical coordinates. Discretizing the unit sphere into 512×256 patches offers a resolution of 0.012 rad or 0.7 degrees between two vectors. Points on the unit sphere are parameterized by two angles Φ and Θ. Rectangular coordinates are mapped to spherical coordinates by following relations:

$$N_x = \cos\Theta \cdot \sin\Phi \quad N_y = \sin\Theta \quad N_z = \cos\Phi \cdot \cos\Theta \qquad (12)$$

$$\Theta = Arc\sin(N_y) \qquad \Phi = Arctg\left(\frac{N_x}{N_z}\right) \qquad (13)$$

Normals are encoded the following way: one 9-bit field and one 8-bit field specify indices (φ, θ) to a horizontal angle Φ and a vertical angle Θ. The first bit in θ denotes the sign of Θ.

$$\theta = \pm\frac{128 \cdot \Theta}{\frac{\pi}{2}} \qquad \varphi = \frac{512 \cdot \Phi}{2\pi} \qquad (14)$$

The unnormalized surface normal vector $\mathbf{N}=(N_x, N_y, N_z)^T$ could be used to index directly into a cubic environment-map and retrieve the corresponding spherical coordinate (φ_N, θ_N). To cover the whole range of surface normals, discretized over the unit sphere, such a map would have a size of $512 \times 256 \times 17$-bit, which is far too big for a simple look-up table. Normal vector compression, as presented by Deering [5], exploits geometrical symmetries. Since the unit sphere is symmetrical in eight pieces (octants) by sign bits of the vector components, the look-up table size can be reduced to 16384×14-bit, which still is very large.

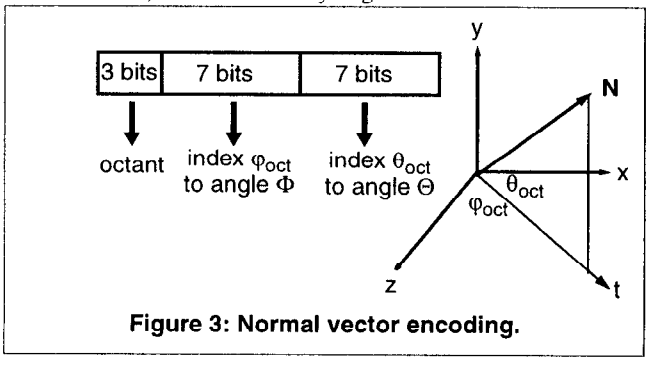

Figure 3: Normal vector encoding.

Any normal vector $\mathbf{N}=(N_x, N_y, N_z)^T$ is assigned to an octant by changing the signs of its components. Indices in a look-up table containing φ_{oct} and θ_{oct} are computed as follows:

$$t_u = \frac{N_x^2}{N_z^2} \qquad \varphi_{oct} = LUT_\theta(t_u) \qquad (15)$$

$$t_v = \frac{N_y^2}{N_x^2 + N_z^2} \qquad \theta_{oct} = LUT_\theta(t_v) \qquad (16)$$

Before evaluating the two divisions (15)(16), the three vector components N_x, N_y and N_z are sorted, in order to divide by the largest component and clamp the results t_u and t_v to the range [0..1]. Look-up table $LUT_\theta(t)$ does now have an acceptable size of 128×7-bit.

Although this transformation necessitates three squares (multiplications), followed by two separate divisions, it is after thorough evaluation an economic and accurate solution, in terms of size and number of look-up tables. To avoid the latency penalty incured by one extra cycle through two consecutive accesses to the same look-up table, we can duplicate $LUT_\theta(t)$ and implement an additional divider. This does make sense in order to reach full performance.

The normal vector is now defined by a tuple (oct, φ_{oct}, θ_{oct}). This tuple will be mapped and expanded at a later stage to (φ_N, θ_N), a spherical coordinate with a vertical polar axis, by the hardware architecture before the bump-mapping and shading happen.

4.2 Deflection Vector Representation

Bump-maps can either be represented as offset vectors **B**, that are added to surface normal vectors to displace them, or as displaced normal vectors **D** which are substituted to the initial surface normals. When choosing the first representation, only two components for the cartesian coordinates of the bump vector **B** are needed. In the second representation, three components of a cartesian coordinate system or two angles in a spherical coordinate system are necessary. Our bump-mapping is based on angular deflection, therefore we will use the second representation.

The bump-map is indexed with a (u, v)-coordinate in the same way as a texture-map. Each bump-map entry contains a tuple (φ_D, θ_D) with a horizontal and vertical angle *relative to the horizontal polar axis* u for the deflection vector **D**, expressed in (u, v, w)-coordinate space.

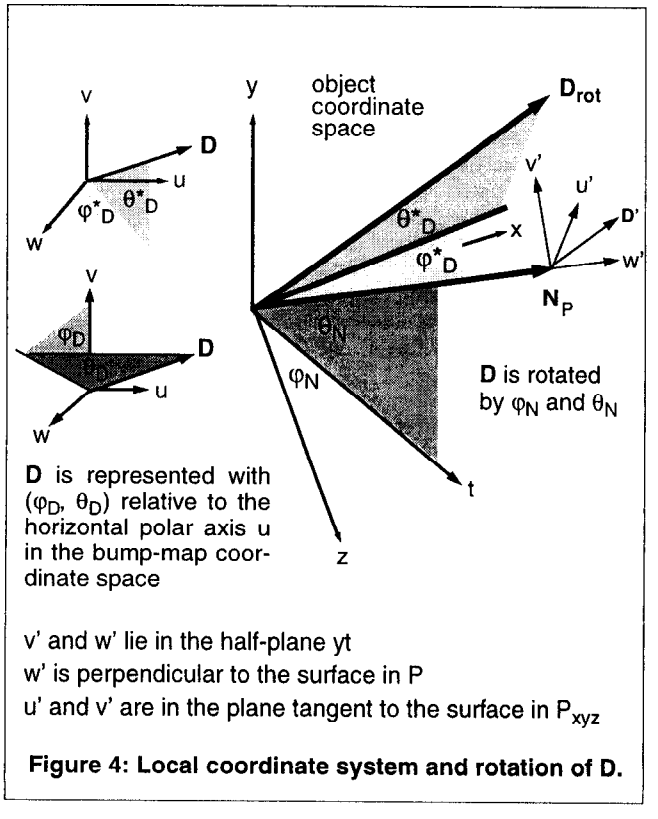

Figure 4: Local coordinate system and rotation of D.

4.3 Local Object Coordinate System

In object coordinates, the origin and coordinate axes remain fixed relative to an object no matter how the object's position or orientation changes in space. Surface normals are defined in their local object coordinate system, where the object's vertices are also defined, before the object is transformed by rotation or translation in world space. All computations relative to bump- or environment-mapping are carried out in the object coordinate system.

Blinn's original bump-mapping method [3] requires the amount of displacement **B** to be scaled at the same rate as the surface normal **N**. Our method displaces the surface by rotating **N** by an amount described with two angles. Since angles are invariant to object scaling, the bump-mapped surface appearance is preserved at different object sizes and scaling of **D** is unnecessary.

4.4 Vector Rotation

The deflection **D** can be viewed as rotating the u and v bump-map coordinate axes about the object normal vector **N**, as shown in Figure 4. By defining the surface normal vector **N** with a signed horizontal angle φ_N and vertical angle θ_N, we can rotate any deflection vector **D** from bump-map space to object space. The rotated bump-map coordinate axes w' and v' are constrained to lie in the plane containing the surface normal and y axis of the object coordinate system. u' is perpendicular to **N**, w' and y. Instead of adding a perturbation vector **B** to **N**, we replace the normal vector **N** by a rotated deflection vector $\mathbf{D}_{rot}$.

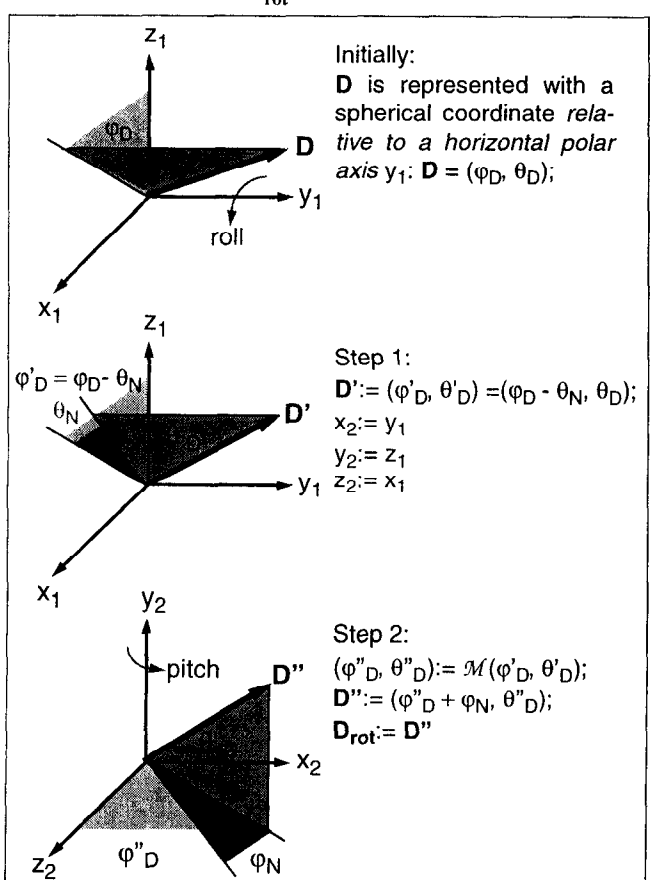

Figure 5: Rotating D from bump-map domain to the object surface by querying a spherical-to-spherical map.

Any rotation can be broken down into a series of three rotations (roll, pitch, yaw) about the major axes of the object coordinate system [9]. To rotate the bump normal **D**, we represent **D** with a spherical coordinate (φ_D, θ_D) *relative to the horizontal polar axis* and use a spherical-to-spherical map $\mathcal{M} : \{(\varphi, \theta) \Rightarrow (\varphi', \theta') :: (z \Rightarrow y', x \Rightarrow z', y \Rightarrow x')\}$ transforming (φ, θ) into (φ', θ'). The rotation of **D** is performed by one rotation (roll) about the object x-axis, followed by a look-up to the spherical-to-spherical map $\mathcal{M}$ and one rotation (pitch) about the object y-axis.

Adding an offset $-\theta_N$ to the horizontal angle φ_D is equivalent to turning the bump-map about the object x-axis: $(\varphi'_D, \theta'_D) = (\varphi_D - \theta_N, \theta_N)$. Then the spherical-to-spherical map is queried once yielding $(\varphi''_D, \theta''_D)$. The second offset φ_N is added to $(\varphi''_D, \theta''_D)$ yielding $(\varphi''_D + \varphi_N, \theta''_D)$, which corresponds to turn the bump-map (u, v, w) coordinate system about the object y-axis. The sequence in Figure 5 formulates the rotation of **D** from texture domain to the local coordinate system, aligned with the object surface normal **N**.

5 REFLECTION-MAPPING

The first step to reflection-mapping is to precompute an environment-map that can be directly indexed with the surface normal interpolated in object space.

In an earlier work [13], we explained how to precompute efficiently environment- and shading-map coordinate generation tables $F_1, F_2, F_3, G_2, G_3, H_2, H_3$, that can be indexed by the interpolated surface normal directly. We apply this method here to precompute an environment-map for the current viewing direction, relative to the object. **R** is precomputed for every object normal $\mathbf{n}_{obj}$ and becomes a function of (φ_N, θ_N).

Below we briefly detail how to derive $F_1, F_2, F_3, G_2, G_3, H_2, H_3$ from $\mathbf{n}_{obj}$, $\mathbf{v}_{up}$, $\mathbf{v}_{side}$ and $\mathbf{v}_{dir}$. The viewing direction $\mathbf{v}_{dir}$ is defined by the vectors $\mathbf{v}_{side}$ and $\mathbf{v}_{up}$: $\mathbf{v}_{dir} := \mathbf{v}_{side} \times \mathbf{v}_{up}$. Since **N** is interpolated in object space, $\mathbf{v}_{side}$, $\mathbf{v}_{up}$ and $\mathbf{v}_{dir}$ must be transformed from world coordinates to object coordinates. If we transform $\mathbf{v}_{side}$, $\mathbf{v}_{up}$ and $\mathbf{v}_{dir}$ to object space, we can use the object surface normal $\mathbf{n}_{obj}$ directly, rather than having to transform **R** to world coordinates, to access the environment-map. The local surface normal $\mathbf{n}_{obj}$ is defined with a spherical coordinate (φ_N, θ_N), and we apply the method outlined in section 2.2 and develop the dot products:

$$u_N = \mathbf{n}_{obj} \cdot \mathbf{v}_{side} = \cos(\theta_N) \cdot (\sin(\varphi_N) \cdot [v_{side}]_x + \cos(\varphi_N) \cdot [v_{side}]_z) + \sin(\theta_N) \cdot [v_{side}]_y \quad (17)$$

$$v_N = \mathbf{n}_{obj} \cdot \mathbf{v}_{up} = \cos(\theta_N) \cdot (\sin(\varphi_N) \cdot [v_{up}]_x + \cos(\varphi_N) \cdot [v_{up}]_z) + \sin(\theta_N) \cdot [v_{up}]_y \quad (18)$$

and factorize expressions (17)(18) into:

$$u_N = F_1(\theta_N) \cdot F_2(\varphi_N) + F_3(\theta_N)$$
$$v_N = G_1(\theta_N) \cdot G_2(\varphi_N) + G_3(\theta_N) \quad (19)$$

with

$$F_1(\theta_N) = G_1(\theta_N) = \cos(\theta_N) \quad (20)$$
$$F_2(\varphi_N) = \sin(\varphi_N) \cdot [v_{side}]_x + \cos(\varphi_N) \cdot [v_{side}]_z$$
$$G_2(\varphi_N) = \sin(\varphi_N) \cdot [v_{up}]_x + \cos(\varphi_N) \cdot [v_{up}]_z$$
$$F_3(\theta_N) = \sin(\theta_N) \cdot [v_{side}]_y \quad G_3(\theta_N) = \sin(\theta_N) \cdot [v_{up}]_y$$

Coordinate (u_N, v_N) is used to read the diffuse term from a spherical shading map as explained in section 2.3.

Similarly, we compute u_R and v_R from u_N, v_N and relations (7)(8):

$$u_R = \boldsymbol{R}_{obj} \cdot v_{side} = 2(H_1(\theta_N) \cdot H_2(\varphi_N) + H_3(\theta_N)) \cdot u_N \quad (21)$$

$$v_R = \boldsymbol{R}_{obj} \cdot v_{up} = 2(H_1(\theta_N) \cdot H_2(\varphi_N) + H_3(\theta_N)) \cdot v_N$$

with

$$H_1(\theta_N) = F_1(\theta_N) = \cos(\theta_N) \quad (22)$$

$$H_2(\varphi_N) = \sin(\varphi_N) \cdot [v_{dir}]_x + \cos(\varphi_N) \cdot [v_{dir}]_z$$

$$H_3(\theta_N) = \sin(\theta_N) \cdot [v_{dir}]_y$$

The spherical environment- or shading-maps are addressed with texture coordinate (u_R, v_R). To generate the texture address we first index with (φ_N, θ_N) into seven one-dimensional tables, containing the precomputed terms F_1, F_2, F_3, G_2, G_3, H_2, H_3. Each table covers the range from 0 to $\pi/2$, and therefore has 128 entries. Values outside this range are obtained by trigonometric symmetry. The resulting (u, v) coordinate can be transferred to a texture memory address generation unit where it is processed as any standard texture coordinate to read and interpolate the projected pixels from a shading- or environment-map texture.

6 HARDWARE ARCHITECTURE

6.1 Operational Framework

The intelligent memory (IMEM) receives a (u, v, w) texture coordinate, a normal vector **N** interpolated in object cartesian coordinates and a RGB color value for the current pixel generated by the rasterizer. An internal texture-mapping pipeline does the perspective division and computes a physical memory address from the texture coordinate. This address is used to retrieve texture data or a bump-mapping deflection vector $\mathbf{D} = (\varphi_D, \theta_D)$ from internal memory.

Before deflecting the current surface normal **N**, its components (N_x, N_y, N_z) go through a comparison logic generating the octant registration. Conversion to angular coordinates happens by squaring N_x, N_y and N_z, making two divisions and looking up the angles φ_{oct} and θ_{oct}. The divider pipelines one Newton-Raphson iteration [16][25], starting with an approximation taken from a look-up table. Octant information together with $\mathbf{N} = (\varphi_{oct}, \theta_{oct})$ are piped to the cache RAM, where **N** is expanded to (φ_N, θ_N) and deflected.

For reflection-mapping and surface shading the spherical coordinate (φ_N, θ_N) indexes into seven 128 × 10-bit tables to look-up the terms to compute a texture address. The address is used to read a texel from the environment-map memory or specular and diffuse shading coefficients from two shading-map memories. The perturbation of **N** by deflection vector **D** happens in the cache RAM. Finally, textured and environment-mapped pixels are interpolated and blended with the shading coefficients.

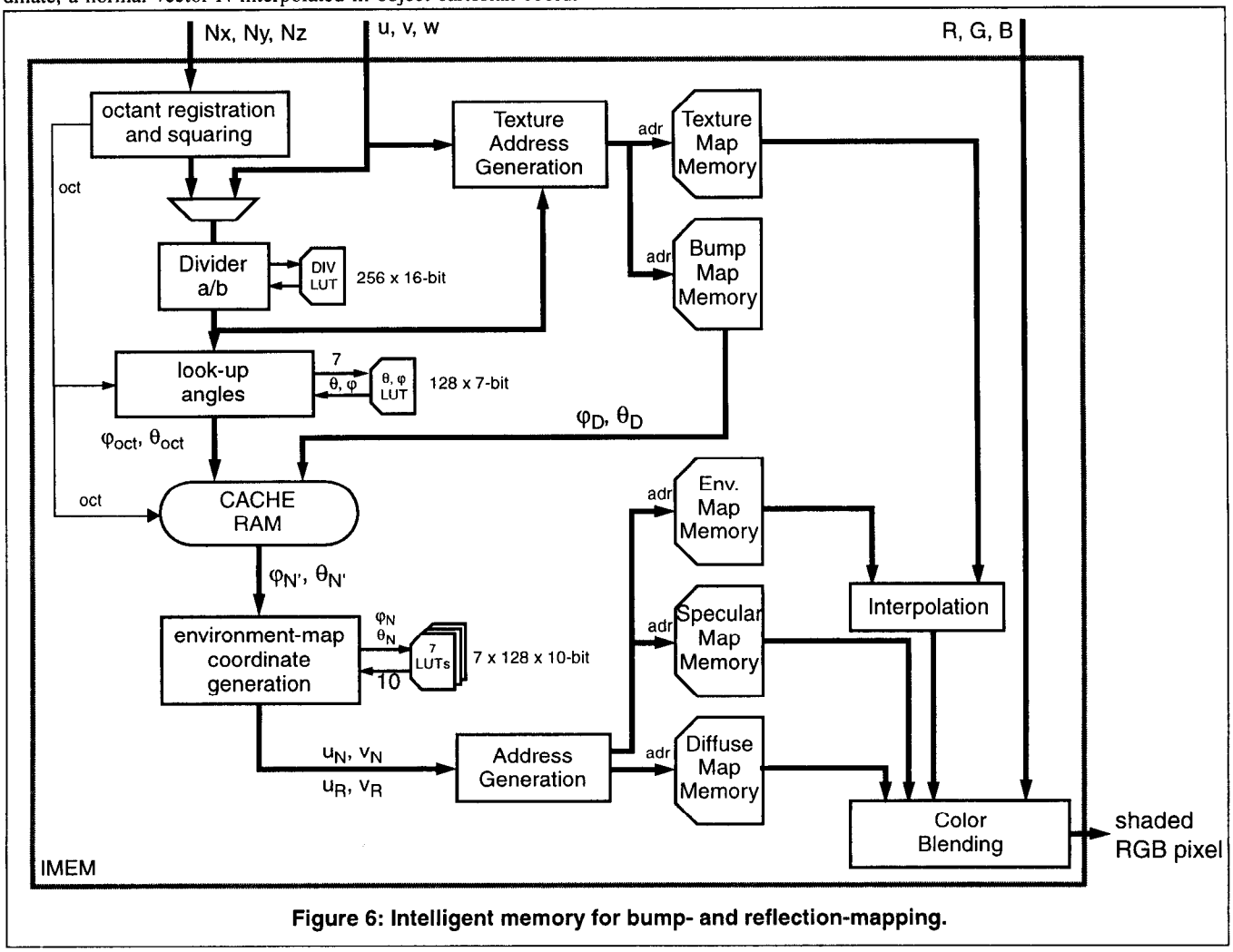

Figure 6: Intelligent memory for bump- and reflection-mapping.

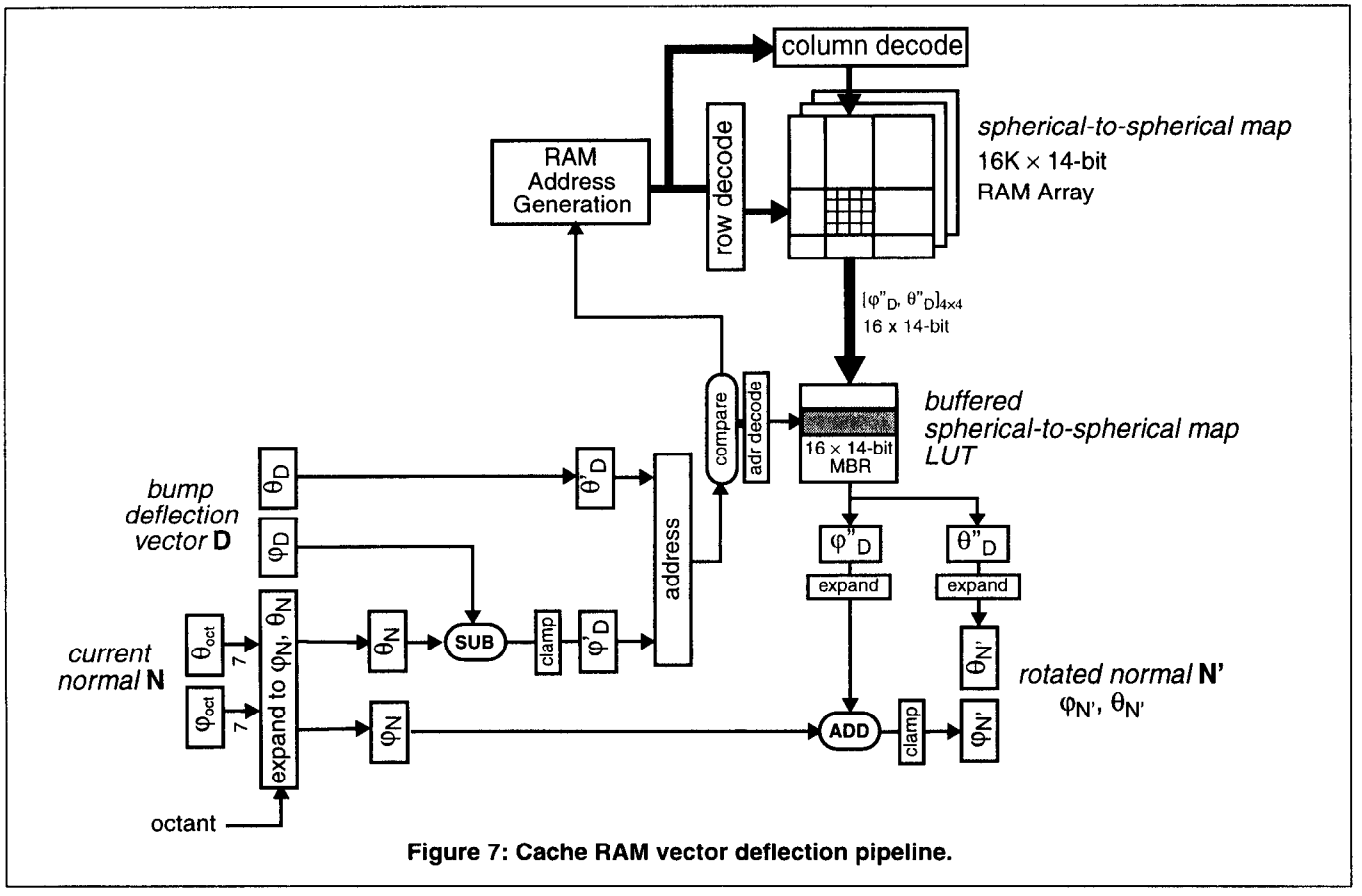

Figure 7: Cache RAM vector deflection pipeline.

6.2 Cache RAM

In section 4.4 we explained how a vector expressed with spherical coordinates can be rotated by adding an angular offset to the coordinate and making one look-up to a spherical-to-spherical map $\mathcal{M}$. The rotation of **D** involves one look-up to the spherical-to-spherical map.

The cache RAM is detailed in Figure 7: it implements the spherical-to-spherical map and is organized in interleaved 4 × 4 blocks. Any 4 × 4 block contains 16 entries of the spherical-to-spherical LUT. Any vector $\mathbf{v}(\varphi, \theta)$ in the first octant ($x, y, z \geq 0$) of the unit sphere is represented *only once* by a spherical coordinate in the cache RAM. Data (4 × 4 blocks) from the spherical-to-spherical map are transferred to a memory buffer register (MBR) which works as a temporary look-up table to rotate **D**. The access scenario to the cache RAM works as follows:

1. To rotate **D**, the angular offset $-\theta_N$ is added to φ_D, yielding $(\varphi'_D, \theta'_D) := (\varphi_D - \theta_N, \theta_D)$, which is translated into a look-up table index. The high-order address bits are compared with the current block address of the data in the memory buffer register to determine whether new data must be transferred or not from memory to the MBR.

2. One look-up to the spherical-to-spherical map produces $(\varphi''_D, \theta''_D)$. The second offset φ_N is added to φ''_D: $(\varphi''_D + \varphi_N, \theta''_D)$.

3. The cache RAM returns the rotated normal $\mathbf{N'} = (\varphi_{N'}, \theta_{N'}) := (\varphi''_D + \varphi_N, \theta''_D)$ to the environment-map coordinate generator for shading and reflection-mapping.

Normal vectors are likely to vary smoothly along one rendered scanline, so buffering 4 × 4 blocks of data permits consecutive queries to access directly the MBR to rotate **D**.

7 DISCUSSION AND EVALUATION

7.1 Results

The algorithms and hardware architecture presented in this paper were simulated in C to validate their feasibility and are implemented in VHDL. Figure 9 shows objects that were bump- and reflection-mapped with the technique presented in this paper.

The amount of DRAM storage for texture-, environment-, bump- and shading maps usually ranges from one to a few MBytes. It can be set deliberately, depending on the number of texture maps. IMEM will have far less external memory accesses, which greatly improves the performance in terms of bump- and reflection-shaded pixels, but still has a given density, limiting the number and size of texture-maps.

In Table 1, we list the different storage components of our architecture.

Memory	type	size
Texture-Map	DRAM	1 MByte
Environment-Map	DRAM	1 MByte
Bump-Map	DRAM	1 MByte
Specular Shading Map	DRAM	1 MByte
Diffuse Shading Map	DRAM	1 MByte
Division approximation LUT	ROM LUT	256 × 16-bit
(φ, θ)-angle LUT	ROM LUT	128 × 7-bit
(u, v)-coordinate LUT	SRAM LUT	7 × 128 × 10-bit
Cache RAM sph-to-sph Map	RAM	16K × 14-bit
MBR LUT	register array	16 × 14-bit

Table 1: Look-up tables and memories inside IMEM.

To summarize our approach, we

- transfer bump-, reflection-mapping and shading to the object coordinate system;
- apply view-dependent shading: precomputed shading- and environment-map coordinate generation tables are indexed directly by the surface normal;
- trade complicated vector arithmetic and normalization operations against a cache RAM, a few look-up tables and two adders for the deflection of normal vector **N**;
- exploit pixel-to-pixel cache coherence and pipeline the deflection of consecutive pixel normal vectors;
- combine processing and data for texture-, environment-, bump- and shading-maps in an intelligent memory.

Efficiency is gained by moving shading and reflection-mapping to object space, where the deflected surface normal can be directly used, instead of having to transform it back to world coordinates. It is also important to note that the method of transferring all computations to object space is invariant to object rotation and position in space. Our bump-mapping algorithm requires very little arithmetic: a few adders and multipliers for environment- and shading-

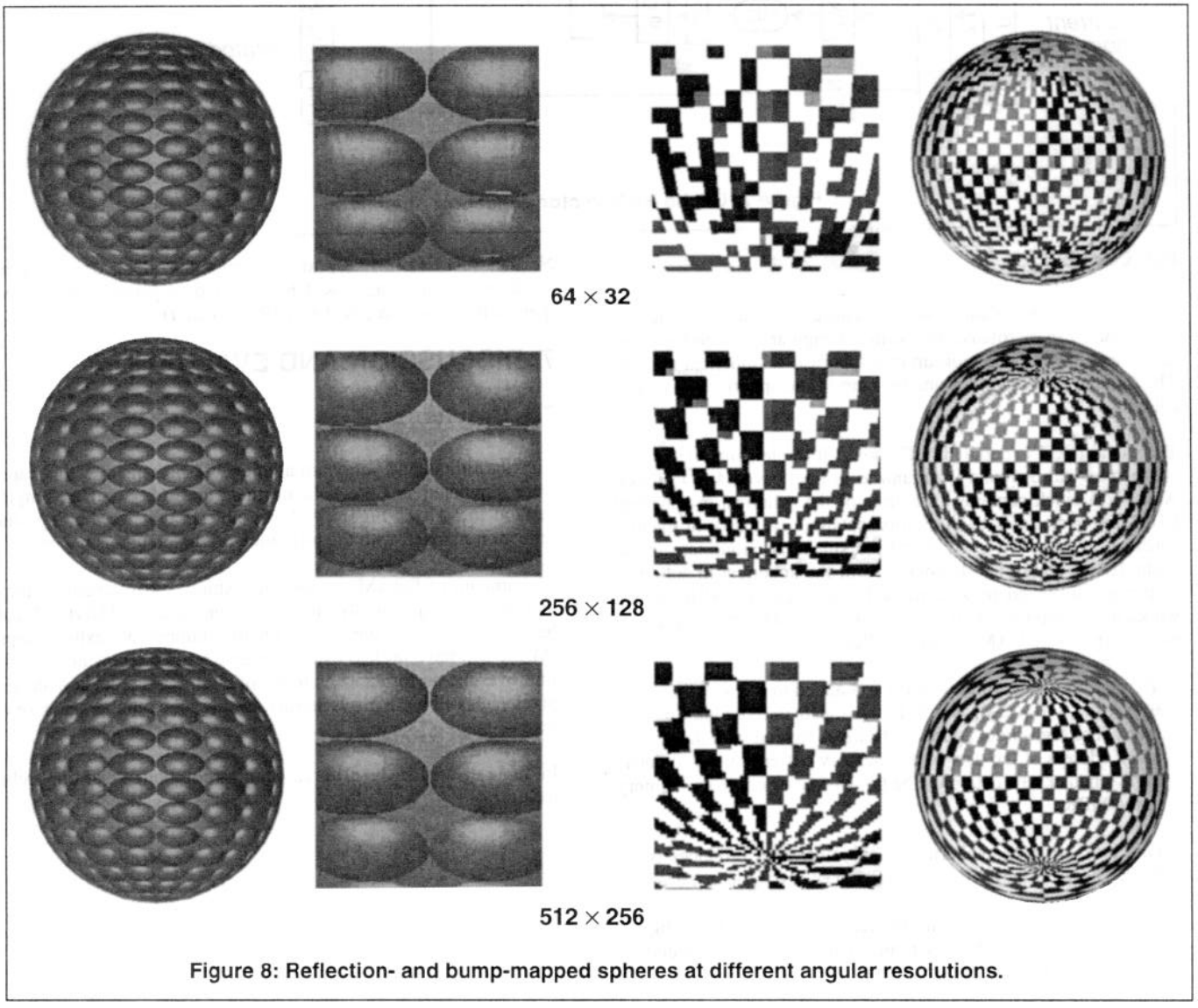

Figure 8: Reflection- and bump-mapped spheres at different angular resolutions.

map coordinate generation; the divider, already available for perspective division, is shared by the normal vector transformation to polar coordinates. The reflection-map and shading-maps can be precomputed in advance for a given environment.

Only the (u, v)-coordinate look-up tables must be updated when the object orientation changes, because these tables are a function of the current view-up and view-side vectors, expressed in object coordinates. These tables are of small size ($7 \times 128 \times 10$-bit) and can be reinitialized before each frame. Reloading these view-dependant look-up tables whenever the object or its orientation changes is a fundamental problem due to performing computations in object space with precomputed tables. The chosen tables permit the generation of coordinates for an environment-map texture having a size of 1024×1024 pixels. To support both texture filtering and a higher texture resolution the look-up table width could be extended to 16 bits (12 integer bits + 4 bits for subpixel resolution and filtering).

7.2 Angle Resolution

We have described our method and hardware architecture for an angular resolution of 512×256 normal vector samples. The unit sphere is sampled 512 times along parallels over the range $[0..2\pi]$ for φ and 256 times along meridians $[-\pi/2..+\pi/2]$ for θ. This resolution was chosen for practical simulation reasons, where memory allocation grows quadratically for any look-up table. The color images in Figure 9 were all generated at this resolution. Such a resolution is sufficient for standard bump-mapping shading and is acceptable for reflection-mapping. Artifacts are barely noticeable on bump-mapped objects, but become obvious on reflection-mapped objects, when resolution is further decreased. Figure 8 compares image rendering quality at different angular resolutions for bump- and reflection-shading.

7.3 Technology Considerations

A standard graphics subsystem consists of one or more processing units retrieving data from external memories over common or separate buses of fixed width, which limits the communication bandwidth between the processing units and memories. Such architectures suffer from an important contradiction: performance requirements for a high-speed memory and high-throughput bus conflict with requirements for low power, small circuit pin-count and cost.

Given the growing processor-memory gap, we find it worthwhile to consider unifying processing logic and DRAM on a single chip. We named such a chip IMEM for intelligent memory, because most transistors on this merged chip will be devoted to memory. DRAM is much denser than SRAM, the traditional choice for on-chip caches, which justifies to merge the processing unit in DRAM rather than increasing on-processor cache SRAM.

The main advantage of integrating the processing unit and memory is the feasibility of using wide and fast internal buses. The feasibility of such a design decision has already been demonstrated by hybrid memory-processor architectures such as the M32R/D, a 32-bit RISC processor [17], or the MSM7680, a multimedia accelerator with 1.25 MByte embedded DRAM [21]. Technology advances will foster this trend and certainly will enable on-chip fusion of a small microcontroller with high-density DRAM in the range of a few MBytes, to attain a potential internal bus bandwidth of several Gigabytes per second.

8 CONCLUSION

Bump- and reflection-mapping demand parallel memory accesses and intensive arithmetic for the computation of the deflected normal vector, the reflected ray vector and the shading. Our approach to bump- and reflection-mapping requires very simple arithmetic and exploits the local geometry of the bump-mapping process, cache coherence of pixel-to-pixel normal vectors, precomputed shading and reflection-map coordinate generation tables, accessed in parallel for each textured pixel, which enabled its implementation in an intelligent memory device.

IMEM is an intelligent memory device, integrating little arithmetic and offering bump-, texture- and reflection-mapping hardware support to an existing surface rendering pipeline. It can be easily interfaced to a standard rasterizer in place of texture memory. The design of IMEM is motivated by the fact that texture-, bump- and reflection-mapping, Phong shading all involve parallel accesses on a per-pixel basis to external memories to fetch the necessary data.

9 ACKNOWLEDGEMENTS

The above work is funded by the Commission of the European Communities (CEC). I would like to thank Dr. Andreas Schilling for approving the correctness and validity of the presented method and for discussing with me several aspects of bump-mapping. Thanks to Dorothea Welte for producing the photo-quality hardcopy for the color plate.

10 REFERENCES

[1] K. Bennebroek, I. Ernst, H. Rüsseler, O. Wittig, "Design Principles of Hardware-based Phong Shading and Bump Mapping", *Proceedings of the 11th Eurographics Workshop on Graphics Hardware*, pages 3-9, 1996.

[2] G. Bishop, D. M. Weimar, "Fast Phong Shading", *Computer Graphics* **20**(4), pages 103-106, 1986.

[3] J. F. Blinn, "Simulation of wrinkled surfaces", *Computer Graphics* **12**(3), pages 286-292, 1978.

[4] J. F. Blinn, M. E. Newell, "Texture and Reflection in Computer Generated Images", *Computer Graphics* **10**(3), 1976.

[5] M. Deering, "Geometry Compression", *Computer Graphics* **29**(4), pages 13-20, 1995.

[6] I. Ernst, D. Jackèl, H. Rüsseler, O. Wittig, "Hardware Supported Bump Mapping: A Step towards Higher Quality Real-Time Rendering", *Proceedings of the 10th Eurographics Workshop on Graphics Hardware*, pages 63-70, 1995.

[7] N. Greene, "Environment Mapping and Other Applications of World Projections", *IEEE Computer Graphics and Applications* **6**(11), pages 21-29, 1986.

[8] P. Haeberli, M. Segal, "Texture Mapping as a Fundamental Drawing Primitive", *Proceedings of the 4th Eurographics Workshop on Rendering*, 1993.

[9] D. Hearn, M. P. Baker, "Computer Graphics", pages 409-420, Prentice-Hall, 1994.

[10] P. S. Heckbert, "Survey of Texture Mapping", *IEEE Computer Graphics and Applications* **6**(11), pages 56-67, 1986.

[11] T. Ikedo, J. Ma, "An Advanced Graphics Chip with Bump-mapped Phong Shading", *Proceedings of the IEEE Computer Graphics International Conference*, pages 156-165, 1997.

[12] D. Jackèl, H. Rüsseler, "A Real-Time Rendering System with Normal Vector Shading", *Proceedings of the 9th Eurographics Workshop on Graphics Hardware*, pages 48-57, 1994.

[13] A. Kugler, "Interactive Bump- and Reflection-Mapping Hardware", *University of Tübingen Computer Science Technical Report*, WSI 97-16, ISSN 0946-3852, 1997.

[14] G. Knittel, A. Schilling, W. Straßer, "GRAMMY: High Performance Graphics Using Graphics Memories", *High Performance Computing for Computer Graphics and Visualization*, Springer, Berlin, 1995.

[15] T. McReynolds, "Programming with OpenGL: Advanced Rendering", SIGGRAPH `97 Course, section 8.3, 1997.

[16] S. Oberman, M. J. Flynn, "Design Issues in Floating-Point Division", *Stanford Technical Report*, CSL-TR-94-647, 1994.

[17] Y. Nunomura, T. Shimizu, O. Tomisawa, "M32R/D-Integrating DRAM and Microprocessor", *IEEE Micro* **17**(6), pages 40-47, 1997.

[18] OpenGL Architecture Review Board, "OpenGL Reference Manual", Addison-Wesley, 1992.

[19] M. Peercy, J. Airey, B. Cabral, "Efficient Bump Mapping Hardware", *Computer Graphics* **31**(4), pages 303-306, 1997.

[20] B. T. Phong, "Illumination for computer generated images", *Communications of the ACM* **18**(6), pages 311-317, 1975.

[21] I. Sase, N. Shimizu, T. Yoshikawa, "Multimedia LSI Accelerator with Embedded DRAM", *IEEE Micro* **17**(6), pages 49-54, 1997.

[22] A. Schilling, G. Knittel, W. Straßer, "TEXRAM - A Smart Memory for Texturing", *IEEE Computer Graphics and Applications* **16**(3), pages 32-41, 1996.

[23] A. Schilling, Towards Real-Time Photorealistic Rendering: Challenges and Solutions, *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 7-15, 1997.

[24] D. Voorhies, J. Foran, "Reflection Vector Shading Hardware", *Computer Graphics* **28**(4), pages 163-166, 1994.

[25] D. Wong, M. J. Flynn, "Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations", *IEEE Transactions on Computers*, **41**(8), pages 981-995, 1992.