# Lighting models in Computer Games Act II

Ivan Viola
viola@cg.tuwien.ac.at

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna / Austria

## 1 Abstract

In this paper I want to show some basic principles of lighting, that are often used in Computer Games Technology. I will focus on the Dungeon game type, where probably the biggest effort in the high quality real-time rendering was put. As a typical member, as well as pioneer of real 3D computer games, I chose the Quake. I will describe what kind of optimizations were done in the software version and show some advantages of using hardware acceleration, which is actually the main stream in developing computer games.

In the second part I will shortly sketch the state of the art of real-time almost photorealistic techniques using hardware acceleration, that handle problems such as film projector simulation, environment mapping, BRDF based rendering or anisotropic lighting.

**KEYWORDS:** computer graphics, computer games, hardware acceleration, real-time graphics.

## 2 Introduction

In the real-time applications there is always a fight between speed and quality. In fact to consider an application as real-time application there must be a refresh frequency of at least 10 frames per second. Six years ago even this poor real-time, was not that easy to achieve without doing any optimization tricks. In that time all the rendering performed in computer games was software based, that means with no additional hardware acceleration. Today, when almost every graphics chip is equipped with 3D operations and supports some standard API, this is not interesting anymore. The graphics card is able to do per-pixel shading so the resulting frames are of high quality without doing some tricky optimizations. Today's reseach upon this area seems to be focusing on taking advantage of the hardware to achieve even more and more photorealistic results.

## 3 Quake

Probably the game, that shocked the world most with its high quality real-time rendering was Quake [1]. Therefore to be familiar with nowadays' concepts of real-time games, there is a need to be familiar with the previous work. Actually most optimizations implemented in Quake are still used today. The lighting can be divided into two parts:

- lighting of the world

- lighting of the creatures, weapons, particles or sprites

According to this parts quite different techniques were used to speed up the rendering process. The lighting of the world uses some preprocessing steps, while rendering of large polygon models is performed in run-time. Quake was also one of the first, that took advantage from hardware accelerators. Actually after GLQuake version, pure software rendering techniques were not developed any more.

## 3.1 World Lighting Model

The traditional way of doing polygon lighting is to calculate the correct light at the vertices and linearly interpolate between those points, also called Gouraud shading or smooth shading, but this has several disadvantages; in particular, it makes it hard to get detailed lighting without creating a lot of extra polygons, the lighting isn't perspective correct, and the lighting varies with viewing angle for polygons other than triangles (see figure 1).

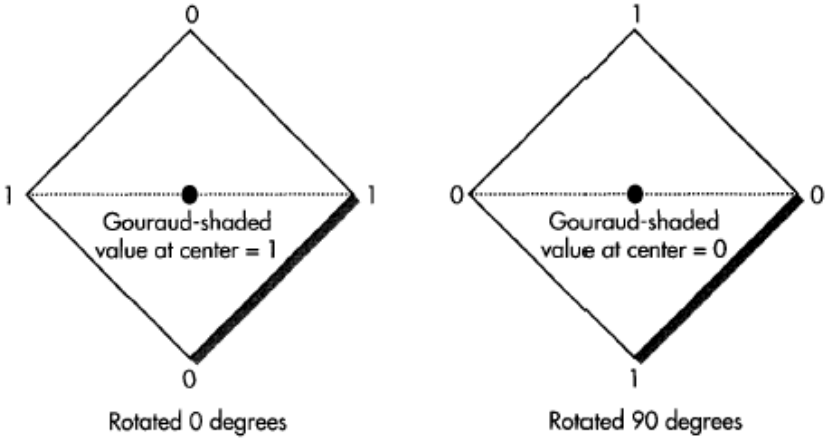


Figure 1: The angle variance using Gouraud shading.

To address these problems, Quake uses surface-based lighting instead. In this approach an extra rendering step is added. During off-line preprocessing, a grid, called a light map, is calculated for each polygon in the world, with a lighting value every 16 texels horizontally and vertically. This lighting is done by casting light from all the nearby lights in the world to each of the grid points on the polygon, and summing the results for each grid point. The Quake preprocessor filters the values, so shadow edges don't have a stair-step appearance. Then, at runtime, the polygon's texture is tiled into a buffer, with each texel lit according to the weighted average intensities of the four nearest light map points (see figure 2).

Then, the polygon is drawn to the screen using the perspective-correct texture mapping described above, with the prelit surface buffer being the source texture, rather than the original texture tile. No additional lighting is performed during texture mapping; all lighting is done when the surface buffer is created. Certainly it takes longer to build a surface buffer and then texture map from it than it does to do lighting and texture mapping in a single pass. However, surface buffers are cached for reuse, so only the texture mapping stage is usually needed. Quake surfaces tend to be big, so texture mapping is slowed by cache misses; however, the Quake approach doesn't need to interpolate lighting on a pixel-by-pixel basis, which helps speed things up, and it doesn't require additional polygons to provide sophisticated lighting. On balance, the performance of surface-based drawing is roughly comparable to tiled, Gouraud-shaded texture mapping-and it looks much better, being perspective correct, rotationally invariant, and highly detailed. Surface-based drawing also has the potential to support

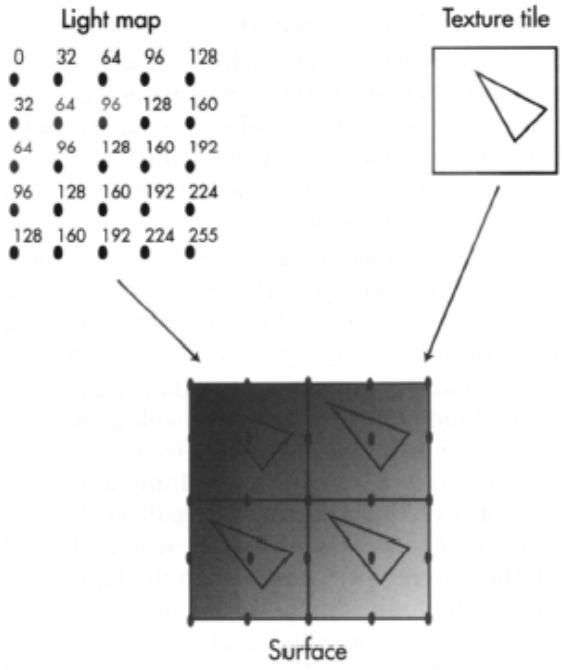


Figure 2: Generating the lighting using lightmap.

some interesting effects, because anything that can be drawn into the surface buffer can be cached as well, and is automatically drawn in correct perspective. For instance, paint splattered on a wall could be handled by drawing the splatter image as a sprite into the appropriate surface buffer, so that drawing the surface would draw the splatter as well.

## 3.2 Dynamic Lighting of the World

After some time the developers of Quake decided to add support of dynamic lighting, because explosions and such effects didn't produce impressive lighting effects. This means in practice that dynamically lit surfaces have to be rebuilt as the lighting changed. This is a significant drawback of dynamic lighting that it makes surface caching worthless for dynamically lit surfaces. The idea was to take the dynamic light sources and project their sphere of illumination into the world, and which would then add the dynamic contributions into the appropriate light maps and rebuild the affected surfaces. One interesting point about Quake's dynamic lighting is how inaccurate it is. It is basically a linear projection, accounting properly for neither surface angle nor lighting falloff with distance-and yet that's almost impossible to notice unless you specifically look for it, and has no negative impact on gameplay whatsoever. Motion and fast action can surely cover for a multitude of graphics sins.

It's well worth pointing out that because Quake's lighting is perspective correct and independent of vertices, and because the rasterizer is both subpixel and subtexel correct, Quake worlds are visually very solid and stable. This was an important design goal from the start, both as a point of technical pride and because it greatly improves the player's sense of immersion.

## 3.3 Polygon Models, Particles, Sprites and Z-Buffering

Polygon models, such as monsters, weapons, and projectiles, consist of a triangle mesh with front and back skins stretched over the model. The particles and sprites are in

fact just a single polygon each. These are used to add some visual effects such as rocket trails, explosions or cores of explosion. For speed, the triangles are drawn with affine texture mapping; the triangles are small enough, and the models are generally distant enough, that affine distortion isn't visible. The triangles are Gouraud shaded; interestingly, the light vector used to shade the models is always from the same direction, and has no relation to any actual lights in the world (although it does vary in intensity, along with the model's ambient lighting, to match the brightness of the spot the player is standing above in the world). Even this highly inaccurate lighting works well, though; the Gouraud shading makes models look much more three-dimensional, and varying the lighting in even so crude a way allows hiding in shadows and illumination by explosions and muzzle flashes.

One issue with polygon models is always how to handle occlusion issues; that is, what parts of models were visible, and what surfaces they were in front of. Quake's engine solves this problem using z-buffering. After all the spans in the world are drawn, the z-buffer is filled in for those spans. This is a write-only operation, and involves no comparisons or overdraw, so it's not that expensive-the performance cost is about 10%. Then polygon models are drawn with z-buffering; this involves a z-compare at each polygon-model pixel, but no complicated clipping or sorting-and occlusion is exactly right in all respects. Polygon models tend to occupy a small portion of the screen, so the cost of z-buffering is not that high.

## 3.4 The Subdivision Rasterizer

The Quake rendering engine does not use only Gouraud shading to shade polygonal models. In case of software based rendering another technique for shading and rasterizing is used called Subdivision Rasterizer. This rasterizer first draws all the vertices in the model. That means the float values are rounded to integral values. Then it takes each front-facing triangle, and determines if it has a side that is at least two pixels long. If it does, we split that side into two pieces at the pixel nearest to the middle (using adds and shifts to average the endpoints of that side), draw the vertex at the split point, and process each of the two split triangles recursively, until we get down to triangles that have only one-pixel sides and hence have nothing left to draw (see figure 3). This approach is hideously slow and quite ugly (due to inaccuracies from integer quantization) for 100-pixel triangles-but it s very fast for, say, five-pixel triangles, and is indistinguishable from more accurate rasterization when a model is 25 or 50 feet away. To achieve optimal results, there is a switch between Gouraud shading with classical rasterizerization and the subdivision rasterizerization. This swithing is based on the model's distance and average triangle size. In almost any scene, most models are far enough away so subdivision rasterization is used.

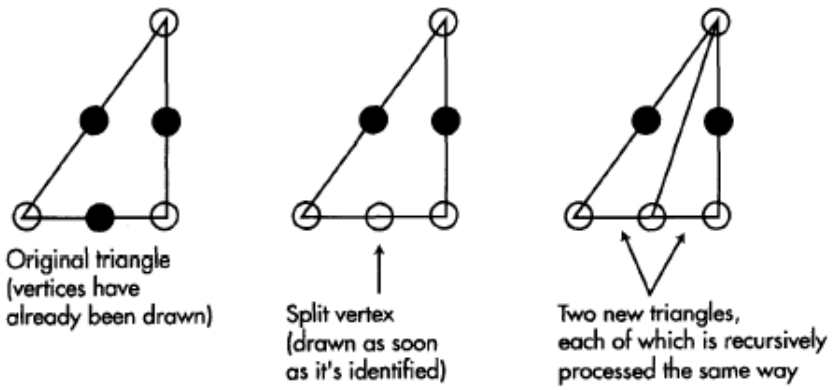


Figure 3: Subdivision Rasterizer.

## 3.5 Verite Quake

Verite Quake (VQuake) was the first hardware-accelerated version of Quake. It looks extremely good, due to bilinear texture filtering, which eliminates most pixel aliasing, and because it provides good performance at higher resolutions such as 512x384 and 640x480 (see figure 4). Interestingly, VQuake is very similar to software Quake; in order to allow Verite to handle the high pixel processing loads of high-resolution, VQuake uses an edge list and builds span lists on the CPU, just as in software Quake, then Verite DMAs the span descriptors to onboard memory and draws them. Similarly, the CPU builds lit, tiled surfaces in system RAM, then Verite DMAs sends them to an onboard surface cache, from which they are texture-mapped. In short, VQuake is very much like normal Quake, except that the drawing of the spans is done by a specialized processor. However there is a slight performance loss due to DMAing.

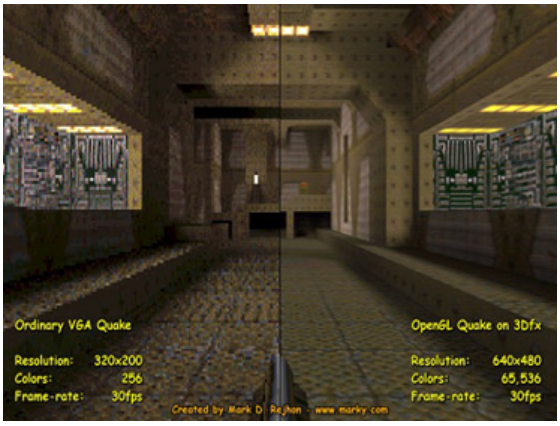


Figure 4: Standard Quake vs Quake using bilinear texture filtering.

## 3.6 GLQuake

The second port of Quake to a hardware accelerator was an OpenGL version. GLQuake uses two-pass alpha lighting, and runs very well on nowdays fast chips. It's generally faster than surface caching, without the need to build, download, and cache surfaces, and much better looking and about as fast as Gouraud shading. Dynamic lighting is done differently in GLQuake than in software Quake. GLQuake simply alpha-blends an approximate sphere around the light source. This requires very little calculation and no texture downloading, and as a bonus allows dynamic lights to be colored, so a rocket, for example, can cast a yellowish light.

There is also one eye-candy feature added to GLQuake: reflections. A special texture is designated as a mirror surface; when this is encountered while drawing, a hole is left. Then the z-range is changed so that everything drawn next is considered more distant than the scene just drawn, and a second scene is drawn, this time from the reflected viewpoint behind the mirror; this causes the mirror to be behind any nearer objects in the true scene. As a final step, a marbled texture is blended into the mirror surface, to make the surface itself less than perfectly reflective and visible enough to seem real.

## 3.7 Quake 2

The Quake 2 rendering engine is not very different from Quake; the improvements are largely in areas such as physics, gameplay, artwork, and overall design. The significant

change was using OpenGL out of the box as hardware acceleration and using of projective textures. Also interesting graphics change is in the preprocessing, the support for radiosity lighting was added; that is, the ability to put a light source into the world and have the light bounced around the world realistically. This is sometimes terrific - it makes for great glowing light around lava and hanging light panels - but in other cases it's less spectacular than the effects that designers can get by placing lots of direct-illumination light sources in a room, so the two methods can be used as needed. Also, radiosity is very computationally expensive, therefore lots of optimizations were done using Potentially Visible Set Algorithm, so only that subset of the BSP is rendered, which is potentially visible. Another replacement is an enclosing texture-mapped box around the world, at a virtually infinite distance; this will allow open vistas, much like Doom, a welcome change from the claustrophobic feel of Quake.

### 3.8 Quake 3

Although the last version of Quake seems to be quite different from the previous versions, this mostly because of using bezier patches and other improvements, that make the world as well as the polygon models more realistic. But the basic idea of lighting was used also here. There are some new features such as higher texture resolution or volumetric fogging that increase the quality of the visual effects. The biggest difference is the game style, which completely changed to a multiplayer game with six degrees of freedom. Comparing to the previous versions is the shading language also another important feature. The level designer can easy set up the shading style using this script language.

### 3.9 Doom 3

This game is not a regular member of the Quake family, but it is worth to mention it, because it is one of the newest dungeon games. The most interesting characteristic is unique style of rendering of the world, as well as the polygonal models. The graphics boards are mostly able to do pretty fast per-pixel shading, so there is not necessary to handle different rendering techniques for the world and mesh objects.

## 4 State of the Art in real-time high quality rendering

The major concern in the development of new graphics hardware has been to increase the performance of the traditional rendering pipeline. Today, graphics accelerators with a performance of several million textured and lit polygons per second are within reach even for the low end. This fact caused that the research was shifted away from higher performance towards higher quality renderings and an increased feature set, that makes graphics hardware applicable to more demanding applications.

One of the main research fields today is the effective use of textures for higher quality rendering. Not the classical textures itself, but available texture mapping hardware.

### 4.1 Projective Textures

This approach [5] is a simple extension to perspective-correct texture mapping, that can be used to create various lighting effects. These include arbitrary projection of two-dimensional images onto geometry, realistic spotlights and generation of shadows using shadow maps. This is something like projecting a slide of some scene onto an arbitrarily oriented surface, which is then viewed from some viewpoint (see figure 5).

The typical application is the simulation of a film projector. Similarly, this technique can also be used to simulate the effects of spotlight illumination on a scene. In this case the texture represents an intensity map of a cross-section of the spotlight's beam. It is as if an opaque screen was placed in front of the spotlight and the intensity at each point on the screen recorded. Any conceivable spot shape may be accomodated. In addition, distortion effects, such as those attributed to a shield or a lens may be incorporated into the texture map image. Another application of this technique is to produce shadows cast from any number of point light sources. The shadow map is created by rendering from the viewpoint of the light source in the preprocessing step.

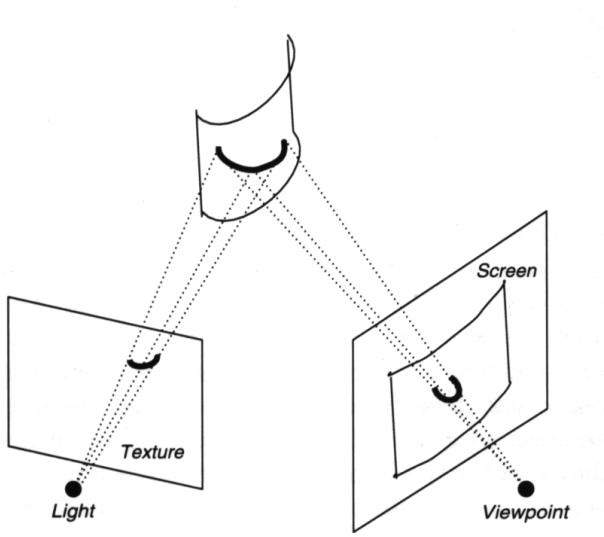


Figure 5: Projected Texture.

## 4.2 Environment Maps

This technique is used to precompute the scene lighting around a particular object and store it somehow in texure memory. After this is done, the scene lighting is simply mapped on the object and no computationally expensive scene rendering is necessary. The first step for using environment maps is a choice of appropriate parameterization. The spherical parameterization is based on the simple analogy of a infinitely small, perfectly mirroring ball centered around the object. The environment map is the image, that an orthographic camera sees when looking at this ball along negative z-axis. The sampling rate of this map is maximal for directions opposing the viewing direction, and goes towards zero for directions close to the viewing direction. Moreover, there is a singularity in the viewing direction, since all points are tangential to the sphere show the same point of the environment.

Another and more efficient parameterization is parabolic parametrization [3]. It is based on analogy similar to the one used to describe spherical maps. This assumes the reflecting object lies in the origin and the viewing direction is along a negative z-axis. The image seen by an orthographic camera when looking at the paraboloid contains the information about the hemisphere facing towards the viewer (see figure 6). The complete environment is stored in two separate textures and two binary masks, each containing the information of one hemisphere. The binary masks mark regions inside a circle that contain the information for the two hemispheres of directions. The regions outside the circle are not part of environment map, but are useful for avoiding seams between the two hemispheres of directions as well as for generating mipmaps of the environment. This parameterization is partly implemented in some graphics hardware.

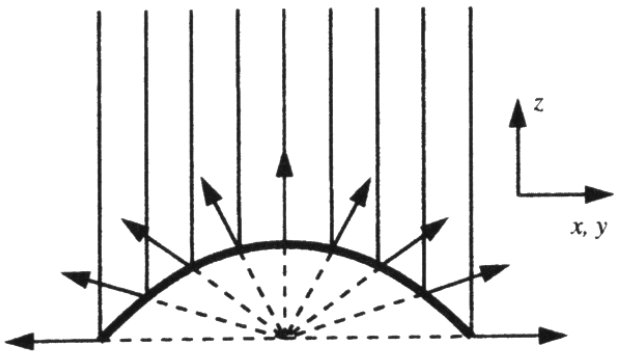


Figure 6: Parabolic Parameterization.

A parameterization which avoids the problem of spherical maps is cubical environment mapping, which consists of six independent perspective images from the center of a cube through each of its faces. This environment mapping is mostly used in the latest graphics boards.

## 4.3 BRDF-based Lighting

Techniques using the Bi-directional Reflectance Distribution Function for high quality rendering are handling the problem how to store and access the BRDF in texture mapping hardware. Since a BRDF is a 4D function (in the case of isotropic lighting), ideally it would be nice to be able to use a hardware accelerated 4D lookup table to store the BRDF and then at run-time do a per-pixel texture lookup to compute the lighting. Unfortunately, the current generation of graphics hardware does not provide support for 4D textures. One way to store the BRDF is by using two 2D textures to approximate the 4D function. The basic idea is to take the 4D BRDF and decompose or separate it into the product of two 2D functions in a preprocess step:

$$BRDF(\theta_i, \phi_i, \theta_o, \phi_o) = G(\theta_i, \phi_i) \cdot H(\theta_o, \phi_o). \tag{1}$$

Then at runtime, rather than using a single 4D texture lookup per-pixel, a pair of 2D texture lookups can be modulated together to compute the BRDF weighting function.

## 4.4 Anisotropic Lighting

This technique tries to render anisotropic surfaces such as satin cloth, compact disks, brushed metal or hail using hardware acceleration. The traditional lighting models consider surfaces as being locally flat, and the normal vector at a point on the surface is ortgogonal to the flatness. It assumes that surface tangents are uniformly distributed. Anisotropic surfaces cannot be considered as locally flat because of their microstructure, that creates rather different lighting results.

The standard local lighting model is given by the equation:

$$I_{out} = k_a \times I_a + k_d \times (\mathbf{L} \cdot \mathbf{N}) \times I_{in} + k_s \times (\mathbf{V} \cdot \mathbf{R})^n \times I_{in} \tag{2}$$

, where $I_{in}$ is the incoming intensity from the light, $k_a$, $k_d$ and $k_s$ are ambient, diffuse and specular constants that describe the material characteristics. Vector $\mathbf{L}$ is the vector towards the light, $\mathbf{N}$ is the surface normal vector, $\mathbf{V}$ is vector to the viewer and $\mathbf{R}$ is the reflection vector. Coefficient $n$ describes the shinness of the material.

To achieve the anisotopic surfaces appear realistic, it is important that their microstructure is taken into account. There is another problem how to assign a normal to a single hair, that can be considered as a line, which has infinite number of normals. A good solution appears to be to take only the most significant normal, normal that is co-planar with the light and view vector, into account. But it is not necessary to determine the most significant normal explicitly, the ($\mathbf{L}\cdot\mathbf{N}$) and ($\mathbf{V}\cdot\mathbf{R}$) can be expressed in terms of $\mathbf{L}$, $\mathbf{T}$, and $\mathbf{V}$, where $\mathbf{T}$ is the tangent vector:

$$(\mathbf{L}\cdot\mathbf{N}) = \sqrt{1-(\mathbf{L}\cdot\mathbf{T})} \tag{3}$$

$$(\mathbf{V}\cdot\mathbf{R}) = \sqrt{1-(\mathbf{L}\cdot\mathbf{T})^2}\times\sqrt{1-(\mathbf{V}\cdot\mathbf{T})^2}-(\mathbf{L}\cdot\mathbf{T})\times(\mathbf{V}\cdot\mathbf{T}) \tag{4}$$

This solution offers to compute the two dot products ($\mathbf{L}\cdot\mathbf{T}$) and ($\mathbf{V}\cdot\mathbf{T}$) in hardware using OpenGL 4x4 texture matrix and to scale it from [-1,1] dot product space to [0,1] texture space:

$$\frac{1}{2}\times\begin{bmatrix} L_x & L_y & L_z & 1 \\ V_x & V_y & V_z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}\cdot\begin{bmatrix} T_x \\ T_y \\ T_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1/2\cdot(\mathbf{L}\cdot\mathbf{T})+1 \\ 1/2\cdot(\mathbf{V}\cdot\mathbf{T})+1 \\ 0 \\ 1 \end{bmatrix} \tag{5}$$

The only thing that is necessary to do, is to compute the $I_{out}$ values of the 2D lookup table (see equation 2). After this is done, whole the computation is reduced into computing of two dot products using mostly hardware. Unfortunately this kind of hardware acceleration assumes constant viewing and light vector.

# References


[1] M. Abrash. Ramblings in Realtime `http://www.bluesnews.com/abrash/`, 2000.

[2] M. Hadwiger. 3D Graphics Technology in Computer Games Past, Present, and Future `http://www.cg.tuwien.ac.at/studentwork/CESCG-2000/MHadwiger/`, 2000.

[3] W. Heidrich, H. Seidel. Realistic, Hardware-accelerated Shading and Lighting *In Proceedings of SIGGRAPH 1999*, pages 171–178, 1999.

[4] M. Kilgard. Hardware Accelerated Anisotropic Lighting `http://www.nvidia.com/Marketing/Developer/DevRel.nsf/pages/7B0FF06DC752E7438825681F0001571A`, 1999.

[5] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, P. Haeberli. Fast Shadows and Lighting Effects Using Texture Mapping *In Proceedings of SIGGRAPH 1992*, pages 249–252, 1992.

[6] M. Wynn. BRDF-based Lighting `www.nvidia.com/Marketing/Developer/DevRel.nsf/045cfdb95e306a4d882568170059fa93/d081b8131427a50b88256a070076d822/\$FILE/BRDFs.pdf`, 2000.

[7] M. Zckler, D. Stalling, H.C. Hege. 3D-Vector Fields Using Illuminated Stream Lines *In Proceedings of Visualisation '96*, pages 107–113, 1996.