# Lighting effects in Computer Games
# Act I

Stefan Felkel, 2001
steffel@utanet.at

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

## Abstract

In this paper you will receive information about basic lighting techniques and lighting effects rendered in real time. I will discuss shading methods and their usage nowadays, for example Phong shading and cube maps. Radiosity is an important technique for light map calculation and often heard word, described in the paper, too. Finally lighting in history is presented by showing screenshots of 3D-games since 1992.

## 1 Introduction

Light and lighting is probably the most important element in computer games and special effects. Without the correct lighting, even the best computer games would be boring and unexciting (Figure 1-2). Why is lighting this important? - If we have no special lighting, we see one texture like one picture and are not able to distinguish between elements that are far away or very near.
We cannot see, whether there is an edge

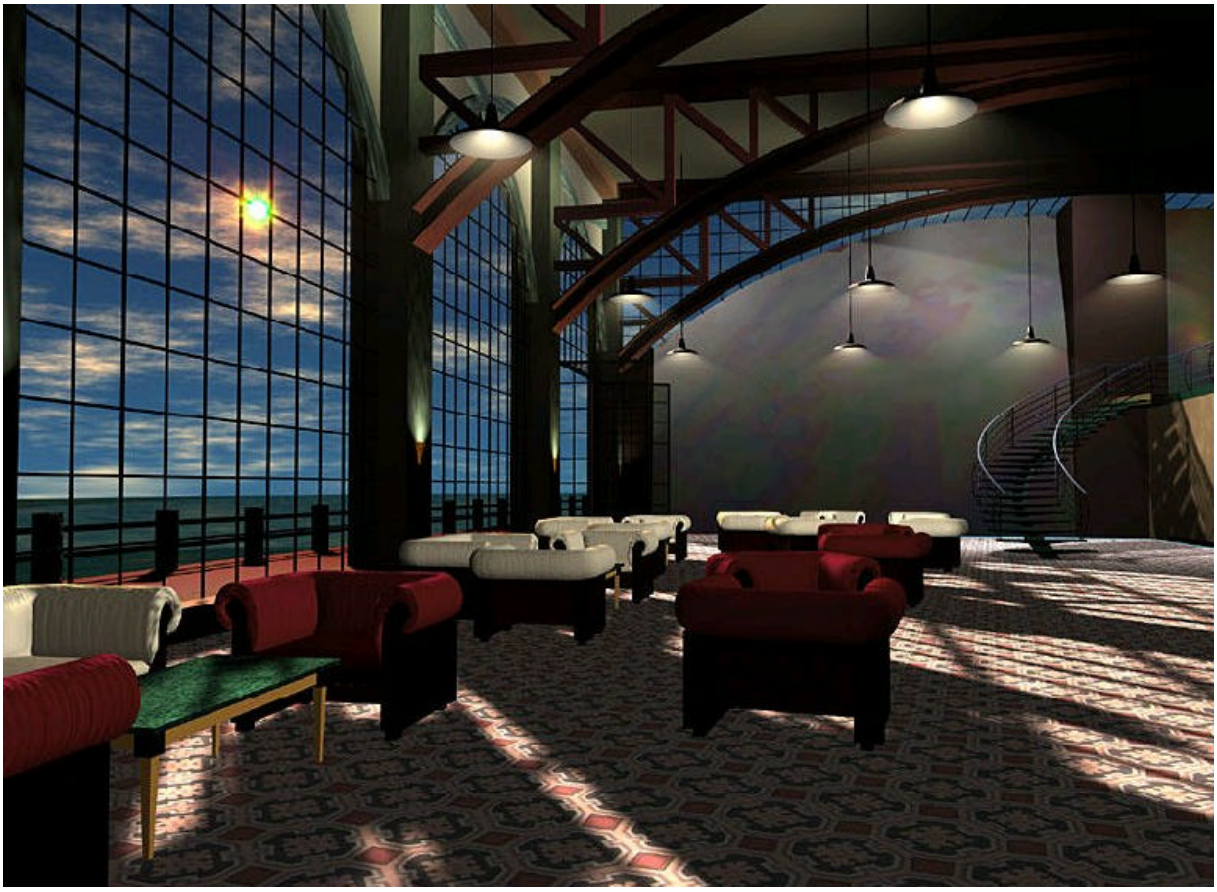

Figure 1-1: Example of lighting

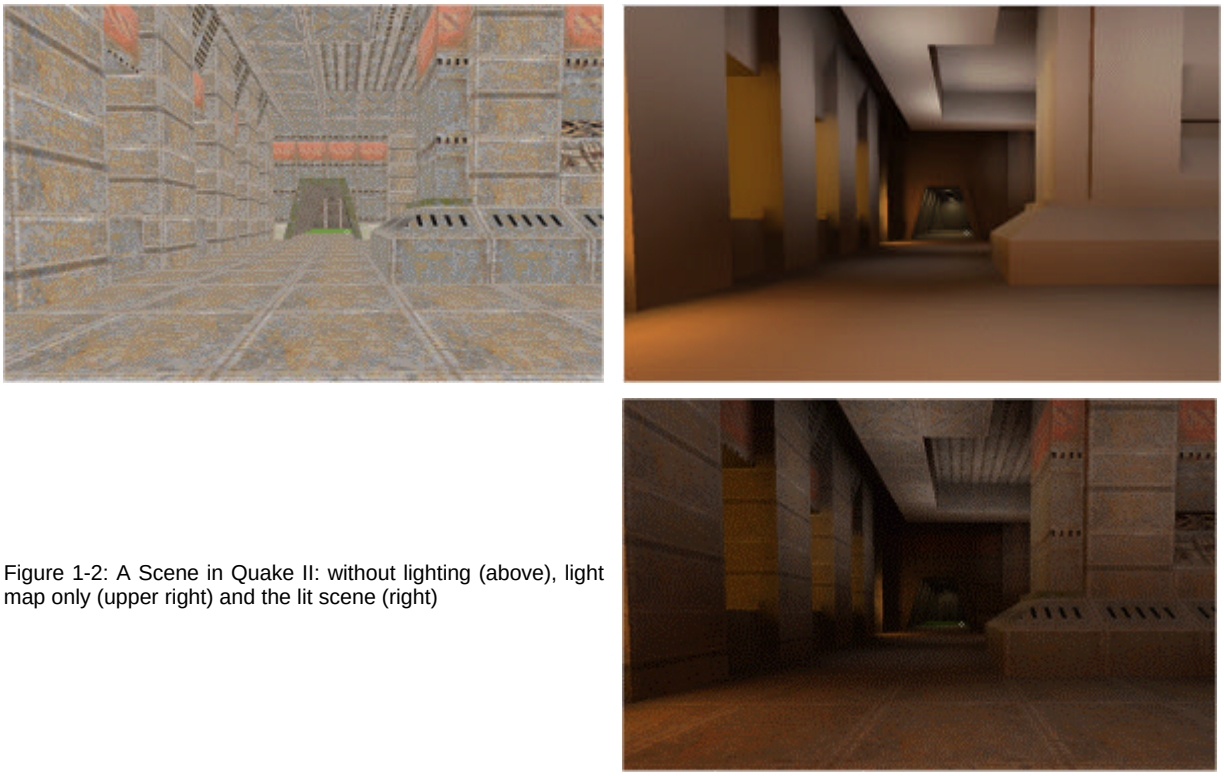

Figure 1-2: A Scene in Quake II: without lighting (above), light map only (upper right) and the lit scene (right)

or a deepening or a recess, because there is no special light and therefore - no shade.
Lighting gives a scene reality and the player thinks more and more that he lives in this world because it is quite similar to ours.

## 2 Light Sources and Lighting model

There are three different types of light sources, directional lights, point lights and spot lights (Figure 2-1).

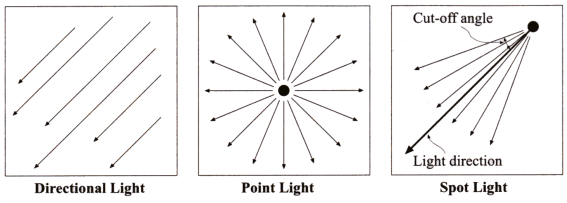


Figure 2-1: Three different types of light sources.

A directional light is positioned infinitely far away from the objects that are being lit. The sun is the best example for directional light.
Point and spot lights are called “positional lights”, because they each have a fix location in space, just the direction of light emission is different.
The positional light is like a single point that emits photons.
Both positional light and directional light sources cast shadows with sharp edges, in contrast to real light sources, where we can observe shadows with soft edges.
Every light source has intensity parameters and sometimes a colour, too. In OpenGL, the common light source parameters are described in Table 2-1.

| | |
|---|---|
| GL_AMBIENT | Ambient intensity color |
| GL_DIFFUSE | Diffuse intensity color |
| GL_SPECULAR | Specular intensity color |
| GL_POSITION | Light source position |

Table 2-1: Common Values for parameters of light sources

Programmers usually use up to eight light sources in the same scene in OpenGL, possibly more, depending on their OpenGL implementation. But the more light sources you have, the more OpenGL has to calculate, e.g. to determine how much light each vertex receives from each light source; and increasing the number of lights affects performance.

Maybe you heard the term “lighting model” already: All the several influences of light sources, the shadow-relationships, the types of surfaces, the material-properties, the position of the viewer and the positions of all objects in the related room are kept in a lighting model. It is not exactly the same we understand under reality and it is impossible to transform all physical calculations our reality into the computer. Nevertheless the lighting models nowadays appear very similar to our world.
There are models describing the interaction of light and material, for instance the Cook-Torrance’s model or models that discuss more intuitive user controls for setting material values, for example the Strauss model.
The illumination at the vertices is computed using this lighting model. It can be parted into diffuse, specular and ambient components.

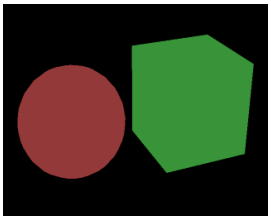

Figure 2-2: A sphere and a cube with ambient lighting

Let's start with the ambient component (Figure 2-2): In our simple lighting model, lights shine directly on surfaces, but nobody cares of the reflection. In our real world, light might bounce off a wall and then reach an object. This reflected light would not be accounted for in either the diffuse or the specular component. To simulate this indirect lighting, the lighting model includes the ambient term which is usually just some combination of material and light constants, as shown in the following equation:

$$i_{amb} = m_{amb} \otimes s_{amb}$$

$m_{amb}$ ... *ambient color of the material*

$s_{amb}$ ... *ambient color of the light source*

Adding this constant term means that an object will receive some minimum amount of color, even if not directly illuminated. In this way surfaces facing away from a light will not appear entirely black.
Another common technique (used e.g. in VRML browsers) is to use a headlight, which is a point light attached to the viewers location. As the viewer moves, so does the light, and the light provides all surfaces with varying degrees of brightness.

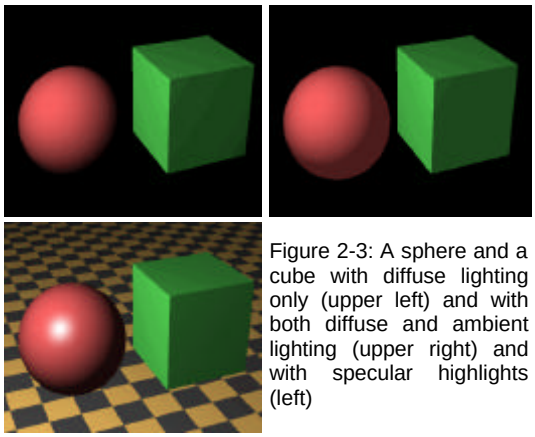

Figure 2-3: A sphere and a cube with diffuse lighting only (upper left) and with both diffuse and ambient lighting (upper right) and with specular highlights (left)

The two major components to simplify the task of modelling the physics of light in 3D graphics are diffuse lighting and specular lighting (Figure 2-3). Diffuse lighting assumes the light hitting an object scatters in all directions equally, so the brightness of the reflected light does not depend at all on the position of the viewer. Sunlight on a playground is an example in the real world of diffuse lighting. The brightness of an object in the scene is primarily determined by the diffuse lighting calculations. The equation for the diffuse contribution is:

$$i_{diff} = m_{diff} \otimes s_{diff} (n \cdot l)$$

$m_{diff}$ ... *diffuse color of the material*

$s_{diff}$ ... *diffuse color of the light source*

*n ... surface normal vector*

*l ... light vector*

Specular lighting is different because it does depend on the position of the viewer as well as the direction of light and orientation of the triangle being rendered. Shining a spotlight into a dark corner of a room onto a TV set and looking at the hot spots on the picture tube will show you specular lighting. Specular lighting captures the mirror-like properties of an object so effects such as reflection and glare are achievable. Figure 2-4 shows two examples of a space station from the 3D WinBench® benchmark that demonstrate the differences between diffuse and specular lighting effects.

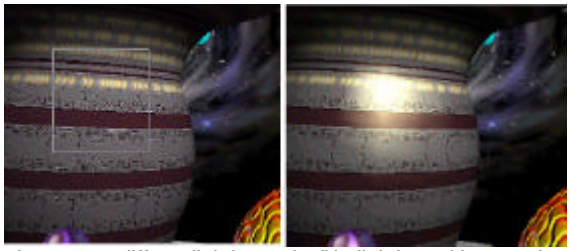

Figure 2-4: diffuse lighting only (l.), lighting with specular component, too (r.)

Specular highlights move on the object if the viewer or the object moves relative to the light source. For this reason they cannot be pre-computed or static.

Specular lighting is also important for representing different materials for objects in a 3D scene. A silk shirt looks different than a cotton shirt, even if they are the same color. A major difference is how the two materials reflect light, which is captured with specular lighting. Another example is polished stone such as marble compared to the same material before polishing. The polishing doesn't change the color or pattern in the marble, but it does affect the way light is reflected. Specular lighting combined with texture mapping creates more realistic objects because they have the visual properties of real materials.

Specular lighting can be done without a dedicated hardware lighting engine, but only with a severe loss of performance. Texture mapping can be used for some specular lighting effects, but if the viewer moves around in the scene, then the environment maps must be re-calculated, which is a time-consuming process unless the graphics hardware supports cube environment mapping. Sphere mapping is the common alternative but it suffers

from performance and quality problems as the viewer moves around in the scene, making it unattractive for interactive 3D environments. [TRL] At least, the equation for the specular component is:

$$i_{spec} = (r \cdot v)^{m_{spec}}$$

$m_{spec}$ ... *diffuse color of the material*

$r$ ... *reflection vector*

$v$ ... *view vector*

With these equations we can evaluate the total lighting intensity with:

$$i_{tot} = i_{amb} + i_{diff} + i_{spec}$$

$$i_{tot} = m_{amb} \otimes s_{amb} + m_{diff} \otimes s_{diff} (n \cdot l) + (r \cdot v)^{m_{spec}}$$

OpenGL approximates light and lighting as if light can be broken into red, green and blue components. Thus, the color of light sources is characterized by the amount of red, green and blue light they emit, and the material of surfaces is characterized by the percentage of the incoming red, green and blue components that is reflected in various directions. The OpenGL lighting equations are just an approximation but one that works fairly well and can be computed relatively quickly. If you desire a more accurate or different lighting model, you have to do your own calculations in software. Such software can be quite complex.

In the OpenGL lighting model, the light in a scene comes from several light sources that can be individually turned on an off. Some light comes from a particular direction or position, and some light is generally scattered about the scene. For example, when you turn on a light bulb in a room, most of the light comes from the bulb, but some light comes after bouncing off one or more walls. This bounced light (called, as we already know, ambient) is assumed to be so scattered that there is no way to tell its original direction, but it disappears if a particular light source is turned off [OGL].

## 3 Material and its Colors

In real-time systems a material consists of a number of material parameters, namely ambient, diffuse, specular, shininess, and emissive. The color of a surface with a material is determined by these parameters, the parameters of the light sources that illuminate the surface, and a lighting model.

The OpenGL lighting model makes the approximation that a material's color depends on the percentages of the incoming red, green, and blue light it reflects. For example, a perfectly red ball reflects all the incoming red light and absorbs all the green and blue light that strikes it. If you view such a ball in white light (composed of equal amounts of red, green, and blue light), all the red is reflected, and you see a red ball. If the ball is viewed in pure red light, it also appears to be red. If, however, the red ball is viewed in pure green light, it appears black (all the green is absorbed, and there's no incoming red, so no light is reflected).
Like lights, materials have different ambient, diffuse, and specular colors, which determine the ambient, diffuse, and specular reflectances of the material. A material's ambient reflectance is combined with the ambient component of each incoming light source, the diffuse reflectance with the light's diffuse component, and similarly for the specular reflectance and component. Ambient and diffuse reflectances define the color of the material and are typically similar if not identical. Specular reflectance is usually white or gray, so that specular highlights end up being the color of the light source's specular intensity. If you think of a white light shining on a shiny red plastic sphere, most of the sphere appears red, but the shiny highlight is white. But the color may change, depending on the materials properties, for example: the difference of plastic and metal. With plastic you have more the color of the light source at the specular reflection, with metal, you will receive more the color of the material, this can be made by setting material's specular color to 0.
In addition to ambient, diffuse, and specular colors, materials have an emissive color, which simulates light originating from an object. In the OpenGL lighting model, the emissive color of a surface adds intensity to the object, but is unaffected by any light sources. Also, the emissive color does not introduce any additional light into the overall scene [OGL].

## 4 Lighting, Shading and Cube Maps

Lighting stands for the interaction between materials and light sources and can also be used with colors, textures and transparency. All these elements are combined into a visual appearance on the screen.

Shading performs lighting computations and determines pixels' colors from them. We distinguish between three main types of shading: Flat shading, Gouraud shading and Phong shading (Figure 4-1). These correspond to computing the light per polygon, per vertex and per pixel.

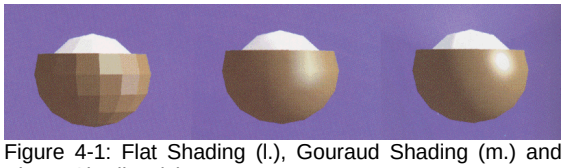

Figure 4-1: Flat Shading (l.), Gouraud Shading (m.) and Phong Shading (r.)

Romney, Warnock and Watkins introduced Flat shading 1967. This method runs fast and is simple

to implement. Each polygon gets an own color, which is calculated related to the corresponding light source. So the color will change abruptly with the appearance of the next polygon. If these objects have an angle of about 90 degrees, it will not look bad, but if we observe a sphere or something round, the quality of this shade is very low.

Henri Gouraud 1971 presented Gouraud shading.
In Gouraud shading, the lighting at each vertex of a triangle is determined, and these lighting samples are interpolated over the surface of the triangle (Figure 4-2). So we get a smooth look to round objects. Gouraud shading is nearly as fast as flat shading; the only problem is the dependency on the level of detail of the objects that are rendered.

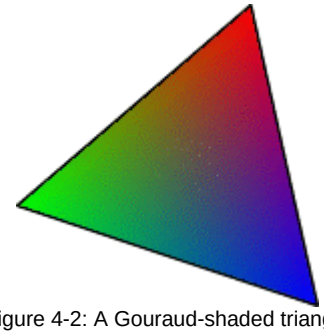

Figure 4-2: A Gouraud-shaded triangle

So we come to Phong shading which avoids this problem by interpolating the surface normal. At each pixel, you need to renormalize the normal vector, and also calculate the reflection vector.
But this per-pixel lighting costs time and needs more calculation operations, so this type of shading is uncommon in commercial graphics hardware, more expensive hardware, like the GeForce 2 chip would support this operations in hardware [RTR].
So, one of the most expensive operations in Phong Shading is the vector normalization. In most hardware this is only possible per-vertex, but not per-pixel. On recent hardware that supports cube maps, however, this feature can be “misused” for performing this vector normalization. In this way normalization per-pixel becomes possible. This works like looking up a Color in the cube map, but instead of the color there will be stored a normalized vector.
A vector is mapped to a two-dimensional value on a surface of the cube map, there will be stored a RGB-color-value:

$$color\ cubemap: \begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} u \\ v \end{pmatrix} \rightarrow \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

The other possibility is to store the normalized vector of (x,y,z) at one point on the cube map:

$$normalized\ vector\ cubemap: \begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} \frac{x}{\sqrt{x^2+y^2+z^2}} \\ \frac{y}{\sqrt{x^2+y^2+z^2}} \\ \frac{z}{\sqrt{x^2+y^2+z^2}} \end{pmatrix}$$

## 4.1 Cube Maps and Environment Mapping

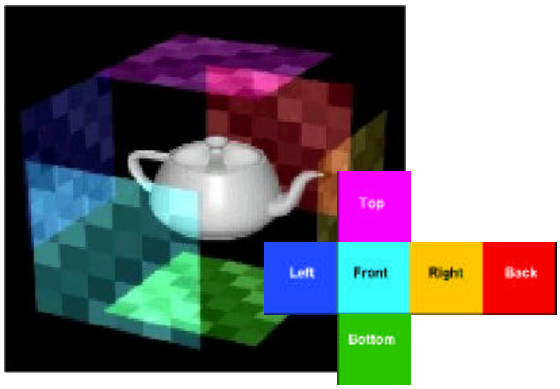


Figure 4-3: Cube map transformed into 2D (lower right) and Cube map in a 3D-environment (left)

Cube environment mapping (Figure 4-3) in hardware is a breakthrough image quality feature of GeForce that is fully supported by DirectX 8 and OpenGL and will allow developers to create accurate, real-time reflections.
Accelerated in hardware, cube environment mapping will free up the creativity of developers to use reflections and specular lighting effects to create interesting, immersive environments. By changing the map shape to a six-sided cube, cube environment mapping offers a simple development path to the creation of stunning, reflective images.

Cube environment mapping is a new form of texture access, and is not just limited to simulating accurate reflections. The GeForce with accelerated cube environment mapping can also look up textures given a normal or any application-specified vector, enabling a whole new set of graphical effects.
For example, cube environment mapping will be used to create realistic specular lighting highlights (Figure 4-4). Typically, simulating a glossy object like a billiard cue ball results in "seamed" specular lighting as vertex, per-vertex lighting is employed. By rendering specular lighting into the cube map, crisp per-pixel specular lighting effects can be created. Cube Mapping allows specular lighting lookups against the cube map on a per-pixel basis, enabling seamless specular highlight effects. The cube mapping technique gives developers lighting capabilities superior to per-pixel Phong shading with an arbitrary number of lights, including

extended light sources. With a little creativity, a broad range of interesting, photo realistic effects can be created. For example, by blurring the cube map, objects that have rough surfaces such as brushed aluminium can be simulated. NVIDIA's GeForce supports very large cube maps, for very high quality lighting effects [PHO].

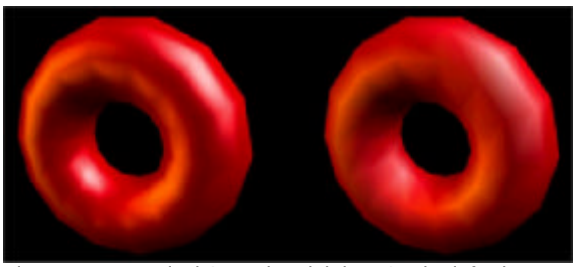

Figure 4-4: Per-Pixel Specular Lighting: On the left "donut", per-pixel lighting through cube mapping enables crisp, smooth specular highlights. The donut on the right uses vertex lighting. Notice the "banding" and edges that are clearly visible on the specular light on the upper right of the image.

## 5 Multipass Rendering

In software rendering, all illumination equation factors are evaluated at once and a sample color is generated. In hardware and so in practice, the various parts of the lighting equation can be evaluated in separate passes, and each successive pass is modifying the results of the pass before. This technique is called multipass rendering.
Hardware accelerators are not flexible enough to do all lighting effects in a single pass. For instance, you want to have light, modified by a texture and another part of the light, maybe the specular part, not modified by the texture. So you have to handle this with two passes: the first will be the one you compute and interpolate the light and modulate it by the texture, the second pass is: you compute and interpolate the specular part without modifying it by the texture and at least add these two textures with the two light effects on it.
Multipass rendering is useful when you want to enable a rendering system to work on several kinds of hardware. For example, Quake III uses up to 10 passes, depending on the number of effects.

## 6 Multitexturing

The more passes you use for rendering, clearly the more the performance will decrease. To reduce the number of passes, most graphics accelerators support multitexturing, in which two or more textures are accessed during the same pass (the standard is two-texture multitexturing). To combine the results of these texture accesses, a “texture blending cascade” (pipeline) is defined that is made up of a series of “texture stages”, also called “texture units”.
The first texture stage combines the fragment color and the first texture, typically RGB, and this result is then passed on to the next texture stage. Second and successive stages then blend another texture’s or interpolant’s values with the previous result.

## 7 Light mapping

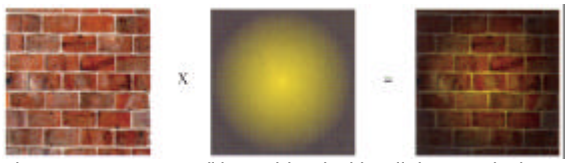

Figure 7-1: A texture (l.) combined with a light map (m.)

Mapping is the process of laying a texture on an object. If you want to do this procedure with light, we have to use a light map, which stores information of the illumination for this surface. So light maps are low-resolution textures that are calculated for every polygon in a scene. Because they are precalculated, the values of pixels can be calculated using very expensive techniques, like radiosity – you will hear about this topic later. This way, you can have very good static lighting, complete with shadows [FTL].
Today every upcoming game uses light maps. Light mapping was introduced with Quake's lighting model. Some years ago, programmers designed their games by rendering light and textures in one texture and put these textures in the game.
With light maps you are more flexible, maybe you want to change illumination on a specified wall suddenly, so you just have to get another light map and render this in real time.
Furthermore you obtain more memory. You need only one big light map, in which many light maps are stored and accessed by 2D-coordinates (Figure 7-2).

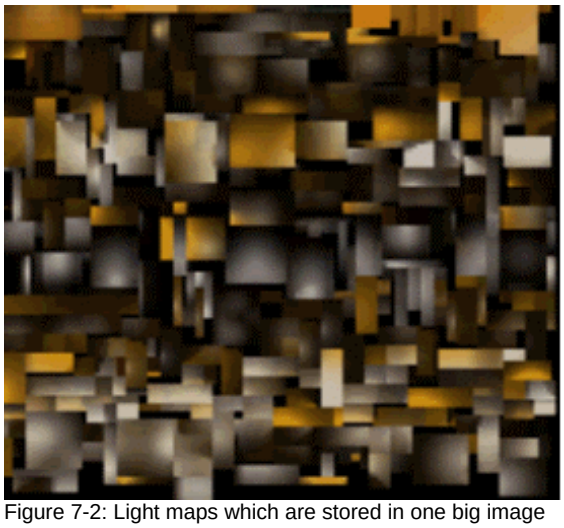

Figure 7-2: Light maps which are stored in one big image

To generate a light map, look at the following picture (Figure 7-3):

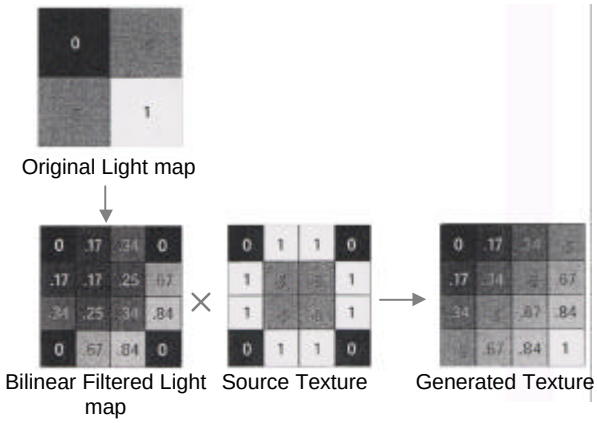


Figure 7-3: A light map at level 1 is bilinear filtered up to level 2 so that it can be combined with a source texture at level 2 to generate (by multiplying) the resultant texture [GDM].

## 8 Gloss Mapping

This technique is declared as followed:
Image light maps are often typically used on diffuse surfaces, and so are sometimes called diffuse light maps. The specular component can also be affected by light mapping, but here the effect is sometimes a little more involved.
The specular component is computed using the eye direction and the light direction. In real-time work it is typically calculated at polygon vertices and interpolated. Just as a texture can be used to modulate the diffuse lighting on a surface to produce a brick wall effect, so can a different texture produce varying specularity. Such a texture is typically a monochrome (gray-scaled) texture, where 1.0 means the full specular component is to be used, and 0.0 means none is used.

## 9 Radiosity

The radiosity method of computer image generation has its basis in the field of thermal heat transfer. Heat transfer theory describes radiation as the transfer of energy from a surface when that surface has been thermally excited. This includes both surfaces that are basic emitters of energy, as with light sources, and surfaces that receive energy from other surfaces and thus have energy to transfer.
This "thermal radiation" theory can be used to describe the transfer of many kinds of energy between surfaces, including light energy.

As in thermal heat transfer, the basic radiosity method for computer image generation makes the assumption that surfaces are diffuse emitters and reflectors of energy, emitting and reflecting energy uniformly over their entire area. It also assumes that a balance solution can be reached; that all of the energy in an environment is described - through absorption and reflection.

It should be noted that the basic radiosity method is viewpoint independent: the solution will be the same regardless of the viewpoint.

The radiosity equation describes the amount of energy which can be emitted from a surface, as the sum of the energy inherent in the surface (a light source, for example) and the energy which strikes the surface, being emitted from some other surfaces [SPE].

$$B_i = E_i + r_i \sum B_j F_{ij}$$

$B_i$ = Radiosity of surface i

$E_i$ = Emissivity of surface i

$?_i$ = Reflectivity of surface i

$B_j$ = Radiosity of surface j

$F_{ij}$ = Form Factor of surface j relative to surface i

The radiosity of a surface is the energy that is given off. This is what is used to determine the intensity of the surface and is what is being solved for. The amount of light emitted from a surface must be specified as a parameter in the model, just as in traditional lighting methods where the location and intensity of light sources must be specified. The reflectivity of the surface must also be specified in the model, just as in traditional lighting methods. The only unknown in the equation is the amount of incident light hitting the surface. This can be found by summing for all other surfaces the amount of energy that they contribute to this surface [RAD].

### 9.1 Meshing the Environment

The accuracy of the radiosity method depends very much on the underlying meshing techniques used to represent each surface. Each surface is divided into a mesh of polygons, commonly known as patches (Figure 9-1). A patch is the synonym for surface or area.

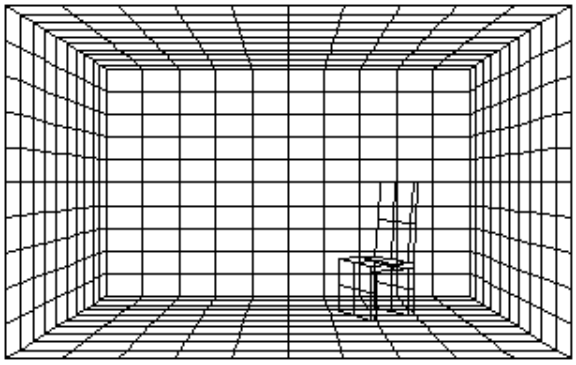

Figure 9-1: A room with a chair in patches. Scene generated using Helios.

Each patch will receive some light energy from the surrounding patches while emitting its own light energy.
Likewise, all other patches will behave the same. This process is iterative and continues until all reflected light energy is finally absorbed [LEE].

In general, there are more meshing strategies, the simplest ones are uniform meshing and non-uniform meshing [WAT].

## 10 Examples, Screenshots

### 10.1 Wolfenstein 3D

This game was released 1992 by id Software and opened a new computer game genre: the first-person shooter, commonly abbreviated as FPS.
The most important contribution of Wolfenstein 3D was that it was to become the prototype for graphically and technologically much more sophisticated first-person shooter in the following years, like Doom and Quake.
In this game only wall polygons were textured-mapped, floors and ceilings were filled with a solid color.
There was no lighting at all ("full-bright"). Textures got their color at their creation, so brighter textures seem to be illuminated (Figure 10-1), like there is a full-intensity white ambient light.

Figure 10-1: Screenshot of Wolfenstein 3D (id Software 1992)

### 10.2 Doom

When Doom appeared 1993, it made its predecessor Wolfenstein 3D a very simple game. Doom is still one of the most successful computer games of all time.

In relation to lighting, it used "depth cueing", which means that lighting is decreasing the deeper you look into a room or a landscape. The problem with this method is, that you see the stripes clearly; they are modelled horizontal to the viewer's sight (Figure 10-2). We can say, it is a "faked" ambient light, depending on the distance.

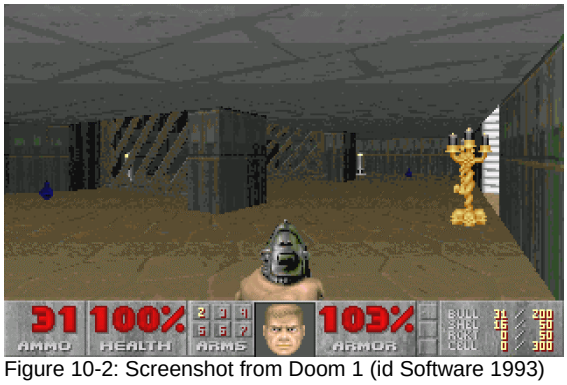

Figure 10-2: Screenshot from Doom 1 (id Software 1993)

### 10.3 GLQuake

In 1996, GLQuake, which differs from its predecessor Quake in the fact that it uses the OpenGL API, came up the first time. GLQuake renders the light maps directly as a second pass - in contrast to software-rendered Quake where light maps were combined with the base texture maps before actually using them for texturing.
If hardware is able to support it, base textures and light maps can even be rendered in a single pass, using single-pass multi-texturing.
GLQuake and all other games using the Quake II engine use radiosity for light map calculation ().


Figure 10-3: Screenshot of Quake II (id Software 1996)

## 11 References


[GDM] Jonathan Blow, “Implementing a Texture Caching System”, Game Developer April 1998

[RTR] Möller / Haines, “Real-Time Rendering”, A K Peters 1999

[PHO] Tom Hammersley, „Phong Shading”
(http://members.nbci.com/3dcoding/tom_h/phong.htm)

[TRL] NVIDIA Corporation, “Transform And Lighting”
(www.nvidia.com/developer)

[SPE] Stephen Spencer, Education Slide Set, SIGGRAPH 1993
(http://www.education.siggraph.org/materials/HyperGraph/radiosity/overview_1.htm)

[FTL] Kurt Miller, “Light mapping Revisited (And My (Re)Introduction)”, October 1999
(http://www.flipcode.com/kurt)

[RAD] Jaemin Lee, “What is Radiosity?”, Fall 2000
(http://www-viz.tamu.edu/students/softviz/viza617/presentation/radiosity.html)

[LEE] Tralvex S L, “Radiosity for VR Systems”, University of Leeds August 1997
(http://www.comp.leeds.ac.uk/cuddles/rover/thesis)

[WAT] Watt Alan, “3D Computer Graphics”, Harlow: Addison-Wesley, 1999

[OGL] Mason Woo, Jackie Neider, Tom Davis, “OpenGL Programming Guide Sec. Ed. V 1.1”, pp. 160-193, 1997