Lighting techniques for real-time 3D rendering

Created by

Kris Garrein

In order to get the degree of being graduated in Multimedia and communication technology.
Academic year 2001-2002
Lighting-techniques for real-time 3D rendering

Index

1) Introduction ..............................................................................................3
2) Tools / libraries / other helps ................................................................. 5
   a. OpenGL API .........................................................................................5
   b. Magic Software Library ......................................................................5
   c. DevIL Image Library .........................................................................6
   d. KRIS Library .....................................................................................6
   e. ExtGL Library ....................................................................................6
3) OpenGL extensions .................................................................................7
   a. Lead-in ................................................................................................7
   b. How are extensions documented ? ....................................................7
   c. Use of extensions ..............................................................................7
4) Possibilities of standard OpenGL lighting ...............................................9
   a. GL_LIGHTING en GL_LIGHTx ..........................................................9
   b. Normals ............................................................................................10
   c. Ambient, Diffuse, Specular, etc .......................................................11
   d. Materials ..........................................................................................12
5) Vertex Color lighting .............................................................................13
6) Blending ................................................................................................15
7) Multitexturing .......................................................................................17
8) Lightmapping .........................................................................................20
   a. Theory ...............................................................................................20
   b. Code example ...................................................................................22
   c. How does this calculation work ? ....................................................22
   d. Dynamic lightmaps via blending ....................................................24
   e. Comparison and conclusion ...........................................................26
9) OpenGL fog ...........................................................................................27
10) Rendering to texture ............................................................................28
11) Texture projection ................................................................................31
12) Bump Mapping ....................................................................................33
13) Plane Projected Shadows ....................................................................35
14) Projected shadows ..............................................................................38
15) Shadow mapping ..................................................................................41
16) Vertex projection ................................................................................42
17) Shadow Volumes ...............................................................................43
18) Comparison of shadowing-methods .....................................................44
   a. Comparisation ..................................................................................44
   b. Hybrid shadows .............................................................................45
19) Introducing nVidia nFiniteFX : Hardware programmability ..................46
   A) Lead-in ..............................................................................................46
      a. Vertex shaders ...............................................................................46
      b. Pixel shaders ................................................................................47
   B) Vertex Programming .........................................................................47
      a. Lead-in ..........................................................................................47
      b. Vertex Attribute Registers .........................................................48
      c. Program Parameter Registers ...................................................49
      d. Temporary Registers ..................................................................49
      e. Vertex Output registers ...............................................................49
f. Instruction set.................................................................49

g. Examples...........................................................................51

C) Pixel Programming.............................................................53
   a. Texture Shaders.............................................................53
   b. Register combiners.......................................................54
   c. Example texture shaders.............................................55
   d. Example pixel shaders...............................................55
   e. Example nVidia............................................................55

20) Conclusion.........................................................................56

APPENDIXES

21) Appendix A: Code-example lightmapping.................................58
22) Appendix B: Optimization techniques.........................................62
   A) Textures........................................................................62
   B) Vertex Submission.......................................................63
   C) Others..........................................................................63
23) Appendix C: List of word-explanations........................................65
24) Appendix D: Sources..........................................................69
25) Appendix E: Specifications of development platform.....................71
1) Introduction

Computer-technology is still in it’s childhood. Yet it has gone through an incredible development. Going from room-filling calculators to the 8-bit Z80 home-computers, to todays gigahertz systems. An important byproduct of these computer-technologies is the market for computer-games. Parallel to the development of computer-systems, computer games have gone through an immense evolution. You can think of the first commercial success, the game Pong from 1972, for example. The success of this simple game paved the way for the almost realistic PC and console-games of today, 30 years later. The new computer systems, becoming more powerful every day, give us the possibility to produce ever more complicated and realistic computer games. This in cooperation with graphical-cards becoming more powerful all the time, and starting with the success of the first 3D-cards by 3dfx. These 3D-accelerated graphical cards made hardware-accelerated 3D-graphics possible, making a greater level of detail possible. Now, with today’s highly technological graphic-cards from nVidia and ATI, which even have an own graphical processor (GPU – graphics processing unit), one can even display almost realistical 3D scenarios in real-time.

On this screenshot you see what nVidia’s top-card for the moment (Geforce 4 Ti4600 – april 2002) can render in realtime. The branches of the trees are waving in the wind, the birds are flying, and the water is moving calmly.
One of the most, if not the most important aspects for producing realistic looking 3D is correct lighting. Without correct lighting there is no realism. As everywhere in the industry there’s being specialized more and more. The computer industry has started working more with specialists today. An important specialisation today is the one of programmer for lighting-techniques. Especially now, with the current generation of hardware-programmable cards (GeForce 3&4 series from nVidia, radeon 8500 from ATI).

My work is about these diverse techniques for correct lighting. I want to do an investigation of the possibilities for rendering a correct lighting fast and efficiently, using self-programmed examples.

One can clearly see the word “fast” being printed in bold-style. For real-time 3D, especially computer-games, it is extremely important that everything happens as quickly and as efficiënt as possible. That is because an enormous amount of things must happen in an as short as possible period of time. The artificial intelligence must do it’s job, the scene must be rendererd, etcetera.
2) **Tools / Libraries / Other helps**

I wish to discuss the tools and libraries that I will use for this work.

**a. OpenGL API**
In order to render in 3d, an API (Application Programming Interface) is being used. There are two important API's available. On one side, there is Microsoft’s DirectX (current version : 8.1a). On the other side, there is OpenGL, which was originally developed by SGI, and is now being maintained by the ARB (Architecture Review Board), which has, among others, nVidia, Microsoft, SGI and ATI as it’s members.

My personal preference goes out to OpenGL. The big difference between OpenGL and DirectX is that OpenGL is portable to a lot of platforms. It also an extension-system for easy expandability. DirectX is specific for MS Windows, and is only being updated ever now and then. Another difference is the way you should work with both API's. OpenGL has a much less steep learning-curve than DirectX. The most important site for OpenGL: [http://www.opengl.org](http://www.opengl.org)

OpenGL’s current version is 1.3. You can find the specifications of OpenGL 1.3 at [http://opengl.org/developers/documentation/version1_3/glspec13.pdf](http://opengl.org/developers/documentation/version1_3/glspec13.pdf). At the moment, especially 3Dlabs is very busy working on a version 2.0 of OpenGL, which will contain it’s own shading language. ([http://www.3dlabs.com/support/developer/ogl2/index.htm](http://www.3dlabs.com/support/developer/ogl2/index.htm)).

This shading language is needed for the current generation of hardware-programmable cards. More about that later on. For now, you should know that version 1.3 is still using extensions to use this programmable hardware.

**b. Magic Software Library**
The Magic software library is a multi-purpose toolkit, which is freely available. It has an enormous amount of vector, 3D and other graphical/mathematical operations available. The library might not be speed-optimized, but the source is open, so it gives you the freedom to optimize it yourself if you want.

You can download the library at [http://www.magic-software.com](http://www.magic-software.com).

I used this toolkit for example with my investigation of lightmaps, for which several vector-operations are needed.

**c. DevIL Image Library**
DevIL, which was called OpenIL earlier, but renamed to DevIL on SGI’s request, is a multi-purpose manipulation-toolkit for images. With the “ilut” part of this toolkit, you can do several image-manipulations specific for OpenGL and/or DirectX. I use this toolkit amongst others to load textures and to create new textures on-the-fly. This toolkit can be found at [http://openil.sourceforge.net](http://openil.sourceforge.net).
d. KRIS Library
Besides the libraries mentioned above, I also use one of my own libraries, which is still work-in-progress, but quite usable already. I use the library mostly for the use of 3D-models.

An example:
I create a mesh-model in 3D studio Max, but to use this model in my own programs, I must write my own routine to load the model. My library lets me convert .ase models to my own .cqm format. This cqm format is a very easy in use ASCII-format, which stores vertexes, vertex-colors, texture-coördinates, normals and textures. In short: everything I need to effectively use my models in my own programs.

Besides that, my library also contains some more optimized vector and 3D operations.
This library and the other programs used with this work are available from my site: [http://www.blue-print.be.tf](http://www.blue-print.be.tf)

e. extGL
extGL is a library which easies the initialisation and use of extensions. The library is available from: [http://www.uni-karlsruhe.de/~uli2/index.html](http://www.uni-karlsruhe.de/~uli2/index.html)
With only one function-call, you can initialize all extensions supported by your video-card. The library not complete. New extensions for OpenGL are being developed every month. But it is easily extendible.
3) **OpenGL Extensions**

a. **Lead-in**  
In contradiction to Microsoft’s DirectX, OpenGL is easily extendible. OpenGL’s functionality can be expanded by extensions. This explains why OpenGL “always” supports the newest graphical tricks first.

b. **How are extensions being documented?**  
OpenGL extensions are defined by their specification. This specification is typically an ASCII textfile. The complete list of extensions can be found on SGI’s website, the original developer of OpenGL: [http://oss.sgi.com/projects/ogl-sample/registry/](http://oss.sgi.com/projects/ogl-sample/registry/)  
These specifications aren’t written as tutorials, but by the way they are written, the programmer can clearly find out how an extension should be used.

c. **The use of extensions**  
We are speaking of extensions here, so let me clarify just a bit more that this functionality is not in the OpenGL library by default. Because of that, to use an extension one has to initialize some parameters and functions, via a function-pointer.

As an example for this initialization I’ll use the EXT_point_parameter extension. In windows it goes as follows:

**Step 1:** First we declare the function-prototypes, which are the same as the extension’s entry:

```c
typedef void (APIENTRY * PFNGLPOINTPARAMETERFEXTPROC)(GLenum pname, GLfloat param);
typedef void (APIENTRY * PFNGLPOINTPARAMETERFVEXTPROC)(GLenum pname, const GLfloat *params);
#endif
```

**Step 2:** We declare some global variables with the type of these function prototypes:

```c
#ifdef _WIN32 // Only in windows - another way for another OS
PFNGLPOINTPARAMETERFEXTPROC glPointParameterfEXT;
PFNGLPOINTPARAMETERFVEXTPROC glPointParameterfvEXT;
#endif
```
Step 3: First we check if the extension is supported by our driver and our graphical card with the function isExtensionSupported. You should write this isExtensionSupported function yourself, which can easily be done as follows:

```c
// This function returns true if the extension is there.
// With true I mean any value different from 0.
int isExtensionSupported(const char *extstring)
{
    char *s = (char*)glGetString(GL_EXTENSIONS); // Get our extension-string
    char *temp = strstr(s, extstring); // Is our extension a string?
    return temp!=NULL; // Return false.
}
```

Step 4: We first check if the extension is actually there. Then we use the wglGetProcAddress function to assign the address of the functions in the OpenGL driver to these function-variables.

```c
// Check if the extension is there.
int hasPointParams = isExtensionSupported("GL_EXT_point_parameters");

// Initialize the extension-functions.
#ifdef _WIN32
if (hasPointParams) {
    glPointParameterfEXT = (PFNGLPOINTPARAMETERFEXTPROC) wglGetProcAddress("glPointParameterfEXT");
    glPointParameterfvEXT = (PFNGLPOINTPARAMETERFVEXTPROC) wglGetProcAddress("glPointParameterfvEXT");
}
#endif
```

Step 5: Bingo, now we can use the extension-functions freely in our application. An example:

```c
if (hasPointParams) // If the extension is available
{
    static GLfloat quadratic[3] = { 0.25, 0.0, 1/60.0 };  
    glPointParameterfEXT(GL_DISTANCE_ATTENUATION_EXT, quadratic);  
    glPointParameterfEXT(GL_POINT_FADE_THRESHOLD_SIZE_EXT, 1.0);  
}
RenderScene();
```

Important notice:
- The extension-functions are only valid for the OpenGL rendering-context you initialized it with. So if you work with more than one rendering-context, then you MUST use the function-address returned by wglGetProcAddress, or else you can get problems. In short, you need to initialize your function with wglGetProcAddress for every rendering-context.
- Because of the diversity of graphical cards, it is possible that some of the target-audience doesn't have some extensions available. In that case you'll have to use another way to get the desired effect.
4) Standard OpenGL lighting

a. GL_LIGHTING and GL_LIGHTx

By default, OpenGL gives you the possibility to use 8 light-sources, from GL_LIGHT0 to GL_LIGHT7, which can be hardware-accelerated. Of course, there are extensions to increase this amount. The trick is to “recycle” lights that are not visible. Of course, most times there aren’t 8 visible lights in a scenario. In my area of interrest, game-development and real-time rendering, most times some tricks are being used to view a light, and much less the GL_LIGHTs. With tricks I mean lightmaps, vertex-lighting, etc. More about that later.

To use lighting in OpenGL you should turn it on first. This can be done with one simple command:

```c
 glEnable(GL_LIGHTING);
```

Turning lighting off again is just as simple:

```c
 glDisable(GL_LIGHTING);
```

When lighting is activated, you can also switch on or off a GL_LIGHTx (x=0 to 7).

```c
 glEnable(GL_LIGHT0);
```

I programmed a simple demo to show the difference between a lit and an unlit scene. In this case I used a simple sphere. Some screenshots:

On the left side, you see the sphere rendered with lighting turned off. On the right side, a light-blue ambiënt lighting is applied. So an un-lighted scene doesn’t look 3-dimensional at all.

To define a GL_LIGHT, you need to pass some parameters to OpenGL to define how the light should be. In the example above, that’s done in the following way:
So that’s a fairly simple example. First an array is defined which determines the color of the light, which is light-blue here. Then GL_LIGHT0 is specified. It must be an ambiënt light (GL_AMBIENT), and have a light-blue color. Then GL_LIGHT0 is turned on, and of course OpenGL lighting (GL_LIGHTING).

b. Normals
Another important detail is the normal of your geometry. A normal defines how lighting will react to your geometry, that is, how the light will be rendered. The normal can be specified per face or per vertex. In the following example I’ll try to clarify the difference:

On the left screenshot, so called flat shading is being used. Here every vertex of a quad or triangle gets an equal normalvector. On the right screenshots, the normals are taken as an average to the shared vertices of the 2 quads. So the corner of the 2 quads looks much more rounded, and lighting reacts in a totally different way.

You can easily specify a normal in OpenGL with the following function:

```
const GLfloat norm[] = {1.0, 0.0, 0.0};
glnormalfv(norm);
```

So you’re actually only defining a direction-vector.

You can use this to show meshes in a better way. Here are some renderings from 3D studio Max as an example:
On the left screenshot, flat shading was used. On the right one smooth shading. So it can be very useful to use smooth shading. You can think of 3D computer games here. To get the detail of the right rendering you would need a mesh with a way higher resolution with flat shading. With smooth shading, you can get the same effect with meshes with a much lower resolution. This is only done by changing normals, and therefore the influence of light.

c. Ambient, Diffuse, Specular etc.
There are of course more possibilities than to specify lighting in OpenGL. Besides ambient there is among others diffuse, specular, shininess, and of course the position of the light-source. The last one of course is only valid for non-ambient lighting. I programmed an example with multiple parameters for a positional light.

A screenshot:

There the position of the light is right to the teapot. As parameters I passed:

```c
glEnable(GL_LIGHTING);
GLfloat LightAmbient[] = { 0.0f, 0.0f, 1.0f, 1.0f };
GLfloat LightDiffuse[] = { 1.0f, 0.0f, 0.0f, 1.0f };
GLfloat LightSpecular[] = { 0.0f, 1.0f, 0.0f, 1.0f };
GLfloat LightPosition[] = { 10.0f, 0.0f, 5.0f, 1.0f };
gLightfv(GL_LIGHT0, GL_AMBIENT, LightAmbient);
gLightfv(GL_LIGHT0, GL_DIFFUSE, LightDiffuse);
gLightfv(GL_LIGHT0, GL_SPECULAR, LightSpecular);
gLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
gEnable(GL_LIGHT0);
```

So there is blue as an ambient light, red as a diffuse light, and green as a specular light. Spotlights can also be defined, for example as follows:
GLfloat spot_direction[] = { -2, 0, -3};
gllightf(GL_LIGHT0, GL_SPOT_CUTOFF, 10.0f);
gllightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);

(This code has to be added to the last code-example)
This is a spot with the same light-position as in the last example, with a cut-off of 10° (that’s a corned of 20°), with as target the direction via vector (-2, 0, -3).

But there are better ways to get way better spotlight effects. For example projected textures, which will be discussed later on.

d. Materials
Besides specifying ligh-sources, you can also apply materials to a 3D mode, which will have an influence on the result that lighting will have on an object.
This “material” that you apply to your model will determine how your model will react to light and in this way clarify what material your model is made from. For example a metal shininess (much reflection), or a block of wood (not very much reflection).
For this too I programmed a small demo, from which a screenshot is shown here.

For the material of this example I passed parameters for ambiënt, diffuse, specular, shininess and emission. The most clear effect we see here is the shininess (GL_SHININESS), the teapot seems to have a yellow shininess over it.
5) **Vertex Color lighting**

Vertex color lighting is a fairly old, but simple and very fast lighting technique, which is still being used very much.

To every vertex of a model (or full scene) you can apply a color. Take as an example the triangle on the left. To every one of the three vertices of the triangle, a color was applied. What if, for example, you would dynamically apply a grayscale to each of the vertices? Then it would be possible to display a certain lighting and shadow. The only thing you have to calculate is how far your light is away from the vertex, and the effect that the light has on that position.

That is what’s happening on the left screenshot. The down-left corner gets the smallest amount of light and has a much darker grayscale than the vertices that receive more light from the source. This can also easily be done with intensity-values of a color, as shown with the screenshot on the right.

I programmed a more specific example to show the pro’s and con’s of vertex-lighting.

Here are two screenshots:

![Screenshot of yellow sphere representing light-source and white spheres on rendered quad](image)

The yellow sphere represents the light-source. The white spheres on the second screenshots are the vertices of the rendered quad.
Good, a nice lighting you’d say. Or isn’t it?
The problem with vertex-color lighting is that you can only light per vertex, as the name indicates. To get really beautiful lighting via this technique you need to work with high-resolution models (that is: meshes with a lot of triangles or quads). I wish to illustrate this with the above screenshots. You can clearly see there are only 4 vertexes. In the middle of the quad, there is none. When the light goes over the middle of the quad, it stays dark, which isn’t very realistic of course. For static lighting, the use of vertex color lighting is a very good technique. You can say that this technique has it’s own name against it, because you can only light per vertex. This technique clearly can be used in a certains amount for lighting on graphical cards that haven’t got multitexturing available, an extension that is needed for lightmaps. More about lightmaps later on.

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very fast</td>
<td>Only usable for static meshes when low-poly.</td>
</tr>
<tr>
<td>Good looking effect</td>
<td>High mesh-resolution required</td>
</tr>
<tr>
<td></td>
<td>Only per vertex.</td>
</tr>
</tbody>
</table>
Blending

Blending is a standard OpenGL functionality that makes it possible to make 3D geometric figures transparent in diverse ways, dependent on the given parameters.

Blending can be turned on or off as follows:

```c
 glEnable(GL_BLEND); // turn on
glDisable(GL_BLEND); // turn off
```

So that’s just like with GL_LIGHTs. Blending makes it easy to make transparent figures. I programmed a simple example, from which a screenshot is shown here:

3 texture-mapped quads are rendered here, from which the frontmost and the backmost are made transparent with blending.

The blending happens on this screenshot via the following function:

```c
 glBlendFunc(GL_SRC_ALPHA, GL_ONE); // Alpha as source and GL_ONE as target.
```

In RGBA mode, the source RGBA values can be blended with the RGBA values that are already in the framebuffer. This is what you can define with glBlendFunc.

Function definition:

```c
void glBlendFunc(GLenum sfactor, GLenum dfactor);
```
The `sfactor` parameter specifies which of the nine source-parameters is being used for source-values. These nine possibilities are:

```
GL_ZERO, GL_ONE, GL_DST_COLOR,
GL_ONE_MINUS_DST_COLOR, GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA,
GL_ONE_MINUS_DST_ALPHA, en
GL_SRC_ALPHA_SATURATE.
```

The `dfactor` parameter specifies which of the eight possible methods is being used to determine the target colors. These eight possibilities are:

```
GL_ZERO, GL_ONE, GL_SRC_COLOR,
GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, en
GL_ONE_MINUS_DST_ALPHA.
```

On the following link is a full specification of blending via `glBlendFunc`:

```
```

Besides it’s use for transparency, you can, for example, also use blending for masking. This is a technique used quite often in for example user interfaces in games.

Here a good example of masking:

![Masking Example](http://nehe.gamedev.net)

With masking, a certain part of a texture is left away via a mask. This screenshot came from a NeHe demo/tutorial:

```
http://nehe.gamedev.net
```
6) Multitexturing

Multitexturing is a fairly recent technique to apply multiple textures on top of each other in hardware. These textures can then have an influence on textures lying below via certain parameters. Multitexturing isn’t supported by default in OpenGL, and has to be installed as an extension.

Multitexturing is especially used for lightmapping, and also to apply a detail-texture to a texturemap lying beneath.

On this screenshot from Quake2, the follow-up of ID-software’s revolutionary game Quake, you can clearly see what lightmaps are. Later on in this work is a full chapter about lightmapping.

On this screenshot of the Demeter terrain-engine 3.11 http://demeter.sourceforge.net, you can see an example of detail textures. Here a global texture is used for the whole terrain, with a detail-texture to give the terrain a structure.

I programmed a small demo myself to demonstrate detail textures:

In this demo, 4 quads are being rendered which contain 1 global texture and 1 detail texture via multitexturing. This technique gives you the possibility to give, for example, a more detailed look to a terrain.
Here are some close-ups to clarify the difference:

On the left is the terrain without detail-textures, on the right the terrain with detail textures. The terrain on the right looks clearly more realistic than the one on the left, an effect that can easily be achieved via multitexturing with a very small loss of performance.

Because multitexturing works via an extension, it needs to be initialized:

```c
//Initialize the needed parameters:
#define GL_TEXTURE0_ARB 0x84C0
#define GL_TEXTURE1_ARB 0x84C1

//Initialize the function prototypes:
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB = 0;
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = 0;
PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB = 0;

//Assignment of the function pointers:
void InitMultiTexture(void)
{
    glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC) wglGetProcAddress("glMultiTexCoord2fARB");
    glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC) wglGetProcAddress("glActiveTextureARB");
    glClientActiveTextureARB = (PFNGLCLIENTACTIVETEXTUREARBPROC) wglGetProcAddress("glClientActiveTextureARB");
}
```

Here the possibility for two texture-layers is created: GL_TEXTURE0_ARB and GL_TEXTURE1_ARB. The amount of texture-layers you can use depends on the possibilities of your graphical hardware. The texture-layer with the lowest number is the basic layer, the downmost one. To apply a texture to one of the multitexture-layers, it needs to be bound to it. Here an example:

```c
// First texture-unit
glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_2D, Texture0);
glEnable(GL_TEXTURE_2D);

// Second texture-unit
glActiveTextureARB(GL_TEXTURE1_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glBindTexture(GL_TEXTURE_2D, Texture1);
glEnable(GL_TEXTURE_2D);
```

With the second texture-unit, in this case the topmost texture layer is specified how it will be visible on top of the first layer. In this case, I use GL_MODULATE, where the second texture-unit is being mixed with the first texture-unit. Besides GL_MODULATE there is also the possibility to use GL_REPLACE, where the second texture-
unit replaces the first one. Also you need to enable texturing for each texture via glEnable(GL_TEXTURE_2D). After applying textures to their layer, you can texture your 3D geometry as follows:

```gl
glBegin(GL_QUADS);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.5f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 1.0f);
glVertex3f(-3.0f, -3.0f, -4.0f);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.5f, 0.5f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 1.0f);
glVertex3f( 0.0f, -3.0f, -4.0f);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.5f, 0.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 0.0f);
glVertex3f( 0.0f, 0.0f, -4.0f);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
glVertex3f(-3.0f, 0.0f, -4.0f);
glEnd();
```

This is the code-part that represents the down-left quad of my detail-texture demo. So you can pass texture-coördinates for each layer, in this case, ¼ of the base-texture and the whole of the second texture-unit is being placed on the quad.

After the rendering-code, you need to shut down the multitexturing-system in a symetrical way:

```gl
glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_2D);
glActiveTextureARB(GL_TEXTURE0_ARB);
glDisable(GL_TEXTURE_2D);
```

Thus, start with the last defined texture unit, then the first. If you don’t do it this way, the rendering process wil not be done correctly.
7) **Lightmapping**

a. **Theory**
The name says it all: This technique maps textures with lighting-values as a second texture to get a statical light-effect. The principle of lightmapping is fairly simple. Take for example the following figures:

![Lightmapping examples](image)

Lightmapping is most times applied via multitexturing. I say most times, because there is also something like dynamic lightmaps, which can be easily shown with blending. So there are 2 textures being combined. On the picture above, these are the first 2 textures. On the third picture you can see the final effect of the lightmapping. The achieved effect immediately makes it look better, even on this simple quad.

Statical lightmaps, via multitexturing, are now the most used technique to render high-quality light efficiently in real-time, especially in 3D games this technique is being used a lot. I programmed a demo which calculates a lightmap for each triangle of a scene, and maps it onto the geometry. I programmed this demo to illustrate the calculation of lightmaps. It does not contain multitexturing.

Here is a screenshot:
For this scene, here the inside of a house, a texture was calculated for every triangle. In the OpenGL application this screenshot came from there are no light sources. The light-effects are all made with lightmaps. The dark sphere in the middle of the room is a light-source. You can see that the calculation of light-values isn’t done 100% correctly, but this demo is only for the illustration of lightmapping. Every calculated lightmap is begin mapped on the corresponding triangle. What doesn’t happen here, but is recommended of course, is the use of a certain amount of raytracing. Of course not in the strictly raytracing, but only to determine if a triangle recyves light or not. In other words : we must determine if there is no occluding geometry, that blocks the light of a certain triangle. What does happen in my demo is that it doesn’t light backfacing triangles, thus triangles that are with their “backs” to a light-source.

The calculation of lightmaps of course doen’t happen in real-time. Lightmaps are pre-rendered. For the amount of calculations that are nessecary even a gigahertz isn’t fast enough, so that lightmaps are mostly only being used as a statical light-effect. A program specifically developed for that purpose will analyse the data of a scene and generate the lightmaps for it, like in my demo. These lightmaps are then being stored as one or more combined images. A good example of this :

Here you see a combined texture with lightmaps for the scene of the picture above. For each triangle, a square texture is assigned with lighting values. This way, you loose a lot of space of course. It is better to only use one triangle per triangle. This way you are able to order lightmaps way more efficiënt in this texture.
b. Code example
My next demo illustrates how the calculation of a lightmap happens.
Firstly a screenshot:

In this demo, a lightmap is being calculated in real-time for only one triangle. Because it is only a low-res map, and only one map needs to be calculated, it can be done in real-time. A full scene wouldn’t be possible in real-time.

The yellow sphere on this screenshot is a light-source, the triangle is the rendered and lightmapped “scene”. The square shows the texture of the lightmap. The square is not lightmapped, it is only texture-mapped with the lightmap of the triangle.

Look at Appendix A for the code.

c. How does this calculation work?
On the triangle in the screenshot of the foregoing demo, you can precisely see a blue line. This is the normal of the triangle. In this method used by me, I look first at the vector that the normal has with the axis of the coordinate-system (X,Y,Z). This vector I use to determine on which surface you will project the triangle. To be more precise: I’m seeking the smallest vector, in order to determine on which plane the projection will preserve the most details of the triangle. I’ll illustrate this with a serie of images. These aren’t a 100% exact, but they’ll make clear what I mean.

Here is a drawing that I’ve made. You can already see with the bare eye that the smallest vector between the normal(N) and the X-axis is the smallest.
So we shall choose to project the triangle to the YZ-plane, where the triangle will retain the biggest surface after the projection.
After projection I've got a triangle lying in the YZ-plane. This triangle will enable me to calculate a lightmap easier for this triangle.

After projection of the triangle I determine the square that fits around it. With this square, I determine the new texture-coordinates for the lightmap-texture.

You determine which resolution you'll take for your lightmap-texture, which will most times be something like 64x64 pixels. Using the earlier mentioned square, you create an array of lumels. This is an array of pixels, which will contain the calculated light-values. This array is, of course, the size of your texture, which in this case is 64x64. Using the height and width of your square, you can determine the lumels position in the YZ-plane.

Now you determine the plane in which the original triangle lies. When you have this plane, you project the 64x64 lumels back to this plane in 3D. Now you have the lumels in their 3D-position.

Via the 3D-position of a lumel and the lighting-parameters, for example light-strength, color, decay and of course position, you can determine the color-value for each lumel. This can among others be done with the formula on the left.

\[ b = \frac{\lambda \cdot \cos \theta}{L} \]

\( \lambda \) is the brightness

The calculated values of the lumels are then put in a 64x64 pixel texture, which is the final lightmap. I do this using the DevIL library.

From all calculated lightmaps, I create one combined texture, which contains all lightmaps of a model or scene. Of course, a data-file with texture-coordinates is also generated.

With another, also frequently used method of lightmapping, you also look at the smallest vector with one of the axis of the system, but when this vector is determined, the triangle is being rotated, in such a way that it will lie parallel with a plane of the system. This way, you can clearly determine the highest resolution for a lightmap. The rest of this technique is almost identical to the one I used.
d. Dynamic lightmaps via blending
Because it is nearly impossible to calculate even low-res lightmaps in real-time to get dynamic lightmaps, we must look further for more possibilities to do this. An easy way to do real-time lightmaps, is via blending.

Here, a separate 3D geometric figure is used, on which a lightmap is mapped as a texture. This figure, in my example a quad, is being rendered on a certain height above the scene, and blended.

Here are 2 screenshots of the application:

![Screenshot 1](image1.png)

Here is the scene with it’s 2 dynamic lightmaps. As you can see on the next screenshot, the dynamic lightmaps are 2 quads on which a texture is mapped.

![Screenshot 2](image2.png)

With the simple blend-command,
```cpp
glBlendFunc(GL_DST_COLOR, GL_ONE);
```

2 quads are made transparent, and you get the lightmap-effect. Using this method, you can create dynamic lightmaps with a dynamic light-strenght, for example by applying a color-intensity on the rendered squares.

In my next application, the light-intensity changes depending on the height of the light-source.

![Screenshot 3](image3.png)
This can be done simply with vertex-colors on the lightmap-quad, or via the glColor function as in my demo. Via glColor I apply intensity-values to the next quad. When the light-source is too far away from the floor, I make the lightmap darker, and when it’s closing in, I make it brighter.

It’s best to turn of depth-testing to render the lightmap-quad, because else Z-fighting can occur on hardware with a less precise Z-buffer.

Why then use no lightmaps via blending instead of the statical maps used with multitexturing? A great con of the blended lightmaps is the fact that you must render extra 3D-geometry, and because of that, your rendering-speed, most times expressed in Frames Per Seconds (FPS), will go down. The static lightmaps of multitexturing on the other hand will cause almost no decrease in the amount of FPS.

The best effect can be obtained combining both techniques. Static lightmaps for static lighting, and dynamic lightmaps via blending for dynamic lighting.

e. **Comparison and conclusion**

<table>
<thead>
<tr>
<th></th>
<th>Via multitexturing</th>
<th>Via blending</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very fast</td>
<td>Slower because of extra rendering</td>
<td></td>
</tr>
<tr>
<td>Only for static lighting</td>
<td>Dynamic lighting</td>
<td></td>
</tr>
<tr>
<td>Almost none to no calculation</td>
<td>More calculations</td>
<td></td>
</tr>
<tr>
<td>High-res lighting</td>
<td>Low-res rendering</td>
<td></td>
</tr>
<tr>
<td>Render less geometry</td>
<td>Extra geometry to render</td>
<td></td>
</tr>
</tbody>
</table>

**Conclusion**

A combination of both is best for getting the best results.
8) **OpenGL fog**

With the `glFog` function, you can create a mist-like effect. This fog can be used. In 3D computer games, distance-fog is used most times. This is a technique lying closely to another technique, which is Level Of Detail (LOD). 3D engines build objects that are further away from the camera most times from meshes and low-res textures. This can extremely increase rendering-speed, because there is less to render. Because these objects are farther away from the spectator, the difference most times isn’t noticeable. These high or low-res meshes “suddenly” visualize far away, at the point of the distance-plane. To work away the sudden appearance of meshes, distance-fog can be used. It will cover the sudden appearance of the objects.

This demo uses distance-fog to work away the sudden appearance of objects from behind the distance-plane.

The fact that a color can be assigned to the fog can be used very creatively. It is best to choose a fog-color that matches the color of your background (horizon). In this demo, I chose blue for both.

The use of mist is very much like the use of `GL_LIGHT`s. Here is the code-example from my demo:

```c

```
glEnable(GL_FOG);
glFogf(GL_FOG_MODE, GL_EXP);
glFogf(GL_FOG_START, 35.0f);
glFogf(GL_FOG_END, 40.0f);
glFogf(GL_FOG_DENSITY, 0.0755);
glHint(GL_FOG, GL_NICEST);
```

In this code, OpenGL fog is being turned on first. Then the color is defined. I choose the `GL_EXP` type of fog, I set where the fog begins and where it ends, and it’s density. Lastly, I tell OpenGL to render the fog as beautiful as it can.
9) Rendering to texture

Rendering to texture is a fairly simple technique, in which you copy a rendered scene to a texture. You can then use this texture for example mirror-effects.

A small demo, demonstrating the basic principles:

In this demo, a scene with 5 triangles is rendered to a texture. Then it is mapped on a quad which is rotating in 3D.

How does this work?
First, an empty texture-object is created, which will contain the scene rendered to texture. Also, the size of the texture is defined:

```c
#define T_SIZE 256
GLuint texture;
```

After this, the texture is initialized with the following function:

```c
void InitTexture(void)
{
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0, T_SIZE, T_SIZE, 0);
}
```

The render-function looks as follows:

```c
void Render(void)
{
    UpdateTexture();
    glViewport(0, 0, Settings.scrwidth , Settings.scrheight );
    glClearColor(0,0,1,1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    //Main Rendering Function
    DrawScene();
    //FPS
    fontDrawString(10, Settings.scrheight-20, "FPS: %0.1f", UpdateFPS());
    //screensettings
    fontDrawString (10, 10, "%ix%i", Settings.scrwidth, Settings.scrheight);
    glFlush();
    glutSwapBuffers();
}
```
So first the texture is created -in the UpdateTexture function-, which contains the scene with the 5 triangles. This UpdateTexture function looks as follows:

```c
void UpdateTexture(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glViewport(0, 0, T_SIZE, T_SIZE);
    DrawTriangles();
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, T_SIZE, T_SIZE);
}
```

So you start with a black screen, on which the 5 triangles are being rendered in 3D. The rendered data is then being copied to the “texture”-texture. These rendered triangles however aren’t visible when you run the application. This is because just after the call of the UpdateTexture function, the buffer is being cleared with a blue background, and we didn’t do an OpenGL call to swap buffers yet. Now the quad is being rendered with the rendered texture mapped on to it. This times, the buffers are swapped, so that the scene with the quad is being rendered to the screen.

This technique can clearly be use for all sorts of reflection and special effects.

An example of rendering to texture, used as an effect:

![An amount of quads is being rendered, with a texturemap of those same quads on it, but then rendered in another way. To make the effect more beautiful, the final quads where rendered transparant via blending.](image)

Demo from Exsanity (Phil Freeman).
I programmed one other demo to demonstrate the mirror-effect:

Here a second camera was used, which stands symmetrically with the viewer’s camera, but then mirrored in the X-axis. From the view of the second mirrored camera, the texture is created, and then mapped onto the mirror.

10) Texture Projection

Texture-projection is a technique to project textures onto a scene, like a dia-projector. I start with a screenshot of a demo that I programmed to illustrate this technique:

You can see a small scene, made out of 2 walls, a floor and a sphere. On this scene, a logo is projected with texture-projection.

The mapping of the projected texture is done via texgen as follows:

```c
float PS[]= {1, 0, 0, 0};
float PT[]= {0, 1, 0, 0};
float PR[]= {0, 0, 1, 0};
float PQ[]= {0, 0, 0, 1};
glTexGenfv(GL_S, GL_EYE_PLANE, PS);
glTexGenfv(GL_T, GL_EYE_PLANE, PT);
glTexGenfv(GL_R, GL_EYE_PLANE, PR);
glTexGenfv(GL_Q, GL_EYE_PLANE, PQ);
```
The render function looks like this:

```c
void DrawGL(void)
{
  glClearColor(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  glRotatef(20, 1, 0, 0);
  glRotatef(-20, 0, 1, 0);
  glTranslatef(0, 0, -50);
  glTexGenfv(GL_S, GL_EYE_PLANE, PS);
  glTexGenfv(GL_T, GL_EYE_PLANE, PT);
  glTexGenfv(GL_R, GL_EYE_PLANE, PR);
  glTexGenfv(GL_Q, GL_EYE_PLANE, PQ);

  // FIRST PASS
  // First pass: draw the scene with some standard diffuse lighting.
  glBlendFunc(GL_ONE, GL_ZERO);
  glDisable(GL_TEXTURE_GEN_S);
  glDisable(GL_TEXTURE_GEN_T);
  glDisable(GL_TEXTURE_GEN_R);
  glDisable(GL_TEXTURE_GEN_Q);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  // We don't want the texture projection matrix to affect this!
  glMatrixMode(GL_TEXTURE);
  glPushMatrix();
  glLoadIdentity();
  glMatrixMode(GL_MODELVIEW);
  DrawFloor(floorTex);
  DrawWalls(wallTex);
  DrawSphere(whiteTex);
  glBindTexture(GL_TEXTURE_2D, projTex);
  glMatrixMode(GL_TEXTURE);
  glPopMatrix();
  glMatrixMode(GL_MODELVIEW);
  glDisable(GL_LIGHTING);
  glEnable(GL_TEXTURE_GEN_S);
  glEnable(GL_TEXTURE_GEN_T);
  glEnable(GL_TEXTURE_GEN_R);
  glEnable(GL_TEXTURE_GEN_Q);

  // SECOND PASS
  glBlendFunc(GL_DST_COLOR, GL_ONE);
  // spotlight
  glMatrixMode(GL_TEXTURE);
  glPopMatrix();
  glPushMatrix();
  // Set the projector's position and orientation:
  glRotatef(cos(a*PI/180)*30, 0, 1, 0);
  glRotatef(-30, 0, 1, 0);
  glRotatef(25, 1, 0, 0);
  glTranslatef(0, 0, -20);
  glMatrixMode(GL_MODELVIEW);
  DrawFloor(projTex);
  DrawWalls(projTex);
  DrawSphere(projTex);
}
```

So in the first rendering-pass, the scene is being rendered with its original textures, after which a second rendering pass is made, that gives the scene the projected texture as a texture-map. Also, the coordinates of the projected texture are generated via texgen.
You can use this method also for spotlights. These have a way better quality than the spotlights you define with standard OpenGL lighting. Of course, a spotlight does cast a strongly lined-out shadow, which you can generate via projected shadows or shadow-mapping.

11) **Bump Mapping**

Bumpmapping is a technique that gives the illusion of non-flat surfaces without having to render extra geometry. A relatively quick method to render way more realistic objects. Bumpmapping is done by combining a texture and a normalmap. This normalmap can be generated with a photoshop plugin, which can be found on the nVidia webpage:


Of course, I created a small demo for this, from which I have a screenshot over here:

This demo uses the so called dot3 bumpmapping. This means that NxL is calculated per pixel. We do this by storing the N vector in the dot3 texture (the normalmap). Then we calculate L per vertex, and store it as a vertex color. These colors are then automatically interpolated by OpenGL. NxL is calculated by the texture combiner, and modulated with the base texture. Note: to do this, the L vector has to be rotated in tangent space, so that it will lie in the same space as N.
The normalmap for the scene above looks like this:

This is the normalmap containing the N-vectors. Using this map, the illusion of rough surface is rendered. Every pixel of this map is seen as a normal. As I discussed before, the normal determines the way the lighting will react to the geometry. With this normalmap, we can create the illusion that every tile has a thickness and a rough surface. There are of course other forms of bumpmapping than dot3. Later on, at pixel shaders, I've got another example of that.

I'll quickly discuss the code:

With glTexEnv we define how the textures are going to be combined with each other:

```
// UNIT 0
// find dot product of N (stored in the texture map) and L (stored
// as the PRIMARY_COLOR).
glActiveTexture(GL_TEXTURE0_ARB);
glBindTexture (GL_TEXTURE_2D, dot3tex[texnum]);
glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_DOT3_RGB);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE1_RGB, GL_PRIMARY_COLOR);
```

```
// UNIT 1
// modulate the base texture by N.L
glActiveTexture(GL_TEXTURE1_ARB);
glBindTexture (GL_TEXTURE_2D, basetex[texnum]);
glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_MODULATE);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_PREVIOUS);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE1_RGB, GL_TEXTURE);
```

As you can see here, we work with multitexturing. Every time, the needed texture is readied, and the way the textures react to each other is defined.

After this some other calculations are done to calculate the right color-values per vertex. These are ten combined by the texture-combiner with the textures. This creates the bumpmapping effect.
12) **Plane Projected Shadows**

Plane projected shadows is a technique that uses 2 rendering passes to project a mesh unlit on a plane, so that it looks like a shadow.

A screenshot of a demo I made for this:

In the first pass, the scene is being rendered (light-source, torus and plane). Then in the second pass, the shadow is rendered via plane-projection.

Firstly we must create an array in which we put the data for the shadow projection matrix. With this matrix we'll project the scene on the plane later on, and create the shadow this way.

```
GLfloat shadowMat[4][4];
```

We must also define our light-position:

```
GLfloat lightPos[] = { -20.0f, 50.0f, 0.0f, 0.0f };
```

We need this position to calculate the projection matrix. We also need to define three points lying on the projection plane.

```
GLfloat points[3][3] = {{ -50,-50, 50 }, { 50,-50, 50 },{ 50,-50,-50 }};
```

With this data, we can determine our shadow projection matrix, with the following function:

```cpp
void MakeShadowMatrix(GLfloat points[3][3], GLfloat lightPos[4], GLfloat destMat[4][4])
{
    GLfloat planeCoeff[4];
    GLfloat dot;

    // Find the plane equation coefficients
    // Find the first three coefficients the same way we find a normal
    calcNormal(points, planeCoeff);

    // Find the last coefficient by back substitutions
    planeCoeff[3] = -(planeCoeff[0]*points[2][0]) + (planeCoeff[1]*points[2][1]) + (planeCoeff[2]*points[2][2]);

    // Dot product of plane and light position

    // Now do the projection
    // First column
    destMat[0][0] = dot - lightPos[0] * planeCoeff[0];
    destMat[1][0] = 0.0f - lightPos[0] * planeCoeff[1];
    ```
I did not create this function myself, but it is available on the cd-rom with the book OpenGL Superbible (Waite Group Press). What this function does is calculate via which matrix we can project a 3D-scene to a plane.

We firstly define this projection-matrix, before we start to render, as follows:

```c
MakeShadowMatrix(points, lightPos, shadowMat);
```

If you look at the function above, you see that the matrix is being stored in the shadowMat array.

Let’s go to the render function:

```c
void Render(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0, 10, -150);
    glRotatef(rot2, 0, 0.1f, 0);
    glRotatef(rot3, 0, 0, 0.1f);
    glPushMatrix();
    // Draw the scene
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    DrawShadowCasters(false);
    DrawLightPos();
    DrawPlane();
    glDisable(GL_LIGHTING);
    // Multiply with the shadow projection matrix.
    glMultMatrixf((GLfloat *)shadowMat);
    glScalef(2, 0, 2);
    DrawShadowCasters(true);
    glPopMatrix();
    glEnable(GL_DEPTH_TEST);
    //... (to be continued)
}```
glEnable(GL_LIGHTING);

glPopMatrix();

rot2+=0.1f;

// FPS
fontDrawString(10, Settings.scrheight-20, "FPS: %0.1f", UpdateFPS());
// screen settings
fontDrawString (10, 10, "%ix%i", Settings.scrwidth, Settings.scrheight);

glFlush();
glutSwapBuffers();
}

So, we firstly render the actual scene, which is lighted. Then we turn off depth-testing, because otherwise we would get geometry fighting. I mean by this that OpenGL will not be sure which of the two, the shadow or the plane, should be on top. If we turn off depth-testing, things that render last lie the most on top. Also, lighting is being turned off, because we want a black shadow. We multiply the current matrix with our shadow-projection matrix and render the scene again (that is, only the torus), this time with no lights on. This way, the shadow is rendered on the plane.

13) Projected Shadows

Another technique to show shadows is projected shadows. Here, a texture is created from the point of view of the light source. This texture is a black and white texture, containing the shadows. So instead of a flatly rendered mesh like with plane projected shadows, rendering to texture is used. A huge pro of this technique when looking at plane projected shadows, is that you can cast a shadow on a behind laying mesh. Only not on the shadow-casting mesh itself, so no self-shadowing.

Here is a screenshot of projected shadows:

The white dot is the light-source. The teapot is the shadow caster. You can clearly see that the shadow also falls on the torus (donut). If you take a good look you can see pixelblocks on the shadow, which shows that the shadow is made with a projected texture.
The texture we’re going to use to project the shadow is initialized in a way that is quite similar to the way used with “rendering to texture”:

```
// Create the shadow texture object:
glGenTextures(1, &shadow);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, shadow);
glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, 0, 0, S_SIZE, S_SIZE, 0);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

// Prepare the texture matrix for the projected texture:
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glTranslatef(0.5, 0.5, 0);
glScalef(0.5, 0.5, 1);
gluPerspective(90, 1, 1, 1000);
glPushMatrix();

// Set up texGen:
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glEnable(GL_TEXTURE_GEN_Q);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
```

Only here, we go a step further, and prepare the texture-matrix for projected textures by scaling and positioning them in the right way. After that, we also define the way the shadow texture will be mapped onto the rendered scene via texGen.

The update of the shadow texture goes like this:

```
void UpdateShadow(void)
{
    // Set a square viewport for the texture:
    glViewport(0, 0, S_SIZE, S_SIZE);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(90, 1, 1, 1000);
    glMatrixMode(GL_MODELVIEW);

    // Render the teapot black-on-white:
    glDisable(GL_LIGHTING);
    glColor3f(0, 0, 0);
    glClearColor(1, 1, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glLookAt(cam[0], cam[1], cam[2], cam[3], cam[4], cam[5], cam[6], cam[7], cam[8]);
    DrawTeapot();
    glBindTexture(GL_TEXTURE_2D, shadow);
    glCopyTexImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, S_SIZE, S_SIZE);

    // Re-enable regular lighting:
    glEnable(GL_LIGHTING);
    glColor3f(1, 1, 1);
    glClearColor(0, 0, 0, 1);
    // Reset view
}
```
As you can see, the scene is rendered here without lighting from the lights position, after which a render to texture is done.

After updating the shadow texture, the final rendering is done, like this:

```c
void Render(void)
{
    //first we update the shadow
    UpdateShadow();

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-15, 10, -100);
    glRotatef(30,1,0,0);

    // Draw the light source:
    glDisable(GL_LIGHTING);
    glBegin(GL_POINTS);
    glVertex4f(cam[0],cam[1],cam[2],1);
    glEnd();
    glEnable(GL_LIGHTING);
    DrawLine();

    // Set the TexGen planes:
    glTexGenfv(GL_S, GL_EYE_PLANE, PS);
    glTexGenfv(GL_T, GL_EYE_PLANE, PT);
    glTexGenfv(GL_R, GL_EYE_PLANE, PR);
    glTexGenfv(GL_Q, GL_EYE_PLANE, PQ);

    // Draw the shadow caster:
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, GREEN);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, WHITE);
    DrawTeapot();

    // Draw the shadow receivers using the projected shadow texture:
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, shadow);

    glMatrixMode(GL_TEXTURE);
    glPopMatrix();
    glPushMatrix();

    // Set the projector's position and orientation:
    gluLookAt(cam[0],cam[1],cam[2],cam[3],cam[4],cam[5],cam[6],cam[7],cam[8]);
    glMatrixMode(GL_MODELVIEW);

    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, RED);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, BLACK);
    DrawFloor();

    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, BLUE);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, WHITE);
    DrawDonut();
    glDisable(GL_TEXTURE_2D);
}
```

Shadow casting objects are rendered first. After that, the texture-matrix created at the initialisation is called to project the shadow-texture correctly on the objects recyving the shadow.

The big problem with projected shadows is that you need to render hyrarchically to render shadow-recyving objects correctly with their shadow. This means a lot of extra calculation. Also, self-shadowing is not possible.
14) **Shadow Mapping**

Shadow mapping is a fairly recent technique, that is, it is not until recently that it is possible to do it in hardware. This technique, like projected shadows, renders the scene from the light-source to the texture. One important difference is that we render a depth-texture and other data is rendered to the texture than with projected shadows. To generate a shadowmap texture, we need an extension this time, which is GL_SGIX_depth_texture. I initialize it via the extgl library (see also: chapter 2 – Libraries).

Here is a screenshot of a shadow mapping demo from SGI. I adapted this demo, because it was not suitable for compilation on visual C++ and windows. The original version can be found here: [http://www.sgi.com/software/opengl/advanced96/tomcat/shadowmap.c](http://www.sgi.com/software/opengl/advanced96/tomcat/shadowmap.c)

![Shadow Mapping Screenshot](image)

On this screenshot you can see my shadow mapping demo. You can clearly see that shadow mapping also does self shadowing. In this converted example from SGI, the shadow mapping is not optimal. You can clearly see errors in the mapped shadow texture. But we are talking about an absolute minimum configuration to render simple shadowmaps here.

With shadow mapping, texture coördinates are generated using texgen, which are identical to the vertex coördinates. Using the texture matrix, these coördinates are transformed back to light coördinates. The depth values are now available as texture r-coördinates.

Here is a screenshot of a shadowmapping demo from the nVidia developer's site. With this demo, you can clearly see the beautiful and correct self-shadowing. This demo does use a lot more extensions and advanced OpenGL features than my demo does however.
15) **Vertex projection**

Vertex projection works in a way very similar to plane projected shadow, with as difference that you calculate the shadow yourself here.

![Diagram of vertex projection](image)

Using this technique, you project each vertex of your object to the ground. This means that self-shadowing and shadow-recyclers are not available with this technique. This technique is very simple, and performs very well for objects with not too much vertices. The formula is as follows:

```c
void shadow_poly(Point3D p[], Point3D s[], Point3D l, num_verts) {
    for (i=0; i<num_verts; i++)
        s[i].x = p[i].x - (p[i].y / l.y) - l.x;
    s[i].z = p[i].z - (p[i].y / l.y) - l.z;
}
```

Here, s is the array with the projected vertices, P is the array with the original vertices, and I is the position of the light. So you calculate new X and Z coördinates from the shadow, as Y you use the Y-position of your plane.

So this is very simple. The back of the medal is that you can only project onto a plane. With objects that have more vertices, you’d better also use occlusion culling from the position of the light-source, in such a way that you don’t calculate too much vertices.
16) **Shadow Volumes**

Shadow volumes are an increasingly popular way to render correct shadows in realtime. In OpenGL, the shadows, determined by the shadow-volume, are calculated via the stencil buffer.

So what is a shadow volume?
I'll explain this using the following figure:

With shadow volumes, the volume of the shadow is determined. In fact, a mesh is defined that determines the shadow. On this picture, that is the pyramid, but then looking from the cube. Thus from the point where the shadow is cast.

With this volume and the stencil buffer, we can determine from the spectator’s position what needs to be lit and what needs to be unlit. A good thing to know is that the use of the stencil-buffer doesn’t cost anything extra in performance, as long as depth testing is turned on.

A consequence of this technique is that when you use high-res meshes, a lot of calculations need to be done to determine the shadow volume, which causes an enormous performance drop. For the realtime rendering of games this will most times not be a big problem, because in games the geometry, like objects, buildings, etc, are always heavily optimized for performance. Because of that, in most games no high res meshes will be used, which would bring down performance.

Here is a screenshot from a demo from nVidia, demonstrating infinite shadow volumes. The volume was made visible using the visible blended yellow planes you can see behind the mesh.

17) **Comparison of shadowing-methods**

a. **Comparison**

In the foregoing chapters I discussed several techniques to render shadows. There are a lot more realtime and not realtime techniques to do that, but I discussed only the most important and most used techniques for realtime rendering. All of the discussed shadow techniques also have the possibility to be used for dynamic light sources.

Now, a comparison of the techniques I discussed:

<table>
<thead>
<tr>
<th>plane projected shadow</th>
<th>projected shadows</th>
<th>depth shadow mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick, not much calculations</td>
<td>Quick, almost no calculations</td>
<td>Very quick, not much calculation</td>
</tr>
<tr>
<td>High detail</td>
<td>Detail depends on the texture</td>
<td>Detail depends on texture</td>
</tr>
<tr>
<td>No self shadowing</td>
<td>No self shadowing</td>
<td>Self shadowing</td>
</tr>
<tr>
<td>No shadow recyvers</td>
<td>Shadow recyvers</td>
<td>Shadow recyvers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vertex projection</th>
<th>shadow volumes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow with high-res meshes</td>
<td>Slow, lots of calculations</td>
</tr>
<tr>
<td>High detail</td>
<td>High detail</td>
</tr>
<tr>
<td>No self-shadowing</td>
<td>Self shadowing</td>
</tr>
<tr>
<td>No shadow recyvers</td>
<td>Shadow recyvers</td>
</tr>
</tbody>
</table>

All techniques clearly have their good and their bad sides. Which means it is very important to study which method would work best for your product. For example, it is useless to program for hours on a scene that calculates shadow volumes for only one quad casting a shadow. In this case, you’d better use vertex projection.

The easiest and quickest way to cast correct shadows with selfshadowing and shadow recyvers, is depth mapping (= shadow mapping). This method requires only some minimal calculations. The big consequence is that if you want shadows with a good resolution, you need a fairly big texture. And a bigger texture also means a serious drop of performance. Also, the extensions needed for shadow mapping are not available on all hardware.

The method creating the most beautiful and correct shadows is shadow volumes. The big problem with shadow volumes is the huge amount of calculations that you need to do for meshes with a high resolution. But once you determined the volume, you don’t need to re-calculate it, unless you’ve got a dynamic light-source, or a moving mesh. But why would one use this technique for static light-sources? You’d better use lightmaps. Here is a link where John Carmack, from the almost legendary Id Software, gives his vision and preference around stencil shadows vs depth maps (shadow mapping):
So one of the most important persons in the area of realtime rendering for games prefers shadow volumes above the low-res shadow maps. This clearly because of the more correct and beautiful shadows via shadow volumes.

However, you can combine different methods, to get a hybrid method.

b. Hybrid shadows
Hybrid shadows is the rendering of shadows using combined shadowing techniques. These mixed techniques can be a good way between the two mixed ones.

A good example of this is **shadow volume reconstruction**. This technique combines the 2 most important techniques we discussed, which are depth shadow mapping and shadow volumes. Here, via the depth buffer a depth texture is rendered (upmost figure), after which the contour of the shadow on the texture is determined using the depth values on the texture (figure 2). With this contour and the light position, you can reconstruct a volume, which can be used as a shadow volume (figure 3).

This hybrid technique is not as beautiful and correct as shadow volumes, but it is way more beautiful than shadow mapping. It is a bit slower than shadow mapping, but much quicker than shadow volumes. Which means it is a perfect combination.

The detail of the shadow volume will lie in the generated shadow map. You can get the edges of the shadow better via edge detection.

Via these links, you can find more documents about shadow volume reconstruction:
http://www.gamedev.net/reference/articles/article1300.asp
http://www.cgl.uwaterloo.ca/Projects/rendering/Talks/shadow/
http://www.cgl.uwaterloo.ca/Projects/rendering/shadow.html
18) **nVidia nFiniteFX: hardware programmable 3D**

### A. Lead-in

With the GeForce 256, nVidia brought the first graphical card on the market with a GPU (Graphics Processing Unit) on board. This GPU takes a lot of the needed calculations that are needed for realtime 3D from the CPU, so that the CPU has more time for other things.

Begin 2001 nVidia introduced the first programmable graphical card to the public, which is the GeForce 3. The GeForce 3 had the new nFiniteFX processor on board. This nFiniteFX engine enables you to use assembler scripts to communicate with the 3D-hardware, to make adaptions, define how rendering should be done, and much more. In the meantime, the GeForce 4 is also out, which contains the nFiniteFX 2 engine. Also other manufacturers, like ATI, have a card with a hardware-programmable engine. For example the ATI Radeon 8500.

Where before programmers had to use the graphics card’s possibilities for their applications, they can now define graphical effects themselves with this programmable hardware.

This, in fact, is the T&L (Transform and Lighting) where everyone’s talking about. T&L refers to the hardware-accelerated possibilities of the card.

I’ll discuss these features of the GeForce 3, because the applications for this thesis where programmed with such a card. The two most important possibilities via nFiniteFX are vertex shaders and pixel shaders. The most important is of course the vertex shader, because vertices are usually the bottleneck in real-time rendering.

#### a. Vertex Shaders

A big pro of Vertex Shaders with nFiniteFX is it’s programmability. Instead of just taking the vertex data and use it for some pre-programmed operations, developers can use nFiniteFX to define their own effects. So with vertex shaders, we can amongst others do character movements, keyframe animation, procedural deformation, morphing, environmental effects, lens effects, and lighting.
b. (per) Pixel Shaders
With pixel shaders, programmers can add per-pixel effects. Some of the most popular effects are per-pixel lighting, reflections and bumpmapping. The GeForce2 also contained these possibilities via nVidia Shading Rasterizer (NSR).

On the GeForce3, the nFiniteFX pixel shader has the possibility to handle four textures in one pass, combined with at max 8 texture-operations. This gives the GeForce 3 the possibility to get incredibly beautiful visual effect, like reflective bumpmapping, shadow-effects, realistic textures, etcetera, and still get a descent performance. The GeForce 3 can, for example, do up to 36 pixel-shading operations. The GeForce 2 GTS for example can only do 7.

Programmability is now also a new feature with pixel-shaders. There where you could before only use pre-defined functions, you can now define your own functions, that define the calculation of the pixel-shaders.

B. Vertex Programming (vertex shaders)
a. Lead-in
A vertex-program is in fact a string, that contains special assembly code that is necessary to communicate with your graphical card. This vertex-program is then compiled, and passed through to the hardware, which executes the operations and writes the result into the next position of the graphical pipeline.

A vertex program is made up out of several instructions, that perform actions on a set of parameters, such as vertex data and temporary data, and then generates some per-vertex output parameters.

An application can keep some of these vertex programs in memory. These programs can then always, if needed, be called.
Here is a schematic representation of the vertex program model. Vertex programs read from four types of memory: vertex attribute registers (read only), program parameters (read only), temporary registers (read-write), and per-vertex output registers (write only).

b. Vertex Attribute Registers
The vertex input data is being stored in vertex attribute registers, and can be read by vertex programs. They contain data like position, color, texture-coordinates, etc. On the GeForce3 there are 16 vertex attribute registers, which are v[0] to v[15].

This table shows the default values:

<table>
<thead>
<tr>
<th>i</th>
<th>Register</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>v[OPOS]</td>
<td>vertex position</td>
<td>x,y,z,w</td>
</tr>
<tr>
<td>1</td>
<td>v[WGHT]</td>
<td>vertex weight</td>
<td>w,0,0,1</td>
</tr>
<tr>
<td>2</td>
<td>v[NRML]</td>
<td>vertex normal</td>
<td>x,y,z,1</td>
</tr>
<tr>
<td>3</td>
<td>v[COL0]</td>
<td>primary color</td>
<td>r,g,b,a</td>
</tr>
<tr>
<td>4</td>
<td>v[COL1]</td>
<td>secondary color</td>
<td>r,g,b,1</td>
</tr>
<tr>
<td>5</td>
<td>v[FOGC]</td>
<td>fog coördinates</td>
<td>f,0,0,1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>v[TEX0]</td>
<td>texture coördinaat</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>v[TEX7]</td>
<td>texture coördinaat</td>
<td>s,t,r,q</td>
</tr>
</tbody>
</table>
d. **Program Parameter Registers**
An application can write values to constant memory registers, which can be read by vertex programs. Because you can write values into these to get effects as you want them, these are extremely important for the user. Typical are parameters like matrixes, lighting-parameters, pre-calculated values, etc. Some of the most important matrices in OpenGL, the modelview and the perspective matrix, can be made available automatically in the parameter registers. On the GeForce 3, 96 parameter registers are available, which are c[0] up to c[95].

e. **Temporary Registers**
The temporary registers are 12 floating-point vector registers, which are usable to contain temporary results while the vertex program is executing. These registers are R0 up to R11. They are initialized to (0,0,0,0) and are private for each call to a vertex program. They are read-write registers.

f. **Vertex Output Registers**
The vertex result registers (or vertex output registers) are 15 4-components floating point vector registers, used to write the result of a vertex-program into. The registers are initialized to (0,0,0,1) at the call to a vertex program.

These 15 result registers are:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>o[HPOS]</td>
<td>transformed vertex's homogeneous clip space position</td>
<td>x,y,z,w</td>
</tr>
<tr>
<td>o[COL0]</td>
<td>transformed vertex's front-facing primary color</td>
<td>r,g,b,a</td>
</tr>
<tr>
<td>o[COL1]</td>
<td>transformed vertex's front-facing secondary color</td>
<td>r,g,b,a</td>
</tr>
<tr>
<td>o[BFC0]</td>
<td>transformed vertex's back-facing primary color</td>
<td>r,g,b,a</td>
</tr>
<tr>
<td>o[BFC1]</td>
<td>transformed vertex's back-facing secondary color</td>
<td>r,g,b,a</td>
</tr>
<tr>
<td>o[FOGC]</td>
<td>transformed vertex's fog coordinate</td>
<td>f,**,*</td>
</tr>
<tr>
<td>o[TEX0..7]</td>
<td>transformed vertex's texture coordinates for texture units 0 to 7</td>
<td>s,t,r,q</td>
</tr>
</tbody>
</table>

g. **Instruction set**
So now we have all the registers, but without the instructions that define what to do with them, we wouldn’t get anywhere, right? The instruction-set available on the GeForce 3 exists out of 17 instructions that can do 4 operations at the same time over the 4 components of the vector registers.
These instructions work with either vector or scalar operand. Here is the definition of a vertex program instruction:

```
OpCode dst, [-]s0 [ , [-s [,-]2 ] ];  #comment
```

OpCode= name of the instruction  
Dst= target register  
S0= source 0 register  
S= source 1 register  
S2= source 2 register  
Comment is placed after a “#”

The next table shows the 17 possible instructions. Operands starting with ‘v’ are 4-component float vectors, and the ones starting with a ‘s’ are scalar floats.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARL s_dest, s_src</td>
<td>Loads floor of s_src into one coordinate of the address register</td>
</tr>
<tr>
<td>MOV v_dest, v_src</td>
<td>move source vector into destination vector</td>
</tr>
<tr>
<td>MUL v_dest, v_src0, v_src1</td>
<td>component-wise multiply on 2 vectors</td>
</tr>
<tr>
<td>ADD v_dest, v_src0, v_src1</td>
<td>component-wise add on 2 vectors</td>
</tr>
<tr>
<td>MAD v_dest, v_src0, v_src1, v_src2</td>
<td>adds third vector to the product of the first two</td>
</tr>
<tr>
<td>RCP v_dest, s_src</td>
<td>inverts source scalar, replicates result in vector</td>
</tr>
<tr>
<td>RSQ v_dest, s_src</td>
<td>computes inverse square root of absolute value of source scalar, replicates result in vector</td>
</tr>
<tr>
<td>DP3 v_dest, v_src0, v_src1</td>
<td>three component dot product of 2 vectors</td>
</tr>
<tr>
<td>DP4 v_dest, v_src0, v_src1</td>
<td>four component dot product of 2 vectors</td>
</tr>
<tr>
<td>DST v_dest, s_src0, s_src1</td>
<td>Computes distance attenuation vector (1,d,d²,1/d).</td>
</tr>
<tr>
<td>MIN v_dest, v_src0, v_src1</td>
<td>component-wise minimum on 2 vectors</td>
</tr>
<tr>
<td>MAX v_dest, v_src0, v_src1</td>
<td>component-wise maximum on 2 vectors</td>
</tr>
<tr>
<td>SLT v_dest, v_src0, v_src1</td>
<td>component-wise assignment of 1,0 or 0,0(1 if value of first source is less than second source, 0 otherwise)</td>
</tr>
<tr>
<td>SGE v_dest, v_src0, v_src1</td>
<td>component-wise assignment of 1,0 or 0,0(1 if value of first source is greater than second source, 0 otherwise)</td>
</tr>
<tr>
<td>EXP v_dest, s_src</td>
<td>exponential base 2,z contains 2^s_src,x and y contain intermediate results. w set to 1</td>
</tr>
<tr>
<td>LOG v_dest, s_src</td>
<td>Logarithm base 2,z contains approximation of log2</td>
</tr>
<tr>
<td>LIT v_dest, v_src</td>
<td>Lighting computation. Assumes input x contains (N.L), y contains (N.H), and w contains specular power. Resulting x and w has 1, y and z has diffuse and specular coefficients</td>
</tr>
</tbody>
</table>

So with these instructions, you can do about anything you want. They have been chosen very carefully, specific for 3D and vector operations.
h. **Examples**  
Here are several small examples, to give you an idea:

A three-component cross product:

```plaintext
MUL R2, R0.zyxw, R1.yzxw;  # R2=(z0*y1, x0*z1, y0*x1, w0*w1)
MAD R2, R0.yzxw, R1.zxyw, -R2;  # R2=(y0*z1, z0*x1, x0*y1, w0*w1)-R2
```

Transformation of a vertex position by a matrix, with a homogeneous divide:

```plaintext
# c[20-23] transformation matrix
DP4 R5.x, v[OPOS], C[20];  # position transformation
DP4 R5.y, v[OPOS], C[21];
DP4 R5.z, v[OPOS], C[22];
DP4 R5.w, v[OPOS], C[23];
RCP R11, R5.w;  # computing 1/w
MUL R5, R5, R11;  # homogeneous divide
```

(Voorbeelden van João L. D. Comba:  
[http://www.inf.ufrgs.br/~comba/research/hcg/hcg.html](http://www.inf.ufrgs.br/~comba/research/hcg/hcg.html))

I also programmed a simple example about this:

On the 3D-geometry that is visible on the screenshot above, a simple vertex program is active. It is a program that makes per vertex diffuse lighting.
The following array of strings contains the vertex program:

```c
const unsigned char DiffuseVP[] =
{
    "!!VP1.0"

    "DP4 R0.x, c[0], v[OPOS];"
    "DP4 R0.y, c[1], v[OPOS];"
    "DP4 R0.z, c[2], v[OPOS];"
    "DP4 R0.w, c[3], v[OPOS];"

    "DP4 o[HPOS].x,R0, c[4];"
    "DP4 o[HPOS].y,R0, c[5];"
    "DP4 o[HPOS].z,R0, c[6];"
    "DP4 o[HPOS].w,R0, c[7];"

    "DP3 R1.x, c[0], v[NRML];"
    "DP3 R1.y, c[1], v[NRML];"
    "DP3 R1.z, c[2], v[NRML];"

    "DP3 R1.w, c[8], R1;"

    "MUL o[COL0].xyzw, R1.wwww, v[COL0].xyzw;"

    "MOV o[TEX0], v[TEX0];"

    "END"
};
```

With only this array of strings, we haven’t got very much of course. To use vertex programs, we’ve got to initialize some extensions, which I will do using the extGL library.

Besides the initialisation of the extensions we have to initialize the vertex program, possibly place some constants in the registers, and pass the string with the vertex program with it.

```c
GLuint VPID; //this is the id of the vertex program
glGenProgramsNV(1,&VPID);
```

The above command is going to generate the vertex program, after which we will bind it (make it current).

```c
glBindProgramNV(GL_VERTEX_PROGRAM_NV,VPID);
```
When our “VPID” program is current, we clarify which string-array contains the program:

```c
glLoadProgramNV(GL_VERTEX_PROGRAM_NV, VPID, strlen((char*)DiffuseVP), (unsigned char*)DiffuseVP);
```

When the program-string has been binded, we’d better check for errors in the above code, like any good programmer would:

```c
if(glGetError() == GL_INVALID_OPERATION)
    printf("Error: glLoadProgramNV has returned an invalid operation!\n");
```

I already discussed that it is very easy to get the modelview and perspective matrix. That is what we do here:

```c
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 0, GL_MODELVIEW, GL_IDENTITY_NV);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 4, GL_PROJECTION, GL_IDENTITY_NV);
```

This way, the modelview matrix is being pushed into the first 4 registers (c[0] to c[3]), and the projection matrix into c[4] to c[7]. We can also, for example, put our own data in the registers, like a light position in the following line:

```c
glProgramParameter4fvNV(GL_VERTEX_PROGRAM_NV, 8, (float*)light);
```

This array “light”, which contains the position of the light-source, is put into register c[8].

Now we only should use the vertex program:

```c
void DrawScene(void)
{
    //Enable Vertex Programs
    glEnable(GL_VERTEX_PROGRAM_NV);
    //Bind Our Program
    glUseProgramNV(GL_VERTEX_PROGRAM_NV, VPID);
    glEnable(GL_TEXTURE_2D);
    Mesh.RenderDisplayList();
    glDisable(GL_TEXTURE_2D);
    //Disable Vertex Programs
    glDisable(GL_VERTEX_PROGRAM_NV);
}
```

So you simply need to enable the use of the vertex program with glEnable(). Then specify which program needs to be executed on what data. And lastly, close down if neccessary.
C. **Pixel Programming (pixel shaders)**

Pixel programming is also referred to as pixel shading. It is logical that per-pixel shading gives a much more beautiful effect than per-vertex shading. That is why the newest applications and computer games use them extensively.

On the GeForce 3, this per pixel shading goes mainly via texture shaders and register combiners.

a. **Texture Shaders**

On the GeForce 3, a texture shader consists out of a maximum of 4 texture layers. A texture layer performs operations on the incoming texture coordinates, to generate a new set of texture coordinates or to define a final color for the texture layer.

A texture shader program determines the operation to perform on a texture layer, and the implicit influence of the previous texture layers. Texture shaders can use 1D, 2D, 3D, cubemap or shadowmap textures as target.

There are 23 texture-shaders available on the GeForce 3. Some of them are: CULL_FRAGMENT, TEXTURE_3D, DOT_PRODUCT_DEPTH_REPLACE, OFFSET_TEXTURE_2D, etc. With these texture shaders, you can render special effects like 3D textures, fragment culling and cubemaps using the texture coordinates.

b. **Register Combiners**

As with texture shaders, register combiners are sorted in a set of layers. Where the texture shaders perform actions on the texture coordinates of textures, the register combiners perform operations on the output-colors of the textures, and on other user-constants and defined values.
On the GeForce 3, the register combiners are ordered in a row of maximal 8 general combiners, followed by a single final combiners. This figure illustrates this:

With register combiners, we can render effects like bumpmapping, per-pixel diffuse, etc in hardware.
c. An example of texture shaders
   As an example for texture shaders, I'll take a demo from Richard A. Nutman (http://www.nutty.org), which is: cubemapping. With TEXTURE_CUBE_MAP, a cubemap is being rendered in real-time here, and mapped onto a sphere, so that it looks like the sphere is reflecting the scenario. This demo used texture shaders, vertex programs and pixel shaders.

![Cubemap rendering example](image)

---

d. An example of pixel shaders
   As an example, I'll use a method of bumpmapping: “Per Pixel Diffuse Tangent Space Bump Mapping”. This demo was coded by Richard A. Nutman (http://www.nutty.org).

With this way of bumpmapping, pixel shaders are used to give the illusion of a 3d-structure on 3d-geometry. Here too, pixel shaders and vertex programs are used.

![Pixel shader example](image)

---

e. Another example from nVidia
   Lastly I found a demo on the nVidia site that clearly demonstrates the principle of texture and pixel shaders. http://nvidi.com/view.asp?IO=demo_shading

This demo illustrates several forms of texture and pixel shaders.
Conclusion

With this thesis, I wanted to study some of the mostly used techniques to get quick and real-time lighting in 3D. That is why this thesis consists out of a series of separate techniques. It is important to know that almost all of these techniques can be combined to get more realistic effects.

The specialized hardware to render realtime 3D is growing incredibly fast. The GeForce 4 is just available, and nVidia announces the NV30 and NV35, which will of course again be much more powerful. It is a very live and interesting time for 3D programmers. 3D Labs has just announced its new professional graphical card. This card will contain almost no standard functionality anymore, and will have to be used almost completely with the shading language, which is still in development. For example the OpenGL 2.0 shading language.

A big consequence is of course that consumers use a very wide range of hardware now. Programmers, especially game developers, need to code support for several kinds of hardware in their programs. With the arrival of a universal shading language, like in OpenGL 2.0, this will partially disappear. People will be able to use a standard language.

With my thesis, I tried to study a very varying range of methods to do real-time lighting, going from standard OpenGL lighting techniques, to the high-tech vertex and pixel shaders of the GeForce 3.

By coding several applications to support this thesis, I also acquired a lot of understanding of OO design in C++, and my qualities of a C++ programmer improved dramatically.
APPENDIXES
To give the best possible illustration about the creation of a lightmap, I can best use the code-example for the creation of a lightmap for a single triangle.

Here is the code for the lightmap-calculation of this one triangle. I'll try to explain the code as clearly as possible:

```c
inline void BuildLightMap(void)
{
    //temporary variables
    highx=highy=highz=-100000000;
    lowx=lowy=lowz=100000000;
    Depending on the normal of this triangle, we define the projection surface for this triangle.

    //YZ PLANE
    {
        //take the YZ plane
        for(int i=0;i<3;i++)
        {
            tri[i].x = 0;
            tri[i].y = Triangle.Vertex[i].y;
            tri[i].z = Triangle.Vertex[i].z;
        }
        ProjType=YZ;
    }

    //XZ PLANE
    {
        //take the XZ plane
        for(int i=0;i<3;i++)
        {
            tri[i].x = Triangle.Vertex[i].x;
            tri[i].y = 0;
            tri[i].z = Triangle.Vertex[i].z;
        }
        ProjType=XZ;
    }

    //XY PLANE
    else //else go for XY plane
    {
        //take the XY plane
        for(int i=0;i<3;i++)
        {
            tri[i].x = Triangle.Vertex[i].x;
            tri[i].y = Triangle.Vertex[i].y;
            tri[i].z = 0;
        }
        ProjType=XY;
    }
}
```

Depending on the normal of this triangle, we define the projection surface for this triangle.

//YZ PLANE
{
    //take the YZ plane
    for(int i=0;i<3;i++)
    {
        tri[i].x = 0;
        tri[i].y = Triangle.Vertex[i].y;
        tri[i].z = Triangle.Vertex[i].z;
    }
    ProjType=YZ;
}

//XZ PLANE
{
    //take the XZ plane
    for(int i=0;i<3;i++)
    {
        tri[i].x = Triangle.Vertex[i].x;
        tri[i].y = 0;
        tri[i].z = Triangle.Vertex[i].z;
    }
    ProjType=XZ;
}

//XY PLANE
else //else go for XY plane
{
    //take the XY plane
    for(int i=0;i<3;i++)
    {
        tri[i].x = Triangle.Vertex[i].x;
        tri[i].y = Triangle.Vertex[i].y;
        tri[i].z = 0;
    }
    ProjType=XY;
}
After we define the projection plane, we define the square containing the projected triangle, and therefore the texture-coordinates of this lightmap on the triangle.

```
//create our bounding quad
//get high and low values
for(int i=0; i<3; i++)
{
    if(tri[i].x > highx) highx = tri[i].x;
    if(tri[i].y > highy) highy = tri[i].y;
    if(tri[i].z > highz) highz = tri[i].z;
    if(tri[i].x < lowx) lowx = tri[i].x;
    if(tri[i].y < lowy) lowy = tri[i].y;
    if(tri[i].z < lowz) lowz = tri[i].z;
}
//quad is counterclockwise, so its 'normal' directs into the positive
//axis system
float highminuslowx=fabs(highx-lowx);
float highminuslowy=fabs(highy-lowy);
float highminuslowz=fabs(highz-lowz);
if(ProjType==XY)
{
    Quad[0].x = lowx;
    Quad[1].x = highx;
    Quad[2].x = highx;
    Quad[3].x = lowx;
    Quad[0].y = lowy;
    Quad[1].y = lowy;
    Quad[2].y = highy;
    Quad[3].y = highy;
    for(i=0; i<4; i++) Quad[i].z=0;
    //mapping coordinates
    for(int i=0;i<3;i++)
    {
        MappingCoords[i].x = fabs(tri[i].x-lowx)/highminuslowx;
        MappingCoords[i].y = fabs(tri[i].y-lowy)/highminuslowy;
    }
}
else if(ProjType==XZ)
{
    Quad[0].x = lowx;
    Quad[1].x = highx;
    Quad[2].x = highx;
    Quad[3].x = lowx;
    Quad[0].z = highz;
    Quad[1].z = highz;
    Quad[2].z = lowz;
    Quad[3].z = lowz;
    for(i=0; i<4; i++) Quad[i].y=0;
    //mapping coordinates
    for(int i=0;i<3;i++)
    {
        MappingCoords[i].x = fabs(tri[i].x-lowx)/highminuslowx;
        MappingCoords[i].y = fabs(tri[i].z-lowz)/highminuslowz;
    }
}
else if(ProjType==YZ)
{
    Quad[0].z = highz;
    Quad[1].z = lowz;
    Quad[2].z = lowz;
    Quad[3].z = highz;
    Quad[0].y = lowy;
    Quad[1].y = lowy;
    Quad[2].y = highy;
    Quad[3].y = highy;
    for(i=0; i<4; i++) Quad[i].x=0;
    //mapping coordinates
    for(int i=0;i<3;i++)
    {
        MappingCoords[i].x = fabs(tri[i].y-lowy)/highminuslowy;
        MappingCoords[i].y = fabs(tri[i].z-lowz)/highminuslowz;
    }
}
```
With the quad containing the triangle, we can determine the lumel array.

```c
//create lumel array from bounding quad
for(i=0; i< LMHEIGHT; i++)
{
    for(int j=0; j<LMWIDTH; j++)
    {
        if(ProjType==XY)
        {
            lumels[j][i].x = Quad[0].x + ( j*( (Quad[1].x-Quad[0].x)/LMWIDTH ) );
            lumels[j][i].y = Quad[0].y + ( i*( (Quad[3].y-Quad[0].y)/LMHEIGHT) );
            lumels[j][i].z = 0;
        }
        if(ProjType==XZ)
        {
            lumels[j][i].x = Quad[0].x + ( j*( (Quad[1].x-Quad[0].x)/LMWIDTH ) );
            lumels[j][i].y = 0;
            lumels[j][i].z = Quad[0].z - ( i*( (Quad[0].z-Quad[3].z)/LMHEIGHT) );
        }
        if(ProjType==YZ)
        {
            lumels[j][i].x = 0;
            lumels[j][i].y = Quad[0].y + ( j*( (Quad[3].y-Quad[0].y)/LMHEIGHT) );
            lumels[j][i].z = Quad[0].z - ( i*( (Quad[0].z-Quad[1].z)/LMWIDTH ) );
        }
    }
}
```

Now that we have the lumel array, we need to project it back to the plane in which the original triangle lies.

```c
//now that we have our lumel array, get the lumel quad back on the triangle
//plane
//So, we get the intersection point with our triangle, if there is any...
//define our triangle conform MagicFM library
MgcT.Origin() = Triangle.Vertex[0];
MgcT.Edge0() = Triangle.Vertex[1]-Triangle.Vertex[0];
MgcT.Edge1() = Triangle.Vertex[2]-Triangle.Vertex[0];
//for every lumel
for(i=0; i< LMHEIGHT; i++)
{
    for(int j=0; j<LMWIDTH; j++)
    {
        //origin
        line.Origin() = lumels[j][i];
        //direction
        if(ProjType==XY)
        {
            line.Direction() = Vector3(0,0,1);
        }
        else if(ProjType==XZ)
        {
            line.Direction() = Vector3(0,1,0);
        }
        else if(ProjType==YZ)
        {
            line.Direction() = Vector3(1,0,0);
        }
        //find point on our triangle
        if(FindIntersection(line, MgcT, pointofintersection))
        {
            lumels[j][i] = pointofintersection;
        } else { /*Keep current coordinates, don't really need them anyway*/ }
    }
}
Now we have the lumels in their 3D position, and we can define the lighting values for each lumel.

```c
// Now create a new image and calculate the light influence on it ;-) 
// create an image 
ilGenImages(1, &iluiID);
ilBindImage(iluiID);
ilClearColour(0, 0, 0, 255);
ilClearImage();
unsigned char rgb[3];

for(i=0; i< LMHEIGHT; i++)
{
    for(int j=0; j<LMWIDTH; j++)
    {
        float red,green,blue;
        lambda = Light.pos - lumels[j][i];
        float L = lambda.Length()/Light.strength;
        lambda.Unitize();

        float costheta = lambda.Dot(Triangle.Normal);
        float theta = acos(costheta)*180/3.14152;
        // calculate light strength
        if(theta<90 && theta>-90) // triangle faces the light
        {
            red = 255 * Light.brightness[0] * costheta / L;
            green = 255 * Light.brightness[1] * costheta / L;
        }
        else // triangle is backfacing from light
        {
            red=0;
            green=0;
            blue=0;
        }

        rgb[0] = red > 255 ? 255 : red;

        if(ProjType==XY)
        {
            ilSetPixels(j, i, 0, 1, 1, 1, IL_RGB, IL_UNSIGNED_BYTE, rgb);
        }
        else
        {
            ilSetPixels(j, LMHEIGHT-(i+1), 0, 1, 1, 1, IL_RGB, IL_UNSIGNED_BYTE, rgb);
        }
    }
}

Now we only need to free up some memory, by erasing what we don't need anymore.
```

```c
// close what we need no more ;-) 
glfwDeleteTextures(1, &textures[1]);
textures[1] = ilutGLBindTexImage();
ilDeleteImages(1, &iluiID);
```
Appendix B: Optimization techniques

While the high-tech hardware for 3D rendering can do a lot already, you still need to watch for bottlenecks. For example with the GeForce 3 you have the possibility to use 2048x2048 pixel textures, but that doesn’t mean that you should use such textures everywhere. Your card’s memory would be full within no-time, and your performance would drop quickly. So when you want to have detailed textures, you’d better split the big ones into multiple smaller ones.

In this appendix, I wish to talk about one of the most important optimization techniques. Of course, it ain’t very appealing to render beautiful 3D lighting-effects, when the rendering only goes at 3 frames per second.

A) Textures

As said before in this appendix, it is important to use smaller textures. To get an optimal hardware acceleration, you’d best use textures in the shape of $2^n \times 2^n$ pixels.

The best way is to load textures as objects into the video-memory. That’s the way I do it in all of the demo’s in this thesis. It is extremely important for realtime 3D that these textures don’t have to be loaded again for each rendering pass. This should come very natural to you, but I just wished to press this a little extra.

When textures have to be changed in real-time in the application, the use of glCopyTexSubImage2d is recommended. This is the fastest function to change parts of a texture. This is done in, for example, my own rendering to texture demo’s.

Texture sorting is also important. This technique means that you sort 3D geometry by texture. If there are multiple meshes sharing one texture, then this texture should be binded using glBindTexture, and all geometry using this texture should be rendered. This makes the amount of texture-changes smaller, which speeds up rendering by a huge amount. The amount of times glBindTexture needs to be called also goes down, so you get less function-overhead.
B) Vertex Submission
As should be clear by now, 3D geometry exists out of primitives. Most times triangles, but sometimes sometimes quads too. It isn’t recommended to render a mesh by passing three vertices for each triangle. You’d better create a triangle strip from your mesh data. Triangle strips have a much higher performance than when you pass each vertex from each triangle. With triangle strips, you also get less function overhead, because of less calls to glVertex.

Besides triangle strips, there are some other techniques you can use. You can, for example, use vertex arrays. With these vertex-arrays, you only need to pass an array containing your mesh data, which you can activate with only one function-call.

Besides that, with the current generation of graphical cards you can buffer vertex data into your card's memory. This is the quickest way to make your mesh data available to your graphical card, because it doesn’t need to be transferred from PC-memory.

The last technique I wish to discuss here shortly is the display list. In OpenGL it is possible to generate display lists. This means that you generate a name for a list, and then you can render to it. When you call the list, the rendered “image” is called back. This is a very fast way of rendering, but with the consequence that you cannot change the mesh data cannot be changed once it’s put into the list. That’s why this technique is only recommended for static meshes.

With 3D-rendering, it is important to render only what can be seen. The back of a mesh is invisible, so why should we render those triangles? We can also use frustrum culling. With technique, only what is in the frustrum called the rendering context is rendered. With these two relatively simple techniques you can gain a lot of speed.

C) Others
Of course, there are other optimization techniques. Think for example of assemply code to utilize the 3Dnow! functions on an AMD CPU.

Code optimizing can also be important. A lot of typecasting can seriously slow down your application. For example, take a loop which converts a double to an integer a thousand times, and then a loop which copies an int to an int a thousand times. The difference will be clear. Besides the optimization techniques that I spoke about, there are lots of other things that can be optimized. Think for example about loading and saving data.

More about this here:
http://www.mesa3d.org/brianp/sig97/perfopt.htm
22) Appendix C: Explaining list of words

A dictionary for 3D-rendering terminology can be found here:
http://oss.medialab.chalmers.se/dictionary/

3D geometry:
A vertex is a point defined in 2D or in 3D. From 3 vertices, we can create a triangle. A polygon can be a triangle, a quad, or any other surface created from vertices. A mesh usually exists out of multiple triangles.

Attenuation:
In real-life, the strength of light declines linear to the travelled distance. In 3D, this attenuation can also be set, which can make a scene look more realistic.

Bump maps:
Method using per-pixel shading, giving the illusion of a rough surface. The figures displayed here can give a good idea of that. The first, second and third texture are combined to get 4.
Environment mapping:
A texturemap, giving the illusion of a reflected environment, as with cubemapping or spheremapping.

Fog:
Simple, but often used graphical effect, that looks like mist. It is most times used to camouflage the far plane.

Frustum:
The frustum is the viewable 3D environment. It’s a pyramid, with the top taken off, which contains the full rendering context. The frustum is the green part on the next picture.

Hardware rendering:
Rendering using computer hardware. This hardware is very specialized, and can be specifically made to speed up 2D and/or 3D geometry.

Hidden Surface Removal:
An algorithm to remove non-visible 3D geometry. Backface culling is a quicker and, at the moment, more used technique.

Lightmap:
A map with light values, rendered for each triangle of the scene, giving the impression of high resolution lighting.

Lumel:
Short for LUMinance Element. The lumel is a pixel in a lightmap, containing the strength of the light for each pixel.

Multitexturing:
Applying multiple textures in layers on top of each other, on one and the same triangle or quad. The colors of these textures is then combined in a predefined way.
Pixel:
Short for Picture Element. The basis for each graphical screen on the computer.

Primitives:
Pure shapes, uses often with 3D modeling. They are the building-blocks for 3D scenes. In this thesis, triangles and quads are seen as primitives. Kubes, spheres, etc. are seen as meshes.

Projected Shadows:
A shadow coming from an object and falling onto a surface.

Rendering:
The creation of 2D and/or 3D graphical scenes using math.

Rendering pass:
A unity of rendering of a certain scene. With shadow mapping for example, the scene needs to be rendered twice. One time to the depthmap, and two times to the screen. That's why this technique consists out of 2 rendering passes.

Shading:
A way of rendering. You can render a mesh in wireframe, flatshading, or smooth (= gouraud) shading.

Hardware Rendering Techniques

Wireframe  Flat Shading  Gouraud Shading
**Smoothing:**
Technique to make low-res meshes look smooth using interpolation.

**Stencil Buffer:**
The stencil buffer, if turned on, is a part of the video-memory that contains one to eight bits information about each pixel in your scene.

**Texture Mapping:**
Applying an image on to 3D geometry. Using texture-coordinates, which are normally specified per vertex, these textures (images) can be mapped (applied) to the 3D geometry.

**Z-buffer:**
Also called depth buffer. This is a 16 or 32 bits two dimensional array of integers, with the same dimension as your screen resolution. If it is 16 or 32 bits depends on your graphical card. Every time a dot is rendered to the screen, the rasterizer checks if there is a dot already there that should hide the new dot. If the existing dot is closer in the Z-hyrarchy, the new dot should not be rendered.
23) Appendix D: Sources

WEBRESOURCES

1) General
   http://www.gamedev.net (game design)
   http://www.gamasutra.com (game design/corporate news)
   http://www.gametutorials.com (game programming tutorials)
   http://www.wotsit.org/ (file format descriptions)

2) Tutorials/docs/coding
   http://www.opengl.org (opengl main site)
   http://nehe.gamedev.net (the place to be ;-)
   http://polygone.gamedev.net (lightmapping en raytracing tutorial)
   http://www.scriptspot.com (Max scripting)
   http://romka.demonews.com (demos, tutorials)
   http://www.mesa3d.org/brianp/sig97/perfopt.htm (opengl performance optimizations)
   http://hufo.planet-d.net (demo coding and skeletal deformation)
   http://cal3d.sourceforge.net (free skeletal deformation library)
   http://demeter.sourceforge.net (free terrain library)
   http://www.nutty.org (nutty’s opengl site – vertex program demo’s)
   http://nate.scuzzy.net (tutorial, doc, …)
   http://www.baskuenen.myweb.nl/ (game design toegepast)
   http://graphics.stanford.edu/projects/shading/ (scripted shading projekt)
   http://www.revolution3d.de/ (Visual Basic 3D engine)

3) Specific
   Tools / Libraries
   http://www.opengl.org
   http://www.magic-software.com
   http://openil.sourceforge.net
   http://www.uni-karlsruhe.de/~uli2/index.html
   OpenGL lighting:
   http://www.xmission.com/~nate/tutors.html
   http://nehe.gamedev.net/tutorials/lesson.asp?l=07
   Blending:
   http://nehe.gamedev.net/tutorials/lesson.asp?l=08
   Multitexturing:
   http://gritche.chez.tiscali.fr/OpenGL/
   Lightmapping:
   http://www.flipcode.com/tutorials/tut_advlightmaps.shtml
   http://www.flipcode.com/tfiles/kurt01.shtml
   Rendering to texture:
   http://www.exsanity.freeola.com/opengl.html
Bump mapping:
http://developer.nvidia.com
http://www.ati.com

Shadow techniques:
http://www.gamedev.net/reference/articles/article1300.asp
http://www.extremetech.com/print_article/0,3428,a=5687,00.asp

BOOKS
1) Programmeren Algemeen
   Het complete handboek Visual C++ (Kate Gregory, Academic Service)
   OO Software Design and Construction with C++ (Kafura, Prentice Hall)
   Windows 98 Programming Secrets (Clayton Walnum, IDG books)
   Aan de slag met C++ (Gertjan Laan, Academic Service)
   C++ (Leen Ammeraal, Academic Service)

2) Graphics/Game Programming
   3D Computer Graphics (WATT)
   Game Programming Gems (Mark DeLoura, Charles River Media)
   Compressed Image File Formats (Miano, Addison Wesley)
   OpenGL Programming Guide (Mason Woo, Addison Wesley)
   OpenGL Superbible (Wright&Sweet, Waite Group)

3) Andere
   Dictionary of Mathematics (Unwin Hyman)
   Inside 3D Studio Max Vol II & III (New Riders)
24) **Appendix E: Specs of the development platform**

These are the specifications of the system on which I wrote my demo's:

**Hardware:**
- AMD Athlon 1400 MHz
- Asus A7M266 mainboard
- Corsair PC2400 512 MB DDR Ram
- Harddisk Maxtor 40 GB 7200rpm
- Asus V8200 Deluxe GeForce 3
- Monitor LG 995E Plus

**Software:**
- Microsoft Visual C++ 6.0 (coding)
- Microsoft Visual Studio.NET (tests)
- Adobe Photoshop 6 (textures, screenshots)
- Notepad (script, 3D file editing)