Diploma Thesis

Methods for Real-Time Lighting

Carsten Fuchs

February 18, 2005

Diploma Thesis in Computer Science

Methods for Real-Time Lighting

presented by

Carsten Fuchs

written at

the Faculty of Computer Science
Chair of Computer Graphics, Prof. Dr. Hans-Peter Seidel
Universität des Saarlandes

Supervisors:
Dr. Jan Kautz
Prof. Dr. Hans-Peter Seidel

Assessors:
Prof. Dr. Hans-Peter Seidel
Prof. Dr. Philipp Slusallek

Begin: 1st July 2004
End: February 2005

Abstract

This document presents three individual methods for real-time lighting: radiosity-based light-maps, dynamic Phong shading combined with stenciled shadow volumes implemented in programmable graphics hardware, and lighting with spherical harmonics. Each method is introduced and presented both from a theoretical and practical point of view. All methods have been implemented in the framework of the Ca3D-Engine. Wherever applicable, interesting algorithms and implementation details (such as optimizations) are pointed out and discussed in depth.

New aspects elaborated in this thesis include optimizations of shadow volumes for meshes that are organized in BSP trees and the presentation of the conceptual and algorithmic parallels between Spherical Harmonic Lighting and traditional light-maps with radiosity: Bounce-transfer SH light coefficients are precomputed in a way that is analogous to a typical radiosity algorithm, and the storage of the results is achieved similarly to that of regular light-maps. Moreover, algorithmic enhancements are presented, including per-pixel evaluation of SHL, the combination of SHL with normal-mapping, compression of SH coefficients, and filtering of SH rendering.

Contents

1 Introduction

Recent years have seen a boom and shown substantial advances in 2D and 3D computer graphics: numerous applications like flight simulators, computer games, software for architectural purposes, visualization in medical research and molecular biology, software for testing human effectiveness[1], extensive use in the film industry, visualizations and simulations in automobile and aircraft design and many others express an ever increasing demand in constantly progressing 3D graphics rendering.

The two ultimate goals that 3D graphics has to achieve, and towards which it is constantly developing, are *realism* (or at least the illusion thereof), and *real-time*.

In the past, researchers in computer science, software engineers and hardware vendors have taken great steps towards these goals: new methods and ideas have been developed and simultaneously vendors have put new programmable graphics processors at main-stream prices on the market that permit low-cost implementation of the new theories in consumer hardware.

Nonetheless, limits both in algorithms and hardware power continue to govern the balance between realism and real-time: Realism often requires so much processing time that achieving real-time frame rates is not possible. Inversely, achieving real-time 3D graphics often means cutting realism.

A very important aspect of describing both reality as well as computer synthesized images is *light* and *lighting*. The computer graphics community models the light that occurs in the real physical world with various degrees of accuracy. These models often take widely varying approaches, each with its own interesting aspects, strengths and weaknesses.

This thesis discusses three state-of-the-art techniques for real-time lighting of three-dimensional scenes. Each technique is built on a certain lighting model and a theoretical foundation, and employs contemporary methods for implementation.

1.1 Overview

The contents of this thesis is organized as follows:

- Section 2 outlines the research efforts and products that preceded this thesis.
- The first lighting method that is discussed is Light-Maps, presented in section 3.
- Section 4 details the concepts of contemporary hardware-accelerated lighting. This includes algorithms related to the Phong shading and lighting models, combined with stenciled shadow volumes for casting dynamic shadows.
- The most contemporary and most interesting method for real-time lighting, lighting with Spherical Harmonics, is presented in section 5.

[1] For example, the *U.S. Air Force Research Laboratories for Human Effectiveness* employ 3D graphics powered by the Ca3D-Engine for testing the visuo-spatial working memory of their pilots.

- Section 6 discusses various in-depth implementation aspects that occur with the presented lighting methods.
- The thesis concludes with a summary and discussion in section 7.

2 Previous Work

Light-Maps and Radiosity

Computer-aided synthesis of realistic images by employing precomputed *light-maps* for fake-lighting the scene geometry became popular as soon as consumer desktop computers became fast enough to handle it: When *Id Software Inc.* released their first-person-shooter *Quake* in 1996, all rasterization, texture lookups, and other rendering tasks were achieved in software, on the general-purpose CPU and with handwritten assembly routines. Among other features, Quake became known for its ability to modulate the regular texture-map with the value of a grayscale light-map with a minimum of assembly instructions (see [Abr96] for details), which made it the most advanced game rendering system of its time. The light-maps were simple low-res textures, created by algorithms that were simply "tweaked until they looked good" (John Carmack). No physically sound illumination model was employed at that time.

A couple of years later, the introduction of 3D-accelerating graphics boards spread the use of light-maps even further. Their multi-texturing features that were accessible via convenient high-level APIs such as OpenGL made it easy to employ fully colored RGB light-maps at very high speeds. Also the concepts and applications to light-mapping have been extended and varied over time: Segal, Korobkin, vanWidenfelt, Foran and Haeberli projected in 1992 light-maps into the scene rather than just modulating regular textures with them, and Heidrich, Kautz, Slusallek and Seidel used them for special effects like slide projectors in 1998.

Moreover, the trend moved to precompute the light-map contents with expensive but high-quality global illumination methods, based on thorough and physically accurate light models: Radiosity models have their origin in the field of thermal engineering (heat transfer), for which the first global methods were developed. After the transfer to lighting applications and the acknowledgement of the stunning results, a lot of research efforts were invested into making the implementations more practically feasible, and to overcome the inherent problems and limitations: While the initial formulation of radiosity methods only included Lambertian diffuse reflection, ways have also been found to take specular reflections into account. Other researchers were able to factor additional media like smoke and haze into the radiosity formulation. Another very important aspect is the effort that has been made to reduce the computational complexity of the numerical radiosity solutions. Various methods have been developed that approach the problem in many different ways. A good survey about global illumination and the radiosity method is [CW93]: One chapter covers the history of radiosity, along with dozens of references to original literature. The rest of the book teaches both the theoretical and practical aspects of modern radiosity algorithms. [FvDFH90] also contains a very good treatment. Even today, global illumination remains a topic of ongoing research.

Dynamic Lighting

Independent of and chronologically actually before the exploration of global lighting, the computer graphics community created local *illumination models*. These models describe the color of a point on a surface that is directly lit by a number of light sources. Foley et al. ([FvDFH90]) present these models, which take ambient light, diffuse reflection, atmospheric attenuation and specular reflection into account. Phong and Blinn both provided variants for the specular reflection illumination model, and Warn extended the point-light-source model further. The common property of these models is that they are approximations of the laws of optics and radiation, and thus have no physical foundation. They are still in common use because they yield pleasing and attractive results and are simple to compute with minimal effort.

Foley et al. continue to organize or "fit" the illumination models into the broader framework of *shading models*, that in turn state which, how and when an illumination model is employed. Interpolated or *Gouraud* shading evaluates the illumination model only at each vertex of a polygon, and interpolates the resulting color across the polygon area. *Phong* shading, on the other hand, means to perform the illumination computations at every rasterization fragment.

Phong shading was and still is frequently employed in *Pixar's PhotoRealistic RenderMan*® that is capable of rendering photo-realistic images. This however is usually only possible offline, at noninteractive rates. In parallel, graphics boards with 3D-acceleration started to offer interactive frame-rates, but did not offer enough flexibility to provide much better than Gouraud or very limited Phong shading.

A couple of years ago, 3D-accelerator vendors provided their consumer boards for the first time with a programmable graphics processing unit. Initially only accessible in assembly language dialects, higher level tools and driver bindings to contemporary graphics APIs were quickly developed. Examples include ATIs *RenderMonkey*(TM) tool suite, NVidias *Cg* GPU programming language, the latest versions of Microsoft *DirectX*, and the most recently introduced *OpenGL 2.0*.

It turned out that programmable, dedicated GPUs were indeed a revolutionary step: While they did not introduce anything that was *conceptually* new, freely and flexibly programmable pixel and vertex shaders carried almost all of the power of RenderMan programmability to dedicated graphics hardware, which was able to achieve interactive frame rates while executing the programs. Joining free programmability with the high speed of dedicated hardware thus joined the best of both worlds.

Although arbitrarily sophisticated shading models can now be implemented, the hardware development and performance improvements still continue. For performance reasons, even the most recent commercial games like *Doom3* from *Id Software Inc.* and *Half-Life 2* from *Valve Corporation* often do not implement more advanced shading models than variants of the well-known Phong shading. The associated implementations come with their own interesting sets of properties and subtleties. They frequently obtain their data for Phong shading from special texture-maps like diffuse-maps, bump- or normal-maps, specular-maps, luminance-maps, etc. The central issue of texture (tangent) space is addressed by Dietrich in [Die03].

The above-mentioned previous work in dynamic lighting was paralleled by complementary developments, especially with regards to global dynamic shadows. *Shadow-Mapping* exploits the fact that when the scene is rendered from a light-sources point-of-view, the resulting depth buffer can be considered as a precomputed light visibility test that can be used for determining shadows when the scene is rendered from the observer's point-of-view in a second step. Several papers and publications by graphics hardware vendors detail how this principle can best be exploited on their proprietary hardware. Moreover, researchers have produced interesting variants, improvements, and extensions to the basic shadow-maps principle: Brabec and Seidel combined them with light-maps and extended them for soft shadows in 2000. They also describe antialiased shadows with shadow-maps on graphics accelerated hardware in [BSa].

Another approach, the *Shadow Volumes*, was introduced by Crow in 1977. It was provided with dedicated hardware support in 1985 and generalized over the years e.g. by Bergeron in 1986. Heidmann in 1991 observed that Shadow Volumes can be implemented well by exploiting the stencil buffer of graphics hardware. Unfortunately, the resulting algorithm was not robust in many cases. Several authors (e.g. Diefenbach and Kilgard) suggested workarounds to the problems, but the solutions remained "fragile". Bilodeau and Carmack observed in 2000 that the reversal of the depth comparison tests achieves equivalent results to Heidmann's original approach. The elimination of the far clip plane finally made the method robust, as described in [EK02]. [EK03] and [MHE$^+$03] point out additional optimizations and improvements to the technique. [AAM03] and [ADMAM03] extend the shadow volume technique to soft shadows, and [BSb] describes how all shadow volume computations can be moved to the graphics hardware.

The Ph.D. thesis of Stefan Brabec ([Bra]) treats in detail both shadow-maps and shadow volume techniques, including their extensions to soft shadows.

Spherical Harmonic Lighting

Lighting with Spherical Harmonics was first introduced by Peter-Pike Sloan, Jan Kautz and John Snyder in [SKS02]: They precomputed global light transfer that was represented as a set of spherical functions and efficiently stored by the use of spherical harmonics. The incident (potentially dynamic) light was also expressed via spherical harmonic coefficients, and they showed that the final lighting computations can be performed entirely on the spherical harmonics basis. Robin Green's article [Gre03] is based on that original paper and discusses the same topic from the more practical point-of-view of a game developer. He includes source code fragments and many examples. An exemplary project built on these papers with complete source code is available at *Paul's Projects* at `http://www.paulsprojects.net`. Annen, Kautz, Durand and Seidel continued their research and employed "Spherical Harmonic Gradients for Mid-Range Illumination" in [AKDS04]. Ren Ng, Ravi Ramamoorthi and Pat Hanrahan use a variant of the SH approach in order also to take high frequency shadows into account ([NRH03]).

Other Related Work

Many (sub-)tasks in 3D computer graphics require spatial classification and navigation facilities, for example for ray–polygon intersection tests. One of the most common and best known data structures for these purposes is the *Binary Space Partitioning Tree*. Originally only used for determining visible surfaces, Thibault and Naylor used BSP trees in 1987 to organize arbitrary polyhedra. Despite the fact that BSP trees require static geometry (otherwise the tree must be rebuilt), their benevolent properties have made them indispensable until today: For example, they provide spatial sorting in linear time and spatial classification in logarithmic time.

Seth Teller describes *Potentially Visibility Sets* in [Tel92] as a very powerful means to precompute the mutual visibility of BSP leaf cells. The potential mutual visibility can then be looked up at render time at nearly no cost. Subsequent developments to reduce both the mathematical and computational costs of Tellers complex approach do exist, as for example in some computer games and in my own implementation, however I'm not aware of any other thorough scientific treatment.

3 Radiosity based Light-Maps

This section presents the theory, principles, and algorithmic aspects of radiosity-based light-maps. Light-maps are the oldest and best known lighting technique that is discussed in this thesis.

3.1 Theory of Light-Maps

Light-maps store the solutions of algorithms that compute global lighting. Arbitrary algorithms can be employed for this purpose and their results stored. For example, *Quake1* used an algorithm that was "just tweaked until it looked good" (John Carmack). Even today many contemporary commercial products and games employ similarly simple algorithms.

However, improvements during the last decade both in algorithmic research and computational power clearly suggest employing radiosity algorithms for computing light-maps. Cohen and Wallace [CW93] provide a thorough introduction to radiosity and the underlying concepts of lighting. Ashdown [Ash94] describes radiosity from a more practical point of view and includes concrete examples and code. Foley et al. [FvDFH90] have a text with a very good balance between theoretical backgrounds and suitability for implementation.

Radiosity algorithms propose solutions to the rendering equation under the assumption that only diffuse reflection occurs. The generic rendering equation

$$L(x', \vec{\omega}') = L_e(x', \vec{\omega}') + \int\limits_S f_r(x)\, L(x, \vec{\omega})\, G(x, x')\, V(x, x')\; dA \tag{1}$$

can be simplified by this assumption, which implies that the BRDF component f_r becomes independent of the incoming and outgoing directions, *and* that the outgoing radiance from a Lambertian surface is the same in all directions. As is shown in [CW93], equation 1 therefore reduces quite dramatically to

$$B(x) = E(x) + \rho(x) \int\limits_S B(x') \frac{G(x, x')\, V(x, x')}{\pi}\; dA' \tag{2}$$

This equation is called the *Radiosity Equation.*

As the quantity that is to be computed still appears both on the left-hand side and on the right-hand side under the integral, it is almost impossible to solve this equation analytically in a closed form. Therefore, the domain (surfaces) S must be discretized in order to make them viable for numerical methods.

3.2 Properties of Radiosity Algorithms

Despite considerable efforts put into research of synthesising realistic images in real-time with advanced ray-tracing techniques that produce results comparable to radiosity methods, radiosity – being a method for computing global lighting – is still inherently applicable only to mostly static scenes.

Radiosity is therefore characterized by precomputed light distributions that in turn require a static scene and static area light sources. On the other hand, radiosity is also view-independent, and therefore works well with a dynamic viewer.

3.3 CaLight: Computing the Light-Maps

A full radiosity implementation is provided by the *CaLight* component of the Ca3D-Engine. The algorithms are described in considerable detail below, because in section 5.3 on Spherical Harmonic Lighting it will be shown that many concepts that are discussed here do also apply to SHL. SHL, that can loosely be defined as "Radiosity with dynamic light sources", will employ many analogous but more sophisticated steps that build on the details presented here. The subsequent sections build on the basics of the Ca3DE framework that is laid out in section 6.1.

3.3.1 Extending the PVS to Surfaces

CaLight starts by extending the PVS information that has been precomputed for entire leaves (see section 6.1.1) to individual surfaces.

Initially, a visibility matrix $\text{PVS}_{surfaces}$ (an array) of $n \times n$ elements is allocated for storing mutual visibility for all pairs of n surfaces. Each matrix element $\text{PVS}_{surfaces}(i, j)$ (where $0 \leq i < n$ and $0 \leq j < n$) describes one of three relationships between surfaces i and j:

- Surface i may see surface j *completely.* That is, no line segment that starts in i and ends in j is obstructed by any interfering occluder.
- Surface i may see surface j *not at all.* No unobstructed line segment from i to j exists.
- Surface i may see surface j *partially*: Line segments from a point on i to a point on j exist that are obstructed, and such that are unobstructed.

Obviously, the array is diagonally symmetric, as $\text{PVS}_{surfaces}(i, j) = \text{PVS}_{surfaces}(j, i)$ for all $0 \leq i < n, 0 \leq j < n$.

In a first step, for each leaf L, we mark all surfaces of all leaves that are in the PVS of L as (partially) visible from all faces of L. As this step alone yields only a gross estimation of the information that we actually want (at the level of surface-to-surface rather than leaf-to-leaf visibility, plus the "complete visibility" flag whenever possible), the next steps refine the current matrix. The first idea is to exploit the fact that each surface is planar, and thus can only see others if they are above or intersect the plane of the surface. In other words, we may reset the mutual visibility of two surfaces to "none" if the first is below the plane of the second or vice versa. Nonstandard surfaces like sky surfaces are special cases, and we may reset their mutual visibility to all other surfaces to "none", too. Finally, we determine which of the remaining "partial" visibility relationships are actually "complete visibility" relationships. This is achieved by constructing the spatial convex hull over each pair of partially visible surfaces, and then testing if any other

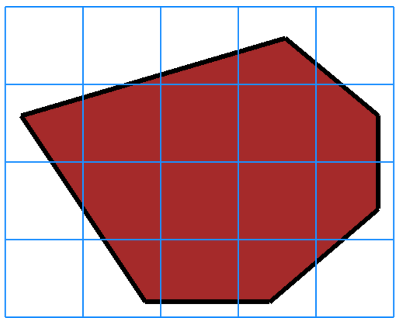


Figure 1: A polygon that is covered with the regular grid of patches. Each patch corresponds to a light-map element.

surface extends into this convex hull. This process is much accelerated by the proper use of bounding boxes.

The such obtained information in PVS$_{surfaces}$ proves very useful later during the core steps of the algorithm.

3.3.2 Preparing the Patches

The patches cover all surfaces like a grid, and are the fundamental data structure on which the radiosity algorithms operate. Therefore, we allocate a grid (array) of patches for each surface. The patches are constructed in such a way that their position, dimensions, and number exactly match the parameters of the light-map that is associated with that surface. That is, each light-map element matches exactly one patch. It is the simultaneous allocation of all patches for all surfaces that makes CaLight expensive in terms of memory space consumption.

After the patches have been allocated, their initial energies (radiant exitances) are set to zero and their spatial center coordinates are computed. We also compute whether the patch is located inside its surface (at least partially). Patches can be outside their surfaces, because patches are arranged in regular grids (as are light-maps), covering convex but irregular surfaces. See figure 1 for an example polygon that is covered with patches. This information will become important later, as it helps us to treat the borders of the polygons correctly. Handling the borders of polygons well in setups like this (regular grids cover irregular polygons) is a tough problem in general, and especially in almost every part of radiosity algorithms. The information computed here helps significantly in this regard.

The final step in patch preparation involves casting sunlight onto the freshly initialized but otherwise still dark patches. This is easily achieved by tracing rays through the Ca3DE world[2]: starting from each patch's center coordinate in the opposite direction of

[2]Due to the hierarchical representation (as a BSP tree) of a Ca3DE world, tracing rays through it can be achieved very efficiently in logarithmic time. This property is also helpful for the main algorithm later, which shoots millions of rays.

the sunlight, we only have to determine whether the ray hits a "sky" surface. On hitting, we simply assign appropriate values to the total energy and the not-yet-radiated energy of the patch. As it has turned out that that approach tends to produce light-maps whose sunlight to shadow transitions suffer from the well-known aliasing or "staircase" effect (as is the nature of low-res light-maps), we employ both multi-sampling of the above described sunlight derivation, plus carefully apply a smoothing filter immediately after all patches got their sunlight energies assigned. The combination of both measures leads to visually pleasing transitions.

3.3.3 Direct Lighting

The direct lighting phase is nothing but a special case of the subsequent bounce lighting phase, and in fact makes use of the same subroutines. Its purpose is to assign initial energies (radiant exitances) to the patches of surfaces that have been defined as light emitters. Then, one shooting step of these energies into the environment is performed. After this step our patches contain only emissive plus *direct lighting* – thus the name for this phase.

Another substep that is performed here is the assignment of energy that is emitted by *point light sources*. Point light sources do not exist in physical reality, but are easy to model mathematically and algorithmically. For each patch that is in the PVS of a point light source, we compute the energy that it receives based on the point light sources irradiance and geometric attributes. This step too is in the sense of "direct lighting".

3.3.4 Bounce Lighting

The bounce lighting phase is the core of the radiosity algorithm, and as such the most interesting part. It consists of an infinite loop that has two essential steps: The determination of the patch that should next shoot its as yet unradiated energy into the environment (or the termination of the loop if no such patch exists), and the actual shooting (radiation) operation of that patch.

The next patch for shooting its energy could in theory simply be selected by choosing the successor of the current patch in the global array of all world patches (i.e. picking the patches in linear sequence). This strategy, however, will cause us to converge towards the final solution as slowly as the *Gathering methods* that employ Gauss-Seidel iteration do. See [CW93] for a treatment of these techniques. As Cohen and Wallace further point out, we can converge much more quickly (i.e. in much fewer iterations) by the method of *Progressive Refinement*, which is a modification of the Southwell iteration method. The physical interpretation of this method is that a patch "shoots" its energy into the environment, and by selecting the patch with the largest as yet unshot energy, we can achieve quick progress. As choosing that very patch in each iteration can be expensive in its own right, I've chosen for CaLight to simply pick the largest patch from a set of 10 random samples of each surface. This selects a patch in a cheap and near optimal manner. Only when the unradiated energy of the chosen patch is below a user-defined threshold, CaLight employs a full search for remaining patches that have a higher value.

If still no adequate patch is found, we can assume that we have reached the desired solution (ignoring the remaining small amounts of unradiated energies that are all below the user-defined threshold), and terminate the loop which finishes the bounce lighting phase.

Shooting Unradiated Energy

If however a patch has been found with a high unradiated energy, we next radiate (or shoot) its energy into the environment.

Before doing so however, we employ a simple optimization that's derived from the chapter about Hierarchical Methods in [CW93]: The number of form factors that have to be computed between individual patches can be reduced when the interaction between two groups of patches that are widely separated is approximated with a single interaction. That means that we start by summing up the unradiated energies of all patches in a square of $n \times n$ patches near the original patch. Simultaneously, we average the positions of the patch centers in order to obtain a position for all patches in the square. In order not to further complicate the algorithm, we normally fix n at 3. This value can be changed by the user, but a better solution of course would be to choose n dynamically, e.g. based on the distance to the other surfaces (the shooting targets). This is fairly simple if all target surfaces are of roughly the same distance to the shooting patch(es), but is more difficult if some shooting targets are very close and others are far away. In this case, we had to build multiple groups of patches for multiple choices of n, and use the smaller groups for shooting at closer surfaces, and the bigger groups for shooting at surfaces that are farther away. The algorithms for this concept are quite delicate (I expect both a performance and quality gain though), and will be added to CaLight in a future release.

Now that we know what to shoot and from where, we continue by looping over all patches of all faces that are visible from the shooting face. If we have previously computed that the surface of the shooting patch is *fully* visible to the face of the current receiving patch, we may save *all* geometric occlusion tests (ray clipping) between these two points. If the mutual visibility is partial, each ray test between the center of the shooting patch group and the current receiving patch is evaluated.

Next, we compute the form factor of the current arrangement of patches. In the actual code, this is somewhat optimized by e.g. *not* computing the square roots of certain values, but rather by continuing the computations with the squares which cancel each other out as the computations evolve. However, these purely technical issues are not to be further deepened here.

Finally, we account for absorption by reducing the incoming energy by the diffuse reflectivity of the surface, and add the remaining energy to the receivers total and unradiated energy value.

This closes the cycle: Energy has been radiated from the shooting patch(es) to the receiving patches in the environment. Some of the received energy has been absorbed as heat, which falls out of our system. The remainder is stored as yet unradiated energy at the receiving patches. It waits for re-reflection into the environment until the formerly receiving patches are selected for shooting in a later iteration of the algorithm. The fact that some of the energy has been absorbed to heat also indicates that we actually made progress in converging towards a solution ("light equilibrium"). While we will not ever be able to reach the analytical correct solution with the presented iterative algorithm, we can achieve a very good approximation.

3.3.5 Tone Reproduction

As the dynamic range of the computed radiosity solution by far exceeds what typical output devices can display, we have to include a processing stage that reduces the dynamic range of the computed results. Reducing the dynamic range is a difficult problem, and several methods have been explored in the past.

CaLight employs the histogram based approach described in [LRP97], which I have found to produce very good results. Larson described his method originally for screen-space images, but its extension to world-space light-maps can easily be derived from his work.

3.3.6 Post-Processing the Borders

The borders of the polygons imply special problems on our algorithm. This is especially true since at runtime, the renderer of the 3D engine will typically employ bilinear[3] filtering when sampling the light-maps.

This implies that we do not only have to restrict our texture filtering technique to bilinear filtering and to keep safety distances between all light-maps such that storing many light-maps in a common texture image file works as described in the footnote, but also that we have to account for the fact that during bilinear rendering, light-map elements contribute colors that never received light in the first place. As shown in figure 2, this occurs at polygon edges because bilinear filtering refers to light-maps elements whose patches have never been inside the polygon, and thus never received light.

It is almost impossible to solve these kinds of problems in a mathematically fully satisfactory manner, but I provide algorithms for a two-step approach in CaLight that produce near-perfect solutions in almost all cases: The first step considers all patches of each surface that did not have their sample points inside the surface polygon. For those, the average of their eight surrounding patches is computed. Only those patches contribute whose sample point was inside the polygon though (and thus have valid values from the main algorithm). This is a quick and simple start, as the method only considers one surface at a time and thus the neighbouring faces do not contribute to the result. The second step considers pairs of surfaces which share a common plane and are spatially

[3]Note that anything "better" than bilinear filtering does not work with light-maps: Light-maps do normally have very small dimensions, and therefore we conveniently store many of them in a single texture image file. As bilinear filtering takes the surrounding 2x2 texels into account, we are forced to keep a certain "safety distance" of at least 1 texel between two neighbouring light-maps. However, employing any other filtering technique that employs even bigger filter kernels (like Mip-Mapping, anisotropic filtering etc.) would "melt" individual light-maps together that are independent from each other but happen to be neighbouring in the bigger texture image file.

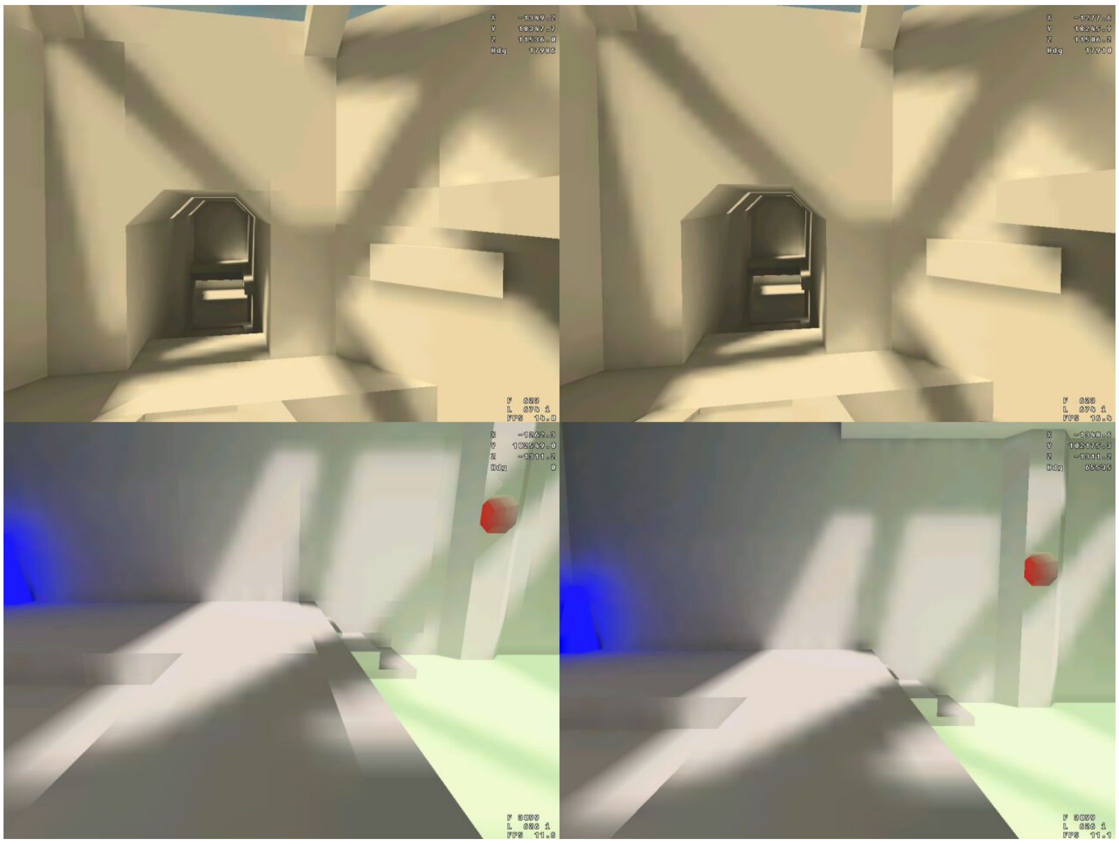


Figure 2: The effect of original (left) versus post-processed light-map borders (right).

located close to each other (i.e. neighbours or near-neighbours). The goal here is to account for high frequency shadows that cast across such neighbouring surfaces. The combination of square overlapping light-maps for irregular polygons, bilinear filtering, and arbitrary edges leads to artifacts that are best demonstrated in the left images of figure 2. Getting this in order is achieved by a careful examination of the patches of two surfaces that are spatially close. Such patches tend to overlap each other, and adapting their values to each other resolves the problem. The adaptation of the appropriate values is made particularly difficult by the facts that the patches have no common world alignment (their overlaps are partial, not tiling), and by the danger of inadvertently introducing a new erroneous effect known as "light bleeding". Light bleeding occurs when two independent rooms are separated through a thin but opaque wall, but light from one can be seen in the other because their patches protrude into the other. The algorithms must carefully make sure not to make mistakes even in such cases. The determination of right and wrong correction attempts makes this part especially delicate.

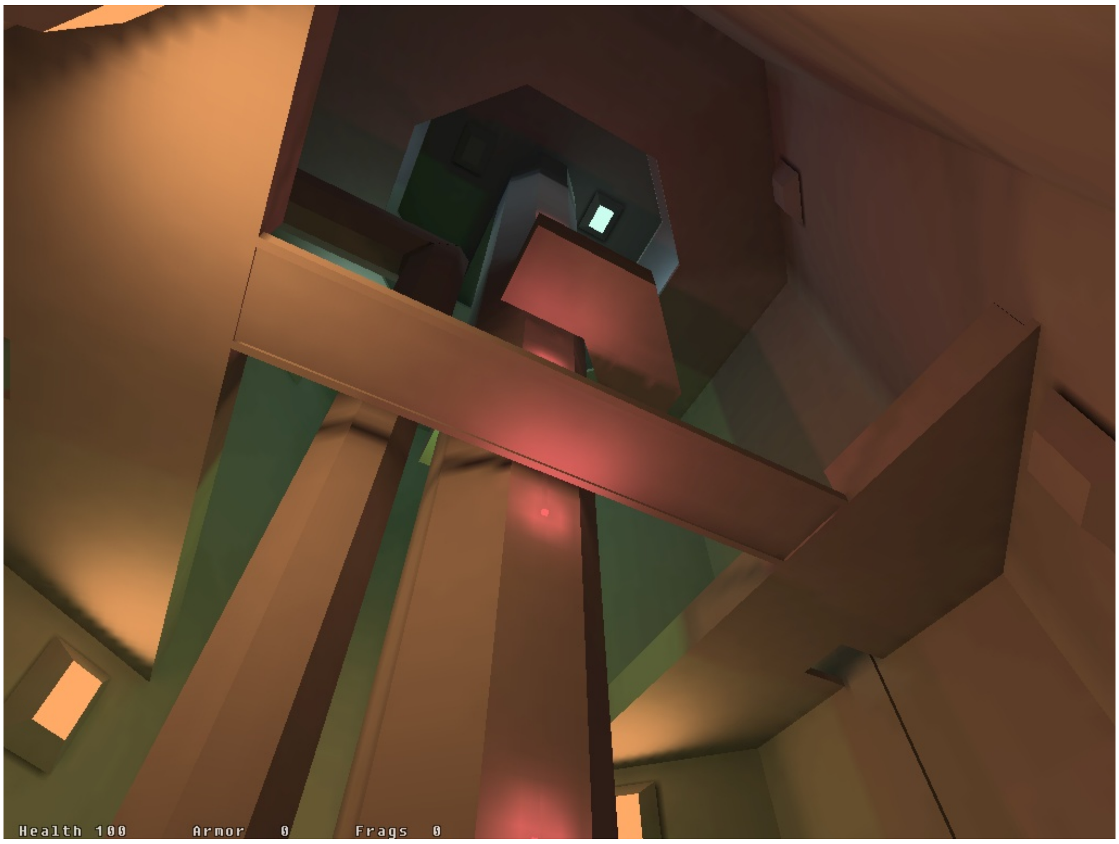


Figure 3(a): A scene with only light-map rendering enabled.

3.4 Rendering Light-Maps

The rendering of the previously computed light-maps is well-known from literature, code samples and tutorials that are available at many websites. It principally reduces to modulating the light-maps with the diffuse textures by simple multiplicative combination.

3.4.1 Results

Figures 3 and 4 show the lighting results obtained with light-map rendering. All pairs of images show the same scene, once with the bare light-maps rendered only, and once the "normal" case, where the light-maps and diffuse textures have been multiplicatively combined.

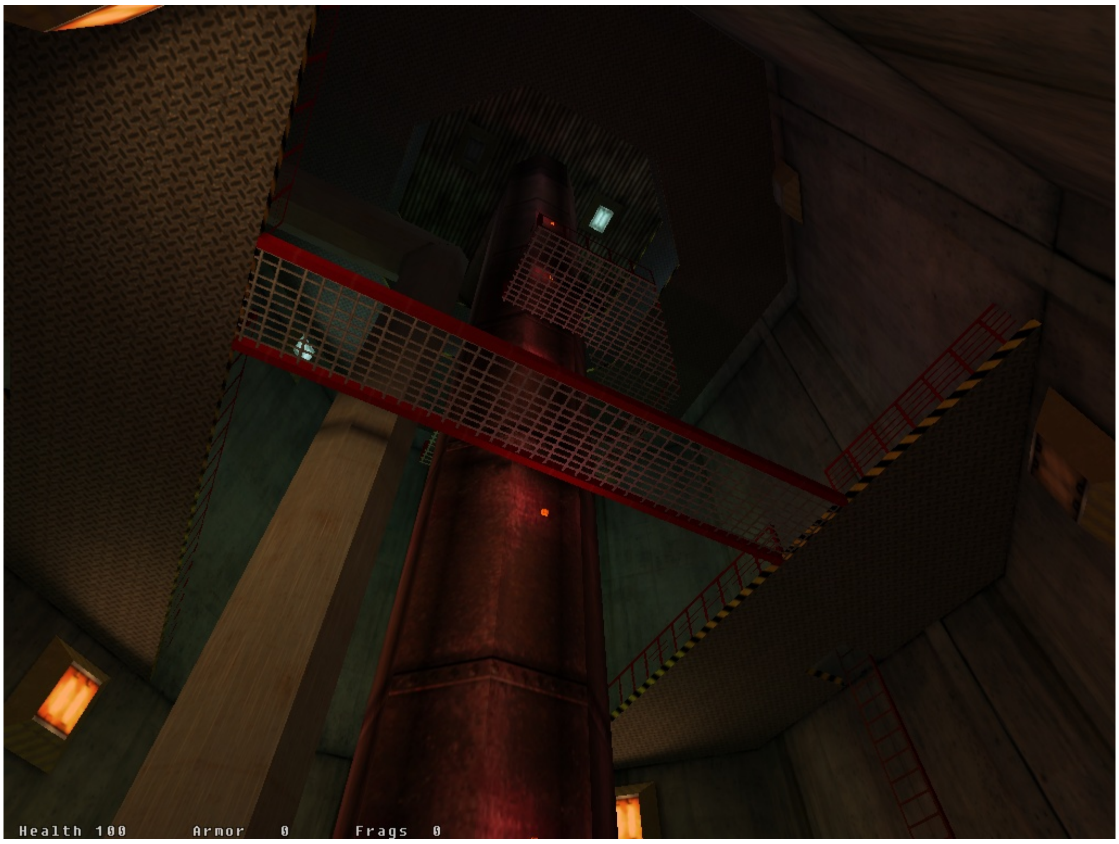
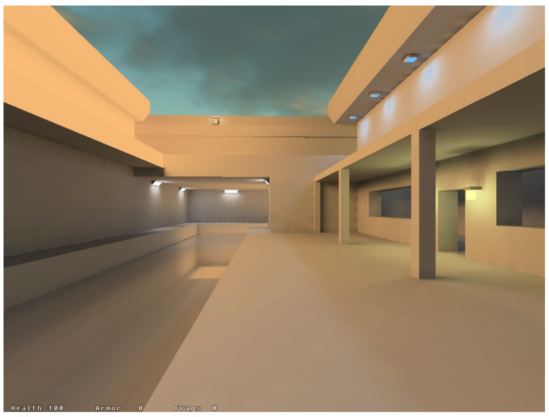
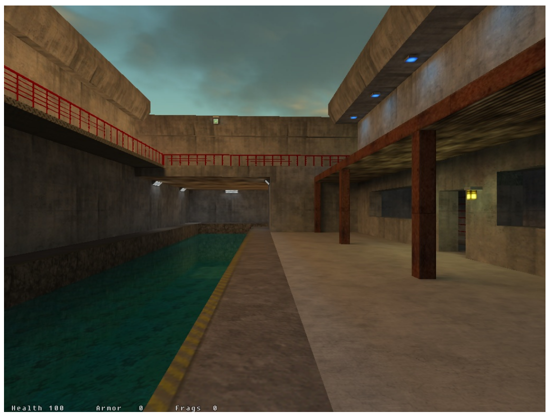


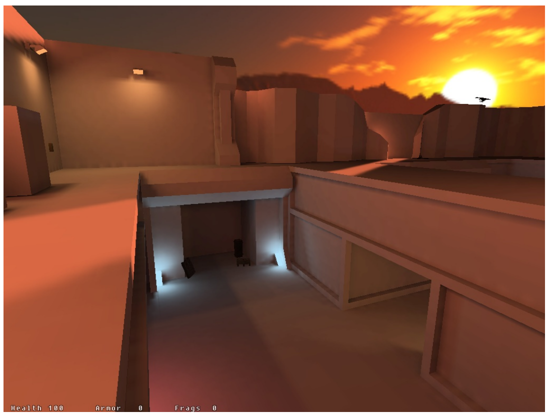
Figure 3(b): The same scene with light-maps and the diffuse textures combined.



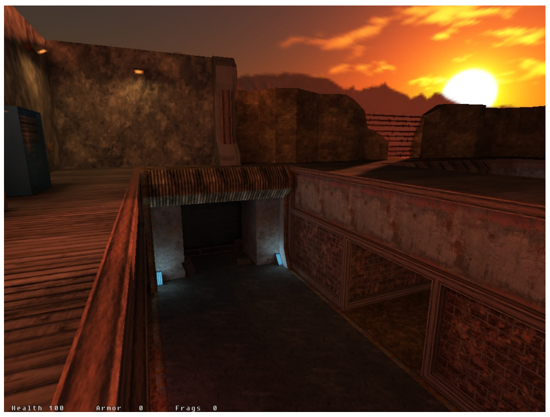
(a)



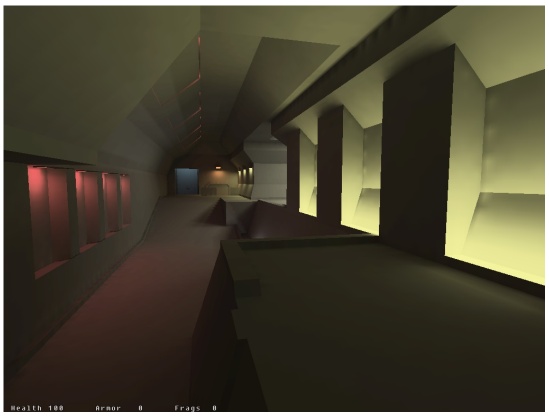
(b)



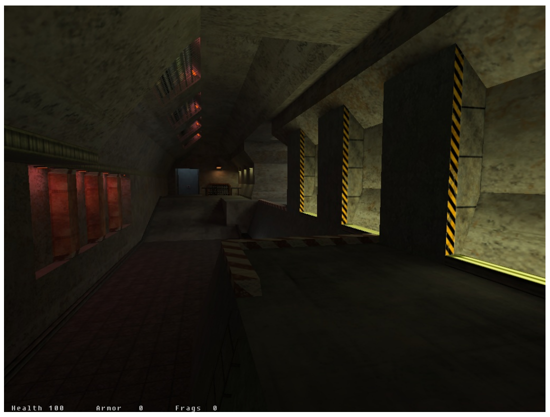
(c)



(d)



(e)



(f)

Figure 4: The images on the left show scenes with only the light-maps rendered. The images on the right show the same scene, respectively, with light-maps and diffuse textures combined.

4 Dynamic Lighting on Dedicated 3D Hardware

Many concepts and ideas in realistic image synthesis, including the proper lighting of scenes, have long been known both in theory and software implementations. Examples include ray tracers like *POV-Ray*, and renderers like Pixar's *RenderMan* which has gained a high profile as a tool in offline movie rendering. However, these implementations have long been unable to produce real-time results.

Advances in hardware technology over the last decade resulted in graphics processing units that are fast enough to allow for implementations of the theory at interactive rates. Such hardware is conveniently available for the consumer mass market, yielding the advances in computer graphics to a broad audience.

The specifics of the hardware technology have also led to new combinations of older ideas and theories, and are setting trends for the close- and midterm future: The hardware is triggering software research and development in order to better exploit it, while the requirements from software development help form the next generations of hardware. For example, while graphics processing units have become almost freely programmable at the time of this writing, they still are much better in *rasterizing* geometry than in ray-tracing it. The mutual dependency manifests itself in the research for and (re-)occurrence of image synthesis techniques that are especially amenable to the new hardware technology, as for example the Phong lighting and shading model or the revival of stencil shadow volumes.

Therefore, one keystone of this thesis was to explore the properties of real-time lighting in the context of contemporary hardware and software. This section describes the theoretical backgrounds of hardware-accelerated lighting (that is, the Phong shading model), and its extensions with several techniques (e.g. shadow volumes).

4.1 The Phong Shading and Illumination Model

For the purposes of hardware-accelerated lighting, it stands to reason to consider the *Phong shading and illumination model*, with variants and extensions where appropriate.

Adopting the terminology of Foley et al. ([FvDFH90]), the term *shading model* refers to the framework within which an *illumination (or lighting) model* is employed. The Phong *shading* model comes close to what is known in the hardware world as *per-pixel* (or *per-fragment*) lighting. That is, some illumination model is applied for each individual rasterization fragment as a consequence from the fact that e.g. the polygon normal vectors are interpolated across the polygon. This is contrary to *Gouraud* shading, where only the final computed color values are interpolated across the polygon area. The fact that we chose per-pixel lighting over per-vertex lighting is thus (almost) equivalent to saying that we employ the Phong rather than the Gouraud shading model for all our purposes. The Phong shading model requires more expensive implementations, but also yields higher quality results, as is detailed in [FvDFH90].

The Phong *illumination* model describes the local illumination of a surface at a given point. That is, contrary to global illumination computations as in CaLight (see section 3), which are much more complex, Phong lighting only considers local effects, and thus can be handled well by the current hardware. Phong lighting is also easy to understand and implement, and therefore the commonly favoured choice of illumination model for implementation on programmable graphics hardware. More complex local illumination models that provide more accurate results can still easily be added in future implementations.

Conceptually, the Phong illumination model was developed out of convenient practical considerations in computer graphics, and it is important to note that everything that is discussed in this regard *has no clear physical interpretation*. For example, material reflection properties and light source colors are described as normalized RGB numbers, that is, value triples in the range from 0.0 to 1.0.

The Phong lighting equation is described in great detail in many publications (e.g. [FvDFH90] or [FK03]), even if with slightly varying terminology. Therefore, I will not repeat its foundations and derivation here, but rather state the Ca3DE lighting equation that I have chosen for further consideration, along with a detailed description of the meaning of its symbols and sub-terms. After that, also the extensions to this technique like stencil shadow volumes are treated.

The complete Ca3DE lighting equation is, at its highest level

$$\begin{aligned} \mathrm{out}_C \quad = \quad & \mathrm{ambientContrib}_C \\ + \quad & \mathrm{emissiveContrib}_C \\ + \quad & \sum_{i=0}^{n-1} \mathrm{diffuseContrib}_C^i \\ + \quad & \sum_{i=0}^{n-1} \mathrm{specularContrib}_C^i \end{aligned} \tag{3}$$

The C subscript in this and the subsequent equations means "color", and the i superscript as in $\mathrm{diffuseContrib}_C^i$ refers to the i-th light source. In words, the lighting equation computes the resulting color as the sum of the ambient and emissive components, plus the diffuse and specular contributions of each light source. The next subsections provide detailed descriptions of each sub-term.

4.1.1 The Diffuse Term

Assuming that there are n relevant light sources in the scene, the diffuse term is the sum of the diffuse contributions of all n light sources:

$$\mathrm{diffuseContrib}_C = \sum_{i=0}^{n-1} \mathrm{atten}(d^i) \cdot \mathrm{diffuse}_C \cdot L_C^i \cdot (\vec{N} \odot \vec{L}_{dir}^i)$$

The meaning of the individual symbols is as follows:

$\mathrm{diffuse}_C$: This is the diffuse reflectivity (diffuse color) of the material expressed as RGB color.

$\text{atten}(d^i)$: The light attenuation. d^i is the distance from the current surface point to the i-th light source. $\text{atten}(d^i)$ is frequently defined in literature as follows:

$$\text{atten}(d^i) = \frac{1}{c_1 + c_2 d^i + c_3 (d^i)^2}$$

c_1, c_2 and c_3 are arbitrary constants that are chosen to yield a "good looking" result. In section 4.5, where the implementation aspects of this equation are discussed, we will see that a slightly different definition of $\text{atten}(d^i)$ is beneficial both for image quality and ease of implementation on all supported graphics APIs and hardware.

L^i_C : The light sources RGB color.

$\vec{N} \odot \vec{L}^i_{dir}$: This term computes the cosine of the angle between the (normalized) direction to the light source L^i, which is denoted as $\vec{L}^i_{dir}$, and the (normalized) surface normal $\vec{N}$ by computing the dot product of the two normalized vectors.

One might be tempted to move the diffuse_C symbol in front of the sum, as it does not depend on i, but see below for a discussion why this is not done.

4.1.2 The Specular Term

Similar to the diffuse term, the specular term is the sum over the contributions of the n light sources:

$$\text{specularContrib}_C = \sum_{i=0}^{n-1} \text{atten}(d^i) \cdot \text{specular}_C \cdot L^i_C \cdot (\vec{N} \odot \vec{H})^s$$

$\text{atten}(d^i)$: This is the attenuation of the i-th light source as defined above.

specular_C : The specular reflectivity of the material expressed as RGB color.

L^i_C : The light sources RGB color. This color may or may not be different from the light sources diffuse color described above. For a "real" light source, it would be the same color as for the diffuse light, but it often yields visually interesting results to assign a different color for the specular highlight here.

$(\vec{N} \odot \vec{H})^s$: The cosine of the angle between the surfaces normal vector $\vec{N}$ and vector $\vec{H}$, raised to the s-th power. $\vec{N}$ is usually obtained from the normal-map of the surface. $\vec{H}$ is the halfway vector between the vector $\vec{L}^i_{dir}$ from the current surface position $\vec{P}$ to the light source position $\vec{L}^i_{pos}$, and the vector $\vec{V}_{dir}$ from the current position to the viewer position $\vec{V}_{pos}$. Thus, if $\text{norm}(\vec{X}) = \frac{\vec{X}}{|\vec{X}|}$ (i.e. norm computes the unit vector of a non-null vector), $\vec{H}$ is defined as follows:

$$\begin{aligned} \vec{H} &= \text{norm}(\text{norm}(\vec{L}^i_{dir}) + \text{norm}(\vec{V}_{dir})) \\ &= \text{norm}(\text{norm}(\vec{L}^i_{pos} - \vec{P}) + \text{norm}(\vec{V}_{pos} - \vec{P})) \end{aligned} \tag{4}$$

4.1.3 The Emissive Term

The emissive term is simply defined as follows:

$$\text{emissiveContrib}_C = \text{luminance}_C$$

That is, the light emissive value is taken from the materials luminance RGB texture-map.

4.1.4 The Ambient Term

Normally, the ambient component of a materials color is formulated as the materials ambient reflection ("color") times the color of the incoming ambient light:

$$\text{ambientContrib}_C = \text{ambient}_C \cdot \text{globalAmbientLight}_C$$

where ambient_C is the ambient reflectance of the material.

The results of this term, however, are uninteresting and of little esthetic value. Therefore, I decided to take the previously computed *global lighting data* as the ambient light contribution (please refer to section 3 for a discussion of global lighting that is computed with radiosity algorithms). While this is not strictly complying to the definition of ambient homogeneous light, it makes perfect sense from a practical point of view. Moreover, I assume that the ambient reflectivity equals the diffuse reflectivity of the material. The ambient term therefore becomes

$$\text{ambientContrib}_C = \text{diffuse}_C \cdot \text{lightmap}_C \cdot N_z$$

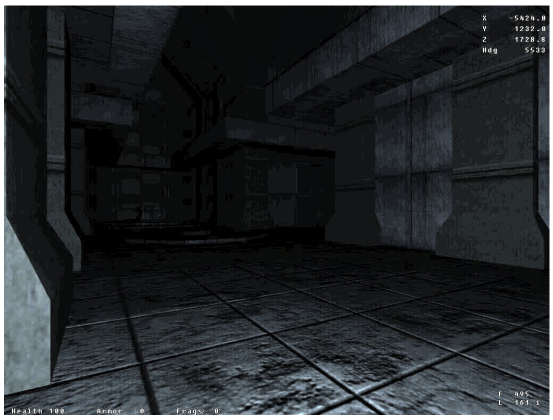
diffuse_C is the materials ambient reflection that is usually obtained from a texture-map lookup. lightmap_C is the materials light-map value that has been computed as in section 3 and is substituted for the global ambient light color.

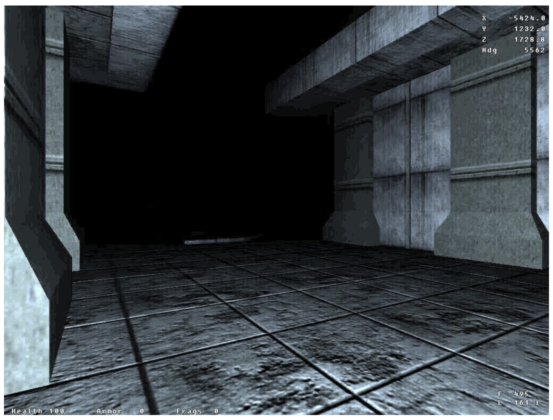
What I have described so far yields exactly the same results as the lighting with radiosity based light-maps alone as described in section 3.

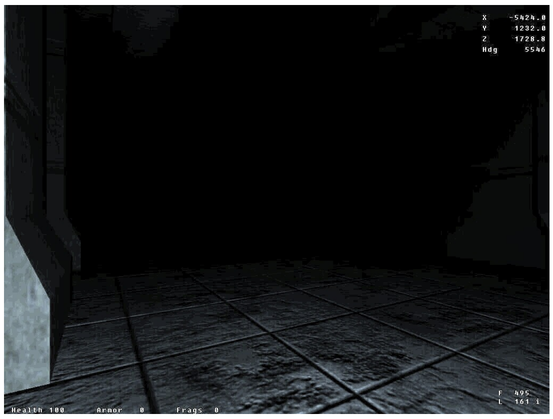
The rationale for the presence of the N_z term, which describes the z-component of the normal vector of the material, is as follows: During my tests and experimentation with the lighting technique that is described in this section, I experienced frequently that the subtle results of the diffuse and specular terms, described below, often went unnoticed or became weakened by the presence of the visually more significant ambient term. I therefore decided to "mix" the so far described ambient term with concepts from the diffuse term: With the ambient light obtained from the light-maps, we have no indication from which direction the light originally arrived. Therefore, let us simply assume that it came from directly "above".

In effect, multiplying the above ambient term with N_z "darkens" the overall result. The net effect is that the diffuse and specular terms become better visible and the spatial effect that they create when normal-maps are used becomes more pronounced.

Please note that computing the ambient term as described above, that is from precomputed light-maps, is in effect an early combination of two vastly different lighting techniques, namely the precomputed, static, radiosity-based light-maps and the hardware-accelerated dynamic lighting. As this approach makes a lot of sense in practise, I did not attempt to artificially separate the two approaches in the implementation.


(a) $\mathrm{atten}_1(d)$


(b) $\mathrm{atten}_2(d)$


(c) $\mathrm{atten}_3(d)$

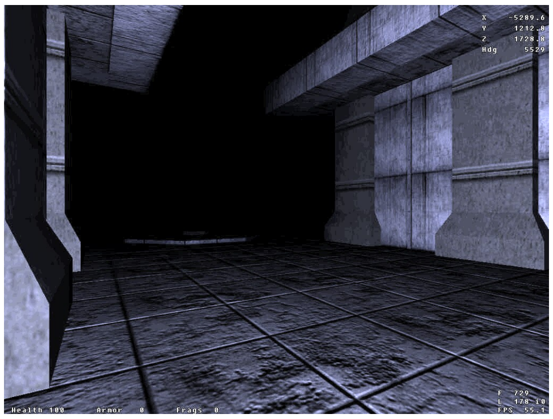
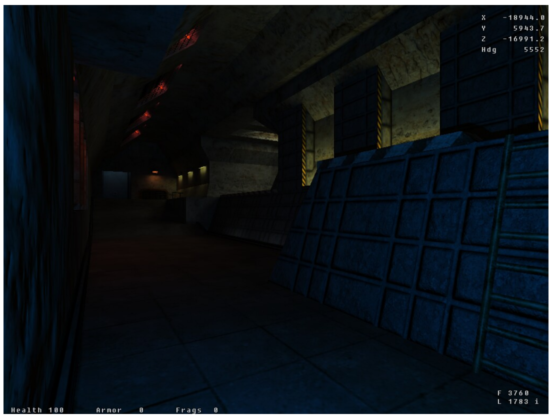

(d) $\mathrm{atten}_4(d)$

Figure 5: Different attenuation functions in comparison. Brightness has been exaggerated in order to display the effect more clearly.

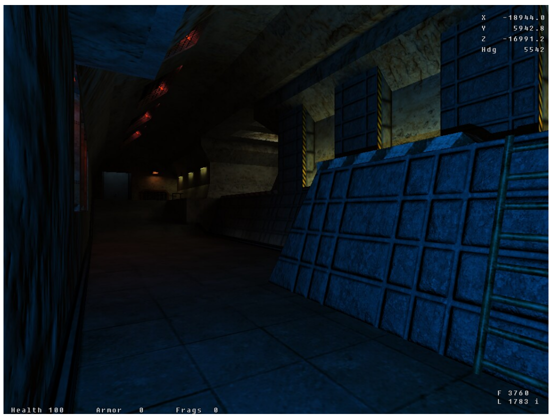
4.1.5 The Attenuation Function

In theoretical treatments of the Phong illumination model, the attenuation function is often defined as $\mathrm{atten}(d) = \frac{1}{a+bd+cd^2}$. d is the distance between the light source and the current surface point, and a, b and c are arbitrary constants. As mentioned before, contrary to radiosity computations for light-maps, there is no physical foundation for such attenuation functions for dynamic lighting.

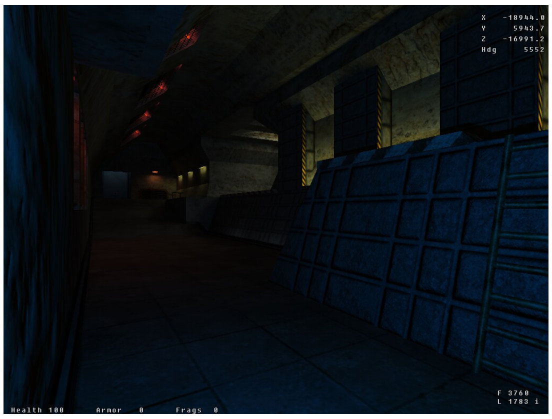
Therefore, my initial attempt was to implement a simple attenuation function of the form $\mathrm{atten}_1(d) = \frac{1}{bd}$, which seemed to look good. b was typically chosen around 0.0005, and the function clamped to not exceed 1.0. The Cg expression that was employed to compute the attenuation was `saturate(2000.0/d-0.01)`. Unfortunately, with this function the light attenuation gets 0 only when d approaches infinity. That means that even very far away surfaces still receive some light from such a light source. This in turn



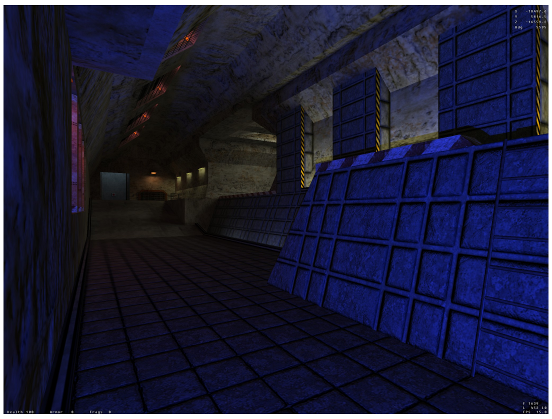
(a) $\text{atten}_1(d)$



(b) $\text{atten}_2(d)$



(c) $\text{atten}_3(d)$

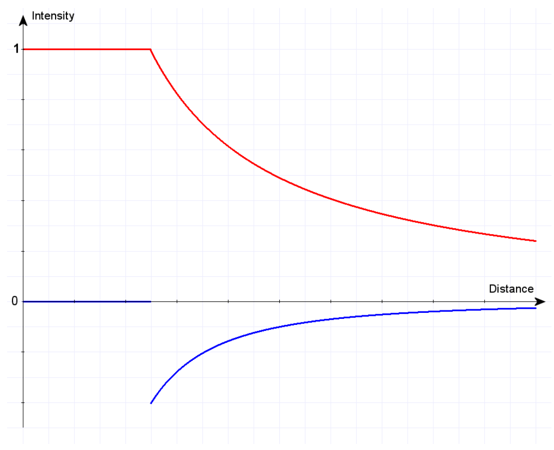


(d) $\text{atten}_4(d)$

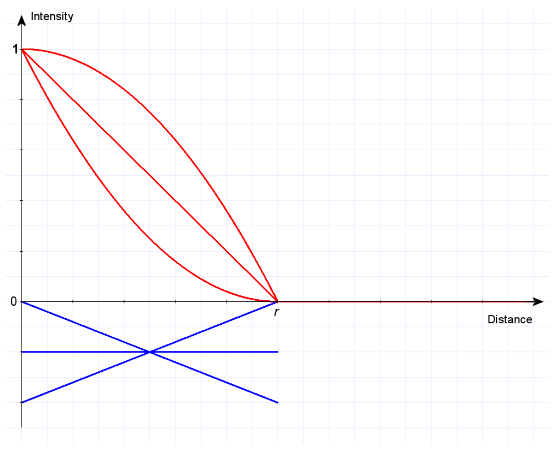
Figure 6: Another set of images, showing the attenuation functions in comparison.

is a big lost opportunity for optimization: If the attenuation reached zero earlier, it was possible to skip the lighting computations for surfaces that do not receive light anyway. The `-0.01` sub-term intends to account for that property, but to no great practical avail.

In a second attempt I therefore introduced a radius or "range" r for each light source. The light sources intensity is defined to be maximum at its center, and to gradually decay outwards such that 0 is reached at the radius. The new attenuation function became $\text{atten}_2(d) = \max(1 - d/r, 0)$, with r being the light source radius. This new definition implies that no surface that is farther away from the light source than r can receive light. Therefore it becomes possible to entirely skip the lighting computations of all surfaces that do not touch the appropriate light sources bounding box. Moreover, such faces also cannot be shadow casters, allowing us to skip all the expensive shadow calculations for such faces. In my practical tests with the NV2X and NV3X Cg profiles this configuration improved the performance considerably. Further implications of the



(a) $\text{atten}_1(d)$



(b) (c) (d) $\text{atten}_{2,3,4}(d)$

Figure 7: Graphs of the four attenuation functions.

new attenuation function are that the decay of the light intensity is no longer reciprocal with respect to d, but linear. The downside is that the abrupt stop of the linear decay at $d = r$ is very noticeable in the final rendered images. This is because $\text{atten}_2(d)$ is G0, but not G1 continuous.

I therefore adapted the attenuation function to $\text{atten}_3(d) = \max((1.0 - d/r)^2, 0)$. This still yields 0 (black) for $d = r$, and yields a softer transition (G1 continuity) at that point. Unfortunately, the resulting images are by far too dark for most practical purposes.

My fourth and final modification of the attenuation function was therefore to define it as $\text{atten}_4(d) = \max(1.0 - (d/r)^2, 0)$. This function has several advantages over the previous attempts:

- Even though there is a "hard", G1-discontinuous transition to black, it is not as visually disturbing as with any of the previous attenuation functions.

- The resulting images get significantly brighter and more contrasted, emphasizing the contribution of the dynamic lighting. This is a welcome side-effect, as the contribution of dynamic lighting often is less noticeable when mixed with other kinds of lighting.

- This attenuation function is the only one among those presented above that can be computed on early generation programmable GPUs, without having to resort to lookups into texture-maps that encode (parts of) the function.

Figure 7 shows all attenuation functions that have been discussed, plotted into graphs together with their respective derivatives. Figures 5 and 6 show the rendering results of the four analyzed attenuation functions in comparison. The four images of figure 5 have been carefully post-processed in order to better demonstrate the effect in printing.

4.2 Summarizing the Lighting Equation

Putting all individual terms together in one big equation, the result looks like this:

$$\begin{aligned}\mathrm{out}_C \quad = \quad & \mathrm{diffuse}_C \cdot \mathrm{lightmap}_C \cdot N_z \\ + \quad & \mathrm{luminance}_C \\ + \quad & \sum_{i=0}^{n-1} \mathrm{atten}(d^i)\Big(\mathrm{diffuse}_C \cdot L^i_C \cdot (\vec{N} \odot \vec{L}^i_{dir}) + \mathrm{specular}_C \cdot L^i_C \cdot (\vec{N} \odot \vec{H})^s\Big)\end{aligned} \qquad (5)$$

One might be tempted to rearrange and refactor the above equation further, for example by moving the diffuse$_C$ and specular$_C$ components in front of the sums. However, equation 5 is in good shape already:

1. It works well with hardware implementation, and especially with the stencil shadow volume technique, which inherently requires a multi-pass rendering approach (one pass per light-source).
2. It works well even on GPUs with significant limits in their execution of fragment programs.
3. A rearranged form of equation 5 tends to exceed the valid range for out$_C$, $[0, 1]$, much sooner than equation 5 as is. Premature and implicit clamping to that range tends to produce unexpected results.

While also equation 5 can easily exceed the range $[0, 1]$ when there are too many or too bright light-sources, this behaviour is inherent to the Phong illumination model. For these reasons, we stay with equation 5 as basis for implementation.

4.3 Stenciled Shadow Volumes

Dynamic lighting with the Phong shading and illumination model can well be augmented with complementary shadow techniques. As has already been mentioned in the section about Previous Work, two main approaches are suitable for combining with dynamic lighting and hardware support: *projective shadow-maps*, and *stenciled shadow volumes.* Each technique has its own strengths and weaknesses. Here is a list of features that I observed about projective shadow-maps:

+ They are geometry independent. Anything that can be rendered can also cast shadows, including non-closed and non-2-manifold or otherwise “broken” polygonal models and polygons with masked (alpha-channel tested) textures.
+ (Fake) soft shadows are well feasible.
− Aliasing and singularity artifacts occur.
− Arbitrary numerous light sources do either require the same number of offscreen buffers, or delicate reuse of one such buffer.

− Not trivially omnidirectional (see [Bra] for solutions).

Here is a list of features that I observed about stenciled shadow volumes:

+ Geometrically and mathematically well founded.

+ Inherently omnidirectional.

+ Stencil-buffer hardware support is available even on very early, non-programmable graphics boards.

− Require geometry to be closed 2-manifolds.

− High fill-rate requirements, and high computational costs to determine the shadow silhouette.

− It is hard to get soft shadows.

Neither technique seems to work naturally well when translucency should be taken into account: An object can either be opaque (and cast shadows), or be transparent like it did not exist at all (and cast no shadows).

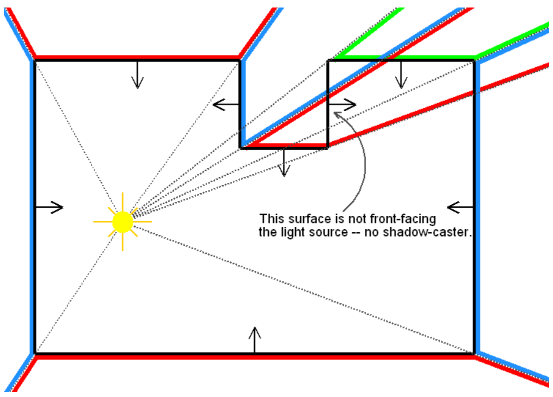
It is hard if not impossible to tell which technique is "better". For best results, they should probably be used in parallel, depending on the requirements of the desired application. For the purposes of this thesis, I've decided to employ the stenciled shadow volume technique as described in [EK02] and [MHE$^+$03].

4.3.1 Shadow Volume Determination in BSP and PVS Models

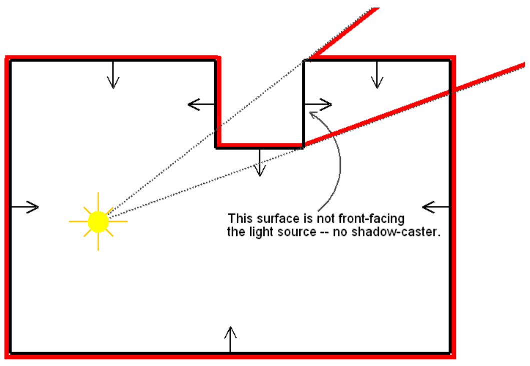
As detailed in section 6.1, the "world" models in the Ca3D-Engine are organized in Binary Space Partitioning trees, and augmented with Potentially Visibility Set data. Moreover, worlds tend to be spatially very large when compared to (the radii of) light sources.

These facts and special-property data structures suggest employing a different approach for shadow silhouette determination than is taken for the usual closed, 2-manifold models as described in [MHE$^+$03]. Actually, the set of surfaces that is potentially visible from a dynamic light source as obtained from precomputed PVS data is typically a loose, highly incoherent, unrelated set. These surfaces often do not share common edges, and are far from being closed 2-manifolds. This means that a new method for silhouette determination is even *required* unless one is willing to ignore the BSP and PVS information of the world models. This is normally a bad idea though, as both the scene complexity as well as the spatial extents are typically enormous, and thus are best managed by BSP and PVS structures. Therefore, three approaches for determining shadow silhouettes for BSP- and PVS-augmented worlds are described:

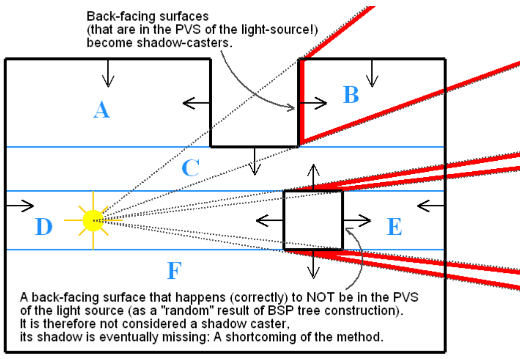
The first, and by far simplest, approach is to simply consider all surfaces that are in the PVS of the current light-source *and* are front-facing the light-source (i.e. the light-source is, with respect to the surfaces planes normal-vector, in front of the plane) as *separate* and *independent* shadow casters. The problem with this approach is that even


This surface is not front-facing
the light source -- no shadow-caster.

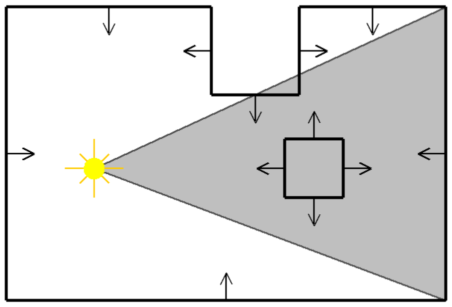
(a) Each polygon that is front-facing and in the PVS of the light-source is a separate shadow caster.


This surface is not front-facing
the light source -- no shadow-caster.

(b) Contrary to (a), neighborhood relationships between polygons are now exploited as is common with "regular" models.


Back-facing surfaces
(that are in the PVS of the light-source!)
become shadow-casters.
A
B
C
D
E
F
A back-facing surface that happens (correctly) to NOT be in the PVS
of the light source (as a "random" result of BSP tree construction).
It is therefore not considered a shadow caster,
its shadow is eventually missing: A shortcoming of the method.

(c) This approach takes the back-facing polygons as shadow casters. A possible BSP subdivision is indicated by blue lines and labels, too.



(d) One of several pyramids (the one for the right-most wall) for making sure that no back-facing shadow caster are overlooked.

Figure 8: Approaches for determining shadow volumes for models with BSP-tree and PVS. The images show a top-down view of a simple room with a light-source. The normal vectors of the wall polygons indicate orientation of the wall polygons. The shadow volumes (that extend to infinity) are indicated by several colors.

though it yields correct shadows, it is highly inefficient: Consider an empty rectangular room with four walls, a floor and a ceiling surface, and a light-source therein. Each of the six polygons would become a shadow caster, even though in fact *no shadows at all* needed to be cast as the room has no door to the outside. Figure 8(a) shows the same principle in a slightly more complicated example: Even though only a tiny fraction of the room is in shadow, *seven* shadow volumes must be processed (or even *thirteen* when also counting the floor and the ceiling at three convex polygons each).

The self-suggesting next step is to augment the BSP data structures to contain neighborhood relationships for the surfaces, similar as is required for “regular” models. This allows us to significantly cut the number of polygons that bound the sides of shadow volumes, as is shown in figure 8(b). However, neither the near nor the far “caps” of the shadow volumes are reduced in their number or importance.

I therefore modified the overall approach to still consider the surfaces that are in the PVS of the current light-source as shadow casters, but now only the subset that is *back-facing* rather than front-facing the light-source. This strategy immediately solves the problem with the empty room: As all six surfaces are front-facing the light source, none is back-facing, and thus no shadow silhouettes are created at all. Figure 8(c) shows the room as with the first two strategies. It becomes immediately clear that the shadow for the upper pillar is the same as with the previous approaches, but this time with only a minimum of shadow volume polygons!

While one can argue about the preference for taking the front- or back-facing polygons as shadow casters for regular models, I strongly expect that “world” models behave differently from regular models, namely as presented in the discussion above: They are huge, both the light-sources and observers are “inside” them, they are preprocessed with BSP and PVS information, and the PVS of any given location tends to contain much more front-facing than back-facing surfaces.

This in turn makes the third approach as presented in figure 8(c) the clear favourite for determining the shadow volume for such worlds.

Unfortunately, the final approach also has a weakness: In some cases, it may happen that a back-facing surface that should cast a shadow is not listed in the PVS of the light-source. This situation is illustrated with the additional pillar in the center of the room in figure 8(c): An arbitrary but correct subdivision has been chosen in order to precompute the BSP tree for the room. For the PVS data, it turned out during PVS computations that leaf **D** can see into all other leaves except for leaf **E**. As all surfaces get assigned to those leaves that they intersect, we say that a surface is visible from a BSP leaf $\mathbf{L}_1$ if it is in a leaf $\mathbf{L}_2$ that is visible from (that is, in the PVS of) $\mathbf{L}_1$. Therefore, leaf **E** (including the indicated surface!) ends not being in the PVS of leaf **D**, whereas *all other surfaces* in figure 8(c) touch or intersect at least one additional leaf that *is* in the PVS of **D**.

Thus, algorithmic details may lead correctly to the inclusion of the other back-facing surfaces that are shown as shadow-casters in figure 8(c), but may well omit the fourth indicated surface. This behaviour results in missing shadows.

Solutions to this problem are not trivial. The best strategy that I have found is to create a second data structure that complements the PVS, namely a “Potential Shadow-Casters Set”.

1. This set is initialized by copying the back-facing polygons of the PVS into it.

2. Then, create pyramids whose apex is the point of the light source, and whose base is a polygon of the front-facing polygons in the PVS, respectively. In the exemplary room in the figures, the pyramid base would be the same polygons that form the near cap of the shadow volumes in figure 8(a).

3. Loop through *all* back-facing polygons in the world (not only those in the PVS), and add those that intersect any pyramid to the set of potential shadow-casters (if not already added in the first step).

Figure 8(d) shows how this algorithm adds the missing back-facing polygon to the set of shadow-casters. This solves the problem.

Finally, the "Potential Shadow-Casters Set" should probably not be computed at rendering time, but rather precomputed to form a lookup table to be used during rendering. Note that precomputing this set is slightly more complicated than the algorithm description above suggests: As a consequence of the fact that the PVS data is stored not per-point, but rather per BSP-tree leaf (cell), and the position of the light-source cannot precisely be known ahead, the computations must assume that the light-source is *anywhere* in a leaf. This turns the pyramids into more complex convex polyhedra, the algorithmic principle however remains the same. If known ahead, the algorithm may even take the light source radii into account in order to minimize the set.

4.4 Rendering Paths for Complex Scenes

Rendering complex scenes with dynamic, per-pixel lighting, accurate shadows, reflective and translucent surfaces, and many other subtle effects *correctly* can be a very difficult problem in rasterizing (compared to ray-casting) rendering systems.

Before this thesis was begun and dynamic lighting and shadows added to the Ca3D-Engine, the old rendering path basically comprised a single rendering pass for each polygon, as only ambient light ("diffuse-map times light-map" as defined above) existed. The old rendering sequence therefore was like this:

1. Render all entity models. These are all the detail models that are nonstructural. Only models in the Potential Visibility Set of the viewer were rendered, and all models were fully opaque. Due to z-buffering and testing, no explicit depth sorting of the models was required.

2. Render the opaque surfaces of the world. These are the structural surfaces of the walls, floors and ceilings. They are stored in the BSP that has been created by the compile tools. These surfaces are rendered front-to-back. This maximizes the utilization of the z-testing and thus improves performance. The proper sorting of the surfaces can easily and essentially for free be obtained by the proper traversal of the BSP tree.

3. Render the translucent surfaces of the world back-to-front. Once more, the sorting is easily obtained by the proper traversal of the binary tree. This time the back-to-front order is mandatory in order to get both the alpha-blended translucency and the depth-buffer right.

4. Render the head-up-displays, particles, and anything else.

In this sequence, the first two steps should have been in reverse order, but as steps 2 and 3 have been in one common function, this order was adopted. This sequence renders the scenes entirely correct *except* for the particles, which had to be sorted into the BSP tree in order to get them right in all cases, too. Another solution for getting the z-order of particles right is presented in my Fortgeschrittenenpraktikum [Fuc03].

Let us now consider the addition of dynamic lighting and stencil shadow volumes: For the high-level rendering code, the most important aspect is that stencil shadow volumes are inherently multi-pass. As shadow volumes are global to a scene, the entire scene is taken into account for determining the shadows for each light source, rather than only individual polygons. Moreover, there is only one stencil-buffer available in practically all rendering systems and APIs. Only if there were as many independent stencil buffers as there are light sources in the scene would it be possible to combine multiple passes into fewer. In other words, employing the shadow volumes technique with stencil buffers means that the number of rendering passes is proportional to the number of light sources, and it is impossible to collapse the contribution of several light sources into a single pass.

The new high-level Ca3DE rendering path is therefore as follows:

1. Of all opaque surfaces, render the ambient and emissive contribution, and basically everything else that is independent of any dynamic light source (e.g. environment-mapped reflections do usually work without light-source, too).

2. Render the light-source independent terms also for entity models. These first two steps are identical with the old rendering path above.

3. Render the per-light-source contribution. That is, loop over all light-sources, and for each one:

    a) Render the stencil shadow volumes for the world surfaces (as determined in section 4.3.1).
    b) Render the stencil shadow volumes for entity detail models (e.g. as described in [MHE+03] or [EK03]).
    c) Render the per-light-source contribution (bump-maps, specular-maps, . . .) of world surfaces.
    d) Render the per-light-source contribution of the entity detail models.

The problem with this rendering path is that it does not handle translucent polygons. The problems that are associated with combining translucency with the above outlined rendering path are complex: Translucent surfaces require a strict back-to-front rendering, whereas the multiple passes that are required for stencil shadow volumes imply a different order and inherently modify the z-Buffer in a way that collides with the demands of back-to-front rendering.

Eventually, I have only found a compromise to overcome this problem: It seems that stencil shadow volumes and rendering translucency cannot be combined in a wholly satisfactory manner. Instead, I have decided to initially omit translucency issues from the above rendering path. This implies that the light of dynamic light sources is cast *through* (the omitted or nonexisting) translucent polygons. Although one would normally expect that a translucent surface (e.g. color tinted glass) modulates or somehow modifies light that shines through it, omitting this is a more pleasing effect than blocking the light like from a opaque polygon. Rather, I *entirely repeat the above rendering path a second time* (with slight modifications), this time taking only the translucent polygons into account. This does still not yield fully correct results when multiple translucent surfaces are layered behind each other, e.g. because *only the nearest* translucent polygon can receive per-light-source contributions. This is hardly ever noticeable in practise, and is by far the best feasible solution to the problem that I have found. However, due to z-Buffer issues it seems that there is *no way at all* to get even those final problems resolved.

4.5 Rendering Dynamic Lighting

Rendering the above presented variant of the Phong shading model for hardware-accelerated lighting is straightforward for most programmable GPUs and their APIs (i.e. OpenGL extensions). Implementations have been provided for the NVidia NV2X profiles of the Cg shading language, the NV3X profiles of Cg, and ATI Radeon 8500 and higher GPUs via the `GL_EXT_vertex_shader` and `GL_ATI_fragment_shader` OpenGL extensions.

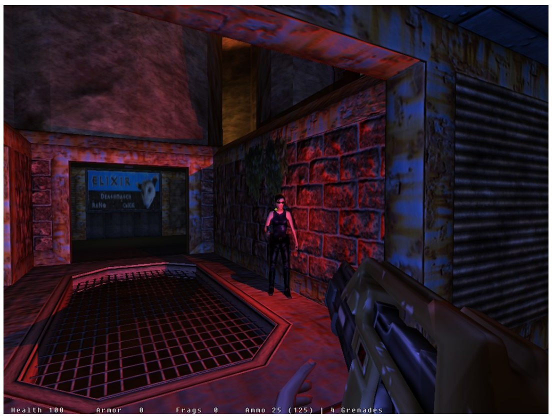
The biggest problems and most important considerations for rendering are performance and software design aspects. In fact it has turned out that the software design has become such a fundamental issue that I wrote an entirely new Material System in order to handle the complexity and achieve the desired flexibility that is inherent to and comes with dynamic lighting in practical applications. Section 6.5 contains additional information about the Material System.

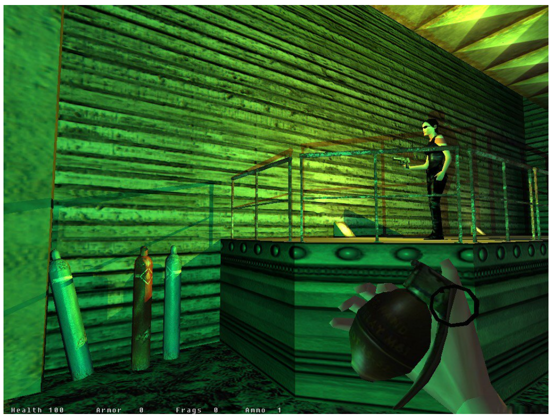
A strategically important consideration for performance is the fact that not all materials come with diffuse-, normal-, specular-, luminance- *and* light-maps. Normally, one or more of these contributor maps are missing, and as such, the relevant term of the Phong lighting equation disappears. For performance, the question is whether or not missing maps should be replaced by default maps that produce the same result as if the appropriate term was properly cancelled from the entire equation, or if specific Cg shaders should be written to handle each specific case. I have created comparative tests in order to answer this question, the results of which are given below.

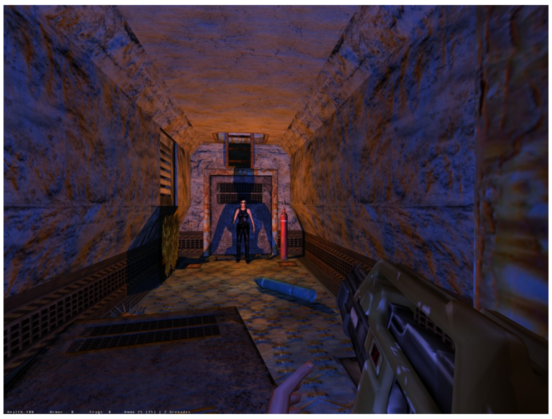
4.5.1 Results

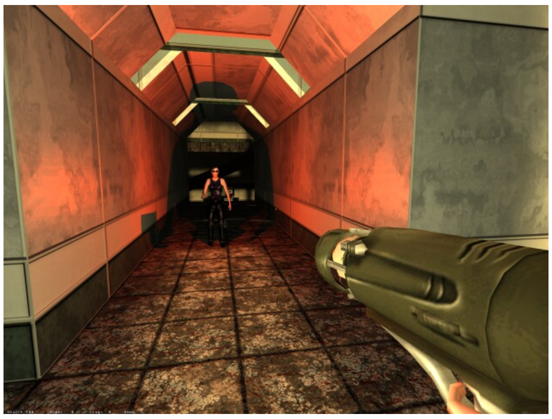
Figure 9 presents several screenshots of dynamic lighting, augmented with light-map lighting as described in section 3, as implemented in the Ca3D-Engine. Normal-map and specular contributions are well visible, as are the shadows by stencil shadow volumes.

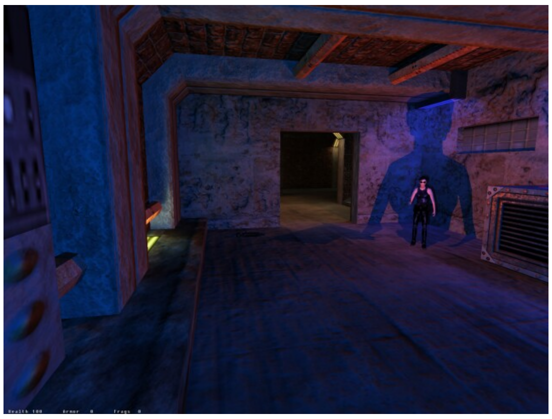
The performance that was achieved when these images were taken depended much on the underlying hardware, the details of the implementation, and possible quality versus speed trade-offs. Together with the exploiting of Potentially Visibility Sets, and the influence of stencil-shadows, the frame-rate also varied widely with the amount and arrangement of the contents of the individual scenes. Nonetheless, I was able to obtain interactive (20+ FPS) or at least near-interactive (10+ FPS) frame-rates in almost all


(a)


(b)


(c)


(d)


(e)

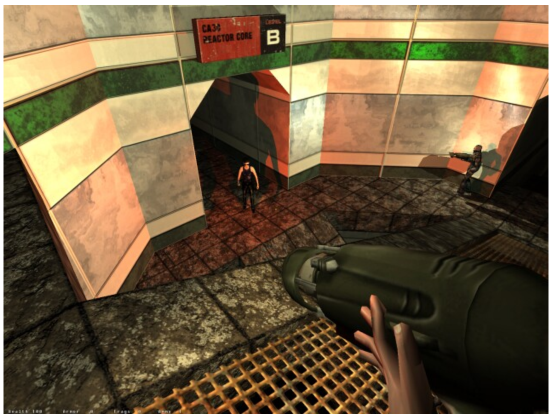

(f)

Figure 9: Rendering results of dynamic lighting with stencil shadows.

	PVS enabled		**PVS disabled**	
World	**NV2X**	**NV3X**	**NV2X**	**NV3X**
BpWxBeta	28	7,4	14	4,2
ReNoEcho	29	7,7	21	5,9
ReNoElixir	19	5,1	12	3,8

Table 1: Performance in various settings where black 1x1 maps were substituted for missing specular-maps.

	PVS enabled		**PVS disabled**	
World	**NV2X**	**NV3X**	**NV2X**	**NV3X**
BpWxBeta	36	9,0	18	4,9
ReNoEcho	37	9,1	27	7,0
ReNoElixir	27	6,5	16	4,6

Table 2: Performance in various settings where special-case fragment shaders were employed in order to account for missing specular-maps.

cases and test scenes with all GPU-specific renderers that I wrote support for (NVidias NV2X and NV3X GPUs, and ATIs R2XX GPUs).

The above mentioned question whether missing contributor maps should be replaced with default maps or rather if special-case fragment programs should be employed was started with a measurement of frame-rates at the starting points of three selected sample worlds. Table 1 shows the results that were achieved when missing specular-maps were replaced with a black 1x1 map, and missing luminance-maps were replaced with a black map, too. The table shows values both for NV2X and NV3X profiles (separate GPU programs specific to each profile), and with the Potentially Visibility Set both activated and deactivated for better expressiveness.

For table 2, missing specular-maps were not any longer substituted by black 1x1 maps, but special-case fragment shaders were employed. This helps performance especially in the NV2X profiles, as their limited capabilities require a separate rendering pass for the specular contribution anyway, which can simply be completely omitted when there is no specular-map present. The performance with the more powerful NV3X profiles can also be observed, although less significantly.

It should be noted in this regard that while multipurpose fragment shaders execute more slowly than their special-case pendants when missing maps have been substituted with default maps, there is also a cost for switching such shaders. It has turned out that binding new shaders can occur very often when the special-case method is employed, and that such bindings are in fact very expensive (that is, time consuming). The balance between both above compared methods is thus more delicate than the numbers in tables 1 and 2 might initially suggest. If all polygon meshes that occur in a scene could be pre-sorted such that they can be rendered by shader order, re-bindings of shaders would be minimized, and the special-case shader method would certainly achieve highest performance. If however the contrary is the case, that is, every mesh requires a shader re-binding, the binding costs might exceed the overhead of a single multipurpose shader than runs with some default-maps on every second mesh. Similar considerations apply to re-bindings of texture-map objects.

In summary, the performance of dynamic lighting on dedicated 3D hardware is highly dependent on external factors. For future research, I'd suggest a test scenario that distinguishes three cases: Meshes pre-sorted by special-case shaders (this probably yields the highest performance), a mesh order that requires shader re-bindings on every rendered mesh, and the same mesh order but with a single multipurpose shader (does not require any re-bindings).

5 Lighting with Spherical Harmonics

One of the key goals of this thesis has been to implement and analyze *Spherical Harmonic Lighting*, and to compare it with the other lighting methods. Thereby, SHL was to be applied not to a dedicated, highly tessellated model, but rather to a reasonably low-polygonal world as it is still common in Ca3DE and many other contemporary applications. This in turn required me to leave the previously described paths and approaches of earlier SHL papers by other researchers, and instead to meet the new requirements as detailed in this section.

5.1 Review of Spherical Harmonics

Spherical Harmonic Lighting (SHL) was first introduced by Jan Kautz, Peter-Pike Sloan, and John Snyder in [SKS02]. Robin Green wrote a subsequent report ([Gre03]) that also provides a view on the mathematical foundations.

The key idea behind SHL is straightforward and simple: We combine descriptions of (dynamic) light sources with descriptions of (static, precomputed) surface properties at the desired points on the surfaces, such that the final illumination at the selected points can be obtained.

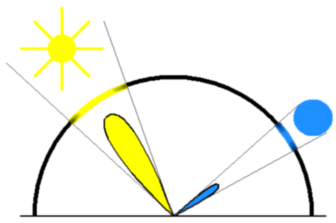


Figure 10: Two plots of a luminance function of two light sources.

In order to achieve this, let us first consider the light sources. All light sources are considered to be at an infinitely large distance, which is an inherent property (and in a sense a restriction) of this lighting technique. Typical examples for such light sources are the celestial bodies like suns and moons, and basically all other light emitters that are located near the sky dome and therefore qualify themselves for being at infinity for all practical purposes. An imaginary observer at a given viewpoint can then obtain a "description" of all light sources by determining for each view direction the incident illumination from the sky dome. The resulting "description" is in the form of a spherical function. Figure 10 shows an example for light sources that are represented by a spherical function. Note that there is only a *single* spherical function that is plotted *twice*: Once as the sky domes spherical arc whose function value is encoded in the hue (color) of the plot, and once as a plot where the spherical function values are expressed as the radii of the plot. Such spherical functions that describe the illumination of the sky dome are called *luminance functions*.

Now let us consider the scene. For each surface point in a scene, we store a description (once more in form of a spherical function) of the "potential incident illumination". That is, the spherical function of a surfaces point P is to answer the question *"Given incident illumination from a direction ω, how much of it does P (diffusely) reflect?"*. It is clear that the answer to this question (and thus the spherical function) can be very complex. Even when we restrict ourselves to purely diffuse reflection (versus taking e.g. BRDFs info account), incident light from ω may or may not be occluded by other geometry, and it may or may not have been reflected from other geometry to reach P. Such spherical functions are called *transfer functions*. Creating transfer functions properly is one of the most important issues of this thesis, and will be discussed in greater detail below.

In figure 11, an arbitrary point P of a scene has been chosen and its transfer function has been visualized. Note that the transfer function in the figure takes both the boolean information "can / cannot be seen by the sky" and the cosine of the incident angle into account, but not (yet) any reflections from the surrounding environment.

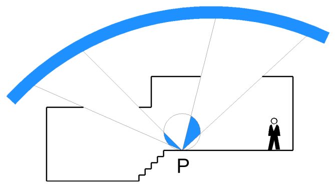


Figure 11: An exemplary transfer function (plotted light blue, as the sky dome) of a point P of a scene.

Finally, when it comes to rendering the scene with a set of light sources, we are given a luminance function (which might have been created in real-time, 'on-the-fly'), and a (usually precomputed) transfer function for the currently rendered fragment at surface point P. Both functions are then combined by modulating (i.e. "multiplying") them, and the result is integrated over P's entire hemisphere in order to obtain the illumination at P.

The special property that makes this approach outstanding and principally (or at least reasonably, in real-time) possible, is the way in which the spherical functions are stored and dealt with: There is *no* explicit storage (e.g. a tabulation of the values by azimuth and elevation), but rather a decomposition into a set of Spherical Harmonics, for which only a very limited set of coefficients has to be stored. Given sets of Spherical Harmonic coefficients, the truly significant and astonishing result is that even the above mentioned multiplication and integration can be reduced to a mere dot-product like operation.

The next sections will discuss the related backgrounds in greater depth.

5.1.1 Orthogonal Basis Functions

Basis functions $B_i(x)$ are small pieces of signal that can be scaled and combined to produce an approximation of an original function $f(x)$. The "scale factors" for each basis function B_i are scalar coefficient values c_i that describe how much the original function $f(x)$ is like the basis function $B_i(x)$. The process of determining the proper coefficient c_i for each basis function $B_i(x)$ is called *projection*. Projection is achieved by integrating the product $f(x)B_i(x)$ over the full domain of f for each basis function B_i, yielding a vector of the desired scalar values:

$$c_i = \int f(x)B_i(x)\ dx$$

Figure 12 shows an example of this process for the first three coefficients. Having obtained a vector of coefficients $(c_1, c_2, c_3, \dots)$ this way, the process may be reversed by scaling the basis functions B_i with their related coefficients c_i, and summing the results. This yields an approximation of the original function, as is shown in figure 13.

While the basis functions in the above example have been linear, the most interesting basis functions are grouped into families of functions that are called *orthogonal polynomials*. The integral of the product of two orthogonal polynomials either yields 0 if they are different from each other, or a constant value if they are the same. A more

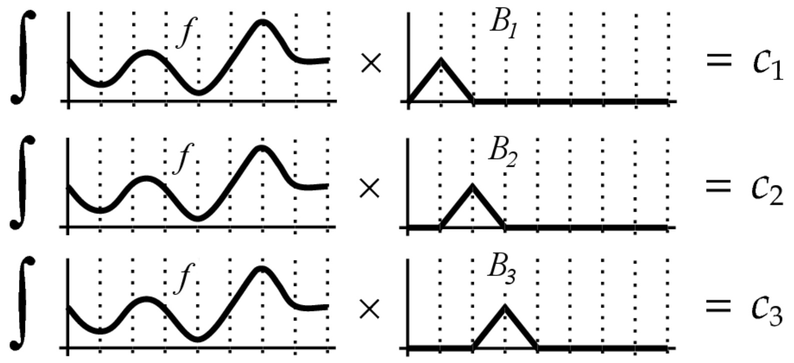


Figure 12: An example for computing the first three projection coefficients. Base image with permission and by courtesy of Robin Green.

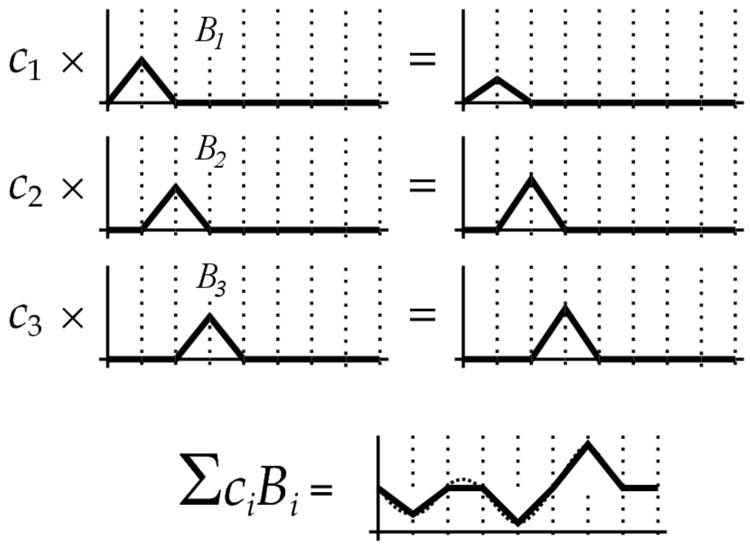


Figure 13: Reversing projection in order to obtain an approximation of the original function. Base image with permission and by courtesy of Robin Green.

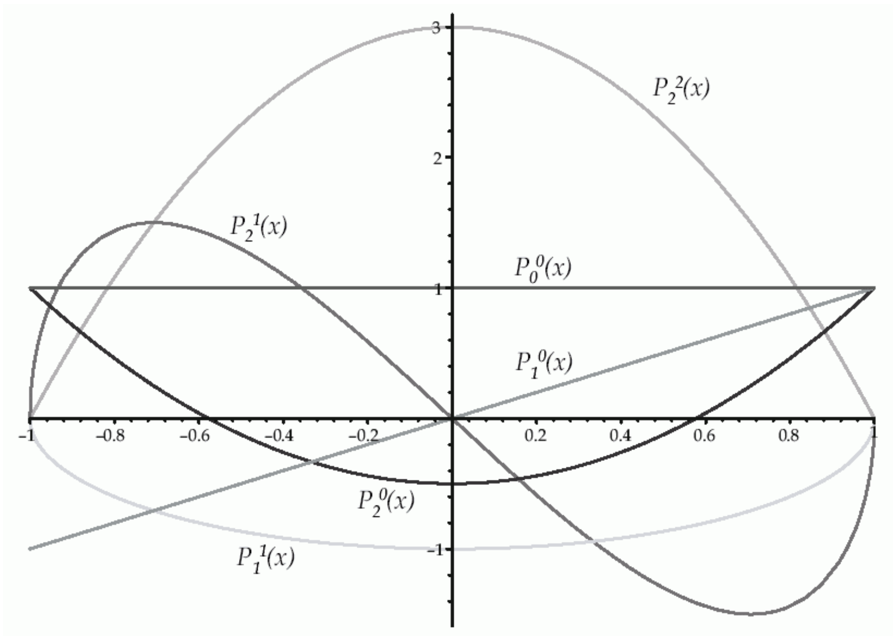


Figure 14: The first six Associated Legendre Polynomials. Image with permission and by courtesy of Robin Green.

rigorous definition requires that the product yields either 0 or 1, respectively, and the corresponding subfamily of functions is called the *orthonormal basis functions*.

Among those, we are interested in the *Associated Legendre Polynomials*, which are conveniently defined recursively:

$$\begin{aligned}(l-m)P_l^m &= x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m \\ P_m^m &= (-1)^m(2m-1)!!(1-x^2)^{m/2} \\ P_{m+1}^m &= x(2m+1)P_m^m\end{aligned}$$

This definition is also well amenable to implementation issues. Plots of the first six Associated Legendre Polynomials are shown in figure 14.

5.1.2 Basis Functions for the Spherical Case

While the previous section was about decomposing one-dimensional functions, we now want to extend the concept to spherical functions.

Basis functions for spherical functions can be obtained from the previously mentioned Associated Legendre Polynomials. Such functions are called *Spherical Harmonics*. While these are generally defined on complex numbers, we are only interested in real-numbered functions over the sphere (e.g. light or transfer functions), and therefore we only consider the definition of *Real Spherical Harmonics*, which are usually denoted with the symbol y:

$$y_l^m(\theta,\phi) = \begin{cases} \sqrt{2}K_l^m \cos(m\phi)P_l^m(\cos(\theta)) & \text{if } m > 0 \\ \sqrt{2}K_l^m \sin(-m\phi)P_l^{-m}(\cos(\theta)) & \text{if } m < 0 \\ K_l^0 P_l^0(\cos(\theta)) & \text{if } m = 0 \end{cases}$$

Here, P denotes the same Associated Legendre Polynomials as above, and K is a scaling factor defined as

$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}}$$

Given that, the projection of Spherical Harmonics works analogous to the projection for the 1D case:

$$c_l^m = \int\limits_S f(s)y_l^m(s)\ ds \tag{6}$$

5.1.3 From Samples to SH Coefficients

With the generic derivations presented in the previous sections, what remains is to find a numerical method for obtaining representative SH coefficients for a given spherical function $f(\theta,\phi)$ that is defined in polar coordinates.

Rewriting equation 6 in terms of f yields

$$c_l^m = \int\limits_0^{2\pi}\int\limits_0^{\pi} (f(\theta,\phi)y_l^m(\theta,\phi)) \sin\theta\ d\theta\ d\phi \tag{7}$$

However, this does not help much with numerical computation.

Now let us assume that f is represented as a set of N discrete samples x_i ($1 \le i \le N$), where $f(x_i)$ is the value of the i-th (θ,ϕ) pair (or look-up key). As [Gre03] has well demonstrated, we may apply the generic Monte Carlo estimator

$$\int g(x) \approx 1/N \sum_{i=1}^{N} g(x_i)w(x_i) \tag{8}$$

to equation 6, with w being the weighting function. As $w(x_i) = 4\pi$ for all x_i, we obtain

$$\begin{aligned} c_l^m &\approx 1/N \sum_{i=1}^{N} f(x_i)\, y_l^m(x_i)\, 4\pi \\ &= \frac{4\pi}{N}\sum_{i=1}^{N} f(x_i)\, y_l^m(x_i) \end{aligned}$$

5.2 Computing Transfer Functions

Now that we have seen how arbitrary spherical functions can be decomposed into SH coefficients, let us consider how transfer functions are created and how their (final) SH coefficients are obtained. We will find that for transfer functions that take no self transfer (bounce or *interreflected* light) into account, a straightforward approach to computing the SH coefficients is feasible. For the case of also dealing with self transfer, the property of SH projection that is described by equation 11 will be exploited to significantly simplify the computations.

5.2.1 Diffuse, Self-Shadowed Transfer Functions

In order to compute our first transfer functions, we reduce our considerations to flat surfaces that only receive direct lighting and reflect that light in a purely diffuse manner. That means that no matter from which direction a point on the surface is receiving light, the light is reflected equally in all directions.

With these assumptions, the generic rendering equation 1 can be significantly simplified and becomes:

$$L(x') = \int_S L(x, \vec{\omega}) \, \frac{\rho_x}{\pi} \max(N_x \odot \vec{\omega}, 0) \; dA \tag{9}$$

Our transfer function is described by the sub-term

$$\frac{\rho_P}{\pi} \max(N_P \odot \vec{\omega}, 0) \tag{10}$$

of this equation, writing P for the position now rather than x. It is determined as follows:

Initially, we create a set of uniformely distributed sampling normal vectors. An example for such a set of samples is shown in figure 15(a).

One can imagine that this set of samples is then "moved" to the surface point P at which the transfer function is to be computed (see figure 15(b)). At P, the transfer function 10 is evaluated by computing a value for each sampling normal vector. The obtained numbers are another, intermediate representation of the transfer function. They are shown as the length of the sample normal vectors in figures 15(c) and 15(d), where the "missing" ones have value (or length) 0. Compared to 15(c), figure 15(d) extends transfer function 10 by also taking the geometric visibility for self-shadowing into account.

The set of sample directions (figure 15(a)) is not tied to a specific surface point P, and it can be re-used for the next point of a surface.

Computing the SH coefficients for transfer functions is typically achieved by first computing the transfer functions as sets of samples as just outlined, and then turning the samples into SH coefficients as described in section 5.1.3. For the more complex case of bounce transfer, we will see that working with the samples during the bounce transfer computations is very expensive, as the intermediate computations contain a factor of O(number of samples). Property 11 below will show how we can reduce this factor to O(number of SH coeffs).

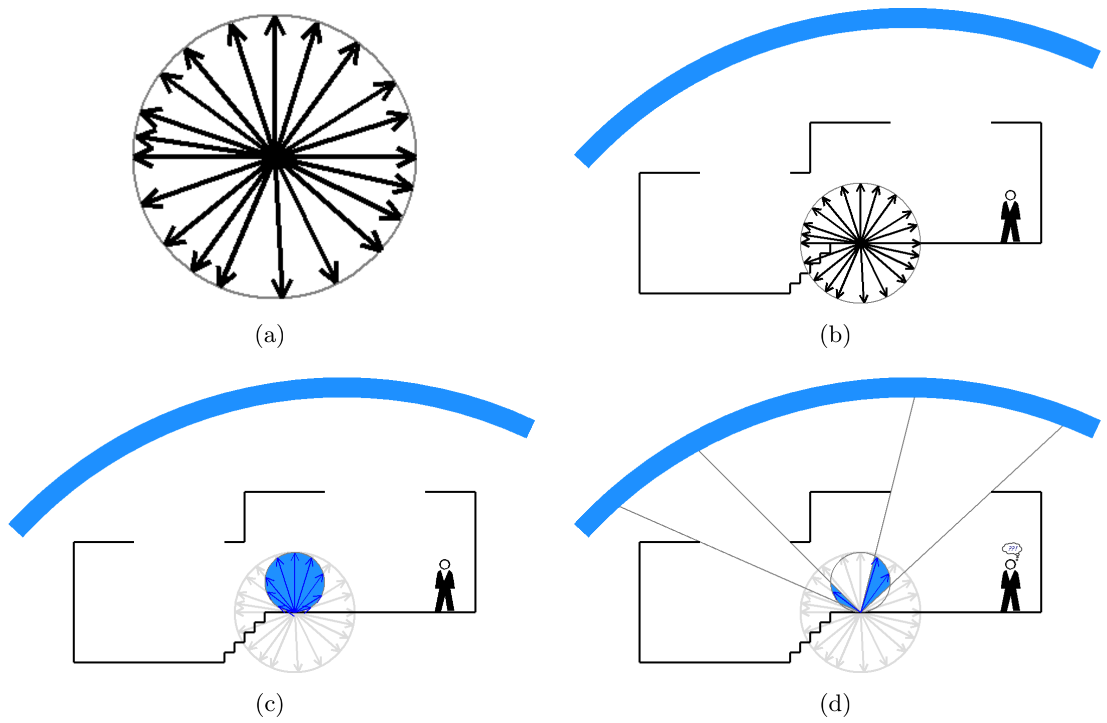


Figure 15: Computing a diffuse, self-shadowed transfer function.

5.2.2 Diffuse Transfer Functions with Bounce Lighting

Contrary to other researchers like [SKS02] or [Gre03], I have decided to examine the *shooting solution* to compute bounce transfer for the reasons that are detailed in section 5.3, where the analogy of SHL to light-maps and radiosity is discussed.

For any approach to bounce transfer, it is required to shoot or gather transfer functions to or from points in the scene from or to other points in the scene. One of the key ideas of this thesis has been to handle transfer functions analogously to the illumination-handling radiosity approaches as described in [CW93], or the specific approach described in section 3. Section 5.3 details that that is actually possible.

However, there is an important insight that helps immensely to actually achieve this goal: In theory, it is attempting to compute all samples for the transfer functions of all points of the scene as described in the previous section. The second step would then implement the bounce transfer part, working on transfer functions that are still represented as collections of samples and passed around as such in the scene. Only at the very end, after the bounce transfer step has converged to a solution, would this method compute the SH coefficients from the final collections of samples.

This approach suffers from two major problems: Combining (i.e. adding) transfer functions that are represented as sets of samples can be computationally both difficult and expensive.

- If each point in the scene got its set of directional vectors for spherical sampling created individually, these sets would in general be disjoint. Then, the only way to create a combined transfer function from two source functions would be to join their sets of samples. However, doing so has the tendency to grow the number of samples for the transfer function of a point in an exponential manner.

- In case the set of directional vectors has been chosen identically for each point (which is both a feasible and reasonable assumption), combining two sets of samples reduces to adding their values. That is, if the (ordered) set of samples of the transfer function for point P is $S_P = \{s_1, s_2, s_3, s_4, \ldots\}$, and the (equally ordered) set of samples of the transfer function for another point Q is $T_Q = \{t_1, t_2, t_3, t_4, \ldots\}$, and if directionalVector(s_i)=directionalVector(t_i) for all i, then the transfer functions of P and Q can be combined by copying their directional vectors and setting the appropriate sample values to $s_i + t_i$. While this is much better than the first case, this is still a forbiddingly expensive operation (often including many thousands samples) that is at the innermost loop of a radiosity-like algorithm.

Fortunately, there is an interesting observation that helps with both problems: Let $c_l^m(f)$ be the SH projection for the arguments l and m and for a given spherical function f as defined by equation 6. That is, analogous to equation 6, define $c_l^m(f) = \int\limits_S f(s) y_l^m(s)\ ds$. Then does the following property for $c_l^m(\ldots)$ and two spherical functions f and g hold:

$$c_l^m(f+g) = c_l^m(f) + c_l^m(g) \tag{11}$$

The proof is straightforward:

$$\begin{aligned} c_l^m(f+g) &\iff \int\limits_S (f+g)(s)\, y_l^m(s)\ ds \\ &\iff \int\limits_S (f(s)+g(s))\, y_l^m(s)\ ds \\ &\iff \int\limits_S f(s)\, y_l^m(s) + g(s)\, y_l^m(s)\ ds \\ &\iff \int\limits_S f(s)\, y_l^m(s)\ ds + \int\limits_S g(s)\, y_l^m(s)\ ds \\ &\iff c_l^m(f) + c_l^m(g) \end{aligned} \tag{12}$$

In other words, this means that is does not matter whether we add two spherical functions before *the projection or* after *the projection!* This in turn implies that we are not forced to actually work with the samples of the transfer functions during the bounce lighting phase, but we can rather resort to a much more natural method: The first step for bounce transfer is *exactly* what we did for computing the diffuse, self-shadowed transfer functions (section 5.2.1). This includes the projection of the samples into SH coefficients. Then we may add bounce transfer as a second, entirely optional step. This leads to a very modular algorithm design.

5.3 The Analogy to Light-Maps and Radiosity

Virtual worlds that are intended for interactive display are even today normally tessellated in a low-polygonal manner. This is especially true for *structural* elements, i.e. objects in the 3D world that are large with respect to the observer and light source attributes. Therefore, the problems associated with lighting where intensities are interpolated across polygon vertices (Gouraud shading) are still common. [FvDFH90] has an excellent description of Gouraud shading (pp. 736), as well as of the associated problems (pp. 739) like perspective distortion, orientation dependence, problems at shared vertices, and others.

Before we proceed though, let us briefly recap how *CaLight* works. CaLight is my progressive refinement radiosity implementation for computing high quality light-maps, presented in detail in section 3.3. Understanding the principles of CaLight will be a great help in making decisions about the implementation of SHL, discussed below.

- We start with the assumption that the entire world (scene) is uniformly covered with square patches.
- Each patch holds and maintains a “Total Energy” and “Unradiated Energy”. Total Energy is the total light energy that this patch reflects into the environment, that is received minus absorbed light energy (plus emissive energy if the surface is a light emitter). Unradiated Energy is the energy that has not yet been emitted (or “shot”) into the environment.
- Then the following steps are repeated continuously in a loop:

  1. Search the set of all patches for a patch A whose Unradiated Energy is reasonably high.
  2. Shoot As Unradiated Energy at all other patches in the environment, taking mutual visibility, spatial arrangement, distance etc. into account.
  3. Increase both the Total Energy and the Unradiated Energy of all hit patches by the appropriate amount.
  4. Reset the Unradiated Energy of A to 0.

  This is repeated until no patch with a noteworthy Unradiated Energy can be found in the first step of the loop.

Therefore, one of the early key decisions in my implementation had been to *not* implement SHL on per-vertex basis, but rather per-pixel on programmable graphics hardware. This in turn, with the above summary of CaLight in mind, suggests to treat SHL analogously to the light-maps that have been presented in section 3. Despite the fact that all literature and researchers known to me (especially [SKS02] and [Gre03]) refer to vertex-based SHL implementations, the following considerations led me to choose the per-pixel approach right from the start:

- Polygons can even today be very large, as they are used as world polygons for walls and floors. In order to avoid the same problems that ordinary lighting (Gouraud shading) has, implement SH lighting on the per-pixel level.
- If it works – good. If it turns out being too problematic (e.g. too slow, problems in FS profiles, ...), we can still increase SHL-map texel-size or revert to the vertex-shader-level.
- Have experience with traditional light-maps already, and "SHL-maps" at per-pixel level have been expected to behave similarly. (For example, we can already deal well with borders of polygons.)
- In current profiles, even in those for NV3X GPUs, textures cannot be accessed from within vertex shaders. However, we need such textures to store our SHL coefficients.
- Some matters in [Gre03] were unclear to me – a new thorough implementation gives room to clarify also these issues. The details are given in the discussion below.

Nonetheless, the SHL implementation provides convenient ways to fallback to per-vertex or even software lighting if required.

The next section discusses *CaSHL*, which is for SHL what CaLight is for light-maps. The section after that will then explain the rendering of the SHL-maps as generated by CaSHL, and the subsequent sections discuss various extensions to the standard SHL that have been explored in this thesis.

5.4 CaSHL: Computing the SHL-Maps

One of the key questions that were subject to investigation within this thesis was how SH lighting can be well computed and employed in practical contexts. As has already been indicated above, my first research suggested to store all SH data in "patches" that in turn are organized in "SHL-maps", the SHL equivalent to traditional "light-maps".

Further experiments with implementations for precomputing SH bounce transfer then quickly showed that such *bounce transfer can actually be computed in a way that is roughly analogous to the way bounce* light *(in the radiosity tool) works.* That in turn encouraged me in fact to lay out CaSHL analogously to CaLight, that is, to create the bounce transfer computations in a similar fashion as the radiosity bounce light computations.

Assuming that the reader is sufficiently familiar with shooting-based radiosity solutions in general, and with CaLight as presented in section 3.3 and summarized in the preceding section, I will now detail how CaSHL proceeds in order to precompute the SHL-maps.

5.4.1 Initialization

CaSHL initializes exactly like CaLight did: It extends the precomputed PVS information from the leaves of the binary space partitioning tree to individual surfaces. The details of this step are given in depth in section 3.3.1.

Once more, it is required that we can hold all patches in memory simultaneously, and therefore the memory for all coefficients of all patches is allocated next. (As discussed above, our patches do not now store light *energy* any more, but rather light *transfer functions*, which are represented as SH coefficients. This is the first difference from CaLight.) The transfer function at each patch is initially set to the “null” transfer function.

The initialization is completed by another step that is entirely identical to CaLight: The computation of geometric information about the patches’ spatial orientation and location is important for the postprocessing at the polygon borders later. This step also involves the introduction of the per-patch normal-vector as discussed in section 5.7.

5.4.2 Direct Transfer

The Direct Transfer step assigns initial transfer functions to each patch as described in section 5.2.1, and is crucial for the rest of the algorithm: Starting from each patch’s origin, the algorithm casts several thousand “shadow-feelers” into the environment in order to determine for each sampled direction if a sky surface or an obstacle was hit. This yields the initial transfer function (represented as a set of samples), which is then converted into SH coefficients representation and finally stored with its associated patch.

It is possible to exit CaSHL at this stage and save the transfer functions that have been computed as far as the final results. These transfer functions only account for direct, self-shadowed, but not interreflected diffuse light.

5.4.3 Bounce Transfer

As was the case with CaLight, the “bounce stage” is the heart of the program. While section “3 Diffuse Interreflected Transfer”, pp. 31 in [Gre03] describes the principles of this stage, at the latest now it becomes obvious that mine and [Gre03]’s approaches diverge: CaSHL employs an infinite loop that picks in each iteration the patch with the largest or a reasonably large un-shot transfer function, and shoots its transfer into the environment. The loop is exited when no patch with a sufficiently large transfer function can be found. How “large” a transfer function is is determined by computing the sum of the absolute values of its SH coefficients.

5.4.4 “Tone Mapping” Transfer Functions

Extending the analogy between CaLight and CaSHL led me to the conclusion that transfer functions undergo the same mathematical operations in CaSHL that radiosity quantities do in CaLight. That means that after the bounce transfer phase, the resulting transfer functions are in a high dynamic range. This in turn eventually yields the same problems for rendering transfer functions as for regular radiosity computations, namely the fact that the range of the transfer functions by far exceeds the dynamic range that computer monitors can reproduce.

Therefore, I applied the same tone reproduction operator by [LRP97] that I use for the radiosity quantities also to the transfer functions in CaSHL. In fact, doing so produced the expected results: While the contribution of the bounce lighting phase was practically unnoticeable without the tone mapping step, including it yielded observable interreflected lighting!

5.5 Rendering Lighting with SHL-Maps (Native Approach)

5.5.1 Inputs

Rendering SH lighting in the engine takes the precomputed SHL-maps as input. The second required input is the SH light source. It is possible to base the SH light source on realistic sky models as mentioned in [Gre03], and I added an appropriate implementation for the CIE Clear Sky model to the Ca3D-Engine. However, for testing and research purposes it is often better to employ purely artificial light sources, which are also easier to compute, especially for the dynamic case when the light source parameters change on each frame. All screenshots of SH lighting in this thesis were made with an artificial light source.

The light source must be given as a spherical function, represented by SH coefficients. These coefficients are normally obtained in the same way as for the patches' transfer functions in CaSHL: First the function is defined in the representation of a set of spherical samples which are then transformed into SH coefficients in a subsequent step.

5.5.2 Storing the SH Coefficients

As the rendering is intended to employ contemporary 3D accelerator hardware, we are required to encode the SH coefficients that are stored in the SHL-maps such that they fit into regular texture-maps. In order to achieve this, consider a single instance: One SHL-map stores n^2 SH coefficients (for n SH bands) per element. Therefore, I put the first four coefficients of the SHL-map into the RGBA elements of a texture-map of the same dimensions. The second quadruple of coefficients goes into the RGBA elements of another texture-map, the third quadruple into the RGBA elements of a third texture-map, and so on. Thus, a total of $n^2/4$ texture-maps for an n band SHL-map are required.

As the components of each RGBA texture-map element are normally limited to represent numbers from 0 to 1 in steps of 1/255, range compression of the SH coefficient values is required for the storage. Figures 16 and 17 show scenes where the first texture-maps of the SHL-maps have been rendered directly, showing the first three color-encoded SH coefficients as RGB triples.

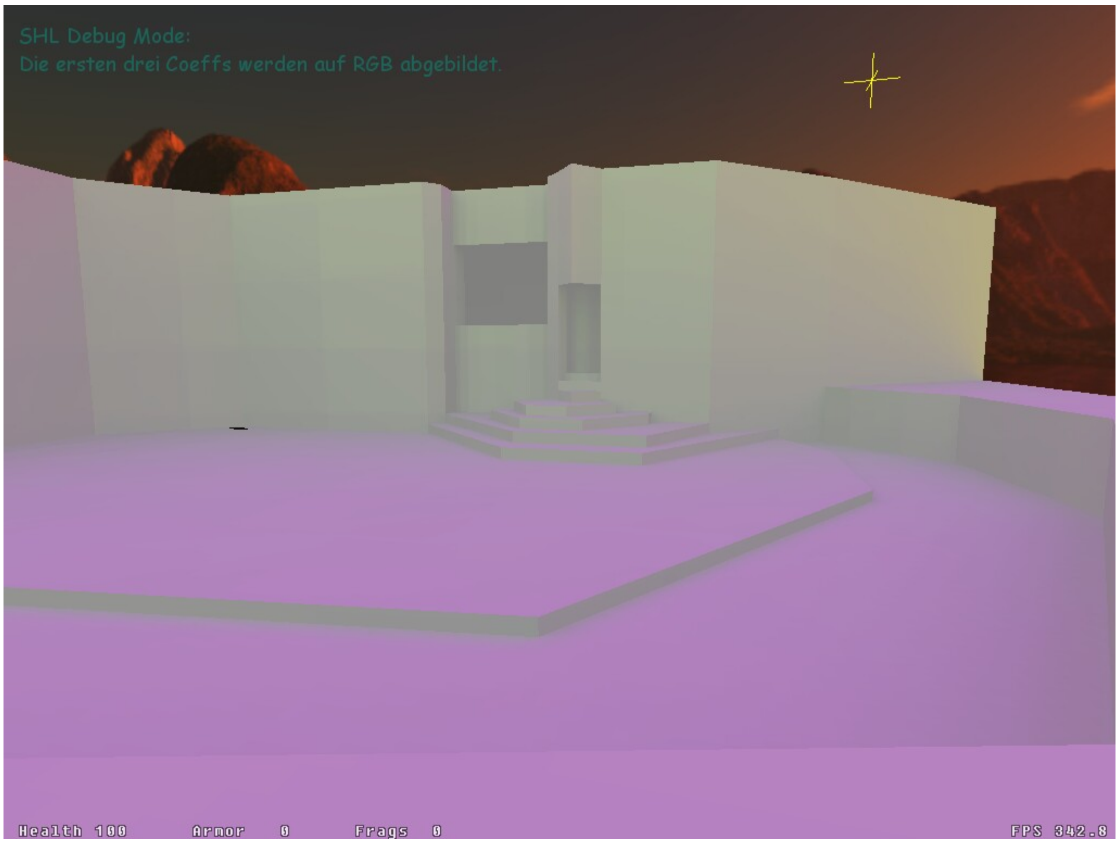
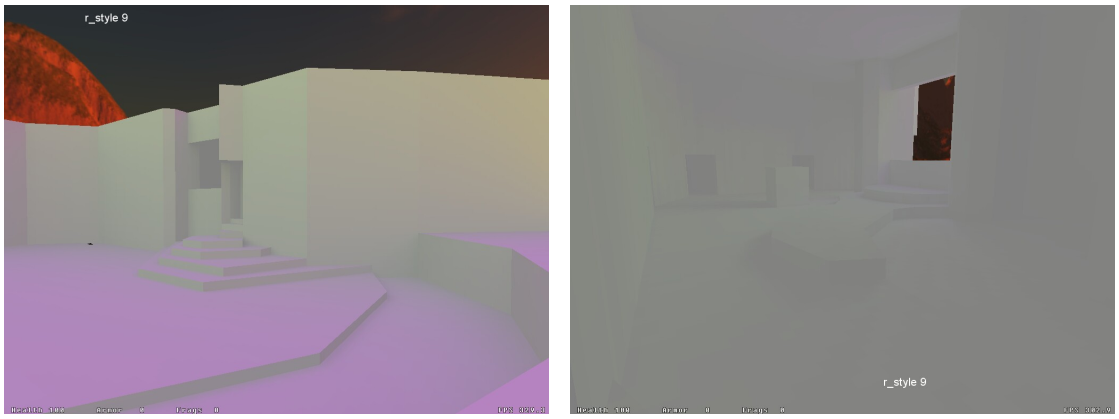


Figure 16: The first three SH coefficients color-encoded into the RGB channels.



(a)

(b)

Figure 17: Two more scenes that directly show the color-encoded SH coefficients.

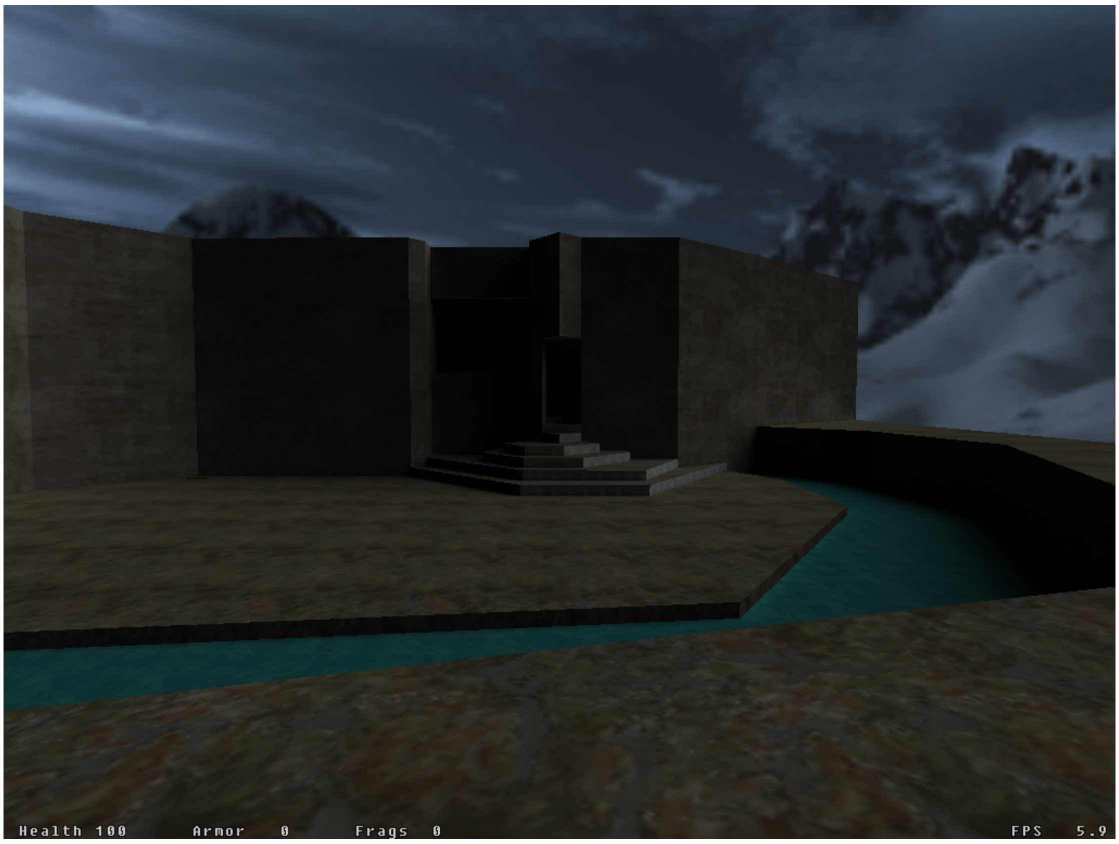


Figure 18: A first result of my SHL rendering implementation.

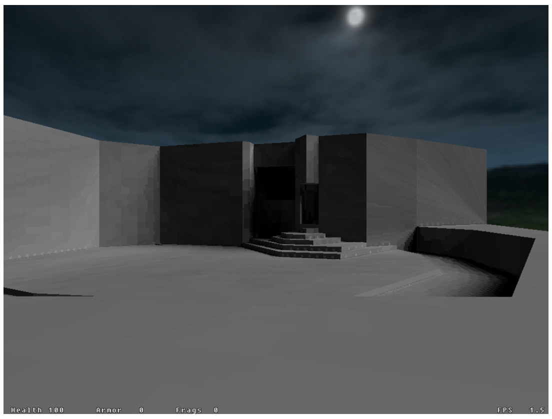
5.5.3 Results

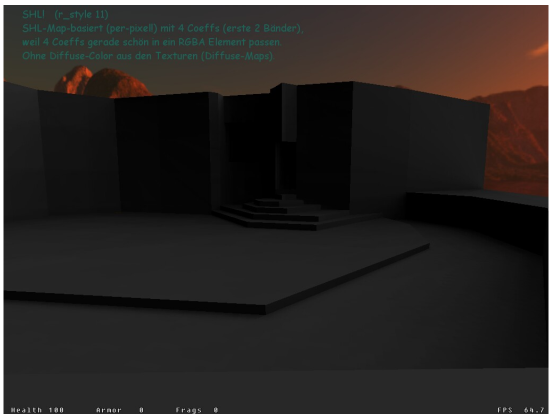
Given both the light source and the per-patch transfer functions, the latter stored in texture-maps as explained above, a per-pixel program that runs directly on the GPU of modern 3D accelerator hardware is at the core of SHL rendering. As OpenGL 2.0 has not yet been available at the time of the implementation, I have chosen to employ NVidias Cg using profiles for NV3X GPUs for the per-pixel programs.

Figures 18 and 19 present example rendering results that were obtained with this method. Implementation details are given in section 6.4.1. (Figures 19(e) and 19(f) were actually obtained by the means described in the next section, but they show visually identical results.)

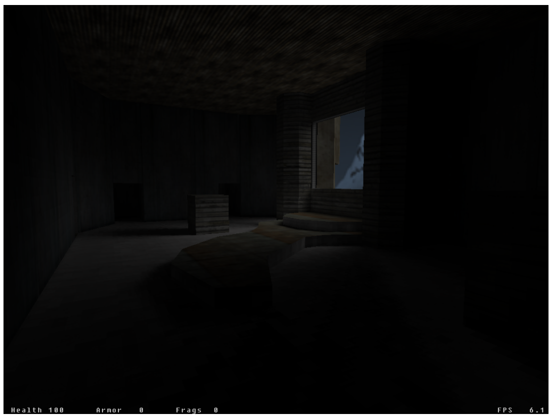
5.6 Compressing SHL Data

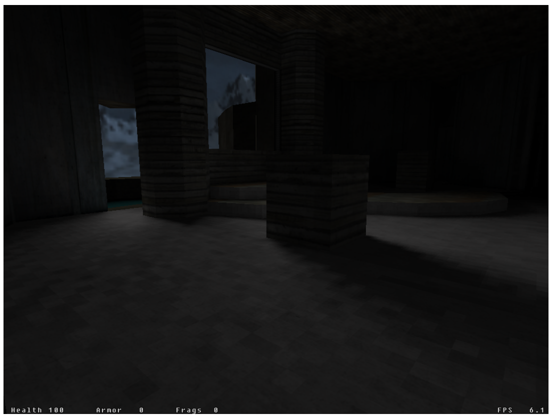
After the first practical tests with SH lighting where the coefficients are all stored in SHL-maps rather than only per-vertex, it turned out that the storage requirements for SHL-maps easily lead to very large world files on disk, and – more importantly – easily

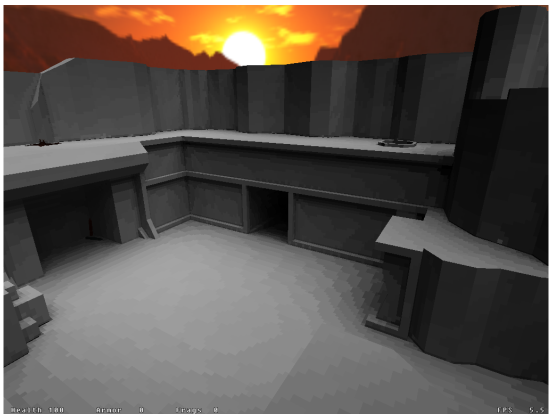

(a)


SHL!   (r_style 11)
SHL-Map-basiert (per-pixel!) mit 4 Coeffs (erste 2 Bänder),
weil 4 Coeffs gerade schön in ein RGBA Element passen.
Ohne Diffuse-Color aus den Texturen (Diffuse-Maps).

(b)


(c)


(d)


(e)

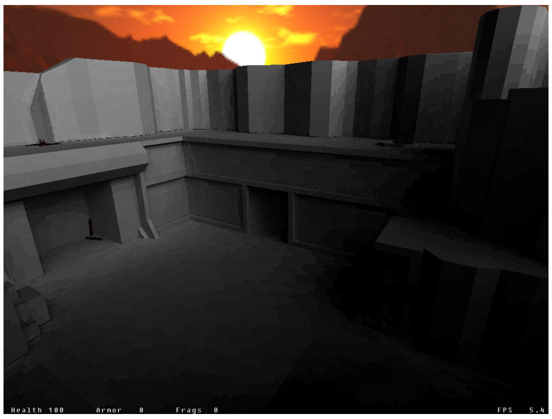

(f)

Figure 19: Additional results of my first SHL implementation. The celestial light source (sun, moon) is animated.

exceed the limits of available video memory when they are uploaded to the video board for hardware-accelerated processing.

More precisely, a world that has a total of S sample points (that is, SHL-map elements) at n SH bands, requires the storage of n^2S SH coefficients. With n being typically at least 4, S in other order of several hundred thousands or millions (say 500000 for a small world), and the coefficients being stored as 4-byte floats on disk, this yields $4^2 \cdot 500000 \cdot 4\frac{\text{bytes}}{\text{coeff.}} = 32 \cdot 10^6$ bytes or roughly 30 MB just for storing the SH coefficients. Before uploading this data to the video card, I range compress each coefficient into a single byte, which yields about 7.5 MB video memory consumption. While this example was for a pretty small world, average worlds normally have roughly ten times as many sample points, yielding 75 MB of required video memory. Increasing n from 4 to 6 increases video memory consumption from 75 MB further to about 169 MB, and increasing n to 8 yields 300 MB of required video memory. This in turn is unfeasible for, or at least inconvenient with, today's consumer video boards.

Therefore, one of the key goals of this thesis became the compression of SHL coefficients. As suggested by Jan Kautz, a practical and straightforward method to achieve this is *vector quantization*, which in our context works in the following steps:

1. Start with computing all SHL coefficients in CaSHL as before, including the bounce-lighting and postprocessing steps.

2. For easier reference, consider the n^2 SH coefficients at each SHL-map element (sample point) as *vectors* for the rest of this section.

3. Instead of writing the (vector) results immediately to disk now, do rather determine a set of *representative vectors* from the set of all vectors (which is of size S).

4. Assign each of the S original vectors another vector from the set of representative vectors that is closest to that vector. (Pick the closest representative for each original vector.)

5. For each original vector, just save the index to its closest representative (rather than the vector itself).

6. Save the representatives.

As the number of representative vectors is normally only several thousand (and in practise even limited to 2^{16} such that indices into the array can be stored in 16 bits), this approach yields a significant compression ratio. Both the compression ratio as well as the impact of this method on image quality are discussed below.

The above outlined method contains two interesting sub-problems, namely how the set of representatives is best determined, and how decompression is best handled rendering time.

5.6.1 Computing the Set of Representatives

Given a set O of original vectors, we wish to compute a set R of “good” representative vectors.

This problem is known as vector quantization, and in similar form known from the problem of reducing a 24 BPP color image to an 8 BPP palette-indexed image. For image color reduction, [Pip98] describes a simple octree based algorithm. [Dek94] describes a neural-net based algorithm that yields very good results.

However, in CaSHL I implemented the *Iterated Closest Points* method in order to determine the representative vectors. This method is simple, yields good results, and has reasonable space and time requirements. The algorithm works as follows:

1. Initialize R by picking $|R|$ arbitrary vectors from O. That is, let R be a subset of O.

2. For each vector in O, compute the distances to all vectors in R, and store the one that it is closest to (i.e., save the index into R). The result is for each vector in O an index into R.

3. From that information, compute the reverse assignment: For each representative in R, build the list of indices of original vectors that refer to this representative as being the closest. The result is a cluster (list of indices into O) of the closest original vectors.

4. Delete R and compute its elements completely anew from the center (average) of this element’s cluster vectors.

5. Go to step 2.

While this algorithm is simple in principle, interesting questions arose in the implementation with respect to the number of iterations and the termination of the algorithm. My initial approach to break out of the above mentioned loop was to assume that the algorithm strictly converges. Thus, I assumed that in step 2, once a solution was reached, no vector in O gets another index into R assigned than in the previous iteration. Unfortunately, this assumption was wrong – not only can it take several hundred iterations until termination is detected (i.e. step 2 yields the same index list as in the previous iteration), even worse is that the method can cycle! That is, with this criterion for termination and in the presence of cycling, the algorithm is in an infinite loop.

The second approach therefore was to determine the largest distance of the minimum distances after step 2. If this distance is smaller than the value computed in the previous iteration, the algorithm made progress. Otherwise, the new assumption is that no further progress is possible and that an optimal solution has been found. In order to account for rounding errors or other ill-cased conditions however, I allow for up to three iterations to fail in a row before the loop is finally left. That is, the algorithm terminates only if after three subsequent iterations no maximum distance is found that is smaller than the currently best (smallest) value.

In my first practical tests with a small world, the second, revised termination method reduced the number of iterations until a solution was detected from 150 to 6 in one case, and similar orders of magnitude in other test cases.

Finally, I tried to shortcut the algorithm by skipping some of the original vectors from O in step 2 in early iterations, in order to make the loop execute faster. Even though it was cleverly implemented, this trick did not yield much temporal gain, and even decreased the quality of the computed result (the largest distance measured after step 2, taken as an error measure, grew with this shortcut).

5.6.2 Storing the Indices and Representatives

After the set of representative vectors has been computed, our SHL-maps do not store any longer the n^2 SH coefficients per element directly, but rather just a 16 bit index into the representatives. As the indices must later be accessed from within a fragment program that executes on the graphics processor, they must be encoded in regular texture-maps. I therefore decided to put the lower 8 bits into the red texture channel, and the upper 8 bytes into the green texture channel.

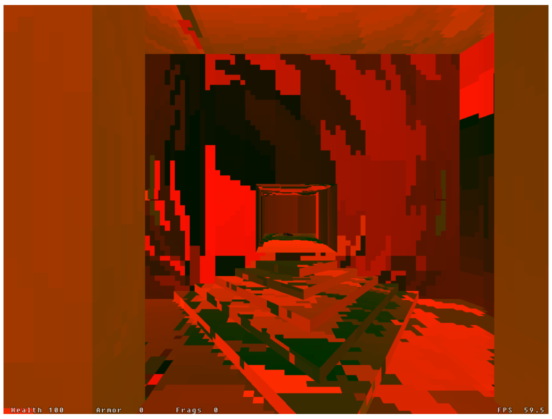
Rendering these color-encoded index-maps directly is not only very useful for debugging, but also provides interesting insights into how the above discussed algorithm works, and indicates the correctness of the implementation. Figure 20 shows images that have been taken in the index-map debug rendering mode of the Ca3D-Engine. The shown world is *DmBase* at 8 bands and 16384 representative vectors.

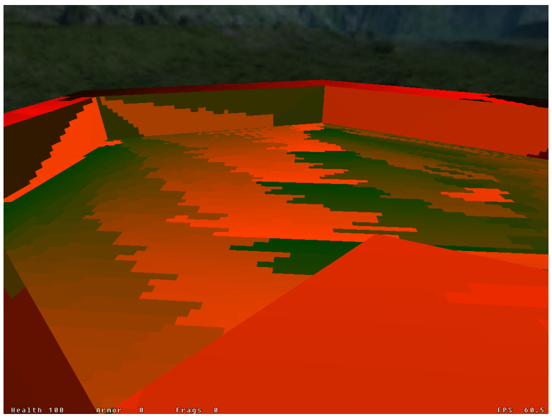
The representative vectors themselves, each consisting of n^2 SH coefficients, must also be encoded in regular texture-maps, such that the GPU fragment program can access them in a second step. The lookup of the representative vectors depends on the preceding lookup into the bare index-textures (shown in figures 20 and 22) that provide the index/location into the representative texture-map for reading the SH coefficients.

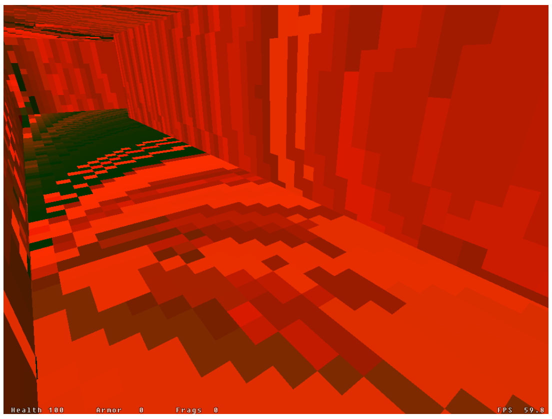
For my original, uncompressed implementation of SH lighting, I decided to store the n^2 SH coefficients per SHL-map element by "layering" multiple texture-maps: The first four coefficients were range-compressed into the RGBA-quadruple of the appropriate element (pixel) of the first texture, the fifth to eighth coefficients were stored in the RGBA-quadruple of the same pixel of the second texture, and so on. This approach thus required $\lceil n^2/4 \rceil$ "layered" texture-maps that together combine to one n-band SHL-map. (And one world in turn often requires several dozen SHL-maps for full coverage.)

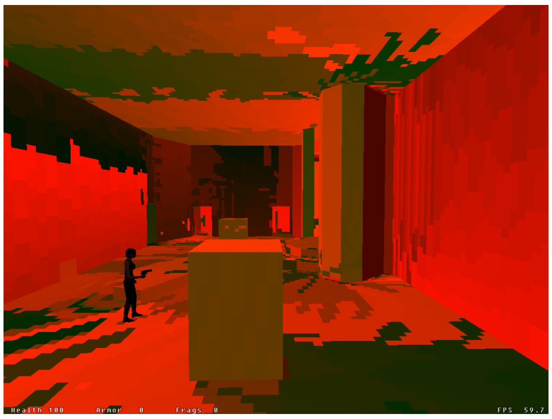
When it comes to storing the representatives for the compressed-SHL implementation, it initially seemed promising to organize the representatives in a simple tabular layout, where each row contains exactly one representative and there are as many columns as there are SH coefficients (per representative). This avoids the requirement to store arbitrarily many coefficients in multiple layered textures, but creates texture-maps whose height matches the number of representatives (up to $2^{16} = 65536$) and whose width for n bands is $\lceil n^2/4 \rceil$ pixels. In practise, the texture-map width must even be increased to the next integral power of two.

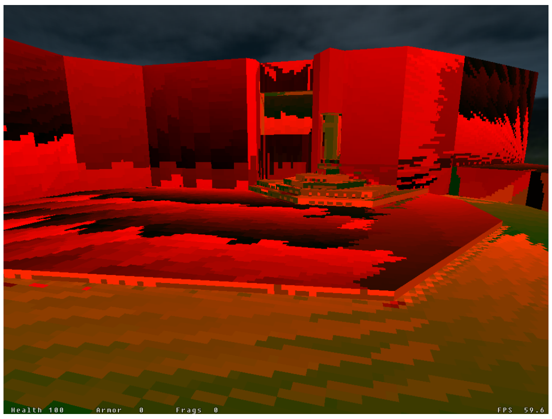
However, the splitting of the index numbers into the low and high byte requires recomposing the original 16-bit number before the tabular texture-map can be accessed.


(a)


(b)


(c)


(d)


(e)

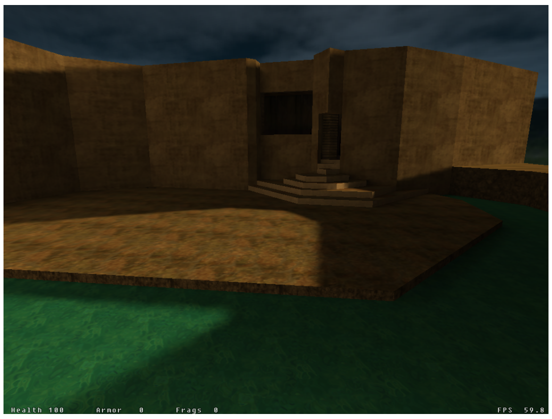

(f)

Figure 20: Index-maps that were rendered directly into the color-buffer for debugging purposes. All images were taken in *DmBase* at 8 bands and 16384 representative vectors.

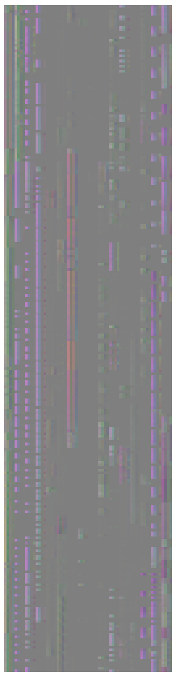


Figure 21: An example for a color-encoded representative-map. The texture is 64 pixels wide and 256 pixels high. It stores 4096 representatives in 16 columns of 4 pixels width each. Thus each representative has 16 coefficients or 4 SH-bands.

Given today's precision and width of GPU registers, and the fact that integer computations are simulated by floating-point operations, this computation is likely to be subject to rounding errors. Rounding errors however are, for our purposes, *fatal, as neighbouring representatives in the lookup-table are generally incoherent!* That is, accessing the $(i+1)$-th or $(i-1)$-th representative rather than the i-th yields not just mildly inaccurate results, but rather something that is *entirely wrong.*

This problem can be avoided by storing the representatives in a 2-dimensional layout in order to account for the low and high byte of the index number: The table of representatives is cut every 256 entries. This is the range that the low byte can address. The sub-tables of 256 entries each are then stored in columns next to each other, yielding $\lceil |R|/256 \rceil$ columns ($|R|$ being the number of representatives), where each column in turn consists of $\lceil n^2/4 \rceil$ pixels as in the original table. As this avoids additional computations, there is no room for rounding errors to occur. Figure 21 shows an example for such a "raw" representative-map into which the SH coefficients have been color-encoded by the same range-compression method as with the uncompressed implementation.

Finally, the remarks in this section also imply that both index-maps as well as representative-maps *must always by accessed* by "nearest" (`GL_NEAREST`) filtering. Any other regular filtering method mixes up the index numbers and/or the (incoherently stored) representatives, such that any of these destroys the function of the algorithms.

Scene #	FPS w/o compression	FPS with compression
1	6.7	6.5
2	6.7	6.6
3	6.7	6.4
4	6.6	6.5
5	6.6	6.5

Table 3: A FPS comparison of native vs. compressed SH lighting.

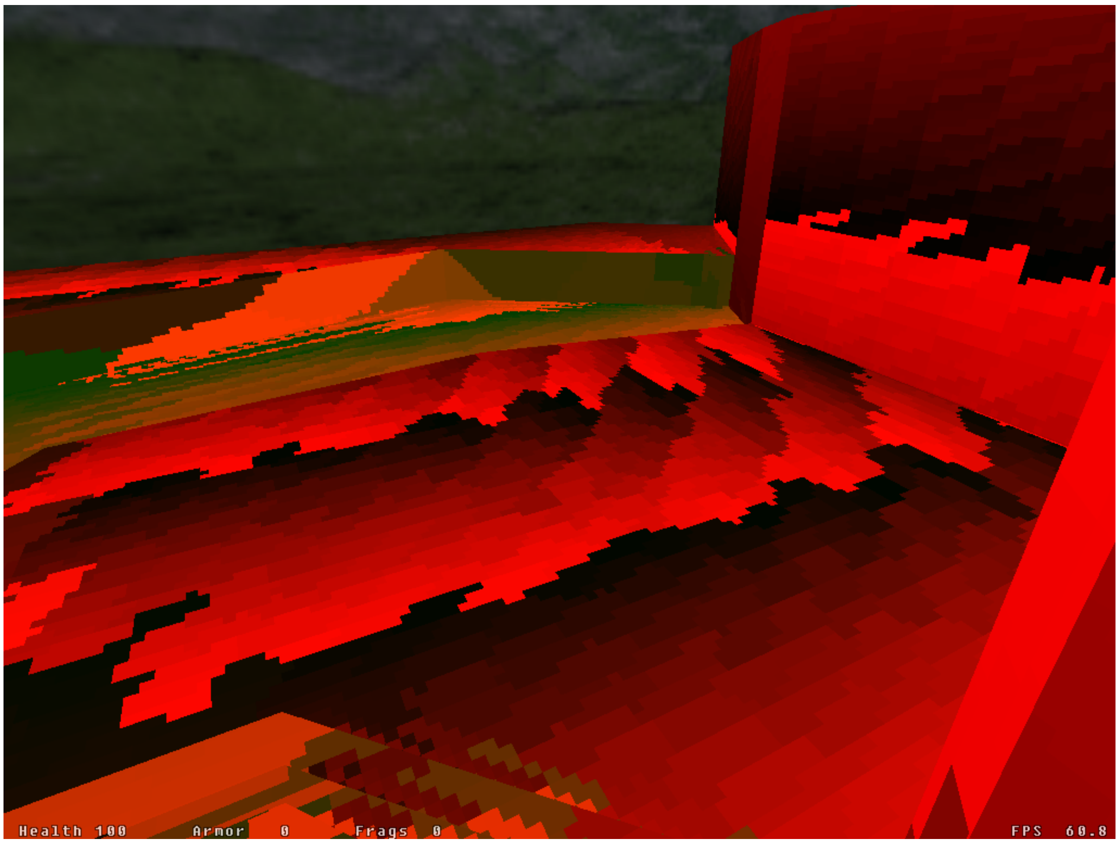
5.6.3 Results

Figures 22 and 23 show exemplary results of rendering with the SH coefficients compression technique detailed above. The images have intentionally not been modulated with the regular diffuse textures of the polygons, in order to clearly demonstrate the visual effects of the (de-)compression.

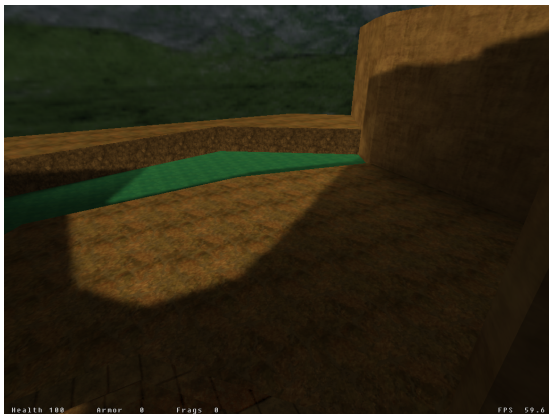
The most noticeable difference between the native (non-compressed) and compressed SHL rendering is the fact that the images where compression was employed look "patchy" or "blotchy". This is an immediate consequence from grouping similar SH vectors and assigning such groups just one representative vector. Several approaches can be taken to deal with this appearance:

- Ignore it. Once we combine the bare SHL lighting results with their diffuse texture, as is the normal case in practical applications, the blotchy patterns become less perceptible, at least partially. Figures 23(b) and 23(d) demonstrate this.
- In the next section, we present a method to combine SHL and normal-mapping. It turned out that as a side-effect, the technique examined there breaks up the blotchy patterns.
- An explicit way of reducing the pattern artifacts is to filter the SH lighting. This approach is examined in section 5.8.

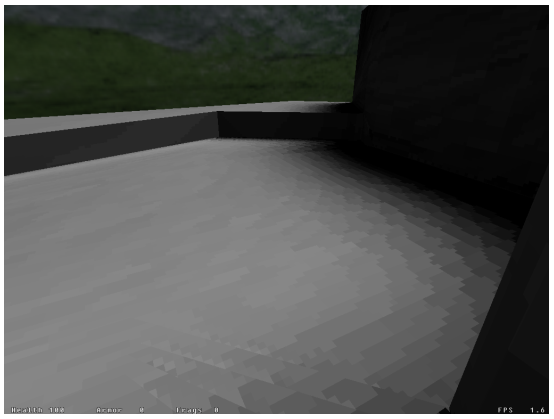
Due to the additional texture lookup for the dependent SH coefficient reads, and the relatively expensive "navigation" in the map of representative vectors, both of which are detailed in the implementation section 6.4.2, the frame-rate that the decompression method can produce is expected to be generally lower than that of the native method without compression. I've run a small test series in order to verify this assumption: A world was chosen and preprocessed both with and without our SH compression method. I then picked a set of "representative" test scenes, and measured the FPS for both cases. As table 3 indicates, the results are surprising: Employing SH compression only has a marginal penalty on rendering FPS in our implementation. However, it is probably too early to draw early conclusions from this test, as the test series was too limited to have any significance, other bottlenecks in the rendering pipeline have not been considered, and factors like differences in GPU memory consumption, texture object switching costs



(a)

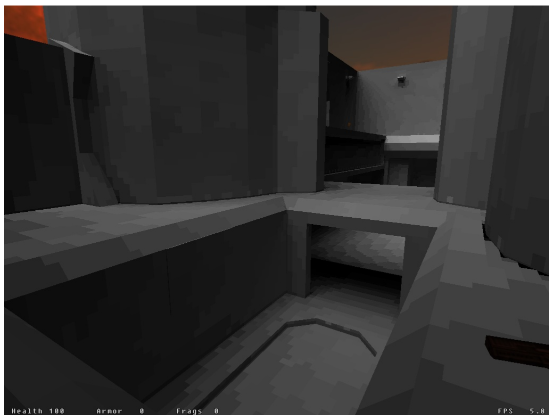


(b)

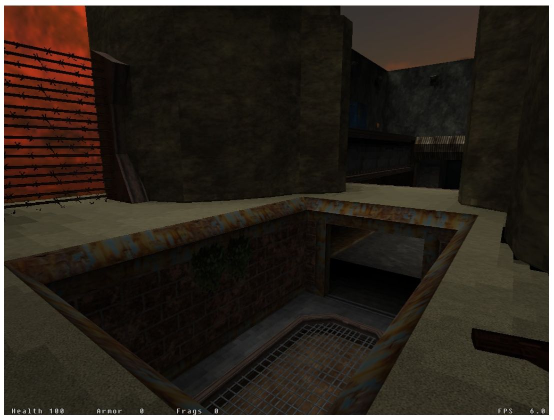


(c)

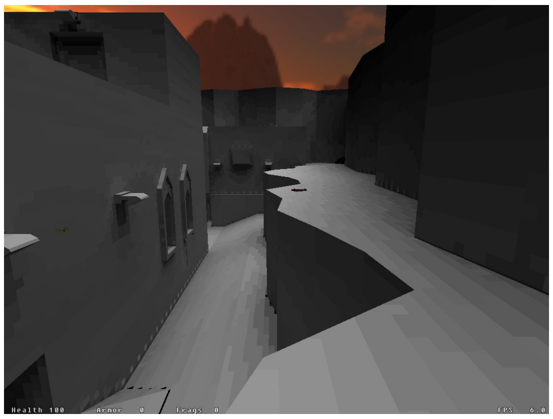
Figure 22: (a) shows another screenshot of the index-maps. Figures (b) and (c) show the same scene for reference, once with regular light-map lighting only, and once with bare but dynamic SH lighting, based on the indices shown in (a).



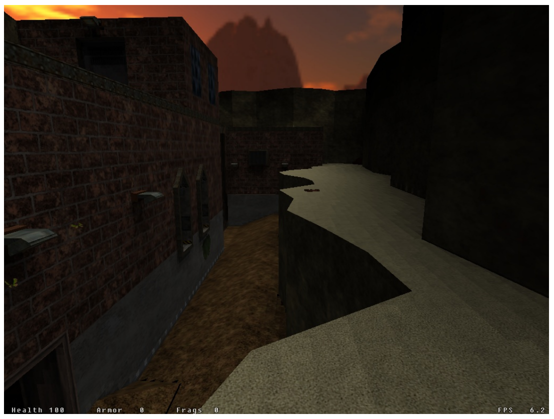
(a)



(b)



(c)



(d)

Figure 23: Rendering results with compressed SHL data. The images on the left show bare lighting results, the right images are combined with the diffuse textures.

etc. have not been taken into account. A more detailed analysis of the performance of compressed SHL is subject to future investigation.

The final important aspect of the compressed data approach is the compression ratio, that is the savings with respect to file size. Table 4 shows the file sizes of two world files, for each of which several variants of SHL data have been computed. The first line of each world (those with 0 SH bands) indicates the file size when no SHL information at all is stored in the respective file. The respective subsequent lines (those with 4 SH bands, but 0 representatives) indicate SHL data with 4 bands that is stored in uncompressed (native) form (zero representatives means no compression). The other lines state file sizes for 4 and 8 SH bands at varying numbers of representatives. It is clearly visible that the file size savings are enormous: For DmBase, the ratio of the pure SH data (that is, with the base file size of 1093 kB subtracted) between the uncompressed version

World File Name	**File Size (in kB)**	**# SH bands**	**Compression**
DmBase	1093	0	n/a (no SHL data contained)
DmBase	20305	4	0 Reps. (no compression)
DmBase	2717	4	16384 Reps.
DmBase	1949	4	4096 Reps.
DmBase	5789	8	16384 Reps.
ReNoElixir	7792	0	n/a (no SHL data contained)
ReNoElixir	ca. 74000	4	0 Reps. (no compression)
ReNoElixir	10985	4	32768 Reps.

Table 4: World file sizes at different compression settings.

and the version with the 4 SH bands and 16384 representatives is 11.83, and the same computation for the two ReNoElixir worlds with 4 SH bands each yields a size-saving factor of 20.74 (!).

It turned out that another property of compressed SH rendering is that, surprisingly, the implementation can be written in a technically more graceful way than can the implementation for native rendering. The details are provided in section 6.4.2.

These facts, combined with the beneficial implementation, clearly suggest favoring the presented compression of SH coefficients over the straightforward, native, non-compressed approach, as it is clearly the best solution to SHL rendering that is presented within the scope of this thesis.

5.7 Combining Normal-Mapping and SHL

Another subject for investigation within the scope of this thesis was the question whether and how Spherical Harmonic Lighting may be combined with Bump-Mapping (i.e. Normal-Maps).

5.7.1 Algorithmic Considerations

The key for combining SHL with Bump-Mapping is as follows: During the preprocessing steps in CaSHL, the sub-term

$$\max(\vec{N} \odot \vec{s}, 0) \tag{13}$$

accounts for the spatial orientation of the surface at the sample point with respect to the potential incoming light from the surrounding (hemi-)sphere. This does apply even to shadowed and interreflected diffuse transfer. Combining this with bump-mapping perturbs $\vec{N}$ as defined by the appropriate (tangent space) normal-map for the surface.

My initial idea was to actually factor the normal vectors from the normal-maps into the overall SHL equation analogously to their contribution in hardware-accelerated lighting as detailed in section 4.5: Initially *omit* the sub-term 13 from CaSHL (and thus make it *not* enter the SH coefficients generated by CaSHL), but rather account for it at the very end, during rendering, when $\vec{N}$ is known on a per-pixel basis. Unfortunately, it becomes clear very quickly that this approach is not practically feasible. Deferring the sub-term 13 to the end of the computational pipeline not only requires rearrangements of all involved computational steps that by doing so become far too expensive to be computed at rendering time, it also counteracts the bounce lighting phase of CaSHL, making interreflected transfer impossible to precompute.

Therefore, the second approach was to perturb $\vec{N}$ according to the normal-map right from the start in CaSHL. That is, instead of obtaining $\vec{N}$ from the unique normal vector that is stored with the surface (or the cross-surface interpolated normal vector if smooth-groups are used), CaSHL loads the appropriate normal-map for the surface, manually samples the normal-map at the appropriate sample point, and rotates the obtained normal vector from tangent- into object-space (or rather world-space in this case), once more taking any smooth-groups into account. The normal vector that is obtained is finally used as $\vec{N}$ for all further computations.

The consequences of this approach are very straightforward:

- As the normal vectors from the normal-maps are directly factored into the SH coefficients that CaSHL creates, all changes are local to CaSHL, and no other software component that participates in the entire chain requires any changes. In particular, the SHL renderer implementation does not require any code modification in order to handle the situation, no matter whether normal-maps are taken into account or not. Once more, this is because all relevant information is inherently stored in the SH coefficients that are created by CaSHL.

- Besides the above mentioned substitution of the per-surface normal vector with the normal-vectors from the normal-maps for $\vec{N}$, CaSHL requires only a few subtle changes in its direct- and bounce- lighting code:

  One change is related to *self-shadowing*, a well-known problem from Phong lighting (see e.g. [FK03], page 230): We now deal not only with a single normal vector, but rather with two. One is the “old” per-surface normal vector that we used earlier, the other is the newly introduced $\vec{N}$ that we obtained from the normal-maps. One way to think about these two normals is that the old normal is a large-scale approximation of the surface orientation, and the new $\vec{N}$ is a small-scale approximation of the surface orientation. Therefore, we do not only have to test if $\vec{N}$ faces away from incoming light transfer (as is achieved with the $\max(\ldots, 0)$ in term 13, but the same test also has to be done for the old, “large-scale” per-surface normal vector.

  Another change arises from an optimization in the core `RadiateTransfer()` function of CaSHL: In this function, we optionally radiate or “shoot” the not-yet-radiated transfer from several neighbouring patches of a common surface all at once, as if it were a single big patch instead of several smaller patches. While without normal-maps, all those patches used to share a common normal vector (as they are all from the same surface and thus all shared the single per-surface normal vector), with normal-maps they now all have an individual, unique normal vector. This in turn requires us to redetermine the normal-vector of the big accumulative patch by properly computing the average normal vector for the contributing original patches.

- The most uncomfortable consequence of this approach inherently results from the way SHL-maps and normal-maps are combined: As you can see in the screenshots throughout this thesis, the elements of both light-maps and SHL-maps are relatively big tiles that cover the world surfaces. This is because each tile represents a "patch". As explained earlier, patches are used in both CaLight and CaSHL for the radiosity-based light and transfer computations. As such, each patch is unique. Even though several patches of several surfaces are stored in common light- or SHL-maps, the limited storage for such maps plus the inherent properties of the radiosity shooting solution enforces a relatively big size of the patches, and thus the resulting tiles that cover all world surfaces.

  The SHL patches' size however is in high contrast to the size of the elements of the normal-maps. The normal-map elements are neither unique to each world surface (and thus a set of relatively few and small normal-maps can be repeated over and over), and their size doesn't affect the space and time requirements of the bounce transfer stage in CaSHL either.

  Consequently, a single SHL patch normally covers several hundred or even thousands of normal-vectors of a normal-map.[4] Therefore, the above briefly mentioned sampling of the normal-map in order to obtain a new $\vec{N}$ for each SHL patch is in fact much more difficult.

  One possible solution to this problem is to actually rasterize the normal-map along the shape of the currently considered SHL patch, and then to compute the average of all normal-maps within the rasterized result. This however must normally be done manually (although an implementation that employs dedicated graphics hardware is conceivable), and thus is truly cumbersome. I.e., the problems of tiling, wrapping, clamping, aliasing etc. all have to be taken into account, just as with a full graphics renderer.

  Therefore, I implemented a simpler method that just takes a fixed number of random samples of the normal-map within the "superimposed" rectangle of the SHL patch, and averages them. This yields a much simpler implementation that accounts for all the subtleties of the full rasterization method, whereby the obtained results are estimated to be of almost equal or just marginally worse quality than those that the rasterization method would provide.

The last mentioned property suggests that, depending on the actual normal-map, the average of several thousand normal-map vectors tends towards the (0 0 1) default vector that yields the same result as if no normal-mapping was used. That in turn suggests

[4]In the comments to the source code of my implementation (see `CaSHL/Init2.cpp`), I show that in the default case at least 6200 (!) normal-map elements are covered by a single SHL element.

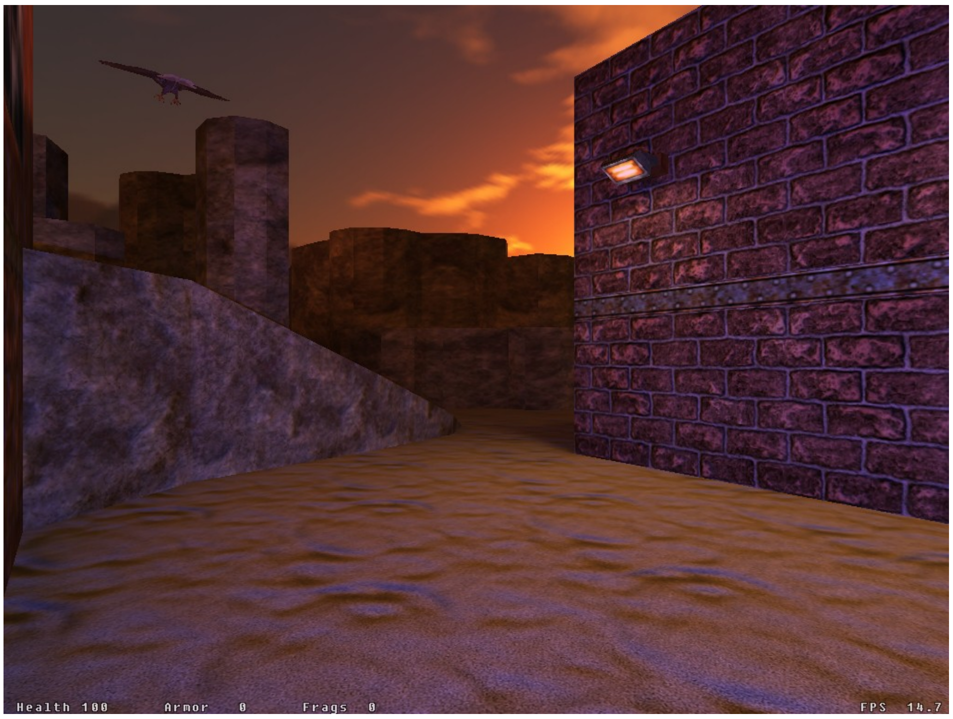


Figure 24(a): A reference shot of the first scene.

omitting accounting for normal-maps right from the start, but the results that I obtained were very interesting nonetheless.

5.7.2 Results

This section presents my results of combining SHL with Normal-Mapping in several series of commented screenshots. Each series shows several screenshots of the same scene, where the effect with and without normal-maps combined with SHL as described in the preceding section is presented.

The screenshots that show the SHL results have intentionally not been modulated with the appropriate diffuse-maps, and are therefore grayscale rather than colored. This pronounces the pure SHL effect, and is thus much better suited for comparing the results before and after the introduction of normal-mapping.

First Series

Figure 24(a) shows the common scene of the first series of screenshots. Only Phong-style lighting has been used to create this screenshot, Spherical Harmonic Lighting has intentionally *not* been employed. This image is for giving an idea how the scene looks when color-textured, and it also shows some of the features of the normal-maps. E.g., the small sand dunes are a result of the normal-maps. Also the brick wall on the right

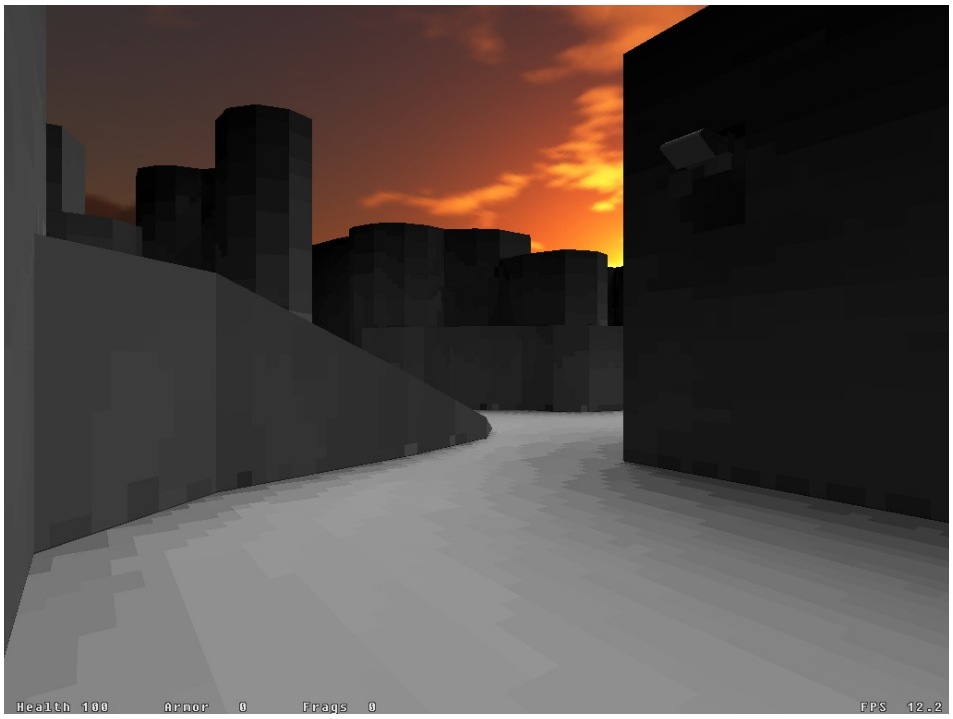


Figure 24(b): SHL-maps only (no normal-mapping and no color modulation).

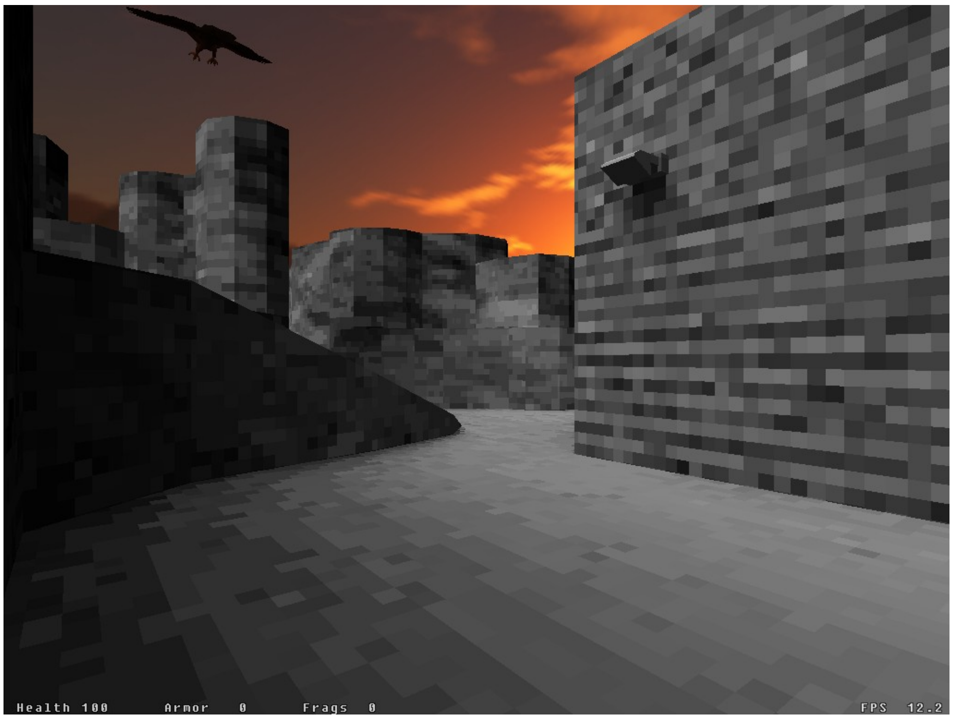


Figure 24(c): SH lighting with normal-maps taken into account.

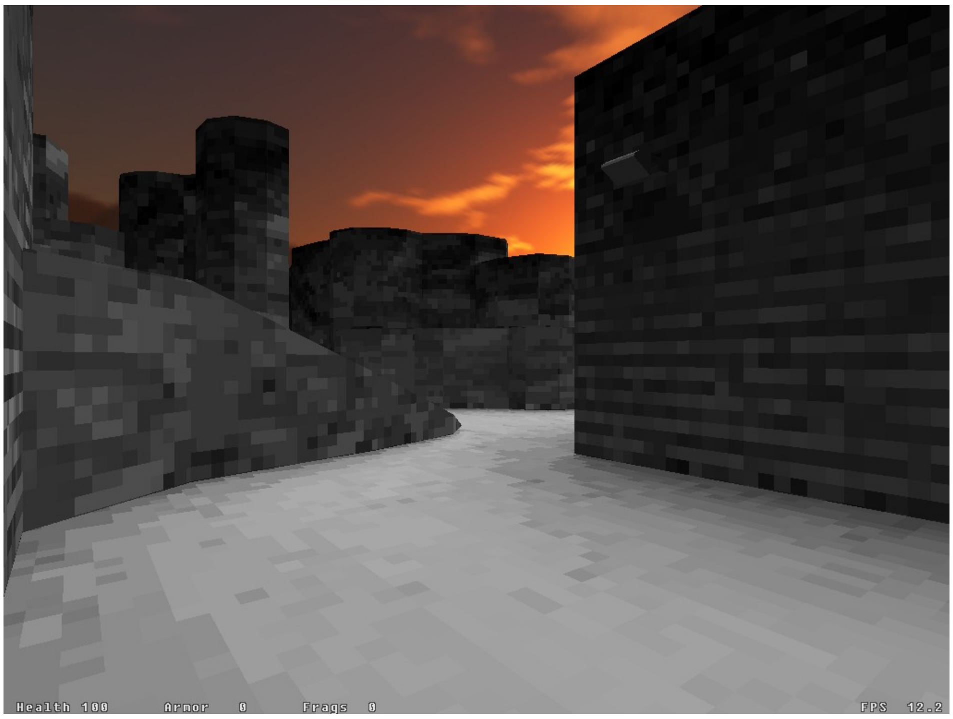


Figure 24(d): SH lighting with normal-maps taken into account.

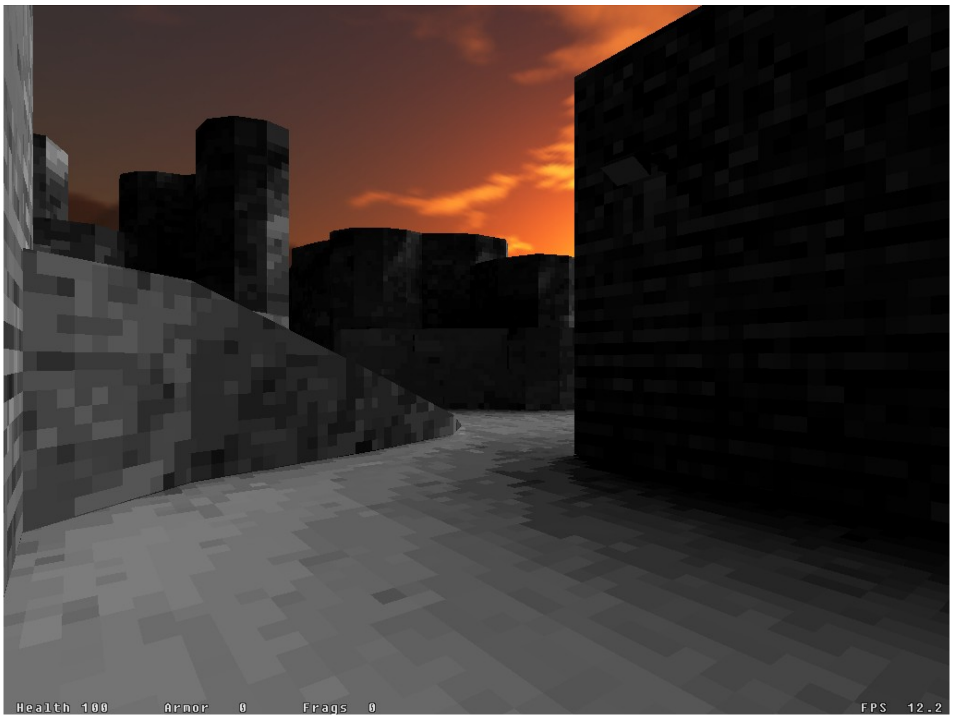


Figure 24(e): SH lighting with normal-maps taken into account.

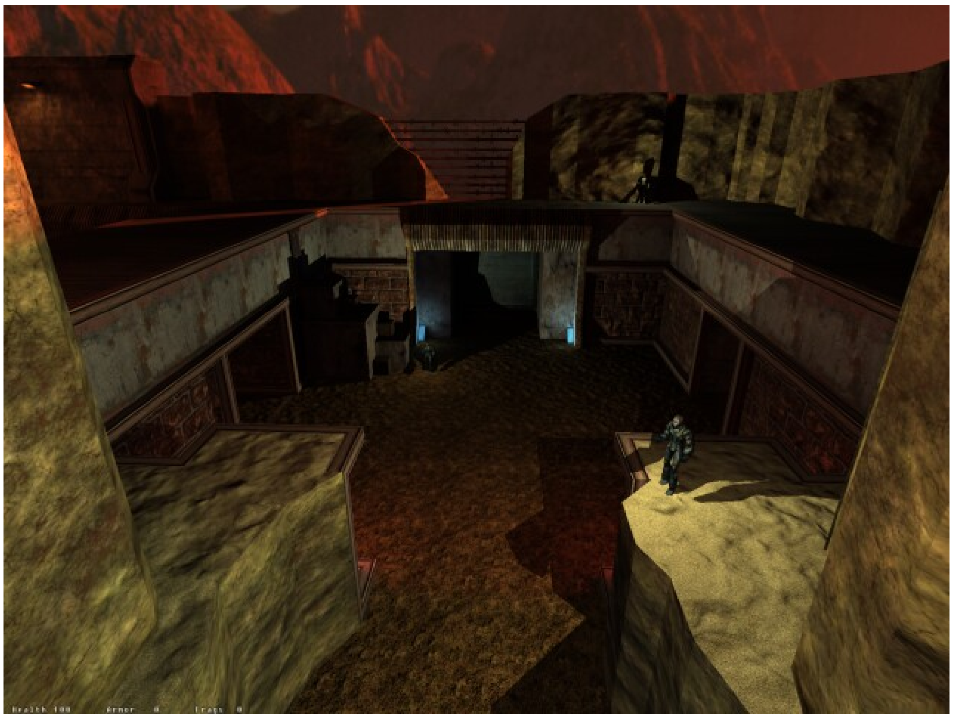


Figure 25(a): A reference shot of the second scene.

and the stone walls on the left employ normal-maps, although their effect is not clearly pronounced in this image.

Figure 24(b) shows the same scene, but rendered with SHL only. This is the situation *before* normal-maps were combined with SHL. For clarity of the SHL contribution, the diffuse-maps have intentionally been omitted.

Figures 24(c) to 24(e) present the same Spherical Harmonic Lighting as figure 24(b), but this time combined with normal-maps. The three images have been taken at different times of day, i.e. at different positions of the SHL light source, in order to demonstrate the effect clearly.

The structure and effect of the underlying normal-maps is clearly visible, both at the sand floor, the brick wall, and the stone walls in the background.

Second Series

As with the first series, the second series starts with a reference screenshot of the scene, see figure 25(a). Once more, the contribution of the normal-maps is well visible at the structure of the near sand floor and the far stone walls.

Figures 25(b) and 25(c) present the same scene with Spherical Harmonic lighting only. Compare that to figures 25(d) and 25(e), that additionally take normal-maps into account.

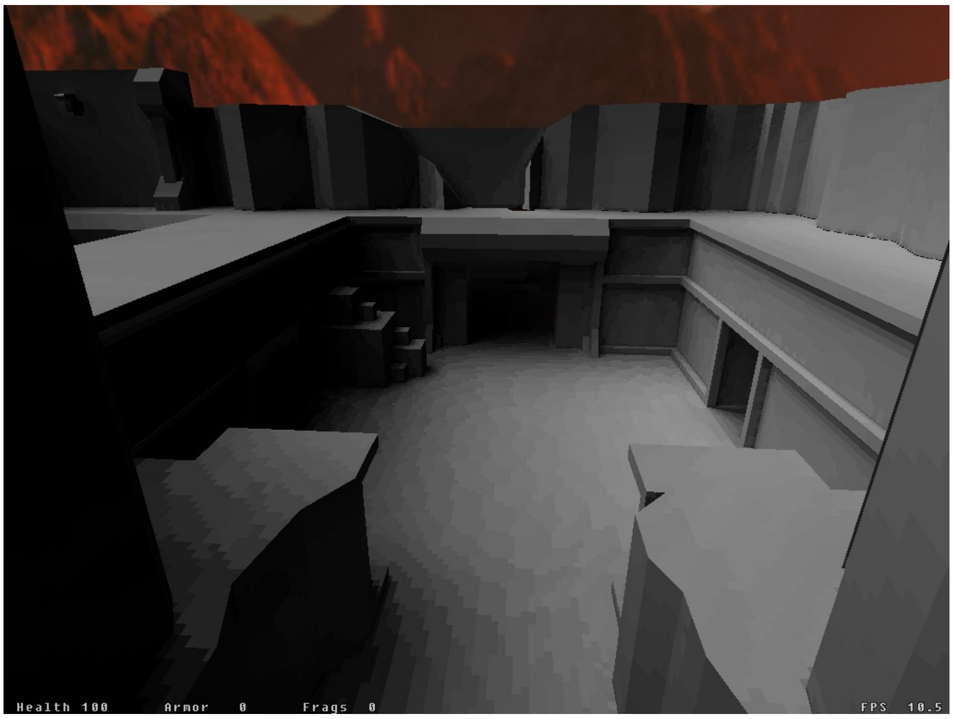


Figure 25(b): SHL-maps only (no normal-mapping and no color modulation).

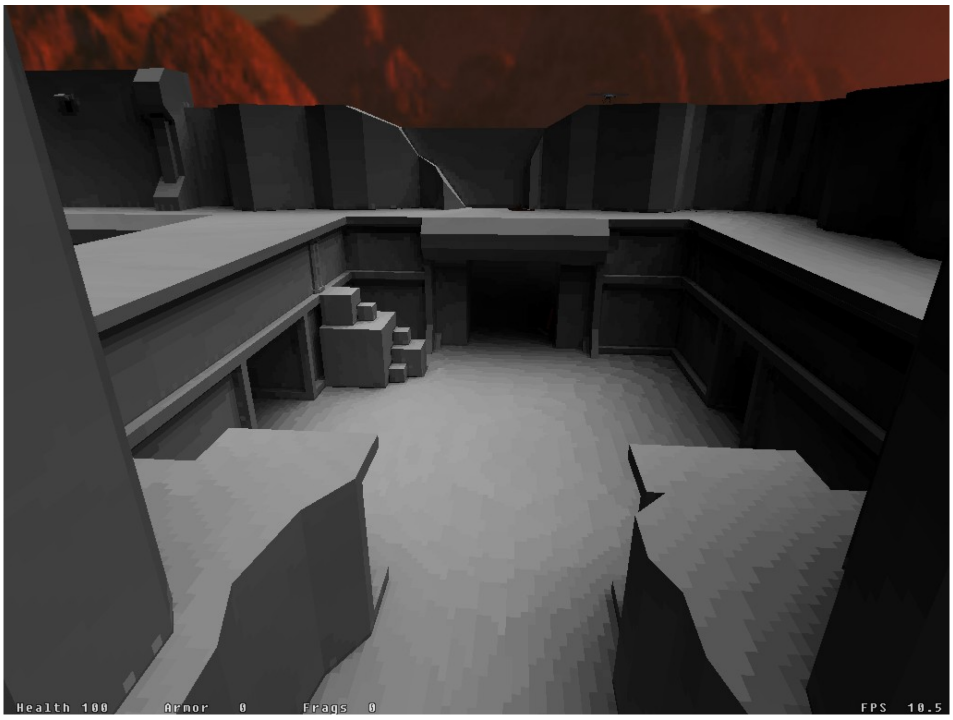


Figure 25(c): SHL-maps only (no normal-mapping and no color modulation).

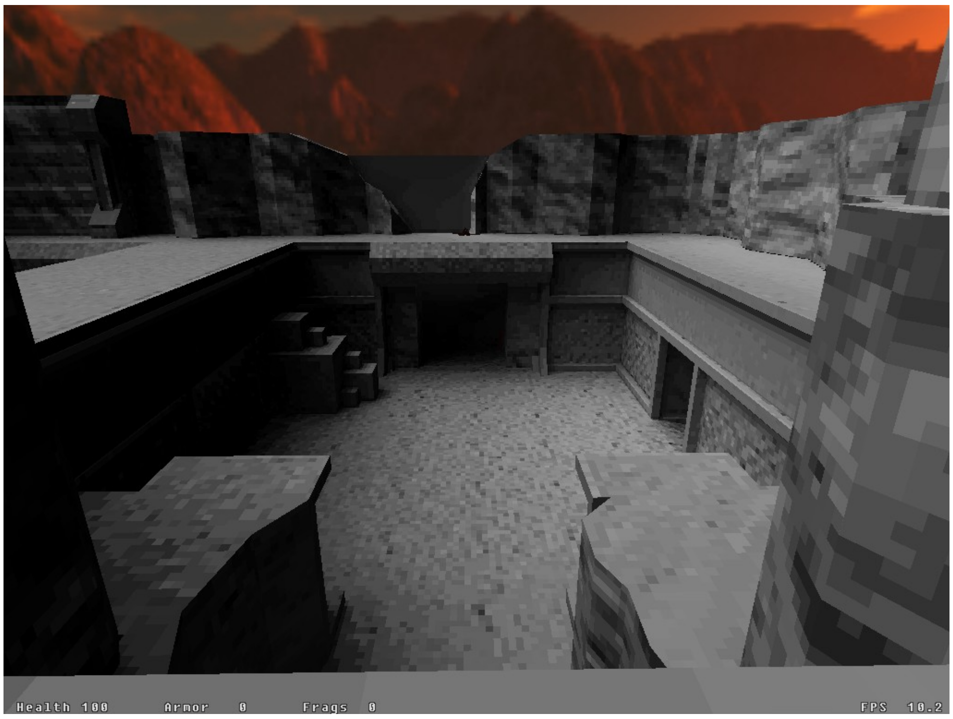


Figure 25(d): SH lighting with normal-maps taken into account.

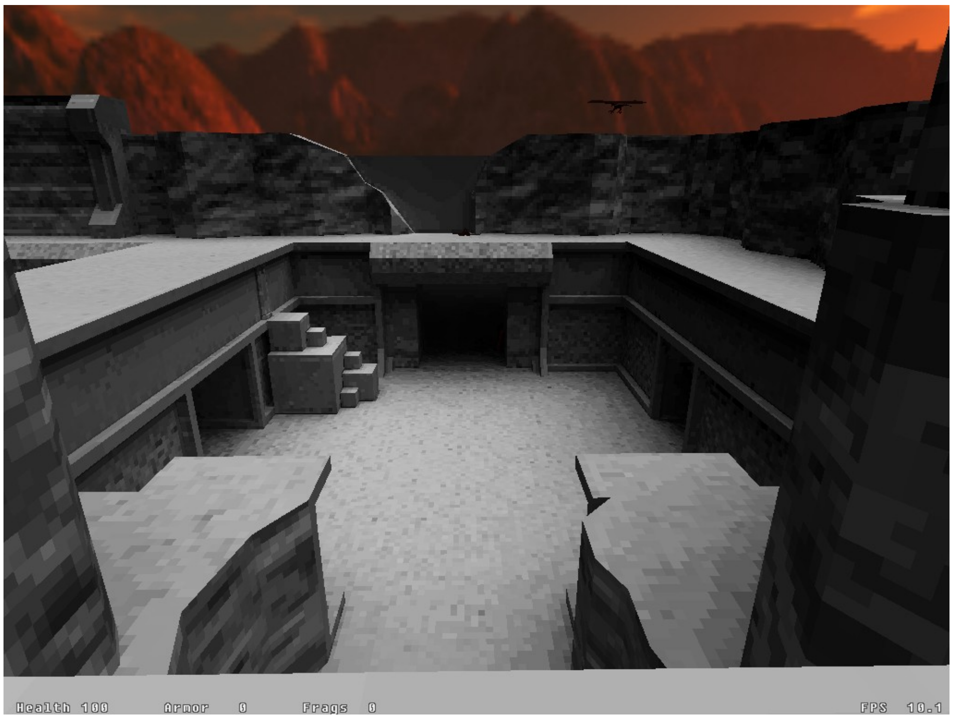


Figure 25(e): SH lighting with normal-maps taken into account.

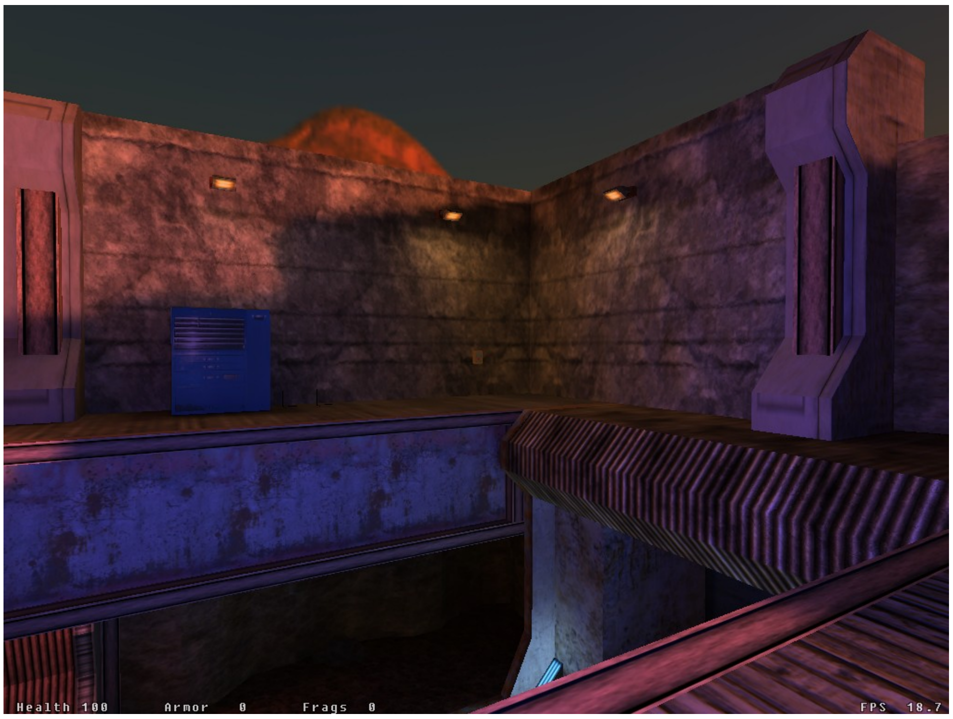


Figure 26(a): A reference shot of the third scene.

As a side-effect, this series of images demonstrates another feature of SH lighting: The blotchy appearance of scenes that are lit with SH lighting from compressed SH coefficients as detailed in section 5.6 is broken by the irregularities that are introduced by perturbed normal vectors. This is especially noticeable on the large floor surface in the above mentioned figures.

Third Series

The third series is organized like the previous two: Figure 26(a) provides a common impression of the scene, whereas figures 26(b) and 26(c) present the scene with SHL lighting, both with and without normal-mapping.

In figure 26(d), I've repeated 26(a) and 26(c), as their minification and direct facing for better comparison demonstrates very clearly the results of implementing normal-mapping for SH lighting.

5.8 SHL Filtering

The final subject of investigation within the scope of this thesis was the question if and how SH rendering output can be filtered.

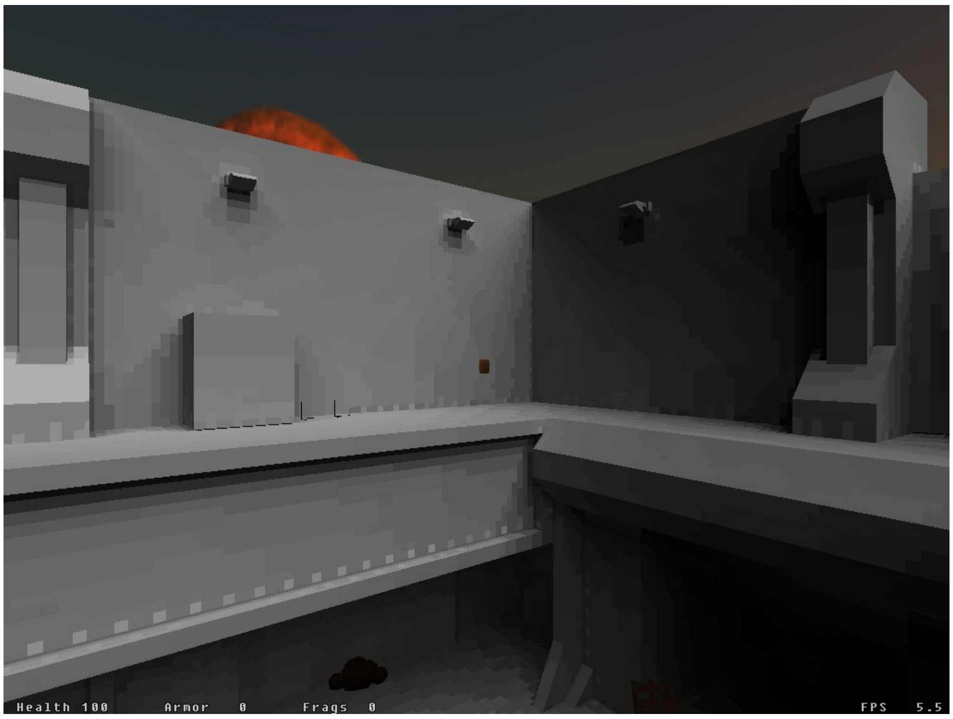


Figure 26(b): SHL-maps only (no normal-mapping and no color modulation).

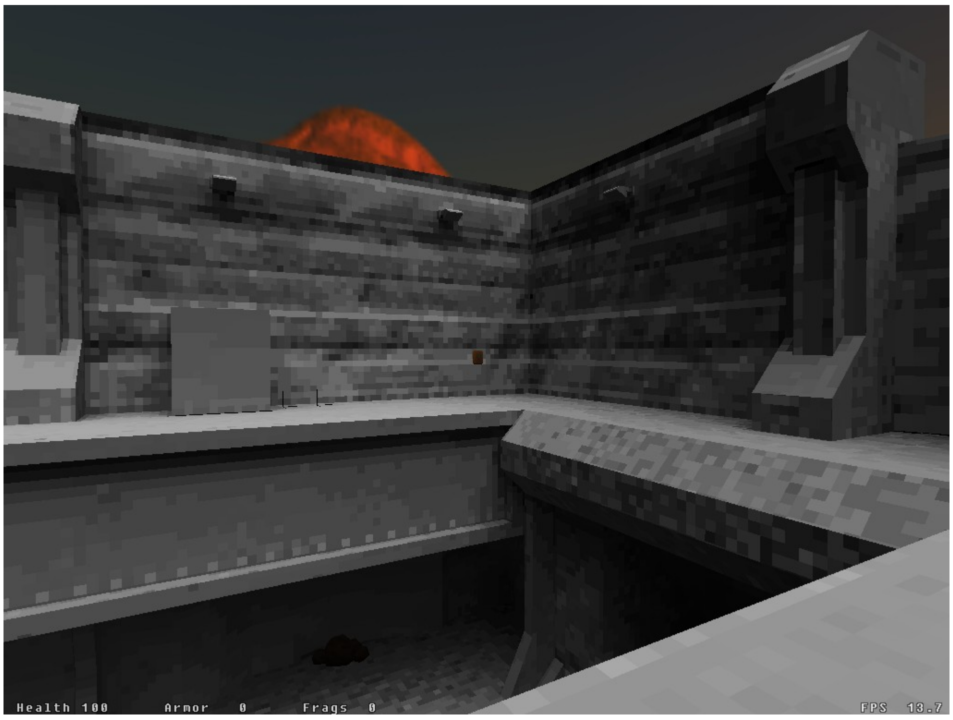


Figure 26(c): SH lighting with normal-maps taken into account.

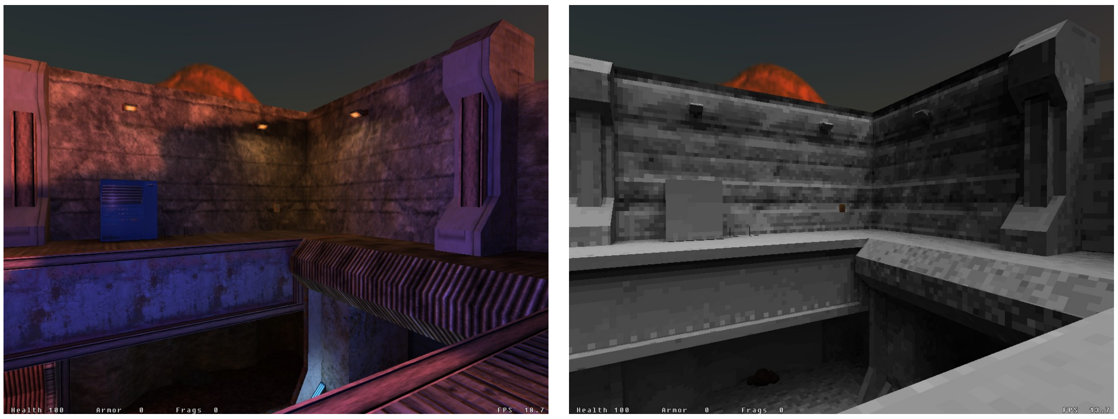


Figure 26(d): Scaled down repetitions of figures 26(a) and 26(c) that have been arranged for direct comparison.

5.8.1 Motivation and Approach

The matter was mostly motivated by the “blotchy” appearance of the rendering output that occurred as a result of SH compression with representative vectors, with the goal to at least render the effect a little milder.

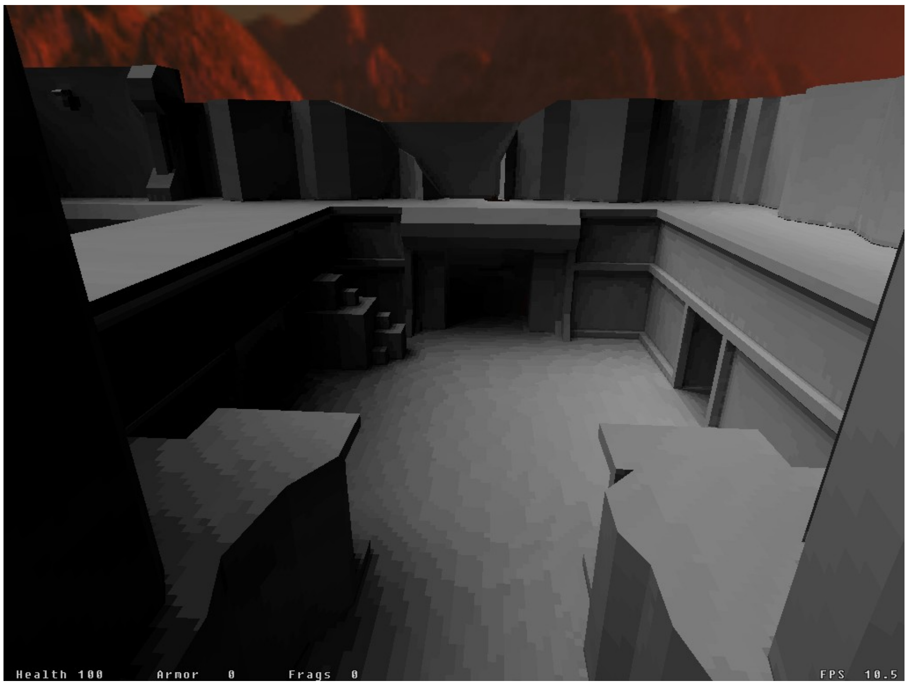
Unfortunately, achieving hardware-supported, perspective correct filtering, is inherently difficult in the context of the above presented SHL (de-)compression routines: In order to obtain the SH coefficients, a two-step dependent texture lookup is required, first into the index texture, and with the resulting value into the table (texture) of encoded representatives. This in turn renders all use of hardware-provided filtering methods other than “nearest” futile.

Therefore, the decision was made to simply employ bilinear filtering by quadruple multi-sampling: Each fragment is computed as the average of four samples, taken at quarters of an SHL-map element offset in each direction.

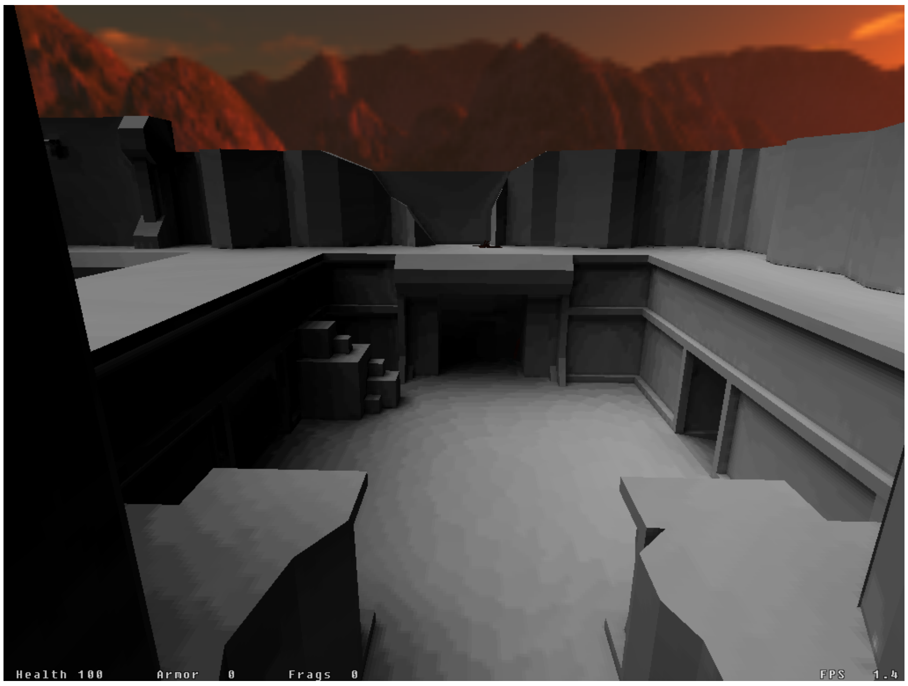
5.8.2 Results

Figure 27 shows two images of the same scene. Both images are based on the same compressed(!) SH data. Figure 27(a) shows the original scene, figure 27(b) is the same, except with filtering enabled. Figure 28 shows additional rendering results from the filtering by multi-sampling, where the effect is clearly visible.

Performance-measurements indicated that the frame-rate with the quadruple subsampling indeed quartered. As can be expected from table 3, all frame-rates that I measured in the same test scenarios were below 2 frames per second (usually between 1,5 and 1,8). While the visual results are promising, it remains a matter of faith that either faster filtering methods or faster hardware can bring the performance of this approach into acceptable bounds.

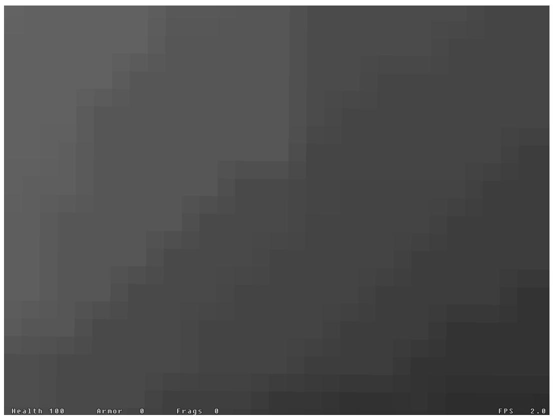


(a) Without SHL filtering.

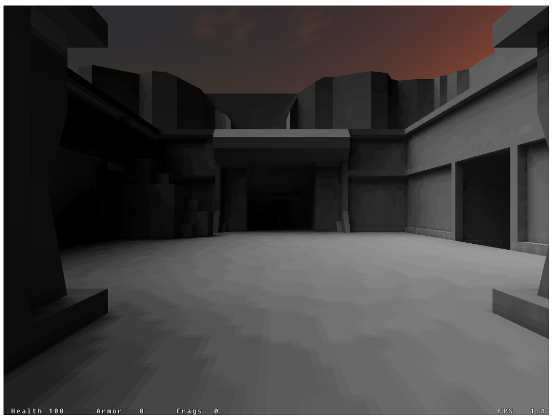


(b) With SHL filtering.

Figure 27: The same scene twice, once before and once after filtering was employed.



(a)



(b)

Figure 28: More results from SH filtering.

6 Implementation

6.1 Introduction to the Ca3D-Engine

The entire theory that is presented in this thesis has been implemented in the framework of the Ca3D-Engine in clearly separated, well-defined modules.

Before this thesis was begun, the Ca3D-Engine qualified as the framework for implementation due to its existing features at that time: Its cross-platform code proposed a reasonable development both on the Windows as well as the Linux platforms, and its underlying, optimized data structures for storing and rendering scene descriptions proved to be ideal ground for implementing all of the presented lighting methods.

In order to facilitate the discussion in other sections, I will provide a brief introduction into the internals of the Ca3D-Engine here. Please note that familiarizing yourself with the Ca3D-Engine also from a *user, artist, or developer's* perspective is highly recommended for the purposes of this thesis. This is best achieved by visiting the Ca3D-Engine website at `http://www.Ca3D-Engine.de`, where demo and development-kit downloads are provided, along with exhaustive documentation (see [Fuc04c] and [Fuc04a]).

6.1.1 The Preprocessing Pipeline for Worlds

A scene for the Ca3D-Engine is commonly called a *World File*. World files consist both of static, precomputed geometry, as well as lists of and references to dynamic components. The static geometry is typically the most prominent part of a world, created by convex polyhedra that form the walls, the floors, ceilings, and basically all other gross architecture. The dynamic parts are usually smaller *entities*. Entities are free to move within the static geometry in an arbitrary fashion. Typical examples for entities include the human players (both the local observer as well as other human players that join a world via a network link), all server-controlled lifeforms like opponents, animals, monsters, items, plants, and many more.

The World Editor

Worlds are initially created by the artist within a specialized world editor. Such editors permit modeling the architecture of a world with polyhedron-based building blocks, referred to as *brushes* in the nomenclature of the editor. The accumulated set of all brushes forms the static geometry of the world.

The same editor also permits populating the static architecture with entities. Entities retain a dynamic functionality during their entire lifetime, and can consist of brushes, special properties, or both.

The details about world editors are provided in their respective manuals, and also in [Fuc04a]. For now, it is important to note that worlds are the collection of static brushes (convex polyhedra) plus a set of dynamic entities.

Binary Space Partitioning

Maybe the most important step that the Ca3DE preprocessing compilers undertake is the hierarchical subdivision of a world into spatial convex cells.

In order to achieve this, the binary space partitioning compiler (*CaBSP*) performs the following steps:

1. Immediately after the world file from the editor has been loaded, CaBSP computes the bounding polygons of each polyhedron, which have originally been stored just as sets of intersecting planes. The polygons are flat, 2-dimensional surfaces, embedded in 3-dimensional space. The original polyhedra are kept as auxiliary structures, but they are no longer needed for essential computations.

2. A binary space partitioning (BSP) tree is built over the set of the resulting surfaces. As finding an optimal tree is exponentially expensive, a clever split-choosing heuristic is employed in order to obtain a good tree in $O((\text{total }\#\text{surfaces})^2)$ time. The BSP split planes are chosen from the set of planes of the remaining surfaces. As a result, all surfaces eventually get stored in a BSP tree node plane. Moreover, we also assign a list of surfaces to each leaf. This list is composed of the surfaces that are in the node planes that form the leaf and that simultaneously “touch” the leaf.

3. Then the tree is “portalized”. Portals are the regions where two adjacent leaves touch each other but are not separated by a regular solid surface. Portals are also always planar surfaces. They essentially define where one can pass from one leaf into the neighbouring leaf. This property makes them extremely useful for many subsequent processing steps, and even for real-time portal-based rendering techniques.

4. Using a proper starting point, the world is next flood-filled by walking from one leaf to the next, passing through portals. This reliably identifies regions in the world that are never accessible by observers (assuming they cannot walk through walls). Throwing away all surfaces that cannot be reached by this kind of flood fill does not change the architectural impression for the observer, but reduces the surface count often by more than 50%, yielding a big performance win. This is also a big improvement over the Quake series of engines, which used to flood fill from outside in, yielding far less optimal results than the fill from inside-out as in the Ca3D-Engine.

5. After the removal of unused surfaces, the remaining surfaces are split whenever two of them intersect. Then, steps 2 to 4 are repeated, because removing surfaces essentially invalidates the previously created BSP tree (which also becomes suboptimal after the underlying geometry has been modified).

6. Now the remaining surfaces span the minimum overall surface of the specified architecture. As the surfaces might have become very fragmented in the previous

steps, we now add a merge step that merges two surfaces whenever they are adjacent, coplanar, and the result is also a convex surface. This once more enforces a repetition of steps 2 to 4, where step 4 is mostly required for informational purposes, not for removing additional geometry.

This procedure is actually performed twice, once for the draw hull of the world, and once for the clip hull, as these two hulls are generally different (e.g. you can see through glass, but cannot walk through it, or you can walk through water, but cannot see the ground.)

Finally, the CaBSP compiler also covers every surface with an empty, dummy light- and SHL-map. That means that every surface of the world is already prepared at this point to receive light-maps and SHL-maps. Moreover, the memory both in RAM as well as on disk (within the output file) is allocated for light-maps (filled with pure white data as dummy light-maps) in *full depth even at this point.* This is not so bad though, as in nearly all cases the CaLight tool is run to overwrite these dummies with meaningful light colors. SHL-maps are better, because their number of bands is kept dynamically changeable at a later time. Therefore, CaBSP prepares everything for SHL-map storage for each surface, but only inserts a 0-band SHL-map, keeping the memory overhead initially negligible.

Computing the Potentially Visibility Set

The second big step in preprocessing worlds for the Ca3D-Engine is to determine which leaves of the previously created BSP tree can potentially be seen from any given leaf. The result of these computations is commonly called the *Potential Visibility Set* (PVS) of a world, determined by the CaPVS compiler. The purpose of the PVS is almost solely optimization for increasing performance, *as PVSs are a great method to determine very quickly (in $O(1)$ time) the (gross) relevance of objects to each other on many occasions.* Both the engines' real-time rendering as well as all subsequent compile tools for light-maps and SHL-maps profit significantly from PVS information.

Computing PVSs is a nontrivial task though. Teller [Tel92] describes a method that is mathematically thorough, but also mathematically and computationally challenging. His methods starts by considering portal sequences. Teller's portals are conceptually the same as those constructed by CaBSP in the previous step. Portals also have an orientation in the sense that they permit passage of light or objects only from one side to the other, pretty much like a one-way road does. He then determines all[5] possible ways to "walk" from a given BSP leaf L_1 to another leaf L_2, and thereby writes down the portals that have been passed during that walk. One such set of recorded portals is called a "portal sequence". The next step is to determine if at least a single stabbing line through the entire portal sequence exists. If it does, L_1 can see L_2 (and vice versa), otherwise the two leaves cannot see each other. Teller determines the existence of a stabbing line in a mathematically and computationally complex manner: He first transforms all edges of all

[5]Only "reasonable" ones of course, i.e. the number of portal sequences can easily be cut by simple checks against a bounding box etc.

portals of in the sequence into the space of *Plücker coordinates*. The resulting numbers then form a system of linear inequalities in 6-dimensional space, that can be solved by methods of linear optimization. During my own tests with Potential Visibility Sets my own implementations of Teller's theory suffered from degeneracies, rounding errors in the Simplex method, and the sheer computational demand. Although portal sequences are typically short (e.g. 10 portals on average), each such sequence yields normally a system of linear inequalities of 40 and more constraints in 6 variables. Multiplying this with the number of possible portal sequences between a pair of leaves yields a very expensive algorithm for determining the visibility just between *a single pair* of leaves! Multiply this again with essentially $O(\#\text{leaves}^2)$ for all pairs, and the entire matter becomes practically infeasible.

For CaPVS, I therefore developed another strategy that also operates on portal sequences, but determines the existence of stabbing lines differently, by a method that works in the usual 3-dimensional space: Given an arbitrary portal sequence, the first portal is considered as a unidirectional, planar light-source. The light passes through the other portals (if the portal is passed in its proper direction), which shape it into a light pyramid. This method is mathematically not fully equivalent to Tellers approach. Rather, it tends to conservatively overestimate the PVS on rare occasions. I've never had the opportunity to reliably compare the differences in the results of both methods (rounding errors and the far too expensive Plücker approach dilute any results), but I submit the overestimation of my method is significantly less than 1%.

Trivial implementations of either Teller's method or the 3D-method of CaPVS tend to construct their portal sequences by exploring neighboring leaves in a strictly recursive manner. This, however, leads to an exponential running time per leaf, and thus for the overall algorithm. It also does not permit exploiting previously computed knowledge, i.e. no advantage can be taken from the previously computed fact "L_2 is visible from L_1" when later the question arises if L_1 is visible from L_2. (This seems counterintuitive, and has its roots in the recursive proceeding of the algorithm – it might later discover additional visibility that it can only retrieve without taking such abbreviations.)

Fortunately, the overall algorithm can be changed such that its runtime moves from exponential behaviour to essentially polynomial behaviour. This is achieved by changing it from exploring the world and recording all found visibility relationships to simply and strictly answering the question "Can L_1 see L_2?". Doing so has several advantages: First, by learning that L_1 can see L_2, we know that also the reverse visibility is established: L_2 can see L_1. Second, when the algorithm works on determining the visibility between L_1 and L_2, and as a by-product finds the mutual visibility of other pairs of leaves (e.g. all combinations of leaves that are spatially located between L_1 and L_2), explicit tests for all those pairs can be saved later. Third, if we run a very quick and simple ray-sampling test on all pairs of leaves as a first step in the program, we may save running the more expensive main algorithm on all previous successfully tested pairs later.

The latter, essentially polynomial method has been implemented in the CaPVS compiler. It is numerically extremely stable, and runs in long (often several hours for complex worlds), but reasonable time.

Precomputing Lighting

At this point, the next and final steps in compiling a map include precomputing the radiosity-based and Spherical Harmonics-based lighting. These are the essential topics of this thesis, and have therefore been presented in their own sections.

6.1.2 The Ca3DE World File Format

For compiling a world for Ca3DE, only the CaBSP compile step is mandatory. It creates full-bright lighting and full-visibility PVS information. All other compiler runs are optional, as is their order. The respective compilers overwrite the defaults in a world file with their computed values.

6.2 Implementation of Lighting with Light-Maps

The implementation of lighting with light-maps reduces to considering the light-maps as the diffuse lighting component, and as such modulating them with the regular surface textures which define the diffuse reflectivity of the surface.

With the OpenGL 1.1 API [WNDO99], the multiplication is performed as a two-pass blending operation: The first pass renders the diffuse textures as normal, and the second pass renders the light-maps with blending enabled and `glBlendFunc(GL_DST_COLOR, GL_ZERO)`.

With OpenGL 1.2 and newer APIs [WNDO99], multi-texturing can be exploited to achieve the same result in a single pass.

On programmable GPUs with vendor specific API extensions (e.g. [HM01]) or high-level shader languages like Cg [FK03], HLSL or GLSL [Ros04], the multiplication can conveniently be performed in custom code.

The same concepts are true for the appropriate DirectX equivalents and other graphics APIs [FvDFH90].

6.3 Implementation of Dynamic Lighting on Dedicated 3D Hardware

6.3.1 The Acquisition and Organization of Artwork

The first side-effect that any implementors of dynamic lighting encounter is the fact that the old artwork that they used to use is no longer sufficient to support the new technology. While earlier it was sufficient to cover polygon surfaces with texture images that were pre-lit like photographs, possibly enhanced with light-maps (see section 3), more complex surface descriptions are required for dynamic lighting. The surface descriptions data is therefore stored in (sets of) texture map images that store the data either directly or in some kind of color-encoded form, whatever is best suited.

Hardware-accelerated Phong lighting typically requires the presence of diffuse-maps, normal-maps, specular-maps and luminance-maps. The creation of these map images is mostly an artistic problem, where the artists have to understand the use and purpose of each image within the Phong lighting equation.

The individual files are typically kept in groups of related image file sets that are distinguished by additional file suffixes. Another way of file organization is to describe the file relationships in separate material definition scripts that are normally handled by Material Systems.

Nomenclature and Organization of new Texture Images

For Ca3DE, I have decided to suffix the base file names with `_diff` for diffuse-maps, `_norm` for normal-maps, `_bump` for height-maps, `_spec` for specular-maps, and `_luma` for luminance-maps. Each of these suffixes is further followed by one of the supported file format suffixes like `.png`, `.tga`, or `.bmp`. Therefore, examples for complete texture image file names in this naming scheme are `BaseDoor1_diff.png`, `Barrel07_norm.tga`, or `Wall28_luma.png`.

For organizing all these texture files on disk, it is desirable to organize them hierarchically in an own directory and arbitrary subdirectories. The problem with this kind of organization was that the level editors that I used to create Ca3DE worlds could only deal with images files that were stored in proprietary "WAD" files that in turn could only handle file names without path specifications and with lengths of less than 16 characters. Really the best and most expensive solution to this problem was to create a new editor for Ca3DE editing, but that was far too complex a subproblem to be solved within the scope of this thesis.

Instead, I organized all textures as desired in a directory and subdirectories. The only condition that the textures have to satisfy is that no pair of two texture files must have a common name, not even if they are stored within different subdirectories. Then I use a Perl script that first scans the entire textures directory (and recurses into subdirectories), and records all occurrences of `*_diff.*` files. The found files are then converted into WAD file format and stored in a newly created WAD file. Thereby, both their path as well as their suffix is omitted in order to account for the 16 character limitation.

This WAD file can then be used for world editing with the existing editors. Later, during level load time, the Ca3D-Engine reverses the process by scanning the textures directory recursively for the previously stored short texture name, which is extended with the well-known suffixes as described above.

Aspects of Normal-Map creation

Normal-maps are somewhat interesting, as there are several complex ways to create them. One method that requires a lot of work but yields optimal results is to create the shape of the surface as a highly detailed polygonal model in a 3D modelling program. The model's surface is then orthogonally projected onto a much lower detail surface. The projection allows the recording of the high-polygonal surface directly in a normal-map that covers the low-polygonal surface and is the receiver of the projection. This method works both for individual (tiling) texture images (especially for those that have sharp and well defined contours as e.g. technical panels) as well as for highly complex models like human characters.

The second (and simpler) technique for creating normal-maps is to first create a grayscale height-map that represents the height of the surface. A height-map can either be created by geometric means similar to the first method mentioned above, but is often also hand-painted by artists. As hand-painting a height-map requires an enormous spatial sense, making good height-maps is very difficult and works best with irregular, organic, or high frequency surfaces. Examples include sand dunes, bubbles, scratches or small holes.

Best results are achieved by employing both methods for creating a single normal-map simultaneously: First, a normal-map is created that contains the well-defined contours with the first method. Another normal-map is then created with the less well-defined shape elements, e.g. scratches and bullet holes. Finally, both normal maps are combined into a single resulting normal map.

6.3.2 Rendering

The implementation of the Phong shading model on programmable GPUs, with the help of high-level languages such as NVidias Cg, is straightforward. As for almost all combinations of diffuse-, normal-, specular-, luminance- and light-maps specific Cg vertex- and fragment-shaders exist for both the ambient and the per-light-source rendering passes, I do not reproduce them in full detail here. Rather, here is a short and simple example of such a fragment shader for NV3X GPUs that handles the per-light-source contribution of a material that comes with a diffuse-, normal-, and specular-map:

```
void main(in float2   InTexCoord     : TEXCOORD0,
          in float3   InEyeVector    : TEXCOORD1,
          in float3   InLightVector  : TEXCOORD2,
          in float3   InLightVectorA : TEXCOORD3,
         out float4   OutColor       : COLOR,
     uniform float3   LightColor,
     uniform sampler2D DiffuseMapSampler,
     uniform sampler2D NormalMapSampler,
     uniform sampler2D SpecularMapSampler)
{
    const float3 EyeDir   =normalize(InEyeVector);
    const float3 LightDir =normalize(InLightVector);
    const float3 Halfway  =normalize(EyeDir+LightDir);
    const float3 Normal   =(tex2D(NormalMapSampler, InTexCoord).xyz-0.5)*2.0;

    // IMPORTANT: Note that 'InLightVector' and 'InLightVectorA' are
    // ENTIRELY DIFFERENT!
    // Only 'InLightVectorA' is good for attenuation computations (world space),
    // while 'InLightVector' (local tangent space) takes SmoothGroups into account!!!
    // So they must never be collapsed, even if the profile permitted that!
    const float  Atten    =saturate(1.0-length(InLightVectorA));

    const float  diff     =            saturate(dot(Normal, LightDir));
    const float  spec     =diff>0.0 ? saturate(dot(Normal, Halfway )) : 0.0;

    const float4 DiffuseC =tex2D(DiffuseMapSampler, InTexCoord);
    const float4 SpecularC=tex2D(SpecularMapSampler, InTexCoord);
    const float4 LightC   =float4(LightColor, 0);

    OutColor=Atten*LightC*(diff*DiffuseC + pow(spec, 32)*SpecularC);
}
```

Listing 1: A Cg fragment program for NV3X GPUs for the diffuse and specular lighting contributions with normal-mapping.

6.4 Implementation of Spherical Harmonic Lighting

6.4.1 Native SHL Lighting

The implementation of native SHL lighting as explained in section 5.5 is straightforward, but the inherent limits of GPU programs require having separate per-pixel programs for each number n of SH bands. I therefore provide implementations for the most common cases of 2 and 4 SH bands (4 and 16 coefficients, respectively). The section about compressed SHL data will show a solution that can deal with fluctuating n. For reference, here is the per-pixel program for rendering SH lighting with 16 coefficients:

```
void main(in float4   InColor             : COLOR,
          in float2   InTexCoord_Diff     : TEXCOORD0,
          in float2   InTexCoord_SHLMap   : TEXCOORD1,
          out float4  OutColor            : COLOR,
     uniform float4   LightSourceCoeff1,          // Light source coeffs  0.. 3
     uniform float4   LightSourceCoeff2,          // Light source coeffs  4.. 7
     uniform float4   LightSourceCoeff3,          // Light source coeffs  8..11
     uniform float4   LightSourceCoeff4,          // Light source coeffs 12..15
     uniform sampler2D DiffuseMapSampler,
     uniform sampler2D SHLMapSampler1,            // For SHL coeffs  0.. 3
     uniform sampler2D SHLMapSampler2,            // For SHL coeffs  4.. 7
     uniform sampler2D SHLMapSampler3,            // For SHL coeffs  8..11
     uniform sampler2D SHLMapSampler4)            // For SHL coeffs 12..15
{
    const float4 DiffuseC=tex2D(DiffuseMapSampler, InTexCoord_Diff);

    const float4 SHLMap1C=tex2D(SHLMapSampler1, InTexCoord_SHLMap)-0.5;
    const float4 SHLMap2C=tex2D(SHLMapSampler2, InTexCoord_SHLMap)-0.5;
    const float4 SHLMap3C=tex2D(SHLMapSampler3, InTexCoord_SHLMap)-0.5;
    const float4 SHLMap4C=tex2D(SHLMapSampler4, InTexCoord_SHLMap)-0.5;

    const float4 v=4.0*(dot(SHLMap1C, LightSourceCoeff1)+
                        dot(SHLMap2C, LightSourceCoeff2)+
                        dot(SHLMap3C, LightSourceCoeff3)+
                        dot(SHLMap4C, LightSourceCoeff4));

    const float  Result  =clamp(v, 0.0, 1.0);
    const float4 Result4 =float4(Result.xxx, 1.0);

 // OutColor=Result4;      // For debugging.
    OutColor=DiffuseC*InColor*Result4;
}
```

Listing 2: Rendering SH lighting with 16 coefficients.

6.4.2 Procedurally generated Cg Shaders for Compressed SHL

The implementation of decompressing the precomputed compressed SHL data requires a custom fragment program that runs on the graphics processor. My wish code had looked like the following listing, which however only shows nonfunctional pseudo code due to the limitations of the Cg language profiles for NV3X GPUs:

```
1   void main(in float4    InColor           : COLOR,
              in float2    InTexCoord_Diff   : TEXCOORD0,
              in float2    InTexCoord_SHLMap : TEXCOORD1,
             out float4    OutColor          : COLOR,
5        uniform float4    LightsourceSHLCoeffs[], // Lightsource SHL coeffs in 4−tuples.
         uniform float     NrOfColumns, // Number of vector columns in the SHLCoeffTable.
         uniform float     TableWidth,  // Width of the SHLCoeffTable, in pixels.
         uniform int       NrOfPixels,  // =ceil(n∗n/4), #pixels per representative vector.
         uniform sampler2D DiffuseMapSampler,
10       uniform sampler2D IndicesSampler,
         uniform sampler2D SHLCoeffTableSampler)
    {
        const float4 DiffuseC=tex2D(DiffuseMapSampler, InTexCoord_Diff);
        float4       Index   =tex2D(IndicesSampler, InTexCoord_SHLMap);
15      float        Result  =0.0;

        // Index.r specifies the row number 0..255, scaled by 1/255 (!) to range 0..1.
        // Index.g specifies the column number, also scaled by 1/255 (!) to range 0..1.
        // However, we need it to be scaled by NrOfPixels/TableWidth.
20      for (int i=0; i<NrOfPixels; i++)
        {
            // This can be optimized by computing some values outside of the loop.
            const float2 LookupPos=float2((Index.g∗255.0∗NrOfPixels+i)/TableWidth,
                                          Index.r∗255.0/256.0);
25          const float4 FourCoeffs=4.0∗(tex2D(SHLCoeffTableSampler, LookupPos)−0.5);

            Result+=dot(FourCoeffs, LightsourceSHLCoeffs[i]);
        }

30      Result=clamp(Result, 0.0, 1.0);

     // OutColor=float4(Result.xxx, 1.0);        // For debugging.
        OutColor=DiffuseC∗InColor∗float4(Result.xxx, 1.0);
    }
```

Listing 3: Wish-code in pseudo Cg.

Note the careful transfer from the Index variables .r and .g values to the LookupPos value in lines 23 and 24. These computations seem to be simple in hindsight, but initially it took me considerable effort to figure them out. Also, debugging is often a non-trivial task with GPU programs (there is no `printf()` function to print out intermediate values) and requires a lot of creativity and patience. (E.g. I frequently transformed intermediate results into color values, emphasizing the numerically interesting ranges, in order to "see" their correctness visually. This is still much harder than seeing them numerically printed on screen, though.) Unfortunately, some important constructs in the above (pseudo-)code are not supported by any current Cg language profile: neither arrays of unknown size (line 5), nor variable array indexing (line 27), nor arbitrary for-loops (line 20) are supported.

In order to make the above code conform to e.g. the NV3X Cg profiles[6], I employed a loop-unrolling technique, similar to the old optimization method from assembly programming that was employed mostly on processors with no internal cache:

```
void main(in float4    InColor            : COLOR,
          in float2    InTexCoord_Diff    : TEXCOORD0,
          in float2    InTexCoord_SHLMap  : TEXCOORD1,
         out float4    OutColor           : COLOR,
     uniform float4    LSC[25],         // LightSource SHL Coeffs in RGBA 4-tuples
                       // (25 elements provide enough space for up to 10 SHL bands).
     uniform float     NrOfColumns,     // Number of vector columns in the SHLCoeffTable.
     uniform float     NrOfPixels,      // Number of pixels (4-tuples) per SHL vector.
     uniform float     TableWidth,      // Width of the SHLCoeffTable, in pixels.
     uniform sampler2D DiffuseMapSampler,
     uniform sampler2D IndicesSampler,
     uniform sampler2D S)               // "SHLCoeffTableSampler"
{
    const float4 DiffuseC=tex2D(DiffuseMapSampler, InTexCoord_Diff);
    float4 i=tex2D(IndicesSampler, InTexCoord_SHLMap); // The index.
    float  r=0.0;                                       // The result.

    // i.r specifies the row number 0..255, scaled by 1/255 (!) to range 0..1.
    // i.g specifies the column number, also scaled by 1/255 (!) to range 0..1.
    // However, we want it to be scaled by NrOfPixels/TableWidth, see below.

    // Reverting to unrolled loops ...
    i.r=i.r*255.0/256.0;                       // Correct the scale.
    i.g=i.g*255.0*NrOfPixels/TableWidth; // Correct the scale.

    // The coordinate distance between two horizontally neighbored pixels.
    const float OfsX=1.0/TableWidth;
```

[6]The NV3X profiles were the most advanced Cg profiles at the time of writing, although newer NVidia and ATI GPUs are already available. However, my future implementations will all employ OpenGL 2.0 rather than Cg for fragment-shaders, but I expect the principle statements of this section to remain valid.

```
    if ( 0<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 0]); i.g+=OfsX; }
    if ( 1<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 1]); i.g+=OfsX; }
    if ( 2<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 2]); i.g+=OfsX; }
    if ( 3<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 3]); i.g+=OfsX; }
    if ( 4<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 4]); i.g+=OfsX; }
    if ( 5<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 5]); i.g+=OfsX; }
    if ( 6<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 6]); i.g+=OfsX; }
    if ( 7<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 7]); i.g+=OfsX; }
    if ( 8<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 8]); i.g+=OfsX; }
    if ( 9<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[ 9]); i.g+=OfsX; }
    if (10<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[10]); i.g+=OfsX; }
    if (11<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[11]); i.g+=OfsX; }
    if (12<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[12]); i.g+=OfsX; }
    if (13<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[13]); i.g+=OfsX; }
    if (14<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[14]); i.g+=OfsX; }
    if (15<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[15]); i.g+=OfsX; }
    if (16<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[16]); i.g+=OfsX; }
    if (17<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[17]); i.g+=OfsX; }
    if (18<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[18]); i.g+=OfsX; }
    if (19<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[19]); i.g+=OfsX; }
    if (20<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[20]); i.g+=OfsX; }
    if (21<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[21]); i.g+=OfsX; }
    if (22<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[22]); i.g+=OfsX; }
    if (23<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[23]); i.g+=OfsX; }
    if (24<NrOfPixels) { r+=dot(4.0*(tex2D(S, i.gr)-0.5), LSC[24]);             }

    const float Result=clamp(r, 0.0, 1.0);

 // OutColor=float4(Result.xxx, 1.0);          // For debugging.
    OutColor=DiffuseC*InColor*float4(Result.xxx, 1.0);
}
```

Listing 4: Initial code for the Cg NV3X profile.

The resulting code is fairly ungainly[7], but it *does* work on the NV3X Cg fragment shader profile.

The problem with the second fragment program is that it is either written for a fixed number of SH bands/coefficients, or that it is longer than necessary, unnecessarily wasting GPU processing time. Therefore, I finally came up with the solution of generating

[7]The printed code above got many variable names significantly shortened, in order to account for the limited page width of this book. The real code uses variables with more descriptive names. The real code is also even more complicated, for reasons that are irrelevant to this discussion (debugging, tone reproduction, etc.). These issues have also been omitted here. Please refer to the implementation for full details.

the code for the fragment shader procedurally. That is, the Cg fragment program's code is generated by C++ code at run-time. As the number of SH bands/coefficients is well known at run-time, the Cg code can be generated exactly to meet the requirements of the current and actual number of SH bands/coefficients, avoiding the problems of the previously presented fragment program. As all other Cg programs in the Ca3D-Engine, the resulting Cg fragment program is compiled and linked dynamically by the Cg compiler at run-time anyway. As compiling and linking at run-time is a necessary prerequisite for employing procedurally generated code, all technical assumptions are met for making this approach work. The following listing shows the relevant section of C++ code for generating the desired Cg fragment program:

```
const unsigned long NR_OF_SH_COEFFS=
    FaceT::SHLMapInfoT::NrOfBands*FaceT::SHLMapInfoT::NrOfBands;
const unsigned long NrOfPixels=(NR_OF_SH_COEFFS+3)/4;

static char DynamicCode[20000];

sprintf(DynamicCode,
  "void main(in float4    InColor            : COLOR,                              \n"
  "          in float2    InTexCoord_Diff    : TEXCOORD0,                          \n"
  "          in float2    InTexCoord_SHLMap  : TEXCOORD1,                          \n"
  "          out float4   OutColor           : COLOR,                              \n"
  "      uniform float4   LightsourceSHLCoeffs[%lu], // Light coeffs in 4-tuples.   \n"
  "      uniform float    NrOfColumns,  // # vector columns in the SHLCoeffTable.   \n"
  "      uniform float    TableWidth,   // Width of the SHLCoeffTable, in pixels.   \n"
  "      uniform sampler2D DiffuseMapSampler,                                       \n"
  "      uniform sampler2D IndicesSampler,                                          \n"
  "      uniform sampler2D SHLCoeffTableSampler)                                    \n"
  "{                                                                                \n"
  "    const float4 DiffuseC=tex2D(DiffuseMapSampler, InTexCoord_Diff);            \n"
  "    float4       Index   =tex2D(IndicesSampler, InTexCoord_SHLMap);             \n"
  "    float        Result  =0.0;                                                  \n"
  "                                                                                 \n"
  "    // Index.r specifies the row num 0..255, scaled by 1/255 (!) to range 0..1.  \n"
  "    // Index.g specifies the column num, also scaled by 1/255 (!) to range 0..1. \n"
  "    // However, we want it to be scaled by NrOfPixels/TableWidth, see below.     \n"
  "                                                                                 \n"
  "    // Still unrolling the for-loop, but this time exactly as needed!           \n"
  "    Index.r=Index.r*255.0/256.0;                // Correct the scale.            \n"
  "    Index.g=Index.g*255.0*%lu.0/TableWidth; // Correct the scale.                \n"
  "    const float PixelOffsetX=1.0/TableWidth; // Hor. distance betw. two pixels. \n",
  NrOfPixels>0 ? NrOfPixels : 1, NrOfPixels);

for (unsigned long PixelNr=0; PixelNr<NrOfPixels; PixelNr++)
{
    sprintf (DynamicCode+strlen(DynamicCode),
          "Result+=dot(4.0*(tex2D(SHLCoeffTableSampler, Index.gr)−0.5), "
          "LightsourceSHLCoeffs[%2lu]);",
          PixelNr);
    if (PixelNr+1<NrOfPixels) strcat(DynamicCode, " Index.g+=PixelOffsetX;");
    strcat (DynamicCode, "\n");
}

strcat (DynamicCode,
    "     Result=clamp(Result, 0.0, 1.0);                              \n"
    "                                                                  \n"
    "  // OutColor=float4(Result.xxx, 1.0);       // For debugging.   \n"
    "     OutColor=DiffuseC*InColor*float4(Result.xxx, 1.0);           \n"
    " }                                                                \n");

FragmentShader30_SHL_Compressed=UploadCgProgram(CG_PROFILE_FP30,
                                               DynamicCode);
```

Listing 5: C++ code for generating Cg code on demand.

In fact, the above final solution in listing 5 works due to the assumption that the `NrOfPixels`, or rather, the number of SH bands that are present in the world file, remains constant after loading the world. This is a very reasonable assumption, as the number of SH bands can only be changed by reprocessing a world with CaSHL.

The advantages and flexibility that is introduced by this final approach are subtle, but worthwhile: The `NrOfPixels` is now directly built into the fragment program, such that there is no need to keep it as an external C++ variable that must be set as a fragment program parameter on each binding of the program.

Moreover, this means that this approach is even more flexible than my non-compressed SH fragment shaders: Whereas I previously had to provide a separate, “hardwired” fragment program for all numbers of SH bands that I'd ever expect to load into the Ca3D-Engine (obviously a very inflexible situation), the above code can conveniently handle arbitrarily many SH bands that are within reasonable bounds.

6.4.3 Normal-Mapping and SHL

Due to the way how we algorithmically take normal-maps into account for our SHL computations, only the very first initialization steps of CaSHL need customization. After that, the normal-vectors are directly factored into the spherical functions that all subsequent code can treat unmodified. That is, no changes to the rest of CaSHL and no changes at all to the rendering modules were required in order to augment SH lighting to take normal-maps into account!

6.4.4 Implementing SHL Filtering

As the two-step dependent texture lookups that were used throughout section 5.6 make perspective correct filtering practically impossible with current 3D-graphics accelerator hardware, we simply employ bilinear filtering by quadruple multi-sampling. Listing 5 was augmented accordingly, and we obtain the following:

```
const unsigned long NR_OF_SH_COEFFS=
    FaceT::SHLMapInfoT::NrOfBands*FaceT::SHLMapInfoT::NrOfBands;
const unsigned long NrOfPixels=(NR_OF_SH_COEFFS+3)/4;

unsigned long PixelNr;
static char DynamicCode[30000];

sprintf(DynamicCode,
  " void main(in float4   InColor            : COLOR,                                   \n"
  "           in float2   InTexCoord_Diff    : TEXCOORD0,                               \n"
  "           in float2   InTexCoord_SHLMap  : TEXCOORD1,                               \n"
  "          out float4   OutColor           : COLOR,                                   \n"
  "      uniform float4   LightsourceSHLCoeffs[%lu], // Light coeffs in 4-tuples.        \n"
  "      uniform float    NrOfColumns, // # vector columns in the SHLCoeffTable.        \n"
  "      uniform float    TableWidth,   // Width of the SHLCoeffTable, in pixels.       \n"
  "      uniform sampler2D DiffuseMapSampler,                                           \n"
  "      uniform sampler2D IndicesSampler,                                              \n"
  "      uniform sampler2D SHLCoeffTableSampler)                                        \n"
  " {                                                                                   \n"
  "   const float4 DiffuseC=tex2D(DiffuseMapSampler, InTexCoord_Diff);                  \n"
  "   const float SOfs=0.25/%lu.0;   // 1/4pixel sample offset in texture space.        \n"
  "   const float4 Indices[4]=                                                          \n"
  "   {                                                                                 \n"
  "       tex2D(IndicesSampler, InTexCoord_SHLMap + float2(-SOfs, SOfs)),               \n"
  "       tex2D(IndicesSampler, InTexCoord_SHLMap + float2( SOfs, SOfs)),               \n"
  "       tex2D(IndicesSampler, InTexCoord_SHLMap + float2( SOfs, -SOfs)),              \n"
  "       tex2D(IndicesSampler, InTexCoord_SHLMap + float2(-SOfs, -SOfs))               \n"
  "   };                                                                                \n"
  "                                                                                     \n"
  "   float Total=0.0;                                                                  \n"
  "                                                                                     \n"
  "   // A simple optimization. This code unfort. crashes the Cg compiler, though!      \n"
  "   if ( all(Indices[0]==Indices[1]) &&                                               \n"
  "        all(Indices[1]==Indices[2]) && all(Indices[2]==Indices[3]))                  \n"
  "   {                                                                                 \n"
  "       // All four indices are identical, so just look ONCE into the table of        \n"
  "       // representatives, and use the result for all four samples.                  \n"
  "        // (This is the same situation as with no multi−sampling at all.) Done.   \n"
  "        float  Result=0.0;                                                        \n"
  "        float4 Index=Indices[0];                                                  \n"
  "                                                                                  \n"
  "        // Essentially same code as below.                                        \n"
  "        Index.r=Index.r∗255.0/256.0;                // Correct the scale.         \n"
  "        Index.g=Index.g∗255.0∗%lu.0/TableWidth; // Correct the scale.             \n"
  "        const float PixelOffsetX=1.0/TableWidth;                                  \n",
  NrOfPixels>0 ? NrOfPixels : 1, SHLMapT::SIZE_S, NrOfPixels);

for (PixelNr=0; PixelNr<NrOfPixels; PixelNr++)
{
    sprintf(DynamicCode+strlen(DynamicCode),
            "Result+=dot(4.0∗(tex2D(SHLCoeffTableSampler, Index.gr)−0.5),"
            " LightsourceSHLCoeffs[%2lu]);",
            PixelNr);
    if (PixelNr+1<NrOfPixels) strcat(DynamicCode, " Index.g+=PixelOffsetX;");
    strcat(DynamicCode, "\n");
}

sprintf(DynamicCode+strlen(DynamicCode),
  "        Total=Result;                                                       \n"
  "    }                                                                       \n"
  "    else                                                                    \n"
  "    {                                                                       \n"
  "        for (int SampleNr=0; SampleNr<4; SampleNr++)                        \n"
  "        {                                                                   \n"
  "            float  Result=0.0;                                              \n"
  "            float4 Index=Indices[SampleNr];                                 \n"
  "                                                                            \n"
  "            // Still unrolling the for−loop, but this time exactly as needed! \n"
  "            Index.r=Index.r∗255.0/256.0;                // Correct the scale. \n"
  "            Index.g=Index.g∗255.0∗%lu.0/TableWidth; // Correct the scale.     \n"
  "            const float PixelOffsetX=1.0/TableWidth;                        \n",
  NrOfPixels);

for (PixelNr=0; PixelNr<NrOfPixels; PixelNr++)
{
    sprintf(DynamicCode+strlen(DynamicCode),
            "Result+=dot(4.0∗(tex2D(SHLCoeffTableSampler, Index.gr)−0.5),"
            " LightsourceSHLCoeffs[%2lu]);",
            PixelNr);
    if (PixelNr+1<NrOfPixels) strcat(DynamicCode, " Index.g+=PixelOffsetX;");
    strcat(DynamicCode, "\n");
```

```
   }
83
   strcat (DynamicCode,
85   "            Total+=0.25*Result;                  \n"
     "         }                                       \n"
     "    }                                            \n"
     "                                                 \n"
     "    Total=clamp(Total, 0.0, 1.0);                \n"
90   "                                                 \n"
     "// OutColor=float4(Total.xxx, 1.0);   // For debugging.  \n"
     "    OutColor=DiffuseC*InColor*float4(Total.xxx, 1.0);    \n"
     " }                                               \n");

95 FragmentShader30_SHL_Compressed=UploadCgProgram(CG_PROFILE_FP30,
                                                   DynamicCode);
```

Listing 6: C++ code for generating Cg code with multi-sampling on demand.

The above listing enhances listing 5 by adding quadruple multi-sampling. The multi-sampling implementation also contains an optimization that is based on the idea that some of the four dependent texture lookups are redundant and can be saved in the case that two or more indices are identical (and thus refer to the same values in the lookup texture). In order to keep the fragment program simple, I decided to only cover the case that all four indices are identical (see lines 33 and 34 in listing 6), which reduces the multi-sampling problem to the same single-sampling problem that was already addressed in listing 5.

Unfortunately, lines 33 and 34 in listing 6 trigger a bug in the Cg compiler (see section 6.7 for full details). This implied that I had to disable the optimized section of code (lines 33 to 58) in listing 6, and gather all observations and results from the generic general-case code only.

6.5 The Ca3DE Material System

The theory that has been presented in the previous chapters leads, when implemented trivially, to a huge amount of source code that suffers from a lot of practical issues, mostly with regard to software design. In my initial implementations, I found that the complexity that comes with fragment- and vertex-shaders on programmable GPUs tends to explode easily: It might well happen that you find yourself writing essentially the same code over and over again, duplicating it for all variants of GPUs, for variants of windowing APIs, for various types of models, for various platforms, for new effects, and so on. At the same time, code and resource management grows to levels of complexity that are almost impossible to handle.

Therefore, I also created a new *Material System* for the Ca3D-Engine within the course of this thesis (partly as a by-product) in order to face all these problems. The goals of the Material System are as follows:

1. Provide a thorough and clean software design, and localize all rendering know-how. The user code becomes only responsible for the “What is rendered?”, the MatSys is responsible for the “How is it rendered?”. Thus, the rendering code becomes *fully separated* from the user code, which doesn’t even have to `#include "GL/gl.h"` etc. any more.

2. Make the interface independent from the underlying platforms and APIs. Note that this goal is *entirely* independent from goal 1.

3. Facilitate providing implementations and support for each platform and API, including such that did not exist when the Material System was designed. For example, separate interface implementations for OpenGL 1.2 and 2.0, DirectX 7 to 9, software rendering, consoles, etc. are desirable.

4. Make it equally easy to add new rendering effects that were not planned when the MatSys was designed.

5. Provide proper and thorough resource management, including shared resource handling like common texture images etc.

6. Make surface properties “scriptable”, that is, easily and externally controllable via human-readable text files.

The Ca3DE Material System achieves all these goals. Its details are presented in great depth in [Fuc04b].

6.6 Porting Ca3DE to the Linux platform

One of the key requirements for implementing the contents of this thesis within the framework of the Ca3D-Engine was that all related components compiled and worked on the Linux operating system. While at the start of this thesis Ca3DE neither featured hardware-accelerated lighting nor lighting with Spherical Harmonics, it was a pure Microsoft Windows based application suite until then. Although many of its components were written with cross-platform portability in mind right from the start, the port to Linux proved to be a challenging but very interesting and highly educational task. A very useful resource in this regard is [LSH01], which is a good introduction to Linux programming especially if you have previous knowledge about Windows programming.

The following is a list of key problems that I encountered during the porting, where the specifics of Windows programming are opposed to the specifics of Linux programming. While this section is not strictly a contribution to the contents of this thesis, it might be helpful for anyone who faces a similar porting task.

- The compiler that I used on Linux was `g++`, while on Windows, the free OpenWatcom compiler (`http://www.OpenWatcom.org`) was employed. Some of the differences below are due to the differences in the compilers and their standard libraries, others are due to the differences in the underlying operating systems.

- Case independent comparison of character pointers (strings): On Windows, you can either use `stricmp()` or the equivalent `strcmpi()`, whereas on Linux, `strcasecmp()` is defined. I eventually encapsulated them all in a small auxiliary library that acts as a compatibility layer, but the best long-term solution is probably to directly employ the `std::string` class from the C++ standard library (which is unfortunately not yet available for OpenWatcom), or the similar `wxString` class from the wxWidgets project (http://www.wxWidgets.org), which is available for both compilers and solves the problem perfectly.

- Structure packing and member alignment: Sometimes, the contents of a `struct` are directly written from memory to disk and read back. This only works if all compilers align the members of the `struct` in the same fashion. As the compilers normally align the contents of structures onto 2, 4 or 8 byte boundaries for optimization by adding the appropriate number of padding bytes between individual members of the `struct`, the best way to resolve this is to explicitly force structure packing, which yields tight alignment. Structure packing is enforced by decorating the `struct` declaration with the `_Packed` keyword for the OpenWatcom compiler and the `__attribute__ ((packed))` expression for the `g++` compiler.

- Opening a window with support for OpenGL rendering is conceptually similar on both systems, but differs much in the underlying windowing APIs. The differences are best seen in the respective source code, which contains detailed comments.

- Mouse and keyboard input. This has been the toughest problem I faced during the entire porting, because I used the DirectInput component of Microsoft DirectX for this purpose on the Windows platform. There is no equivalent for DirectX on Linux. Therefore, I use the regular X window message queue for mouse and keyboard input under Linux, and continue to use DirectInput on Windows. Offering a common interface to other source code for both approaches required a significant redesign and rewrite of the previous user input handling code.

- Many other parts like the OpenGL rendering, networking with Berkley network sockets, sound support etc. were all written with portability in mind right from the start, and thus provided a much easier porting experience than the mouse and keyboard input. The only slight problems were with the networking APIs, where error constants are prefixed with `WSAE` on Windows and simply `E` on Linux, and the specification of the `select()` function differs on both systems: The first parameter of `select()` is ignored on Windows, and has to be the highest socket number +1 on Linux, the violation of which silently introduces a problem which is very hard to diagnose.

- The scope of variables that are defined in `for`-loops ends at the closing brace in `g++` but used to extend beyond with OpenWatcom. While OpenWatcom modified this in its latest release, at that time I was forced to replace many cases of `for (int i=0; ...)` with `int i; for (i=0; ...)`.

- The default size of `enums` differs, and can even change with different compiler options such as for space optimization. This in turn causes dangerous problems when `enums` are read from and written to files on disk that are used on both platforms, and has to be worked around explicitly, e.g. by casting them to types of known and constant size.

- OpenWatcom provides a compiler specific `_splitpath()` function that splits a full file name into the path, the base name, and the extension. The `g++` libs do not seem to provide anything comparable, and thus I had to implement a similar function on Linux myself.

- The function names for functions that take variable argument lists differ. E.g. there is `_vbprintf()` on Windows, and `vsnprintf()` on Linux. Once more, the best remedy is probably to switch to the appropriate C++ string classes directly.

- While the Win32 API provides a function `SystemTime()`, there is nothing directly equivalent on Linux. However, both platforms have the `time()` and `str???time()` functions, which can also replace `SystemTime()` on Windows.

- There are no equivalents for `getch()` and `kbhit()` on Linux, which are sometimes quite useful despite the fact that they are possibly not multitasking friendly. Linux has `getchar()`, which however does not behave like `getch()` on Windows. Instead of employing some 'dirty hacks' in order to get the desired behaviour on Linux anyway, I eventually worked around the problem and now simply omit these function on Linux. The best solution in this case is probably to turn the affected programs into full windowing apps entirely, preferably using wxWidgets as the underlying windowing system.

- Using dynamic link libraries comes with many inherent subtleties on both systems anyway, and even more subtle are the differences between Windows and Linux dynamic link libraries. It took me a very long time to figure out the related techniques for those alone. Please refer to the code and makefiles for full information.

- Both compilers have some "lint" capabilities, where most of which are common to both compilers, but each also has nice features that the other doesn't have. `g++` detects using the use of `"%u"` within a `printf()` format string when the respective argument is a *long* unsigned integer, and `"%lu"` is actually required. It also warns about comparisons between signed and unsigned variables, and the use of variables of type `char` as array indices. Another beneficial feature of `g++` is that it enforces proper function exception specifications both at compile and run time.

6.7 Implementation Issues

The implementation of dynamic lighting that is treated in section 4.5 has not been as smooth as one might expect. Besides all the problems and bugs that I take on myself (all fixed in the final release), there were also several issues that were either inherent

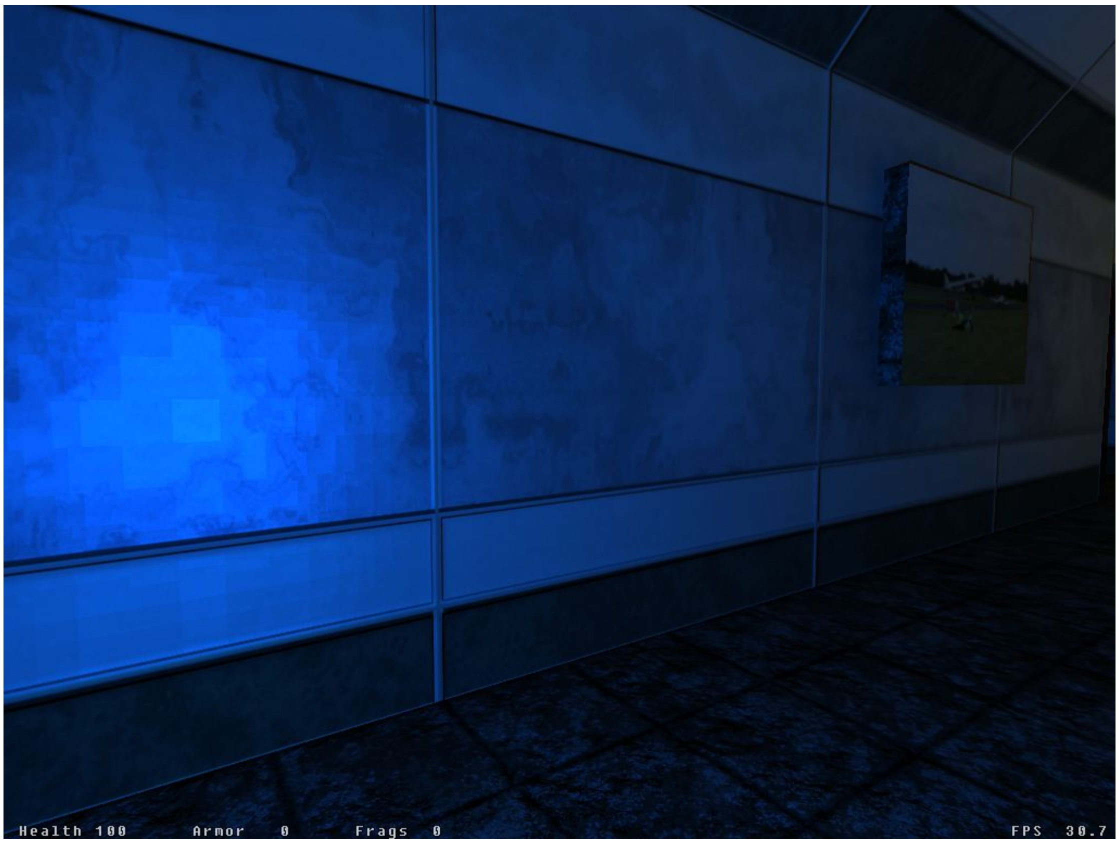


Figure 29: A blocky specular highlight on an NVidia NV2X GPU.

in the underlying technologies, or even bugs in the 3D graphics drivers. Whenever I encountered problems with the 3D graphics drivers, I contacted the respective manufacturer (NVidia or ATI) and collaborated with them to get the problems fixed. In all cases, I either got an acknowledgement that the issue was actually a driver bug or other software problem with the manufacturer, or was able to reproduce the problem even with the manufacturers very own (demo) software.

Below I present a list of the most subtle issues that are either inherent to the technology and thus cannot really be fixed at all, or are driver-related and whose fix was only available after the writing of this document. None of these issues is really a death sentence, but several of them took me days and weeks to diagnose and fix.

Blocky Specular Highlights on early generation GPUs

Smooth and planar surfaces with high specular reflectivity exhibit specular highlights that look somewhat gross and “blocky”. An example of this behaviour is the light blue specular highlight that can be seen in figure 29. The cause for the blocky appearance is the fact that on GPUs that do not directly support exponentiation (as for example by a `pow()` function), the rise of the specular term $(N \odot H)^n$ to the n-th power has to be computed by other means.

One such means is to encode the `pow()` function for a fixed exponent in a one-dimensional texture, and to compute the exponentiation by a simple texture lookup. Especially ATI presents a clever trick with its `ATI_fragment_shader` and `EXT_vertex_shader` OpenGL extensions at `http://www.ati.com/developer/R8500PointlightShader.html` for Radeon 8500+ GPUs that employs "NHHH" lookup textures for solving the problem even more elegantly. Unfortunately, though, none of these lookup techniques can be brought forward to NVidia GPUs of the same generation (NV2X), as these GPUs are further restricted in their texture lookups. Moreover, as this generation of GPUs is rendered obsolete by more sophisticated GPUs that provide a `pow()` function directly, I did not take the effort to actually implement the lookup technique for the ATI rendering path in Ca3DE.

Another means to compute exponentiation that works both on NV2X and Radeon 8500+ GPUs is to simulate it by repeated multiplication. Here is the relevant section from the appropriate Cg fragment shader:

```
const float3 HalfwayDir=2.0*(texCUBE(NormalizeCubeForHalfwayVector,
                                     InHalfwayVector).xyz-0.5);
const float3 Normal    =2.0*(tex2D(NormalMapSampler,
                                   InTexCoord_norm).xyz-0.5);

float spec=saturate(dot(Normal, HalfwayDir));

spec=spec*spec; // Simulate   spec=pow(spec, 32.0);
spec=spec*spec;
spec=spec*spec;
spec=spec*spec;
spec=spec*spec;
```

Listing 7: Exponentiation by repeated multiplication.

While this practise is highly questionable from a performance point-of-view, as the previously mentioned lookup methods may be a lot faster than the repeated register combiner operations here, the repeated multiplication in combination with the limited bit depth of the GPU registers is the cause for the blocky appearance of the specular highlights.

I've made considerable efforts to overcome this problem, especially on NV2X GPUs. However, as these GPUs do not even provide a chance to dodge to the previously mentioned means of simulating the exponentiation by a 1D texture lookup, it seems that the problem cannot be properly be solved at all (for NV2X GPUs), or at least I've not become aware of any such solution during my entire work on the thesis.

Torn and Distorted Specular Highlights

Another artifact that occurs with specular highlights and the presence of normalization cube-maps (mostly used on early generation GPUs) is the torsion and distortion of the shape of the specular highlights. Figure 30 shows three examples of this problem with both a single polygon that shows a mildly distorted highlight, as well as an extreme case where a highlight that is cast across two adjacent polygons is highly distorted. (The subfigures also happen to repeat the blockiness of the highlights as discussed above.) In the figure, the left series of images shows the problem before the fix, the right half showing the improved highlights after the fix.

The cause for the distortions in figure 30 was actually a proper and correct implementation: In order to compute the sub-term $(\vec{N} \odot \vec{H})^n$ of the specular contribution of the Phong lighting equation, I computed the halfway vector $\vec{H}$ in the vertex program for NV2X GPUs as follows:

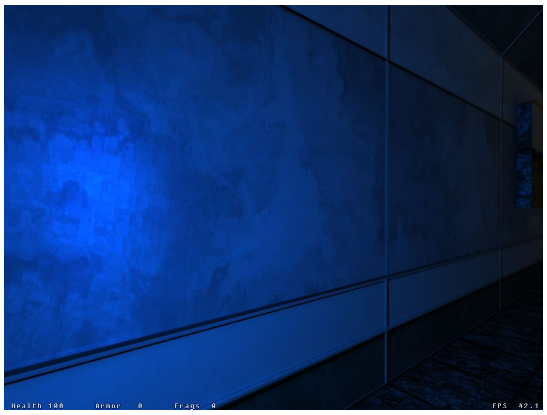
```
const float3 EyeDir  =normalize(EyePos −InPos.xyz);
const float3 LightDir=normalize(LightPos−InPos.xyz);

OutHalfwayVector=mul(RotMat, EyeDir+LightDir);
```

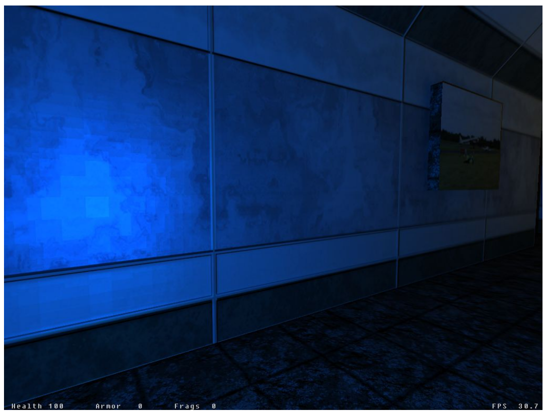
Listing 8: Halfway vector computation in the vertex program.

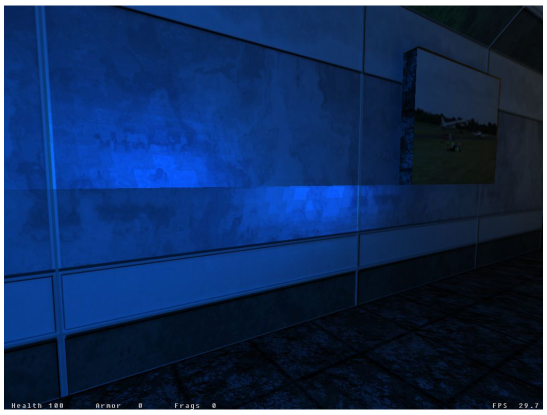
This section of Cg vertex program computes first the vector from the current vertex position to the eye, then the vector from the vertex position to the light source, and finally the unnormalized halfway vector by adding the normalized eye vector and the normalized light vector. The final multiplication with `RotMat` is just for rotating the halfway vector into tangent space, and not relevant for the current discussion. The normalization of the just computed halfway vector (and the rest of computing $(\vec{N} \odot \vec{H})^n$) is deferred to the appropriate Cg fragment shader.

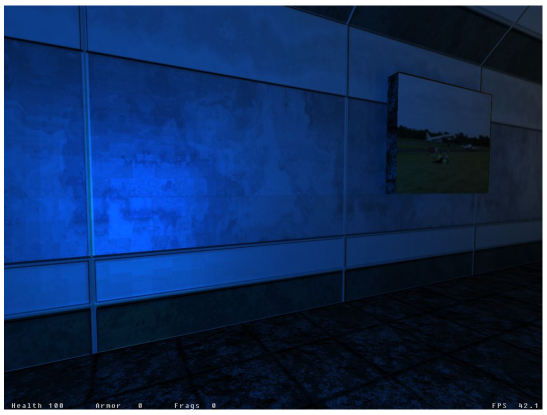
Here comes the crucial point: The stage of the rendering pipeline that is between the GPU vertex and fragment programs assembles and rasterizes the polygon and interpolates the previously computed halfway vector across the polygon. The interpolated values are then accessible in the fragment shader. With the unnormalized halfway vectors given as input to the interpolation, their lengths are according to the above excerpt of vertex program between 0.0 and 2.0, and thus generally extremely short even before the interpolation begins. Especially if the light vector and eye vector initially point in roughly opposite directions, the length of the halfway vector easily gets near zero. The (linear) interpolation operation across the polygon then tends to yield even shorter interpolated vectors. As a result, the lookup into the normalization cube-map in the fragment program for normalizing the halfway vector is performed with a near-zero length input. Degenerate input to cube-map lookups may however return undefined results. It is the undefined results that cause the observable artifacts that constitute the distorted specular highlights.

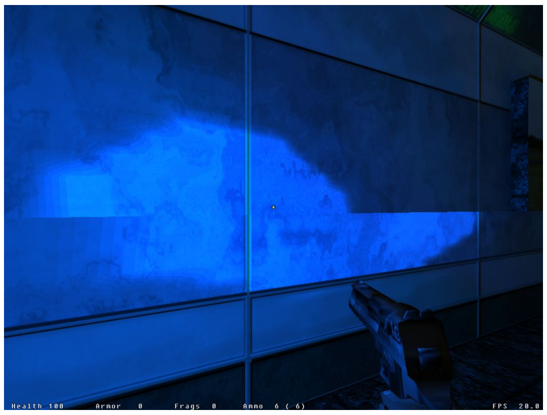
Fixing the matter in a straightforward and robust manner is simple: Instead of computing $\vec{H}_{old} = \frac{\vec{L}}{|\vec{L}|} + \frac{\vec{E}}{|\vec{E}|}$ as above, multiply everything with $|\vec{L}|$. This works because $\vec{H}$


(a)


(b)


(c)


(d)


(e)

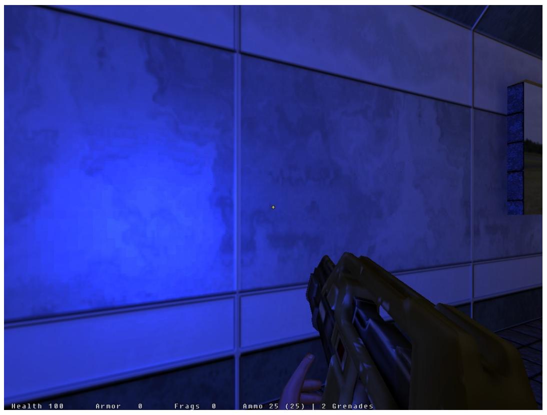

(f)

Figure 30: Torn and distorted specular highlights. The images on the left show the original appearance of the artifacts, the images on the right show the appropriate situation after the lookup into the normalization cube-map was fixed.

is initially (in the vertex program) computed as an unnormalized vector anyway, and the normalization is deferred until after interpolation to the cube-map lookup, where a much longer vector is desired. Thus, compute $\vec{H}_{new} = \vec{L} + \frac{\vec{E}}{|\vec{E}|}|\vec{L}|$. The new vertex program becomes

```
const float3 LightVector     =LightPos−InPos.xyz;
const float  LightVecLen     =length(LightVector);
const float3 EyeVectorScaled=normalize(EyePos−InPos.xyz)∗LightVecLen;

OutHalfwayVector=mul(RotMat, EyeVectorScaled+LightVector);
```

Listing 9: Improved halfway vector computation.

This fixes the problem.

Note that this problem only occurs when the normalization of $\vec{H}$ is performed by a normalization cube-map lookup. When the normalization is instead performed with e.g. the Cg `normalize()` function (that is e.g. supported on NV3X and newer GPUs), no suffering from too short input vectors is experienced, and thus the problem is avoided right at its source.

Moiré patterns and wrong pixels on ATI Radeon GPUs

An entire series of strange problems occurs on ATI Radeon graphics boards on which I employ the `ATI_fragment_shader` and `EXT_vertex_shader` OpenGL extensions for rendering.

Although hardly noticeable, there were still wrong pixels observable on ATI Radeon 9200 boards. Figure 31(a) shows such wrong pixels even in the midst of otherwise perfectly valid polygons. For reference, 31(b) shows the underlying polygonal tessellation of the same scene.

Another problem observed on the Radeon 9200 series is shown in figure 31(c): The flicker of entire polygons. The occurrence of this effect is very rare. Sometimes it occurs every few seconds, sometimes only every few minutes, but most often not at all. It took me a good deal of patience and several dozen screenshots in order to catch figure 31(c). Figure 31(d) shows the correct image for reference.

A third problem that was never observed on the Radeon 9200 or any other series based on the same GPU, but on basically all newer boards like the 9700 and 9800 series is demonstrated in figure 31(e) and 31(f). This is the most severe and noticeable of all.

I invested considerable effort to make sure that none of these problems was indeed my own fault, e.g. by bad programming or by mis-reading the extension specs. However, especially the third, most significant of the above problems can be reproduced with one of ATI's very own demo programs. Contrary to my own implementation, which is based on OpenGL, their program is based on DirectX. This strongly indicates that the problem is indeed with the driver rather than the application.

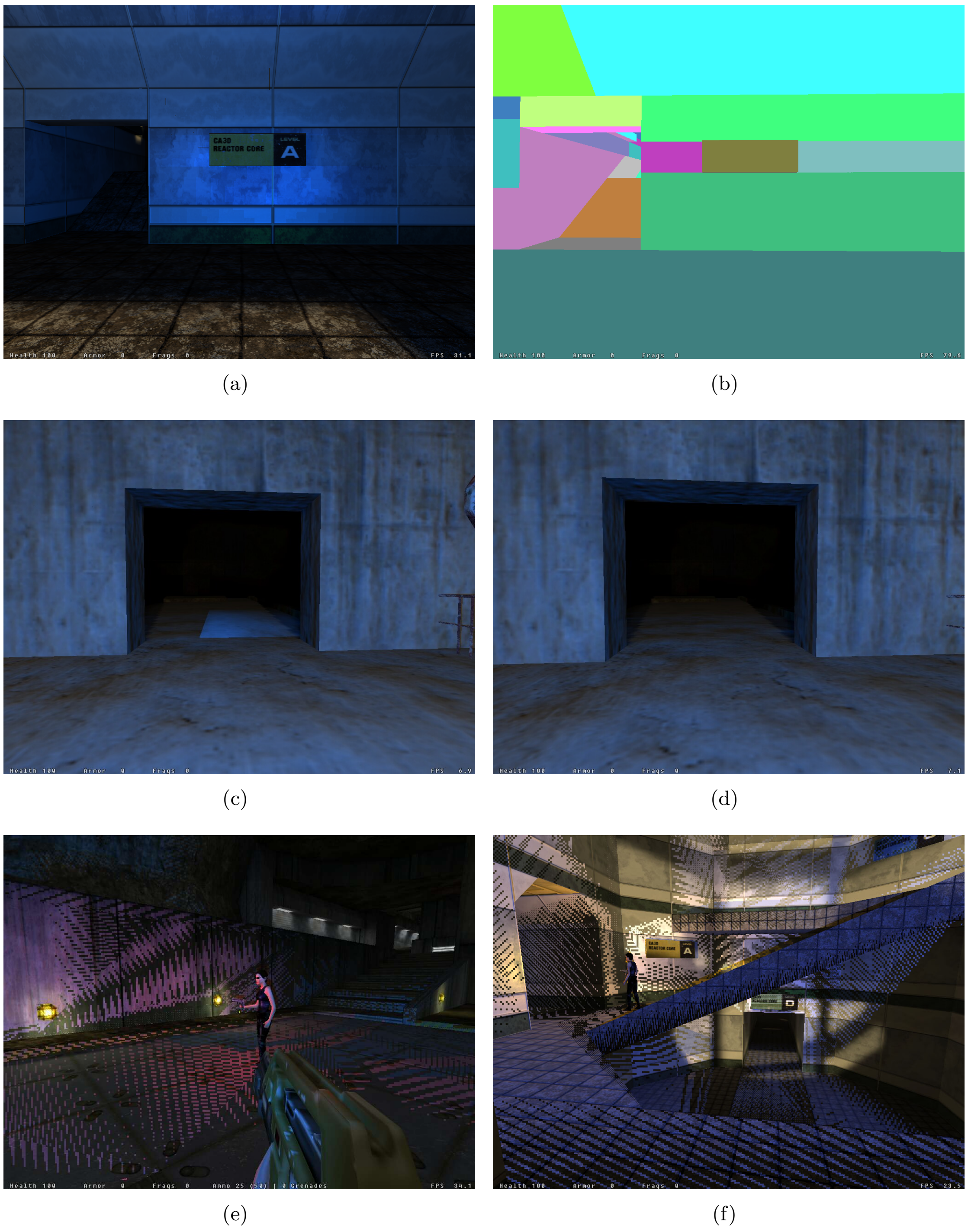


Figure 31: Several display problems on ATI Radeon graphics boards.

Issues with the Cg compilers

The final Cg fragment program for Spherical Harmonic Lighting that has been discussed in section 5.8 features decompression of SHL values, a variable number of SHL coefficients by procedurally generated code, and quadruple subsampling. Unfortunately, this most sophisticated of all GPU programs in my thesis caused the NVidia Cg compiler v1.1 to behave erroneously. Both the dynamic run-time as well as the offline command-line compilers were affected. Symptoms included the reporting of internal compiler errors or perfectly innocent statements to be syntactically or semantically wrong, and thus prematurely abort the compilation. Once more, I invested a lot of time in finding a work-around for this problem, to no avail.

More precisely, lines 33 and 34 in listing 6

```
33    if ( all(Indices[0]==Indices[1]) &&
           all(Indices[1]==Indices[2]) && all(Indices[2]==Indices[3]))    // ...
```

yield the compiler error "`Internal error C9999 (92): Expected a scalar constant but found something else.`". In fact, replacing this with

```
33    if (false)      // ...
```

gets proper acceptance, whereas the simple change to

```
33    const bool myBool=false;
      if (myBool)     // ...
```

exhibits the previous problem.

Moreover, upgrading from Cg SDK version 1.1 to version 1.2, which I hoped would solve the problem, proved to be unfeasible: Despite the documentation claims to the contrary, version 1.2 seems to be incompatible and non-backwards compatible to version 1.1. I did not manage to successfully upgrade to version 1.2 even after hours and days of effort. Error messages included more internal errors like "`Unknown builtin_vmath.`" even for the smallest (and correct) programs.

However, the Cg *command-line compiler in version 1.2* managed to compile the source code properly for which version 1.1 had failed. I therefore decided to pre-compile the source code to GPU assembly in this way, but stay with version 1.1 of the Cg run-time otherwise. Of course, pre-compiling in turn defeated the benefits of the procedurally created code (i.e. the ability to deal with an arbitrary number of SH coefficients), as a constant for the number of coefficients has to be preselected. However, I decided to create a special-case treatment for this case rather than let this chain of bugs stop my research.

Consequently, I pre-compiled the most common case of 4 SHL bands (16 coefficients) offline into assembly, and changed the main C++ code to choose the pre-compiled but correct assembly whenever 4 SHL bands are used, and to fall back to the actually intended (but wrongly compiled, and thus actually broken) program for any other number of SHL bands. This in turn required me to restrict all my tests to 4 SHL bands only, but that turned out to be not much of a restriction in practise, and of course did not affect the results and conclusions of the thesis at all.

The pre-compiled assembly comprised 225 instructions. Here is an excerpt from the first few lines:

```
DECLARE LightsourceSHLCoeffs$0;
DECLARE LightsourceSHLCoeffs$1;
DECLARE LightsourceSHLCoeffs$2;
DECLARE LightsourceSHLCoeffs$3;
DECLARE NrOfColumns;
DECLARE TableWidth;
ADDR R0.xy, f[TEX1].xyxx, {-0.0009765625, 0.0009765625}.xyxx;
TEX R0, R0.xyxx, TEX1, 2D;
MULR R1.x, R0.x, {255}.x;
MULR R1.x, R1.x, {0.00390625}.x;
MULR R1.y, R0.x, {255}.x;
MULR R1.y, R1.y, {0.00390625}.x;
MOVR R2.yzw, R0.yyzw;
MOVR R2.x, R1.x;
ADDR R3.xy, f[TEX1].xyxx, {0.0009765625, 0.0009765625}.xyxx;
TEX R3, R3.xyxx, TEX1, 2D;
SEQR H0, R0, R3;
MULX H1.x, H0.x, H0.y;
MULX H1.x, H1.x, H0.z;
MULX H0.x, H1.x, H0.w;
MOVR R0.x, R1.y;
MULR R1.x, R3.x, {255}.x;
MULR R1.x, R1.x, {0.00390625}.x;
 ...
```

Listing 10: The beginning of the precompiled fragment shader code.

In summary, a special-case work-around was created to face two independent issues with the NVidia Cg suites: Version 1.1 was not able to compile this thesis's most sophisticated fragment program, while an upgrade to version 1.2 was unfeasible for reasons I could not eventually determine. This is true for all supported platforms (Windows and Linux).

My intention is to provide a complete new renderer that is based on the OpenGL Shading Language (GLSL) in the future. This renderer will replace the Cg based renderer, and I have good faith that it will avoid the aforementioned problems right from the start.

7 Conclusions

This thesis presented three different methods for real-time lighting in environments that are a composition of static and dynamic (animated) elements.

It turned out that dynamic lighting on dedicated hardware is well founded on the Phong shading and illumination model (section 4). It can be augmented with stencil shadow volumes. Section 4.3.1 showed a method for quick shadow volume determination with very large BSP models that represent the hull of entire worlds. The strengths of this technique are that it is able to treat all components in a unified fashion and no inherent preprocessing is required. That means that both animated as well as static components are lit uniquely, and all cast and receive shadows to and from all others. The weakness is that the physical plausibility is marginal, which can often be observed in the final images.

While lighting with light-maps (section 3) is straightforward in principle, the light-maps may be created by means of radiosity algorithms. This results in very realistic images. Unfortunately, the method works best with fully static scenes, where neither light sources nor geometry change in any of their attributes (e.g. light intensity and color, position of geometric objects, etc.). Dynamic objects like player models that are put into the scenes after preprocessing can neither participate in lighting nor cast any shadows. The tricks and hacks that are required to obtain (fake) lighting and shadows for animated objects are, in any case, outside the scope of this thesis.

For Spherical Harmonics Lighting (section 5), a relatively new field of investigation, it turned out during my research that shooting-based radiosity solutions (CaLight) and precomputing SH coefficients with bounce transfer (CaSHL) can be considered as partially comparable problems and it turns out that algorithms can be achieved that have analogous implementations. The main difference between radiosity and SH computations is that patches are associated with light *energy* in the radiosity case, whereas they are associated with light *transfer functions*, represented by spherical harmonics coefficients, in the SH case. The development and execution of this analogy is one of the main contributions of this thesis. To the best of my knowledge, no other implementation of SH lighting that is "patch based" and employs this analogy exists. Similar to the light-maps method, SHL requires expensive preprocessing of the scene. It does not inherently work with local light sources, but light sources in the far range (like the celestial bodies) can be arbitrarily dynamic, that is, change in color, position and shape. If keyframe interpolation is an option, the method can even be applied to animated objects. Each such object would be properly lit and cast shadows onto itself, but it would not cast nor receive shadows from other objects.

A Acknowledgements

This thesis would not have been possible without the help of many people. In particular, I would like to thank my supervisors, Dr. Jan Kautz and Prof. Dr. Hans-Peter Seidel, who gave me the freedom to implement the thesis contents in the framework of the Ca3D-Engine.

I'm also very grateful for Jan's constant encouragement and enthusiasm for the development of this thesis, and for his patience in discussing all problems, even across the Atlantic.

Special thanks go to Kai Schadwinkel, who has created the artwork (normal-maps, specular-maps, etc.) that is required to visualize and operate the presented theory and implementation of dynamic lighting. Therefore, in many senses, Kai provided the fuel for my engine. He also introduced me to the artistic viewpoints and approaches of mesh modelling and dynamic lighting.

I also thank Philip Whiston for proof-reading the entire thesis. Philip provided me with interesting insights into the differences between American and British English, and many other fine points of written English.

B Erklärung

Ich versichere, daß ich diese Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und wurde von dieser als Teil einer Prüfungsleistung angenommen.

Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, daß die Arbeit veröffentlicht wird und daß in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Saarbrücken, February 18, 2005

Carsten Fuchs

References

[AAM03] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics*, 22(3):511–520, July 2003. 2

[Abr96] Michael Abrash. *Zen Of Graphics Programming*. The Coriolis Group, Scottsdale, AZ, USA, second edition, 1996. 2

[ADMAM03] Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. An Optimized Soft Shadow Volume Algorithm with Real-Time Performance. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 33–40. Eurographics Association, 2003. `http://www.eg.org/EG/DL/WS/EGGH03/033-040-assarsson.pdf`. 2

[AKDS04] Thomas Annen, Jan Kautz, Frédo Durand, and Hans-Peter Seidel. Spherical harmonic gradients for mid-range illumination. In Henrik Wann Jensen and Alexander Keller, editors, *Rendering Techniques 2004 Eurographics Symposium on Rendering*, pages 331–336, Norrköping, Sweeden, June 2004. Eurographics Association, Eurographics Association. 2

[Ash94] Ian Ashdown. *Radiosity: A Programmer's Perspective*. John Wiley & Sons, New York, NY, 1994. 3.1

[Bra] Stefan Brabec. *Shadow Techniques for Interactive and Real-Time Applications*. PhD thesis. 2, 4.3

[BSa] Stefan Brabec and Hans-Peter Seidel. Hardware-accelerated Rendering of Antialiased Shadows with Shadow Maps. pages 209–214. 2

[BSb] Stefan Brabec and Hans-Peter Seidel. Shadow Volumes on Programmable Graphics Hardware. 2

[CW93] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Boston, MA, 1993. 2, 3.1, 3.1, 3.3.4, 3.3.4, 5.2.2

[Dek94] Anthony Dekker. Kohonen Neural Networks for Optimal Colour Quantization. *Network: Computation in Neural Systems*, 5:351–367, 1994. `http://members.ozemail.com.au/~dekker/NEUQUANT.HTML`. 5.6.1

[Die03] D. Sim Dietrich. Texture Space on Real Models. Technical report, NVIDIA Corporation, 2003. 2

[EK02] Cass Everitt and Mark J. Kilgard. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. Technical report, NVIDIA Corporation, April 2002. `http://developer.nvidia.com/docs/IO/2585/ATT/RobustShadowVolumes.pdf`. 2, 4.3

[EK03] Cass Everitt and Mark J. Kilgard. Optimized Stencil Shadow Volumes. In *Proceedings of the Game Developer Conference*, San Jose, March 2003. `http://developer.nvidia.com/docs/IO/4449/SUPP/GDC2003_ShadowVolumes.pdf`. 2, 3b

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley, Boston, MA, USA, 2003. 4.1, 5.7.1, 6.2

[Fuc03] Carsten Fuchs. *Point Based Rendering of Animated Objects in Real-Time.* 55765 Birkenfeld, Germany, 2003. Fortgeschrittenenpraktikum an der Universität des Saarlandes. 4.4

[Fuc04a] Carsten Fuchs. *Ca3D-Engine: Making New Materials.* 55765 Birkenfeld, Germany, 2004. `http://www.Ca3D-Engine.de`. 6.1, 6.1.1

[Fuc04b] Carsten Fuchs. *Ca3D-Engine: The Ca3DE Material System.* 55765 Birkenfeld, Germany, 2004. `http://www.Ca3D-Engine.de`. 6.5

[Fuc04c] Carsten Fuchs. *Ca3D-Engine: User Manual.* 55765 Birkenfeld, Germany, 2004. `http://www.Ca3D-Engine.de`. 6.1

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics. Principles and Practice.* Addison-Wesley, Reading, MA, USA, second edition, 1990. 2, 2, 3.1, 4.1, 5.3, 6.2

[Gre03] Robin Green. Spherical Harmonic Lighting: The Gritty Details. Technical report, Sony Computer Entertainment America Inc., Research and Development, January 2003. `http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.html`. 2, 5.1, 5.1.3, 5.2.2, 5.3, 5.4.3, 5.5.1

[HM01] Evan Hart and Jason L. Mitchell. Hardware Shading with EXT_vertex_shader and ATI_fragment_shader. Technical report, 3D Application Research Group, ATI Research, September 2001. 6.2

[LRP97] Gregory Ward Larson, Holly Rushmeier, and Christine Piatko. A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):291–306, October–December 1997. ISSN 1077-2626. 3.3.5, 5.4.4

[LSH01] Inc. Loki Software and John Hall. *Programming Linux Games: Learn to Write the Games Linux People Play.* Linux Journal Press, San Francisco, CA, USA, 2001. 6.6

[MHE$^+$03] Morgan McGuire, John F. Hughes, Kevin Egan, Mark Kilgard, and Cass Everitt. Fast, Practical and Robust Shadows. Technical report, NVIDIA Corporation, Austin, TX, November 2003. `http://developer.nvidia.com/object/fast_shadow_volumes.html`. 2, 4.3, 4.3.1, 3b

[NRH03] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. All-frequency shadows using non-linear wavelet lighting approximation. *ACM Transactions on Graphics*, 22(3):376–381, July 2003. 2

[Pip98] Nils Pipenbrinck. Octree Color Quantization. 1998. `http://www.cubic.org/~submissive/sourcerer/octree.htm`. 5.6.1

[PSS99] A. J. Preetham, Peter Shirley, and Brian E. Smits. A Practical Analytic Model for Daylight. In Alyn Rockwood, editor, *Proceedings of the Conference on Computer Graphics (Siggraph99)*, pages 91–100, N.Y., August8–13 1999. ACM Press.

[Ros04] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley, Reading, MA, USA, 2004. 6.2

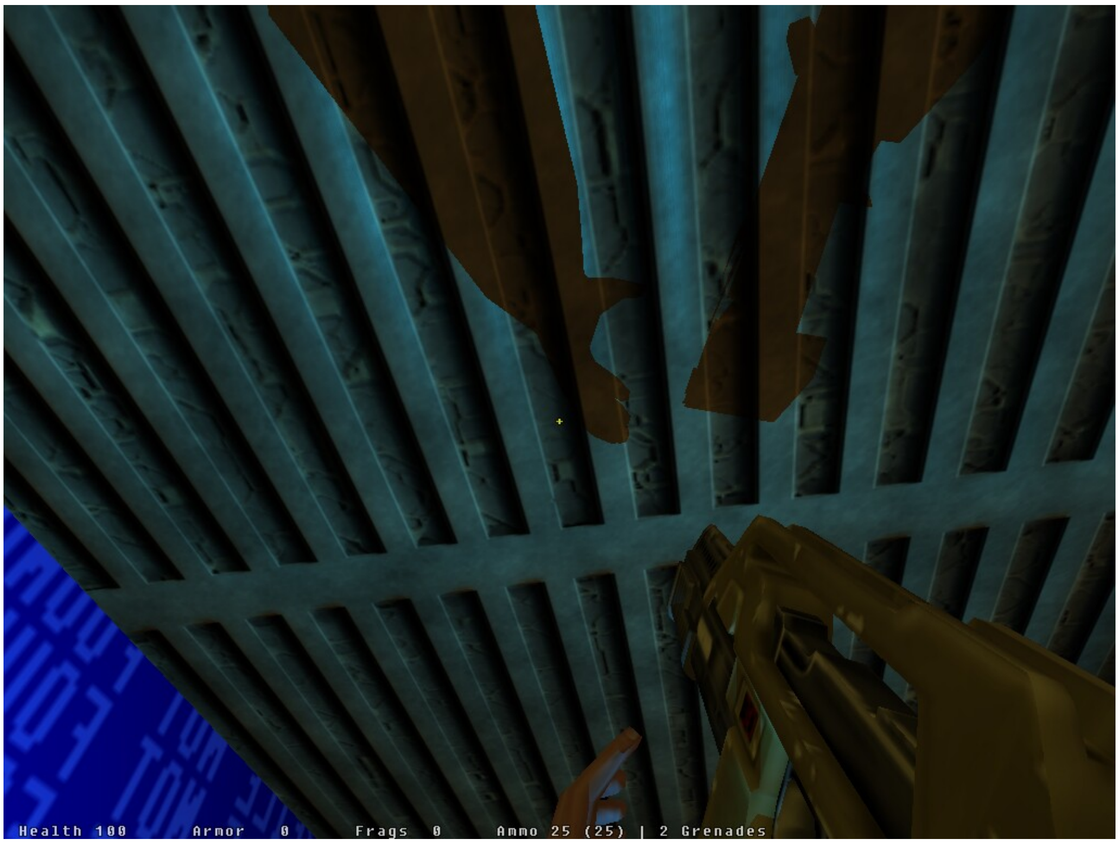
[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. In Stephen Spencer, editor, *Proceedings of the 29th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH-02)*, volume 21, 3 of *ACM Transactions on Graphics*, pages 527–536, New York, July 21–25 2002. ACM Press. 2, 5.1, 5.2.2, 5.3

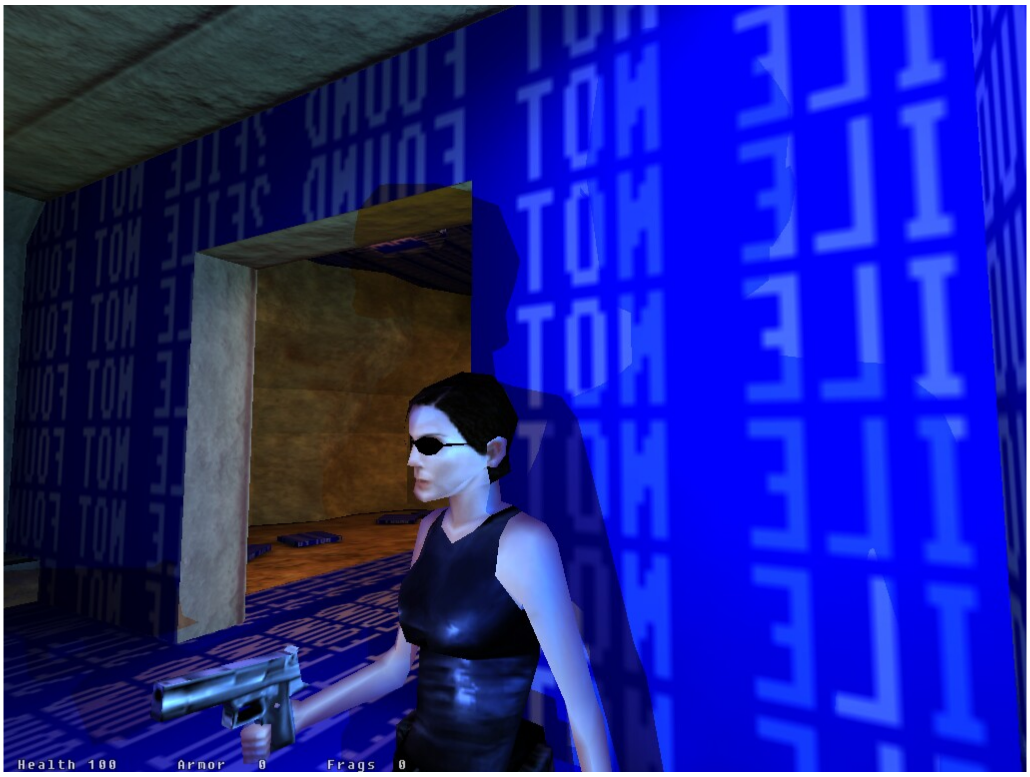
[Tel92] Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, CS Division, UC Berkeley, October 1992. Tech. Report UCB/CSD-92-708. 2, 6.1.1

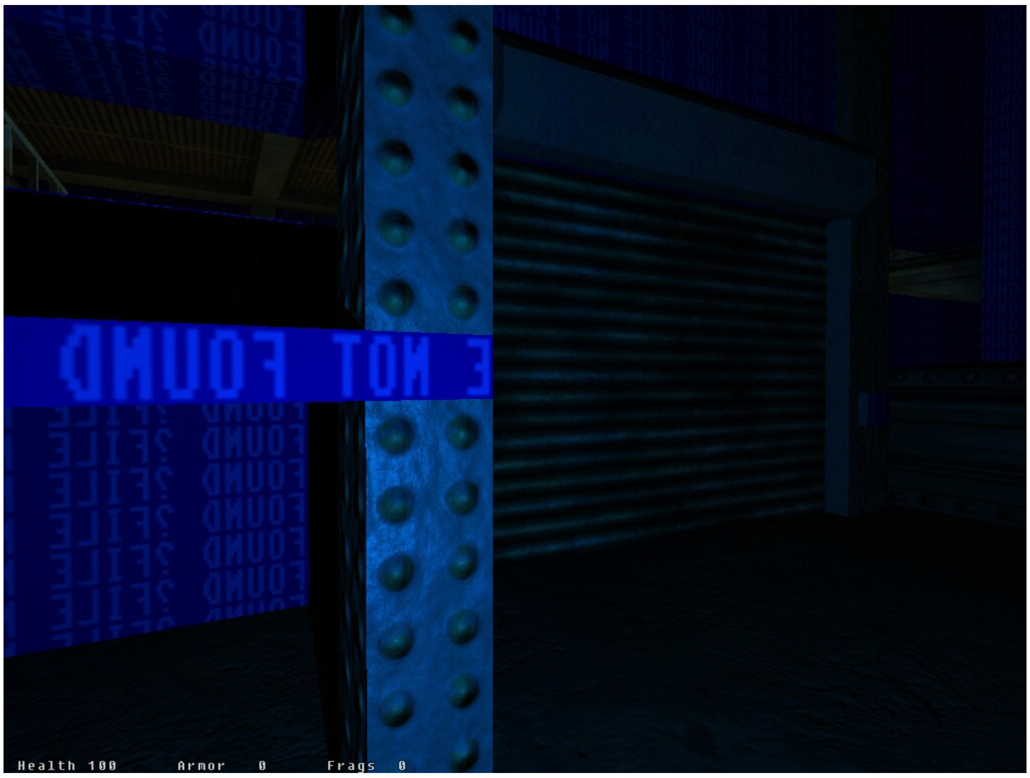
[WNDO99] Mason Woo, Jackie Neider, Tom Davis, and OpenGL Architecture Review Board. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, Reading, MA, USA, third edition, 1999. 6.2

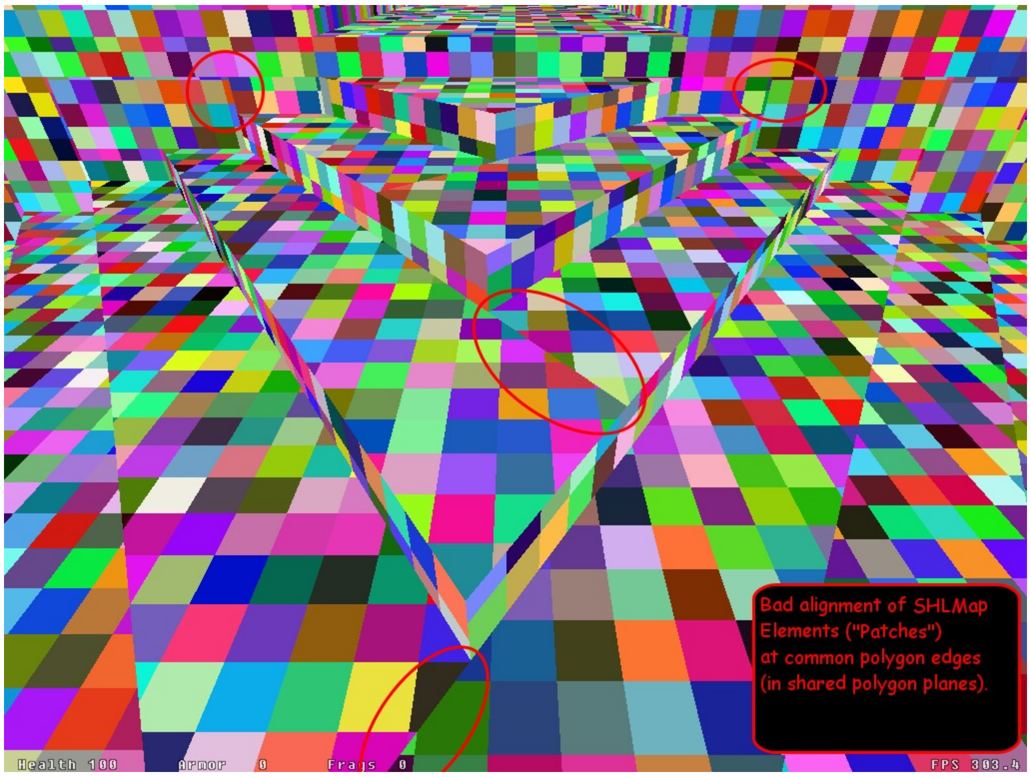
[Wol90] George Wolberg. *Digital Image Warping*. IEEE Computer Society Press, 10662 Los Vaqueros Circle, Los Alamitos, CA, 1990. IEEE Computer Society Press Monograph.

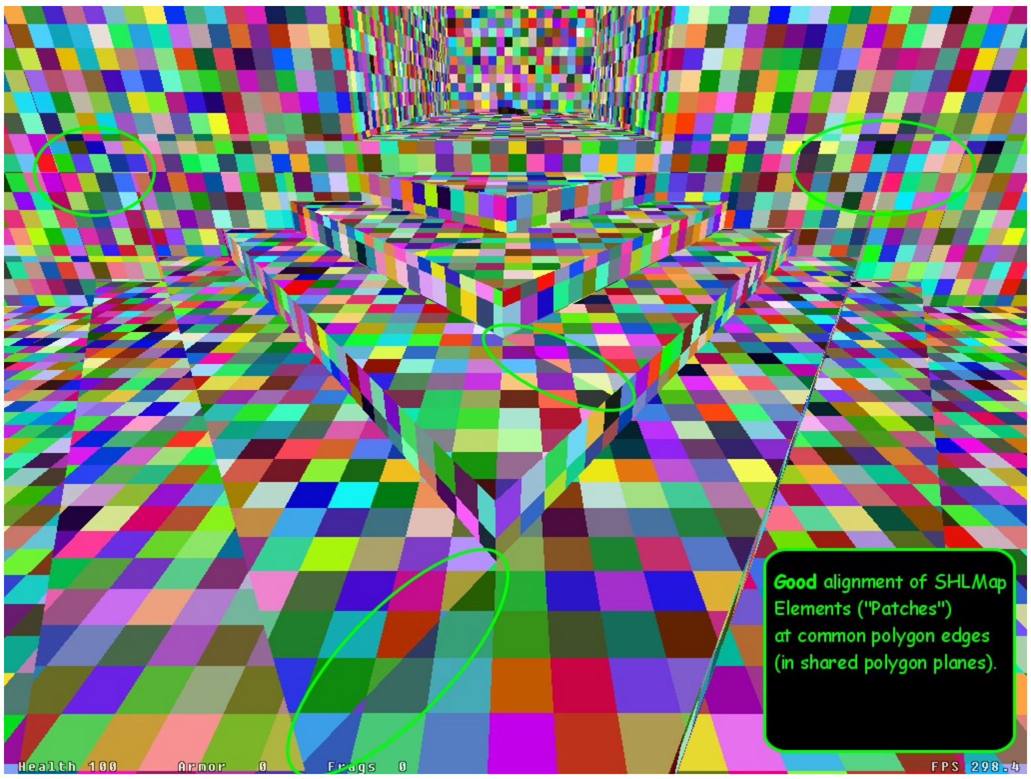
Health 100
Armor 0
Frags 0
Ammo 6 (6)
FPS 14.4


Health 100
Armor 0
Frags 0
Ammo 25 (25) | 2 Grenades


Health 100
Armor 0
Frags 0


Health 100
Armor 0
Frags 0


Bad alignment of SHLMap
Elements ("Patches")
at common polygon edges
(in shared polygon planes).
Health 100
Armor 0
Frags 0
FPS 303.4


Good alignment of SHLMap
Elements ("Patches")
at common polygon edges
(in shared polygon planes).
Health 100
Armor 0
Frags 0
FPS 298.4