Forschungsseminar aus Computergraphik und Digitaler Bildverarbeitung (SS2006)
A. Wilkie, G. Zotti, R. Habel (Editors)

Directional Lightmaps

Peter Houska[†]

Institute of Computer Graphics & Algorithms, TU Vienna, Austria

Abstract

This report explains the basic idea behind directional lightmaps. The explanation is based on a presentation about Valve Software's Source Engine. The engine was used for the game Half-Life 2 and heavily relies on directional lightmaps to produce realistic lighting effects.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three Dimensional Graphics and Realism

1. Introduction

While computer hardware keeps improving, also the costumer, e.g. someone who buys and plays a 3D computer game, expects the graphics to look more realistic. Lighting effects are crucial for providing this perception. It is of great interest for software-developers to come up with solutions that look good and work fast. In the early days of hardware accelerated graphic-cards, it was sufficient to apply a so called **lightmap** together with the "traditional" texture. This concept was first introduced with the game **Quake** by id Software. In the lightmap the static diffuse light interaction in the scene is stored and since the diffuse lighting solution is view-independent, the light-intensities at each position in the scene can be precalculated. At runtime, only a lookup is performed. This means, that the lightmap is just another texture that is attached to each polygon and if multitexturing is supported by the video adapter, then there is no additional cost at runtime, since both textures are rendered in one pass. A well known and often applied technique for generating lightmaps in the preprocessing step is the **radiosity**. For further information, read [Had02] and [HA01].

In the **first part** of this paper **directional lightmaps** are presented in more detail. By doing so, the advantages and boundaries of the technique are pointed out, too. With the help of various illustrations each step for creating the needed data is presented in a less abstract way. It is also shown, that directional lightmaps are a refined version of classic lightmaps. Only minor changes are made to the preprocessing step and there is more work to do at runtime, but it is still a fast and yet accurate method. It remains to say, that **Valve** refers to directional lightmaps as **radiosity normal maps**.

Besides directional lightmaps, another innovative solution, which was given the name **ambient cube** is presented in the **second part**. This approach helps integrate characters with the world and its static lighting-environment. One can think of ambient cubes as **cubemaps** with a resolution of 1x1 pixels where the stored images essentially only represent a single color-value instead of an environment.

2. World vs. Model

Valve Software's Source Engine distinguishes 2 classes of geometry, namely world and model geometry, respectively. Each of these 2 classes is handled differently by the renderer.

The term **world geometry** denotes large and static data, e.g. the level in which the player moves around. Radiosity light maps are used to give the scene a more realistic look. See section 4 for a detailed description in how the Source Engine enhances traditional radiosity light maps.

Physics props and animated characters belong to the

[†] e9907459@student.tuwien.ac.at

model geometry class. Ambient cubes are employed for their realistic integration with the world. Again, an in depth look at the topic is given later, in section 5.

3. Important terms

As mentioned earlier, the methods described in this paper are enhancements to well known techniques. To improve the readers understanding for the following explanations, a brief revision of these concepts will be presented in this section. If the reader is familiar with **radiosity** (section 3.1), the **tangent space** (section 3.3), **normal maps** (section 3.2) and **environment maps** (section 3.4), the following subsections can be skipped. For those who want to know more about a specific topic, some references are given, too.

3.1. Radiosity

Half-Life 2's radiosity normal maps rely on a radiosity solution. The radiosity solution determines the diffuse lightinteraction in the scene. This includes subtle effects such as indirect lighting (even if from a paricular position in the scene the lightsource is not visible, light can still be transfered through reflection from other positions in the scene) and colorbleeding (if white light is reflected from a red wall, it will become reddish for the next bounce). If enough preprocessing time is invested, soft shadows are produced. Look at figure 1 to get an impression of a sample radiosity solution.

Figure 1: *This figure shows a scene that was calculated with the radiosity method. It shows subtle effects like indirect lighting, color bleeding as well as soft shadows in a diffuse lighting setup. Image taken from [Wil05]; see section 3.1.*

To process a scene, first all surfaces are subdivided into so called **patches**. Each patch will finally be assigned one light intensity, so the more patches a surface is subdivided in, the smoother the shadows will look. The light intensity for a single patch is computed iteratively. Initially, only the patches that represent light sources have positive intensities, all other patches' intensities are set to zero. Since the influence of patch A on patch B depends on their relative orientation, **form factors** are introduced. These form factors encode the energy transfer between patches. In the literature, form factors are written as F_{ij}, which denotes the fraction of light originating at patch i that reaches patch j. It is important to note, that they depend only on the geometry, not on the illumination of the scene. This is a convenient property, since form factor calculation is time-expensive, but needs to be done only once per geometric setup and can then be reused, if for example lightintensities of the lightsources are altered.

There are various ways to calculate the form factors for a scene. The main distinction is between analytic and numeric approaches. In computer games, **hemisphere sampling**, which is a numeric method is commonly used. The hemisphere is placed on each patch (according to the surface normal that the patch belongs to) and then for every other patch it is determined, how much of the hemisphere is "occluded" by this other patch. This fractional part for each pair of patches is their shared form factor. Additional information can be otained by studying [Wil05] or [HB04], in particular the radiosity rendering equation. It is not so important to understand the complete radiosity solution, but rather how the form factors are determined to understand directional light mapping.

In fact, the hemisphere is often replaced by a more convenient hemicube (see Figure 2). By doing this, an ordinary scanline renderer can be used to determine the form factors. The camera that is placed on each patch and five snapshots, that represent the "panorama" of the patch are taken. For rendering, the patches only get unique colors as identifier - so the result looks somehow abstract. Compare this technique to the automatic creation of environment maps in section 3.4.

To actually determine the form factor for each pair of patches, consider this example: the camera is placed on patch **A** and the panorama, which has $\mathbf{N}_A$ pixels, is generated by rendering all other patches with their respective identifier-colors. Patch **B**, which has color $\mathbf{C}_B$ is visible on $\mathbf{N}_{AB}$ pixels in this panorama. This means that the form factor between patch A and B is $\mathbf{N}_{AB}$ / $\mathbf{N}_A$. Note that this technique is similar to generating cube maps [FK03], which are discussed in more detail in section 3.4.

3.2. Normal Maps

In the early days of computer games, textures were just "photographs" attached to polygons. This increased the per-

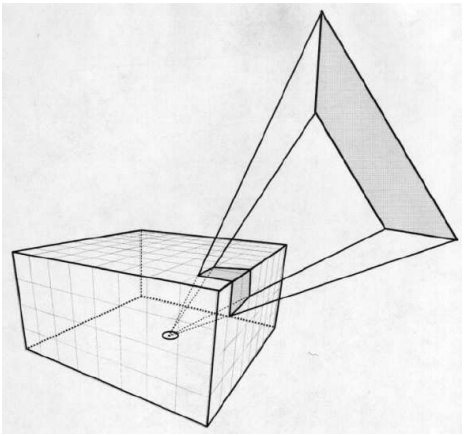

Figure 2: *A hemicube is used to determine the form factor between 2 patches. The dark gray area on the surface of the hemicube corresponds to the projection of one patch onto the hemicube. The fractional part of dark gray hemicube "pixels" to the overall hemicube pixel count is the form factor between those two patches. Image taken from [Wil05]; see section 3.1.*

cepted detail - a brick wall consisted only of a single surface with a corresponding texture applied to it. One disadvantage was that the lighting was statically incorporated into the texture and could not be altered at runtime to adapt to a new lighting situation in the scene.
To overcome this limitation, **bump maps** were introduced [Koe00]. Bump maps are special textures that carry depth information in one RGB channel, just like a height field. **Normal maps** are an improvement to bump maps since they carry information in each of the three RGB channels. This makes it possible to encode a normal for each texel of the normal map. The red channel encodes the normal vector's x component, the green channel encodes the normal vector's y component and the blue channel specifies its z component. 100% red denotes a vector facing right, 0% red represents a vector facing left. Note that the blue channel in a normal map is never smaller than 50%, since this would describe a vector that points "behind" the surface [HT]. This is also the reason why normal maps appear bluish when viewed with an ordinary image viewer.

While the silhouette of the surface is not affected by a normal map, it associates a particular direction with each texel. This vector can be used for lighting calculations (`http://developer.valvesoftware.com/wiki/Normal_Maps`). More about geometry detail classification can be studied in [Pre06].

3.3. Tangent Space

If normal maps are used to increase surface detail, calculations for lightintensities are usually carried out per pixel. Since the normal map is defined with respect to the attached surface, there is now yet another coordinate system - the coordinate system that is attached to this surface. This coordinate system is called **tangent space** and the per-pixel calculations are carried out in this space. The three, pairwise perpendicular basis vectors are the **normal** $\vec{N}$, the **tangent** $\vec{T}$ and the **binormal** $\vec{B}$. The vector $\vec{N}$ is equal to the surface normal, $\vec{T}$ points in the direction in which the first texture coordinate u increases and $\vec{B}$ points in the direction in which the second texture coordinate v increases [Pre06], [Koe00].

3.4. Environment Maps

Environment maps simulate reflection on object surfaces. Imagine a ray that hits a mirror. What is seen is not the mirror, but rather what is hit by the reflected ray. Since it is computationally expensive to actually determine the correct color for each such reflected ray - this is what is done in raytracing [Wil05] - a simplification is necessary. In a computer game it is often enough to see the reflection of the sky in objects that are highly reflective. Only the reflected ray is computed and then used to access a previously generated texture map - the environment map. It is common to organize the environment map as a cube map.

A cube map consists of 6 images which correspond to what is seen along the positive and negative X, Y and Z axis, respectively. Apparently without extensions this method cannot handle reflection of objects in the scene that move around or of anything that is not stored in the cube map. This is a consequence of the assumption that what is seen in the cube map is "infinitely" far away. The reflective object itself may move around without destroying the illusion of a correct reflection.

Cube Maps can easily be created by taking six snapshots when viewing along the positive and negative x, y and z axis, respectively. The camera should be configured to have a field of view of 90° and an aspect ratio of 1. Interestingly, this technique is quite similar to form factor calculation using hemicubes and a camera, already presented in section 3.1

For further information, consult [FK03], [Koe00] or [HA01].

4. Radiosity Normal Mapping

Radiosity normal mapping combines radiosity and normal mapping. Realistic diffuse lighting with soft shadows can be obtained from a radiosity preprocessor. Even though this is computationally expensive, it is an acurrate, stable and fully

automatic way for producing game content. Normal maps on the other hand introduce higher surface detail through elaborated lighting calculations and work with both diffuse and specular lighting models.

Normal mapping typically requires each lightsource to be handled in a separate pass. This is done by summing multiple $\vec{N} \cdot \vec{L}$ terms, where $\vec{N}$ is the current surface normal from the normal map and $\vec{L}$ is the current lightsource's light vector. It is obvious that the number of lights is limited because of this property.

Radiosity normal mapping on the other hand effectively bump maps with respect to an arbitrary number of lights in one pass [McT04]. This property is achieved by preprocessing the lightsources with the radiosity method. In contrast to the traditional radiosity method, one lightmap is computed for each of the three vectors of the basis for radiosity normal mapping (see sections 4.1 and 4.2). It does not matter how many diffuse lights are placed in the scene, since all information is extracted in a preprocessing step and made available during runtime through the three lightmaps for each surface. These three lightmaps together with the normal map can always be combined in one pass!

4.1. Basis for Radiosity Normal Mapping

In order to produce three lightmaps per surface, three slightly different radiosity solutions are computed. Traditionally one patch gets is assigned a single color value, because the form factor is calculated with respect to the surface normal only (for example by placing a hemicube at the surface, pointing into the direction of the surface normal). If we were to place three hemicubes on each patch, facing into the direction of three vectors, this would result in three different form factors and therefore three different color values per patch. These three surface vectors are the **basis for radiosity normal mapping**. In order to do so, the basis must be transformed into the tangent space of this surface.

Figure 3 shows how the basis looks like if the surface lies in the ε_{xy} plane and its normal is $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$. The individual vectors are $\begin{bmatrix} -\frac{1}{\sqrt{6}} \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}, \begin{bmatrix} -\frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}, \begin{bmatrix} \sqrt{\frac{2}{3}} \\ 0 \\ \frac{1}{\sqrt{3}} \end{bmatrix}$.

At first sight the basis looks quite arbitrarily chosen. This is actually not the case. The two constraints for the basis vectors are, that they need to be pairwise orthogonal to each other and that they should evenly cover the hemisphere in the direction of the surface normal. Figures 4 and 5 should convince the reader, that both properties hold for this particular choice for the vectors.

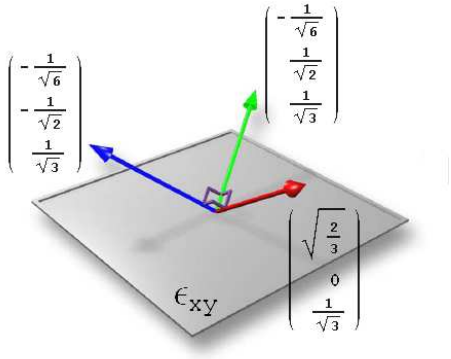

Figure 3: *Basis for radiosity normal mapping. Note that the basis vectors form a cartesian coordinate system, where every two basis vectors are orthogonal to each other. Image taken from [McT04] (slightly modified); see section 4.1.*

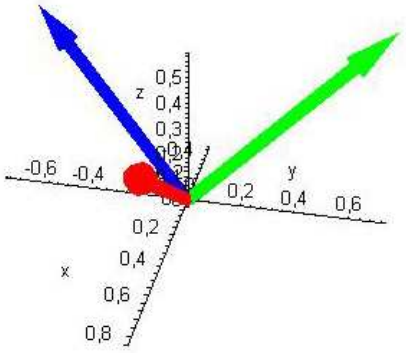

Figure 4: *The basis vectors are "centered" around the surface normal, which is the z-axis in this case. The angle between each basis vector and the surface normal is approximately* $54,74^\circ$*; see section 4.1.*

4.2. Directional Components

By calculating 3 different radiosity solutions, three slightly different lightmaps per surface are generated. Each lightmap is associated with the corresponding basis vector that was used for form factor calculation. Therefore each lightmap is bound to a certain direction and the lightmaps are referred to as **directional components**. Take a look at figure 8 to get an impression of how the directional lightmaps look like.

Given this information, it is easy to realize that, depending on a normal from the normal map, the three lightmaps

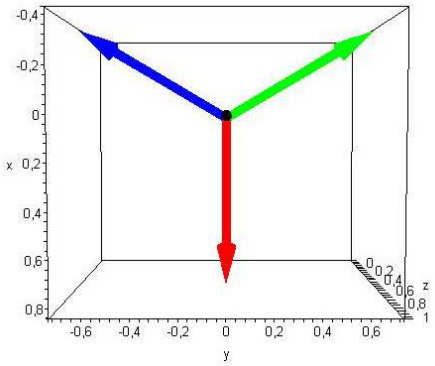

Figure 5: *The angle between the projected vectors onto the surface, which is the ε_{xy} plane in this example, is $\frac{2}{3}\pi$. This can be illustrated by looking at the vectors along the negative surface normal vector; see section 4.1.*

can be combined (for details, see section 4.3) so that finally the result looks approximately like a complete radiosity solution for this direction. Since this combination is done efficiently at runtime, even animated normal maps or dynamically blended normal maps are possible. For example when a bullet produces a hole in a wall, the shading is adapted according to the new "geometry" of the surface.

4.3. Radiosity Normal Mapping Math

During runtime all that remains to be done for each pixel in the final image, is transform the normal from the normal map into the new basis and blend between the three precomputed lightmaps based on the direction of the transformed normal with respect to each of the basisvectors [McT04]:

$$
\begin{aligned}
&diffuseLightColor = \\
&lightmapColor[0] * dot(bumpBasis[0], normal) + \\
&lightmapColor[1] * dot(bumpBasis[1], normal) + \\
&lightmapColor[2] * dot(bumpBasis[2], normal)
\end{aligned}
$$

As a simple example consider a normal from the normal map that coincides with the first basis vector $bumpBasis[0]$. The first dot product is equal to 1 (the vectors are assumed to be normalized), while the other two evaluate to 0 since the basis vectors are orthogonal to each other. The color that is assigned to this pixel is simply the color $lightmapColor[0]$ from the first lightmap.

4.4. Digression: Specular Lighting

So far, only diffuse lightinteraction is taken into account. To improve the visual appearance, the Source Engine also incorporates specular lighting that is based on cube maps. This has nothing to do with radiosity normal mapping, but in section 4.5 the individual results from each step are shown and there is a path for a specular component, too, which shows that specular lighting can easily be incorporated with radiosity normal mapping. This is the reason for this short excursus.

In section 3.4 it was already pointed out, that the environment that is stored in a cube map is assumed to be infinitely far away, so that the relative position can be neglected and all that matters is the reflected vector's direction. If this property does not hold, the observer will most likely notice that the reflection does not look correct.

The approach that is realized in the Source Engine is to use many cube maps. The sample points for those cube maps are set by the designer inside the level-editor through point entities. Each of the cube maps is valid only for a small area in the level; this means that even though now the environment that is stored in the cube map may be near the observer, the reflection looks realistic anyway. Once the reflecting object moves too far away from such an area, another area is entered with yet another stored cube map which is then assigned as the new environment map for this object [McT04].

Please note that because the cube maps are preprocessed, only the static world geometry is reflected on objects. In order to reflected animated objects as well, the cube maps would need to be updated periodically during runtime.

4.5. Shade Tree

In this section, the final rendering result is put together step by step. Consider figure 6 which is our desired image. Figure 7 shows an overview of the "assembly-process".

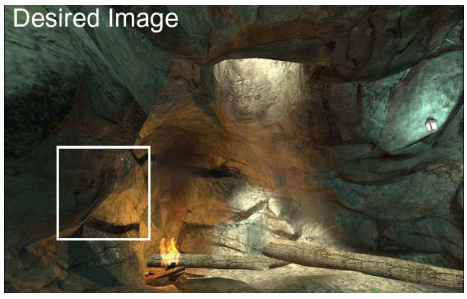

Figure 6: *Final result that is built step by step. Image taken from [McT04]; see section 4.5.*

Figure 8 shows the three different directional lightmaps as well as a traditional lightmap for the same scene. The traditional lightmap can be reproduced by blending the three

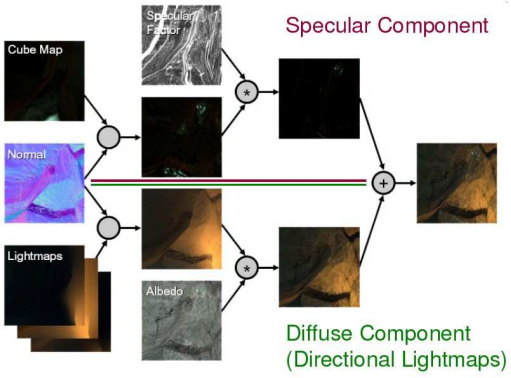

Figure 7: *Schematic overview for how the individual parts from several stages in the pipeline are combined to form the desired result. The upper half shows the paths for the specular part, the lower half deals with the diffuse component - this is where directional lightmaps are used. Image composition is from left to right. Image taken from [McT04] (slightly modified); see section 4.5.*

directional components.

Figure 9 illustrates, how the lightmaps can benefit from normal mapping. Even though the geometry has not changed, it seems that scene-detail has increased.

Figure 10 shows the signifcant improvement of directional lightmapping upon traditional lightmapping. The figure reflects the enhancements of texturing in games since texturing was first introduced.

Figure 11 shows the final result for this snapshot. The only difference to the previous figure (10) is the integration of the specular component. While in this image the difference is not striking, the view-dependent specular part is well visible when the camera moves through the scene. The individual steps for producing the specular component are not presented in detail in this paper, which focuses on the diffuse component only, but can be looked up in [McT04].

5. Ambient Cube

A problem that needs to be solved is the integration of mobile props and animated characters into the complexly lit scene. Since processing power is limited, it is impossible to compute a new radiosity solution for each frame. It turns out that the radiosity solution can however be used to address the problem. In the preprocessing step, right after the radiosity solution is computed, so called **ambient cubes** are generated at some predefined 3D grid-positions in the whole level. Although in [McT04] there is no explanation

Figure 8: *The three directional components in comparison to a traditional radiosity solution, which can be otained by blending the three lightmaps from the directional components. Each of the directional components is a radiosity solution with the hemicube placed on the same patch but each time facing along another basis vector. Through the 3 lightmaps directional dependencies are preserved and can therefore be used together with normal maps for elaborated shading effects on surfaces (see figure 9). Image(s) taken from [McT04]; see section 4.5.*

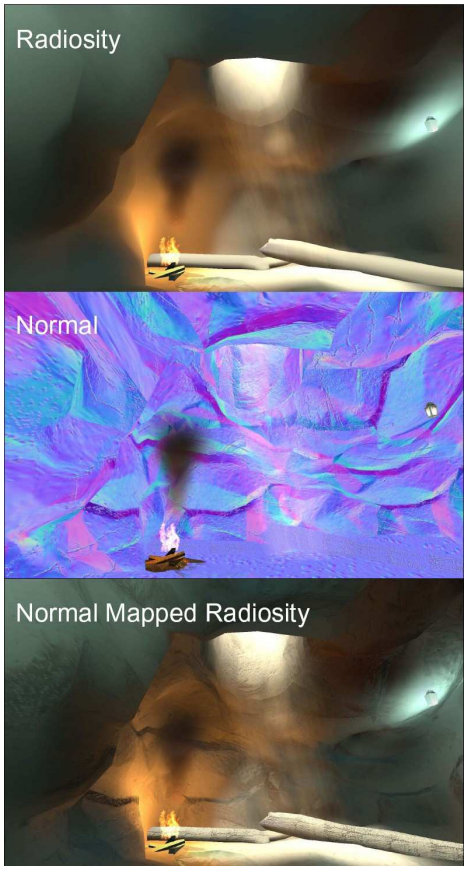

Figure 9: *This figure illustrates the combination of lightmaps and a normal map. Through elaborated shading the detail in the scene is increased. Image(s) taken from [McT04]; see section 4.5.*

Figure 10: *From top to bottom: Simple texturing (e.g. Ultima Underworld, Castle Wolfenstein, Doom I) - photos of a rock can be used for texturing the cave; Traditional lightmapping (e.g. Quake I) - combine the lightmap from the radiosity preprocessing step with the photo texture; Directional lightmapping (e.g. Half-Life 2) - three lightmaps and a normal map increase the scene detail significantly when combined with the photo texture. Image(s) taken from [McT04]; see section 4.5.*

for how these ambient cubes are generated, one possible approach would be to generate an environmental cube map at each grid-position (see section 3.4).

While generating this special purpose cube map, only the lightmaps are attached to the surfaces. In order not to consume too much memory, the cube map's six faces are postprocessed. The mixed color of all pixels for a face is determined, so finally a resolution of 1x1 pixels is sufficient for each face of the ambient cube (see figure 12). Essentially now each face stores a single color, which approximately represents the ambient light flow through this volume in space.

It is important to note, that again the ambient cubes are generated in the preprocessing step. It is further possible to compute them only once after level creation and store them together with the geometric data, so that it does not take too much time to load a level. At runtime only a lookup into the ambient cube data is performed to sample the colors for the volume the object is currently in so that it can be seamlessly integrated with the world via **indirect lighting** (take a look at figure 13). Again a directional dependency emerges. Ambient cubes can be seen as the **volumetric equivalent** to directional lightmaps, which are defined for 2D surfaces.

Ambient cubes however do not produce shadows - the shadows that objects cast, are generated with yet another technique (again this is not mentioned in [McT04]). The specular lighting for models works similar to the world specular lighting - the best / nearest cube map sample is cho-

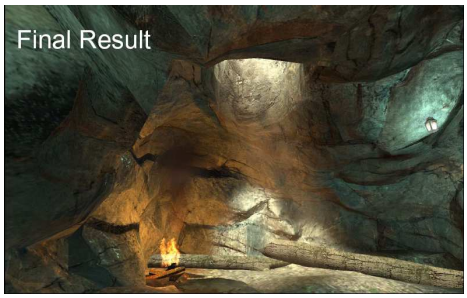

Figure 11: *The final result. Additionally to directional lightmapping, the specular part has been integrated as well. In the image the difference is hardly visible; since the specular part is view-dependent, the difference is better visible when the camera moves through the scene. Image taken from [McT04]; see section 4.5.*

sen every frame and then used for generating the reflection. In the actual implementation, 2 local lights are additionally used for rendering models (necessary for the shadows) and any other local lights that are not important enough are added to the ambient cube as well.

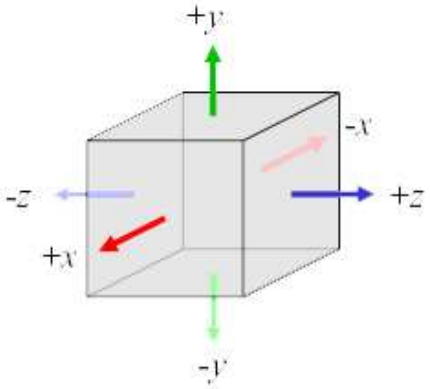

Figure 12: *Ambient Cube. A single color value for each face determines the lightflow along this direction through this volume in space. Image taken from [McT04]; see section 5.*

5.1. Ambient Cube Math

Assume that *worldNormal* , *nSquared* and *linearColor* are vectors with 3 float components and that *isNegative* is a vector with 3 integer components. The ambient light is sampled from the ambient cube's data as follows [McT04]:

$$nSquared = worldNormal * worldNormal$$
$$isNegative = (worldNormal < 0.0)$$

Figure 13: *Ambient Cube in action - two characters integrated into a complexly lit scene - from the left white / bluish light strikes the characters, while from the right they are softly lit by red light originating from the liquid in the background. Image taken from [McT04]; see section 5.*

$$linearColor = nSquared.x * cAmbientCube[isNegative.x] + nSquared.y * cAmbientCube[isNegative.y + 2] + nSquared.z * cAmbientCube[isNegative.z + 4];$$

The value that is assigned to *nSquared* is actually the length of worldNormal to the power of 2. Since it is normalized, the length is equal to 1.

The value that is assigned to *isNegative* is a componentwise encoding of the test to determine, if *worldNormal* is facing into the positive or the negative direction of the x, y, or z axis. For example, the x-component of *isNegative* is set to 1, if the x-component of *worldNormal* is smaller than 0, otherwise the x-component of *isNegative* is set to 1.

In the final assignment *nSquared* is used as the scaling factor (the sum of its components is equal to 1 as mentioned before) and the encoded values in the new *isNegative* vector are used to distinguish "front- from backfacing" ambient cube faces for the current normal vector.

6. Conclusions

Once again interactive frame-rates are bought at the cost of complex preprocessing-calculations and increased memory requirements (for the lightmaps, the ambient cubes and the cube maps). Additionally throughout the text one could see that an intelligent mixture of various approaches and techniques can produce better effects than "monolithic" solutions. Here, by monolithic is meant, that for example specular and diffuse lighting effects are handled by the same method instead of using two different methods that can only

handle either of the cases.

Through the preprocessing step it is possible to handle an arbitrary number of lightsources at constant frame rates. What is thereby achieved is a decoupling of complexity and this is also the real strength of directional lightmaps. The method is highly efficient and still introduces impressive improvements in quality.

Ambient cubes address the need to integrate animated characters and other, non static objects with the "perfectly lit" world geometry. Again the method mainly increases preprocessing time and memory requirements but consumes little processing power during runtime.

Directional lightmaps and ambient cubes are not used solely by the Source Engine. The **Unreal3** engine (see http://www.unrealtechnology.com/html/technology/ue30.shtml) and the **Crysis** engine (see http://crysis.4thdimension.info/dlsarelogged/_images/videos/crysis_gdc2006_4.jpg and http://selectivegamers.com/content/view/390/146) will both be using similar techniques for world and model shading. Other developers are likely to integrate similar ideas into their future state of the art engines.

References

[FK03] FERNANDO R., KILGARD M. J.: *Cg Tutorial, The: The Definitive Guide to Programmable Real-Time Graphics*. Addison Wesley Professional, 2003. http://developer.nvidia.com/object/cg_tutorial_excerpts.html, http://www.aw-bc.com/samplechapter/0321194969.pdf.

[HA01] HAWKINS K., ASTLE D.: *OpenGL Game Programming*. Prima Publishing, 2001.

[Had02] HADWIGER M.: *Game Technology Evolution*, 2002. http://www.vrvis.at, http://www.cg.tuwien.ac.at/courses/CG2/SS2002/Gaming.pdf, http://www.cg.tuwien.ac.at/courses/CG2/SS2002/Gaming_slides.pdf.

[HB04] HEARN D., BAKER M. P.: *Computer Graphics with OpenGL, 3rd Edition*. Pearson Prentice Hall, 2004.

[HT] HASTINGS-TREW J.: *Creating Normal Maps with Cinema 4d*. http://planetpixelemporium.com/tutorialpages/normal.html.

[Koe00] KOENIG A. H.: *Texturing*, Mar. 2000. http://www.cg.tuwien.ac.at/courses/CG2/SS2002/Texturing.pdf, http://www.cg.tuwien.ac.at/courses/CG2/SS2002/Texturing_slides.pdf.

[McT04] MCTAGGART G.: *Half-Life 2 / Valve Source Shading*. Tech. rep., Valve Software, Mar. 2004. http://www2.ati.com/developer/gdc/D3DTutorial10_Half-Life2_Shading.pdf.

[Pre06] PREMECZ M.: *Iterative Parallax Mapping with Slope Information*, 2006. http://www.cescg.org/CESCG-2006, http://www.cescg.org/CESCG-2006/papers/TUBudapest-Premecz-Matyas.pdf.

[Wil05] WILKIE A.: *VO Rendering SS 2005, Unit 2: From CG1 to Rendering*, 2005. http://www.cg.tuwien.ac.at/courses/Rendering/Slides_2005/RenderingVO_SS2005_02.pdf.