POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

COMPUTER GRAPHICS: SHADING WITH DYNAMIC LIGHTMAPS

Relatore: Prof. Marco Gribaudo

Tesi di Laurea di:
Davide Bianchi, matricola 817048

Anno Accademico 2015-2016

Dedicata ai miei genitori e agli amici che mi hanno sempre sostenuto e sopportato.

Abstract

Nowadays video-games industry is focused on providing video-games with astonishing graphic and realistic animations. In order to be able to further increase the quality of their graphics video-games are now hungry of computational resources especially from the graphic adapter. One of the most difficult and computationally expensive process in the graphical realization of a 3D scene was the computation of the correct distribution of light on the surfaces.

In the early '90s John Carmack and Michael Abrash where trying to get rid of some of the Gouraud Shading technique disadvantages to enhance the quality of their video-game Quake and developed a system called Lightmap. The lightmap system has been widely used for more then a decade thanks to it's good results and extremely low resource consumption to recreate the light effect of almost every scene of every video-game until the graphic resources became powerful enough to fulfil the performance needs of better algorithms. Now lightmaps are still used, even if not as much as twenty years ago, to speed up the computation of some lights baking everything possible (e.g. Unreal Engine 4).

In this thesis we will try to implement a system that will bypass one of the major disadvantages of the lightmaps: the fact of working only with static lights.

We'll try to provide a method that, preserving the performance advantages of the common lightmap and the quality of other algorithms, will allow to use them on a specific subset of dynamic lights; lights that can not be moved by the player and that has fixed or looping path.

Contents

List of Figures

Chapter 1

Introduction

In this chapter will be presented the field which this project will interact with, the author contribution to the field of research and the structure of the paper.

1.1 Field of research

In modern video-games, the graphic aspect is gaining more and more importance. When we talk about "Graphic" in a video-game we refer to a variety of factors, both technical and artistic that mixed together give a feeling of uniqueness to the player. Some of the most important factors are: image quality, models precision and definitions, animations correctness and realism, and how engaging is the audio. This paper will put its focus on the illumination of scene, how to render the lights and all the performance implications of the suggested method in modern video-games.

As we will see in the next chapters, calculating the correct illumination of a *scene* is a very difficult operation that require the computation of extremely complex equations which takes a large amount of time and it's not always possible to compute at *runtime*. There are several models that try to provide a viable solution depending on the performances of the system on which they are running and the quality of the final result, each of them with its advantages and disadvantages.

1.2 Contribution

This paper aims to utilize and old technique widely used in the '90 and in the first 3D video-games, developed by John Carmack and Michael Abrash

for their game Quake. This technique is called Lightmap.

The lightmap technique was created to get rid of some technical disadvantages of the Gouraud Shading Technique and was based on a pre-calculated image (or Texture) on which was represented the light distribution of the scene. This image was then overlapped to the in-game scene when the light that had generated that scene was turned on. Since all the computational process was done *offline* it was possible to create extremely accurate scene from the light point of view almost for free, which was an enormous advantage for video-games in that period.
This characteristic gave the possibility to utilize more complex and heavy method than the Gouraud which were able to provide much better results without having an impact on the game performances. This system came with a big disadvantage, it could work in fact only with static light, for instances chandeliers. Light sources such as torches, vehicles lights and strobe lights couldn't work because since they move along the scene during time was impossible to imprint them in a static image. In later years, with computers beaming more and more powerful, lightmaps were slowly abandoned to use better algorithm that could handle static and dynamic light with the same quality.
In modern video-games there are still some situations in which lightmaps are used (Unreal Engine 4) because they still provide some benefit both from a quality and a performance point of view. This papers want to suggest a way to utilize lightmaps when the light source is not static and can be defined a precise loop for its path.

1.3 Structure

The structure of the paper is the following.

Section "State of art": Will be described the terminology and the various technique used to distribute the light in a scene.

Section "Problem": The problem is described along with the various technique that will be used to solve it.

Section "Solution": The procedure and the algorithms used to solve the problem are described.

Section "Performance analysis": The performance, both from a quality and computational point of views, are compared with other known lighting systems and described.

Section "Conclusions": Possible improvements and future works are described.

Chapter 2

State of art

In this section will be presented the technologies and the theory behind the computer graphic and illumination systems.

2.1 Computer Graphics

Computer Graphics is a field to which belong several disciplines, not only technical but also artistic, and it study the creation, editing and representation of digital images trough the use of computers. The term *Computer Graphics* has been used for the first time by researchers William Fetter and Verne Hudson in 1960 and an equivalent way to call Computer Graphics is CGI; acronym of Computer-Generated Imagery.
Some example of disciplines that belongs to CGI are: vectorial graphic, 3D modelling, shading, animations, image editing and much more.

2.1.1 Colors and monitors

If it is on a screen it is CGI.
In order to show on a screen what's inside a computer we need a tool to use as interface between the monitor and the computer itself, this device is called *Graphic adapter*. We know three different type of them but only one is used nowadays.

2.1.1.1 Vector graphic adapter

Vector graphic adapter was the first type of graphic adapter ever created. They work in a quite simple way. Behind the screen there is laser, called *beam* that can be turned on and off and can be move along all the surface of the screen leaving a visible trace that last few milliseconds, the graphic

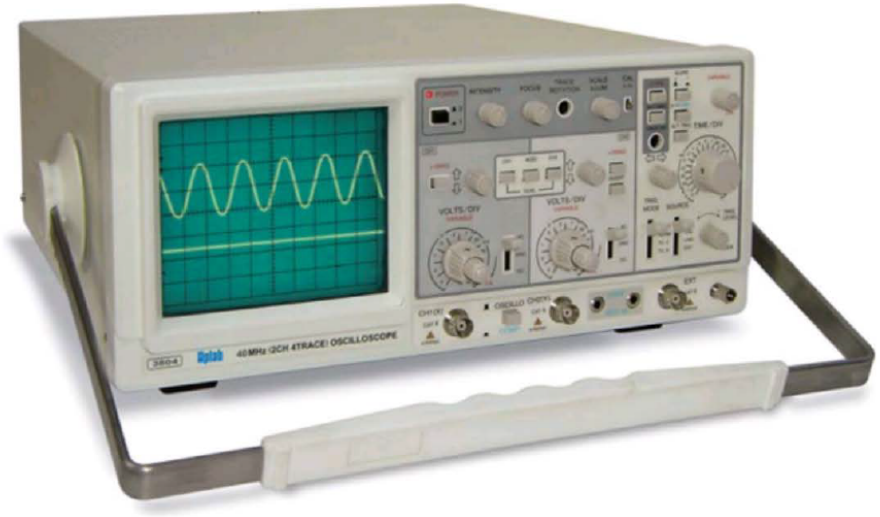


Figure 2.1: An oscilloscope, classic example of Vector graphics adapter.

adapter turned the image in a series of movement and on/off commands to reproduce the image on the screen. This kind of tools were available from the '70s and were the system used by the old cabs until the first '80s. One of the most famous cabs was Battlezone, from Atari (1980) 2.2.

Figure 2.2: Battlezone - Atari - 1980.

To use this graphic adapter programmers had to specify step by step, all the movement of the beam. In the example below the beam will draw a triangle on the screen.

```
move 20,20
beam on
```

```
move 40,60
move 60,20
move 20,20
beam off
```

2.1.1.2 Raster

New generation of monitors were provided with a grid, or matrix, of singularly accessible element, individually programmable also known as pixels (Picture elements).

Figure 2.3: An LCD monitor

The graphic adapter choose the colour of each pixel based on the image that it has to represent.
This graphic adapter has a special type of memory called Video Memory (VRAM). The colour of each pixel on the screen was directly related to the value of each VRAM byte (or bytes) which is converted by a component called RAMDAC in a way that it could be used by the monitor to apply the colour to the corresponding pixel.
Below you can see an example of how is possible to assign to the pixel number 64 (40h) the colour of value 33 (21h):

```
writeScreen()
{
    *(0x0040) = 0x21;
```

}

2.1.1.3 Accelerated

When the cost of the memory dropped, became possible to build raster graphic adapter with much more memory than the one required to simply draw the image on the screen, this kind of graphic adapters were called Accelerated graphic adapter.
In these devices the portion of memory used to decide the colour of the pixels on the screen is locked by the device itself; the programmer can send images or other data in particular addresses of the VRAM and then send commands to modify the images themselves. For instance the programmer can send the image of a landscape along with the set of commands: draw a segment from $(x1, y1)$ to $(x2, y2)$, print some text, rotate 13 degrees. All the computation are done in this separate portion of the VRAM called *screen buffer*. When the composition of the image is complete the memory dedicated to the screen buffer is locked and will be use to display the next frame as the monitor will request it, the old locked memory is released and can now be used to prepare the next frame.

2.2 Lighting and Shading

Light is an electromagnetic wave that is perceived in different colours by our eye depending on its frequency.

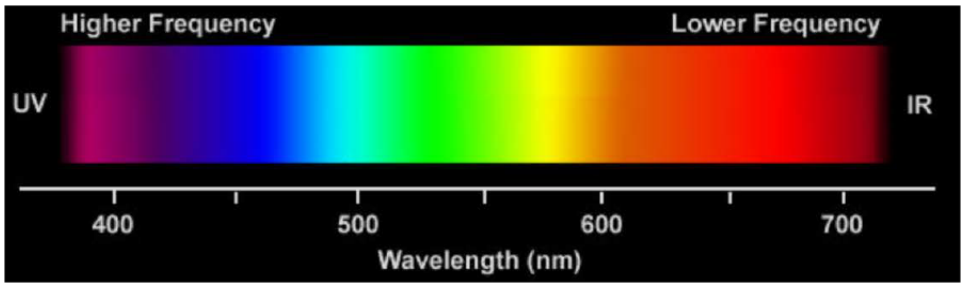


Figure 2.4: Light specter

2.2.1 How does the human eye works?

The human eye is a complex video acquisition system and works like a video-camera or a microscope.

The light enter in the eye trough the cornea, iris get stretched to tune the amount of light getting in and the crystalline project on the retina the 2D image. On the retina there are spread all over different sensors that are stimulated by electromagnetic waves of a with between 250 and 780 nanometres, these sensors read the image which is sent to the brain that elaborate the images from both eyes and recreate the 3D scene. 2.5.

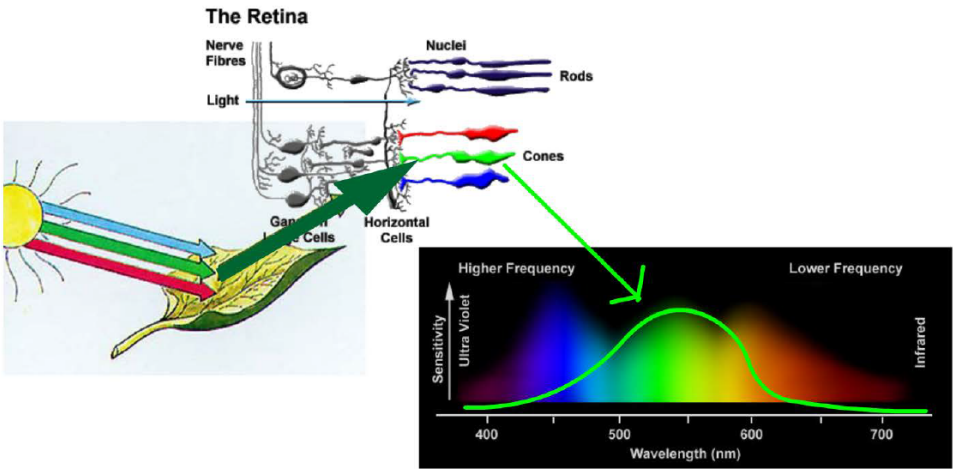


Figure 2.5: Light travelling to the brain

One type of sensor is the *rod*, extremely sensible and used in night vision, can not perceive colours. The other type is the *cone* which is used to distinguish the colours. The amount of rods and cones define the "resolution" of the human eye, or the smallest pair of points that we are able to distinguish.

Human eyes sensors doesn't react equally to all the frequencies of the light spectrum, not only because we have one type of rod and three type of cones but also because each type of cone has a different sensibility with respect to the others. In fact, given two equally intense source of light, we perceive red light brighter than green light. But having just three type of cones is also an advantage because since we base the colours that we see on the amount of red, green and blue light we receive we can build monitors and encode colours in the same way, so using the amount of red, green and blue. This encoding is called RGB (Red, Green, Blue).

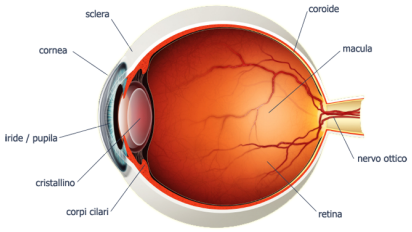


Figure 2.6: Human eye

2.2.2 Light and matter

When a 3D model is loaded into memory and go through the rendering pipeline, one of the most important steps is the computation of the color of each pixel composing its surfaces. The color can be considered as the light that a particular pixel emit by itself or by reflection in a given moment. For this reason, in this chapter, we will consider equivalent talking about color and light emitted (or reflected) by a point.

This, as we will see, is not a trivial problem and require many and different techniques. In order to understand the computational effort and the complexity of this important step we need to introduce the *Rendering equation*.

2.3 Rendering equation

From a physical perspective, a point can either emit light by itself, like a light bulb or a fluorescent surface, or reflect light, like a mirror or a smooth metal, that come from the environment. Some points may concurrently have a reflective and emissive component. If we consider different objects in an environment, the color that we perceive from each single point is given by the summation of the light emitted by the point plus the light incoming from the other points that is reflected plus the light of the original point that is reflected back from the points surrounding.

The correct mathematical approach to precisely calculate the color of each pixel is the following:

$$L(x, \omega_r) = Le(x, \omega_r) + \int L(y, \overrightarrow{yx}) f_r(x, \overrightarrow{yx}, \omega_r) G(x, y) V(x, y)\, dy \qquad (2.1)$$

The (2.1) is the so called *Rendering equation* which return the radiance of a point x towards a direction ω_r.

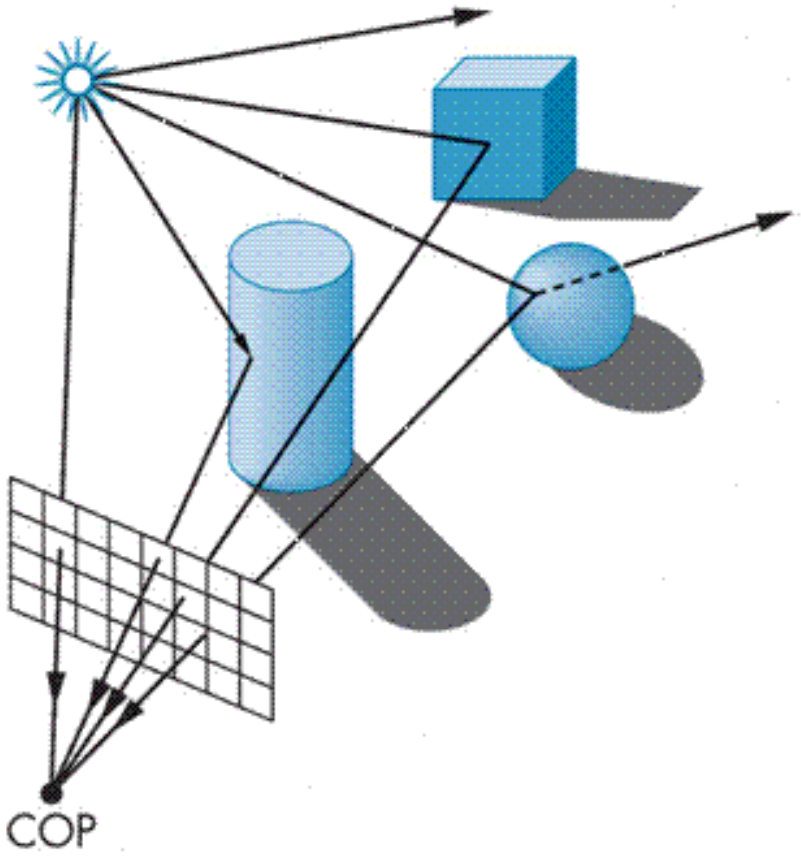


Figure 2.7: Light emitted by a point bouncing on multiple surfaces.

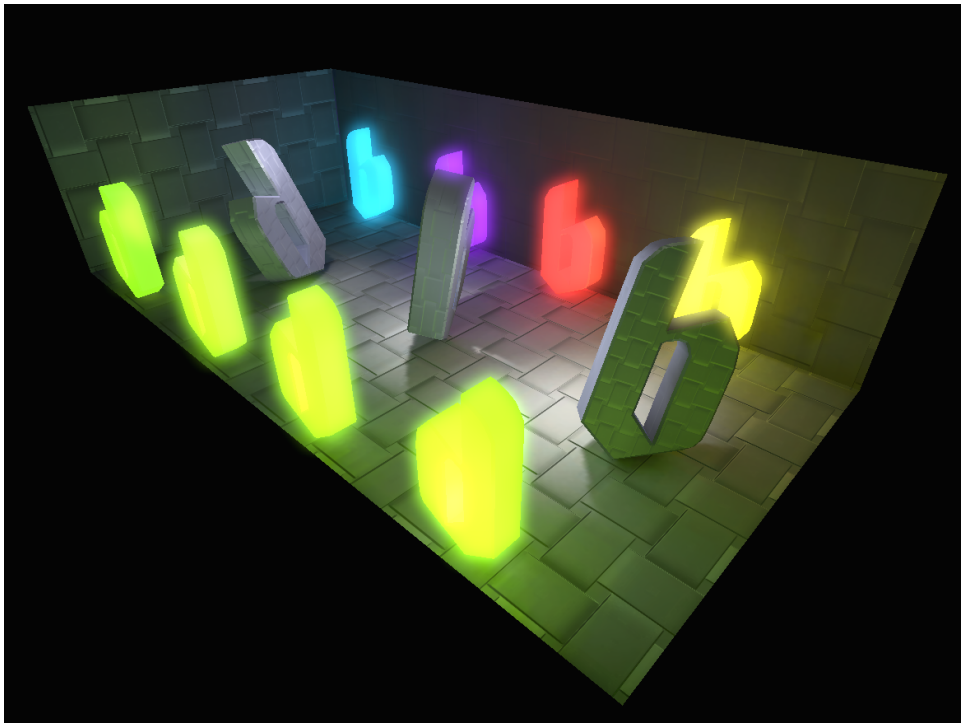


Figure 2.8: Objects with a strong emissive component in a scene

This equation, in principle, can be used to find the correct color for each pixel in the scene. Unfortunately it can not be solved analytically in the general case. One possibility is to try to use numerical methods. This approach is used in several rendering engine of 3D editors but is still too complex to be used in real-time application with the actual computers performances.

Do exist several approximate approaches, that we will see in the next chapters, each of which is an excellent approximation to the rendering equation for particular type of surfaces depending on how it deal with each part of the equation.

Before the illustration of the different techniques we will go into detail of each part of the (2.1).

2.3.1 Emissive component

The first component of the (2.1) is the emissive term. This is simply the amount of light that the point emit by itself in the direction ω_r. In 2.8 we can see an example of the effect of the emissive light coming out from the glowing elements. The emissive component is usually set to zero in a normal

scene except for light sources and some rare shining material. This is the only term that depends by the point only.

2.3.2 Non emissive components

The integral of (2.1) determines the effects that the light irradiated by the surrounding points in the scene has on the color of the pixel. This amount of light is calculated by multiplying together four functions:

- The radiance emitted toward the considered point x by y, defined as *Radiance component*
- The Bidirectional Reflectance Distribution Function (2.2) on x from y to ω_r
- The geometric relation between x and y (2.4)
- The visibility function between x and y.

2.3.2.1 Radiance component

$L(y, \overrightarrow{yx})$ is the amount of light incoming from a point y toward x and is calculated using the (2.1).

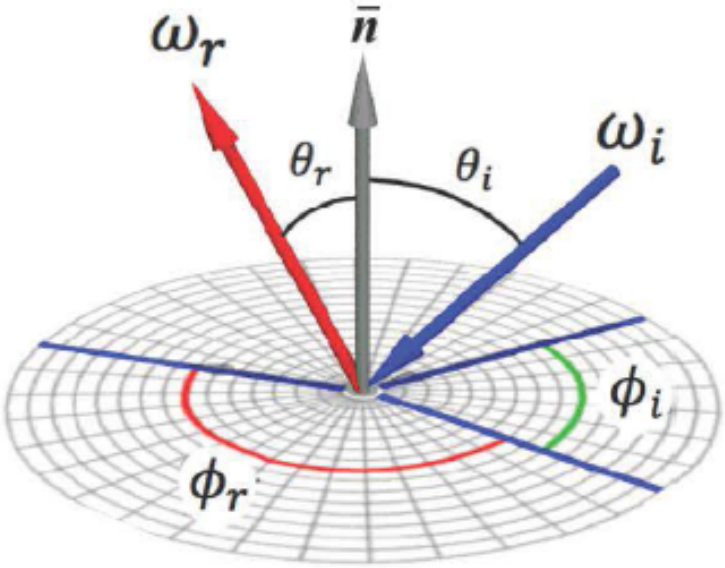
We define as L the rendering equation (2.1), as x the considered point in the scene and as y another point in the same scene. The normalized vector $\overrightarrow{yx}$ represent the direction ω.

2.3.2.2 Bidirectional Reflectance Distribution Function

The second term in the integral equation is the BRDF. This equation is defined as follow:

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = f_r(\omega_i, \omega_r) = \frac{dL_r(\omega_r)}{dE_i(\omega_i)} \tag{2.2}$$

The (2.2) encode the property of the surface material.



The function input are the incoming and outgoing light directions with respect to the normal of the surface. This function tells how much of the light in the outgoing angle is reflected by the incoming angle. $dL_R(\omega_r)$ Represent the amount of light hitting the surface in a direction and $dE_i(\omega_i)$ the fraction of light that is effectively received.

By considering that the input irradiance is proportional to the cosine of the incoming angle with respect to the normal vector of the considered surface we can express $dE_i(\omega_i)$ as $dL_i(\omega_i)cos(\theta_i)\omega_i$. The final result is expressed by the new equation:

$$f_r(\omega_i, \omega_r) = \frac{dL_r(\omega_r)}{dL_i(\omega_i)cos(\theta_i)\omega_i} \tag{2.3}$$

The first parameter x in the (2.1) represent the surface, $\overrightarrow{yx}$ is the incoming angle and ω_r is the outgoing angle.

2.3.3 Geometric relation and visibility

The last two terms that we need to consider in the (2.1) are the *Geometric Relation* and the *Visibility Function*. Starting from the simplest component, the Visibility Function $V(x, y)$ simply represent how much of the point y is visible by the point x. This amount is usually 1 or 0 depending on the presence or not of an obstacle between the two points but it can be a number between 0 and 1 in the presence of an opaque glass.

The Geometric Relation instead depend on the Rendering Equation in use. For the ideal point case it is defined as:

$$G(x, y) = \frac{cos(\theta_x)cos(\theta_y)}{r_{xy}^2} \tag{2.4}$$

The r_{xy}^2 represent the squared distance between the two points x and y.

2.4 Light Source

We have seen that light can leave a surface through two fundamental processes: self-emission and reflection. Talking about self-emission we usually think of a light source as an object that emits light, however it can also reflect some light that is incident on it. If we remember, in the (2.1) we consider the emissive property of a material but it does not preclude it from reflect the light that hits it.

Fortunately this reflection is so tiny with respect to the total amount of light usually emitted by an object such as a light bulb that we can consider it null when we have to calculate it. Indeed if our purpose is to display the correct color of a pixel on the screen we do not have enough bits to encode that undetectable amount of light. Things may change in scientific studies about light reflection but this is not our case.

Another real fact about light sources is that not only do light sources emit different amount of light at different frequencies, but their directional properties can vary with frequency as well. A physically correct model is again really hard to compute but the model of the human visual system is base on three-color theory that tells us we perceive three tristimulus values, rather than a full-color distribution. For this reason we describe a light source through a three-component luminance function:

$$I = \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix} \tag{2.5}$$

The last thing we have to consider about light sources is the possible presence of a diffuser. In many cases lights are designed to provide uniform illumination throughout rooms and this is achieved with using a special material that scatter the light in all directions called diffuser. We could calculate an approximation of such illumination by modeling all the distributed sources and integrating the illumination from these sources at all points of

the reflecting surface but this would be again very heavy and computationally expensive. A smarter technique is to integrate the desired effect that this kind of light should have on the surface directly on the surface itself. This uniform lightning is called *ambient light*. In order to do this we postulate and ambient intensity I_a that is identical at every point in the scene.

Summing up all these approximations we can define three different kind of light sources.

2.4.0.1 Point Sources

A Point Source or *Point Light* emits light equally in all directions from a point p_0. This point p_0 is then characterized by a three-component color matrix:

$$I = \begin{bmatrix} I_r(p_0) \\ I_g(p_0) \\ I_b(p_0) \end{bmatrix} \tag{2.6}$$

and the intensity of illumination received from a point source is proportional to the inverse square of the distance between the source and surface.

$$i(p, p_0) = \frac{1}{|p - p_0|^2} I(p_0) \tag{2.7}$$

The 1 at the numerator can vary depending on how much a unit represent in the scene and the power of two at denominator can be linear or null depending on a constant, inverse linear or inverse square decay of light.

2.4.0.2 Spotlight

Spotlights are characterized by a narrow range of angles through which light is emitted (θ). Spotlights behave exactly like point light with the exception that more realistic models vary the intensity in function of the angle ϕ between the direction of the source and a vector $\vec{s}$ to a point on the surface.

2.4.0.3 Distant Light Sources

Distant Light Sources or *Directional Light* are characterized by a constant angle of the outgoing light ray and a constant intensity regardless of the distance.

2.5 Phong reflection model

Now that we have the basic concepts about how to represent light and how it is emitted and reflected it's time to use those concept to introduce one of

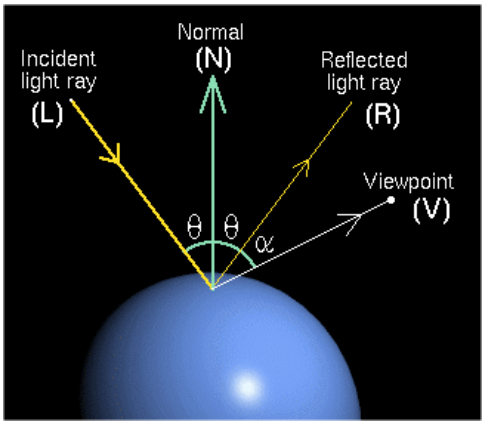


Figure 2.9: Phong relfection model vectors

the most famous reflection model known as Phong.

The Phong reflection model was introduced for the first time by Bui Tuong Phong [6], and later modified by Jim Blinn.

The Phong model uses for vectors 2.9 to calculate the colour of a point P on a surface. The vectors are: n the normal to the surface at point P, v which is the direction $\overrightarrow{PV}$ from the point to the viewer, l is the direction to the light-source $\overrightarrow{PL}$ and r is the direction in which the ray is reflected. For simplicity we will always consider this vectors normalized in the next formulations.
Phong supports three different base type of materials interaction with light: ambient, diffuse and specular. We can use three vectors to describe them then we can put them together to form a matrix L.

$$L = \begin{bmatrix} L_{ra}, L_{ga}, L_{ba} \\ L_{rd}, L_{gd}, L_{bd} \\ L_{rs}, L_{gs}, L_{bs} \end{bmatrix} \tag{2.8}$$

The first row contains the ambient vector, the second row the diffuse and the third the specular. Each coefficient represent the amount of light for each type hitting the surface.

Give that we need to define the interaction of the material with those kind lights, we'll so create another matrix R of the same type of L and its coefficients will depend on several factors, such as: the material properties, the direction of the light source, the orientation of the surface and eventually the distance from the light source.

An example of the amount of red colour seen from a point is (2.9).

$$I_r = R_{ra}L_{ra} + R_{rd}L_{rd} + R_{rs}L_{rs} = I_{ra} + I_{rd} + I_{rs} \tag{2.9}$$

If we have multiple light sources we have to add them all together in addition with an eventual global ambient light.

$$I_r = \sum_i (I_{ra} + I_{rd} + I_{rs}) + I_{ar} \tag{2.10}$$

where I_{ar} is the global ambient light.
The sum of all the colours give us the final result.

$$I = I_r + I_g + I_b \tag{2.11}$$

2.5.1 Ambient Reflection

The ambient light intensity is always the same in every point of the scene hence on each point of the surfaces. Even if it has the same intensity on each point it can still be partially absorbed so the reflected amount doesn't have to be 1. We call this amount the *ambient reflection coefficient* k_a ($R_a = k_a$). Obviously this amount can only be between 0 and 1.

$$0 \leq k_a \leq 1 \tag{2.12}$$

so,

$$I_a = k_a L_a \tag{2.13}$$

2.5.2 Diffuse Reflection

The diffuse reflection accounts for the directed light reflected by the surface equally in all directions. This is possible if the material is rough on a microscopic scale, with small ripples that allows it to reflect lights in all the directions. A surface rough enough to reflect perfectly equally the light in all direction is called a *Lambertian surface* and can be modelled following the Lambert's law.

The amount of light reflected is proportional to the angle of incidence of the light striking the surface. The diffuse contribution at any particular point on a surface is the same, regardless of where the viewpoint is. The mathematical formulation to calculate the diffuse reflection is:

$$I_d = k_d L_d max(n \cdot l, 0) \tag{2.14}$$

2.5.3 Specular Reflection

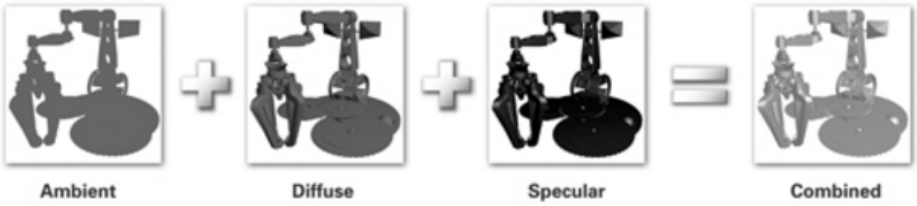
The specular reflection represents the light scattered from the surface around the mirror direction and is quite the opposite of the diffuse reflection. The specular reflection is indeed most prominent on very smooth and shiny materials, such as polished steel. This is the only reflection among the three that depends on the viewer location. In addition to the already described terms that we've used so fare, the specular reflection requires a specific one for each material that is its *shininess* α.

The mathematical formulation for to calculate the specular reflection is:

$$I_s = k_s L_s max(r \cdot v, 0)^{\alpha} \tag{2.15}$$

2.5.4 Final result

A possible implementation of the Phong reflection model using the Phong shading technique is the one suggested below.




```
void PhongLight( float4 position : POSITION,
                 float3 normal: NORMAL,
             out float4 oPosition : POSITION,
             out float4 color : COLOR,

         uniform float4x4 modelViewProj,
         uniform float3 globalAmbient,
         uniform float3 lightColor,
         uniform float3 lightPosition,
         uniform float3 eyePosition,
         uniform float3 Ke,
```

```
        uniform  float3 Ka,
        uniform  float3 Kd,
        uniform  float3 Ks,
        uniform  float shininess)
{
        oPosition = mul(modelViewProj, position);
        float3 P = position.xyz;
        float3 N = normal;
        float3 emissive = Ke;
        float3 ambient = Ka * globalAmbient;
        float3 L = normalize(lightPosition - P);
        float diffuseLight = max(dot(N,L),0);
        float3 diffuse = Kd * lightColor * diffuseLight;
        float3 V = normalize(eyePosition - P);
        float3 H = normalize(L + V);
        float specularLight = pow(max(dot(N, H), 0),
                                  shininess);
        if(diffuseLight < 0) specularLight = 0;
        float3 specular = Ks * lightColor
                             * specularLight;
        color.xyz = emissive + ambient
                       + diffuse + specular;
        color.w = 1;
}
```

For simplicity we consider the light color unique so $L_a = L_d = L_s$.

2.6 Gouraud shading technique

The Gouraud shading technique [4] was named after it's inventor Henri Gouraud in 1971 and is an application of the Phong reflection model.
It perform the calculation as described by the Phong reflection model to determine the colour of a point, a vertex, and instead of calculating again the light for each pixel in the scene, Goruaud do the calculation on just the vertexes of each polygon. The results are then interpolated over the surface of the polygons to give to each pixel its own colour.

This technique had a great success because it was extremely fast, since heavy calculation were done just once per vertex and not per pixel, and was able to provide incredible results compared with the old methods of the time.

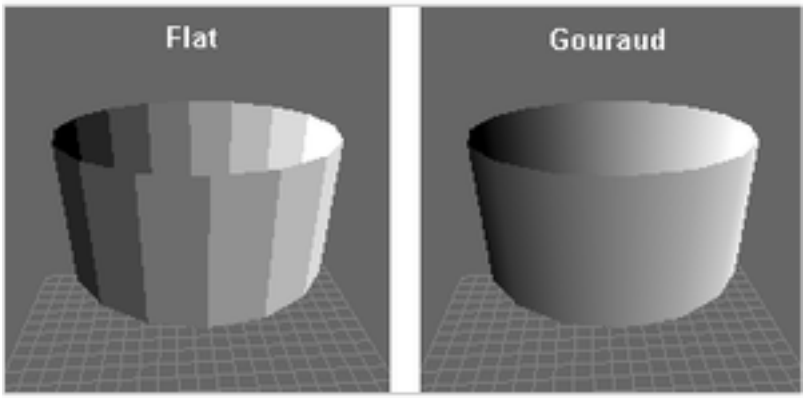


Figure 2.10: Flat shading vs Gouraud shading

As can be seen in the image 2.10 curve surfaces were much better represented by this technique as long as they were rotated around just one axe. The problem comes when the object is a sphere or something that has curves on multiple axis.

In the image 2.11 is clear that the fewer the vertexes the poorer are the results. So in order to have an acceptable quality for sphere and other spherical objects we need to use a lot of polygons which will mean a lot more calculation since the number of vertexes drastically increase. Tessellation is not the only problem of the Gouraud shading technique, here is a list of the most important issues to consider when using this implementation:

- The quality heavily depend on the size of the polygon being drawn
- Highlights are calculated only at vertices and the interpolation is linear leading to monotonic colour gradient across polygon surfaces.
- Shading isn't perspective correct because light change linearly across the surface and unless the polygon is parallel to the screen the same sort of perspective correction is needed to step lightning across the polygon properly.
- Shading a polygon with more than three vertices corrupts the consistency of the technique 2.12.

The Phong shading technique solve all these problems since it is calculated per pixel and not per vertex but is much heavier than its per-vertex counterpart 2.13.

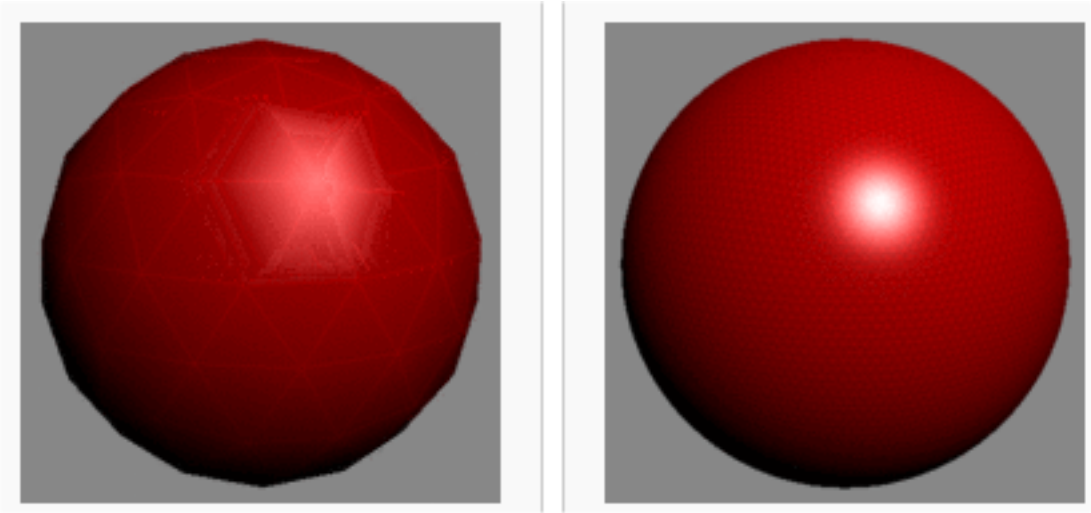


Figure 2.11: Gouraud shading on a sphere with few polygons and on a sphere with a lot of polygons

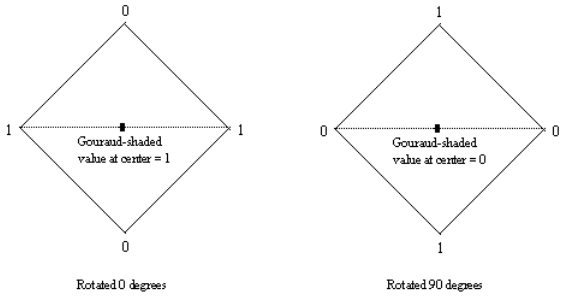


Figure 2.12: Gouraud shading varies with polygons orientation

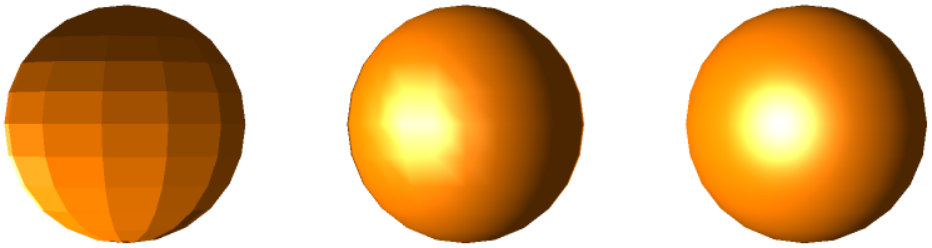


Figure 2.13: Flat vs Gouraud vs Phong

2.7 Lightmap

A lightmap is a data structure, usually encoded in an image, that contains the information about the brightness of a surface in a 3D environment.

Lightmaps are pre-computed and are usually used for static object only, such as open spaces, urban or indoors with long planar surfaces.

2.7.1 Lightmap history

Before lightmaps were invented, real-time computer graphics applications relied purely on Gouraud shading techinque.

In the early '90s John Carmack and Michael Abrash where trying to get rid of some of the Gouraud Shading technique disadvantages to enhance the quality of their video-game Quake.

> *For weeks, we kicked around and rejected various possibilities and continued working with Gouraud shading for lack of a better alternative–until the day John came into work and said, "You know, I have an idea..."*
>
> [1]

Carmack had the idea to split lightning and rasterization in two separate steps. During offline processing a grid (the lightmap) is calculated for each polygon in the world. The lightning is done by casting light from all the nearby lights in the world to each of the grid points on the polygon, and summing the results for each grid point.

Once the grid has been calculated, additional preprocessing can be performed to further improve the quality of the results.

Then, at runtime, the polygon's texture is tiled into a buffer, with each texel lit according to the weighted average intensities of the four nearest light map points, as shown in 2.14.

Quake was the first computer graphic application to use lightmaps to improve rendering quality.

Later on, as hardware capable of pixel-shading became more popular, the technique was abandoned in place of screen-space combination of lightmaps in rendering hardware.

2.8 Shadowmap

The shadowmap technique is quite simple. We render the scene from a camera at the light source and store the resulting depth buffer, which is called the *shadow buffer*. If there are multiple light sources, we render the scene

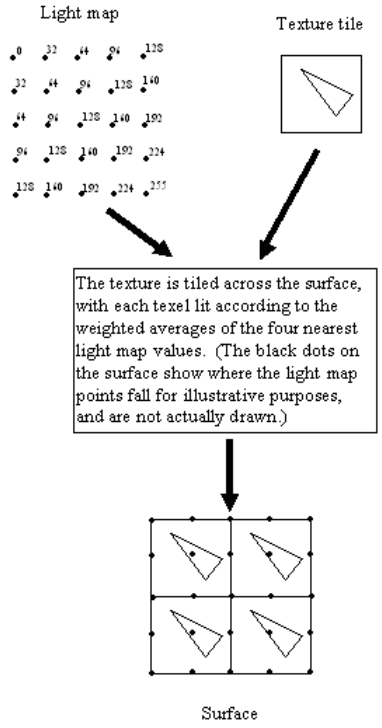


Figure 2.14: A surface is built by tiling the texture and lighting the texels from the lightmap

from the location of each on and store its depth buffer. Note that since we are only interested in the depth buffer, we can do each rendering without colours, lighting, or textures. We then render the scene normally from the location of the camera. For each fragment, we transform its depth to light space and compare that distance to the corresponding location in the depth buffer(s). If it is less than the distances in any of the depth buffers, we use the computed colour, otherwise we use the shadow colour.
The main problem with shadow mapping is aliasing. The shadowmap is limited by the resolution of the depth buffer, and the transformation between light coordinates and object coordinates can highlight the issues of the visual effects from this limited resolution.

Chapter 3

Problem

In this section the problem is described along with the various technique that will be used to solve it.

3.1 Dynamic Lightmap

The main constrain in the use of lightmaps is the fact that, being precomputed, they can not reflect the effect of an environment illuminated by dynamic lights.

Modern computer graphic applications are usually enriched with dynamic lightning for indoor environment and several open world games implement a twenty-four hours illumination cycle that even if it can be exploited by sampling six or seven different states of the daylight the player will still perceive the switch from a state to the next one.

This characteristics lead the computer graphic industry to abandon the lightmap solution in place of heavier but less binding shadowing techniques such as *shadow buffer*.

3.1.1 Dynamic but still static

In this thesis we aim to allow lightmaps to works in presence of cyclic dynamic lights. In some cases, like the day-night cycle, it is possible to sample a small set of illumination states to switch between the hours but that this approach can lead to the feel the switch if the number of states is not high enough, in addition if the light has a fast cycle with shadows moving far and fast it's very easy to spot the trick. In order to get rid of this problem we could use smart sampling, taking more samples when the shadows movements are fast and fewer when they're slow. But if we have several shadows moving at different speed in different moments of the cycle then we have to

take again an enormous amount of sample.
Let's suppose that we need to compute 1024 light states to reconstruct the original light movements with almost the same quality of a shadow mapping technique. Without using any partitioning technique and without streaming the data, if we are looking at a Full HD scene (1920x1080 pixel) we will have to store roughly eight gigabyte of textures just to reproduce the lighting of a single frame of the scene. This approach would be completely unworthy because all the performance gain we may have are brutally countered by the enormous amount of memory that the application will require to be installed.

If the linear interpolation is not a viable solution we need to find a way that allow us to reconstruct the light in the scene using a reasonable amount of samples or a way to reduce the space occupied by those samples.

3.1.1.1 The frequency domain

A possible solution to reduce the number of samples required can be to move the problem in the frequency domain. For the moment let's consider a more specific case; from now on we will focus on a scene where we have one light source called "sun" that moves following the day-night cycle and a simple plain floor with one big cube at the centre and one small cube for each cardinal point 3.1.

We will model the sun as a *Directional Light* since it is distant enough to consider its light-rays always perpendicular and with constant intensity. We will move the sun in circle around the scene, rotating it on the z axe.

On the floor surface we will see five shadows cast by the five entities on it. Each shadow will move accordingly to the sun position so each point of the floor surface will have for each object a time-span in which it receive its shadow. This behaviour can be modelled treating the shadow on each point as a signal, the signal is a function in the time domain and each entity produce a signal over all the surfaces of the scene.

$$\begin{cases} l_t = \overrightarrow{OL(t)} \\ P = (x, y) \\ c(P) = \sum_{entity} f(entity, P, t) \\ f(entity, P, t) = 0 \Rightarrow \overrightarrow{PL(t)} \cap entity, 1 otherwise \end{cases} \tag{3.1}$$

Note that this is a very simple case in which we are not considering ambient light, diffusion and other kind of reflection that can soften the shadow.

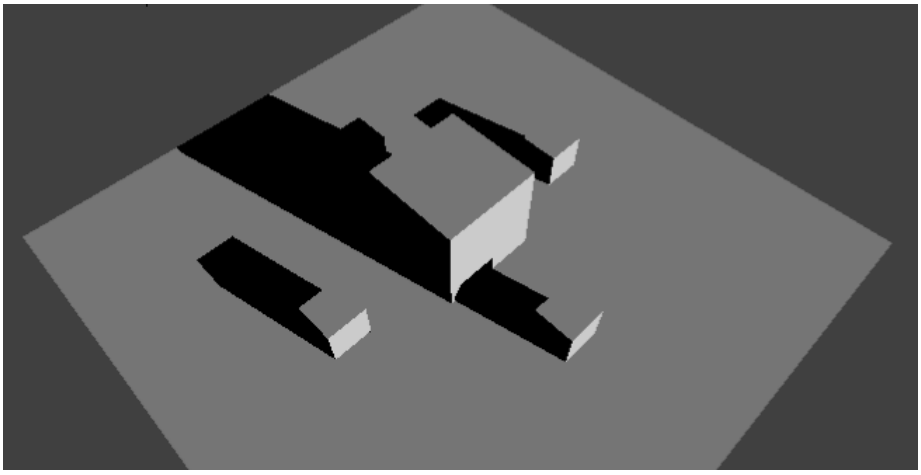


Figure 3.1: A cube surrounded by 4 other small cubes

Entity are made of polygons and we can further improve the formulation saying

$$f(polygon, P, t) = 0 \Rightarrow \overrightarrow{PL(t)} \cap polygon, 1otherwise. \tag{3.2}$$

Given that, we have to discover how many times, during the cycle of the light source, the light position is such that P can't be seen from its perspective, and we will call this value N. Once we know this information we can calculate for the pair pixel-polygon a function that tells us if the polygon covers the pixel from the sun at any given time. Thanks to Nyquist–Shannon sampling theorem [5] [3] we know exactly how many samples we need to recreate that function, and the number is equals to $2N + 1$.

With this system we reduce by two or three order of magnitude the number of samples to take if we want to know the effect of a polygon on the scene from a shadow point of view, the problem is that we still have a lot of polygons to consider and we must consider them all to recreate the original scene!

3.1.2 Simplifying

Scenes in modern video-games are composed of millions of polygons but even if they were just thousands the solution was still impracticable.
But do we really need to consider the outcome of the shadow for each polygon ?
The answer is simple and is NO. If we consider the average model of an

object we will see that each polygon directly affect only few pixel of the scene, it could also affect none of them! In addition, in general, depending on the complexity of a scene each pixel change its shadow value only from 1 to 10 times per light source. The worst case scenario is when we have a grid in front of the light source because the grid shadow will not appear like a grid but more like a blurry homogeneous shadow on the surface.
The only problems with this solution are that the reverse Fourier transform is computationally heavy and that in presence of several light sources we may still need to create many samples.

3.2 Transforms and compressions

Up 'till now we've found that moving into the frequency domain could bring consistent results in order to reduce the number of samples to have a smooth transition between two states. A perfect candidate for this task is the DCT, already used by several compression algorithms (JPEG, MPEG, DV, Daala).

3.2.0.1 DCT

The Discrete Cosine Transform [2] [7] is often used in signal and image processing. The DCT turns sequences of real points into a summation of cosine functions with specific coefficients and arguments.
The DCT appear in several variants but the most commons are the DCT-II and the DCT-III which is the inverse of the DCT-II.

The DCT is a linear and invertible function: $f : \mathbb{R}^N \to \mathbb{R}^N$.

The DCT-II, or simply the DCT, is, as we said, the most common version and is exactly equivalent to a Discrete Fourier Transform of $4N$ real inputs were even-indexed elements (the sine coefficients) are zero.

DCT-II

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] k = 0, \ldots, N-1 \tag{3.3}$$

It is also possible to make the function orthogonal, breaking the correspondence with the DFT, by multiplying the X_0 term by $\frac{1}{\sqrt{2}}$ and the results by $\sqrt{\frac{2}{N}}$.
The DCT-III, or inverse DCT, share the same consideration about the correspondence with the DFT and the orthogonality that we've made for the DCT-II.

DCT-III

$$X_k = \frac{1}{2}x_0 + \sum_{n=1}^{N-1} x_n \cos\left[\frac{\pi}{N} n \left(k + \frac{1}{2}\right)\right] k = 0, \ldots, N-1 \qquad (3.4)$$

3.2.1 Quantization

DCT compression produce as a result an array of coefficient that are not discrete, in fact the DCT is defined as a function from $\mathbb{Z}$ to $\mathbb{R}$.
Since we aim to store these coefficients into another image we have to reduce the output not only to an integer number but to an integer between 0 and 255. To achieve this we have to define a *quantizer* (or quantization function). A quantizer is a function q such that, given a scalar function g with values in range $g_{min} \leq g \leq g_{max}$, if $g_i \leq g \leq g_{i+1}$,

$$q(g) = q_i. \qquad (3.5)$$

Thus, for each value of g, we assign it one of k values. In general, designing a quantizer involves choosing the $\{q_i\}$, quantization levels, and the $\{g_i\}$, the threshold values. If we know the probability distribution for g, $p(g)$, we can solve for the values that minimize the mean square error:

$$e = \int (g - q(g))^2 p(g) dg. \qquad (3.6)$$

A simple rule of thumb is that we should not be able to detect one-level changes, but should be able to detect all two-level changes. Given the threshold for the visual system to detect a change in luminance, we usually need at least 7 or 8 bits.

Since we are using the same DCT that is also used in the JPEG compression we are going to use the same quantization table defined for it.

$$\begin{bmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{bmatrix} \qquad (3.7)$$

We use a 1D DCT instead of a 2D so we will apply them in "zig-zag order" (52, 55, 63, 62, 59, 61, 66 ...). To use the JPEG quantization table

we just divide the $F(x)$ by the value and then we use the inverse function multiplying the argument by the same value $F^{-1}(x * Q)$.

Chapter 4

Solution

In this chapter we will describe the tools we used and the steps we followed in order to gather the required data and solve the problem.

4.1 Creating the scene

The scene has been created using the program Blender available at https://www.blender.org/.

We built up a simple scene with a floor, two walls and a cube as you can see in 4.1 everithing illuminated by a Lamp acting as Directional Light with the properties visible in 4.2.

We used for the first iteration the following Phyton script to create 128 different lightmaps of the scene on an image 1024x1024 in BMP format. At each cycle the lamp light direction is rotated by a one hundred-twenty-eighth of a plain angle.

```
for i in range(0,128):
    rot = ((64 - i)/64) * math.pi / 2
    bpy.data.objects['Lamp'].rotation_euler[0] = rot
    bpy.ops.object.bake_image()
    image = bpy.data.images["Untitled"]
    image.file_format = 'BMP'
    filepath = "%Path%\\LM%d.bmp" % (i)
    image.save_render(filepath)
```

4.3 shows one of the backed lightmaps.

Once collected all the lightmaps in BMP format we developed some tools to perform the direct and reverse DCT and to operate with the BMP format.

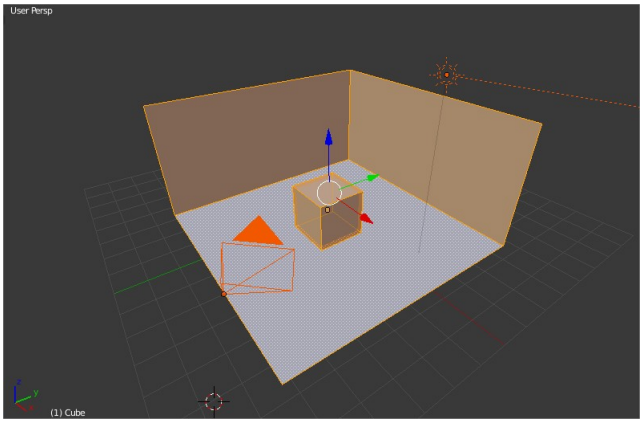


Figure 4.1: Scene created with Blender

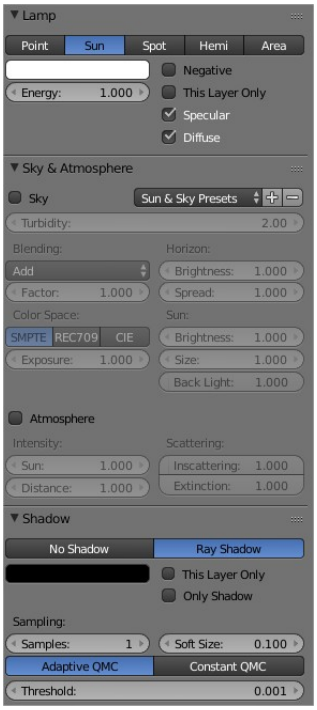
figureI0pt 

Figure 4.2: Lamp Settings

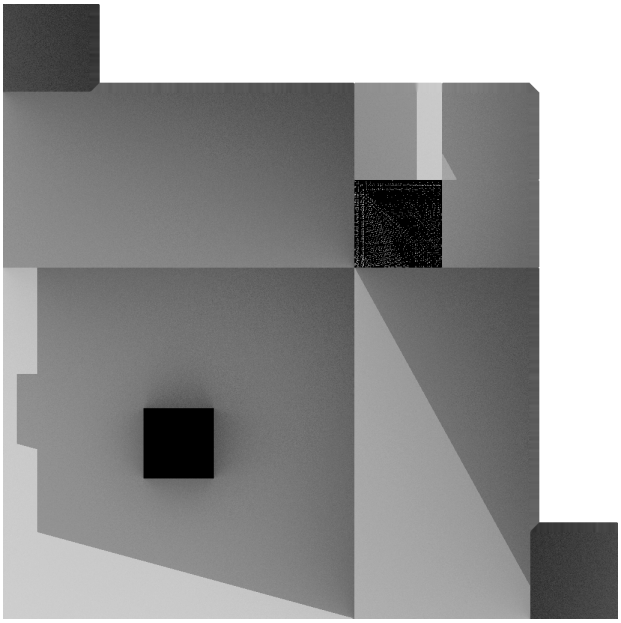


Figure 4.3: Lightmap number 108

4.2 BMP file format

The BMP (bitmap) file format is used to store bitmap digital images on Microsoft Windows and OS/2.
BMP is capable of storing two-dimensional digital images of any width, height and resolution and with various colour depths, including monochrome. We used the simplest implementation of the format without any non-mandatory parameter. Here is a brief description of the content of the header of a bitmap file:

- Bitmap file header: 14 bytes in which there are general information about the bitmap image file.
  - 00: two characters that identify the BMP and DIB, BM on Windows.
  - 02: four bytes in which there is the size in bytes of the BMP file.
  - 0A: four bytes that indicates the offset at which the image data starts.
- DIB header: Fixed-size but depends on the version (there are 8 different versions) and is used to store detailed information about the image and the pixel format.

  - 0E: size of the DIB
  - 12: image width in pixels
  - 16: image height in pixels
  - 1A: number of colour planes which must always be 1
  - 1C: number of bits per pixels
  - 1E: compression method used
  - 22: image size
  - 2E: number of colours in the palette

- Pixel array: The actual image, each row in the pixel array is padded to a multiple of 4 bytes in size.

In the pixel array the first byte correspond to the bottom right corner and the last to the top left corner and the colours are store in the BGR order (opposite of RGB).

4.3 Linear interpolation

The first experiment was meant to use the 128 samples and interpolate them to obtain the shadow position at each frame. The fragment shader is quite simple:

```
layout(location = 0) out vec3 out_color;

in vec2 UV;

uniform sampler2D sample1;
uniform sampler2D sample2;

uniform float timeZone;

void main(void)
{
  vec3 l1 = (1-timeZone) *
                    texture2D( sample1, UV, -2.0 ).rgb;
  vec3 l2 = timeZone * texture2D( sample1, UV).rgb;
  out_color = l1 + l2;
}
```

the previous and the next sample are sent to the fragment shader along with the current *timeZone* calculated as:

$$timeZone = \frac{currentTime \bmod K_s}{K_s} \tag{4.1}$$

where K_s is the amount of milliseconds between one sample and another.

4.4 Calculating DCT

In order to calculate the Discrete Cosine Transform we've created a library with some utility functions:

- CalculateNTransform(in Samples, in numberOfArmonics, out Transforms, in sizeOfSamples): Given a list of images already converted in pixel array (Samples) and the number of coefficients that we want to produce (numberOfArmonics) the function return a list of transformed images (Transforms).
- CalculateInverseTransform(in Transforms, out Samples, in sizeOfTransforms): Given a list of transformed images the function recreate the original array of samples.
- CalculateInverseTransofrmAtIndex(in Transforms, in index, in sizeOfTransforms) out Sample: Given a list of transformed images and an index, the function return the index-th original sample.

The computation of the DCT even if it's not as time consuming as the Fourier Transform it's still quite heavy to compute so we adopted some tricks to speed up the process.

First of all let's see the correct way to calculate the DCT.

```
for (int k = 0; k < _iHarmonicsNumber; k++)
{
  byte* dct = new byte[_iSize];

  //for each point, calculate the F(k)
  for (int x = 0; x < _iSize; x++)
  {
    float dctTmp = sqrtf(2.0f / samples.Size()) * c(k);
    float sum = 0;
```

```
    for (int j = 0; j < samples.Size(); j++)
    {
      //We remove 128 because of the conversion from
          signed to unsigned.
      sum += (_vSamples.Get(j)[x] - 128) * cos((2 * j +
          1) * k * PI / (2 * samples.Size()));
    }

    dctTmp *= sum;

    //Q is the array of coefficient to encode the
        output values between 0 and 255.
    dct[x] = dctTmp / Q[k];
  }

  _vTransform.push(dct);
}
```

From this function we see some possible optimization:

- Moving the variable declaration outside the loops.
- The first square root is constant and depends on the number of samples that we have in input. Hence it can be turned into a constant value and move it outside the loops.
- The cosine depends on the j-k iteration and on the number of samples, so it's useless to calculate it again inside the x loop. We can create a precomputed table as we know the size of the samples array and the number of harmonics.

Another possible optimization could be to rearrange the cycles to have a straight access to the samples array in order to speed up reading operations, unluckily if we rearrange the cycles we end up accessing not-in-order the transforms array so the two possible sequences of loops are equivalent.

The final result looks like this:

```
UINT8* dct;
const float sqrtTwoOverSize = sqrtf(2.f / samples.Size
    ());
float sum = 0;

for (int k = 0; k < _iHarmonicsNumber; k++)
```

```
{
  dct = new UINT8[_iSize];

  //for each point, calculate the F(k)
  for (int x = 0; x < _iSize; x++)
  {
    sum = 0;

    for (int j = 0; j < samples.Size(); j++)
    {
      sum += (_vSamples[j][x] - 128.f) * _CosTable[k][j
          ];
    }

    dct[x] = sum * sqrtTwoOverSize * c(k) / Q[k];
  }

  _vTransform.push(dct);
}
```

The resulting image of the DCT coefficient is like the 4.4.

Once we have the list of transformed images we need to define a function to get back the desired sample. Using the IDCT and performing the same type of code optimization, the final code is:

```
UINT8* idct = new UINT8[_iSize];
const float sqrtTwoOverSize = sqrtf(2.f / _vSamples.
    Size());
float sum = 0;

for (int x = 0; x < _iSize; x++)
{
  sum = 0;
  for (int k = 0; k < _vTransform.size(); k++)
  {
    sum +=  _vTransform[k][x] * _CosTable[k][_iIndex];
  }

  //Again we add 128 for the signed-unsigned conversion
  idct[x] = sum * sqrtTwoOverSize * c(k) * Q[k] + 128;
}
```

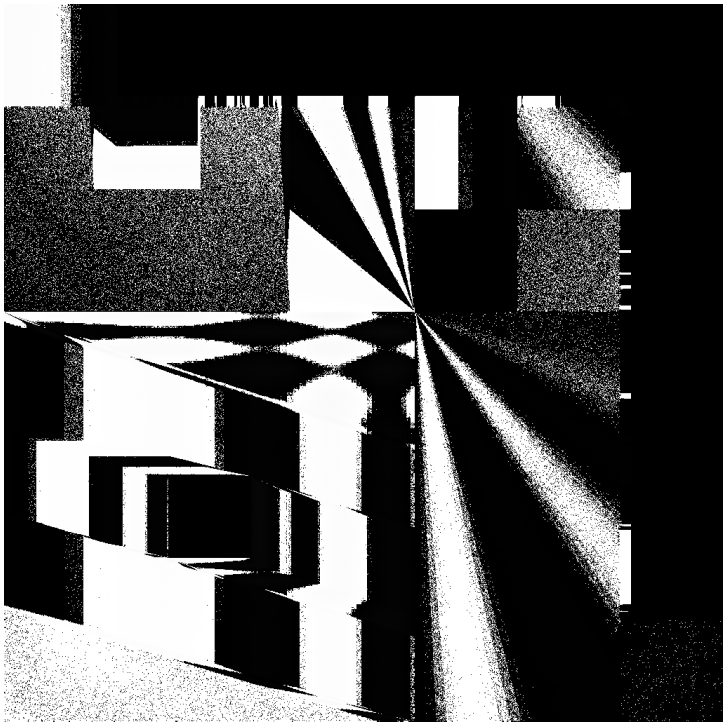


Figure 4.4: Image containing the 12th coefficient of the DCT compression.

```
return idct;
```

There is one last consideration to do; in the previous IDCT code we have used the cosine table as for the DCT, but if we are interested in the status that is in between two samples, for instance the moment between the fourth and the fifth sample, we could avoid to use the cosine table and calculate the cosine like in the first example, in this way index doesn't have to be integer and we can access to the sample number 4.3.

4.5 Precalculated try

Once we have the list of coefficients of the DCT encapsulated in a set of transformed lightmaps, which is much smaller than the original set of samples, we are ready to run the first test.
We started from the simplest scenario and where we were trying to expand the DCT to get the two closest samples to the current time and then interpolate them to obtain the final result. Since the samples passed trough the DCT compression the interpolation of the reconstructed images should

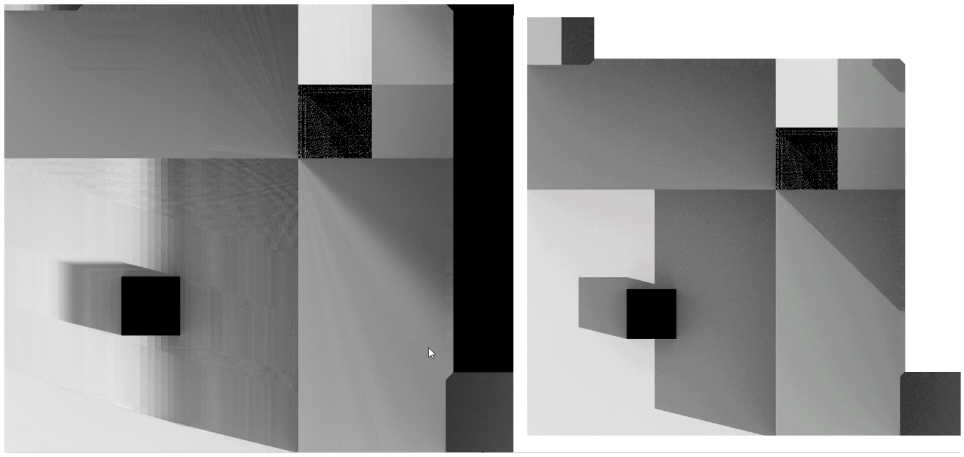


Figure 4.5: On the left the sample after the IDCT, on the right the original sample

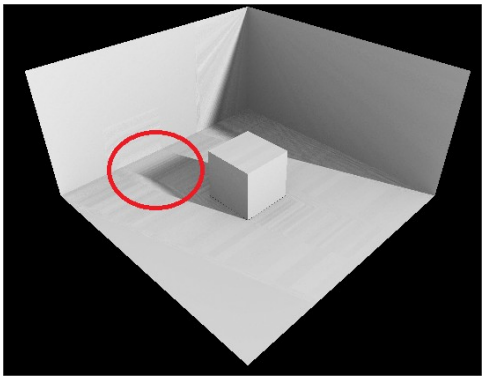


Figure 4.6: In the red circle is visible the blending of the linear interpolation.

blend better with a linear interpolation.

As can be seen in the 4.5, the reconstructed images has a sort of insight of where the shadow is going to be and this improve the interpolation results.

The 4.6 shows the final results of this first try. We can see how the surfaces on which the illumination blend from bright to dark behave exactly like in the linear interpolation scenario, so we didn't have a loss of quality on that side. The pixels on which the shadow move upon instead are better blended during the interpolation but because of some approximations and some data loss during the encoding process of the coefficients we end up having some weird wavy pattern.

To mitigate this effect we've added a convolution function and we've blended the result on a sigmoid. We've used an uniform kernel 5x5 for the

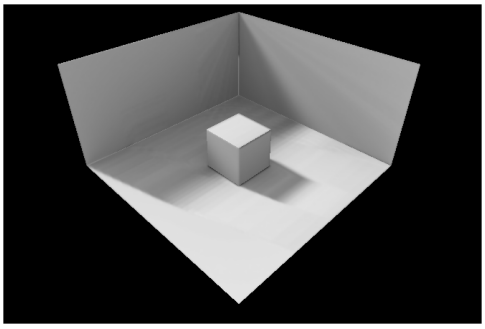


Figure 4.7: The wavy pattern is mitigated by the convolution

convolution with all the coefficients at $\frac{1}{25}$ and a variable sigmoid-like function of equation (4.2) with $\lambda = 0.35$ which should increase the contrast at noon and reduce it at midnight. You can see the results in 4.7.

$$\begin{cases} f(x,t) = a(t)x^3 + b(t)x^2 + c(t)x \\ k(t) = \frac{(1+\lambda)}{4} - \lambda \left| \frac{t}{12} - 1 \right| \\ a(t) = \frac{(1-4k(t))}{k(t)(1-k(t))(1-2k(t))} \\ b(t) = \frac{3a(t)}{2} \\ c(t) = \frac{a(t)}{2} + 1 \end{cases} \tag{4.2}$$

4.6 Runtime try

For this test we've tried to expand the IDCT directly inside the fragment shader. This method allows to calculate only the necessary pixels instead of the whole lightmap each frame. We've also decided to calculate directly the cosine instead of using the cosine table in order to be able to extract each frame in between two samples.

The fragment shader used is the following:

```
void main()
{
  vec2 bmpUV = vec2(UV.x, 1 - UV.y);

  int sample1 = int(Timespan) / 500;
  int sample2 = (sample1 + 1) \% 128;

  float timeFraction = (Timespan - 500 * sample1) /
     500.0;
```

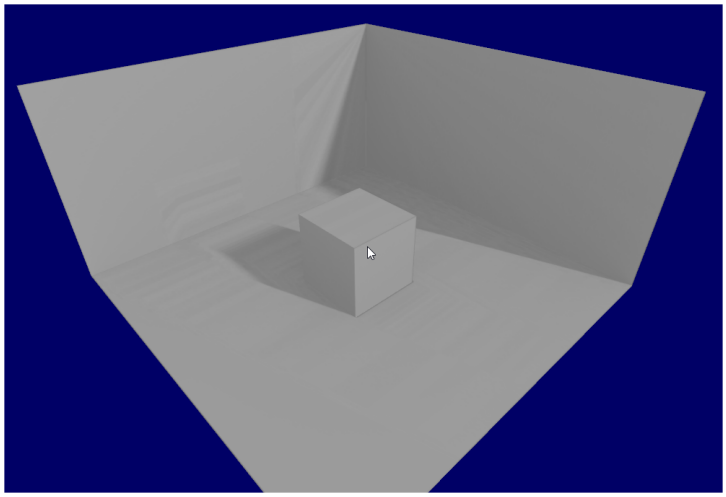


Figure 4.8: Result of the shader expansion of the IDCT

```
  float index = 1-timeFraction) * sample1 +
     timeFraction * sample2;

  float tScaloid = (Timespan / 64000.0) * 12.0 + 6.0;

  vec3 sum = vec3(0.0);
  float dct = 0.f;

  for (int k = 0; k < 32; k++)
  {
    dct = c(k) * cos((2.0 * (index + 1.0) * k * 3.1415
       / (2.0 * 128.0));
    dct *= texture2DArray( Lightmaps, vec3(bmpUV +
       offset5[i], k)).rgb - vec3(0.5, 0.5, 0.5);
    dct *= float(K_Q[k]);
    sum += tmp;
  }

  color = sum;
}
```

We've also added, as before the convolution and the sigmoidal function to clean up the results 4.8.

4.7 Shadowmap

We've implemented shadow mapping using OpenGL as follows:

4.7.1 First implementation

The shadow map was calculated using:

```
layout(location = 0) in vec3 vertexPosition_modelspace;

// The MVP from the light source point of view.
uniform mat4 depthMVP;

void main(){
 gl_Position =  depthMVP * vec4(
    vertexPosition_modelspace,1);
}
```

```
layout(location = 0) out float fragmentdepth;

void main(){
    // Not really needed, OpenGL does it anyway
    fragmentdepth = gl_FragCoord.z;
}
```

the resulting image was used by a simple vertex shader:

```
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec3 vertexNormal_modelspace;

out vec3 Normal_cameraspace;
out vec3 LightDirection_cameraspace;
out vec4 ShadowCoord;

uniform mat4 MVP;
uniform mat4 V;
uniform mat4 M;
uniform vec3 LightInvDirection_worldspace;
uniform mat4 DepthBiasMVP;

void main()
{
```

```
    // Output position of the vertex: MVP * position
    gl_Position =  MVP * vec4(vertexPosition, 1);

    // Same, but with the light's view matrix
    ShadowCoord = DepthBiasMVP * vec4(vertexPosition, 1);

    // Vector that goes from the vertex to the light, in
       camera space
    LightDirection_cameraspace = (V*vec4(
       LightInvDirection_worldspace,0)).xyz;

    // Normal of the the vertex, in camera space
    Normal_cameraspace = ( V * M * vec4(
       vertexNormal_modelspace,0)).xyz;
}
```

which was simply sending to the fragment shader the position of the vertex as seen from the light (ShadowCoord) and two vectors used to determine if a surface is facing the light source. The fragment shader was also quite simple:

```
vec3 MaterialDiffuseColor = texture2D( myTextureSampler
    , UV ).rgb;
vec3 n = normalize( Normal_cameraspace );
vec3 l = normalize( LightDirection_cameraspace );
float cosTheta = clamp( dot( n,l ), 0,1 );
float visibility = 1.0;
if ( texture( shadowMap, vec3(ShadowCoord.xy,
    ShadowCoord.z / ShadowCoord.w) ) < ShadowCoord.z){
     visibility = 0.5;
}
color = visibility * MaterialDiffuseColor * cosTheta;
```

4.7.2 Shadow acne

This implementation had an issue called "Shadow acne" 4.9.

This phenomenon is easier to understand using the image 4.10. To solve this issue is enough to add a small bias to the visibility check:

```
float bias = 0.005;
float visibility = 1.0;
```

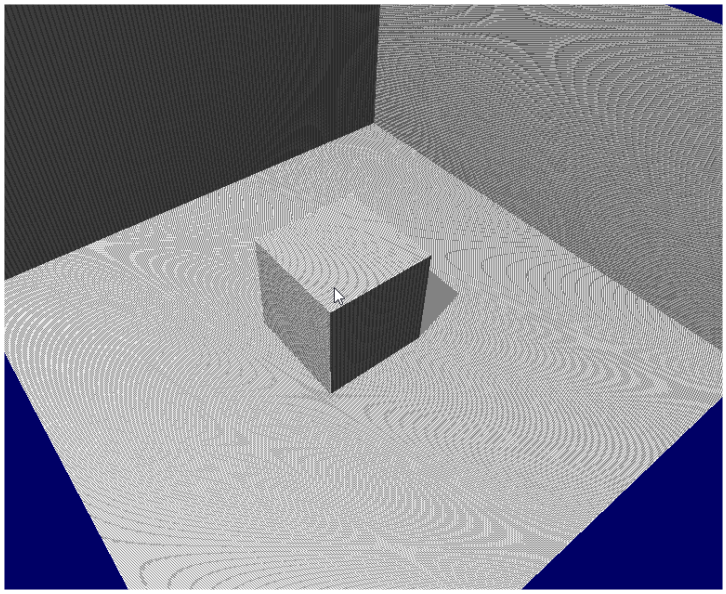


Figure 4.9: Shadow acne

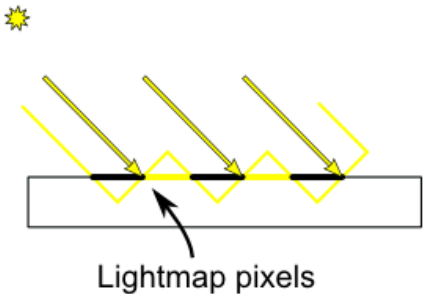


Figure 4.10: Cause of the shadow acne

```
if ( texture( shadowMap, vec3(ShadowCoord.xy, (
    ShadowCoord.z - bias) / ShadowCoord.w) ) <
    ShadowCoord.z-bias){
     visibility = 0.5;
}
```

4.7.3 Peter panning

No we don't have the shadow acne any more but we can still notice another unpleasant phenomenon called "Peter Panning" 4.11 that introduce a gap between the an object and its shadow making the cube to look as if it's flying.

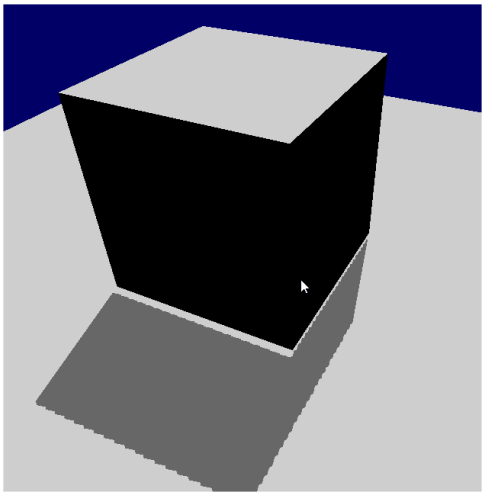


Figure 4.11: Peter Panning

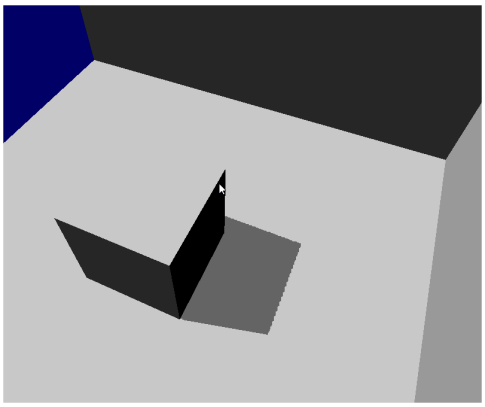


Figure 4.12: Final result of the shadowmap

We have 2 ways to solve this issue:

- Change the CullFace from GL_FRONT to GL_BACK and be sure that there are no polygon not facing the light that are not "occluded" by another polygon.
- Avoid thin geometry which automatically grant the first condition.

Since all the dynamic lightamp have been calculated using that model we will use the first approach even if the second is a more solid solution.

In the 4.12 you can see the final result.

4.8 Performance analysis

The performance analysis are done by recording the drawcall time of the scene shown before with the cube and the 3 walls starting far from the object and getting closer in order to increase the number of pixels to compute.

4.8.1 CPU IDCT expansion performances

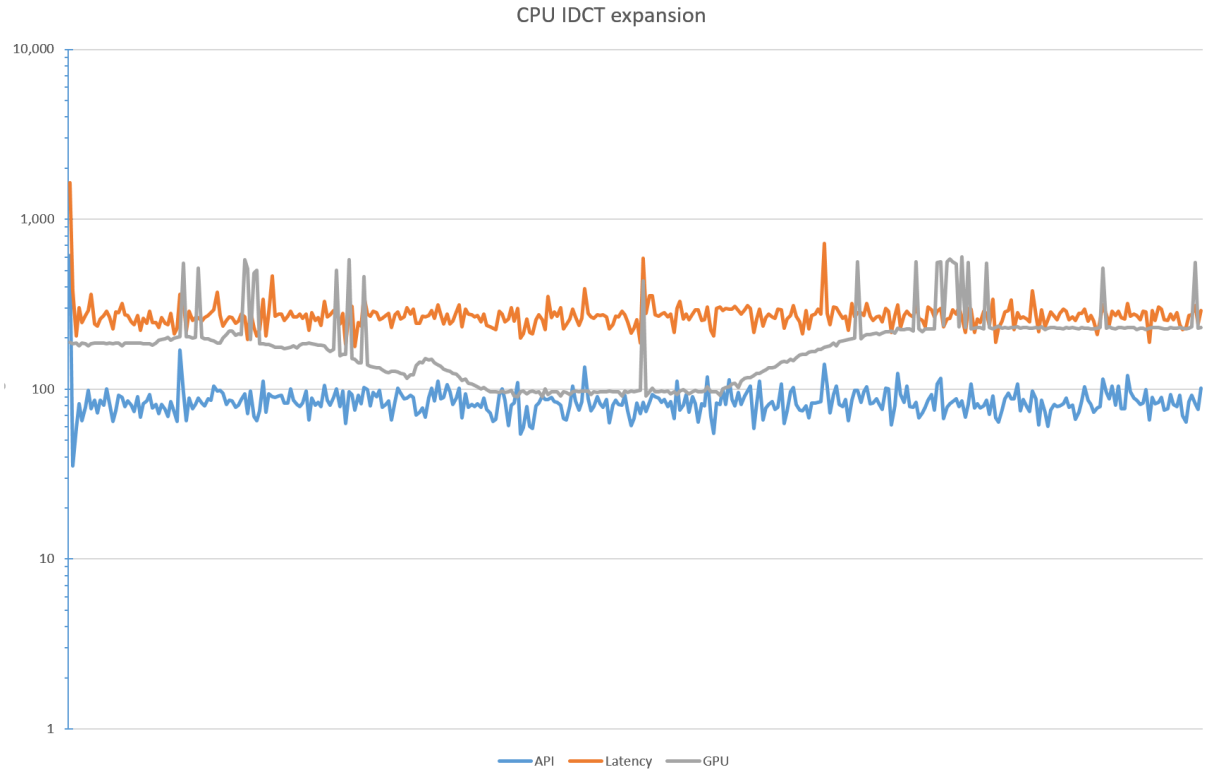


Figure 4.13: Drawcall time when the expansion is performed CPU side

In figure 4.13 we can see that the computational effort in terms of microsecond to render the frame using the dynamic lightmap system expanding the DCT with the CPU is around 200 microseconds. This computation is relatively fast considering that is not affected by the number of polygons but by the number of pixels hence that the complexity of the scene doesn't affect the performance.

4.8.2 Shader IDCT expansion performances

In figure 4.14 we can see that expanding the lightmap at runtime inside the GPU is incredibly expensive since it may take up to 10 milliseconds on

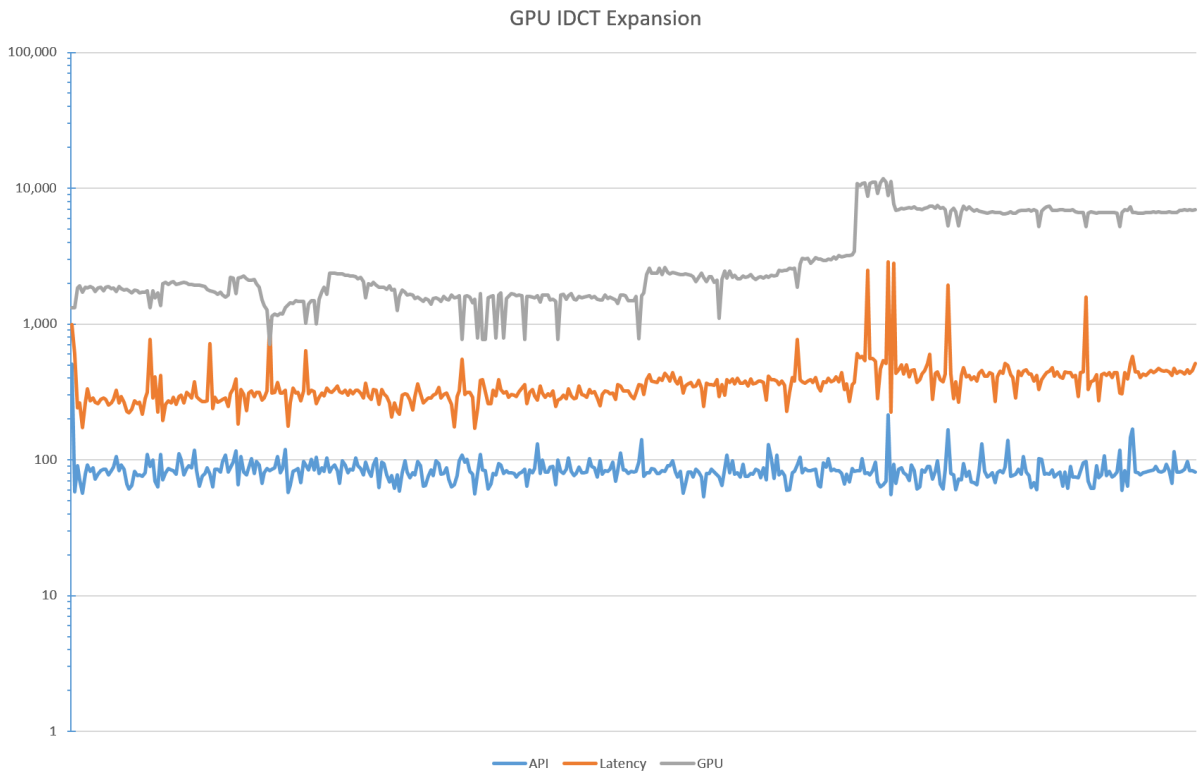


Figure 4.14: Drawcall time when the expansion is performed GPU side

GPU side, 15 times more than the CPU expansion, if we add this to the big amount of memory required to load all the lightmap of coefficient this method appears to be far too inefficient.

4.8.3 Shadowmap performances

The performances of the shadowmap, as show in 4.15, are affected, given the simple scene, mainly by the latence of executing two shader program doubling the time required by the latency and the API. The time spent by the GPU is quite low but it rumps up as the number of pixels in the scene increase reaching the time required by the CPU expansion.

4.8.4 Comparison

The first image 4.16 shows the time required by the CPU only to complete the computation of the frame.
We can see that the GPU expansion is far too expensive and that the shadowmap appears to be faster than the CPU expansion. Even though we must

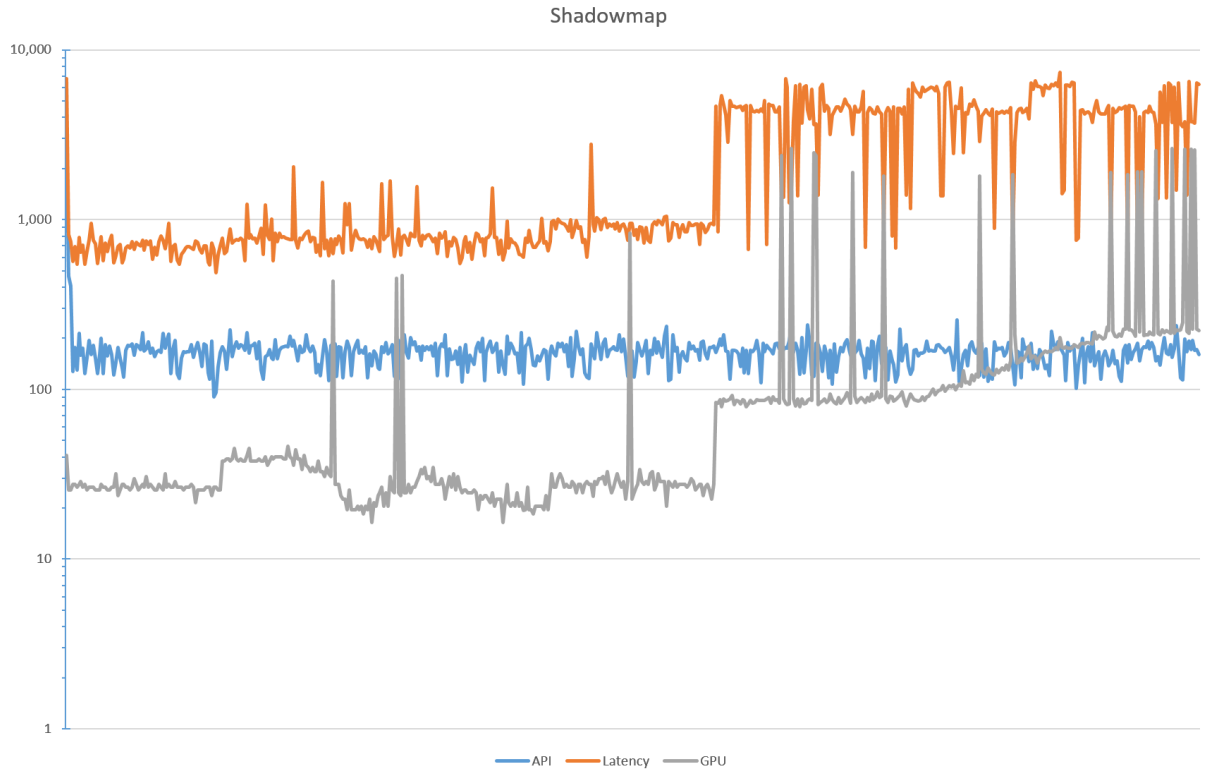


Figure 4.15: Drawcall time using shadowmaps

keep in consideration that the time required by the shadowmap increase also with the number of polygons of the scene and that this is not true for the CPU expansion scenario.

In addition, even for simple scenes, when we look at the overall drawcall time 4.17 we notice that, due to the double latency introduced by the two shader program, the performances of the CPU expansion perform better than the shadowmap technique; the GPU expansion remains out of scale.

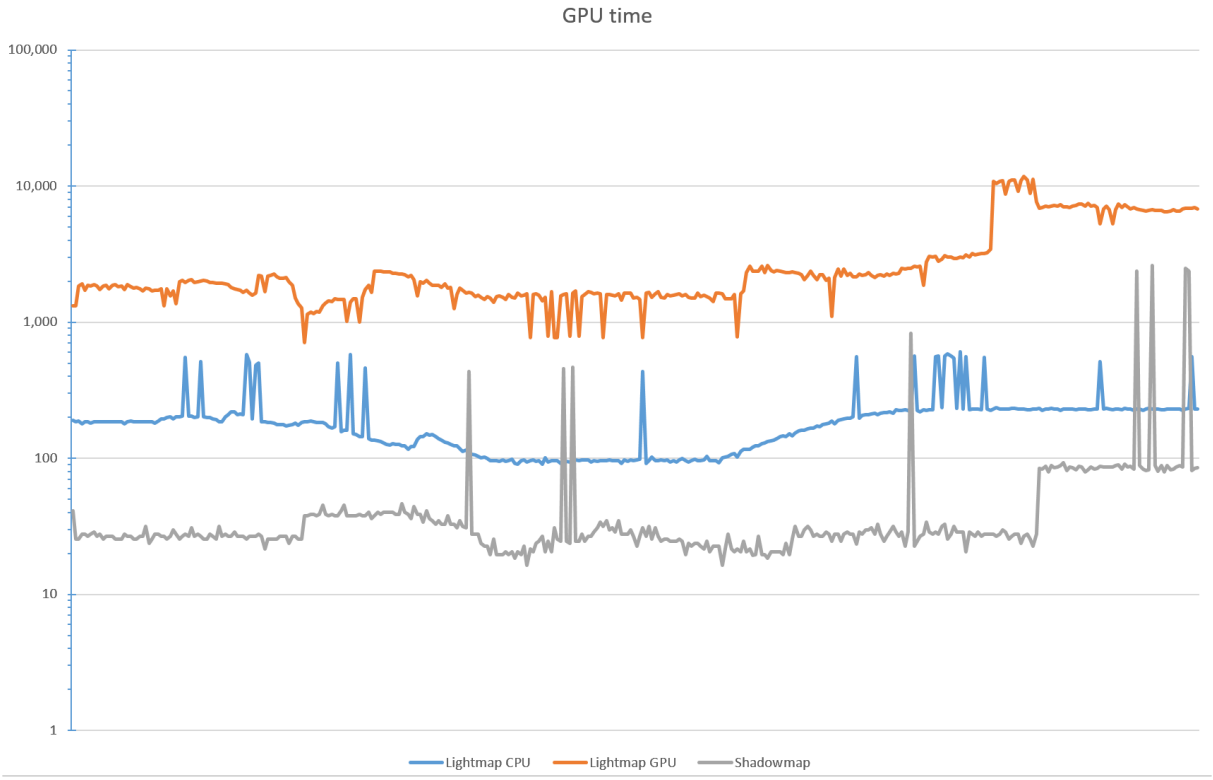


Figure 4.16: Time required to complete the execution of the shader program

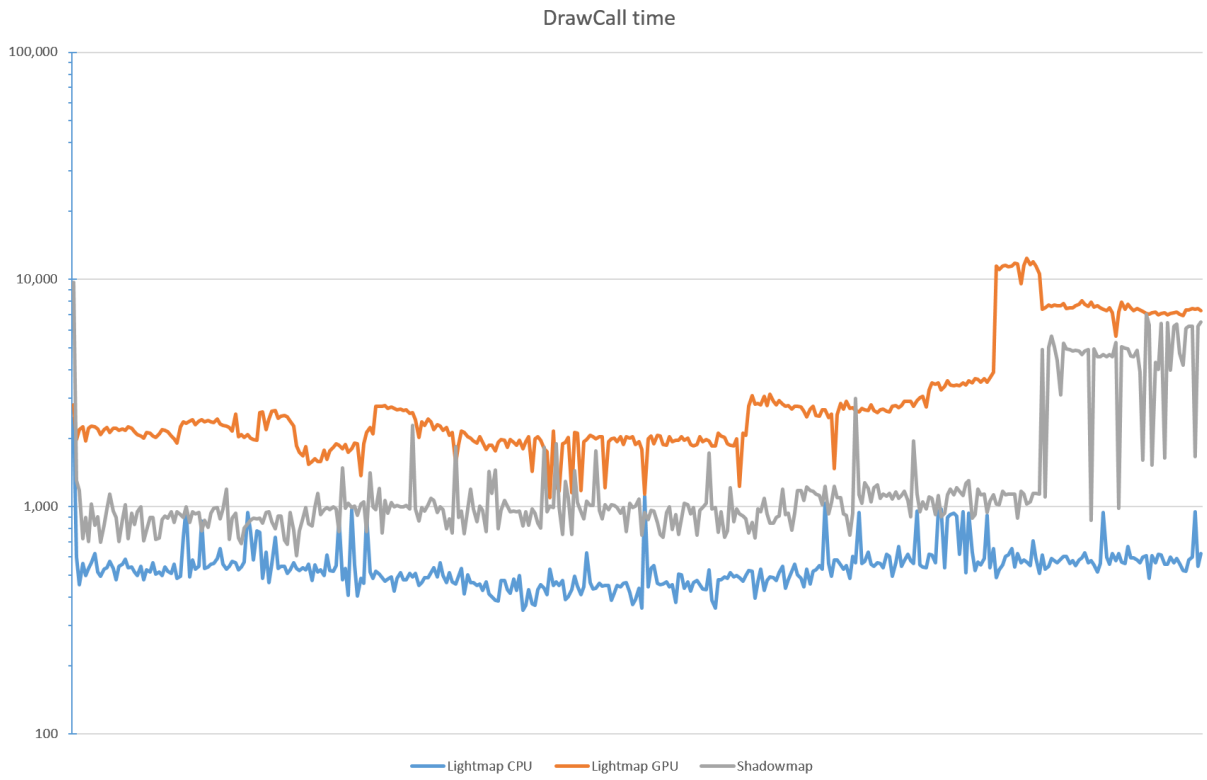


Figure 4.17: Time required to complete the execution of the draw call

Chapter 5

Conclusions

In this section we will describe the obtained result and will explain what could be the subjects of further researches.

5.1 Conclusions

We've proven that is possible to store the light distribution among a scene from a moving light source with a cyclic path in a small finite amount of processed samples preserving the feeling of continuity between one sample and another.
Even if in the test scene the Shadowmap technique performed better, we've demonstrated that increasing the number of polygons in the scene the difference in milliseconds between two frames with the two approaches get thinner. Shadowmap still provide results that are more precise than the Dynamic Lightmap method if we are looking for ultra-realism but the smoothness of the Dynamic Lightmap can be associated to a blurry environment which is usually quite computationally expensive to reproduce.
Dynamic Lightmap is not going to replace the Shadowmap technique but is a new viable solution to reproduce the light in specific environment better than any other technique. It also gives the possibility to pre-process and bake many information that are impossible to calculate using runtime techniques without a considerable impact on the overall performances of the program.

5.2 Further researches

Dynamic Lightmap has still several improvement that can be added and open several new field of research.

5.2.1 Frequency partitioning

The first quality improvement is to generate multiple layer of coefficients and each layer add more byte to the coefficient representation in order to produce better results. This other layer can be added to the current scene to improve the results while the CPU usage is low.
For instance, when we calculate the $F(x)$ we usually have a float value that is converted into a byte size unsigned integer. We can put the four bytes of the floating point representation in four different images and load them when we aren't loading any other image to improve the quality of the already calculated lightmaps.

5.2.2 Fire shader

A possible way to use the Dynamic Lightmap approach, other than the classic one, is to create dynamic mask for the light cast by the fire. Baking few different samples from the shape of the fire and generating few coefficients, thanks to the intrinsic blurriness of the Dynamic Lightmaps we should be able to obtain extremely realistic results.

5.2.3 Improve the quantization algorithm

The algorithm used to store a float into a byte is similar to the one used by the JPG compression, using a constant quantization table to reduce the values to numbers between -128 and 127. A good way to improve the results is to define a less lossy algorithm.

5.2.4 Fast DCT

The Fast DCT or FCT (Fast Cosine Transform) is an algorithm used to reduce the complexity of the transformation from $O(N^2)$ to $O(N \log N)$. This will dramatically reduce the computational resources needed to calculate the inverse transform at runtime, allowing to take much more samples and increasing the quality of the final result.

5.2.5 Enhance the DCT results

Since we perform all the computation offline we can try several image processing algorithm to improve the result of both the IDCT and the IDCT interpolations.

Bibliography

[1] Michael Abrash. Quake's lighting model: Surface caching.

[2] Nasir Ahmed, T Natarajan, and Kamisetty R Rao. Discrete cosine transform. *Computers, IEEE Transactions on*, 100(1):90–93, 1974.

[3] Salomon Bochner and Komaravolu Chandrasekharan. *Fourier transforms*. Number 19. Princeton University Press, 1949.

[4] Henri Gouraud. Continuous shading of curved surfaces. *IEEE transactions on computers*, 100(6):623–629, 1971.

[5] Abdul J Jerri. The shannon sampling theorem—its various extensions and applications: A tutorial review. *Proceedings of the IEEE*, 65(11):1565–1596, 1977.

[6] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[7] Arai Yukihiro, Agui Takeshi, and Masayuki Nakajima. A fast dct-sq scheme for images. *IEICE TRANSACTIONS (1976-1990)*, 71(11):1095–1097, 1988.