Visualizing 3D models with fine-grain surface depth

Author:

Roi Méndez Fernández

Supervisors:

Isabel Navazo (UPC)

Roger Hubbold (University of Manchester)

Index

1. Introduction ... 7
2. Main goal ... 9
3. Depth hallucination ... 11
   3.1. Background ... 11
   3.2. Output ... 14
4. Displacement mapping ... 15
   4.1. Background ... 15
      4.1.1. Non-iterative methods ... 17
         a. Bump mapping ... 17
         b. Parallax mapping ... 18
         c. Parallax mapping with offset limiting ... 19
         d. Parallax mapping taking into account the slope ... 19
      4.1.2. Unsafe iterative methods ... 20
         a. Iterative parallax mapping ... 20
         b. Binary search ... 21
         c. Secant method ... 22
      4.1.3. Safe iterative methods ... 22
         a. Linear search ... 22
         b. Other methods ... 23
   4.2. Displacement mapping implementation ... 24
      4.2.1. Relief mapping ... 24
      4.2.2. Interval mapping ... 25
      4.2.3. Relaxed Cone stepping ... 27
   4.3. Results of implementation ... 31
5. Illumination ... 37
   5.1. Background ... 38
   5.2. HDR Images ... 39
      5.2.1. Environment map creation ... 40
      5.2.2. Median cut sampling ... 44
         a. Implementation ... 44
   5.3. Results of implementation ... 52
6. Geometry images ... 55
   6.1. Background ... 55
   6.2. Implementation and use ... 55
   6.3. Results of implementation and use ... 57
   6.4. Conclusions and future work ... 59
7. Acknowledgements ... 61
   References ... 63
   Appendices ... 65

Figures index

Figure 1: Model reconstructed from a set of photographs 7
Figure 2: Model reconstructed from photographs lit with one point light 9
Figure 3: Original photograph from a real Mayan building, derived model relighted with different light conditions and the same model with another illumination....................... 11
Figure 4: Flow chart showing the steps in the process 12
Figure 5: Example of an input photograph pair 12
Figure 6: Example of a profile of a textured surface and the separation between the above-plane and below-plane surface models... 13
Figure 7: Difference between a render using a depth hallucinated map and a render using a laser scanned map.. 13
Figure 8: Original photograph and reconstructed and relighted model.......................... 14
Figure 9: Depth map and computed normal map using the Sobel operator 14
Figure 10: Displacement mapping scheme 15
Figure 11: Displacement mapping process scheme 16
Figure 12: Displacement mapping 16
Figure 13: Bump mapping.. 17
Figure 14: Texture mapping and Bump mapping 17
Figure 15: Parallax mapping... 18
Figure 16: Comparison between bump mapping and parallax mapping 18
Figure 17: Parallax mapping with offset limiting 19
Figure 18: Parallax mapping and Parallax mapping with offset limiting 19
Figure 19: Parallax mapping taking into account the slope 20
Figure 20: Comparison between parallax mapping with offset limiting and parallax mapping taking into account the slope 20
Figure 21: Comparison between parallax mapping with slope and iterative parallax mapping 21
Figure 22: Binary search.. 21
Figure 23: Binary search results 21
Figure 24: Secant method... 22
Figure 25: Linear search... 22
Figure 26: Linear search results 23
Figure 27: Cone stepping.. 23
Figure 28: Interval mapping 25
Figure 29: Comparison between interval mapping and relief mapping with two steps in the interval mapping and binary search step respectively 26
Figure 30: Comparison between conservative and relaxed cones 27
Figure 31: An intermediate step during computation of a cone ratio............................ 28
Figure 32: Some viewing directions and relaxed cone... 28
Figure 33: Cone map and relaxed cone map... 29
Figure 34: Ray-relaxed cone intersection process.. 29
Figure 35: Relief mapping artifacts with 10 linear search steps and relief mapping with 20 linear search steps 32
Figure 37: Interval mapping using 10 linear search steps and interval mapping using 20 search steps. 32
Figure 38: Interval mapping with 10 linear search steps and 2 interval mapping steps 33
Figure 39: Image from a grazing angle with 5 relaxed cone steps, with 10 relaxed cone steps and 20 relaxed cone steps. 33
Figure 40: Relaxed cone mapping with 10 relaxed cone steps and 8 binary search steps 34
Figure 42: Images rendered using interval mapping. Notice the hard shadows and the artifacts in the shadows.. 35

Figure 43: Visualization of a model using interval mapping. Notice the hard shadows when we look perpendicular and the artifacts in the shadows when we look from grazing angles. Notice also how the shadows are not realistic at all as they seen to be points instead of a continuous shadow due to the point light. 35
Figure 41: Shadows rendered with relief mapping 34
Figure 44: Images rendered using relaxed cone mapping 36
Figure 45: Figures rendered using relaxed cone mapping. 36
Figure 46: Parts of a shadow 37
Figure 47: Mirrored ball reflecting a complete bathroom 37
Figure 48: Example of a set of seven images used to create a HDR image 40
Figure 49: HDR image 41
Figure 50: 0º photograph and 90º photograph 41
Figure 51: Light probes of 90º and 0º after warping 42
Figure 53: Light probe after the process. 43
Figure 52: Mask used to blend the images 43
Figure 54: Final environment map prepared to be mapped on a sphere 44
Figure 55: Rectangular area of a summed area table 45
Figure 56: Different results for the median cut sampling algorithm for a given latitud-longitude image 48
Figure 57: Mapping of 128 lights to a sphere 49
Figure 58: Mapping of 1024 lights to a sphere 50
Figure 59: Mapping of 128 lights to a sphere. 50
Figure 60: Original image 51
Figure 61: Processed lights and regions on the image 51
Figure 62: Output image encoding the position and color of the lights 51
Figure 63: Detail of the shadows. 53
Figure 64: Rocks lighted using different light probes taken from Devebecs web and using the one we showed before how we constructed it (the garden of the university). 53
Figure 65: Reconstructed wall lighted using different light probes taken from Devebecs web and using the one we showed before how we constructed it (the garden of the university). ..53
Figure 66: Objects lighted using different environments. 54
Figure 67: Geometry Image, texture from this geometry image and relief map computed from the texture 56
Figure 68: Textured 3D model, triangles taken from the Geometry image and normals of the model used as a texture 56
Figure 69: Comparison between texture mapping and interval mapping of a geometry image. 57
Figure 70: Wall from the University church lighted with a glacier light 58
Figure 71: Wall lit using two different environment maps: University gardens and Ennis 58

1. Introduction

Throughout history, people have tried to preserve their historical and architectural inheritance in many different ways in order to know how a building was at some point in the past or just for the pleasure of viewing these objects and buildings from a different point o view. To record this information they first made paintings, then small physical models, and when photography was discovered they took photos of historically important buildings, just to preserve this information in case a disaster happened (war, natural disasters, etc.).

Computer graphics have provided a new tool to preserve the architectural and cultural inheritance of our buildings by reconstructing them as 3D models. This approach seems to be the best because of its accuracy, the fact that we can travel all around the model, the small physical space we need to store this information (just a hard drive) and the chance it affords for sharing this information with people from all around the world.

There are many ways to create these models. As in the past with paintings, we can have an artist constructing 3D models, but this approach presents several problems such as the low accuracy of the model or the particular view that the artist has of the building. Another way is to use a laser scan all around the building in order to get a point cloud and then store this information to reconstruct the model. The main problem with this approach is that sometimes it's impossible to scan a building because of the huge amount of information that is generated and the difficulty of registering and work with all this information. This method is the most accurate but also the most expensive.

Finally, new approaches have been developed that reconstruct the models from photographs. This is probably the cheapest approach, because we only need a camera and the software and then we can reconstruct any building just by taking some photographs of it. The main problem with this approach is that it is not as accurate as a laser scan.

Figure 1: Model reconstructed from a set of photographs, notice that it is a point cloud model

The University of Manchester has developed a software system called *Daedalus* that implements this approach. It's able to reconstruct 3D geometry from multiple photographs using a structure from motion (or multiview geometry) approach. Fine surface detail is added to the reconstructed geometry using a shape from shading method.

Once we have created a model using one of these approaches we must visualize it on the computer screen. The main problem with this visualization is that usually the models we get from the laser scan or the photographs are very dense because we get very fine-grain surface detail (there are a few millimeters from one point to the next one). As we have many points we will have many polygons when we reconstruct the model from the point cloud. The more polygons we have the slower the visualization will be. If we want to interact with the model (move it, zoom, etc.) we need to have real time (or at least interactive) frame rates. If we have millions of polygons and we want to display them on the screen we won't be able to have interactive frame rates.

We can solve this using the new GPU programmable pipeline. In recent years we have become able to write shaders (code that we can execute directly on the GPU) to use all the power of the new GPUs computing many operations in parallel. This programmable pipeline allowed people to create new approaches that allowed us to visualize these huge models in real time, while also optimizing the space that we need to store these models. One of these approaches involves substituting textures in place of millions of polygons, and using these to store the height and normal of every point in the original point cloud. Using this information we can display the model using only a few polygons much faster, by changing the normals and depth of each pixel on the GPU.

As we have seen there are many difficulties in visualizing the models that we obtain from photographs. In this thesis we focus on the two main problems of this realistic visualization. The first one is visualizing the model with a single light (just visualizing the polygons) and the second one is using a realistic illumination. We want to display the models as they are in their actual location, and we want to be able to relight them, using for example a sunny day or a cloudy day. The main problem of illumination in 3D is that we work with point lights, and we want to simulate the sun (that has a surface).

2. Main goal

The Daedalus Project has devised new methods for recovering 3D models of scenes from wide-baseline photographs. Current work is focused on developing novel shape-from-shading methods (referred to as Depth Hallucination) to add fine-grain surface detail to the reconstructed models. In doing this, our goal is to reconstruct models that appear visually correct under varying illumination, including subtle effects such as surface self-shadowing. Output from the current software is in the form of a dense polygon mesh and corresponding albedo and normal-depth maps. The main goal of this thesis is to explore GPU algorithms for rendering such models in real time or at interactive frame rates.

The aspects explored include rendering with relief textures, and how best to store the raw data and process it on the GPU. We also study the best way to illuminate the scene in a realistic way, keeping the interactive frame-rates as the most important characteristic.

Evaluation includes measures of performance, and test cases with varying illumination to compare the results of the project with those achieved with a global illumination algorithm.

Another goal of the project is to use only free software. This will concern from the programming environment to the libraries used including the programs that we use for working with images.

Figure 2: Model reconstructed from photographs lit with one point light

The structure of this document follows our work process in Manchester. First we explain how the depth hallucination approach works and the output it produces, which will be the input of the thesis work. Then we explain that the best way to achieve the desired interactive frame rates is displacement mapping, and we show the different approaches that have been studied to decide which one to use. Afterwards we explain the three approaches that have been implemented, and finally we compare their performance and the visual quality of the results.

The second part of the document deals with illumination. As in the previous section, we explain the different ways we have considered to light our models and give a short overview of previous work. Then we explain the algorithm we have implemented, justifying why it was chosen, how the output data is stored, and presenting some results. Finally we explain how we integrated the two parts and show the results we achieved.

The third part discusses how to generalize this approach to complete models (rather than simple planes). We explain why the project chose geometry images as the best solution, their properties and how they help us to solve our rendering problem. We also show some images to explain how the geometry images work and how the results are. Then we describe some further work that could be implemented to improve this project.

Finally we have some appendices where the commented implementation (code) of all the algorithms that are explained in this document can be consulted. We also have a small user guide of the program we have implemented.

3. Depth hallucination

As we have explained in the main goal section we want to display huge models with fine-grain surface detail. We reconstruct a rough model from these photographs and then we want to add small details to its surface in order to keep the realism when we look at it from a very close position. Once we have the reconstructed model we get it as a point cloud that must be triangulated to get the polygonal model from it. Once we have this model made of triangles, we want to add to its surface the fine grained information of the low level details (as the small holes and details of every stone, very small defects on the building, etc.). We get this information from an approach called depth hallucination [GWJ08] that gets the surface details from a depth from shading algorithm. In the next section of this document we explain how this algorithm works.

3.1. Background

Representing surface details is useful to increase the visual realism in a big range of application areas as historical inheritance preservation and architectural reconstruction, graphically rich games and movies, etc.

Figure 3: Original photograph from a real Mayan building (left), derived model relighted with different light conditions (middle) and the same model with another illumination (left) [MGC08]

Capturing detailed surface geometry currently requires specialized equipment such as laser range scanners, which despite their high accuracy leave gaps in the surfaces that must be reconciled with photographic capture for relighting applications. Depth hallucination presents a method that, only using a standard digital camera and a single view, can recover models of predominantly diffuse textured surfaces that can be plausibly relit and viewed from any angle under any illumination.

The main contribution of this idea is to have a shape from shading method, which takes diffuse-lit/flash-lit image pairs and produces a plausible textured height field that can be viewed from any angle.

To capture the input images they employ a standard digital SLR camera mounted on a tripod. Their method requires that the sample of the textured surface hasn't a global curvature as a wall or a floor and the surface must be plausibly represented as a height field. We can see the flow chart of the algorithm in Figure 4.

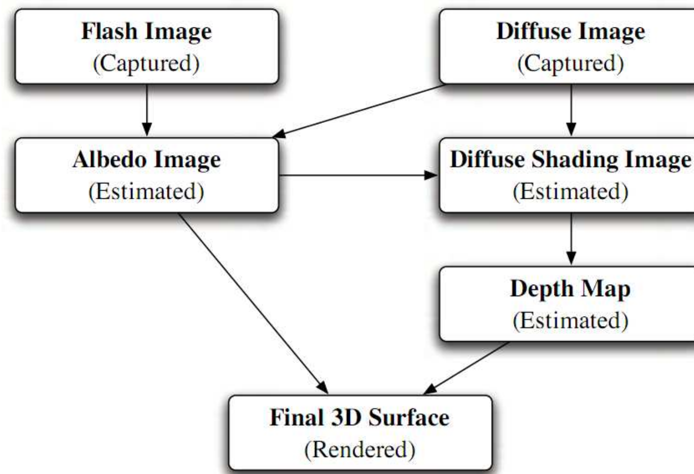


Figure 4: Flow chart showing the steps in the process [MGC08]

First they capture a RAW format image under indirect illumination in diffuse-lit conditions (i.e. overcast skies or shadow). Them a flash-lit image is captured (Ideally the flash should be mounted as close to the camera lens as possible in order to minimize shadows). After this process they compute the albedo map (equivalent to diffuse reflectance) from these two images.

Figure 5: Example of an input photograph pair: Shadowed daylight conditions (left), flash-lit conditions (right) [MGC08]

In order to estimate the albedo map they compute for each pixel (J) the following operation.

$$I_a(j) = \frac{I_f(j) - I_d(j)}{I_c(j)} \tag{1}$$

Pixel values in the diffuse-lit image Id are subtracted from the flash-lit capture If, and they divide the result by pixel values in the flash calibration image Ic taken from a white Lambertian surface at a similar distance and aperture. This yields approximate reflectance values at each pixel.

The next step is the depth estimation. The [LZ94] method is designed to recover shape from shading on a cloudy day, which is precisely what they capture in this technique. Applying their relaxation method entails iteratively ray-tracing a discretely sampled hemisphere of light source directions at every surface point. Instead in this paper they develop an approximate solution that works entirely in image space and yields a direct estimate of depth at each pixel. A conservative model basis ensures that they do not

exaggerate depth variations, and a final, user-specified scale factor achieves the desired roughness, allowing the user to obtain different effects in the final result.

The main idea is that the surface mesostructure (the fine grained details))can be approximated as a terrain with hills and valleys. The orientation of the surface to the sky dominates on the hills, while the visible aperture effect dominates in the valleys, where the sides are at least partly in shadow. So they developed two local models to approximate these different types of relationships between the mesostructure depth and shading. They use one method or the other taking into account if the pixel is in a hill or in a valley as we can see in figure 6.

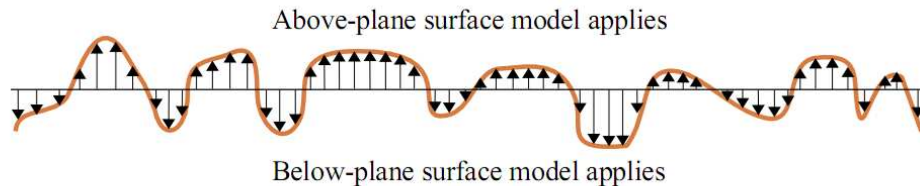


Figure 6: Example of a profile of a textured surface and the separation between the above-plane and below-plane surface models [MGC08]

A shading change over a large region generally corresponds to a greater depth difference than the same shading change over a small region. Since their aperture model estimates depth from diffuse shading relative to a specific feature size they must consider each scale in their captured diffuse shading image separately. Separating their diffuse shading image into scale layers, they efficiently convert their aperture estimates into depth estimates as required for a geometric model.

Figure 7: Difference between a render using a depth hallucinated map (left) and a render using a laser scanned map (right) [MGC08]

Once they had the depth maps they validated the model visually just showing some images to 20 people and asking them if the image was synthetically generated or it was a photograph taken from a real model. They saw that people couldn't difference between their images or the real photographs and that they were also unable to say which was better, a model reconstructed from a laser scan or a model reconstructed using the hallucinated depth. We can see this difference in figure 8 where we have an original photograph and a reconstructed and relit model. As we see in the result image of the relit brick path is really difficult to distinguish if it is synthetic or not. It seems to be a photograph of the same path under a different illumination.

Figure 8: Original photograph (left) and reconstructed and relighted model (right) [MGC08]

3.2. Output

As we have seen the output produced by the depth hallucination algorithm is a depth map of the surface. This is the information that we will have as an input for our algorithm. As we just have this depth information to visualize the model, we have to compute the normal map from this depth map in order to have a proper illumination. Illumination algorithms use the normal information to compute the color of every pixel. If we have a rough model and we want to superimpose this fine-grain surface depth we can't use the normals of the original model because the real depth map normals would change many times inside a triangle (which has just one normal). So if we want to have a realistic relighting of the object we must compute the normal map for the depth map we get as an input.

To compute this normal map we use the Sobel operator. It's very commonly used in image processing, particularly within edge detection algorithms. It is a discrete differentiation operator that computes an approximation of the gradient of the image intensity function. At each point in the image, the result of the Sobel operator is either the corresponding gradient vector or the normal of this vector. It is based on convolving the image with a small separable and integer valued filter in horizontal and vertical direction and is therefore relatively inexpensive in terms of computations. On the other hand, the gradient approximation which it produces is relatively crude, in particular for high frequency variations in the image. We can see the Sobel operator in equation 2.

$$d_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad d_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{2}$$

The code of the program that computes the normal map from the height map using the Sobel operator is commented on the Appendix 1. In figure 9 we can see a depth map and its computed normal map. We store the normal components (x,y,z) in the (r,g,b) channels of color of the texture. Once we have this normal map we can combine it with the depth map to get a relief map, and then use some GPU algorithms, that we will study in the next section, to visualize the object in real time.

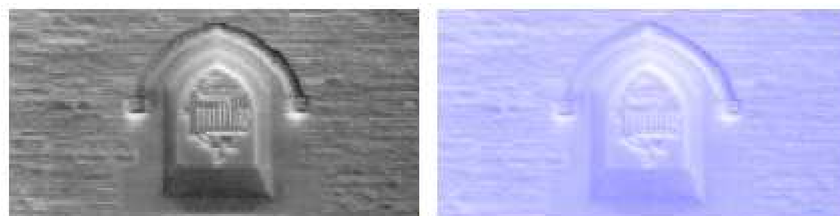


Figure 9: Depth map (left) and computed normal map (right) using the Sobel operator

4. Displacement mapping

Once we have a normal and depth maps, we want to visualize the information that they store. This display can be done in many ways, but all of them have the common name displacement mapping. In the next part of this document we are going to make an overview of some of these approaches, and explain the algorithms we have chosen for the thesis, explaining how they work, how they were implemented, and why we have chosen them.

4.1. Background

Displacement mapping is a technique used to visualize very complex geometry in real time (or interactive frame rates) and also to save space, because it usually works with very huge models. We achieve this performance and save of space by substituting the triangle mesh with a relief map and a simpler underlying geometry. The original surface shape (the one with the complete triangle mesh) is called mesostructure and the new simpler base geometry is called macrostruture. The macrostructure is a simple triangle mesh similar to the original one but without any low level detail. This detail will be encoded as a relief map, a texture whose pixels store the depth and normal of every triangle in the original mesostructure to allow us to map them to the new macrostructure. In figure 10 we can see a schematic illustrating how relief mapping works.

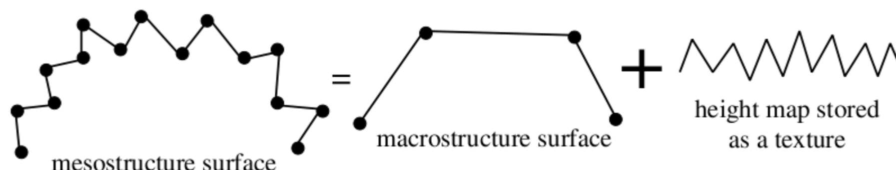


Figure 10: Displacement mapping scheme [LT 08]

A relief map is a texture in which we store a height-field (height difference between the macrostruture and the mesostructure) in the alpha channel, and the (x,y,z) components of the normal of each height in the RGB channels. We use the height stored in the alpha channel to displace the pixels and simulate the mesostructure surface details and the normals to compute the illumination for every pixel that is displayed on the screen. This is the reason why relief mapping is a per-pixel displacement mapping.

As in every visualization algorithm we must know what the eye is seeing and what not. To compute this we must trace a ray from the eye in the direction that it is looking and see which part of the object it hits first. This will be what the eye is seeing, because everything after this point will be invisible (unless the material is transparent), occluded by the first object the ray hit.

To combine the relief map and the macrostructure we use the fragment shader of the GPU programmable pipeline. In the shader we compute the intersection between the visibility rays, traced from the user, and the height-field encoded in the relief map as the usual GPU ray tracing, with one main difference: everything is computed in tangent space (tracing the ray through the texture). In figure 11 we can see the relief mapping process scheme.

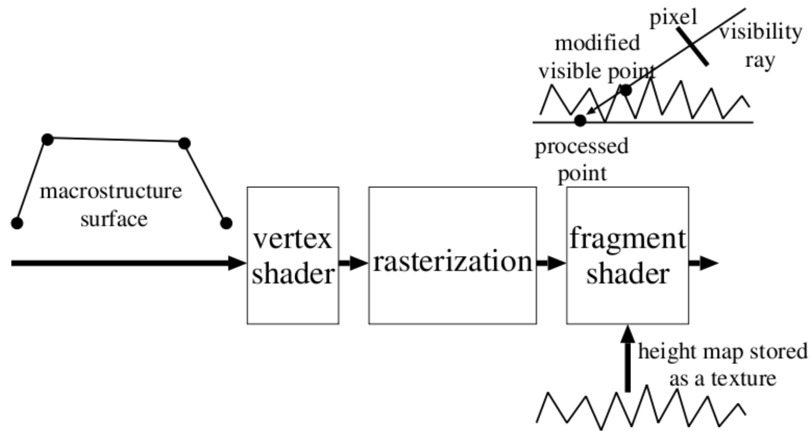


Figure 11: Displacement mapping process scheme[LT 08]

The main goal of this approach is to improve the realism of the scene by including this fine grained surface detail, especially surface self-shadowing effects, in the visualization. In our case this detail will come from the depth hallucination algorithm.

At the beginning of computer graphics we just used textures to improve the realism of the scenes; when we see a textured object from far away we perceive it as realistic because we don't expect to see all the fine grained surface detail, and the texture colors are effective for this. The problem with simple texturing of a surface is that as we get nearer to the object's surface this realism decreases because we see bumpy surfaces as flat ones and the illumination is inconsistent with the things we should see.

Many techniques have been proposed in the last few years to improve this effect. We explain some of them in the following subsections. The main approach is relief mapping and its derivatives and, as we will see, the main problem to solve is the way we compute the intersection between the viewing ray and the height field in texture space in GPU. We can see in figure 12 the main idea of these approaches.

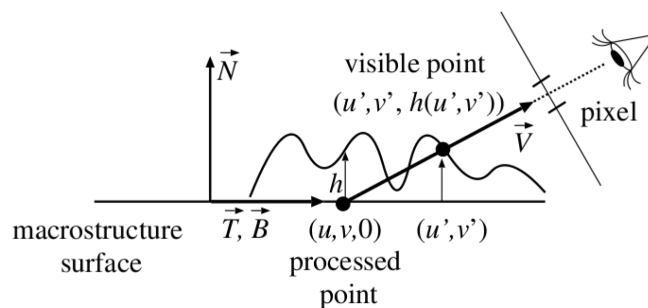


Figure 12: Displacement mapping[LT 08]

We trace a ray from the eye to the (u,v,0) point which corresponds to the texture coordinates (u,v), which the texture mapping tells us that correspond to the geometry (the macrostructure) for a given pixel, and the height 0 which is the height of a usual texture mapping. Then we have to trace a ray from the eye to this point and find the first intersection with the height-field (of course, taking into account the height of the texture in each pixel). The main problem is that if we explore every texel (every texture unit between the values 0 and 1) to see if we find an intersection the algorithm would be very slow because it would have to make many comparisons for each pixel and we will never have real time

rendering. In the other hand if we don't explore every texel we may not find the correct first intersection, but this is a risk that we must assume because we want the algorithm to run in real time. All the approaches will try to find this first intersected texel (u',v',h') in the minimum time trying to get the most accurate result.

4.1.1.Non-iterative methods

The first methods described next are non-iterative methods. These approaches read the height map at the processed point (the point that texture mapped point) and use this local information to obtain more accurate texture coordinates to index the normal map and color textures.

a. Bump mapping

The first approach to generating surfaces with a bumpy appearance was bump mapping [Bli 78]. This technique takes the mesostructure normal vectors and applies them to the macrostructure, which is usually locally flat. We map the normal map, using some suitably chosen mapping function, onto the macrostructure and we use the normal stored in the texel that corresponds to each pixel to compute the illumination. We can see how this works in figure 13.

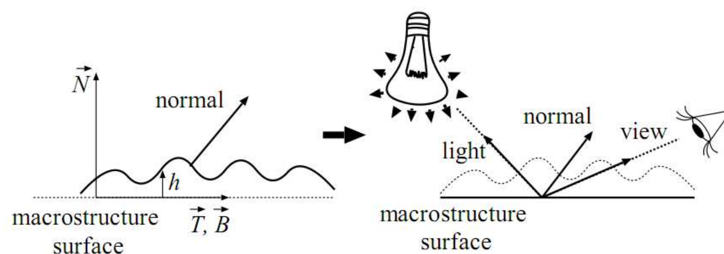


Figure 13: Bump mapping

As we compute the color (illumination) using the mesostructure normals, from far away and if the bumps are not very pronounced, it achieves realistic results. The problem appears when we get closer to the surface of the object and we see it illuminated as if it had bumps but without having them, appearing to be a flat surface illuminated in a strange way as we can see in figure 14.

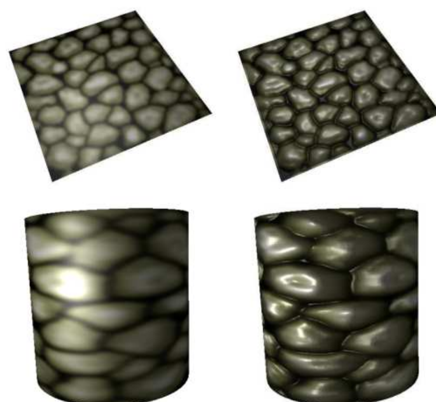


Figure 14: Texture mapping (left) and Bump mapping (right) [LT 08]

Bump mapping was the foundation for all the techniques that we will explain from now on, and it was very important in its time. The fact that it was implemented in hardware, even before the emergence of programmable GPUs, indicates its influence.

b. Parallax mapping

The next approach that appeared to solve the problem was parallax mapping [KKI 01]. It controls not only the shading normals, but also modifies the texture coordinates used to obtain the mesostructure normals and colors. Unlike bump mapping, this approach uses a relief map because it needs the height-field in order to compute the new value of the texture coordinates, which are then used to obtain the normal and color in a given pixel.

The texture coordinates are modified assuming that the height field is constant h(u,v) everywhere in the neighborhood of (u,v). As we can see in figure 15 the original (u,v) coordinates (the coordinates that the texture mapping associates to a given pixel) are substituted by the new ones (u',v') which are calculated from the direction of the tangent space view vector V and the height value h(u,v) read from the texture at the point (u,v). This assumption of a constant height surface simplifies the ray equation very much.

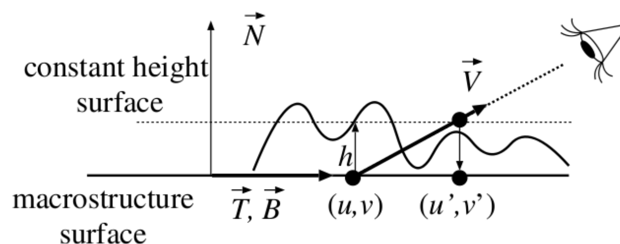


Figure 15: Parallax mapping[LT 08]

This approach improves a lot the viewing realism of the objects, because it takes into account the height of the mesostructure and uses a more accurate normal and color due to the way it computes the new texture coordinates that will be mapped to a pixel. As we can see in figure 16 it really makes a difference with bump mapping at (more or less) the same speed. The bumps produced are much more plausible because of their silhouette and the fact that they are really bumps, not just normals that have been distorted over a plane.

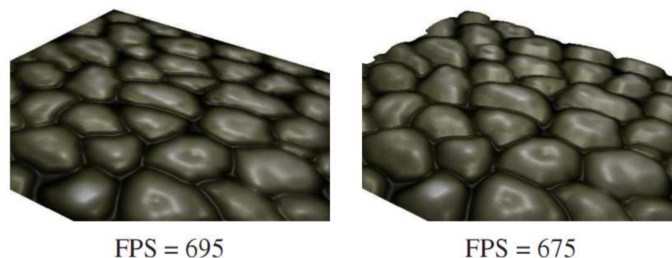


Figure 16: Comparison between bump mapping (left) and parallax mapping (right) [LT 08]

The main problem with parallax mapping is that as the viewing angle becomes more grazing offset values approach to infinity, because the viewing ray and the assumed constant height became nearly parallel. When this happens the chances of (u',v') to index a similar height to the (u,v) one fade away and the result seems to be random.

c. Parallax mapping with offset limiting

To solve the problem with the grazing angles of the original parallax mapping [Wel04] proposed to use an offset limit instead of letting it approach infinity. So the shader will include a limit to make the offset lower than the height of the height-field at (u,v) assuming that the view vector is normalized (as the texture space goes from 0 to 1 we need the vector to be normalized). Using this approach the offset values will never approach to infinity and the values stop seeming random. We can see in figure 17 how this method works.

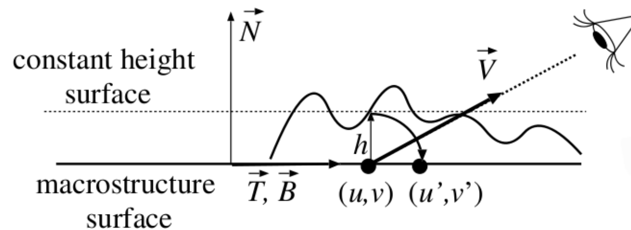


Figure 17: Parallax mapping with offset limiting[LT 08]

This approach achieves better results, especially at grazing angles, in which the offset values never approach infinity making the method more robust and the shader even smaller and easier. We can see the difference between the usual parallax mapping and the offset limiting approach in figure 18. It appears the problem that we pointed before, as the viewing rays become more grazing (nearly parallel to the plane surface) the results seem to be random, and some strange effects appear in the horizon of the figure in the left.

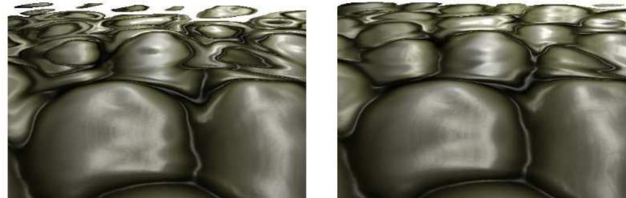


Figure 18: Parallax mapping (left) and Parallax mapping with offset limiting (right) [LT 08]

As we can see in figure 18 on the left as the viewing angle becomes more grazing (as we go far away in the surface) some strange effects start to appear making the object lose all the realism. As we can see this artifacts are smaller in the figure in the right in which they used offset limiting to control them, but the result isn't also the best, because as the angles become more grazing some artifacts (smaller than in the original approach) appear and break all the feeling of realism when we look to the object.

d. Parallax mapping taking into account the slope

Parallax mapping assumes that the mesostructure has a constant height plane at every pixel. This assumption is completely incorrect even though, in practice, it works quite well. We can have a better approximation of the real surface if we assume that the surface is still planar, but its normal vector can be arbitrary [MM05]. This is not a problem as we can take this normal from the normal map (RGB channels) and the height from the height-field (alpha channel) so we don't need to add any further texture lookup. We can see a schematic explaining how the method works in figure 19.

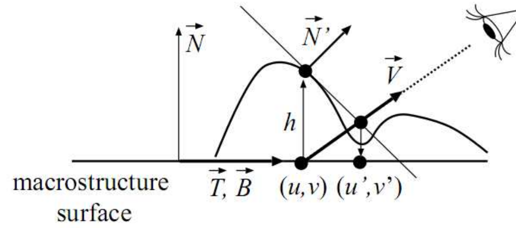


Figure 19: Parallax mapping taking into account the slope[LT 08]

The main advantage of this approach is that the shader is as simple as the original one but the results are improved even getting small speed increase as we can see in figure 20. We can see how the illumination produces softer and more plausible results and the speed of the algorithm is even increased.

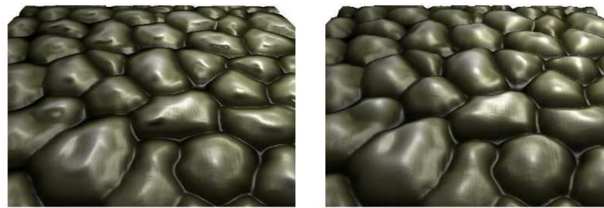


Figure 20: Comparison between parallax mapping with offset limiting (left) and parallax mapping taking into account the slope (right) [LT 08]

4.1.2. Unsafe iterative methods

Iterative methods explore the height field globally to find the intersection between the ray and the height field. The main problem of these methods is that they find an intersection quite fast, but they don't pay attention to obtaining the first one (the one closest to the eye) so sometimes we will see some things that we shouldn't, because what we see in the real world is the first intersection in the direction we are looking.

a. Iterative parallax mapping

[Pre06] is the most recent attempt to improve the performance of parallax mapping. The classical approach tries to move the texture coordinates by an offset towards the really seen height field point, but with a single attempt the results are far away to be perfect. The accuracy of this solution can be improved by repeating the offset correction step a few (3-4) times. The result is a very fast, but unsafe, method. It can't guarantee that the found intersection point is the closest to the eye. Even though unsafe the method is popular because it can cheaply but significantly improve the original parallax mapping approach, as we can see in figure 21. Again we can see how the illumination produces softer effects as the hits are more accurate and the normal variations are smaller and more continuous than in the previous approach. As we can see it is a little bit slower but the results are better and the speed is more than enough to have real time rendering.

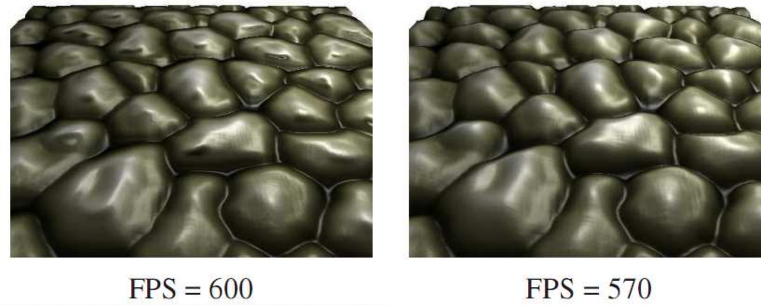


Figure 21: Comparison between parallax mapping with slope (left) and iterative parallax mapping (right) [LT 08]

b. Binary search

[POC 05,PO 05] is an application of the standard binary search method to solve the problem of finding the intersections between the viewing ray and the height-field. We suppose we have two guesses on the ray that enclose the real intersection point, where one guess is above the height-field (origin) and the other is below it (original texture coordinates (u,v) that correspond to a pixel). Binary search halves the interval (Hmin,Hmax) containing the intersection in each iteration step putting the next guess at the middle of the current interval. Comparing the height of this guess and the height-field we can decide whether or not the middle point is below the surface. If it is below, we keep the half-interval where the end point is above and conversely, if it's above we keep the interval where the end point is below the height-field. Then we halve again the interval and we repeat this process as many times as we want. We can see how it works in figure 22.

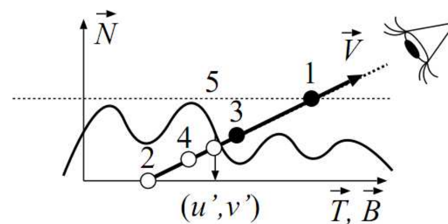


Figure 22: Binary search [LT 08]

This method converges quickly to an intersection but it doesn't always result in the first one, as we can see from the artifacts of figure 23. It's not the first one because when we halve the intervals we don't take into account what happens in the one that we reject and it could happen that the ray enters the height field and comes out again before the end of the interval. If this happens this method will never notice it, and will just jump above it.

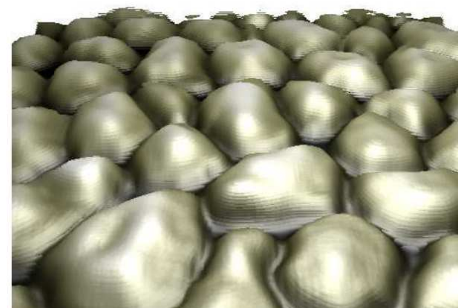


Figure 23: Binary search results [LT 08]

c. Secant method

[YJ 04, RSP 06] instead of just halving the interval assumes that the surface is planar between the two guesses and computes the intersection between this planar surface and the viewing ray. This means that if the surface was really a plane between the first two candidate points this intersection point could be obtained at once. The height-field is checked at the intersection point and one endpoint of the current interval is replaced keeping always a pair of end points where one end is above while the other is below the height field. We can see how it works in figure 24. We get the last two points from a linear search (one above and one below the relief map. Once we have this we get the height of the relief map in this two points, we trace a line from one to the other and compute the intersection between this line and the ray. Once we have this new point we substitute the point of the same type (above or below the relief map) by the new one and repeat the procedure. It could happen that this process never converges. This is the reason why we have to iterate just a fixed number of steps.

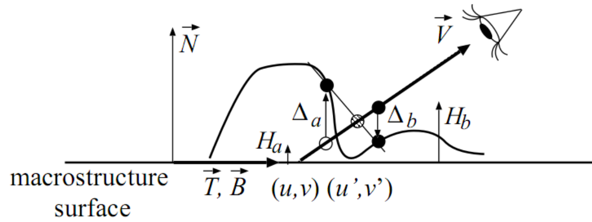


Figure 24: Secant method [LT 08]

4.1.3. Safe iterative methods

Safe methods pay attention to finding the first intersection even if the ray intersects the height-field several times. In some methods if the steps are larger than the texel size these algorithms may still fail, but the probability of this is low. Other techniques are completely safe, but require preprocessing that generates data encoding the empty spaces in the height-field. We have to take into account that safe methods guarantee that they do not skip the first intersection, but they cannot guarantee that the intersection is precisely located if the number of iterations is limited.

a. Linear search

[Lev90] is just "quasi-safe" because if the steps are bigger than the pixel size it may fail, but the probability of this is very low, because it requires a very spiky surface to fail. It works in a similar way to the binary search in the sense that it finds a pair of points on the ray that enclose the first intersection (the same pair of points as the binary search), but then, instead of halving the interval we start taking steps of the same length on the ray between these two points until we find a jump from a point above the height-field to one below it. Then we have found the intersection. Usually this method is used as a first pass to obtain a small interval with just one intersection between the ray and the height-field and then refine the result using another algorithm. We can see how it works in figure 25.

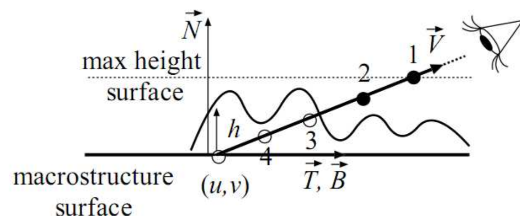


Figure 25: Linear search [LT 08]

The smaller the steps the more accurate is the result because there appear less artifacts in the visualization but also, as we decrease the step size we increase the number of steps that we need to reach the first intersection, and then we have a slower performance. Notice in figure 26 the stair-stepping artifacts that appear and how they reduce when we double the number of iterations (but they are still there). Notice also how the frame rate decreases when we increase the number of linear search steps to perform (we decrease the step size).

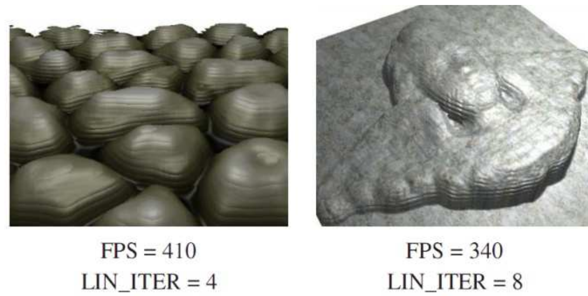


Figure 26: Linear search results [LT 08]

b. Other methods

There are many safe iterative methods that encode the empty space in the height-field in different ways. One approach that needs preprocessing is pyramidal displacement mapping, whose main idea was mentioned in [MM05] and was fully implemented in [OK06]. It encodes the empty space as a mip-map like, hierarchical scheme. A pyramidal displacement map is a quadtree image pyramid. Each leaf texel at the lowest level of the mip-map indicates the difference between the maximum height surface and the current height at the given texel. The root texel, or the top mip-map level, denotes the global, minimum difference.

Sphere tracing is another approach that was introduced in [Har93] to ray trace implicit surfaces and was applied to height field tracing in [Don05]. It precomputes a 3D texture within an axis-aligned bounding box of the bumpy surface patch. Texels of the texture correspond to 3D points and the value stored in the texel is the distance from the corresponding point to the closest point on the bumpy surface. This distance can also be interpreted as the radius of the largest sphere centered at the corresponding point that does not intersect, only touches, the height-field. That's why the method is called sphere tracing. Finally it uses this information of the empty space to jump from one sphere to another in a completely safe way, because every sphere has only empty space inside.

Another method that precomputes some information is cone stepping [Dum 06]. It encodes the empty space over every pixel as the radius of a cone and then the stepping, instead of being linear, is made from one cone to the next one, ensuring that we will always find the first hit as any space jumped is known to be empty. Of course the information related to the cones must be precomputed for its use in the algorithm. We can see how the algorithm uses the cone information to jump over the empty space in figure 27.

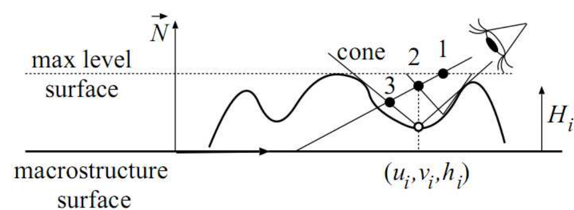


Figure 27: Cone stepping [LT 08]

Each one of these methods is described as a single one but they are usually used in a combined way to obtain better results. We usually divide the ray tracing in two parts. The first one is used to find an interval in which just one hit can be found by the ray we trace. The second looks for the intersection inside this interval. We usually use iterative safe methods to find this interval and unsafe methods (they will be safe as there is just one hit in the interval), that are faster, to find the intersection in this interval.

4.2. Displacement mapping implementation

We decided to implement some different algorithms to be able to compare the results and performance on each algorithm. We decided to start implementing relief mapping, the most common and popular approach, after this we decided to improve its performance by implementing the interval mapping for the second search step and make a comparison between them. Finally we decided to implement one of the last methods that appeared called relaxed cone stepping, that is similar to the cone stepping using different cones. Implementing this method we could be able to compare the performance of a method that requires a preprocessing step and two methods that don't. After implementing these three methods, which are easy to understand and have proven results, we decided to implement self shadowing for each approach as we want to have in further steps a realistic illumination.

4.2.1.Relief mapping

The first method we decided to implement was the most usual (classical) one called relief mapping [OBM00, Oli00]. It combines a linear search to find an interval with just one hit in it and them it performs a binary search on this interval to find the texel where this intersection is located. As these two algorithms are quite simple the result is a very simple shader that works quite fast and with accurate results. The performance depends on the number of linear and binary search steps we perform, but the quality of the result depends also on these two variables. It exhibits some problems at grazing angles due to the fact that some intersections may not be found in the steps we have chosen, as the normals of the mesostructure are quasi-perpendicular to the viewing direction. It also appears distorted when we look from these angles.

After implementing this algorithm we visualize an illuminated scene. Using this approach makes sense for bumpy surfaces in which we want to preserve the detail but keeping a reasonably cheap mesostructure with few triangles. For bumpy surfaces realism one of the most important things is the self-shadowing [Wil78], because the height differences create lots of shadows and illumination effects on the surface of the objects. To obtain this self-shadowing using relief mapping is almost trivial. We just have to run the same algorithm substituting the eye by the light and the texture coordinate where de eye is looking by the texture coordinate that the relief mapping has given as an output. Then we trace a ray from the light to the point we want to illuminate and we compute the first intersection between the ray and the height-field. If the given point (the one resulting from relief mapping) is the same as the first intersection the point will be lit, if not it will be in shadow.

This simple approach gives as a result hard shadows that are not realistic at all and create a strange visual effect on the user because there is not penumbra on them, but as we just have one point light and it really casts hard shadows we have enough with this result.

The commented code of the implemented shaders can be found in appendix 2.

4.2.2.Interval mapping

Next, we searched for algorithms to improve the speed results without losing visual quality. As we want to have a more complex illumination algorithm in the following experiments, and the algorithm will slow down depending on the number of lights that we use, we must have an algorithm that works as fast as possible. We decided to improve the second step of the algorithm substituting the binary search by the algorithm called interval mapping [RSP06].

The interval mapping approach starts with two points on the viewing ray, one above the height field and another one below it. This is the reason why it's used as a second step in the relief mapping approach. We first perform a linear search to find these two points and then we run the interval mapping algorithm. The main advantage of this method is that it gets the same results as the binary search but in just a few iterations instead of many.

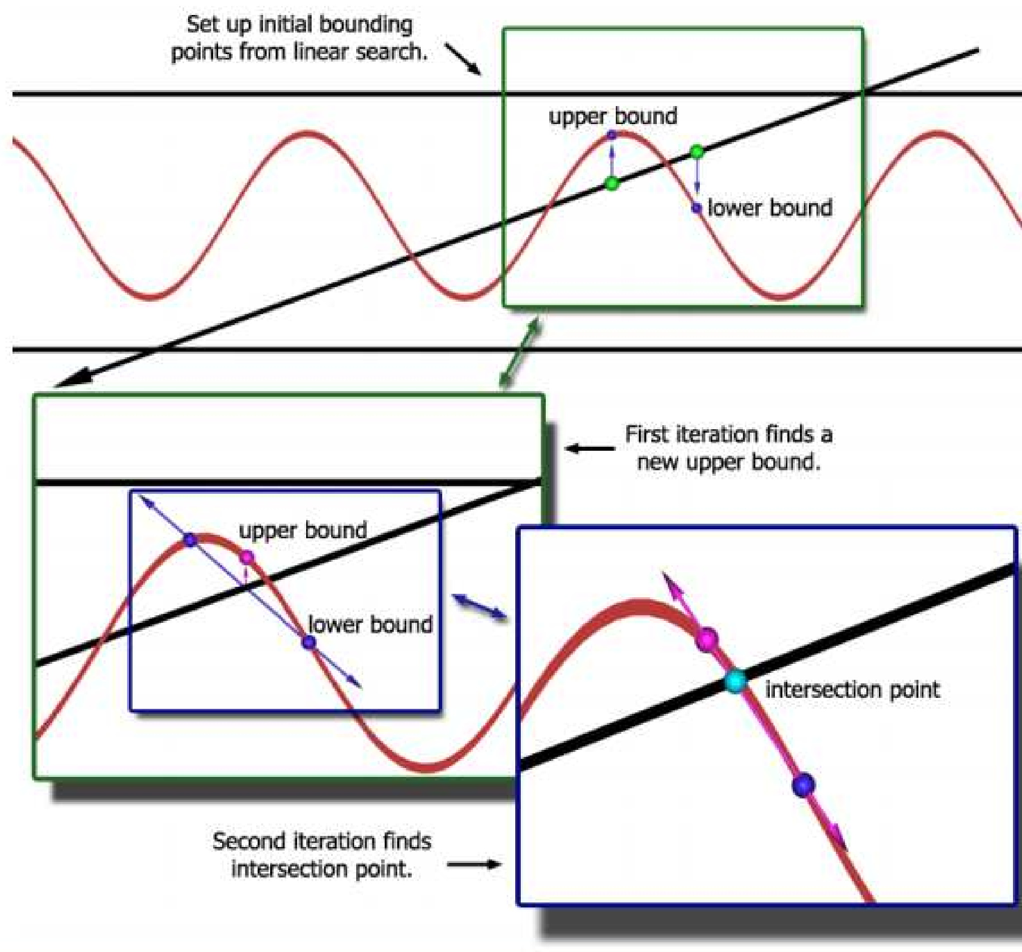


Figure 28: Interval mapping [RSP06]

Once we have these two points (as we see in figure 28) the algorithm sets one as an upper bound and the other one as a lower bound and computes the intersection between the viewing ray and the line that goes from one to the other. This computation is very simple, and that's the reason why this algorithm is fast. We have to perform the following computations to have the new point. We get the line from the upper bound to the lower bound (taking into account the X and Y components). And finally with a simple computation we just get the new values for x and y and we just have to decide if this new point is the upper bound or the lower one. We show in equation 3 the computations that we perform.

$$X = C_{line} \frac{B_{view}}{A_{view}B_{line} - A_{line}B_{view}} \tag{3}$$

$$Y = C_{line} \frac{A_{view}}{A_{view}B_{line} - A_{line}B_{view}}$$

being

$A_{line} = Y_{UpperBound} - Y_{LowerBound}$

$B_{line} = X_{LowerBound} - X_{UpperBound}$

$C_{line} = -B_{line}Y_{UpperBound} - A_{line}X_{UpperBound}$

Then, the upper or the lower bound is updated depending on the position of the intersection with respect to the height field. We update the value of the bound that is in the same relative position as the point we found. This way we always have a point above and another below the height-field and we can always perform a new iteration of the algorithm.

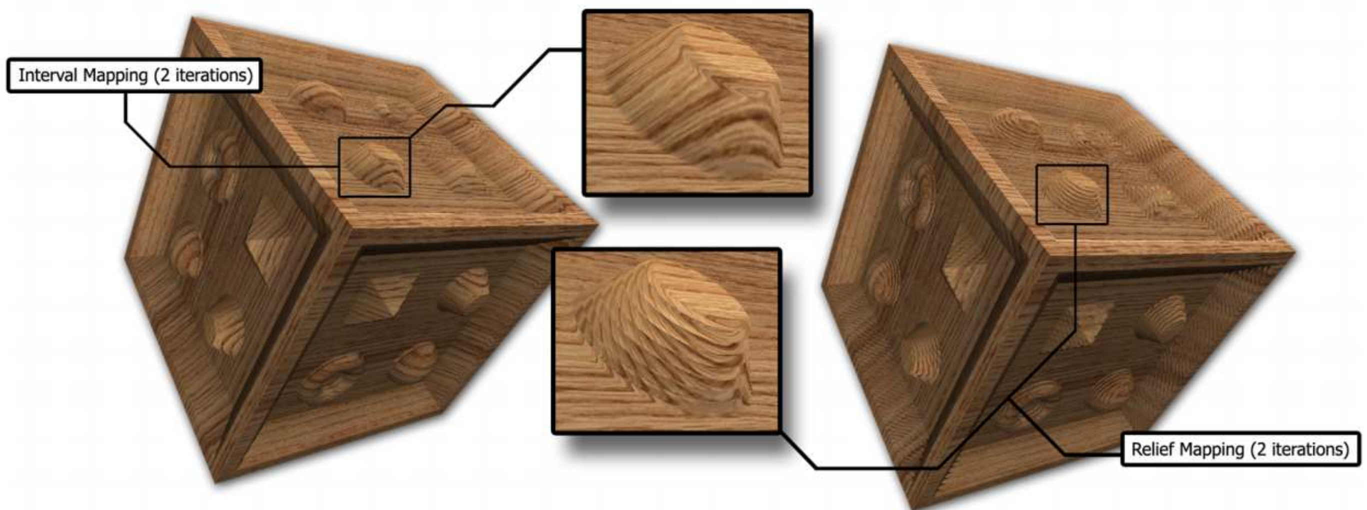


Figure 29: Comparison between interval mapping (left) and relief mapping (right) with two steps in the interval mapping and binary search step respectively [RSP06]

This way we achieve really good results in a few steps. Usually two steps are enough to have good results. As we can see in figure 29 if we perform just two binary search steps in relief map we have lots of artifacts on the result, but using interval mapping the results are much better. These artifacts are easier to see as we use the same texture coordinates that we get from the relief map for the color mapping. As we see for the relief mapping approach the color makes no sense and seems to be distorted. This shows that interval mapping allows us to use less iterations than relief mapping. If we use Interval mapping it really seems to be wood as the color is correctly mapped. It has the same results as classical relief mapping but using fewer steps to perform the search.

Using interval mapping we had a sensible increase in the speed of the visualization without losing the fine grained surface detail that we want to keep. The commented code of the implementation can be found in the appendix 3.

4.2.3.Relaxed Cone stepping

Relaxed cone stepping [PO07] is a similar approach to cone mapping. As we explained before, cone mapping encodes the empty space above every pixel as a cone to allow the algorithm to jump through this empty space without ignoring any hit and finding a suitable interval to perform a new and faster search. These cones never enter the height-field because they just encode this empty space between the bumps. Relaxed cone stepping relaxes this condition. The cone of each pixel encodes the fact that any ray that comes into the cone in any direction could enter the height field but never get out of it. This will allow wider cones and hence faster stepping, and this is actually the main goal of this approach (to get as wide cones as possible). In figure 30 we can observe the difference between a cone for cone mapping and a relaxed cone for relaxed cone stepping in 2D. As we can see the relaxed one is much wider than the original one.

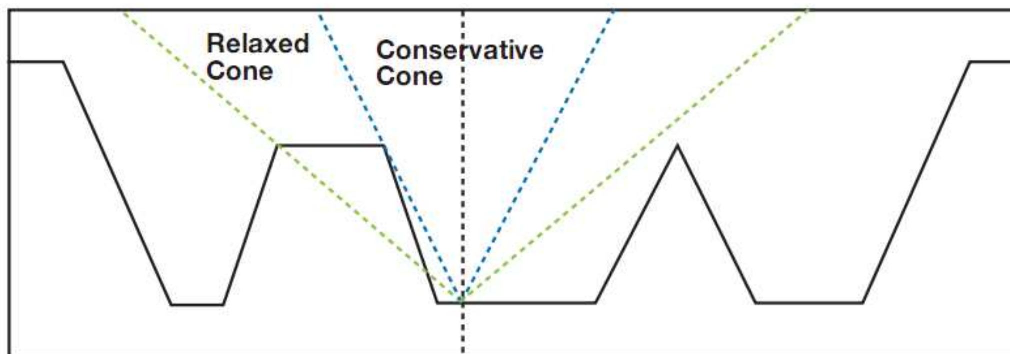


Figure 30: Comparison between conservative and relaxed cones[PO07]

The first thing we have to do is assigning a cone to each texel of the depth map. These cones will be stored by their width / height ratio because this value can be stored in a single texture channel and then both a depth and a cone map can be stored using a single luminance-alpha texture. Alternative (and this is the approach we will use) the cone map could be stored in the blue channel of a relief texture (with the first two components of the normal stored in the red and green channels only and the depth in the alpha channel) because we can compute in real time the third component of the normal by multiplying the red and green components as they store two of the normal components.

For each reference texel on a relaxed cone map, the angle of the cone centered there is set so that no viewing ray can possibly hit the height field more than once while traveling inside this cone. Note that as cone maps are used to accelerate the viewing ray hit they can be used to accelerate the self-shadowing too.

These cone maps are computed offline using an O(n^2) algorithm described by the pseudocode shown in code1. The main idea is that for each source texel ti, we trace a ray through each destination texel tj, such that this ray starts at (ti.texCoord.s, ti.texCoord.t,0.0) and points to (tjj.texCoords.s, tj.texCoord.t, tj.depth). For each such ray we compute its next intersection with the height field and use this intersection point to compute the cone ratio cone_ratio(i,j). Figure 31 illustrates the situation for a given pair of (ti, tj) of source and destination texels. Ci’s final ratio is given by the smallest of all cone ratios computed for ti, which is shown in figure 32. The relaxed cone map is obtained after all texels have been processed as source texels.

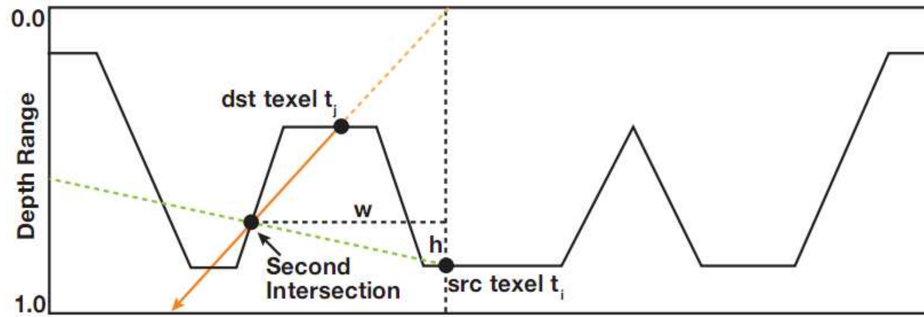


Figure 31: An intermediate step during computation of a cone ratio. We trace the ray and find the two intersections; the second one gives the cone ratio for the src texel [PO07]

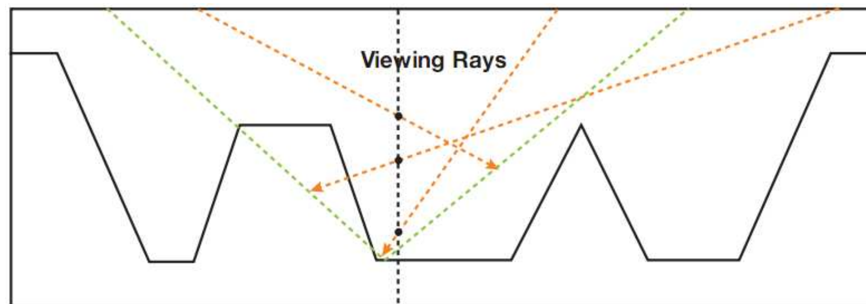


Figure 32: Some viewing directions and relaxed cone [PO07]

```
for each reference texel ti do

    radius_cone_C(i) = 1;
    source.xyz = (ti.texCoord.s, ti.texCoord.t, 0.0);

    for each destination texel tj do
        destination.xyz = (tj.texCoord.s, tj.texCoord.t, tj.depth);
        ray.origin = destination;
        ray.direction = destination – source;
        (k,w) = text_coords_next_intersection(tj, ray, depth_map);
        d = depth_stored_at(k,w);

        if ((d  – ti.depth) > 0.0)  // dst has to be above the src
            cone_ratio(i,j) = length(source.xy – destination.xy) /(d – tj.depth);

            if (radius_cone_C(i) > cone_ratio(i,j))
                radius_cone_C(i) = cone_ratio(i,j);
```

Code 1: Pseudocode of the preprocessing for relaxed cone mapping

In figure 33 we can see the difference between relaxed cone maps and classical cone maps. As we can see the relaxed map has bigger values (nearer to white) for each pixel as the cone ratios are bigger for this approach.

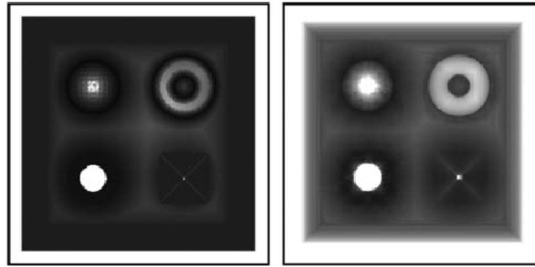


Figure 33: Cone map (left) and relaxed cone map (right), as we can see the values of the cone ratio are much bigger in the relaxed cone map, as they approach to 1 and the entire map is brighter than the classical cone map

Once we have the cone map we have to implement the stepping algorithm. To shade a fragment, we step along the viewing ray as it travels through the depth texture, using the relaxed cone map for space leaping. We proceed along the ray until we reach a point inside the surface. Then we use the last point outside and the first inside as an interval to proceed with the binary search of the intersection.

To step along the ray we must find a way to compute the intersection between the viewing ray and the relaxed cones; this will allow us to jump from one cone to the next one. We can see the process in figure 34. In the figure m is the distance, measured in 2D, between the current texel coordinates and the texel coordinates of the intersection point. The difference between the depth at the current ray position and the depth of the current texel is d+g.

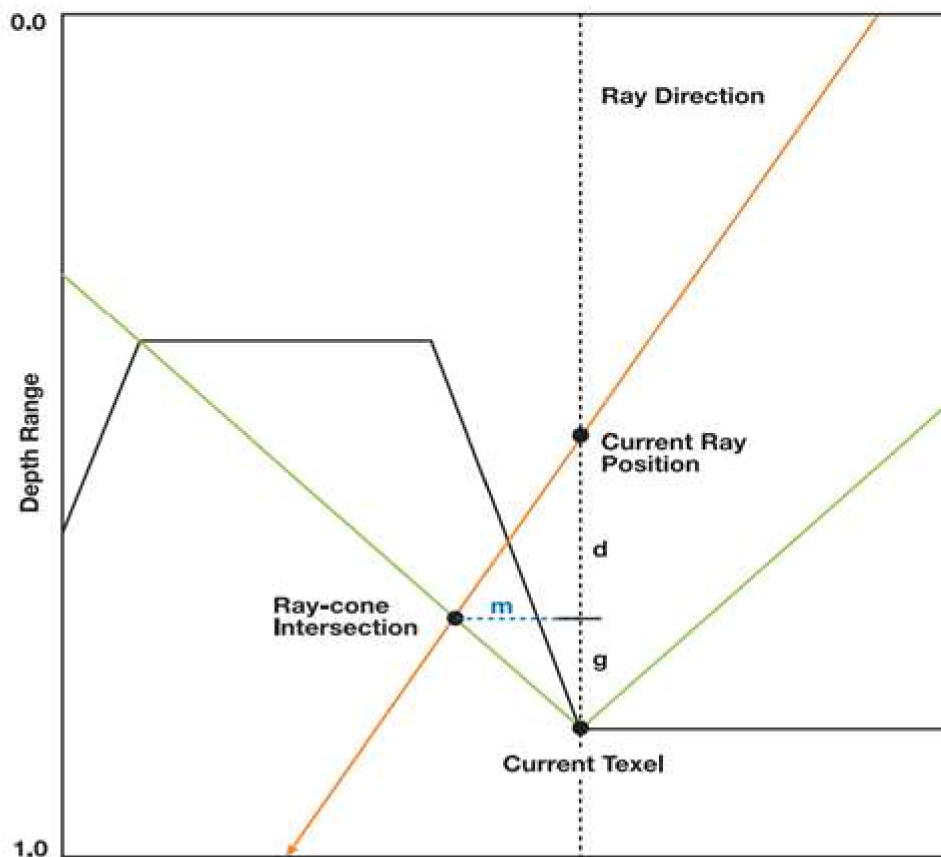


Figure 34: Ray-relaxed cone intersection process [PO07]

Taking into account this figure we have the next equations that allow us to compute the intersection point and take another step along the ray to find the intersection.

The first thing we have to do is to scale the ray direction by its z component. The distance between the actual texel and the intersection one will be the distance d multiplied by the length of the scaled ray direction.xy (as we have scaled the direction the z component will be one). We will call this length the ray_ratio.

$$scaledRay.direction.xyz = \frac{ray.direction.xyz}{ray.direction.z} \quad (4)$$

$$m = d * length(scaledRay.direction.xy) = d * rayRatio$$

Likewise we can compute m from g taking into account that multiplying this value by the cone ratio we will have this distance.

$$m = g * coneRatio \quad (5)$$

$$m = \big((currentTexel.depth\ -\ rayCurrentDepth)\ - d\big) * (coneRatio)$$

Then solving equations in 5 for d we get equation 6.

$$d = \frac{(currentTexel.depth - rayCurrentDepth) * coneRatio}{rayRatio + coneRatio} \quad (6)$$

Once we get d from equation 6 we can compute the intersection point I easily just computing equation 7. This equation is really the one that encodes the stepping. As we see it adds to the current position of the ray, a certain distance d in the ray direction. Once we have this we will compute the same for the next texel (corresponding to I).

$$I = rayCurrentPosition + d * scaledRay.direction.xyz \quad (7)$$

The shader implementing this approach can be seen in appendix 4. Some important things that must be taken into account are the following. For performance reasons, we step along the ray a fixed number of times. If we find the suitable interval (a point under the surface) we break the loop, if not we will have an error. We do this because for some grazing angles we will be stepping too long before finding an intersection and this will have a great importance in the final performance. Once we have this interval we perform the same binary search that we performed in relief mapping.

4.3. Results of implementation

Once we have implemented the three approaches that we have explained in the previous chapters we wanted to compare their performance. We wanted to see how they improve a simple texture on a polygon and the frame rates we achieved using this approaches. The first thing we did was comparing their performance. We took a 800x800 window and displayed two triangles conforming a square. Using a GeForce GTX 260 we got for the relief mapping a frame rate of 2825.5 FPS, for the interval mapping 2903 and for the relaxed cone map 2957. From these results we can see that improving the second step in relief mapping really makes a difference, because it improves in 76.5 FPS when we use the interval mapping approach. This is due to the fact that we have to perform 8 steps at least to get a noise free visual result. Instead in interval mapping we can perform just 2 steps to get a visually similar result. Relaxed cone mapping also improves sensibly these two approaches, but it has the problem of the preprocessing. To preprocess a 512x512 texture it takes to the preprocessor approximately three hours.

Of course, at the same speed we had better results for interval mapping than relief mapping because we will be able to perform more linear search steps and then find the most suitable interval to perform the interval mapping. When we look the object from a grazing angle, if we perform just a few steps we probably won't find this interval, because the direction will be very near to a perpendicular of the macrostructure and then it could take more steps to find it. We can see this effect in figure 35 where (inside a blue circle) some artifacts in the first image (with 10 linear search steps) appear (inside a blue circle) that seem to be pieces of the object floating in the space, and they are reduced in a sensible way when the linear search steps increase.

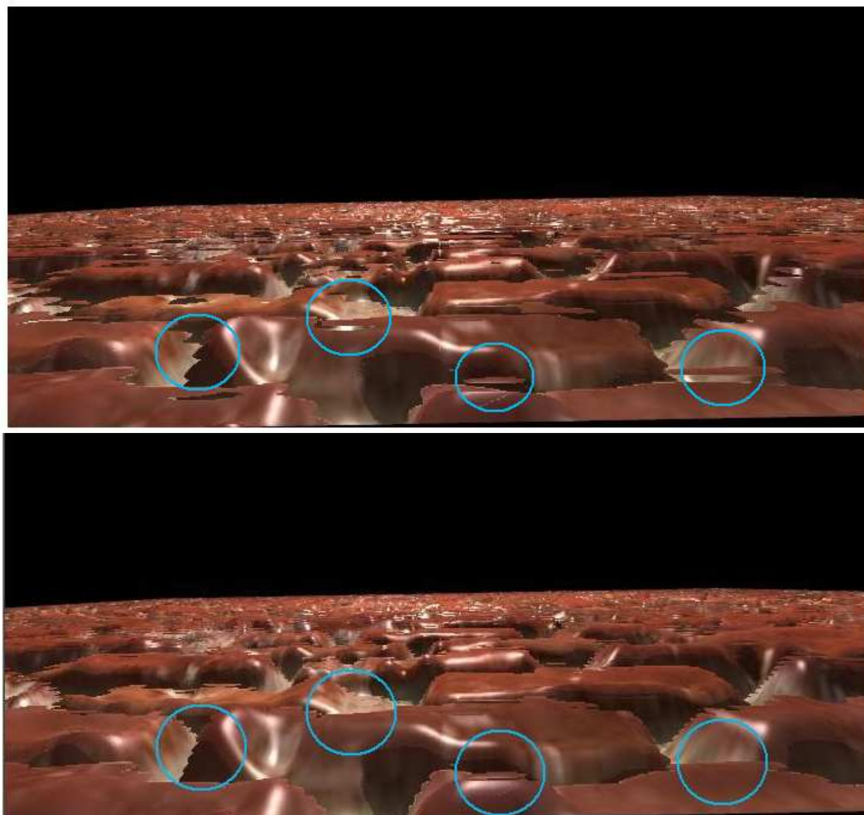


Figure 35: Relief mapping artifacts with 10 linear search steps (above) and relief mapping with 20 linear search steps (below). Notice how the artifacts have been reduced.

Figure 36: Relief mapping result with 10 linear search steps

This happens when we have grazing angles. If we don't look from this angles the result is perfect as we can see in figure 36 that is taken from the relief mapping approach with 10 linear search steps and 8 binary search steps. What we had to decide in that moment was the configuration we wanted to use for our shader. We decided to use 10 linear search steps because we wanted to have a real time (or interactive) application and, at this point in the project we could have chosen 20 because the speed is not the main problem as we have very high frame rates, but as we wanted to add a realistic illumination and we knew that this will be a great speed-down for our application we preferred to keep the speed as the most important think above a perfect (noise–free) visualization. We decided to adjust this kind of things if possible when the complete algorithm (including illumination) was finished.

With interval mapping we had similar problems, but using just 10 steps of linear search and only 2 of interval mapping we found that the artifacts were less noticeable than the ones in relief mapping. As we can see in the first image of figure 37 there are no "flying" parts of the object and the artifacts are more noticeable because of the color that seems to be distorted than for the shape of the object. There are some problems with the borders of the bricks also. If we increase the number of linear search steps we find that, as in relief mapping, the artifacts are also reduced drastically.

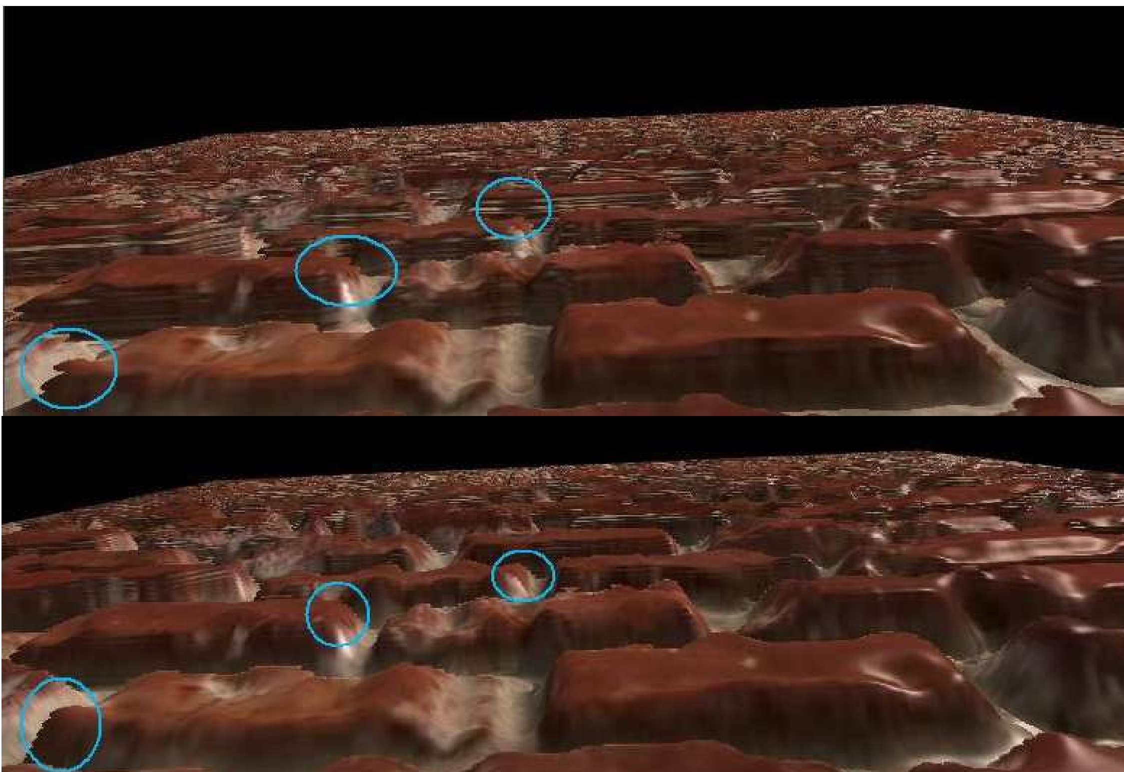


Figure 37: Interval mapping using 10 linear search steps (above) and interval mapping using 20 search steps (below). Notice how the artifacts highlighted in the images are reduced

We made the same decision as before because we wanted to increase the speed as much as possible and we decided to change these parameters if possible when we had the illumination implemented. As you can see in figure 38 if we don't have these grazing angles the visualization is completely correct.

Figure 38: Interval mapping with 10 linear search steps and 2 interval mapping steps

Finally we analyzed the cone mapping implementation. It was known that with relaxed cone mapping we will need fewer steps than linear search, because this is the main advantage of the algorithm. So, it should be faster and reduce the artifacts when we perform a few steps in the search of the interval to perform the binary search. There should be fewer artifacts also because it is a safe method and linear search is not as we explained before. We tested the algorithm with 10 relaxed cone search steps and we got the best results. There were no noticeable artifacts even looking from grazing angles and as the artifacts were nearly inexistent it didn't make a difference to increase these steps to 20 steps and it had a big overhead. If we reduced the cone steps to 5 some artifacts appeared. So as it was the fastest and the one with the best visual quality we decided to keep these 10 relaxed cone steps and 8 binary search steps (as in relief mapping). We got the results showed in figure 39.

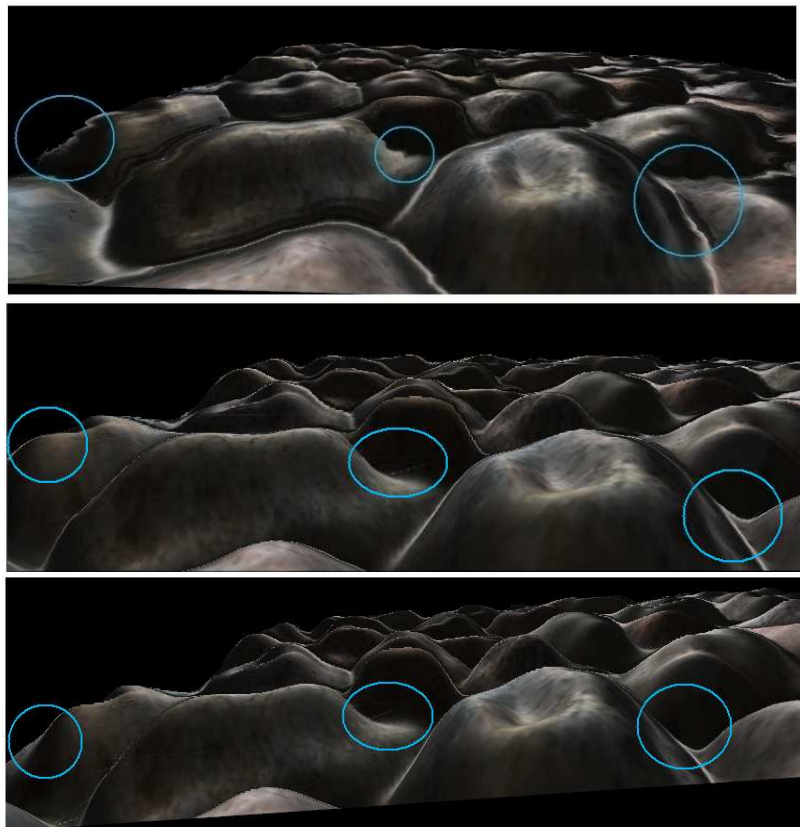


Figure 39: Image from a grazing angle with 5 relaxed cone steps (above), with 10 relaxed cone steps (middle) and 20 relaxed cone steps (below). Notice the differences between the first and the second image, and how these differences almost disappear between the second and the third one

As in relief mapping and interval mapping if we look at to the object from a non grazing angle the visualization is perfect, and there is no visible artifact. We can see in figure 40 a visualization of the same object from a non grazing angle. Notice the silhouettes that are computed just with four if, that simply look if the texture coordinates that we compute from the algorithm are between 0 and 1 as they should be.

Figure 40: Relaxed cone mapping with 10 relaxed cone steps and 8 binary search steps

The artifacts that appeared in the visualization of the objects appear also on the shadows, but as we have adjusted the values of the linear and relaxed cone step search to the values that give as a result a good view having a good frame rate this problem is reduced. In figure 41 we see one of the models recovered from the depth hallucination algorithm lit with one point light using relief mapping. Notice the hard shadows, and how they seem to be made of points because of the artifacts we have seen before. As the mapping is not completely correct some pixels will seem to be in shadow and some not in the borders of the hard shadow.

Figure 41: Shadows rendered with relief mapping

We can see here what we said before. The shadows are just hard shadows because a pixel is in shadow or in light. This is due to the fact that the light is punctual and real lights have a surface. As we can see in figure 41 these kind of shadows seem to be painted over the object instead of being the real shadows produced by the light.

In figure 42 we can see some objects rendered and lighted using interval mapping. In figures 43, 44 and 45 they are lighted using relaxed cone mapping. Notice the hard shadows and how the quality of the rendering is proportional to the quality of the shadows. If we have a good visualization we will have good shadows, without artifacts. This is due to the fact that we use the same algorithm for lighting that we used for determining the height, normal and color, and, if it fails in some points of the rendering it will also fail computing the rays from the light to the computed point.

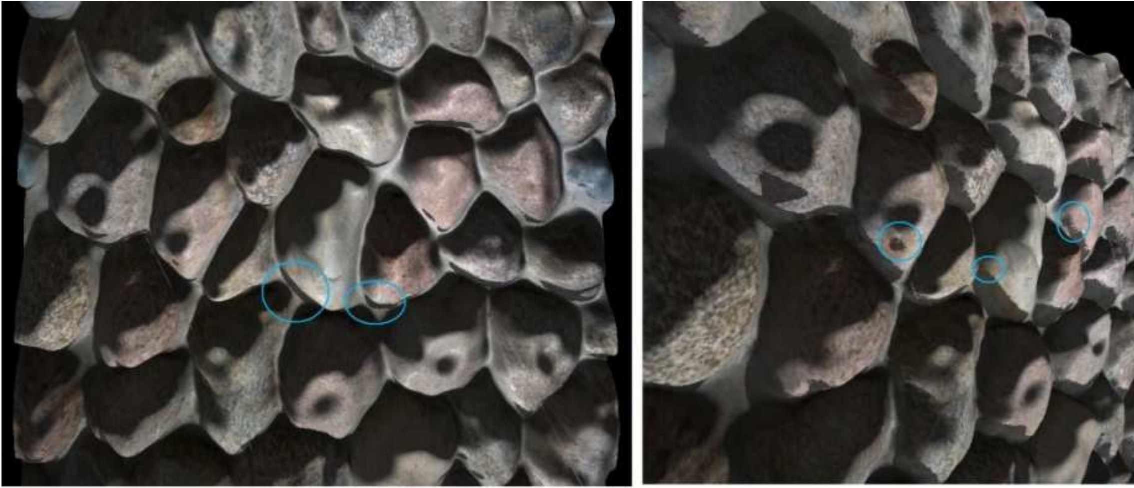


Figure 42: Images rendered using interval mapping. Notice the hard shadows (left) and the artifacts in the shadows (right)

Figure 43: Visualization of a model using interval mapping. Notice the hard shadows when we look perpendicular (above left) and the artifacts in the shadows when we look from grazing angles (above right). Notice also how the shadows are not realistic at all (down) as they seen to be points instead of a continuous shadow due to the point light.

Figure 44: Images rendered using relaxed cone mapping, notice the hard shadows and the absence of artifacts even at grazing illumination or viewing angles

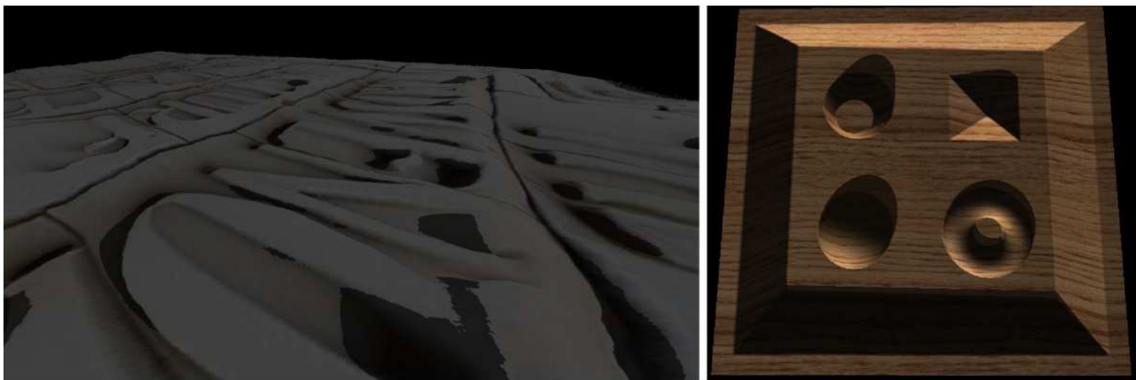


Figure 45: Figures rendered using relaxed cone mapping. Notice the difference between this image (left) and figure 40 rendered with relief mapping. Look at the hard shadows (left) and how their perfect borders break the perception of realism. Notice also how the normals affect to the illumination in the top of the semiesphere or inside the torus

5. Illumination

Now that we have implemented a complete visualization algorithm for the objects including self shadowing, the next step is to have a realistic illumination. As the single light we are using is a punctual light it casts just hard shadows and then a pixel is or lighted or completely dark, making the illumination unreal because in real shadows this doesn't happen. As real lights have a surface there are some pixels that are completely lighted, some pixels that are completely in shadow (called umbra) and some pixels that are lighted by a part of the light surface (penumbra), but not the complete light. We can see this effect as if the eye was placed in every pixel that we want to light and we trace a ray from it to every point in the light, that is not a point light (it has surface), then we can know if a pixel is in umbra if it doesn't see the light at all, lighted if we see the complete light and in penumbra if we just see a part of the light but not it's complete surface. We can see these different parts of a shadow in figure 46.

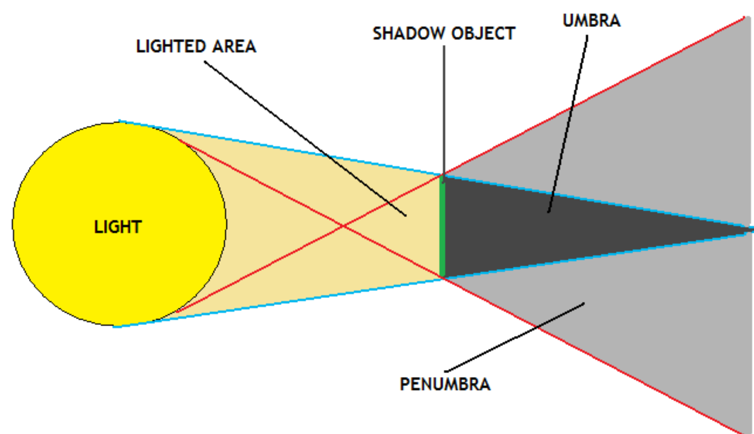


Figure 46: Parts of a shadow

As we can see the main problem with illumination is that in the real world the lights have a surface and, as we said before, in computer graphics they don't. We have to simulate a continuous surface with a discrete number of point lights that represent it. We can model a realistic lighting environment just giving some intensity values, positions, colors, etc. to a number of lights that we can create "artificially" to get the desired effect, but the main approach to simulate these environments if we want to have a realistic illumination is with 360 degrees photographs that capture the complete environment. From this environment photograph we can get the lighting characteristics using various algorithms. Once we have this photograph we can follow two main approaches to light the scene. The first one traces a number of rays from each pixel in some directions (random or not) and computes the hit of this ray with a bounding sphere textured with the environment map. Then it can compute the intensity and color of the light by reading the value of the texture in that texel. The second type of approach tries, using a preprocess step, to sample the lighting environment with a number of lights that represent all the illumination, then we map this lights to a sphere around the scene and get each light position in space.

Figure 47: Mirrored ball reflecting a complete bathroom

When we have this lights (that are point lights) mapped we just have to compute the illumination for all of them as we compute it for one.

5.1. Background

There are two main types of illumination algorithms. One deals with global illumination. It computes the illumination of one point not only taking into account the light that comes directly from the light source but also the light that comes reflected from other objects in the scene. This requires that once we have computed the ray from the light source to a point we compute several rays that come out from the point in many “random” directions to compute these reflections. As we can see it requires lots of computations that make this approach not suitable for real time rendering. There are simpler approaches that just take into account the light that the light source gives to a point. These local models are simpler and more suitable for real time (that is what we want to have in this project). We decided to implement a non-global illumination algorithm because we think that, for our purposes, it will be the best due to two main reasons. We have already implemented a model with self shadowing in relief mapping (for one point light) so we just need to make these shadows look realistic. The other reason is that we thought that, for the models we had to light (we want to focus in the fine-grain surface details whose shadows are mostly self shadows), this kind of approaches would have enough quality for our purposes and would allow us to have real time renderings easily.

Correct perception of materials requires complex, natural illumination [Fle03]. Thus for material or lighting design applications, realistic, interactive rendering of objects with arbitrary materials under natural illumination is essential.

As we have said before there are two main approaches for getting a realistic illumination from a photograph of a real world scene (image based lighting). In this section we will take a brief look to some of the techniques that have been studied before we made a decision of how best illuminate our scenes.

There are a group of methods that use a technique very similar to environment mapping. This technique traces a ray from every point in the direction of the normal and computes the intersection between this ray and the environment map that is a texture mapped on a cube (cube mapping) or a sphere (sphere mapping) that surrounds the scene representing objects far away from the scene. With this technique we can see this objects reflected on the mirrored objects of the scene. This technique is not useful for our purposes because we want to light objects with diffuse illumination, and then, we don’t have mirrors.

If we want to compute non-specular illumination using a similar technique we have to trace more than one ray, as many as we think that are enough for sampling the complete lighting environment. The first approach was just tracing rays in a random way and adding the color they match in the environment map as if they were punctual lights. The main problem of this approach is that we can trace many rays from every point, but, as they have a random direction we could never get a hit of these rays with the main light (for example the sun in a sunny day), and then the illumination will not be realistic at all. In order to solve this problem many authors have studied different ways of improving and simplifying the lighting computations.

In [CK07] they show a way to perform a filtered importance sampling. They present a technique for real-time image-based. They combine BRDF-proportional importance sampling with environment map filtering to attain computationally efficient rendering that can be implemented on GPU. There are some previous similar approaches as [KaVaHeSe00] that provide real time visualizations, but they need a huge amount of precomputation and/or a

sizable code base. The main advantage of their approach is that it requires minimal pre-computation and operates within a GPU shader fitting into almost any application needing real-time, dynamically changing materials or lighting.

The color of one pixel is given by the illumination integral (the integral of the lighting and BRDF product). Their approach to evaluating this integral on the BPU is motivated by BRDF-proportional Monte Carlo importance sampling. For each pixel, they take uniformly distributed numbers, transform them in the GPU shader into important sample directions and evaluate the samples. The uniformly distributed numbers are a precomputed deterministic set used for all pixels. These results in aliasing in their estimate of the illumination integral causing image artifacts. Monte Carlo would use random numbers to trade aliasing for more visually acceptable noise. Since generating random numbers on GPU is expensive, they reduce the artifacts via a filtering operation.

Other authors instead use spherical harmonic lighting that is a family of real-time rendering techniques that can produce highly realistic shading and shadowing with comparatively little overhead. These techniques involve replacing parts of standard lighting equations with spherical functions that have been projected into frequency space using the spherical harmonics as a basis.

In [RH01] they render diffuse objects under distant illumination, as specified by an environment map. Using an analytic expression of the irradiance in terms of spherical harmonic coefficients of the lighting they show that one needs to compute and use only 9 coefficients, corresponding to the lowest frequency modes of the illumination, in order to achieve average errors of only 1%. In other words, the irradiance is insensitive to high frequencies in the lighting and is well approximated using only 9 parameters. They show that the irradiance can be procedurally represented as a quadratic polynomial in the Cartesian component of the surface normal, and give explicit formulae. With these observations they implement a simple and efficient procedural rendering algorithm that could be implemented in hardware, a prefiltering method up to three orders of magnitude faster than previous techniques and new representations for lighting design and image based rendering.

In [KHH07] they use this approximation with 9 parameters to implement a seamless integration of synthetic objects within video images. They use a GPU-based irradiance environment map approach that constructs the environment map in real time from the images provided by a camcorder with a fish eye lens. They approximate the irradiance using this 9 parameters in order to render diffuse objects within real images at 18 ~ 20 frames per second.

In [BSS05] they use a different approach, getting some representative lighting points using a Voronoi decomposition. Radiance values of texels within the Voronoi areas are summed to compute their total radiance and the final Delaunay grid on high dynamic range image. The centers of Voronoi cells are the sample points that we could use to light our relief maps mapping them to a sphere around the scene we want to light.

5.2. HDR Images

High dynamic range imaging (HDR) involves a set of techniques that allow a greater dynamic range of luminance between light and dark areas of a scene than normal digital techniques trying to represent the big dynamic range that the human eye has. The intention of HDR is to accurately represent the wide range of intensity levels found in real scenes ranging from direct sunlight to shadows.

We will use this kind of images because we want to capture the entire range of intensities that we (our eye) capture in a real environment to use this information for lighting purposes. To be able to get the most representative lights, or even get good information of the intensity of a light from an image we must get the smaller differences that we can capture, and HDR allows us to do this.

To capture a 360º image we can take a photograph to a mirrored ball. With this image we have on the ball the complete environment reflected. Once we have this information we want to follow a process that leads us to a environment map (a texture that we can map on a sphere as if it was the environment of our scene. This environment map was the base of our computations for getting the lights that will light our relief map.

5.2.1. Environment map creation

To capture a HDR image we must capture some images with different time apertures to capture all the range of intensities that we need to use for our application. In our project we captured seven images with different apertures, as we can see in figure 48, for every high dynamic range image that we needed.

Figure 48: Example of a set of seven images used to create a HDR image

Once we have taken this photographs we are ready to create the HDR photograph. We used the software Qtpfsgui that is available for windows, linux, or mac for free. To get the HDR image from this set of images we just have to follow some simple steps. First we click on new HDR and select the seven images that will be the origin of the HDR one. Then the program will automatically adjust the exposure values (if we want we can set them manually but it showed that the automatically calculated values worked well). We click on next, and then we can apply anti-ghosting and some other techniques. We didn't apply any of this techniques in our project because it was unnecessary for our purposes. Finally the program will ask us for the steps to follow in the HDR creation. We have chosen the profile number one that uses a triangular weight function, a lineal response curve and the Paul Devebecs model for HDR creation. After all this steps, and using the images in figure 39, we get the image 49 as a result.

Figure 49: HDR image

As we can see in figure 49 there is a camera in the middle of the ball. This camera occludes part of the environment, and could make us loose an important part of the lighting conditions and change a part of the illumination. In addition, things that are reflected near the edge of the ball will become extremely stretched and distorted, giving a poor image when it is unwrapped. To alleviate these problems, we must take two pictures of the mirrored sphere from different angles and blend them together to remove the camera and the regions of poor sampling. Since the two 'bad' spots in the mirrored ball are directly towards the camera, and directly away from it, the two pictures should be taken from positions 90º apart from each other. This way the regions of bad sampling and camera interference will be in different locations in the two images. If we took this images from opposite sides of the ball will not work, as the region of bad sampling in one image will be the location of the camera in the other image and vice versa. We should have two images like the ones in figure 50.

Figure 50: 0º photograph (left) and 90º photograph (right)

In order to blend these two images and compute the light probe image from this photograph we used the HDRShop software from Paul Devebecs group that can be got for free from their web page. Now we are going to explain how to get an environment map from this two images that we see above.

The first step is to crop the images to the very edge of the mirrored ball. In order to do this we have to make sure that the "Circle" option is checked in the "Select" menu under "Draw Options". Checking this will draw an ellipse inscribed in your selection rectangle which can be useful for matching up the edge of the mirrored ball with your selection. If the mirrored ball goes off some of the edges in your photograph make sure htat the "Restrict Selection to Image" option is unchecked. This can be found in the "Select" menu under "Select Options". Unchecking this will allow the selection tool to select regions outside the bounds of the image. Select the region around the mirrored ball, and adjust it until the circle borders the edge of the ball. When you have the circle lined up, crop the image using the "Crop" command in the "Image menu".

In order to match the two images, we will need to find the rotation between them. HDRShop can do this semi-automatically, but you will need to provide the coordinates of two points (corresponding to the same features in the environment) in each mirrored ball image.

The easiest way to get these coordinates is to use the “Point Editor”, which is available under the “Window” menu (cancel out of the “Panoramic Transform” window if you have it open). Once the “Points” window is open, Ctrl-clicking on an image will create a new point. You can also drag existing points around. For our purposes, we will need two points in each image, positioned on the same features. In our example the easiest features to find are the corners of the buildings.

Now we can tell HDRShop to apply a 3D rotation to one of the panoramas in order to line it up to the other one, while simultaneously warping the images to the light probe (angular map) panoramic format. To do this we will use the Panoramic Transform command. It can be found on the “Image menu”, under “Panorama”, “Panoramic Transformations”.

The source image should be the image you wish to unwarp. The destination image can be left as *New Image*. Our source image format is *Mirrored ball*, and we wish our destination image to be in *Light Probe (Angular Map)* format. You can change the destination image resolution to the desired size, and increase the super sampling if you want a better quality result.

In this case, we will warp the image rotated 90º with no rotation, and then warp and rotate the original 0º image to match the 90º image. So we choose the 90º one as our source image, select “None” for 3D Rotation, and click OK. This should produce a warped version of this image. Then going back to the “Panoramic Transform” dialog, we can set everything up again for the original image. This time, we select “Match Points” under “3D Rotation”. Once all the fields are filled in correctly click OK for both “Match Points” and “Panoramic Transform”. You should now have two panoramas in light-probe format; something like the images in figure 51.

Figure 51: Light probes of 90º (left) and 0º (right) after warping

At this point you can turn off the points, either by deleting them or deselecting “Show Points” from the “Display” menu on the “Points” window.

The final step is to merge these images together. For this we will need a mask: an image whose pixels are 0 where we wish to use the original 0º image, 1 where we wish to use the other one and an intermediate value when we wish to blend between them. You can create a mask in any paint program, though Gimp has some nice features that make it easier.

If you are familiar with Gimp, save out JPEG versions of our two panoramas and load them into Gimp. You can copy/paste one panorama as a layer on top of the other one, and then add a mask to that layer. This way, as you paint on the mask you can see the result visually. If you don't have Photoshop, you'll just have to draw a mask and try it. When you have a good mask, save the mask out as a Windows BMP, TIFF, or other uncompressed format that HDRShop supports. Figure 52 is an example of a mask that we used to blend these images.

Figure 52: Mask used to blend the images

The next step is loading this mask into HDRShop to merge the two panoramas using it. Choose "Calculate" from the "Image" menu, and set image A and B as the panoramas we had computed before and C as the mask. Then select in operation $A * C + B * (1 - C)$ to perform an alpha blend between image B and image A using the mask. The result of this is the finished light probe. The two cameras and the regions of bad sampling have been removed, and it is ready to use as an illumination environment as we can see in figure 53.

Figure 53: Light probe after the process. Notice that there is no camera and the regions of undersampling around the image have been improved

Once we have computed this light probe we can get the environment map performing just one more step. We must go to the "Panoramic Transform" menu and load this light probe as the source image with its correspondent format. The destination image can be "New Image" and the format option allows us to select which format we want for our environment map. We can choose between the "Cubic Environment" to map it into a cube or the "Latitude-Longitude" one to map it to a sphere. As we want to do this we select this option. The sampling box allows you to choose a super-sampling pattern, as well as toggling bilinear interpolation. Increasing the super-sampling helps remove aliasing. This is especially useful if you are creating a small panorama from a big panorama. Note that the number of samples used is actually the square of the number entered; so if you enter a 3, then HDR Shop will calculate 9 samples per pixel. Bilinear Interpolation, on the other hand, helps most when you are going the other way, creating a big panorama from a small one, though it doesn't hurt to leave it on otherwise.

After all this process we get figure 54, which is the Latitude-Longitude image from our panorama.

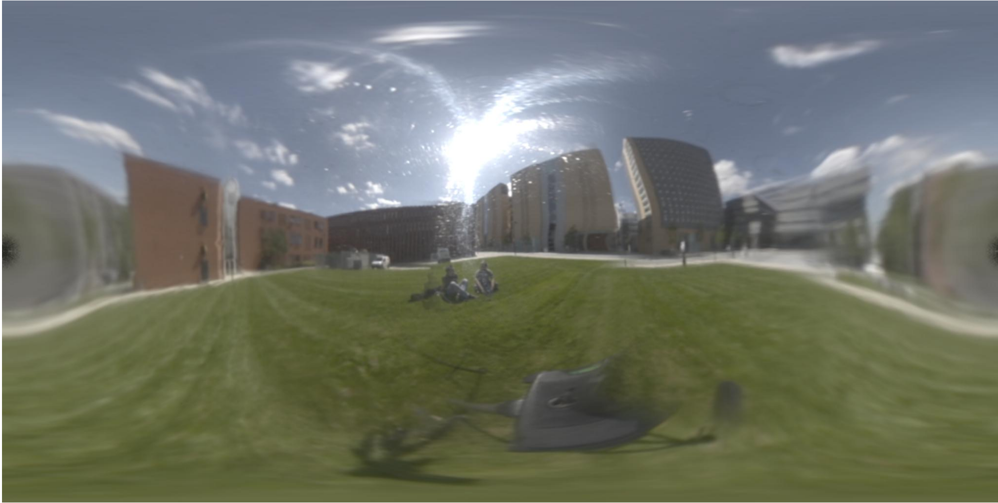


Figure 54: Final environment map prepared to be mapped on a sphere

5.3. Implementation

We decided that we wanted to use the self shadowing and the lighting we had already implemented for the relief mapping, so we decided to implement a simple algorithm with a small preprocessing that allowed us to transform a HDR image, taken from any real place (for example, the site were the real building has being captured and reconstructed), to a fixed number of lights. It is the median cut sampling algorithm [Debevec05].

5.3.1.Median cut sampling

The quality of approximating an image-based lighting environment as a finite number of point lights is increased if the light positions are chosen to follow the distribution of the incident illumination. They choose these positions by subdividing the image into regions of equal energy, giving as a result a simple and efficient algorithm that can realistically represent a complex lighting environment with as few as 64 light sources.

It is an algorithm for approximating a light probe image as a constellation of light sources based on the median cut algorithm. They take inspiration from [Heck82] color quantization algorithm to divide a light probe image in the rectangular latitude-longitude format into 2^n regions of similar light energy as follows.

First we add the complete light probe image to a region list as a single region. Then for each region in the list, they subdivide it along the longest dimension such that its light energy is divided evenly. They repeat this subdivision n times. Finally they place a light source at the center (or centroid) of each region and set the light source color to the sum of pixel values within the region.

The main advantage of this technique is that it is extremely fast compared to most other sampling techniques and produces noise-free renderings at the expense of bias inversely proportional to the number of light sources used. We have chosen it because it gives as an output a number of point lights, making it trivial to integrate this illumination with the

previous one we have implemented. We just have to iterate along the light vector and repeat the relief mapping algorithm for each light adding them at the end.

a. Implementation

We want to compute the lights position and color as faster as possible. In order to accelerate the process we will use a summed area table [Crow84]. A summed area table is an algorithm based on a special data structure for quickly and efficiently generating the sum of values in a rectangular subset of a grid. We have a table, and the value of each point (x,y) in this table will bet the sum of all the points above and to the left of it including (x,y) as we see in equation 8. Computing the summed area table is easy and fast taking into account the fact that we the value of each point depends on the value of the points on its left, up and diagonal as we can see in equation 9. Finally once we have computed the summed area table it is trivial to calculate the weight of any rectangle region using just four array references as we can see in equation 10 taking into account that the points A,B,C and D are the points in figure 55.

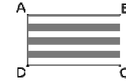


Figure 55: Rectangular area of a summed area table

$$sat(x,y) = \sum_{x' \leq x,\, y' \leq y} i(x',y') \quad (8)$$

$$sat(x,y) = i(x,y) + sat(x-1,y) + sat(x,y-1) - sat(x-1,y-1) \quad (9)$$

$$\sum_{A(x) < x' \leq C(x),\ A(y) < y' \leq C(y)} i(x',y') = sat(A) + sat(C) - sat(B) - sat(D) \quad (10)$$

Once we compute the summed area table for the 360 degrees image it is very easy to implement a fast version of the median cut sampling algorithm. Computing the total light energy is most naturally performed on a monochrome version of the lighting environment rather than the RGB pixel colors of our photograph. We can get the luminance (and the transform our RGB image to a luminance one) as a weighted average of the color channels of the light probe image. We used the weights recommended by the author (equation 11). While the partitioning decisions are made on the monochrome image, the light source colors are computed using the corresponding regions in the original RGB image.

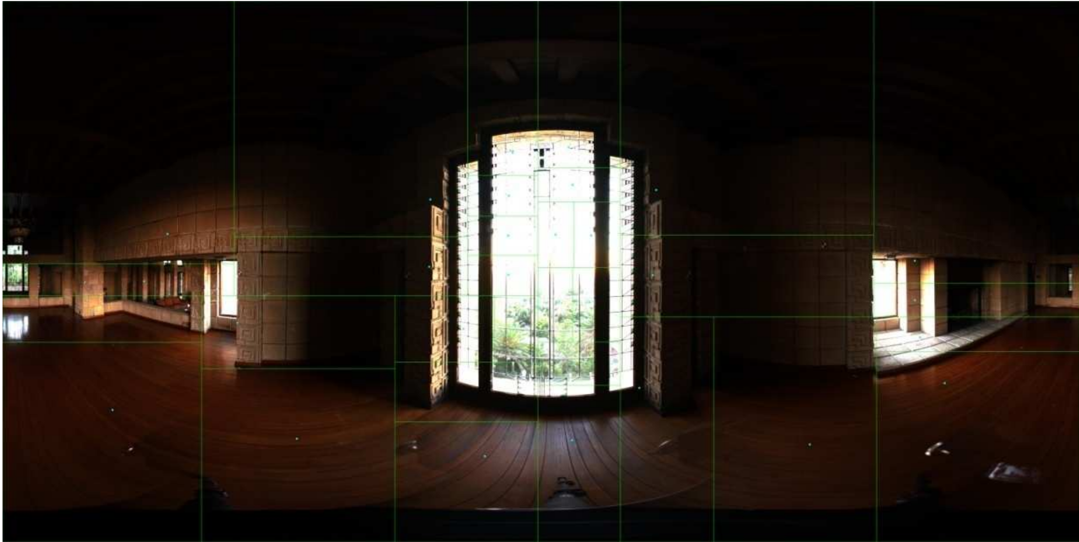
$$Y = 0.2125R + 0.7154G + 0.0721\,B \quad (11)$$

So the algorithm starts partitioning the image into regions and each region by its longer dimension. We repeat this process until the number of regions computed is the same as the number of lights we want to get as an output. It will be a multiple of two because in every step we divide every region in two.

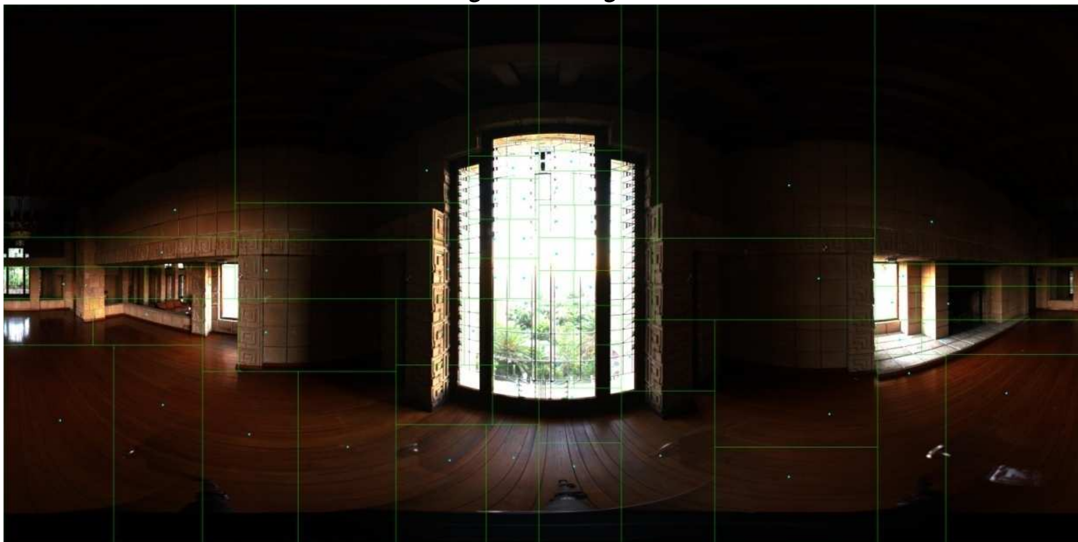
Once we have computed all the regions (we use the summed area table to compute the luminance of each region faster) we must find the real position of the point light in the space. We need to compute the centroid of every region in order to have the position of the light inside the region and then be able to map it into an sphere. We compute the centroid of each region finding the point that has at its left the same luminance as in the right, and above the same as below. To get this point we just iterate along the two dimensions of the region until we find first the horizontal (x component) component of the centroid and then

the vertical component (y component). The point that we will get after this process will accomplish the required characteristics of a centroid.

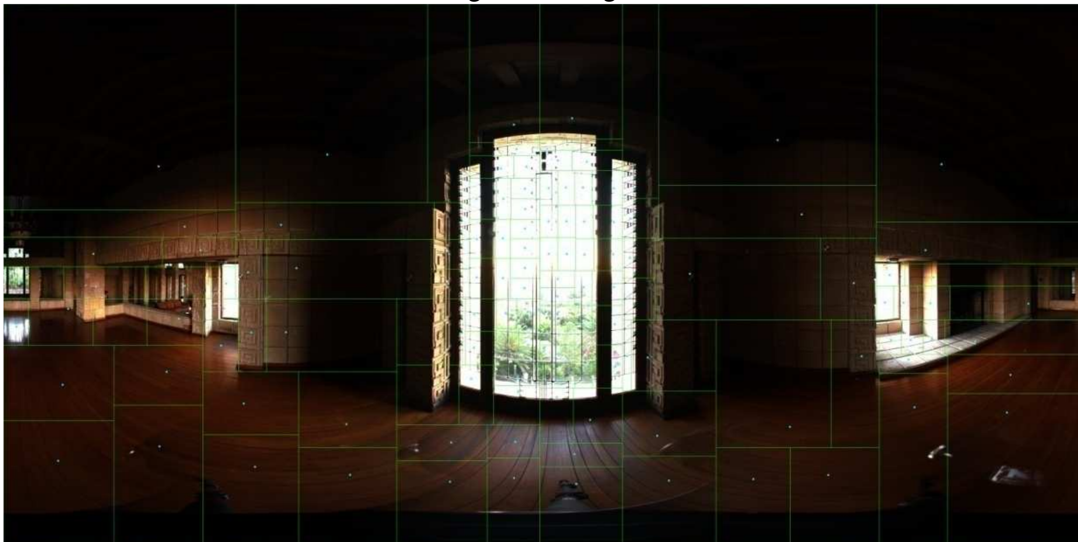
In figure 56 we show some different results for a light probe. We can see the regions that are computed with their limits in green and the light position in light blue. As we can see in all these figures the lights are mostly placed in the center of the image where the sunny window is. They are also more common around the two windows in both sides of the image. This is the effect we wanted, because as more lights are placed around a point as more important this point will be in the illumination and harder shadows will this lights cast. This effect is due to the fact that where the image is brighter the luminance that we compute as a weighted average is bigger, and then the regions are smaller because they will have as much luminance as bigger and darker ones.



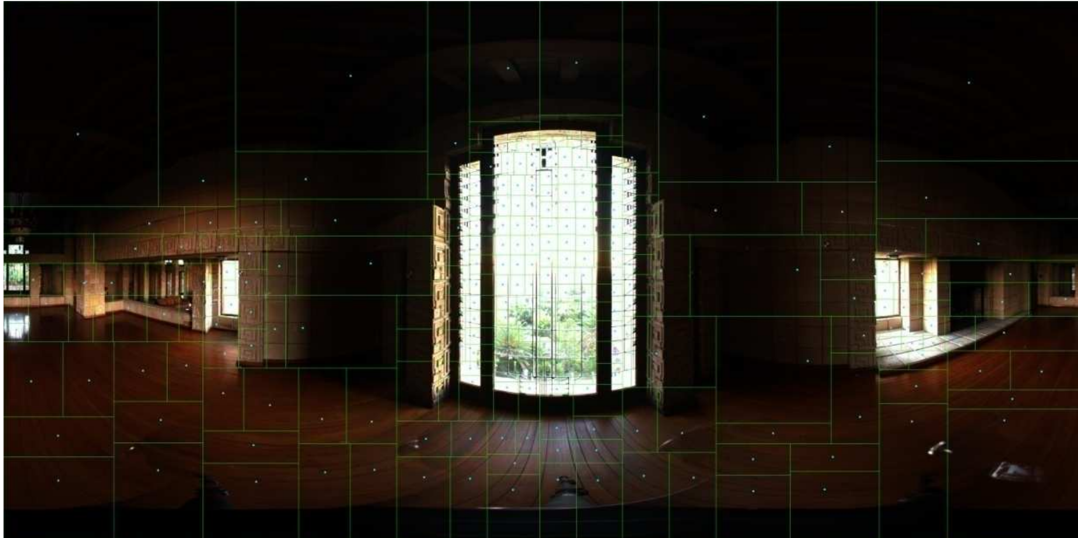
32 lights and regions



64 lights and regions

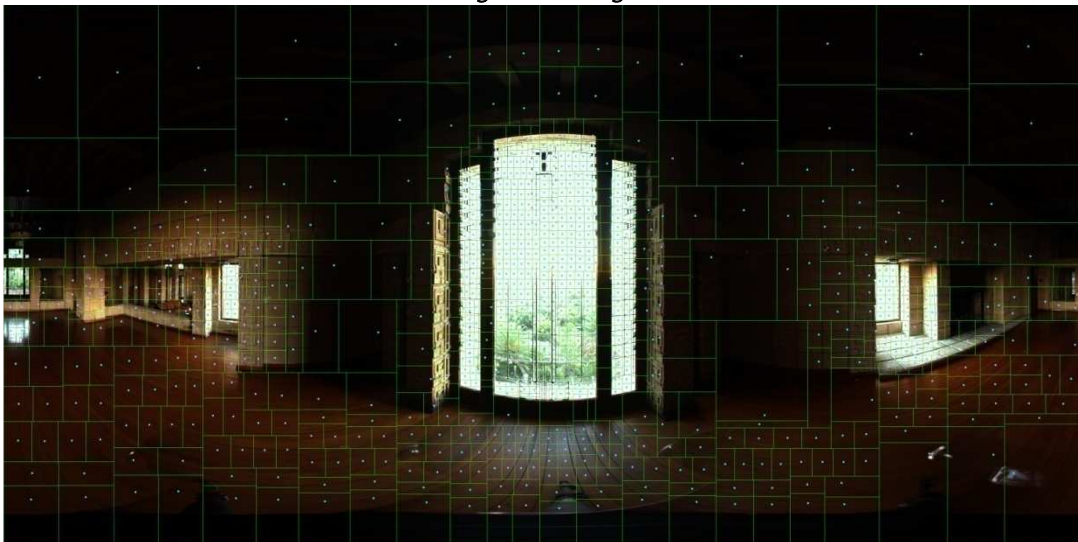


128 lights and regions



256 lights and regions

512 lights and regions



1024 lights and regions

Figure 56: Different results for the median cut sampling algorithm for a given latitud-longitude image

Once we have all the centroids in the image space we must map them to the sphere, to get the 3D point of the light position. As it is well known the sphere equation (equation 10) allows us to compute the different components of a 3D point using the mapping showed in equation 11 having a radius r and a center of the sphere(x_0, y_0, z_0). In order to be able to do this mapping we divided the horizontal and vertical number of pixels in the original image by π and $\pi/2$ respectively to compute the size in radians of a pixel. Then we are able to compute the angles φ and Θ by multiplying this value by the x and y components of the centroid respectively.

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2 \quad (12)$$

$$x = x_0 + r * \sin\theta * \cos\varphi \quad (13)$$

$$y = y_0 + r * \sin\theta * \sin\varphi \qquad (0 \leq \varphi \leq 2\pi \ and \ 0 \leq \theta \leq \pi)$$

$$z = z_0 + r * \cos\theta$$

Finally we compute the color of the light as an average of the colors of the region it represents, because we want the light to be a sample of this region and it must represent the most similar color to the color of the whole region. In the next figures we can see a cube, colored with its correspondent color, displayed where the mapped light will be in 3D. We can notice that the mapping is perfect and the lights stay in the same position with respect to the image we used to compute them that has been mapped to the sphere using the automatic Opengl mapping. We can notice in these figures how all the lights accumulate around the brighter zones of the image as windows and the sky.

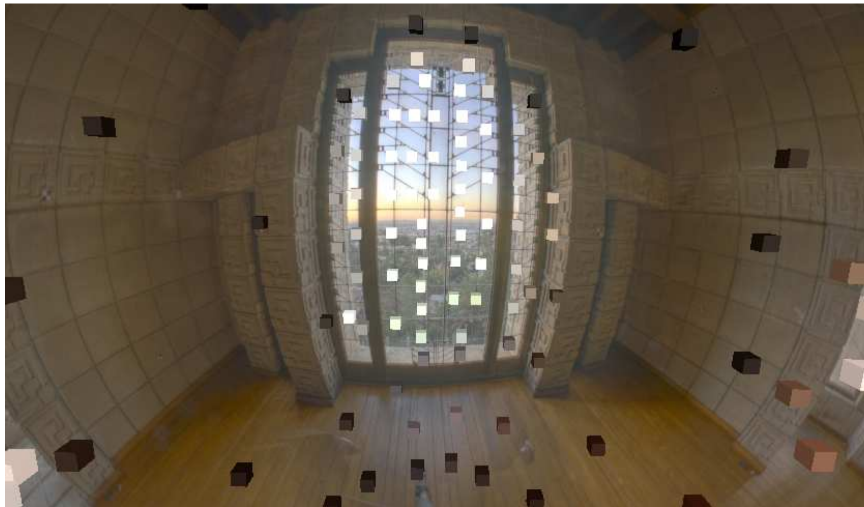


Figure 57: Mapping of 128 lights to a sphere

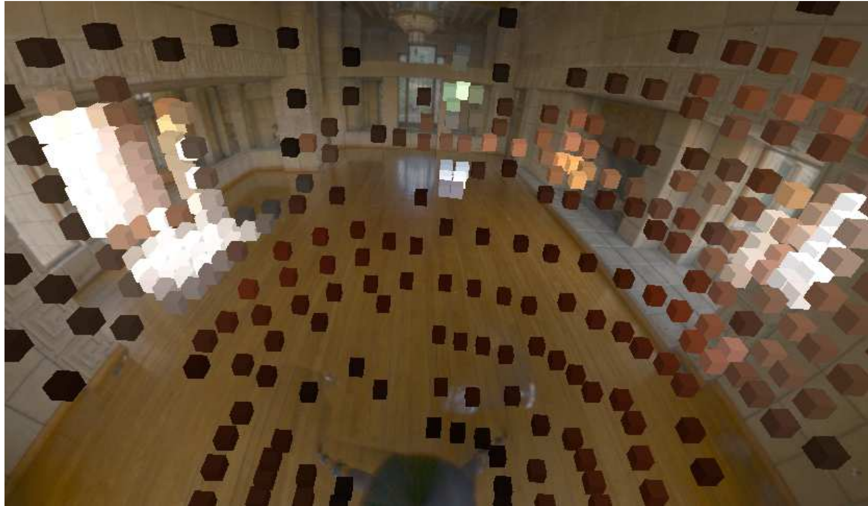


Figure 58: Mapping of 1024 lights to a sphere. Notice that there are more lights around the windows than in the rest of the room

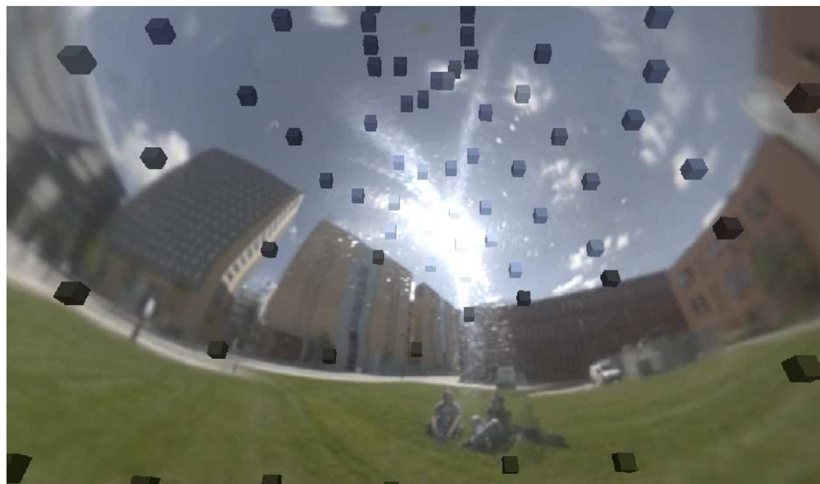


Figure 59: Mapping of 128 lights to a sphere. Notice that there are more lights in the sky than in the ground.

After all this process we have an array of light positions and colors that we must store as an output of our median cut algorithm. As we are going to use this data on GPU we were not able to store it in a common text file, we stored it as a texture to be able to access its values from the GPU. We stored in a 2*n pixels image, being n the number of lights, this information. In the first row we stored the (x,y,z) components of the 3D position in the RGB channels respectively and in the second one we stored the color of the light. To be able to store the 3D positions we first normalized the position to have the component values between -1 and 1, then we added one to the result in order to have positive values (one image stores just positive values) and finally we divided by two the result to have all the values between 0 and 1. Once we have this texture we can in GPU access in real time these values, multiply then by two and subtract one to have the normalized values of the 3D positions, then we can just scale them by the radius and we will get the light position in 3D space. In figures 60, 62 and 62 we can see the process of the algorithm from the original photograph to the final array of lights encoded as a texture.

Figure 60: Original image

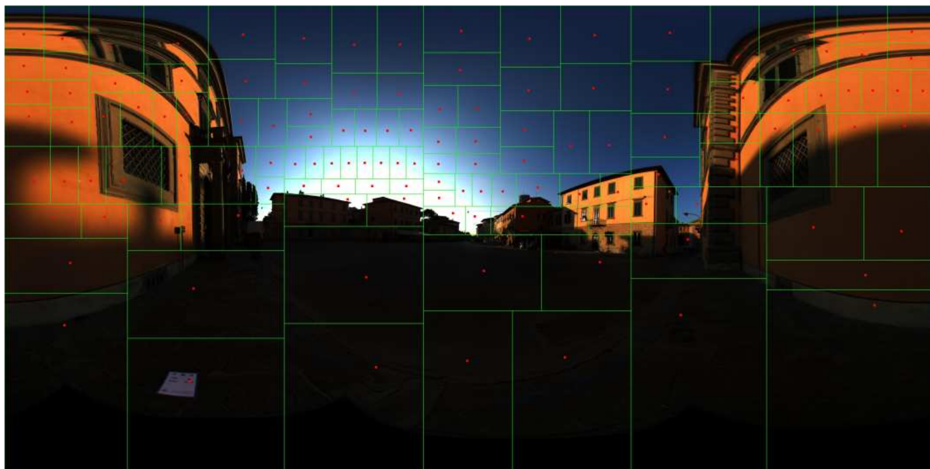


Figure 61: Processed lights and regions on the image

Figure 62: Output image encoding the position and color of the lights

5.4. Results of implementation

Once we implemented the median cut algorithm and adapted the shaders to the new illumination conditions we analyzed the results we got from our project. The first think we had to do is compare the frame rates. We wanted to see how many lights we could use with each method keeping the interactive frame rate as the main objective. We lighted an 800X800 window with a number of lights varying from 32 to 1024 with the three algorithms. We knew from the results of the relief mapping with one light that the relaxed cone mapping approach was going to be the faster, but we wanted to keep interval mapping and relief mapping to compare how the interval mapping operation improved the binary search. We also kept these algorithms because they didn't need any preprocessing and one of the purposes of our research was implementing an algorithm that allowed us to visualize the results of the Daedalus project immediately. The main idea was implementing interval mapping for a fast and poor quality visualization and relaxed cone (because of its preprocessing) for a better visualization but we had to wait for the preprocess. In other words, we wanted interval mapping to visualize if the result was correct and when we had the desired output we precomputed the relaxed cone map and visualize it with good quality using this algorithm.

Algorithm / Nº of lights	Relief mapping	Interval mapping	Relaxed cone mapping
32	136	160	250
64	70	80	128
128	34.5	40	62.5
256	17.5	20	31
512	9	10.5	15.5
1024	4.5	5.5	8.5

Table 1: Comparison between the 3 methods implemented using different numbers of lights for lighting the scene using a 800x800 window

What we can see in these results is obvious looking at the way we have implemented the algorithm. In the three algorithms if we double the number of lights the frame rate is divided by two. As we just have a loop going through the light texture as we increase the number of lights we add more steps to this loop and we have to compute the complete algorithm for lighting with one light in every step. The result shows that interval mapping is a little bit faster than relief mapping and we get interactive frame rates even with 512 lights reducing this to 5.5 with 1024 lights. If necessary we could interact with the object even using 1024 lights, but it stopped being really comfortable with 256. Relaxed cone mapping shows a very huge improvement compared to the other two methods. It gets comfortable frame rates for working even with 512 lights and interactive frame rates (we can interact with an object rendered at 8.5 frames per second) even with 1024 lights.

Taking into account the information of [Deb05] that says that we can simulate realistic lights with 64 lights using they algorithm tells us that we have achieved our result. In fact we could light without any problem our scene even using 256 lights and having real time. So, we had as a first result that the speed objective was accomplished and we needed to show that visually this algorithm really made a difference compared with the point light original algorithm and showing that it can really simulate in a good way complex (real) lighting environments.

In fact when we proved the visualization we noticed that if we used 64 lights it was enough to get a good effect in almost every model. As we were able to have real time frame rates using 128 lights we decided to use this number of lights because it allowed us to have soft shadows and realistic effects in most of the cases.

Figure 63: Detail of the shadows. Notice the difference between the interval mapping approach (right) and our approach using 64 lights(left). As we can see now we have soft shadows produced by all the lights that we are using instead of the hard original shadow creating a more realistic effect.

Another thing that we noticed is that if we changed the environment the objects changed completely. If we used an interior environment as the Ennis one the object took the red color of the windows and the reflections, but if we used open environments as the glacier or the Pisa environment it was completely different. Finally we used the one we created in previous parts of this document (the university gardens) and it also made a difference with the other environments.

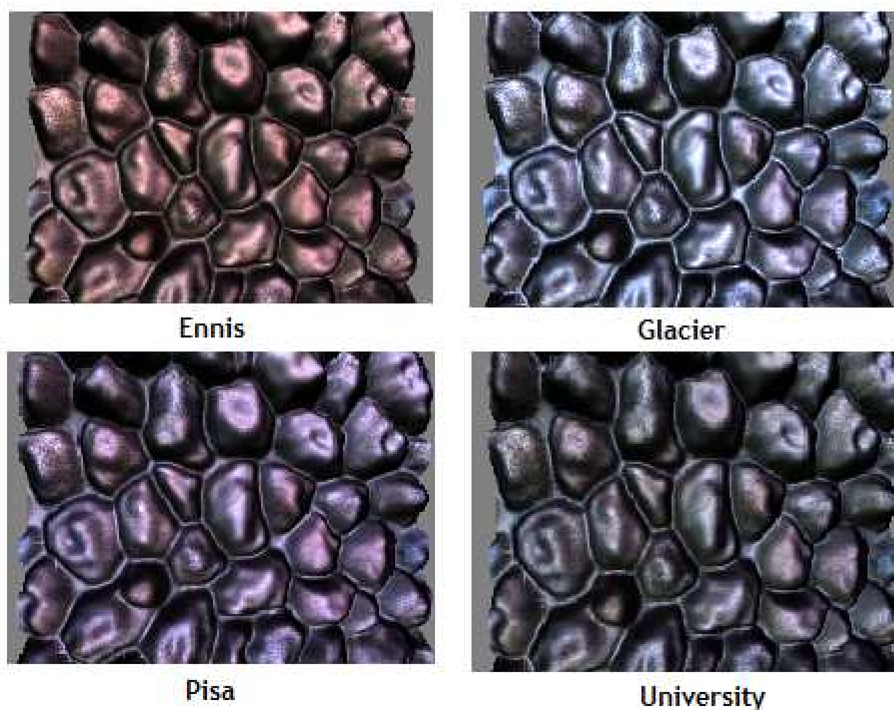


Figure 64: Rocks lighted using different light probes taken from Devebecs web and using the one we showed before how we constructed it (the garden of the university). The latitude-longitude image for these light probes can be consulted in appendix 5.

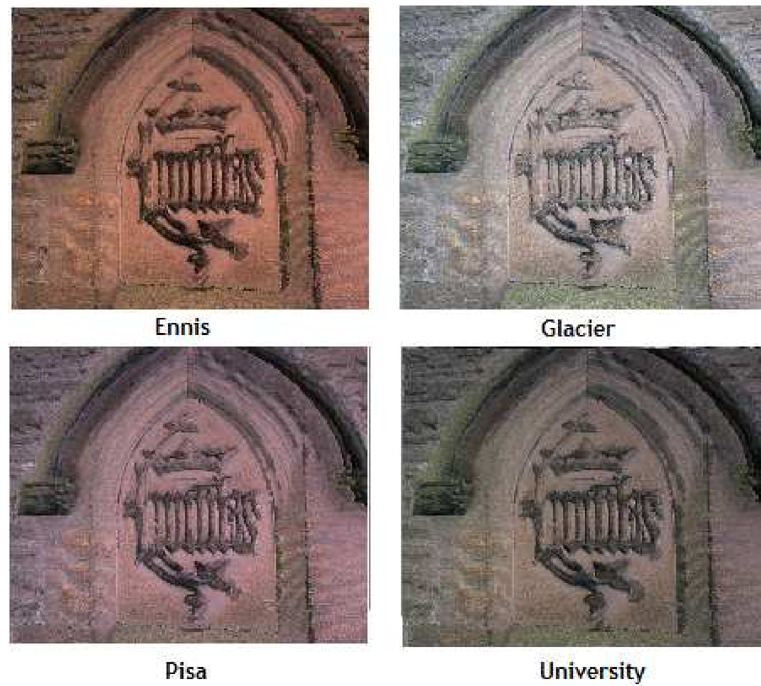


Figure 65: Reconstructed wall lighted using different light probes taken from Devebecs web and using the one we showed before how we constructed it (the garden of the university). The latitude-longitude image for these light probes can be consulted in appendix 5.

Finally we wanted to see if the shadows were correctly casted. The result is that, as we use many single lights and each light casts a shadow the result are many aggregated shadows around every hill in the object. The union of these shadows made that the resulting shadow was darker where many lights had a common space and lighter if just a few shadows were coincident. The result was that soft shadows appeared. In figure 65 we can see this effect.

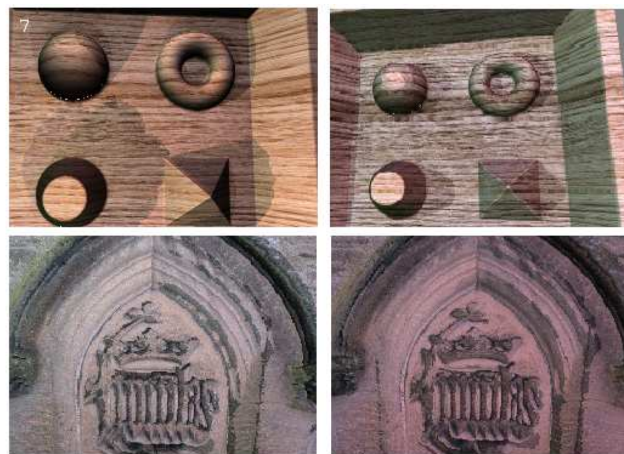


Figure 66: Objects lighted using different environments. Notice the shadows casted by the different point lights.

The final thing that we noticed is that, even using the normal to see which lights lighted a pixel and which lights didn't, the results were not always the desired ones because they were too dark or too light depending on the environment and the position of the lights with respect to the object. We fixed this problem just by adding an ambient term in case it was too dark or adding a factor to the division we computed to make the average of final color.

6. Geometry images

Once we have the relief mapped lighted in a proper way we want to extend this concept to more general models, with a more complex geometry. In order to get these results we decided to base our render on geometry images.

6.1. Background

Surface geometry is often modeled with irregular triangle meshes. The process of remeshing refers to approximating such geometry using a mesh with semi-regular connectivity, which has advantages for many graphics applications. However, current techniques for remeshing arbitrary surfaces create only semi-regular meshes. The original mesh is typically decomposed into a set of disk-like charts, onto which the geometry is parameterized and sampled. Geometry images try to remesh an arbitrary surface onto a completely regular structure. It captures geometry as a simple 2D array of quantized points. Surface signals like normals and colors are stored in similar 2D arrays using the same implicit surface parameterization. Usually, to create a geometry image, we have to cut the arbitrary mesh along a network of edge paths and parameterize the resulting single chart onto a square, but as all the models we used were just facades we didn't have this problem (they were nearly a square patch and we just took as a reference the four corners to do the parameterization). Geometry images are used for many purposes because many algorithms (or computations) are simpler to run on a good structured structure (as a 2D array). For example, in [LoHoScWa03] they use geometry images smoother models because they use a subdivision process in their algorithm and the regular structure of a geometry image is perfect for that.

One of the main problems of Geometry image is the distortion produced by the parameterization in which big triangles can be mapped in neighbor pixels and smaller ones can be mapped to a bigger piece of the image This creates distortions when we visualize the model. In [SaWoGoSny03] they try to improve the quality of Geometry images and reduce this problem by making smaller patches and using an atlas approach for geometry images, demonstrating that the distortions are sensibly reduced.

6.2. Implementation and use

The main advantage that Geometry images give for our purposes is the fact that we will have the complete model stored as an image whose pixels will store the 3D position of a vertex and will be connected with their neighbors, the texture stored with a direct mapping from it to the geometry image (every texel in the texture will be mapped to the same texel in the geometry image) and we will have a relief map (normal and depth map) with the same properties. These properties allow us to apply directly the relief mapping to the geometry image model as we have the depth map for every triangle that we sample on the geometry image (we can vary the size of the triangles to get several pixels in every triangle).

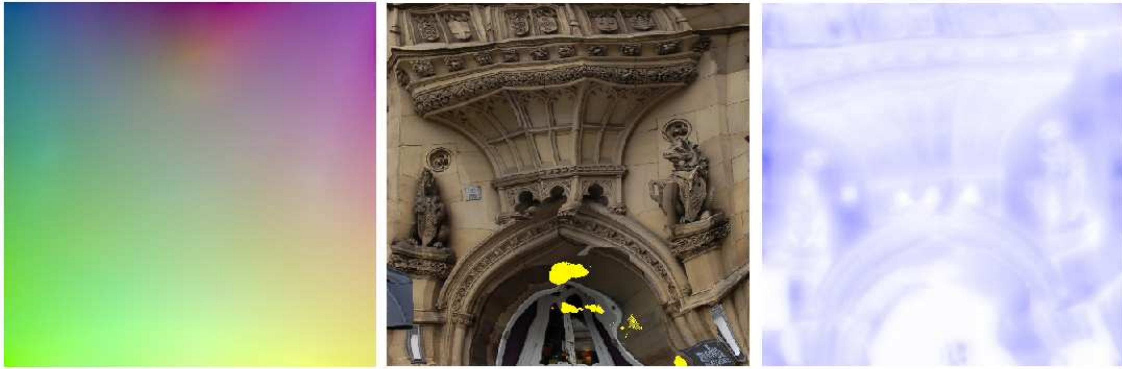


Figure 67: Geometry Image (left), texture from this geometry image (middle) and relief map computed from the texture (right)

So, the main idea of using geometry images is to have a base mesh (stored on the geometry image) that will be our macrostructure for the relief mapping, and a relief map, that has been computed from the parameterized texture using the depth hallucination algorithm to get the fine grained surface details of the model. Geometry images have another important advantage. As we reconstruct the model (that will lead us to the geometry image) from a set of images, we can select for the texture the pieces of each image where we have a better visualization of every piece of the model. This means that the visibility problem is reduced as we will get all the visible parts of the model from the different images and then we will compute over the resulting image the depth hallucination. Notice that there could be some zones of the image that could not be visible from any photograph so we can't say what is the color of the texture or the relief map in that point. In figure 66 we can see some yellow pixels in the texture, this are the pixels that are not visible from any photograph. So we have depth information of all the triangles that are visible in one photograph and we are able to apply relief to them. Once we have these three components (geometry image, texture and relief map) we just execute the same algorithm we had implemented for relief mapping.

When we visualize the model stored in the Geometry image of figure 66 we obtain the results in figure 67. In this figure we can see the model using a triangle size of 20 pixels regularly (every vertex is taken with 20 pixels difference horizontally and vertically in the geometry image) and the model with the normal map as a texture.

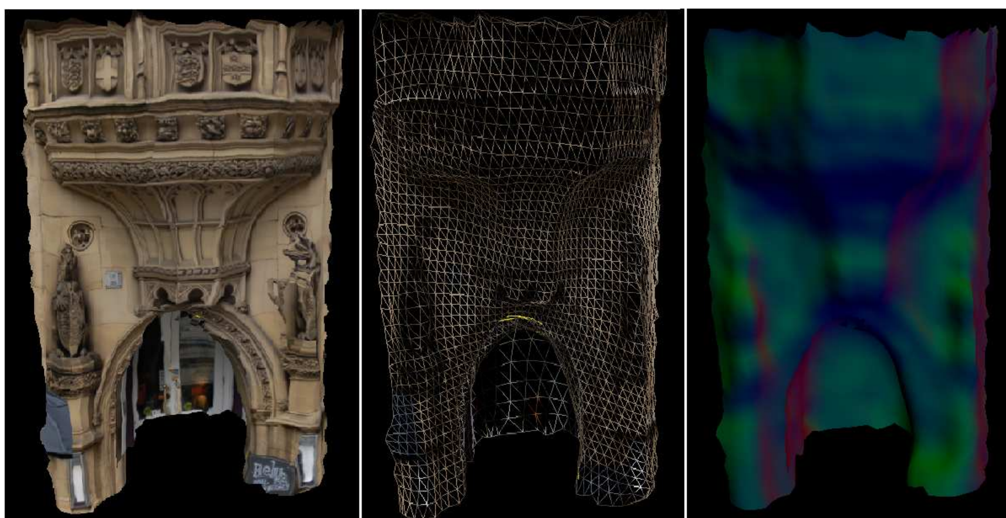


Figure 68: Textured 3D model (left), triangles taken from the Geometry image (middle) and normals of the model used as a texture (right)

The main problem that appeared to us using this approach is that using the texture taken from the parameterization to perform the depth hallucination there appear some artifacts as planar surfaces that seem to be bumpy or depth estimations that are not completely correct or clear. For some models it worked well, but for some other models it didn't make a difference ore even made them look worse.

6.3. Results of implementation and use

After implementing a program that made possible to load the geometry images as a vertex array (linking its texture coordinates and normals) we just had to substitute the usual Opengl pipeline by the shaders we implemented before. In figure 68 we can see the difference between using interval mapping and using simple texture mapping. The triangles that are painted over the texture are the triangles that are sampled in the geometry image. It's a geometry image reconstructed from the University church following all the process. First they reconstructed the macrostructure and encoded it as a geometry image. Then, they constructed the depth map using the depth hallucination algorithm on the texture that was reconstructed from all the different photographs that they used to reconstruct the macrostructure. Then we constructed the normal map from this depth map and applied the interval map shader. The result is the one you can see in figure 68.

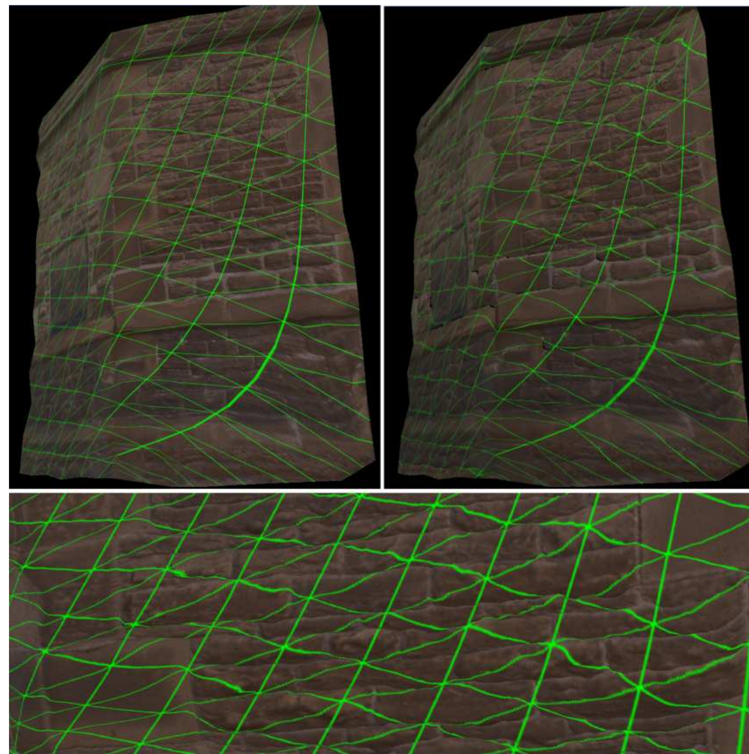


Figure 69: Comparison between texture mapping (above left) and interval mapping (above right) of a geometry image. Notice (below) how the triangles are correctly distorted by the relief of the texture in the shader

Applying the illumination algorithm many things change. The hard shadows disappear, and there are just soft shadows. In figures 69 and 70 we can see a few examples of the geometry image above lighted with different light conditions.

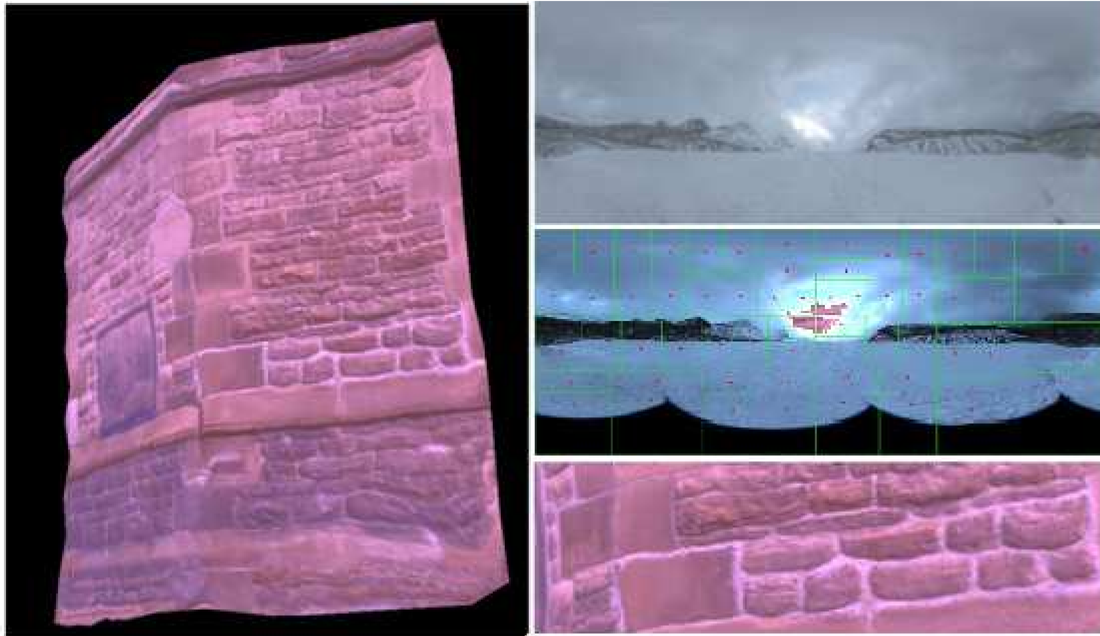


Figure 70: Wall from the University church lighted with a glacier light

Figure 71: Wall lit using two different environment maps: University gardens (left) and Ennis (right). Appendix 5

7. Conclusions and future work

We have accomplished the objectives of the project. We have made a comparison between 3 different algorithms to visualize models with fine grain surface depth using relief textures. We have studied their performance and when should it be better to use one or another. We have measured the FPS of each approach and decided to implement the three of them with the complete illumination algorithm showing that the faster (and the one with a best visualization) one needed some information to be computed in a preprocess step. We implemented interval mapping to allow the user to have an immediate visualization when the depth map is computed (not waiting for the preprocess step).

We have studied different ways to light our scene with the information taken from a light probe image. We decided to implement the median cut sampling algorithm because it is simple to implement and was easy to incorporate with our shaders. We also studied how to best construct the light probe image, capture an HDR image, erase the camera from the middle of it and solve the under sampling problems in the surround of the ball.

We have lighted simple objects (from the depth hallucination algorithm) successfully as the input had a good quality. We have also studied geometry images and how to use them for our purposes. We used them to create the vertex arrays that we sent to GPU to be painted. For some models the result satisfied our purposes, but in some models it didn't happened this way. This is due to the fact that the algorithm that computes the depth map is not completely accurate and the depth maps it computes had lots of noise transforming a planar surface into a bumpy one. The group is working now on improving these results for complete models.

We have also implemented a small program for the group to use. With it you can do all the things that have been explained in this document. Compute the normals for the relief map, use the 3 different relief mapping algorithms, compute the lights from an image, use this lights, show the geometry image with the normals or the tangent space as a texture, etc. This program will be useful for the group to visualize their models in the moment they get them from the depth hallucination algorithm. They just have to select the lighting conditions (light texture) they prefer, the model and its texture, compute the normals from the relief map and just visualize it immediately.

We have also accomplished our goal of using just free software. All the programs were written using C++ and Opengl, and for the shaders we used GLSL (the Opengl shading language). All the code was written using kate, the free text editor. We used a free image-processing library (Magick++) and Gimp for all the image transformations we needed to use. The only program that we used that needed windows was HDRshop. We used it to deal with the HDR imaging and it had just a windows version. It was free in the sense that we didn't have to pay for it.

There is much work to do in the future. We have created a base program that allows us to visualize the objects in real time, allowing us to change the lighting conditions, but we can improve something in every step of the process. The first thing to improve is the relaxed cone preprocessor performance. If must be faster if we want it to be useful. We could try a GPU implementation and study some other ways to make it faster. We could also improve the lighting algorithm by using a global lighting approach on GPU, but this will be slower than the one we have now. It could be useful for getting some renders of higher quality than the ones we have now, but not for having real time frame rates.

Another thing that could improve the cone map shader is to implement the interval mapping approach for substituting the binary search. It has not been implemented because the difference won't be very noticeable, but in case it was necessary to improve some frames per second it could be a useful approach.

We must improve also the way we sample the geometry images. We just map them in a continuous way, just linearly saying how many pixels are between every vertex and then getting two equilateral triangles for every square with this number of pixels of side. We could study a way of sampling them in a more proper way, using the shape of the model to simplify it and get the triangles as big as possible without loosing much information from the original model taking into account that it is a macrostructure model and we can loose some information that could be replaced by the mesostructure encoded in the relief map.

Finally the application user interface should be improved to make it easier for a user to use it without thinking about how to use it instead of what he wants to do.

8. Acknowledgements

I would like to thank my father, his partner and my mother for their continuous support and aid with all the problems I have had during the process of working on this project.

I also want to thank Roger Hubbold for his guide, support and for the discussions we had concerning my project and other aspects of my life, because he made the experience of working in their group in Manchester an unforgettable time in my life.

I want to thank Isabel Navazo for her support from Spain guiding me with this document and dealing with all the administrative issues. Thanks for making this experience possible.

I also want to thank Francho Meléndez for the great times we spent on the office working and talking about graphics, future and life. For his continuous help and guide and all the support he gave to me during the project. Without his help this project wouldn't be here today.

Thanks to Ridwan Ibn Sa'id for his help with GPU programming, his pieces of code that made my life easier and for showing to me that programming on GPU is not as difficult as it could seem to be. Salil Deena and Shaobo Hou for the great times in lunch and the office discussing about university and all the problems we had with our respective projects.

Finally, thanks to all the people that made possible the pleasant experience of working with them in this project.

References

[BSS05] Barsi A., SZIRMAY-KALOS L., SCECSI L. 2005. Image-based Illumination on the GPU. In Machine Graphics & Vision International Journal 2005: 159 - 169

[Bli78] BLINN J. F. 1978. Simulation of wrinkled surfaces. In Computer Graphics (SIGGRAPH '78 Proceedings) , pp. 286–292.

[CK07] COLBERT M. , KRIVANEK J. 2007. Real-time Shading with Filtered Importance Sampling. In Proc. of SIGGRAPH 2007,ACM Press.

[CROW84] CROW F. C. 1984. Summed-area tables for texture mapping. In Computer Graphics(Proceedings of SIGGRAPH 84), vol. 18, 207–212.

[Dum06] DUMMER J., 2006. Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm. Tech. rep.

[FDA03] FLEMING R. W., DROR R. O., ADELSON E. H. 2003. Real-world illumination and the perception of surface reflectance properties. In Journalof Vision 3 (July), 347–368.

[GGH02] GU X., GORTLER S. J., HOPPE H. 2002. Geometry Images. In ACM Transactions on Graphics 2002.

[GWJ08] GLRENCROSS M., WARD J. G., JAY C. 2008. A Perceptually Validated Model for Surface Depth Hallucination, In ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.

http://www.debevec.org/

http://developer.nvidia.com/page/home.html

http://www.hdrshop.com/

http://www.imagemagick.org

http://www.lighthouse3d.com

http://www.opengl.org/documentation/glsl/

[KHH 07] KIM J., HWANG Y., HONG H. 2007. Using Irradiance Environment Map on GPU for Real-Time Composition. In PCM 2007: 704-713

[KKI 01] KANEKO T., KAKAHEI T., INAMI M., KAWAKAMI N., YANAGIDA Y., MAEDA T., TACHI S. 2001. Detailed shape representation with parallax mapping. In Proceedings of ICAT 2001 , pp. 205-208.

[KVHS00] KAUTZ J., VAZQUEZ P.-P., HEIDRICH W., SEIDEL H.-P. 2000. A unified approach to prefiltered environment maps. In 11th Eurographics Workshop on Rendering, Eurographics Association, 185–196.

[LHSW03] LOSASSO F., HOPPE H., SCHAEFER S., WARREN J. 2003. Smooth geometry images. In Eurographics Symposium on Geometry Processing 2003, 138-145

[LT 08] LAZLO S-K., TAMAS U. 2008. Displacement Mapping on the GPU, State of the Art. In Computer Graphics Forum, Volume 27, Number 6, pp. 1567-1592.

[LZ94] LANGER M. S., ZUCKER S. W.: Shape-from-shading on a cloudy day. Journal of the Optical Society of America 11,2, 467-478 (1994).

[Lev90] LEVOY M. 1990. Efficient ray tracing of volume data. In ACM Transactions on Graphics 9, 3, 245–261.

[MM05] MCG UIRE M., MCGUIRE M. 2005. Steep parallax mapping. In I3D 2005 Poster .

[OBM00] OLIVEIRA M. M., BISHOP G., MCALLISTER D. 2000. Relief texture mapping. In SIGGRAPH 2000 Proceedings, pp. 359–368.

[Oli00] OLIVEIRA M. M. 2000. Relief Texture Mapping. In PhD thesis, University of North Carolina.

[PO05] POLICARPO F., OLIVEIRA M. M. 2005. Rendering surface details with relief mapping. In ShaderX4: Advanced Rendering Techniques, Engel W., (Ed.). Charles River Media.

[PO07] POLICARPO F., OLIVEIRA M. M. 2007. Relaxed cone stepping for relief mapping. In GPU Gems 3, Nguyen H., (Ed.). Addison Wesley.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games , pp. 155–162.

[Pre06] PREMECZ M.: Iterative parallax mapping with slope information. In Central European Seminar on Computer Graphics (2006).

[RH01] RAMAMOORTHI R., HANRAHAN P. 2001. An efficient representation for irradiance environment maps. In Proc. of SIGGRAPH 2001,ACM Press.

[RSP06] RISSER E. A., SHAH M. A., PATTANAIK S.: Interval mapping. In Symposium on Interactive 3D Graphics and Games (I3D) (2006).

ROST R.J, KESSENICH J. M. 2006. Opengl Shading language. Ed. Addison-Wesley

[RWPD05] REINHARD E.,WARD G., PATTANAIK S., DEBEVEC P. 2005. High dynamic range imaging: acquisition, display and image based lighting.Ed. Elsevier.

[SJW04] STUMPFEL J., JONES A., WENGER A. 2004. Direct hdr capture of the sun and sky. In Afrigraph 2004

[SSS06] SZIRMAY-KALOS L., SCECSI L., SBERT M. 2006. GPUGI: Global Illumination Effects on the GPU. Eurographics 2006 Tutorial.

[SWGS03] SANDER P., WOOD Z., GORTLER S., SNYDER J. 2003. Multi-Chart Geometry Images. In Eurographics Symposium on Geometry Processing *2003*, 146-155

[SWN] SHREINER D., WOO M., NEIDER J., DAVIS T. 2004. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2. Ed. Addison-Wesley

[Tat06] TATARCHUK N. 2006. Dynamic parallax occlusion mapping with approximate soft shadows. In Proceedings of the 2006 symposium on Interactive 3D graphics and games 2006, 63 - 69

[Wel04] WELSH T. 2004 Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces. Tech. rep., Infiscape Corporation.

[Wil78] WILLIAMS L. 1978. Casting curved shadows on curved surfaces. In Siggraph 1978, Computer Graphics Proceedings, 270–274.

[YJ04] YEREX K., JAGERSAND M. 2004. Displacement mapping with ray-casting in hardware. In Siggraph 2004 Sketches.

Appendices

Appendix 1: Normal mapping computation

We have implemented a class that computes the normal map from the relief map. Here we show the main method that, for a given pixel, computes its normal using the height of the surrounding pixels. Other methods and information about this class can be consulted at the application folder under “Files” and then “Relief map” folders.

```
Color ReliefMap::computeColor(uint i,uint j,Image img){
        ColorRGB c,col,cp;
        double red,green,blue,a,tl,l,bl,t,b,tr,r,br,tmp;
        red=green=blue=tl=l=bl=t=b=tr=r=br=0;
        cp=img.pixelColor(i,j);
        tmp=cp.alpha();
        if(!(i==0 || j==0)){
           c=img.pixelColor(i-1,j-1);  //top left
           tl=c.alpha();
        }
        else
         tl=tmp;
        if(!(i==0)){
           c=img.pixelColor(i-1,j);   //left
           l=c.alpha();
        }
        else
         l=tmp;
        if(!(i==0 || j==img.baseRows()-2)){
           c=img.pixelColor(i-1,j+1);  //botton left
           bl=c.alpha();
        }
        else
         bl=tmp;
        if(!(j==0)){
           c=img.pixelColor(i,j-1);   //top
           t=c.alpha();
        }
        else
         t=tmp;
        if(!(j==img.baseRows()-2)){
           c=img.pixelColor(i,j+1);   //bottom
           b=c.alpha();
        }
        else
         b=tmp;
        if(!(i==img.baseColumns()-2 || j==0)){
           c=img.pixelColor(i+1,j-1); //top right
           tr=c.alpha();
        }
        else
         tr=tmp;
        if(!(i==img.baseColumns()-2)){
           c=img.pixelColor(i+1,j);   //right
           r=c.alpha();
```

```
        }
        else
          r=tmp;
        if(!(i==img.baseColumns()-2 || j==img.baseRows()-2)){
           c=img.pixelColor(i+1,j+1);  //bottom right
           br=c.alpha();
        }
        else
          br=tmp;
  // Compute dx using Sobel:
  //        -1 0 1
  //        -2 0 2
  //        -1 0 1

  float dX = tr + 2*r + br -tl - 2*l - bl;

  // Compute dy using Sobel:
  //        -1 -2 -1
  //         0  0  0
  //         1  2  1

  float dY = bl + 2*b + br -tl - 2*t - tr;

  // Build the normalized normal
  c=img.pixelColor(i,j);
  a=c.alpha();
  red=dX*a;
  green=dY*a;
  blue=1.0f;
  tmp=sqrt((red*red)+(green*green)+(blue*blue));
  red= red/tmp;
  green= green/tmp;
  blue= blue/tmp;
  col.red((red + 1.0f)/2);
  col.green((green + 1.0f)/2);
  col.blue((blue + 1.0f)/2);
  col.alpha(1-a);
 // printColor(col);
  return col;

}
```

Appendix 2: Relief mapping shaders

Vertex Shader

```
attribute vec3 tangent;            //Tangent of the macrostructure
attribute vec3 binormal;           //Binormal of the macrostructure

varying vec2 texCoord;             //Texture coordinates from the texture mapping
varying vec3 eyeSpaceVert;         //Vertex coordinates in eye space
varying vec3 eyeSpaceTangent;   //Tangent in eye space
varying vec3 eyeSpaceBinormal;          //Binormal in eye space
varying vec3 eyeSpaceNormal;   //Normal in eye space
varying vec3 eyeSpaceLight;       //Light in eye space

void main(void)
{
  eyeSpaceVert = (gl_ModelViewMatrix * gl_Vertex).xyz;          //Transform vertex coordinates
to eye space
  eyeSpaceLight = (gl_ModelViewMatrix * vec4(gl_LightSource[0].position.xyz,1.0)).xyz;
//Transform light coordinates to eye space
  eyeSpaceTangent  = gl_NormalMatrix * tangent;        //Transform the tangent to eye space
  eyeSpaceBinormal = gl_NormalMatrix * binormal;      //Transform the binormal to eye space
  eyeSpaceNormal   = gl_NormalMatrix * gl_Normal;     //Transform the normal to eye space
  texCoord   = gl_MultiTexCoord0.xy;              //Get the texture coordinates
  gl_Position = ftransform();                            //Transform the vertex position
}
```

Fragment Shader

```
uniform vec4 ambient;   //Ambient component for lighting
uniform vec4 specular;  //Specular component for lighting
uniform vec4 diffuse;   //Diffuse component for lighting
uniform float shine;      //Shininess of the material
uniform float depth;      // Depth scale

varying vec2 texCoord;
varying vec3 eyeSpaceVert;
varying vec3 eyeSpaceTangent;
varying vec3 eyeSpaceBinormal;
varying vec3 eyeSpaceNormal;
varying vec3 eyeSpaceLight;

uniform sampler2D reliefmap; //Relief map
uniform sampler2D texmap;      //Texture map

float ray_intersect_rm(in vec2 dp,in vec2 ds);

void main(void)
{
  //Variables for further work
  vec4 t,c;
```

```
vec3 p,v,l,s;
vec2 dp,ds,uv;
float d,a;

// Ray intersect in view direction
//Transform to tangent space
p  = eyeSpaceVert;
v  = normalize(p);
v  = normalize(p);
a  = dot(eyeSpaceNormal,-v);
s  = normalize(vec3(dot(v,eyeSpaceTangent),dot(v,eyeSpaceBinormal),a));
s  *= depth/a;
dp = texCoord;
ds = s.xy;
//Find the intersection
d  = ray_intersect_rm(dp,ds);

// Get relief map and color texture points
uv=dp+ds*d;

//If it is out of the bounds of the texture we discard the pixel (we get the silhouette)
if (uv.x < 0.0) discard;
if (uv.x > 1) discard;
if (uv.y < 0.0) discard;
if (uv.y > 1) discard;
t=texture2D(reliefmap,uv); //Relief data
c=texture2D(texmap,uv);          //Color data

//Expand normal from normal map in local polygon space
t.xy=t.xy*2.0-1.0;
t.z=sqrt(1.0-dot(t.xy,t.xy));
t.xyz=normalize(t.x*eyeSpaceTangent+t.y*eyeSpaceBinormal+t.z*eyeSpaceNormal);

//Compute light direction
p += v*d*a;
l=normalize(p-eyeSpaceLight.xyz);

//Ray intersect in light direction
dp+= ds*d;
a  = dot(eyeSpaceNormal,-l);
s  = normalize(vec3(dot(l,eyeSpaceTangent),dot(l,eyeSpaceBinormal),a));
s *= depth/a;
ds = s.xy;
dp-= ds*d;
float shadow = 1.0;

vec3 specular_shadow=specular.xyz;
//Get the intersection of the light ray and the relief map
  float dl = ray_intersect_rm(dp,s.xy);
   if (dl<d-0.05) // if pixel in shadow
   {
      shadow=dot(ambient.xyz,vec3(1.0))*0.333333;
```

```
      specular_shadow=vec3(0.0);
    }


  //Compute diffuse and specular terms
  float att=max(0.0,dot(-l,eyeSpaceNormal));
  float diff=shadow*max(0.0,dot(-l,t.xyz));
  float spec=max(0.0,dot(normalize(-l-v),t.xyz));

  //Compute final color
  vec4 finalcolor;

finalcolor.xyz=ambient.xyz*c.xyz+att*(c.xyz*diffuse.xyz*diff+specular_shadow*pow(spec,shin
e));
  finalcolor.w=1.0;
  gl_FragColor = vec4(finalcolor.rgb,1.0);
}

/*Returns height of the intersection between the ray
(defined by the origin dp, and the normalized vector ds) and the heighfield*/

float ray_intersect_rm(in vec2 dp, in vec2 ds)
{
  float depth_step=1.0/float(10.0);
  // Current size of search window
  float size=depth_step;
  // Current depth position
  float depth=0.0;
  // Best match found (starts with last position 1.0)
  float best_depth=1.0;
  //Linear search
  for( int i=0;i<10-1;i++ )
  {
    depth+=size;
    vec4 t=texture2D(reliefmap,dp+ds*depth); //Get the relief information

    if (depth>=t.w)       //If the point is under the relief map
      best_depth=depth;   // store best depth
    else break;
  }
  depth=best_depth;
  //Binary search
  for( int i=0;i<8;i++ )  {
    size*=0.5; //Middle of the interval
    vec4 t=texture2D(reliefmap,dp+ds*depth);
    if (depth>=t.w) {      //We change the lower bound
      best_depth=depth;
      depth-=2.0*size;
    }
    depth+=size;
  }
  return best_depth;}
```

Appendix 3: Interval mapping shaders

These shaders are the same as relief mapping, but we change the way we compute the intersection between the ray and the reliefmap, so we just change the ray_intersect_rm computation by the following.

```
float ray_intersect_rm(in vec2 dp, in vec2 ds)
{
  float depth_step=1.0/float(12.0);
  // Current size of search window
  float size=depth_step;
  // Current depth position
  float depth=0.0;
  // Best match found (starts with last position 1.0)
  float best_depth=1.0;
  //Initialization of the bounds of the interval
  float lower_h=0.0,upper_h=0.0,lower_d=0.0,upper_d=0.0,last_h=0.0;
   // Linear search for first point inside object (first hit)
   for( int i=0;i<10-1;i++ )
   {
     depth+=size;
     vec4 t=texture2D(reliefmap,dp+ds*depth);
     if (depth>=t.w){ //If the point is below the relief map
       lower_h=last_h;
       //Get the lower an upper bound
       lower_d=(ds.x*(depth-size))/ds.y;
       upper_h=t.w;
       upper_d=(ds.x*depth)/ds.y;
       best_depth=depth;   // store best depth
       break;
     }
     last_h=t.w;
   }

    //Interval mapping
     vec4 t;
     t.w=1.0;
     //Compute the slope and start the interval mapping iteration
     float int_depth=0.0;
     float view_slope=ds.y/ds.x;
     for (int i=0;(i<3)&&(abs(t.w-int_depth) > 0.0001);i++)
     {
       float line_slope =(upper_h-lower_h)/(upper_d-lower_d);
       float inter_pt= (upper_h-line_slope*upper_d)/(view_slope-line_slope);
       int_depth=view_slope*inter_pt;
       t=texture2D(reliefmap,dp+ds*int_depth);
       if (t.w < int_depth)
       {  //new upper bound
         upper_h = t.w;
         upper_d=inter_pt;
         best_depth=upper_h;
```

```
      }
      else
      {  //new lower bound
        lower_h=t.w;
        lower_d=inter_pt;
        best_depth=lower_h;
      }
    }
  return best_depth;
}
```

Appendix 4: Relaxed cone mapping shaders

Vertex Shader

```
attribute vec3 tangent;
attribute vec3 binormal;

varying vec2 texCoord;
varying vec3 eyeSpaceLight;
varying vec3 eye;
varying vec3 eyeSpaceTangent ;          //tangent;
varying vec3 eyeSpaceBinormal;          //binormal;
varying vec3 eyeSpaceNormal;

void main(void)
{

  // Location of the vertex in eye space
  vec3 eyeSpaceVert = (gl_ModelViewMatrix * gl_Vertex).xyz;

  // Convert to eyeSpace the light position
  eyeSpaceLight = (gl_ModelViewMatrix*gl_LightSource[0].position).xyz;

  //  We store the matrix needed to convert to eye space
   eyeSpaceTangent  = normalize(gl_NormalMatrix * tangent.xyz);        //tangent;
   eyeSpaceBinormal = normalize(gl_NormalMatrix * (binormal.xyz));    //binormal;
   eyeSpaceNormal   = normalize(gl_NormalMatrix * gl_Normal); //normal

  //Convert the eye to tangent space
  eye = vec3 (
      dot (eyeSpaceTangent, eyeSpaceVert),
      dot (eyeSpaceBinormal, eyeSpaceVert),
      dot (eyeSpaceNormal, eyeSpaceVert));
  //Get texture coordinates from texture mapping
    texCoord   = gl_MultiTexCoord0.xy;
    gl_Position = ftransform();
}
```

Fragment Shader

```
uniform vec4 ambient;
uniform vec4 specular;
uniform vec4 diffuse;
uniform float shine;
uniform float depth;
uniform sampler2D reliefmap; //Relaxed cone map
uniform sampler2D texmap;


varying vec2 texCoord;
varying vec3 eyeSpaceLight;
varying vec3 eye;
```

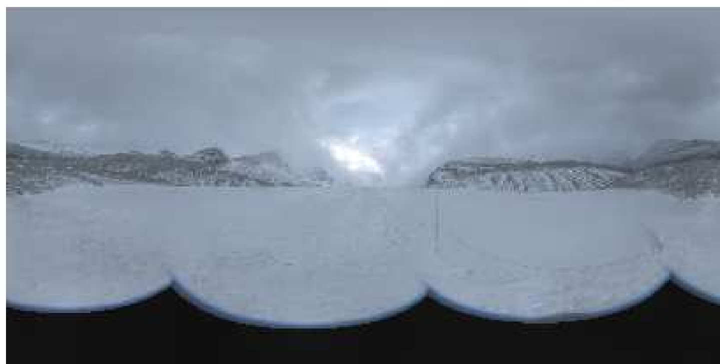
```
varying vec3 eyeSpaceTangent ;            //tangent;
varying vec3 eyeSpaceBinormal;            //binormal;
varying vec3 eyeSpaceNormal;


void setup_ray(out vec3 p,out vec3 v,in vec3 eyelight,in vec2 tex);
void ray_intersect_relaxedcone(inout vec3 p,inout vec3 v);
vec4 lighting(in vec3 texcoord);


void main(void)
{  vec3 p2,v2;
  depth=0.075;
  //Setup the ray to compute the intersection
  setup_ray(p2,v2,eye,texCoord);

  //Compute the intersection between the viewing ray and the reliefmap
  ray_intersect_relaxedcone(p2,v2);

  //Light the pixel
  gl_FragColor = lighting(p2);
}

// Setup ray pos and dir that we will use to compute the
// ray/relief map intersection based on view vector

 void setup_ray(out vec3 p,out vec3 v,in vec3 eyelight,in vec2 tex)
 {
  float a  = -depth / eyelight.z;
  v = eyelight * a;
  v.z = 1.0;
  p= vec3 (texCoord, 0.0);
 }


// Ray intersect depth map using binary cone space leaping
// depth value stored in alpha channel (black is at object surface)
// and cone ratio stored in blue channel

void ray_intersect_relaxedcone(inout vec3 p,inout vec3 v)
{
        const int cone_steps=10;
        const int binary_steps=8;

        vec3 p0 = p;
        v /= v.z;
        float dist = length(v.xy);

        //Use the cone map to  find the first intersection
        for( int i=0;i<cone_steps;i++ ){
                vec4 tex = texture2D(reliefmap, p.xy);
```

```
            float height = tex.w - p.z;
            if(height>0.0){//if we are above the cone map
              float cone_ratio = tex.z;
              p += v * (cone_ratio * height / (dist + cone_ratio));
            }
            else //If it is below we have find the intersection
              break;
        }
        v *= p.z*0.5;
        p = p0 + v;
   //Binary search
        for( int i=0;i<binary_steps;i++ )
        {
            vec4 tex = tex2D(reliefmap, p.xy);
            v *= 0.5;
            if (p.z<tex.w)
                p+=v;
            else
                p-=v;
        }
}

//Light the pixel using the texture coordinates texco
vec4 lighting(in vec3 texco){
        //If the texture coordinates are out of the bounds of the
        //texture we discard them to get silhouettes
        if (texco.x < 0.0) discard;
        if (texco.x > 1.0) discard;
        if (texco.y < 0.0) discard;
        if (texco.y > 1.0) discard;
        float shadow = 1.0;
        // color map
        vec4 color = texture2D(texmap,texco.xy);
        // Cone map
        vec4 normal = texture2D(reliefmap,texco.xy);
        // Get the two components of the normal and compute the third one
        normal.xy = 2*normal.xy - 1;
        normal.y = -normal.y;
        normal.z = sqrt(1.0 - dot(normal.xy,normal.xy));
        //Transform the light position
        vec3 light = vec3 (
     dot (eyeSpaceTangent, eyeSpaceLight),
     dot (eyeSpaceBinormal, eyeSpaceLight),
     dot (eyeSpaceNormal, eyeSpaceLight))- eye;
        vec3 p3,v3;
        //Setup the viewing ray as we do with the viewing one
        float a  = -depth/light.z;
        v3 = light * a;
        v3.z = 1.0;
        p3 = vec3 ( texco.xy - v3.xy * texco.z / v3.z, 0.0);
        //Compute the intersection between the lighting ray and the cone map
        ray_intersect_relaxedcone(p3,v3);
```

```
        vec4 normal2 = texture2D(reliefmap,p3.xy);
        if(normal2.w < normal.w -0.05 ){ //If pixel in shadow
          shadow=dot(ambient.xyz,vec3(1.0))*0.333333;
        }
   // Light and view in tangent space
        vec3 l = normalize(light);
        vec3 v = normalize(eye);

        // compute diffuse and specular terms
        float diff = shadow*(clamp(dot(l,normal.xyz),0.0, 1.0));
        float spec = clamp(dot(normalize(l-v),normal.xyz),0.0, 1.0);

        // Attenuation factor
        float att = (clamp(dot(l,normal.xyz),0.0, 1.0));

        //Compute final color
        vec4 finalcolor;
        finalcolor.xyz=ambient*color.xyz+att*(color.xyz*diffuse*diff)+specular
*pow(spec,shine);
        finalcolor.w = 1;
        return finalcolor;
}
```

Appendix 5: Environment maps

Ennis



Glacier

Pisa

University

Appendix 6: User guide of the application

After implementing all the algorithms we explained in this document we decided to create an application that made easier for the people in the group to use them (that was the original purpose of this project). First of all we must say that all this project has been developed using free software. The code is written in C++ and the shaders in GLSL (the opengl shading language) over a Linux operative system. This means that we can use this application in any Linux computer just following some steps, or even in windows, because all the libraries we used have their windows version.

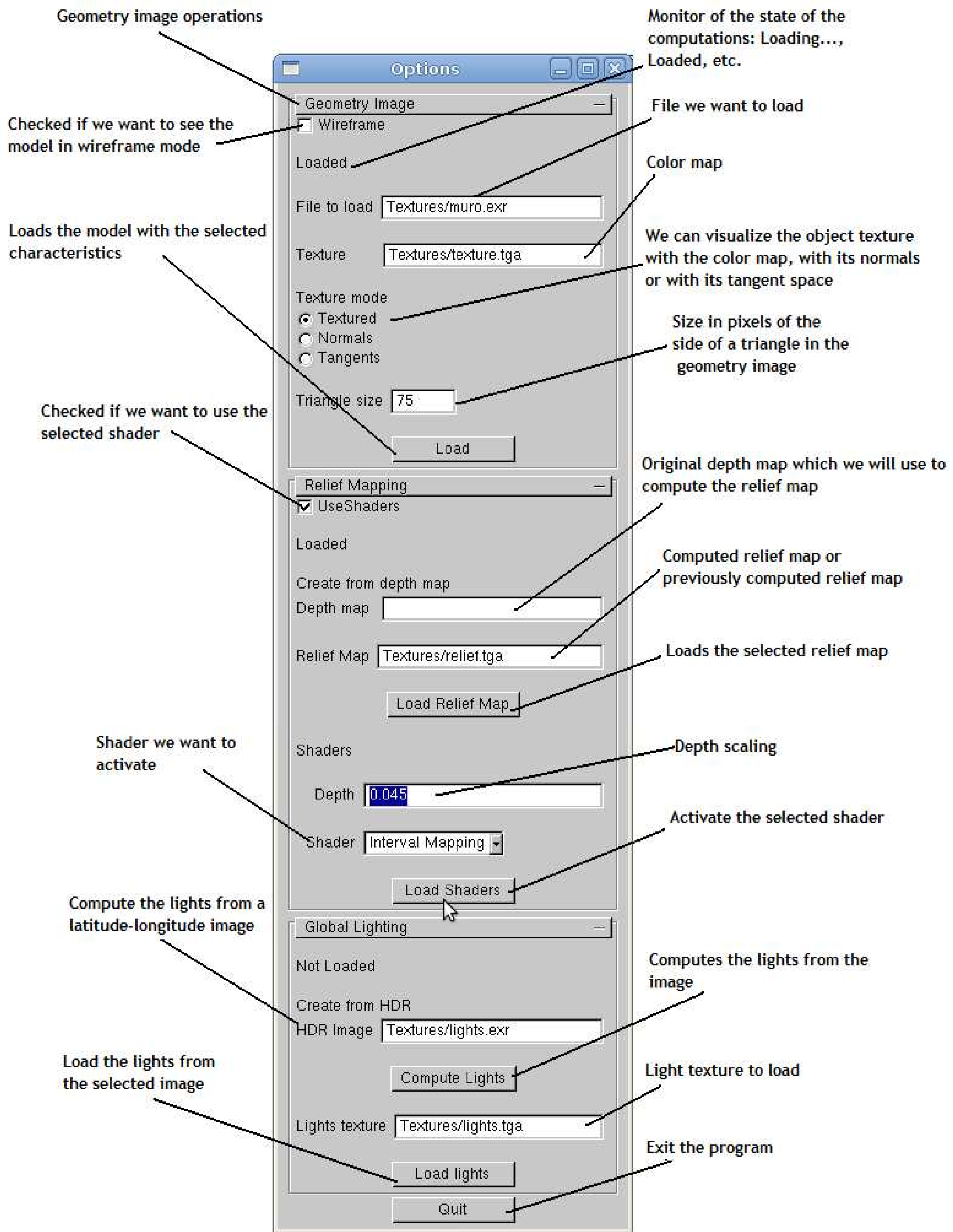
For the imaging we used Magick++, the object-oriented C++ API to the ImageMagick image-processing library. We just have to install it via apt-get or downloading it for free in its web page. To be able to compile Opengl applications we have to install the Mesa 3D graphics library for developers just using apt-get. We must have installed GLUI the user interface library based on GLUT that we used for implementing ours. We can also get this library from the apt-get, synaptic or similar programs.

Once we have this installed we can compile the application just typing make on the console. This will create an executable file called app. To execute the project we just have to type ./app and the user interface will appear.

When we execute the program there will appear two windows. One that will be a black GLUT window where the scene will be displayed when we select the object to display and the other one will be a interactive window where we could select some parameters for the rendering. This window is showed and explained in the figure in the next page.

We have to take into account that first of all we have to select a geometry image (if it is an HDR image it must be in .exr format because magick++ doesn't work with .hdr), its color map and the size of the triangles we want to sample on the geometry image. Once we have the object loaded the GLUT window will be resized to show the object depending on the size of the texture. Then we can select the shader we want to apply because, at the beginning it will be jus texture mapped.

On the GLUT window we can rotate the image using the left button of the mouse and moving it, we can translate the object using the right button in the same way and we can zoom using the scroll wheel. If we have the single light relief mapping (or interval mapping or cone mapping) activated we can move the light by typing l and using the mouse right button to translate it. Then, if we type l again we will be able to translate the model again.


Geometry image operations
Checked if we want to see the model in wireframe mode
Loads the model with the selected characteristics
Checked if we want to use the selected shader
Shader we want to activate
Compute the lights from a latitude-longitude image
Load the lights from the selected image
Options
Geometry Image
Wireframe
Loaded
File to load
Textures/muro.exr
Texture
Textures/texture.tga
Texture mode
Textured
Normals
Tangents
Triangle size
75
Load
Relief Mapping
UseShaders
Loaded
Create from depth map
Depth map
Relief Map
Textures/relief.tga
Load Relief Map
Shaders
Depth
0.045
Shader
Interval Mapping
Load Shaders
Global Lighting
Not Loaded
Create from HDR
HDR Image
Textures/lights.exr
Compute Lights
Lights texture
Textures/lights.tga
Load lights
Quit
Monitor of the state of the computations: Loading..., Loaded, etc.
File we want to load
Color map
We can visualize the object texture with the color map, with its normals or with its tangent space
Size in pixels of the side of a triangle in the geometry image
Original depth map which we will use to compute the relief map
Computed relief map or previously computed relief map
Loads the selected relief map
Depth scaling
Activate the selected shader
Computes the lights from the image
Light texture to load
Exit the program