Imperial College London

Department of Computing

Real-time volumetric shadows for dynamic rendering

by

Alexandru Teodor V.L. Voicu

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science / Visual Information Processing of Imperial College London

September 2011

Abstract

In order to produce realistic renderings for translucent objects, such as foliage or hair, self-shadows have to be used.

The first method that succeeds in casting real-time self-shadows is the Opacity Shadow Maps algorithm. However this method suffers from severe layering artifacts unless a large number of opacity maps are used, but that causes the application to have only interactive and not real-time performance.

Deep Opacity Maps solve the layering artifacts problem by aligning the opacity maps with the shape of the geometry as seen from the light's perspective. The downfall of this method is that only information about the front shape is provided and because of this the algorithm cannot work with multiple objects of various sizes.

The novel method proposed in this thesis, Bounding Opacity Maps, improve Deep Opacity Maps by also giving information about the objects' end position and shape. Furthermore an adaptive splitting scheme is proposed in order to better position the opacity maps based on the density of the objects in the scene.

Acknowledgements

First and foremost, I would like to thank my personal supervisor, Prof. Duncan F. Gillies, for his continuous comments, advice and suggestions involving this thesis. Without his constant guidance I would not have follow the interesting and innovative research path the project took in its final steps.

I would also like to thank the Crystal Space open source community for giving me invaluable insights regarding how their 3D open source render engine works. I personally thank my Google Summer of Code 2011 mentor, Mike Gist, and Frank Richter.

Last but not least, I would like to thank the Computing Support Group within the Department of Computing at Imperial College London for assigning me a work station when my personal computer broke down and I could not work from home anymore.

Contents

1. Introduction

The problem of rendering real-time realistic hair has been a significant topic of research in computer graphics for a long period of time, the first proposed solution going back to 1989.

Early approaches, such as those of Kajiya and Kay [KK89] or Marschner [MJC03], implement hair rendering by using simple models that try to approximate the physical phenomena that occurs when a ray of light intersects a hair strand. These methods are both fast to compute, but they lack realistic rendering, because they do not take into account the influence hair strands have upon other hair strands, which leads to self-shadowing (*Figure 1.1*).

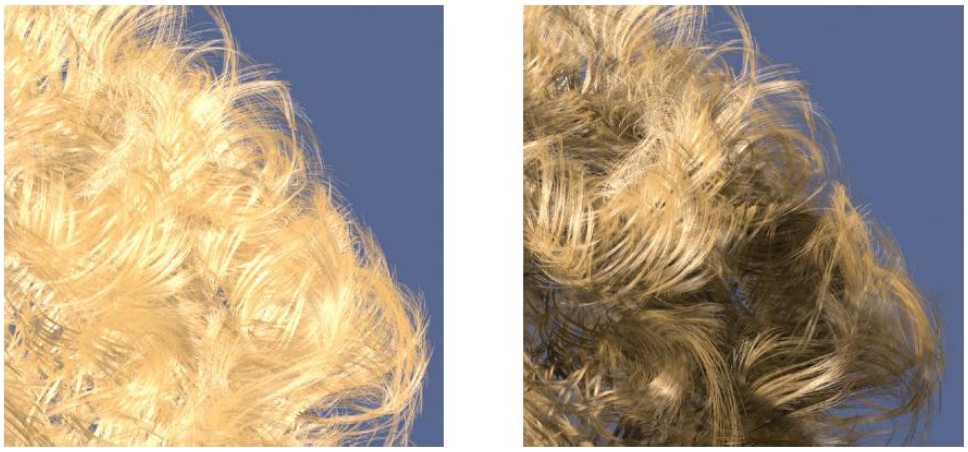

Figure 1.1 Picture produced in 3DS Max. Blonde hair rendered without (left) and with self-shadows (right).

More recent methods include self-shadowing, which is done by volumetric shadows, a technique that can be used to realistically render other dynamic translucent objects, such as smoke, clouds or foliage, as well. Because this technique works with a variety of objects, not just hair, it has been a popular research matter, a lot of improvements being documented, since the first interactive implementation of volumetric shadows, in "Deep Shadow Maps" [LV00].

1.1 Offline implementations

The first, and most realistic implementation of hair rendered with volumetric shadows was done by the film industry years ago, but by using offline implementations.

Offline implementations produce highly realistic visuals at the cost of computational time, and so they are used only by applications that do not require real-time performance, such as those from the film industry: Maya, 3DS Max, Blender or RenderMan.

Volumetric shadows are usually implemented in these applications by making use of volume rendering [L88], a technique that requires the input data to be organized in voxels. Therefore a 3D structured grid is computed for each translucent object, the size of a voxel in the grid being determined by the required rendering quality, smaller voxels giving better visuals at higher computational times.

Interactive implementations try to speed up volume rendering, by using hardware acceleration techniques supported by almost all modern video cards, such as rendering to texture, 3D texture or lookup textures.

1.2 Interactive implementations

The first interactive implementation of volumetric shadows was described in the paper "Deep Shadow Maps" [LV00], and it only slightly improves on volume rendering. Instead of rendering the scene using ray casting from the camera's point of view, a visibility function is first computed (*Figure 1.1.1*). The visibility function stores information regarding the amount of translucency each voxel has, as seen from the light's perspective. To find the alpha value of each voxel a volume rendering approach is used, in which rays are cast from the light's position and the change in transparency along each ray is stored in a texture, representing the visibility function. The overall complexity of this method, O(NlogN) is determined by the volume rendering part, which is implemented by first sorting the geometry according to the distance from the light's position and after that computing, in linear time, the intersection of each ray with the geometry.

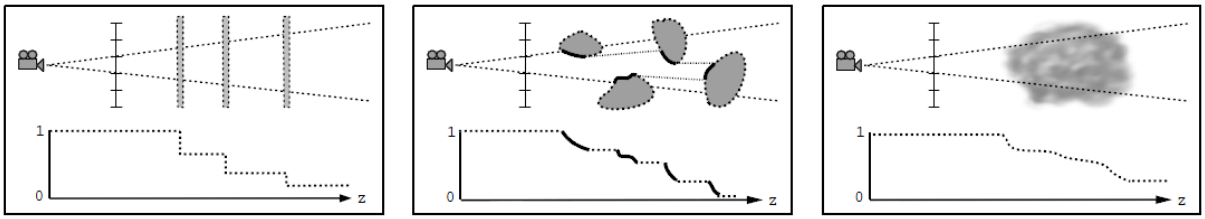

Figure 1.1.1. Picture taken from [LV00]. The visibility function is shown in the lower part of each image for the corresponding rays shoot from the camera and intersecting a central voxel.

A faster way of approximating volume rendering from the light's perspective, is using "Opacity Shadow Maps" [KN01]. Instead of sorting the geometry, the scene is rendered to texture multiple times (*Figure 1.1.2*), with different far clip values, and only the alpha value is stored in the corresponding render textures. The overall complexity is thus O(NM), where M is the number of textures used and N is the number of primitives to be rendered. Because rendering to texture is done using hardware acceleration for a small value of M the performance of Opacity Shadow Maps outperforms the one of Deep Shadow Maps, making it almost real-time on modern computers. However, in order to achieve realistic visuals a significant number of layers have to be used, so that the reference points give a good linear interpolation of the visibility function, making the performance only interactive.

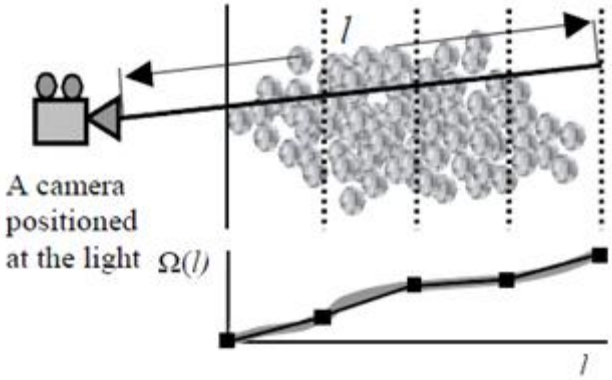

Figure 1.1.2. Picture taken from [KN01]. The opacity function, Ω(*l*), is computed for each pixel by using the information provided by each layer *l*.

Along the years a significant number of papers have been written improving on these two fundamental approaches of doing volumetric shadows.

Improvements of the way the visibility function is stored were proposed in "A Self-Shadow Algorithm for Dynamic Hair using Density Clustering" [MKBR04] and, more recently in "Fourier Opacity Mapping" [LB10]. The first approach uses clustering, k-means clustering in particular [Llo82], to choose the most significant pivot points, i.e. the points that best represent the geometry's changes in transparency. The second method approximates the visibility function using Fourier series, successfully minimizing the overall error between the approximation and the real function.

However, finding better ways of approximating the visibility function gives better visuals, but it does not improve on the bottleneck of the algorithm, which is either the sorting, O(NlogN) in Deep Shadow Maps, or the multiple renders to texture, O(NM) in opacity shadow maps. If the geometry is already sorted, the multiple renders to texture, can be done in O(N+M) ~ O(N), by rendering only the geometry between two textures, or slices, and using additive blending and a 3D texture to compute and store the rest of the pivot points. This means that a fast sorting algorithm could be of benefit to both methods, a fact noticed by Eric Sintron and Ulf Assarsson who proposed two GPU based sorting algorithms, one developed using CUDA [CUDA07], "Fast Parallel GPU-Sorting Using a Hybrid Algorithm" [SA07] and one that uses geometry shaders, "Real-Time Approximate Sorting for Self Shadowing and Transparency in

Hair Rendering" [SA08]. Even though the performance of their sorting methods are better than the classic implementation of quick sort from C/C++ [QSORT], their algorithms rely on state of the art hardware which is still not present on the average computer nowadays.

Approaches that do not require specific hardware, but improve sorting by taking into account some particular properties of the hair geometry, have been proposed in "Rendering real-time self-shadowed dynamic hair" [V10]. Although this approximate sorting has better time results than the quick sort from C/C++, the structures used require at least 3 times more memory than a normal rendering, which might not necessarily be available in a 3D rendering engine.

Nevertheless, the methods that have the best ratio between performance and visuals are the ones that try to minimize the number of layers used. Such an approach is presented in "Deep Opacity Maps" [YK08] where the opacity maps are aligned with the shape of the objects from the scene as seen from the light's perspective. However, because only information about the objects' front shape is given and linear interpolation is used, sometimes the number of layers originally proposed by this method produces visual artifacts.

2. Technical background

In order to develop an easy to maintain and to improve volumetric shadow implementation, writing a render manager in a modern game engine seemed the best solution. Because full access to the source code was needed in order to develop such a plugin, an Open Source Project, or Free Software project, freely available under the GNU Lesser General Public License (GNU LGPL) was chosen.

Crystal Space [CS07] is a project operating under the GNU Lesser licence, already at version 2.1_001, so the code, the available manual [CSM08] and API [CSD08], are at a mature development stage. The full source code for this final project, with all its commits, can be found online at its own SVN branch [CS11].

Crystal Space is a portable modular 3D SDK, and although it is mainly used for building different types of video applications and games, it can also be used for project that do not involve 3D content, but need a portable platform to develop C/C++ code. The project is written to run under a wide variety of hardware and software platforms, currently the following operating system being supported:

- Microsoft® Windows (9x/NT/ME/2000/XP)
- Unix® like Operating Systems (GNU Linux®, FreeBSD®, etc.)
- Apple® Mac OS® X

One of the most important facts about Crystal Space is that in order to achieve a high degree of modularity, all packages of components and libraries are built as plugins. Plugins have the

advantage that they are defined using just an interface and the implementation is hidden from the common user and can be easily updated at any time. Using such plugin libraries is also useful because they can be registered, loaded and queried only when and if needed.

Here is a list of features available in Crystal Space, also used by the render manager:

- Geometric utility library, covering a variety of mathematical notions, such as 2D and 3D vectors, matrices, transforms and quaternions.
- General utility library with template arrays, smart pointers, hash maps, object registry, plug-in manager or configuration files.
- Shared Class Facility SCF, which makes the separation between the interface and the implementation and allows dynamically loadable modules.
- Virtual file system and transparent support for ZIP files, allowing easy access to files on cross-platforms.
- Event system.
- Numerous types of mesh objects, of which only the most common used, genmesh, is currently supported by the new render manager.
- Native widowing system using CEGUI, used for the graphical user interface.
- Cross-platform hardware rendering using OpenGL, or a NULL renderer for applications like game servers.
- Advanced material support, including CG shaders and textures assigned to custom created materials.

2.1 The render manager

The render manager is a plugin which controls the rendering loop in Crystal Space. It can also control various properties, such as post-processing effects, anti-aliasing, z-buffer test or the size of the rendering textures.

The main steps of a render manager in Crystal Space are:

- Get the geometry in the sector to be render.
- Do the visibility culling, based on the camera's frustum.
- Sort the mesh list based on the Z coordinate and assign in-context per-mesh indices, needed only when translucent objects are to be displayed.
- Load and setup the shader variables arrays for each mesh.

However, these are only the main steps for the default unshadowed render manager, steps like rendering to textures, not being included. The render managers available at this moment in Crystal Space are: unshadowed, deferred [H04] and shadowed using parallel split shadow maps [ZFSXS06]. The latter is the most similar render manager to the one that has been recently added for casting volumetric shadows.

2.2 Parallel Splits Shadow Maps

Parallel split shadow maps [ZFSXS06], or cascaded shadow maps [D07], is a technique that splits the view frustum (*Figure 2.2.1*) into multiple depth layers in order to remove some of the visual artifacts common to the original shadows map method [SKWFH92].

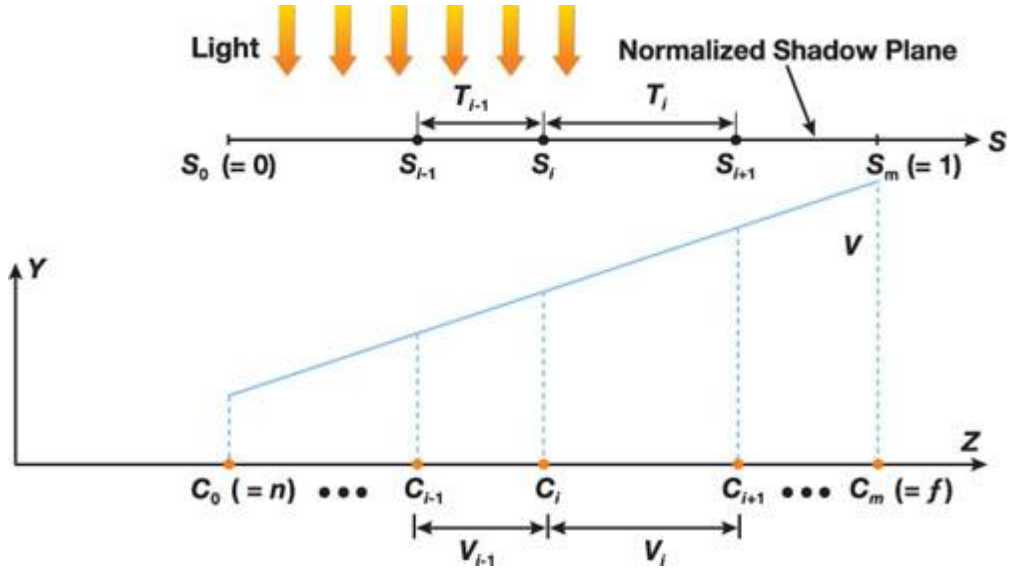

Figure 2.2.1. Picture taken from [ZFSXS06]. Splitting the viewing frustum, *V*, into *m*+1 planes, $C_{0...m}$, which are later normalized to the [0, 1] interval, $S_{0...m}$.

Instead of generating just one high resolution shadow map, the multiple depth layers make it possible to create multiple shadow maps at a smaller resolution and containing just a part of the scene.

Parallel splits shadow maps were already implemented in Crystal Space at the time this project was proposed, and the corresponding render manager contained a couple of additional features from the default render manager. These features are related to rendering to texture multiple times and splitting the view frustum into multiple depth layers, both also needed for the opacity shadow maps implementation. They are achieved by doing the following steps:

- Calculate the split distances, lights' and objects' frustums.
- Set up a new render view for each split and test the intersection of the lights' frustum with the objects' frustums.
- Render to texture for each render view using a depth render manager, without shadows, light or color information, but just storing the distance from the light's position to the objects' position.
- Load and setup the shader variables arrays that contain shadow information.
- Render the scene using a slightly modified version of the default render manager, which casts shadows when the distance from the object to the light is bigger than the distance in the rendering texture of the corresponding projected point.

3. Opacity Shadow Maps

Opacity shadow maps were introduced in Tae-Yong Kim and Ulrich Neumann's paper, "Opacity Shadow Maps", from 2001 [KN01] and it was the first method of obtaining volumetric shadows that had real-time performance at low level of details.

In this method volumetric shadows are obtained by approximating the light transmittance inside a complex volume with a set of planar opacity maps. A volume represented by standard primitives (points, lines and polygons) is sliced and rendered, using graphics hardware, to each opacity map that stores alpha values rather than depth values. The alpha values are sampled in the maps enclosing each primitive point and interpolated for shadow computation [KN01].

Opacity shadow maps were not already implemented in Crystal Space at the time this project was proposed, so such an implementation had to be done as a starting point for the novel algorithm proposed later on in this thesis.

The main modifications done to the parallel split shadow maps implementation were:

- The use of a different split scheme, a linear one as opposed to the existing logarithmic one.
- Create opacity maps instead of depth maps, which involved both rendering the scene to texture based on alpha and not on depth, and using this information as opacity rather than occlusion when doing the final pass from the light's perspective. As an optimization, four different layers were rendered to a single texture using a 32-bit channel for each.

The opacity shadow map render manger has the following steps:

- Calculate the split distances, lights' and objects' frustums.
- Set up a new render view for each split and test the intersection of the lights' frustum with the objects' frustums.
- Render to texture for each render view using an opacity render manager, without shadows, light or color information and without sorting, but just storing the additive alpha component from objects.
- Load and setup the shader variables arrays that contain shadow information.
- Render the scene using a modified version of the default render manager, which renders shadows according to the information stored in the opacity shadow maps: if a point to be rendered is denser from the light's perspective then it will be rendered darker in the final scene.

3.1 A first implementation

The first implementation of the opacity shadow map render manager consisted of splitting the scene into a predefined number of regions, using linear interpolation between the first and the last vertex belonging to any translucent object, i.e. having the alpha rendering priority in Crystal Space.

The linear interpolation scheme (*Figure 3.1.1.a*) was chosen only for its simplicity, but it might not perform very well when having multiple translucent objects in the scene that are far apart, because many opacity maps would not split through any object and contain the same information. Other more advanced schemes are proposed in [KN01]: adaptive slicing when the structure of the objects is known beforehand (*Figure 3.1.1.b*) and if regions farther away from the light's position have decreasing variations in transparency, a non-linear slicing method could be used (*Figure 3.1.1.c*).

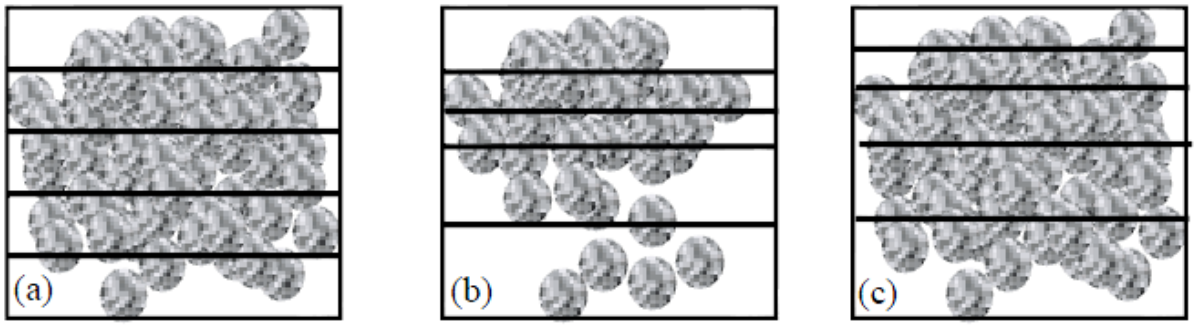

*Figure 3.1.1.*Picture from [KN01] – Various slicing schemes: uniform linear slicing (a), density based slicing (b) and non-uniform slicing (c).

In order to use the maps at their full capacity, i.e. have the objects rendered from the lights point of view cropped to a 2D bonding box (*Figure 3.1.2.b and Figure 3.1.2.d*), the frustums were build as in Parallel Split Shadow Maps [ZFSXS06]. Two axis aligned bounding boxes were used to compute the crop matrix: one for the objects that cast shadows and one for those that receive them. The crop matrix is then computed by constructing an axis aligned bonding box containing both the receivers and casters. The final light's view-projection transformation matrix for the current split is `lightViewMatrix * lightProjMatrix * cropMatrix` [ZSN07].

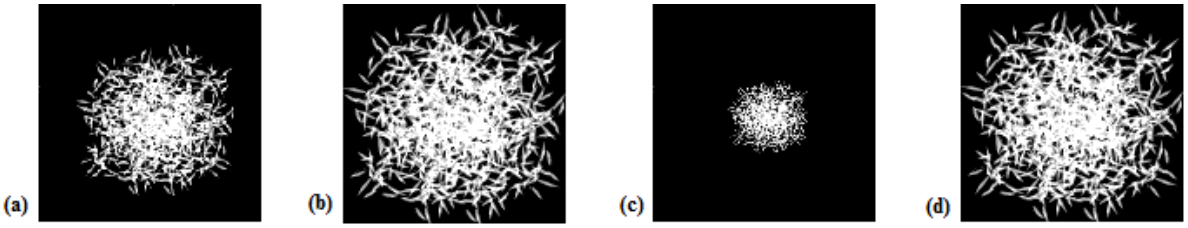

Figure 3.1.2. Picture from Crystal Space. Different opacity shadow maps. Scene rendered without using the whole capacity of the maps, when light is at distance 10 (a) and 20 (c). Scene rendered using bounding boxes for cropping when light is at distance 10 (b) and 20 (d).

After setting up the light's view-projection transformation matrix a new render view is created and associated to a context which renders to texture the data contained in the view-projection matrix, building an opacity shadow map. Then these maps are given to a later rendering pass which computes the final shadow by finding for each point the neighbouring maps and then linearly interpolating the values obtained by projecting the point on these two maps.

3.2 Limitations

The limitations of the first implementation of opacity shadow maps lie in the small number of layers that are created and passed to the final shader, due to the following two causes.

Firstly, even though a map stores information only in one channel, a texture with 4 channels is created for each such map, resulting in using 4 times more memory than actually required.

Secondly, the maps are passed to the final shader as an array of textures, which can only be indexed by constants and not by variables, resulting in the registration of the whole textures array, even though only two maps are used for a certain point. Furthermore, hardware limitations do not allow registering more than 16 such textures on the video card on which the implementation was developed and tested, NVIDIA GeForce GT 335M.

Because only such a small number of maps can be used, this first implementation suffers from severe visual artifacts (*Figure 3.2.1*). These artifacts are due to the fact that some points are present in one of the neighbouring maps, but not the other.

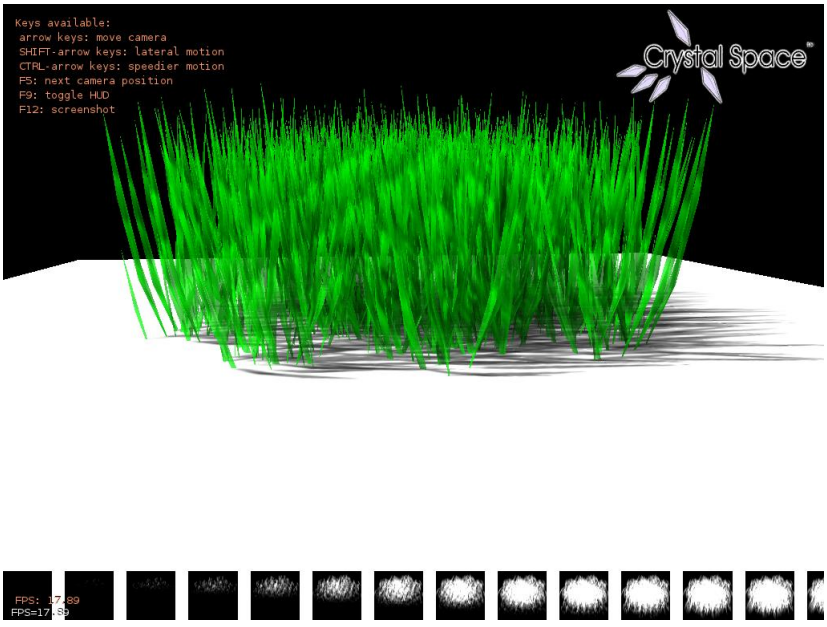

Figure 3.2.1 Picture from Crystal Space. Due to the small number of opacity shadow maps diagonal artifacts can be seen.

A possible solution for removing the layering artifacts is increasing the number of opacity shadow maps, as show in *Figure 3.2.2* from [YK08].

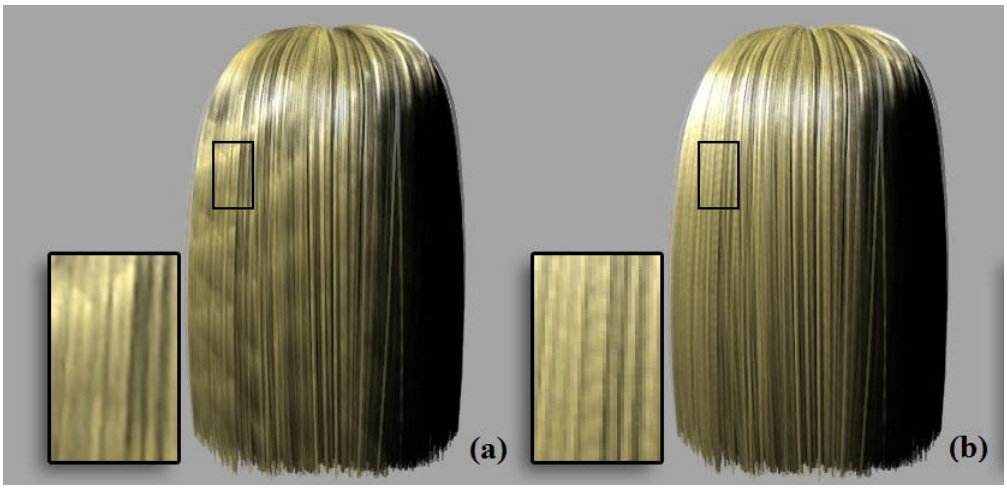

Figure 3.2.2 Picture from [YK08]. Removing Opacity Shadow Maps' visual artifacts by increasing the number of layers from 16 (a) to 128 (b).

Various approaches that try to minimize both the number of textures created and passed to the final shader and the time it takes to compute them have been added to this first implementation.

3.3 Using all available texture channels

A first optimization consisted of reducing the number of textures used, by storing information in the three color channels as well as the alpha channel. OpenGL uses blending functions [OGL97] in order to compute the final value of a channel, using the `glBlendFunc`:

```
void glBlendFunc(GLenum  sfactor,  GLenum  dfactor);
```

having the output definied as:

```
R = sfactor * Rs + dfactor * Rd;

G = sfactor * Gs + dfactor * Gd;

B = sfactor * Bs + dfactor * Bd;
```

where `(Rs, Gs, Bs)` is the source color (object being drawn), and `(Rd, Gd, Bd)` is the destination color (color already in the framebuffer).

The possible values for `sfactor` and `dfactor` are presented in *Table 3.3.1*.

Factor Name	Computed factor
GL_ZERO	0
GL_ONE	1
GL_SRC_ALPHA	As
GL_ONE_MINUS_SRC_ALPHA	1 - As
GL_DST_ALPHA	Ad
GL_ONE_MINUS_DST_ALPHA	1 - Ad
GL_CONSTANT_ALPHA	Ac
GL_ONE_MINUS_CONSTANT_ALPHA	1 - Ac
GL_SRC_COLOR	(Rs, Gs, Bs)
GL_ONE_MINUS_SRC_COLOR	(1 - Rs, 1 - Gs, 1 - Bs)
GL_DST_COLOR	(Rd, Gd, Bd)
GL_ONE_MINUS_DST_COLOR	(1 - Rd, 1 - Gd, 1 - Bd)
GL_CONSTANT_COLOR	(Rc, Gc, Bc)
GL_ONE_MINUS_CONSTANT_COLOR	(1 - Rc, 1 - Gc, 1 - Bc)
GL_SRC_ALPHA_SATURATE	min(As, 1 - Ad)

Table 3.3.1 List of possible values for source and destination factors.

In Crystal Space the blending function is called indirectly by using various mixmodes on different meshes or shaders (*Table 3.3.2*).

Name	Definition	Formula
CS_FX_MULTIPLY	CS_MIXMODE_BLEND(DSTCOLOR, ZERO)	SRC*DST
CS_FX_ADD	CS_MIXMODE_BLEND(ONE, ONE)	SRC+DST
CS_FX_ALPHA	CS_MIXMODE_BLEND(SRCALPHA, SRCALPHA_INV) | CS_MIXMODE_BLEND_ALPHA(ONE, SRCALPHA_INV)	srcAlpha*SRC + (1-srcAlpha)*DST
CS_FX_TRANSPARENT	CS_MIXMODE_BLEND(ZERO, ONE)	DST
CS_FX_DESTALPHAADD	CS_MIXMODE_BLEND(DSTALPHA, ONE)	(dstalpha)*SRC + DST
CS_FX_SRCALPHAADD	CS_MIXMODE_BLEND(SRCALPHA, ONE)	(srcalpha)*SRC + DST
CS_FX_PREMULTALPHA	CS_MIXMODE_BLEND(ONE, SRCALPHA_INV) | CS_MIXMODE_BLEND_ALPHA(ONE, SRCALPHA_INV)	SRC + DST*(1-srcalpha)

Table 3.3.2 List of possible mixmodes' values in Crystal Space.

At first the `CS_FX_ALPHA` mode was used, where the color channels are computed in the most common way, `srcAlpha*SRC + (1-srcAlpha)*DST`, and the alpha channel is `SRC + DST*(1-srcalpha)`. The alpha channel gives the correct result for the opacity function, because for alpha we have `SRC = srcalpha`, thus having the following result in the framebuffer: `srcalpha + DST*(1-srcalpha)`. This result can also be obtained for the color channels either by setting `SRC = 1` or by using `CS_FX_PREMULTALPHA` with `SRC = srcalpha`.

Even though these two approaches manage to store the same information in each texture channel, they cannot be computed independently, because one channel cannot be set

transparent (`srcAlpha = 1`) without influencing the alpha channel, which would be `SRC + DST*(1-srcalpha) = 1 + DST`.

The only way in which one channel can store a value in the framebuffer and all others be set to transparent is by using additive blending, `CS_FX_ADD`, as described in [ND05].

The problem when using `CS_FX_ADD` is that there is a linear increase, `SRC+DST`, instead of a logarithmic one `SRC + DST*(1-srcalpha)`, but that can be fixed by using the exponential function when computing the final opacity [ND05]:

```
half shadow = exp(-density);
```

The four channels are computed in one pass of one shader by calculating the distance from the light's position, which is actually the camera position when rendering from the light's perspective, to each point to be displayed. The red channel contains opacity information only about the first quarter of geometry, the blue channel about half of the geometry, the green three quarters and the alpha contains opacity data for all available geometry (*Appendix - Section 1*):

```
color = color * float4(index < 1, index < 2, index < 3, 1);
```

However this shader only receives a part of the geometry in one call, obtained by setting the near and far z-plane to that particular split, which will be further divided into four parts inside the shader.

Because the geometry is split using both the near and far z-plane, the render texture obtained contains information only about that particularly part of the geometry, and in order to compute the final opacity at a certain point the values from all precedent render textures has to be added (*Appendix - Section 2*):

```
for (int j = 0 ; j < prevIndex ; j ++)
      previousMap += getMapValue(4 * (j + 1) - 1, prevPos);
```

Just by using all four channels the number of available opacity shadow maps increased by 4, being in a 4:1 ratio with the number of available textures, which is 16 due to the hardware limitation mentioned in *Section 3.2*.

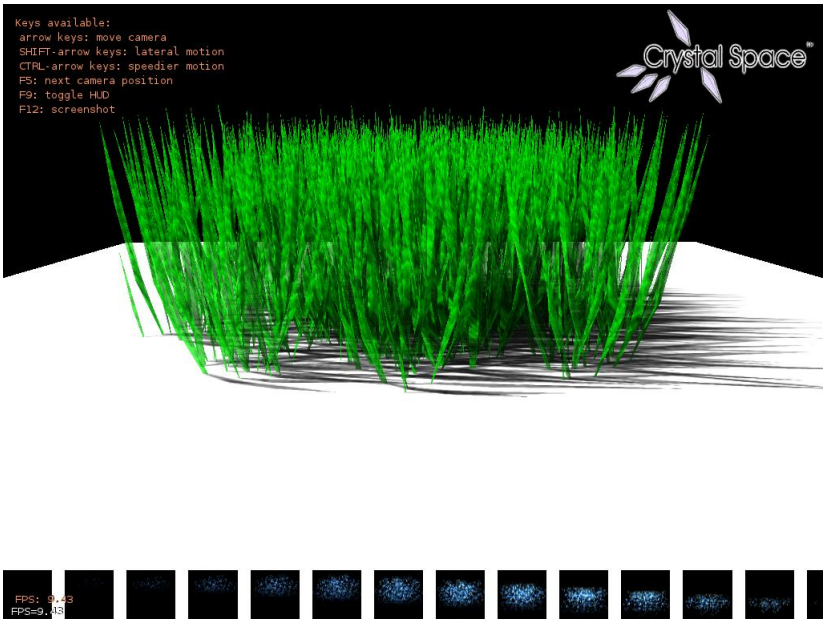

Figure 3.3.1 Picture from Crystal Space. Grass rendered using 16 render textures, yielding 64 opacity shadow maps. The visual artifacts are less visible than in *Figure 3.2.1*.

3.4 Multiple render targets

Multiple render targets [MRT04] is a feature of modern GPUs that allows a single pass of a shader to output information to more than one location (`COLOR0`), making it possible to render images to multiple render target textures at once. They were first introduced in the API of OpenGL 2.0 and Direct3D 9, but video cards supporting this feature appeared years later.

The benefit of introducing this feature to the opacity shadow maps algorithm is that even more opacity maps can be generated in a single pass of a single shader, similar to how 4 maps were generated in a single pass when using all available channels, in *Section 3.3*.

The only disadvantage of MRTs is that different GPUs support a different number of render targets and the code written on the GPU has to avoid calling or writing to invalid memory zones. For instance older video cards that do not support Direct3D 9 or OpenGL 2.0 have the number of MRTs set to 1, those that do support it have it set to 4 and the newer one that are capable of Direct3D 10 and OpenGL 3.0 support up to 8 MRTs [MRT04].

Because of this, three shaders are automatically generated to access and write to 1, 4 or 8 MRT depending on the available number of MRT on the hardware the application is run. The way in which a rendering texture is selected is very similar to the way one of the four channels is selected in *Section 3.1.2*: by calculating the distance from the light's position to each point to be displayed. The closest vertices go to the first rendering target and further points are stored in the following render targets if available (*Appendix - Section 3*).

Thus for video cards supporting 8 MRTs one pass can generated up to 32 opacity maps in one go, giving a considerable speed-up, because 8 times less rendering contexts have to be created.

3.5 Percentage-closer filtering

The same techniques used to improve shadow maps can be applied to opacity shadow maps with good results, as well. This can lead to a significant speed-up because lower resolution maps can be used without introducing any significant visual artifacts.

Because shadow map textures cannot be prefiltered to remove aliasing, like normal textures, multiple shadow map comparisons have to be made per pixel and averaged together, by a technique called percentage-closer filtering (PCF) [RSC87].

Although the original PCF algorithm as described in [RSC87] sampled the region to be shaded stochastically (randomly), by constructing a four-sided micropolygon (*Figure 3.5.1.a*), newer implementations [BP04] use a 4x4 texel sampler region (*Figure 3.5.1.b*).

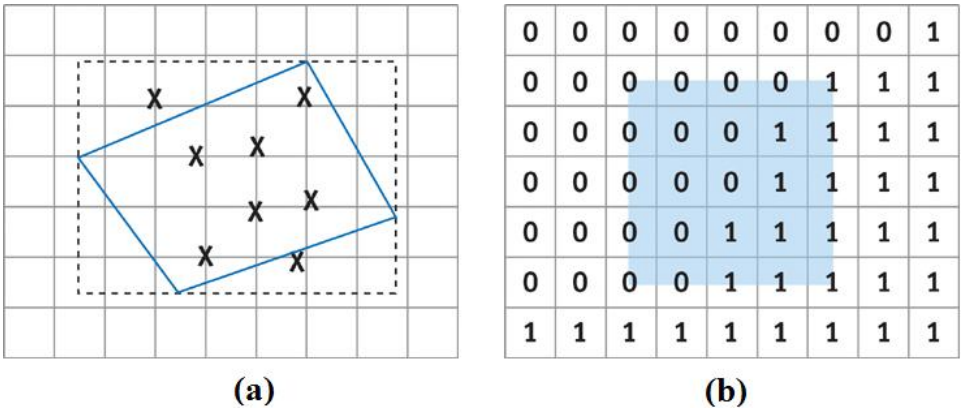

Figure 3. 5.1 Picture taken from [BP04]. Choosing a four-sided micropolygon (a) or a 4x4 texel region (b) as neighbourhood.

A brute force method can be used to compute the 4x4 texel region average with good performance, mainly because most texture fetches are in the texture cache being close to one another [BP04] (*Appendix - Section 4*).

Other ways of improving the opacity maps, such as point splatting [KN01] or variance shadow maps [DL06] have been taken into account, but because they do not benefit from native hardware acceleration they tend to be slow, so the PCF method was preferred.

4. Deep opacity maps

Instead of trying to remove the visual artifacts of the Opacity Shadow Map method by increasing the number of maps, and at the same time decreasing the size of an artifact until it becomes too small to be noticed, Deep Opacity Maps [YK08] try to solve the problem by aligning the opacity maps with the hair geometry (*Figure 4.1*).

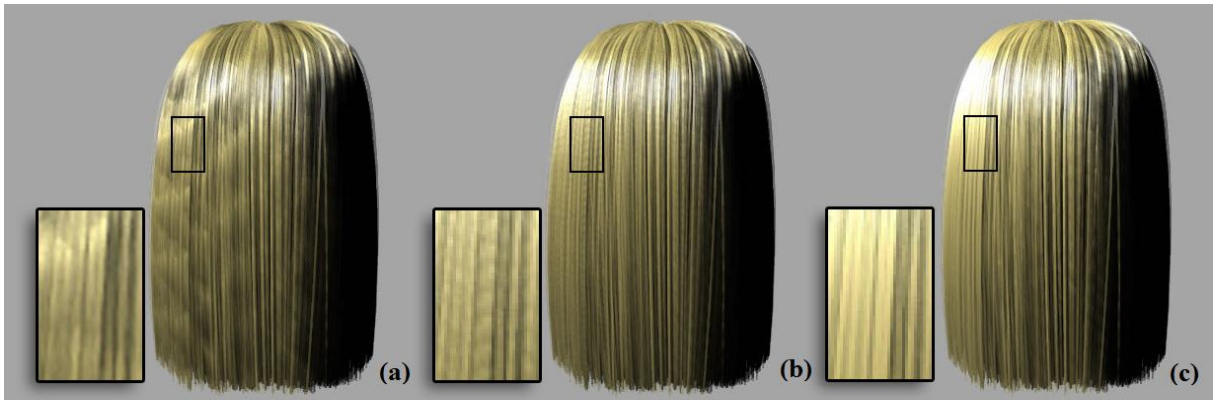

Figure 4.1 Picture taken from [YK08]. Opacity Shadow Maps with 16 layers (a), with 128 layers (b) and Deep Opacity Maps with only 4 layers (c).

The deep opacity maps method combines shadow mapping [SKWFH92] and opacity shadow maps [KN01] to give a better distribution of the opacity layers. In a first render pass the scene is drawn from the light's perspective and only information about depth is stored, by a regular shadow mapping technique. A second rendering pass constructs the opacity maps by doing a linear splitting, similar to the Opacity Shadow Map [KN01] technique, but which uses an offset equal to the value read from the shadow map generated in the previous pass (*Figure 4.2*).

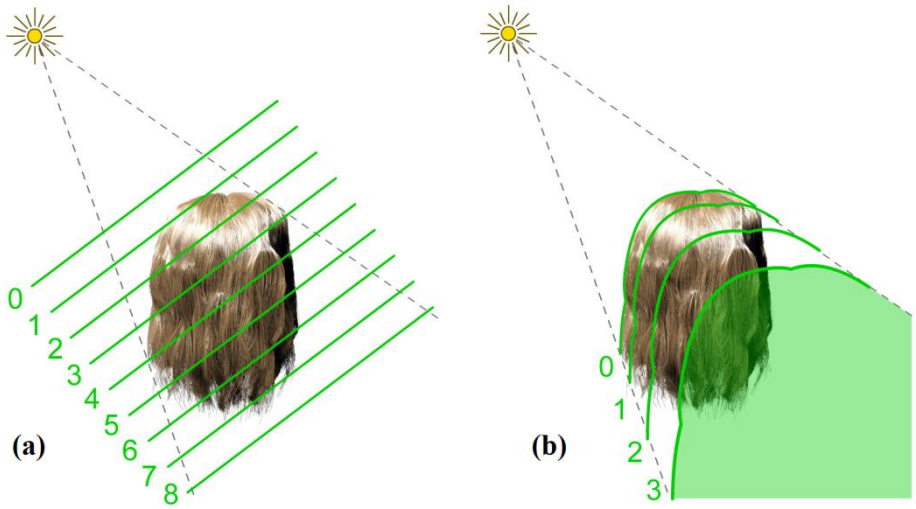

Figure 4.2 Picture taken from [YK08]. Opacity Shadow Maps use linear interpolation (a), while Deep Opacity Maps (b) also use an offset read from the shadow map, so that they conform to the shape of the model.

Because the layer distribution in deep opacity maps guarantees that the direct illumination coming from the light source without being shadowed is captured correctly, by using the shadow map offset, a smaller number of opacity maps can be used to generate high quality shadows [YK08].

The biggest advantage of this new way of splitting is that it can still be done using common hardware and technologies, because it only involves reading and writing to textures inside a fragment shader. The hair volume can be divided into K layers such that each layer lies from $z_0 + d_{k-1}$ to $z_0 + d_k$, where z_0 is the depth read from the shadow map, $d_0 = 0$, $d_{k-1} < d_k$ and $1 \leq k \leq K$. Furthermore the layer size does not have to be the same, i.e. $d_k - d_{k-1} \neq d_{k-1} - d_{k-2}$, and even though the same d_k values are used for each pixel, z_0 varies by pixel, so the layers take the shape of the geometry model in a linear splitting distribution.

Figure 4.3 shows how Deep Opacity Maps remove layering artifacts as compared to the Opacity Shadow Maps on the translucent grass model from Crystal Space.

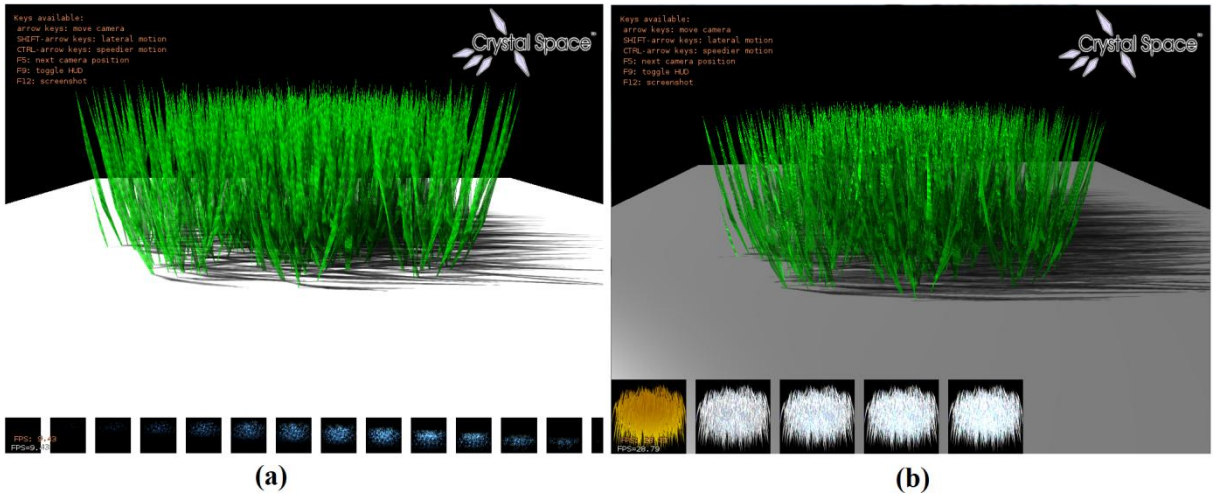

Figure 4.3 Picture from Crystal Space. Opacity Shadow Maps with 64 layers (a) and layering artifact free Deep Opacity Maps with 16 layers (b).

4.1 Limitations

Even though Deep Opacity Maps do not require as many layers as Opacity Shadow Maps to render high quality shadows, just 3 layers, the number typically used in [YK08], is not quite enough for cluster geometry (*Figure 4.1.1*).

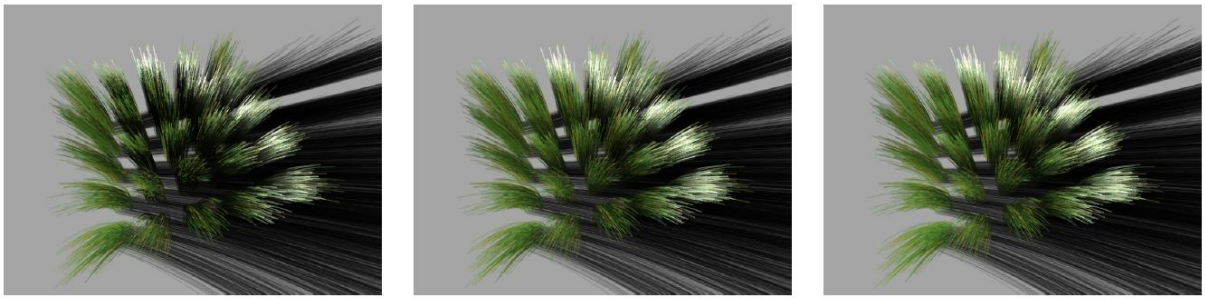

Figure 4.1.1 Picture from [YK08]. Only 3 layers at a low resolution (a) do not produce high quality shadows, so either the number of layers has to be increased (b) or the map's resolution (c).

Because, when supported, multiple render targets [MRT04] represent a fast way of generating up to 32 opacity maps (*Section 3.4*), a basic scheme for controlling the visual quality of shadows, while roughly maintaining the same performance, has been developed for the current implementation.

The maximum number of supported multiple render targets is used to generate as many layers as possible in a single rendering pass. Thus newer hardware gives higher quality shadows, while older video cards, which only use one render target, and still one rendering pass, produce 4 layers and lower quality shadows. However, the performance tends to stay the same because the number of rendering passes used to generate the Deep Opacity Maps remains the same.

Moreover, if the scene to be rendered does not actually require more than a few layers, there is no geometry clustering, the number of multiple render targets used can be forced to a certain value via the scene configuration file.

Another problem with Deep Opacity Maps is that although the initial offset for linear splitting is clearly specified for every point, there is no information regarding where the splitting should stop. This can either lead to visual artifacts or to an inefficient usage of the opacity maps. The solution preferred in [YK08], that the last layer will contain all the remaining points, can lead to visual artifacts, similar to those from clustering (*Figure 4.1.1*), if the last layer begins too early (*Figure 4.1.2.b*). On the other hand if the splitting is chosen so that the last layer lies beyond the hair volume, poor usage of the opacity maps will occur, because not only the last layer would not contain any information at all, but the rest of the layers will contain information only in a small proportion (*Figure 4.1.2.c*). A solution to this problem is proposed further on in the current thesis (*Section 5*).

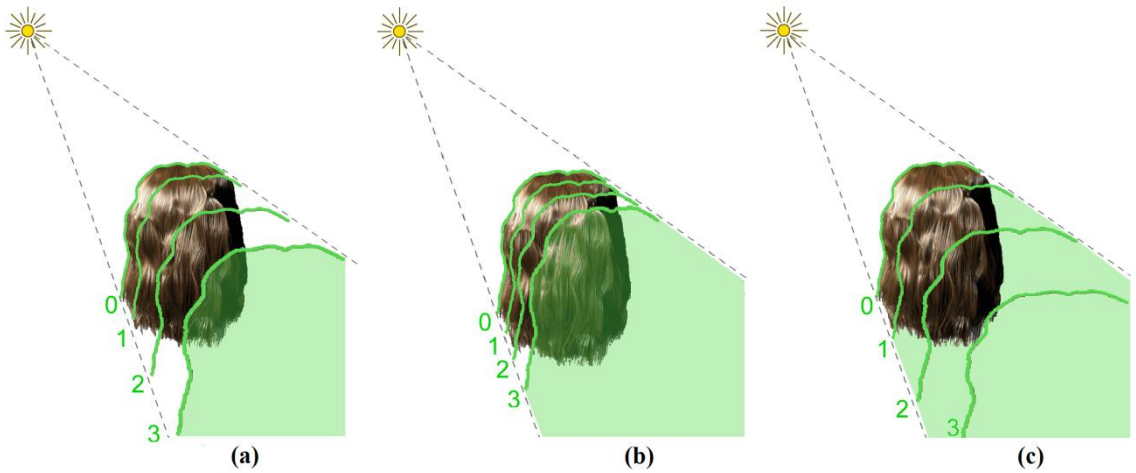

Figure 4.1.2 Modified picture from [YK08]. Standard linear splitting (a), splitting that ends too early (b) and splitting that contains all the hair volume (c). Choosing the end splitting points too near (b) results in the last layer having a lot of information, which produces visual artifacts. Choosing the splitting point too far (c) leads to a big part of the layers not containing any useful information (the green hashed region).

4.2 Implementation details

Even though the implementation of Deep Opacity Maps, required mainly adding another rendering pass to compute the shadow map and making sure the second rendering pass, which generates the layers, is called only one time, some other small optimizations were done in order to increase performance.

One significant benefit of choosing a linear splitting scheme is that the exact position of the splitting points does not need to be stored explicitly, which would involve a costly transfer of an array from the main memory to the video memory. Furthermore, if the position of the splitting points can be determined in constant time, using linear interpolation, this means that any point within the splitting interval can be attributed to a corresponding splitting point, and thus layer, in constant time as well.

Because only one rendering pass is used when generating the layers the whole hair volume is passed to the shader. Therefore each opacity map can be efficiently created so that it contains information about every point until the splitting point and not only points which reside between the previous splitting point and the current one. This is done by adding the current rendered point, from the second rendering pass, to all previous rendering targets and not just the one determined by the splitting point (*Appendix - Section 5*). The advantage of this is that when computing the final opacity value for a certain point all the information needed is stored in just one opacity map and not divided among several, like in the case of Opacity Shadow Maps, so no additional texture lookups have to be done for previous layers.

A regular depth map couldn't be used for the first rendering pass due to the fact that this map needs to be accessed like a regular texture, not a shadow map, which is impossible to do on a

depth map [tex2D]. Instead the depth information was written to a single 32 bit channel of a 128 RGBA texture, which was accurate enough for doing the linear interpolation for deep opacity maps without noticeable visual artifacts.

As a consequence of the fact that neither searching for the linear splitting scheme, nor adding information from previous layers is required, no `for` statements have been used in the vertex or fragment programs, which made the code optimal and compatible with older hardware as well. In order to further increase performance, all conditional branches were avoided, and variables having values based on conditions were used instead.

5. Bounding opacity maps

Because Deep Opacity Maps do not give any information about the end splitting points either visual artifacts or a poor usage of the opacity maps can occurs, as described in *Section 4.1*. Furthermore, if the layers are built by only taking into account the depth map, i.e. the shape of the object as seen from the light's perspective, they only follow the initial light distribution, as it enters the object. For instance, the layers obtained using Deep Opacity Maps would follow the distribution shown in *Figure 5.1.b*, which does not correspond to the way in which the light distribution occurs in the real world *Figure 5.1.c*.

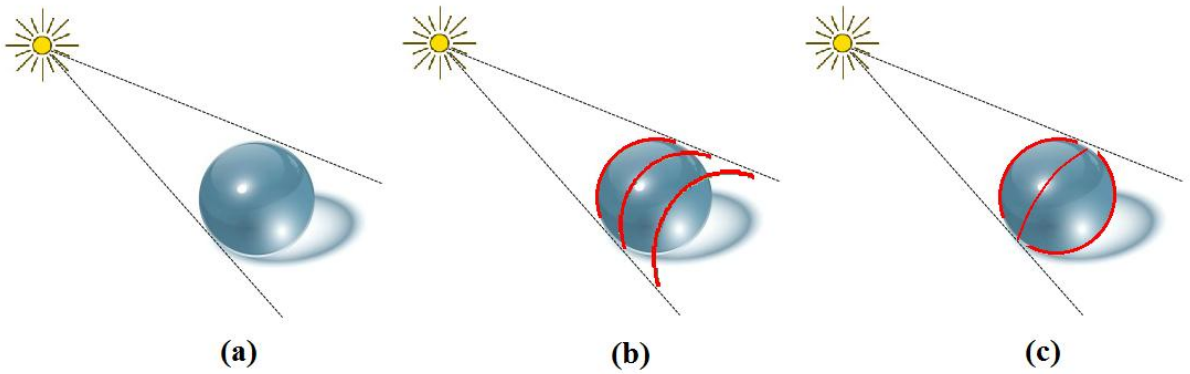

Figure 5.1 A translucent full sphere as seen in real-life (a), the distribution of layers when using Deep Opacity Maps (b) and the way the light is distributed in real-life (c).

Moreover, the example illustrated in *Figure 5.1* is not a particular case, the light distribution following the shape of the object for other translucent real world objects, as can be seen in *Figure 5.2* and *Figure 5.3*.

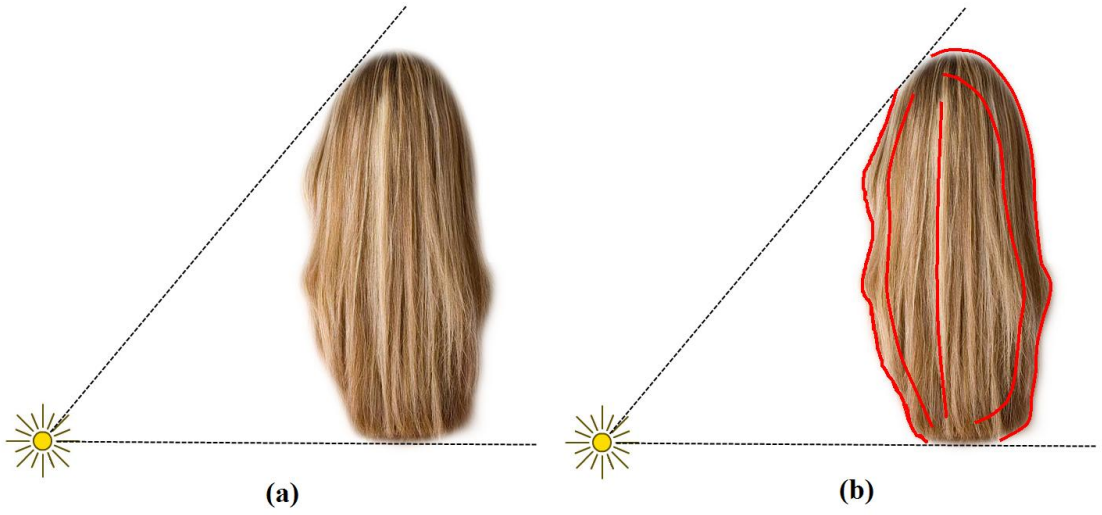

Figure 5.2 Real-life lighting of blonde hair (a) and the corresponding layers and light distribution (b). It can be observed that the layers and the light distribution follow the shape of the object.

By computing an extra depth map, in which depth information about the furthest away points is given, instead of the closest ones, the limitation of Deep Opacity Maps regarding the lack of information for the end splitting points is solved, and more important the layers follow the light's distribution in real-life.

The novel solution proposed in the current thesis is named Bounding Opacity Maps and achieves a layering following the light distribution in real-life by interpolating the values from the two depth maps when choosing the splitting points.

Because the same technologies are used as in Deep Opacity Maps, the splitting can still be done using common hardware and technologies and it still benefits of hardware acceleration.

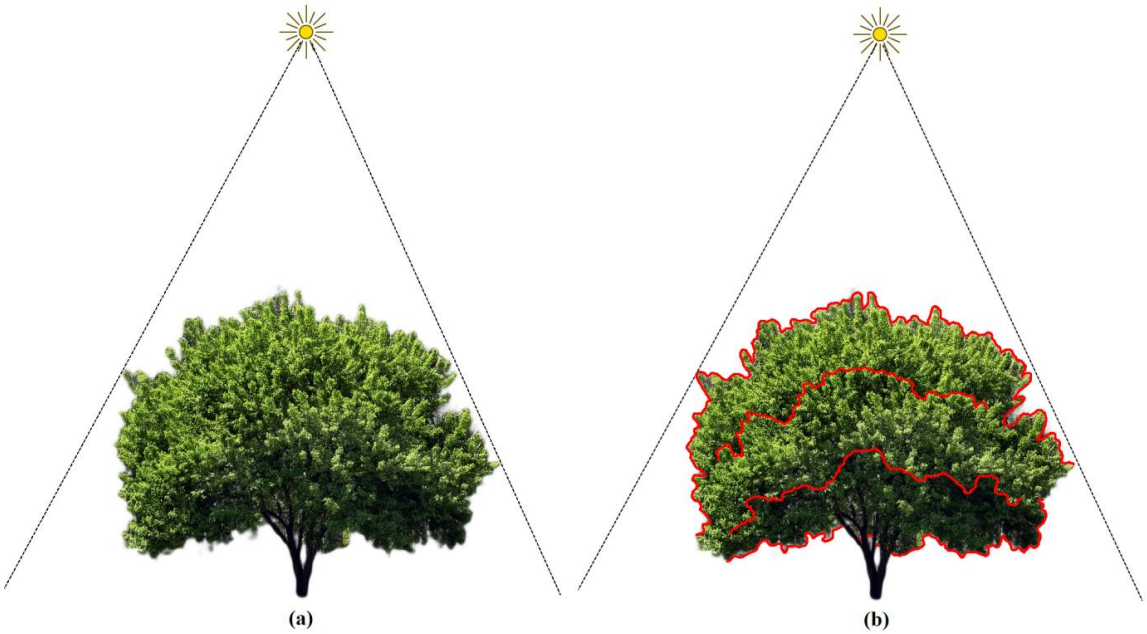

Figure 5.3 Real-life lighting of a tree (a) and the corresponding layers and light distribution (b). It can be observed that the layers and the light distribution follow the shape of the object.

The geometry can be divided into K layers such that each layer lies from $z_0 + d_{k-1}$ to $z_0 + d_k$, where z_0 is the depth read from the first depth map, $d_0 = 0$, $d_{k-1} < d_k$, $1 \leq k \leq K$ and $d_K = z_1$, where z_1 is the depth read from the second depth map. Furthermore, the layer size is not the same for different zones in a layer because the difference between z_0 and z_1 is not constant. As part of the first implementation the interpolation between z_0 and z_1 was a linear one, meaning that $d_k - d_{k-1} = d_{k-1} - d_{k-2}$, but other splitting schemes can be used as well, as shown in *Section 5.2*.

```
1. init:
2.    geometry := load_scene('scene.txt')

3. render_loop:
4.    start_depth_map := render_pass(geometry)
5.    end_depth_map := render_pass(geometry)
6.    render_textures := render_pass(start_depth_map, end_depth_map,
        geometry)
7.    render_scene(render_textures, start_depth_map, end_depth_map,
        geometry)
```

Algorithm 5.1 A basic algorithm for Bounding Opacity Maps. The only difference from Deep Opacity Maps is line 5 in which the end splitting points are computed. There are a total of three render passes that compute the start splitting points (line 4), the end splitting points (line 5) and the opacity layers (line 6). The information provided by these render passes is used to render the final scene on line 7.

Although the Bounding Opacity Maps algorithm (*Algorithm 5.1*) is only slightly different from the Deep Opacity Maps one, both the difference in splitting (*Figure 5.4*) and in rendering (*Figure 5.5*) are quite substantial.

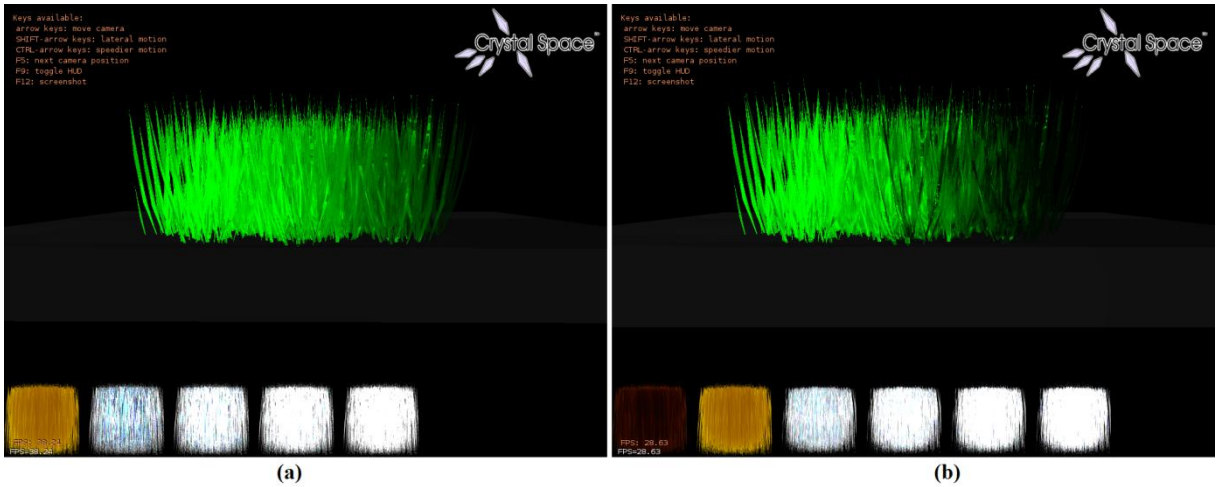

Figure 5.4 Difference in splitting between Deep Opacity Maps (a) and Bounding Opacity Maps (b) when using 16 layers - first layer corresponds to light green and the last layer to black. It can be seen that because the end splitting points are not specified in Deep Opacity Maps the layers do not cover the whole length of the object (the final color is not black as in (b)).

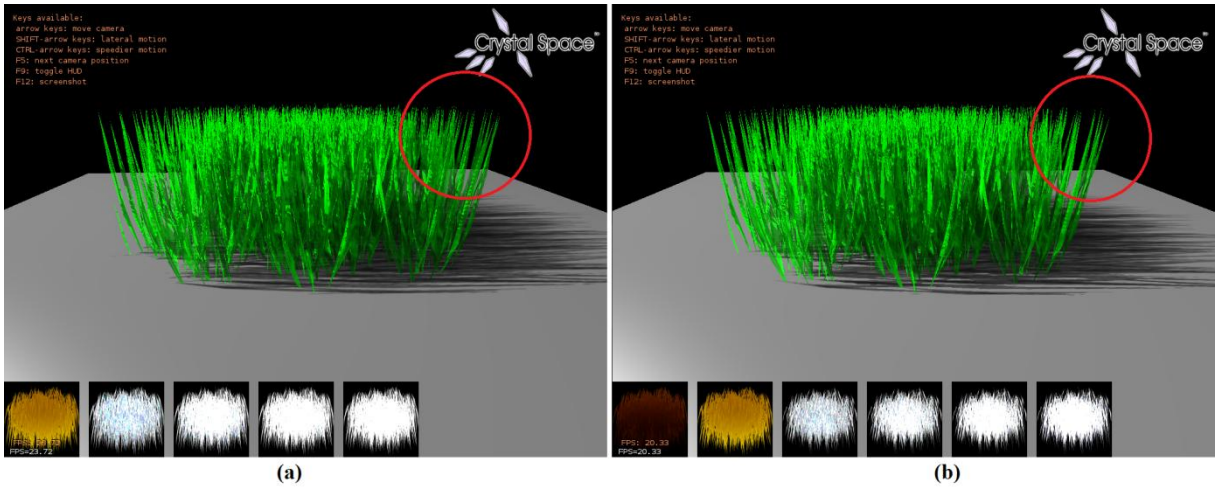

Figure 5.5 Difference in rendering between Deep Opacity Maps (a) and Bounding Opacity Maps (b) when using 16 layers. Because Deep Opacity Maps do not specify the end splitting points some grass strands (from the red circle), corresponding to the last layer, are given false shadow information.

5.1 Implementation details

The implementation of Bounding Opacity Maps is very similar to the one of Deep Opacity Maps, the main difference is that two depth maps, instead of just one, are computed. In order to compute the second depth map another rendering pass was added, but the time lost with this extra pass was recovered by the fact that even fewer layers than in Deep Opacity Maps can be used without losing any details. The correct visual appearance is guaranteed even with just a few layers by the fact that the maps now follow the light's distribution in real-life.

As stated in *Section 5* the second depth map contains information about the depth calculated from the light's position where objects end. In OpenGL this can be achieved directly by calling the `glDepthFunc` function with `GL_GREATER` / `GL_GEQUAL` instead of `GL_LESS` / `GL_LEQUAL` [OGL97]. This sets the zbuffer with new data (depth value and color information) only if the incoming depth is greater / greater or equal than the stored depth value. However because this way of setting the zbuffer is quite specific to OpenGL and there were not any straight forward ways of copying this behaviour in Crystal Space, another more general method was used instead.

This general way is a mathematical solution and it consists of swapping the value of the near plane with the one of the far plane when computing the projection matrix from the light's perspective in the second rendering pass. By doing this the direction of the Z axis is reverted, so every object in the scene looks as if it were mirrored (*Figure 5.1.1*). The useful fact about the mirrored scene is that the closest points from the light correspond to the farthest points from the light from the mirrored scene, and this is valid both ways. Due to this property the same code from the first rendering pass in Deep Opacity Maps, which computed the depth of the points closest to the light's position, could be used to compute the furthest points from the

light's position in the second rendering pass from Bounding Opacity Maps. As a side note OpenGL has a specific easy way of swapping the near and far values by calling the function `glDepthRange` with 1 for the near value and 0 for the far one: "It is not necessary that nearVal be less than farVal. Reverse mappings such as nearVal=1, and farVal=0 are acceptable" [OGL97].

Figure 5.1.1 A scene where the light is at the origin and the light's direction corresponds to the positive Z direction (a) and its corresponding mirrored scene (b).

The fact that both the start and the end position of objects in the scene are known, via the two depth maps, allows finding a precise position of the splitting points by doing an interpolation between the values stored in the depth maps (*Appendix - Section 6*).

5.2 Splitting scheme

Even though the most common splitting scheme is the linear one, the light distribution does not necessarily follow linear interpolation. If we were to look at the light's distribution on real-world translucent objects such as clouds, trees or hair we can observe that for a uniform shape and a constant, high alpha value the lighting caused by self-shadowing changes only at the very beginning of the object (*Figure 5.2.1*).

Therefore depending on the density of the geometry or on the amount of translucency an object has, new shadow information may appear or not onto the next layer. In certain cases, like the one described above, only the first few layers have different information (*Figure 5.2.2*), so a linear distribution of the layers is not a good option. A distribution that has multiple layers near the start splitting position, i.e. at the beginning at the object, and fewer as the layer position reaches the end splitting position would give better results.

Figure 5.2.1 Real-world photographs of clouds (a) and bushes (b). It can be observed that for these objects the lighting only changes at the very beginning of the object.

Figure 5.2.2 Layers obtained using linear splitting on the grass scene. The last four layers contain almost the same information.

5.2.1 Logarithmic splitting

The logarithmic distribution has a slower increase rate and therefore produces a splitting that has a higher density of layers at the beginning of the object (*Figure 5.2.3*). And because there are more layers at the beginning of the object, fewer layers contain the same information (*Figure 5.2.4*). Obtaining layers that have different shadow information prevents artifacts like the ones shown in *Figure 5.2.5*.

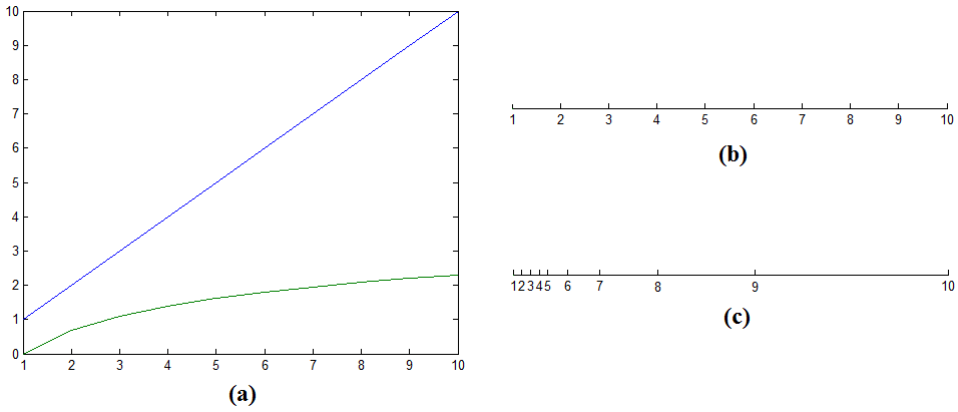

Figure 5.2.3 Comparison between linear and logarithmic distributions. Linear increase, blue, versus logarithmic increase, green (a), linear split (b) and logarithmic split (c).

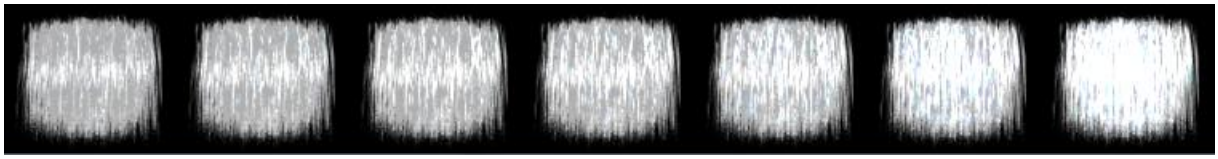

Figure 5.2.4 Layers obtained using logarithmic splitting on the grass scene. Every layer adds new shadow information, i.e. is different from the previous one.

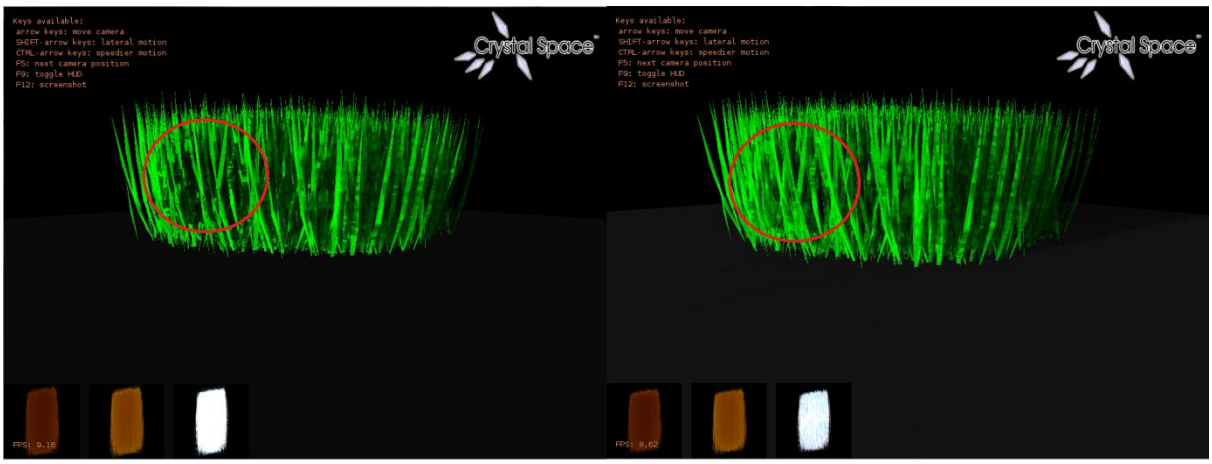

Figure 5.2.5 Picture from Crystal Space. Difference in rendering when splitting using linear splitting (a) and logarithmic splitting (b). Linear splitting (a) produces incorrect self-shadows because most of the layers contain the same information (*Figure 5.2.2*).

One disadvantage of using a non-linear splitting scheme is that the splitting positions cannot be computed without a lot of arithmetical instructions. This is caused by the fact that the range of the object is given in local coordinates from 0 to 1, where 0 corresponds to the beginning of the geometry and 1 to the end of it.

Even though the interval [0, 1] could be used directly for linear interpolation, it is not a proper input interval for the logarithmic function, because the output interval would cover an infinite range from [-∞, 0] and wouldn't have the distribution shown in *Figure 5.2.3.a*. In order to obtain such a distribution the interval has to be converted first to a positive interval greater than 1, which covers a bigger range of values, say [1, N]. After the logarithmic function is applied another interval is output [0, log(N)], which has to be converted to [0, 1] once again.

Therefore converting between intervals in the case of logarithmic splitting requires significantly more arithmetical instructions than linear splitting which only requires a division for finding a splitting position. However, this conversion can be done automatically if we use lookup textures [SL04].

5.2.2 Lookup textures

Lookup textures are usually RGBA 1D, 2D or 3D textures, in which a function having the [0, 1] input and output interval is encoded. For instance the logarithmic function is encoded by first converting the [0, 1] interval to [1, textureSize - 1], afterwards applying the logarithmic function which yields the [0, log(textureSize)] interval, which is finally converted to [0, textureSize - 1] (*Appendix - Section 7*). Because the texture's size is usually a power of 2, due to compatibility reasons [CW04], the logarithmic function used was log2.

The major advantage of using this technique is that the lookup texture is computed only once during the initialization stage, and afterwards it is used instead of the numerous arithmetic operations needed to convert the input and the output interval as well as applying the logarithmic function itself (*Appendix - Section 8*).

5.2.3 Hybrid split

Although the linear splitting scheme falls short for uniform shapes and a constant, high alpha value (*Figure 5.2.1*) and the logarithmic one does not work properly with objects that either have scattered geometry or a low alpha value (*Figure 5.2.3.1*), using one of the splitting scheme when the other falls short produces good results.

Figure 5.2.3.1 Real-world photographs of clouds (a) and trees (b). It can be observed that for objects having a scattered geometry the lighting changes throughout the entire length of the object.

Section 5.2.1 already describes how to use a logarithmic split when the linear split does not produce good enough results, however it can be the case that the layers obtained using linear splitting already give different shadow information, usually for scattered geometry or a low alpha value (*Figure 5.2.3.2*). Furthermore, in this case using logarithmic splitting would result in a fairly poor rendering, because the first layers would contain the same shadow information (*Figure 5.2.3.3* and *Figure 5.2.3.4*).

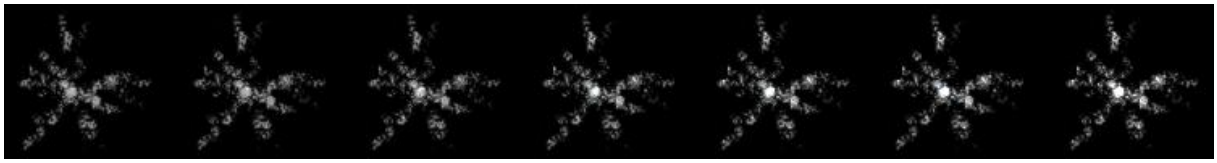

Figure 5.2.3.2 Layers obtained using linear splitting on a scene containing a scattered tree. Every layer adds new shadow information, i.e. is the different from the previous one.

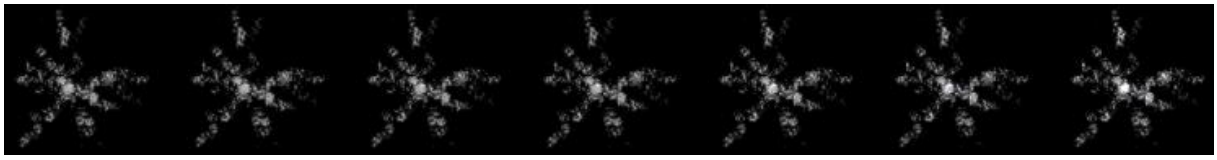

Figure 5.2.3.3 Layers obtained using logarithmic splitting on a scene containing a scattered tree. The first four layers contain almost the same shadow information.

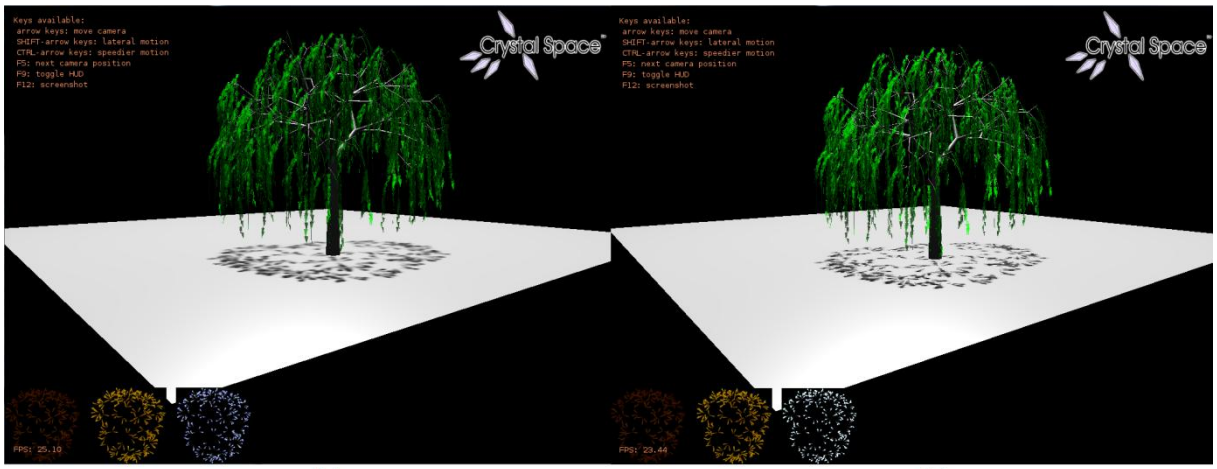

Figure 5.2.3.4 Picture from Crystal Space. Difference in rendering when splitting using logarithmic splitting (a) and linear splitting (b). Logarithmic splitting (a) produces artifacts, the willow is incorrectly lit near the top, because most of the layers contain the same information (*Figure 5.2.3.3*).

Even though neither of these two splitting scheme can be used for any type of object they complement each other, so when one scheme falls short the other can be used instead. Due to this property a hybrid splitting scheme has been proposed in the current thesis that is a mix of the linear and logarithmic splitting schemes.

The hybrid splitting function is built by doing a linear interpolation based on a variable that sets the ratio between the linear and logarithmic splitting. This means that when the variable is close to 0 the linear split is used, and when it is near 1 the logarithmic split is chosen instead (*Appendix - Section 9*).

Many interactive approaches, such as the ones presented in *Section 1.2*, find the best splitting points by first computing the visibility function, which is usually done using ray casting. However such a technique only gives interactive results, unless GPGPU capable hardware is used in which case Opacity Shadow Maps with a high number of maps (128) could also be used offering good visuals and performance.

The novel idea of this thesis is to determine an approximation of the visibility function for each ray by making use of the two contour maps and the opacity of the objects in the current scene. This is achieved by automatically determining the best ratio between the linear and logarithmic splitting from the hybrid splitting function.

The criteria for choosing the best ratio for various scenes with objects varying in density and with a different number of layers was that each new layer should capture information that the previous one did not.

Because the information that had to be measured was not contained in a single data, i.e. a single image, the concept of entropy [KK92] could not be used for the current problem. Instead ideas from computer vision regarding mutual information were applied [YG11].

For image registration, the mutual information between two images is computed using a similarity measure. Because each bounding opacity map contains the same objects, and therefore has the same shape, and only the transparency values differ, the image intensity is meaningful on a pixel by pixel basis. This means that simple similarity measures such as sum of absolute differences and correlation coefficient can be used.

The sum of absolute differences is fast and simple to compute and it involves iterating through the two pictures and summing the absolute differences on a pixel by pixel basis.

In the case of the correlation coefficient the following equation is evaluated:

$$C = \frac{\sum_{x,y}\left(I_1(x,y) - \bar{I_1}(x,y)\right)\left(I_2\left(T(x,y)\right) - \bar{I_2}\left(T(x,y)\right)\right)}{\sqrt{\sum_{x,y}\left(I_1(x,y) - \bar{I_1}(x,y)\right)^2 \sum_{x,y}\left(I_2\left(T(x,y)\right) - \bar{I_2}\left(T(x,y)\right)\right)^2}}$$

Equation 5.2.3.1 Defining the cross-correlation coefficient *C*

Where $\bar{I}(\cdot)$ is the mean intensity of image I. The above equation represents the ratio between the covariance of two images and the product of their standard deviation. The correlation has values on a scale ranging from [-1, 1] and it gives a linear indication of the similarity between images [YG11].

Even though the similarity measures presented above are usually used when trying to maximize the mutual information between images, they can also be successfully applied in order to minimize this mutual information. For instance in the case of the correlation coefficient a value close to 0 means that the two input images are different. Therefore the smallest value generated from different split ratios of the hybrid function corresponds to the value for which the layers have the least mutual information, so each new layer adds new information.

The way in which the split ratio varies for the two similarity measures, the sum of absolute differences and the correlation coefficient, according to the number of layers, the position of the light and the density of the translucent objects are presented in *Appendix – Section 11*.

Both similarity measures tend to have the same variation: the splitting becomes more linear when either the object is sparser (*Figure 5.2.3.5*) or the number of layers increases (*Figure 5.2.3.6*). These two variations can be intuitively explained.

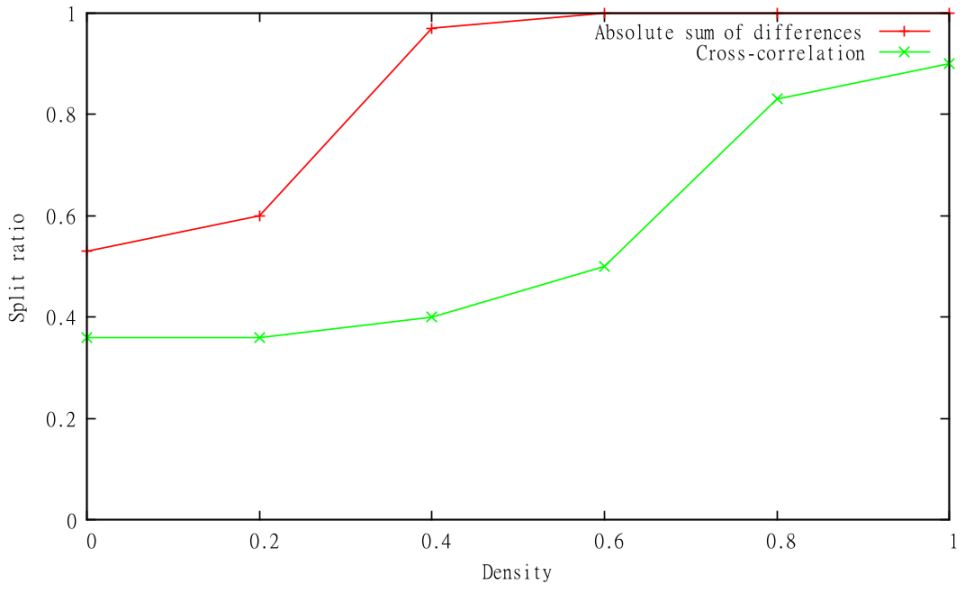

Figure 5.2.3.5 Plot generated using gnuplot. Sparser objects map better to linear splitting (the splitting ratio is closer to 0) and denser ones perform better when logarithmic splitting is used (the splitting ratio is closer to 1)

The fact that sparser objects map better to linear splitting and that denser ones perform better when logarithmic splitting is used has already been described in *Section 5.2*. The objects' density has been altered by either creating sparser versions of the same model or changing the way the light hits the object's surface from sideways to above. For instance, the grass model presented in *Appendix – Section 11* appears sparser when viewed from above and denser when viewed sideways.

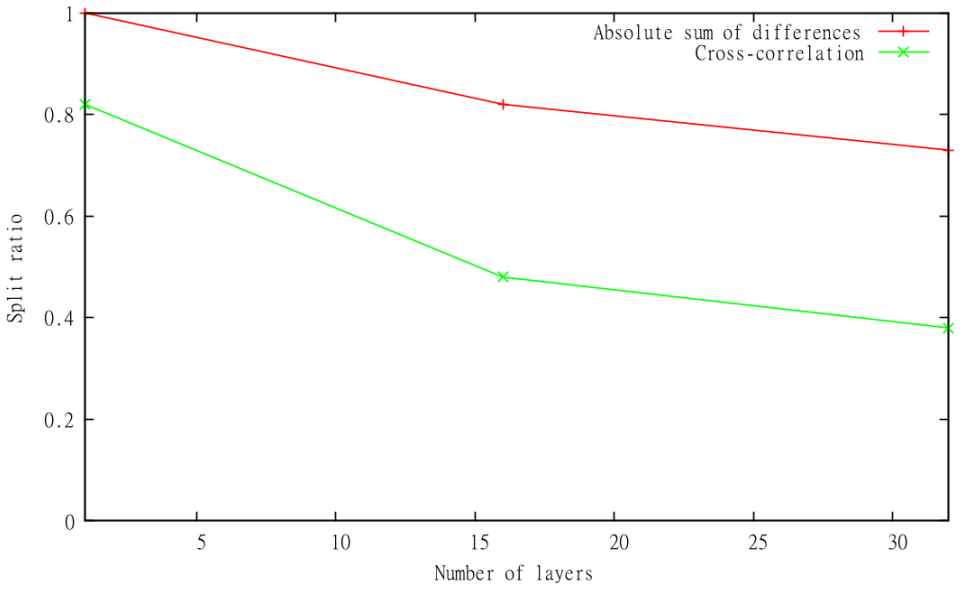

Figure 5.2.3.6 Plot generated using gnuplot. Splitting tends to be more linear (the splitting ratio is closer to 0) when the number of layers increases.

In the second case the splitting tends to be more linear when the number of layers increases because linear splitting with more layers has the first pivot points similar to a logarithmic distribution with fewer layers (*Figure 5.2.3.7*). This means that in both cases (*Figure 5.2.3.7.a*) and (*Figure 5.2.3.7.b*) the splitting tries to adapt itself to capture the most significant details of the object situated usually at the beginning and if further splits are available they are used to capture other possible details along the object as well.

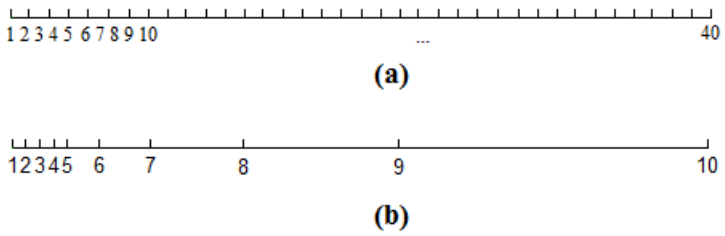

Figure 5.2.3.7 Comparison between linear and logarithmic distributions. Linear splitting with more layers (a) has the first pivot points similar to a logarithmic distribution with fewer layers (b).

Although the two similarity measures follow the same split ratio distribution the cross-correlation covers a wider range of values making it converge faster to the values that are to be expected from *Section 5.2*. This happens because the cross-correlation not only takes into account the difference of intensities between pixels, but their covariance as well. The problem with the sum of absolute differences is that the information provided by the variance is not used at all. Therefore a split ratio that produces a higher sum of absolute differences is chosen each time, without taking into account if the variance increases as well. This can lead to layers having a big covariance which is not desirable because each new layer should add new information. By doing this, cases where a few layers bring a lot of new information and the rest contain roughly the same data can be avoided.

Because the comparison between layers, or images, is done at a global level the render textures have to be readback from the GPU and processed on the CPU. The amount needed to do a readback, even when using optimized texture formats for doing such an operation, exceeds the real-time limits so this operation is not done on a frame by frame basis. However, this does not represent a limitation because there is no need to recompute the splitting ratio unless the density of the objects in the scene significantly changes. This can only happen when the object changes its density or the light hits the object's surface from an entirely different angle, both of which hardly ever occur.

It is important to note that the algorithm presented so far does not find the optimal number of layers and splitting ratio, but only finds the best splitting ratio for a given number of layers. It does this by choosing a more logarithmic split when fewer layers are available and a more linear one when a bigger number of layers is specified so that the significant details from the beginning of the object are always captured (*Figure 5.2.3.7*).

The reason why the number of layers has not been automatically chosen is that even though adding new layers increases the details in rendering it also always increases the cross-

correlation coefficient because the images become closer to one-another and therefore more correlated. Because there is a monotonic increase of the correlation-coefficient while increasing the number of layers (*Table 5.2.3.1*), the minimum or the maximum coefficient could not be used in the way it was used for choosing the optimal splitting ratio.

Scene / Layers	Trees		Grass		Dense grass	
	Correlation	**FPS**	**Correlation**	**FPS**	**Correlation**	**FPS**
4	0.66	98	0.72	65	0.72	49
8	0.83	96	0.86	64	0.86	47
16	0.92	80	0.93	53	0.93	43
32	0.96	60	0.96	40	0.96	34

Table 5.2.3.1 The variance between cross-correlation coefficient and performance (measured in FPS) on different scenes from Crystal Space (*Appendix – Section 11*). Increasing the number of layers causes a monotonic increase of the cross-correlation coefficient and a monotonic decrease in performance.

The only way of choosing the best number of layers is by computing a ratio between the rendering quality and the performance of the program. Even though the results from *Table 5.2.3.1* show a significant decrease in performance between the best and the worst FPS (frames-per-second), with an average of 50%, there is no straight-forward way to automatically measure the rendering quality of a scene. For this reason and because this is a real-time rendering engine designed for possible future games, choosing the optimal number of layers is done by the user. This can be done via a video graphic setting menu that sets the maximum number of multiple render targets from the values supported by the graphics card.

Algorithm 5.2.3.1 illustrates how a splitting scheme can be introduced to the general Bounding Opacity Maps algorithm (*Algorithm 5.1*)

```
1.  init:
2.    split_ratio := 0.0
3.    best_split := 0.0
4.    geometry := load_scene('scene_file.txt')
5.    recompute_split_ratio := true

6.  render_loop:
7.    start_depth_map : = render_pass(geometry)
8.    end_depth_map : = render_pass(geometry)
9.    render_textures : = render_pass(split_ratio, start_depth_map,
        end_depth_map, geometry)
10.   IF recompute_split_ratio = true
11.     best_split := process_image(split_ratio, best_split,
          render_textures, geometry)
12.   END
13.   render_scene(split_ratio, render_textures, start_depth_map,
        end_depth_map, geometry)
14.   IF recompute_split_ratio
15.     split_ratio := split_ratio + 0.1
16.   END
17.   IF split_ratio = 1.1
18.     split_ratio := best_split
19.     recompute_split_ratio := false
```

```
20.   END

21.  needs_recompute:
22.    recompute_split_ratio := true
23.    split_ratio := 0.0
```

Algorithm 5.2.3.1 Integrating a splitting scheme in the Bounding Opacity Maps method. When the hybrid split ratio needs to be updated, either the **`needs_recompute`** (line 21) function or the **`init`** (line 1) one have been called, the best split ratio is determined by the CPU implemented **`process_image`** (line 11) method. The optimal split ratio is determined from values between 0.0 and 1.0 increased by 0.1 (line 15), in a timespan of 11 frames (line 17).

5.2.4 Computing cross-correlation

Both algorithms that choose the optimal split ratio work with entire images and because of that they are implemented on CPU. In order to get the render textures back from the GPU several readbacks are done when the split ratio needs to be recomputed. Because readbacks are quite costly, the computation of the optimal split can become a bottleneck unless the sum of absolute differences or the cross-correlation coefficient are efficiently calculated, i.e. iterating only once on the given images. The sum of absolute difference is computed using only one iteration through the data set by its definition, but according to *Equation 5.2.3.1* it may appear that the cross-correlation needs several iterations. For instance the mean and the covariance from this equation might be computed each using a separate iteration and another final iteration can be added to compute the final division.

However, according to [GZY11] such equations, regarding cross-correlation or moments, can be done using only one pass if we consider the mean as a constant and simplify it from the sum. For instance, *Equation 5.2.3.1* can be split into three independent parts:

$$E_1 = \sum_{x,y} \big(I_1(x,y) - \bar{I_1}(x,y)\big)\big(I_2(x,y) - \bar{I_2}(x,y)\big)$$

$$E_2 = \sum_{x,y} \big(I_1(x,y) - \bar{I_1}(x,y)\big)^2$$

$$E_3 = \sum_{x,y} \big(I_2(x,y) - \bar{I_2}(x,y)\big)^2$$

And because in this case $T(x,y) = (x,y)$ the cross-correlation coefficient can be defined as:

$$C = \frac{E_1}{\sqrt{E_2 \cdot E_3}}$$

Using these definitions E_1, E_2 and E_3 still need the mean of the image, $\bar{I}(\cdot)$, in order to be computed. Applying the observation from [GZY11] E_1 becomes:

$$E_1 = \sum_{x,y} \big(I_1(x,y) - \overline{I_1}(x,y)\big)\big(I_2(x,y) - \overline{I_2}(x,y)\big)$$

$$= \sum_{x,y} \big(I_1(x,y) \cdot I_2(x,y) - I_1(x,y) \cdot \overline{I_2}(x,y) - \overline{I_1}(x,y) \cdot I_2(x,y) + \overline{I_1}(x,y) \cdot \overline{I_2}(x,y)\big)$$

$$= \sum_{x,y} I_1(x,y) \cdot I_2(x,y) - \overline{I_2}(x,y) \cdot \sum_{x,y} I_1(x,y) - \overline{I_1}(x,y) \cdot \sum_{x,y} I_2(x,y) + N \cdot \overline{I_1}(x,y) \cdot \overline{I_2}(x,y)$$

Where N is the image size and because

$$\sum_{x,y} I_1(x,y) = N \cdot \overline{I_1}(x,y)$$

The term E_1 can be further simplified:

$$E_1 =$$

$$= \sum_{x,y} I_1(x,y) \cdot I_2(x,y) - N \cdot \overline{I_2}(x,y) \cdot \overline{I_1}(x,y) - N \cdot \overline{I_1}(x,y) \cdot \overline{I_2}(x,y) + N \cdot \overline{I_1}(x,y) \cdot \overline{I_2}(x,y)$$

$$= \sum_{x,y} I_1(x,y) \cdot I_2(x,y) - N \cdot \overline{I_1}(x,y) \cdot \overline{I_2}(x,y)$$

In the above form E_1 can be calculated using only one loop over the image. Using the same idea E_2 can be simplified as well:

$$E_2 = \sum_{x,y} \big(I_1(x,y) - \overline{I_1}(x,y)\big)^2 = \sum_{x,y} (I_1(x,y)^2 - 2 \cdot I_1(x,y) \cdot \overline{I_1}(x,y) + \overline{I_1}(x,y)^2)$$

$$= \sum_{x,y} I_1(x,y)^2 - \sum_{x,y} 2 \cdot I_1(x,y) \cdot \overline{I_1}(x,y) + \sum_{x,y} \overline{I_1}(x,y)^2$$

$$= \sum_{x,y} I_1(x,y)^2 - 2 \cdot \overline{I_1}(x,y) \sum_{x,y} I_1(x,y) + \overline{I_1}(x,y)^2 \sum_{x,y} 1$$

$$= \sum_{x,y} I_1(x,y)^2 - 2 \cdot N \cdot \overline{I_1}(x,y) \cdot \overline{I_1}(x,y) + N \cdot \overline{I_1}(x,y)^2 = \sum_{x,y} I_1(x,y)^2 - N \cdot \overline{I_1}(x,y)^2$$

Similarly E_3 can be rewritten as:

$$E_3 = \sum_{x,y} I_2(x,y)^2 - N \cdot \overline{I_2}(x,y)^2$$

Using E_1, E_2 and E_3, the cross-correlation can be computed in only one iteration over the images retrieved from the GPU in which the following terms are calculated (*Appendix – Section 12*):

$$\sum_{x,y} I_1(x,y) \cdot I_2(x,y)\,, \overline{I_1}(x,y), \overline{I_2}(x,y), \sum_{x,y} I_1(x,y)^2 \text{ and } \sum_{x,y} I_2(x,y)^2$$

5.3 Adding common lighting effects

Because the techniques described so far only determine how the light contributes to the self-shadowing of a translucent object, similar to the diffuse lighting term in the Blinn-Phong shading model [B77], the other terms had to be added as well.

In the case of opaque objects all the terms from Blinn-Phong are applied, and for translucent objects the diffuse lighting is replaced by the information given by the self-shadowing technique (*Appendix - Section 10*).

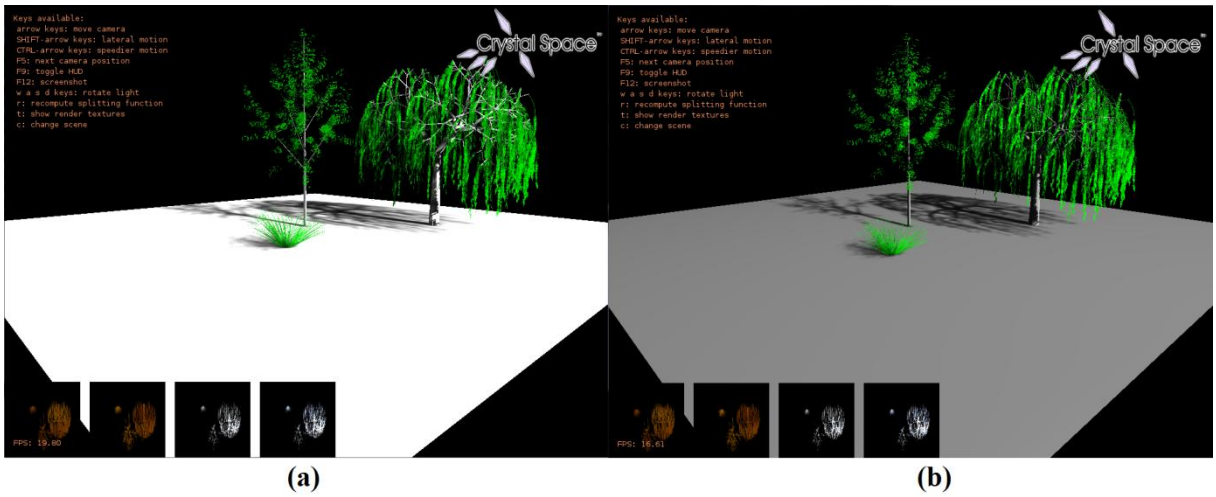

Figure 5.3.1 Picture from Crystal Space. Difference in rendering when using Blinn-Phong terms for opaque objects (b) and when using only self-shadowing information from translucent objects (a).

Moreover, opaque objects had to be introduced in the computation of the shadow maps. However, because a depth shadow map was already used in computing the starting splitting points for translucent objects, making opaque objects cast shadows was easily introduced by adding the opaque objects in the already existing depth shadow map. Even though this depth map could have been used directly for testing if an object is shadowed or not, the depth information was also given to the shadow opacity maps so that all objects would cast shadows using the same technique (*Figure 5.3.2*).

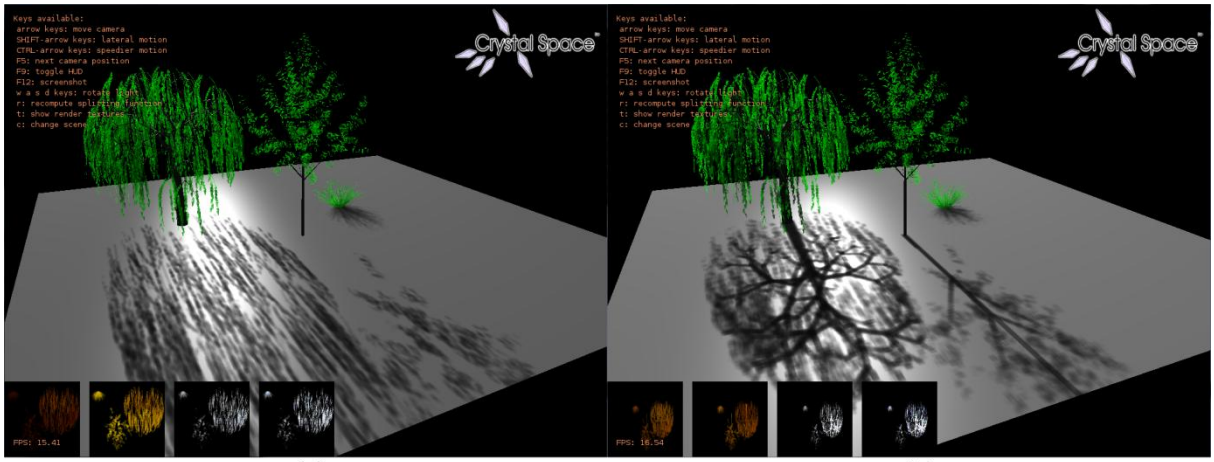

Figure 5.3.2 Picture from Crystal Space. Difference in rendering when opaque objects, the trees' trunk and branches, do not cast shadows (a) and do cast shadows (b).

5.4 Limitations

The limitations of the bounding opacity maps are given by the splitting methods used, either linear, logarithmic or a hybrid of the first two.

A limitation is that only one splitting function is chosen for the whole scene, and this is caused by the fact that the splitting ratio is determined using entire images and not only pixels or regions. Because of this a scene containing a dense object and a sparse one will be treated as a scene which contains an object which is neither dense nor sparse yielding an inadequate rendering.

In the case of some objects that change their density, like smoke for instance which is a translucent model frequently used in real-time applications such as games, the splitting ratio needs to be recomputed. This computation involves texture readbacks and image processing both done on the CPU, in the current implementation, which makes the application turn from real-time to interactive for a couple of seconds. Furthermore the splitting ratio also has to be recomputed when the light changes its position and other models with different translucency levels are brought into the light's perspective.

A last limitation of the splitting method is related to the fact that the number of layers used has to be specified, via the number of available multiple render targets, by the user. This can lead to an inefficient usage of the hardware, by doing a lot more computations than are actually needed in order to produce the same rendering. A better method would be to determine the optimal number of layers and check if they are available on the hardware the application is running on.

6. Results & Performance

In this section the main three algorithms presented and implemented in this thesis, Opacity Shadow Maps, Deep Opacity Maps and Bounding Opacity Maps are analyzed regarding their performance and visual aspect according to the number of layers used.

The performance is measured in frames-per-second (FPS) using the average FPS provided by Fraps [FRAPS] benchmarking tool for a period of 60 seconds. The measures were taken on a Dell Alienware M11x [M11x] having the following hardware components:

- **Processor:** Intel Pentium Dual-Core, 1.3 GHz
- **Memory:** DDR3 SDRAM, 4 GB
- **Graphics:** NVIDIA GT 335M, 1 GB

Moreover the application was tested on the following software configuration:

- **Rendering resolution:** 1024x768
- **Render target's resolution:** 512x512
- **Operating System:** Windows 7 Home Premium, Service Pack 1
- **Compiler / IDE:** Microsoft Visual Studio 2010 Professional, DebugWithDlls Win32.

Because all three algorithms are GPU bound, they involve rendering the scene multiple times and almost no task is done on CPU, the GPU card had the most important contribution in the measurements taken.

6.1 Opacity Shadow Maps

As can be seen from *Table 6.1.1* Opacity Shadow Maps represent a fast way of obtaining volumetric shadow maps, although they suffer from serious visual artifacts as described in *Section 3.2* and *Section 3.3*.

Scene FPS / Layers	Trees	Grass	Dense grass
4	147.58	137.53	107.76
8	124.85	110.00	79.91
16	86.68	73.87	59.53
32	53.80	39.83	36.58

Table 6.1.1 The variance between the number of layers used and the performance measured in FPS, for Opacity Shadow Maps.

Whereas Deep Opacity Maps and Bounding Opacity Maps have entire maps for the start and the end splitting points, Opacity Shadow Maps only use one start splitting point and one end splitting point, which gives the algorithm a good performance, but poor rendering results. In

order to keep the Opacity Shadow Map implementation GPU bond, the start and the end splitting points were obtained by using bounding boxes and not iterating through entire 3D models.

6.2 Deep Opacity Maps

Deep Opacity Maps represent a compromise between performance and visual aspect, fixing the sever artifacts from Opacity Shadow Maps, while having a better FPS than Bounding Opacity Maps (*Table 6.2.1*).

Scene FPS / Layers	Trees	Grass	Dense grass
4	112.10	77.38	70.53
8	90.66	60.56	61.23
16	63.96	47.93	41.85
32	43.90	29.45	27.28

Table 6.2.1 The variance between the number of layers used and the performance measured in FPS, for Deep Opacity Maps.

6.3 Bounding Opacity Maps

Even though Deep Opacity Maps might have a better performance than Bounding Opacity Maps (*Table 6.3.1*) when using the same number of layers, the latter method produces better visuals with fewer layers. This happens because Bounding Opacity Maps give a better bounding of the object, there is a depth map for the end splitting points as well, and on top of that if the hybrid split is used better renderings are obtained with even fewer layers.

Scene FPS / Layers	Trees	Grass	Dense grass
4	73.41	61.18	49.80
8	67.53	55.13	43.61
16	58.91	46.96	34.66
32	41.88	31.93	26.80

Table 6.3.1 The variance between the number of layers used and the performance measured in FPS, for Bounding Opacity Maps.

An overall comparison between the three algorithms implemented in this thesis is show in *Figure 6.3.1*. It can be seen that for a larger number of layers the Bounding Opacity Maps method tends to have the same performance as Deep Opacity Maps, because the extra render pass, computing the end splitting points, becomes computationally insignificant.

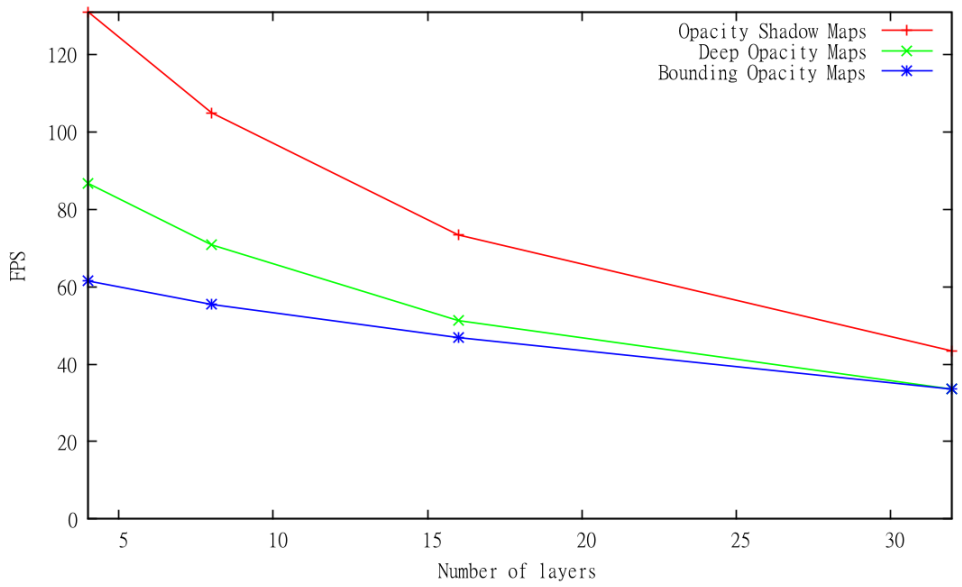

Figure 6.3.1 Plot generated using gnuplot. The variance between the number of layers and the FPS for Opacity Shadow Maps (red), Deep Opacity Maps (green) and Bounding Opacity Maps (blue). Even though Deep Opacity Maps have better performance than Bounding Opacity Maps, the latter method produces better visuals with fewer layers.

7. Future work

In this section various approaches that try to fix the limitations presented in *Section 5.4* are proposed.

Using a different split ratio for each individual ray can solve the fact that for the current version there is only one splitting function for the whole scene, causing incorrect renderings when multiple objects with different densities are present. This can be done by creating a splitting texture (*Figure 7.1.b*), which will store the split ratio for each individual ray in a different pixel.

A straight forward way to produce such a texture is by using the sum of absolute differences method of splitting on individual pixels, or regions (*Figure 7.1.c*), rather than on whole images.

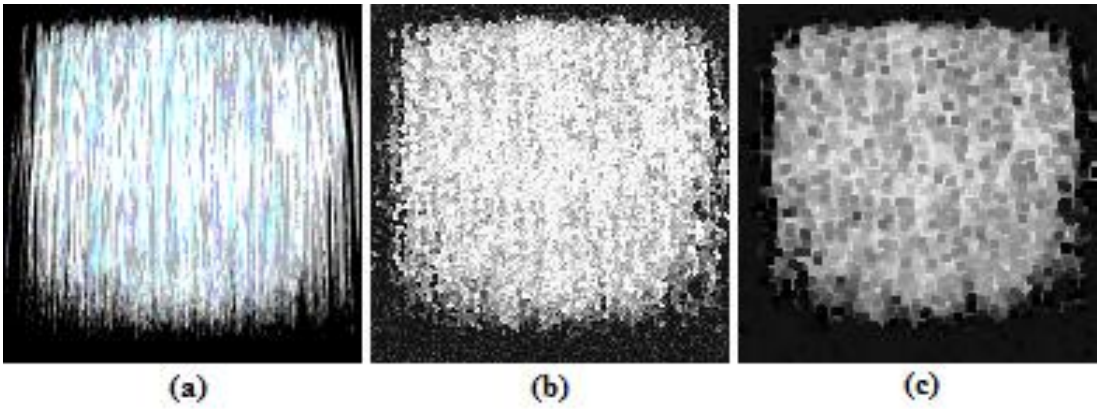

Figure 7.1 Picture created with paint.net. Estimate of how the splitting texture would look for individual pixels (b) and for regions (c) for some given layers (a). A dark color in the splitting texture (b) corresponds to a linear split while a bright one represents a more logarithmic split for the corresponding ray / pixel.

However a disadvantage of splitting on individual rays is that in order for each splitting ratio to always be up-to-date it would have to be recomputed every time a translucent object moves or the light changes its position. These events occur quite frequently unlike the ones from the global splitting function which took place only when an object changed its density or objects with new densities were brought into the light's perspective. Because of this the splitting texture would have to be computed on a frame by frame basis.

Unfortunately this would not be possible for the current implementation because recomputing the global splitting function already represents a bottleneck causing the application to have only interactive performance when such a task is performed. Furthermore, adding another texture read and more importantly a texture write, necessarily for updating the splitting texture, would only result in a decrease in performance.

Luckily because only local information is needed from the layers, the image processing step does not have to be done on the CPU and so no costly texture reads or writes are necessary. The splitting texture can be computed on the GPU by adding a new render pass built like a post-processing shader [JRP95], were both the input and the output are images and fragments from the fragment program represent a pixel in the image space.

For the splitting texture problem the input images are render textures obtained when using a predefined splitting ratio, having values from 0.0 to 1.0 using a 0.1 step. The output for each fragment is the optimal sum of absolute differences and the corresponding optimal splitting value found so far, encoded in two different texture channels. Because finding an optional split ratio takes 11 frames, the number of values from 0.0 to 1.0 using a 0.1 step, two copies of the render textures should be kept: one used for rendering and the other one for finding the future best split ratio. Once 11 frames have passed the opacity layers used for rendering can be recomputed so that they are determined by the newly found optimal split texture, and the process of finding the optimal split can start once more (*Algorithm 7.1*).

```
1. init:
2.   split_ratio := 0.0
3.   best_split_texture := 0.0
4.   geometry := load_scene('scene_file.txt')

5. render_loop:
6.   start_depth_map := render_pass(geometry)
7.   end_depth_map := render_pass(geometry)
8.   [backup_render_textures, split_texture] :=
       render_pass(split_ratio, split_texture, start_depth_map,
       end_depth_map, geometry)
9.   render_textures := render_pass(best_split_texture,
       start_depth_map, end_depth_map, geometry)
10.  render_scene(render_textures, best_split_texture,
       start_depth_map, end_depth_map, geometry)
11.  split_ratio := split_ratio + 0.1
12.  IF split_ratio = 1.1
13.    split_ratio := 0.0
14.    best_split_texture := split_texture
15.  END
```

Algorithm 7.1 Extending Bounding Opacity Maps to support splitting textures. The individual splitting ratios, defined on line 3, are updated every 11 frames (line 12) so there is no need for a **`needs_recompute`** function. Because the splitting ratios are permanently updated the effects that happen when using the predefined split ratios, between 0.0 and 1.0 (line 11), must not be rendered, so another set of render textures is used (line 8). This algorithm should have a real-time performance because the best splitting ratios are computed on the GPU and not on the CPU (line 8).

Although adding another render pass to compute the splitting texture will probably result in a decrease in performance this will also solve the rendering inconsistencies for scenes having object with different density levels. Furthermore, because the individual splitting ratios will be permanently updated on the GPU there will be no CPU bottleneck when the scene will be updated and eligibly the application will run in real-time.

The last limitation found for the splitting method is that it doesn't compute the optimal number of layers, but only the optimal splitting ratio. Because as stated in *Section 5.3.3* it is hard to determine a ratio between the rendering quality and the performance, choosing the optimal number of layers could be done only by taking into account the performance which has to be real-time, i.e. more than 30 FPS. In order to do this a benchmarking fly-through scene in which the number of layers is increased as long as it keeps the application real-time could be used, similar to the one from Half Life 2: Lost Coast Benchmark [HL2LC05]. Because each new layer adds new details the maximum number of layers is actually the optimal number of layers which keeps the application real-time on the hardware configuration the scene was run.

Although the current implementation works with foliage, like trees and grass, and hair, support for other translucent objects, like clouds and smoke, could also be added as another possible future improvement. The reason why these types of translucent object are not currently supported is because they are usually rendered using volumetric rendering which is

yet to be implemented in Crystal Space. With the aid of volumetric rendering clouds could be simulated using metaballs, as describe in "A Simple, Efficient Method for Realistic Animation of Clouds" [DKY00]. The metaballs rendering consists of approximating the cloud volume with a set of spheres which are then used to test the intersection with rays shoot from the light's direction. However, this cannot be achieved with a regular, non-volumetric approach because the spheres would be rendered only as surfaces and not as volumes (*Figure 7.2*).

Figure 7.2 Picture taken from Crystal Space. Using non volumetric rendering techniques produces incorrect renderings because the metaballs spheres are rendered as surfaces and not volumes, both when rendering the final scene and when rendering to texture.

8. Conclusions

Translucent objects, like foliage or hair, need to cast shadows on themselves in order to produce realistic renderings.

The first method that produces realistic self-shadows in real-time for such translucent object appeared in 2000 and it is called Opacity Shadow Maps. The main disadvantage of this algorithm is that it is artifact free only when a lot of opacity maps are used, causing the application to exceed the real-time limits.

Deep Opacity Maps produce quality renderings with just a few layers by aligning the opacity maps with the start shape of the geometry as seen from the light's perspective. However, because no information regarding the end position or shape of the objects is provided this method cannot be used on scenes having objects of different sizes.

Bounding Opacity Maps, a novel approach proposed in this thesis, fixes the limitations of Deep Opacity Maps by specify a bounding volume, via two depth maps, for all translucent objects in the scene. Moreover, various splitting schemes, such as linear or logarithmic, have been tried in order to better determine the splitting positions. In the end a hybrid splitting function that is more linear for sparse object and more logarithmic for dense ones was considered the best choice.

Individual split functions and real-time updates are still to be added, as well as support for finding the optimal number of splitting points and support for rendering other types of translucent objects, such as clouds or smoke.

References

[KK89] James Kajiya, Timothy Kay "Rendering Fur with Three Dimensional Textures". *Proceedings of ACM SIGGRAPH 1989.*

[MJC03] Stephen Marschner, Henrik Wann Jensen, Mike Cammarano "Light Scattering from Human Hair Fibers". *Proceedings of SIGGRAPH 2003.*

[LV00] Tom Lokovic. Eric Veach "Deep Shadow Maps". *Proceedings SIGGRAPH 2000.*

[L88] Marc Levoy "Display of Surfaces from Volume Data". *June 1987.*

[KN01] Tae-Yong Kim, Ulrich Neumann "Opacity Shadow Maps". *Eurographics Rendering Workshop 2001*.

[MKBR04] Tom Mertens, Jan Kautz, Philippe Bekaert, Frank Van Reethy "A Self-Shadow Algorithm for Dynamic Hair using Density Clustering". *Eurographics Symposium on Rendering 2004.*

[LB10] Jon Jansen, Louis Bavoil "Fourier Opacity Mapping". *Proceedings I3D 2010 conference.*

[Llo82] S. Lloyd "Least squares quantization in PCM". *IEEE Trans. on Information Theory 28*, 2 (1982), 127.138. 3, 5

[CUDA07] Compute Unified Device Architecture 2007 - http://www.nvidia.com/object/cuda_home_new.html

[SA07] Erik Sintorn, Ulf Assarsson "Fast Parallel GPU-Sorting Using a Hybrid Algorithm". *Journal of Parallel and Distributed Computing*, Volume 68, Issue 10, Pages 1381-1388, October 2008.

[SA08] Erik Sintorn, Ulf Assarsson "Real-Time Approximate Sorting for Self Shadowing and Transparency in Hair Rendering". *Proceedings of I3D 2008*, pp 157-162, February, 2008.

[QSORT] C/C++ quicksort implementation - http://linux.die.net/man/3/qsort

[V10] Alexandru Teodor Voicu "Rendering real-time self-shadowed dynamic hair". *Independent Study Option, Imperial College London*, May 2010.

[YK08] Cem Yuksel, John Keyser "Deep Opacity Maps". *Eurographis 2008.*

[CS07] Crystal Space - http://www.crystalspace3d.org

[CSM08] Crystal Space Online Manual, Version 1.9.0, 2008: http://www.crystalspace3d.org/docs/online/manual/.

[CSD08] Crystal Space Documentation, Public API Reference, Version 1.9.0, 2008 http://www.crystalspace3d.org/docs/online/api/.

[CS11] Crystal Space Google Summer of Code 2011 selfshadow branch - https://crystal.svn.sourceforge.net/svnroot/crystal/CS/branches/soc2011/selfshadow/

[H04] Shawn Hargreaves "Deferred Shading". *GDC 2004 climax.*

[ZFSXS06] Zhang, Fan, Hanqiu Sun, Leilei Xu, and Kit-Lun Lee. 2006. "Parallel-Split Shadow Maps for Large-Scale Virtual Environments." *Proceedings of ACM International Conference on Virtual Reality Continuum and Its Applications 2006*, pp. 311–318.

[D07] Rouslan Dimitrov "Cascaded Shadow Maps". *NVIDIA Corporation*, August 2007.

[SKWFH92] M. Segal, C. Korobkin, Rolf van Widenfelt, J. Foran, P. Haeberli "Fast Shadows and Lighting Effects Using Texture Mapping". *Computer Graphics (Proceedings of SIGGRAPH '92)*, July 1992.

[ZSN07] Fan Zhang, Hanqiu Sun, Oskari Nyman "Parallel-Split Shadow Maps on Programmable GPUs". *GPU Gems 3, Chapter 10*, December 2007.

[OGL97] OpenGL 2.1 Reference Pages, http://www.opengl.org/sdk/docs/man/xhtml

[ND05] Hubert Nguyen, William Donnelly "Hair Animation and Rendering in the Nalu Demo". *GPU Gems 2, Chapter 23*, April 2007.

[MRT04] Multiple Render Targets, Pixel Pipeline, MSDN Library 2004 http://msdn.microsoft.com/en-us/library/

[DL06] William Donnelly, Andrew Lauritzen "Variance Shadow Maps". *Proceedings of the 2006 symposium on Interactive 3D graphics and games, ACM*, 2006.

[RSC87] Reeves W. T., Salesin D., Cook R. L.: Rendering antialiased shadows with depth maps. *Computer Graphics (Proceedings of SIGGRAPH '87)* (1987), 283–291.

[BP04] Michael Bunnell, Fabio Pellacini "Shadow Map Antialiasing". *GPU Gems, Chapter 11*, September 2004.

[tex2D] Cg 3.0 Toolkit Documentation, Cg Standard Library, NVIDIA February 2011 - http://http.developer.nvidia.com/Cg/tex2D.html

[SL04] Sebastien St-Laurent "Shaders for Game Programmers and Artists". *Chapter 6*, 2004.

[CW04] Chris Wynn "OpenGL Render-to-Texture". *NVIDIA Corporation 2004*.

[B77] J. F. Blinn "Models of Light Reflection for Computer Synthesized Pictures". *Proceedings of SIGGRAPH 1977*, volume 11, pp 192-198.

[KK92] J. N. Kapur, H. K. Kesavan "Entropy Optimization Principles with Applications". May 1992.

[YG11] G. Z. Yang, D. F. Gillies "Image Registration". *Compuer Vision 13th Course, Imperial College London*, November 2011.

[GZY11] G. Z. Yang, "Moments". *Compuer Vision 6th Tutorial, Imperial College London*, November 2011.

[M11x] Dell Alienware M11x Gaming Laptop, First Generation http://www.dell.com/us/p/alienware-m11x/pd

[FRAPS] Fraps, Real-Time video capture & benchmarking - http://www.fraps.com/

[JRP95] J.R. Parker, "A Shaer Memory System – Using the PC Graphics Processor". *Algorithms for Image Processing and Computer Vision, Second Edition*, Chapter 11, pp 444-459, November 1995

[HL2LC05] Half-Life 2: Lost Coast, Valve Software, Steam, October 2005.

[DKY00] Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita, and T. Nishita, "A simple, efficient method for realistic animation of clouds". In *Proceedings of ACM SIGGRAPH 2000*, pp 19–28.

Appendix

Section 1

Excerpt from the shader generating 4 opacity maps in one texture by using all 4 available channels.

```
float value = surface.a;
value = value * (surface.a != 1);

int i;
for (i = 0 ; i <= numSplits ; i ++)
      if (passColor[i] > positionCamera.z)
            break;

int index = i % 4;
float4 color = value;
color = color * float4(index < 1, index < 2, index < 3, 1);

output = color;
```

Section 2

Excerpt from the shader generating the final opacity function by adding information from all precedent render textures.

```
float previousMap = 0, nextMap = 0;

int prevIndex = min((i / 4), numSplits);
float2 prevPos = getPosition(prevIndex);

for (int j = 0 ; j < prevIndex ; j ++)
      previousMap += getMapValue(4 * (j + 1) - 1, prevPos);

int nextIndex = min((i + 1) / 4, numSplits);
float2 nextPos = getPosition(nextIndex);

nextMap = previousMap;
for (int j = prevIndex ; j < nextIndex ; j ++)
      nextMap += getMapValue(4 * (j + 1) - 1, nextPos);

previousMap += getMapValue(i, prevPos);
nextMap += getMapValue(i + 1, nextPos);

inLight = lerp(previousMap, nextMap, (float) (viewPos.z -
      previousSplit) / (nextSplit - previousSplit) );

inLight = inLight * (i != numSplits) + previousMap * (i ==
      numSplits);
inLight = exp(-1.0 * inLight);

return inLight;
```

Section 3

Excerpt from the shader accessing and writing only to valid render targets, depending on the number of render targets actually supported by the video card.

```
<![CDATA[
  struct fragmentOutput
  {
    float4 Color0 : COLOR0;
]]>
<?if vars."mrt".int &gt; 1?>
  <![CDATA[
    float4 Color1 : COLOR1;
    float4 Color2 : COLOR2;
    float4 Color3 : COLOR3;
  ]]>
<?if vars."mrt".int &gt; 4?>
  <![CDATA[
    float4 Color4 : COLOR4;
    float4 Color5 : COLOR5;
    float4 Color6 : COLOR6;
    float4 Color7 : COLOR7;
  ]]>
<?endif?>
<?endif?>
<![CDATA[
    };
]]>

<![CDATA[
  float value = surface.a;
  value = value * (surface.a != 1);

  int i;
  for (i = 0 ; i <= numSplits ; i ++)
    if (passColor[i] > IN.position_camera.z)
      break;

  int index = i % 4;
  float4 color = value;
  color = color * float4(index < 1, index < 2, index < 3, 1);

  // write in the correct render target
  int renderTarget = (i / 4) % mrt;

  output.Color0 = color * (renderTarget == 0);
]]>
<?if vars."mrt".int &gt; 1?>
<![CDATA[
  output.Color1 = color * (renderTarget == 1);
  output.Color2 = color * (renderTarget == 2);
  output.Color3 = color * (renderTarget == 3);
]]>
<?if vars."mrt".int &gt; 4?>
<![CDATA[
  output.Color4 = color * (renderTarget == 4);
  output.Color5 = color * (renderTarget == 5);
  output.Color6 = color * (renderTarget == 6);
  output.Color7 = color * (renderTarget == 7);
]]>
<?endif?>
```

```
<?endif?>
<![CDATA[
  return output;
}
]]>
```

Section 4

Code from [BP04], for doing a brute force percentage-closer filter o 4x4 texel.

```
float3 offset_lookup(sampler2D map,
                     float4 loc,
                     float2 offset)
{
  return tex2Dproj(map, float4(loc.xy + offset * texmapscale * loc.w,
    loc.z, loc.w));
}

float sum = 0;
float x, y;

for (y = -1.5; y <= 1.5; y += 1.0)
  for (x = -1.5; x <= 1.5; x += 1.0)
    sum += offset_lookup(shadowmap, shadowCoord, float2(x, y));
```

Section 5

Excerpt from the second rendering pass, generating the depth opacity maps using multiple targets. Instead of storing information just between two splitting points, the layers save information from all previous splitting points.

```
<![CDATA[
  output.Color0 = color * (renderTarget == 0);
]]>
<?if vars."mrt".int &gt; 1?>
<![CDATA[
  output.Color1 = color * (renderTarget == 1);
  output.Color2 = color * (renderTarget == 2);
  output.Color3 = color * (renderTarget == 3);
]]>
<?if vars."mrt".int &gt; 4?>
<![CDATA[
  output.Color4 = color * (renderTarget == 4);
  output.Color5 = color * (renderTarget == 5);
  output.Color6 = color * (renderTarget == 6);
  output.Color7 = color * (renderTarget == 7);
]]>
<?endif?>
<?endif?>
<![CDATA[
  return output;
}
]]>
```

The above section is replaced without introducing any additional computational cost by:

```
<![CDATA[
  output.Color0 = color * (renderTarget <= 0);
]]>
<?if vars."mrt".int &gt; 1?>
<![CDATA[
  output.Color1 = color * (renderTarget <= 1);
  output.Color2 = color * (renderTarget <= 2);
  output.Color3 = color * (renderTarget <= 3);
]]>
<?if vars."mrt".int &gt; 4?>
<![CDATA[
  output.Color4 = color * (renderTarget <= 4);
  output.Color5 = color * (renderTarget <= 5);
  output.Color6 = color * (renderTarget <= 6);
  output.Color7 = color * (renderTarget <= 7);
]]>
<?endif?>
<?endif?>
<![CDATA[
  return output;
}
]]>
```

Section 6

Excerpt from the third rendering pass from Bounding Opacity Maps. The splitting points are determined by a linear interpolation between the values read from the two depth maps.

```
<![CDATA[
  fragmentOutput main (vertex2fragment IN,
    uniform sampler2D TexDiffuse,
    uniform sampler2D DepthStartMap,
    uniform sampler2D DepthEndMap)
  {
    fragmentOutput output;
    float4 surface = tex2D (TexDiffuse, IN.TexCoord);
    float value = surface.a;

    // compute the projection for the depthStart map
    float3                    shadowMapCoordsBiased                    =
      (float3(0.5)*IN.shadowMapCoordsProj.xyz) + float3(0.5);
    float2 position = shadowMapCoordsBiased.xy;

    float compareDepth = (1 - shadowMapCoordsBiased.z) ;
    float depthStart = tex2D(DepthStartMap, position).x;
    float depthEnd = tex2D(DepthEndMap, position).x;

    int layer;
    layer = min( ( (compareDepth - depthStart) / (1 - depthEnd -
      depthStart) ) * (numSplits - 1), numSplits - 1);

    ...

    return output;
  }
]]>
```

Section 7

Excerpt from the initialization stage where the lookup texture encoding the logarithmic splitting is computed.

```
double end = textureSize - 1;
double range = (int)(log(end - 1.0)/log(2.0));
double start = end - range;

data[0] = 0;
for (int i = 4 ; i < 4 * textureSize ; i += 4)
{
  data[i] = (unsigned char)csMin( ( (log(pow(2.0, start) * (i / 4.0))
    / log(2.0) - start) * end / range ) , end) * 255 / end;
}
```

Section 8

Excerpt from the third rendering pass, which uses only a lookup texture instead of numerous arithmetic operations.

```
<![CDATA[
  fragmentOutput main (vertex2fragment IN,
    uniform sampler2D TexDiffuse,
    uniform sampler2D DepthStartMap,
    uniform sampler2D DepthEndMap,
    uniform sampler2D SplitFunc)
  {
    fragmentOutput output;
    float4 surface = tex2D (TexDiffuse, IN.TexCoord);
    float value = surface.a;

    // compute the projection for the depthStart map
    float3                    shadowMapCoordsBiased                    =
      (float3(0.5)*IN.shadowMapCoordsProj.xyz) + float3(0.5);
    float2 position = shadowMapCoordsBiased.xy;

    float compareDepth = (1 - shadowMapCoordsBiased.z) ;
    float depthStart = tex2D(DepthStartMap, position).x;
    float depthEnd = tex2D(DepthEndMap, position).x;

    int layer;
    layer = min( tex2D( SplitFunc, float2( min( (compareDepth -
      depthStart) / (1 - depthEnd - depthStart) , 0.9999 ) , 0 ) ).x
      * (numSplits - 1), numSplits - 1);

    ...

    return output;
  }
]]>
```

Section 9

Excerpt from the initialization stage where the lookup texture encoding the hybrid splitting is computed. Values near 0 for the logValue variable correspond to linear splitting, while values close to 1 yield a logarithmic distribution of the layers.

```
double end = textureSize - 1;
double range = (int)(log(end - 1.0)/log(2.0));
double start = end - range;

data[0] = 0;
for (int i = 4 ; i < 4 * textureSize ; i += 4)
{
  data[i] = (unsigned char)(csMin( (1 - logValue) * i / 4 + logValue
    * ( (log(pow(2.0, start) * (i / 4.0)) / log(2.0) - start) * end /
    range ) , end) * 255 / end);
}
```

Section 10

Excerpt from the lighting shader that applies the diffuse term from the Blinn-Phong shading model for opaque objects and the self-shadowing term for translucent ones.

```
LightSpaceWorld lightSpace;
lightSpace.Init (i, positionW);

shadow.Init (l, shadowMapCoords[l], lightDir[l].w);

float3 lightDiffuse = lightProps.colorDiffuse[i];

half shadowFactor = shadow.GetVisibility();
Light light = GetCurrentLight (lightSpace, i);
float4 lightAttenuationVec = lightProps.attenuationVec[i];
float3 d, s;
ComputeLight (lightSpace, light, myEyeToSurf, normal, shininess,
  lightDiffuse, lightProps.colorSpecular[i],
  lightAttenuationVec, shadowFactor, d, s);

// translucent objects do not have diffuse light
diffuseColor += shadowFactor * (surface.a != 1) + d * (surface.a ==
1);
specularColor += s;
```

Section 11

This section of the Appendix contains results regarding the way the split ratio modifies for objects with different opacities and different number of layers. The two splitting method tested at a 512x512 layers' resolution were sum of absolute differences and cross-correlation coefficient.

Split ratio (*Table A.11.1*) computed for a scene composed of a single tree viewed from above (*Figure A.11.1*).

Figure A.11.1 Picture taken from Crystal Space

Layers / Split method	1	16	32
Sum of absolute differences	1.0	0.4	0.2
Correlation coefficient	0.7	0.2	0.2

Table A.11.1 Split ratio variation for the scene in *Figure A.11.1*

Split ratio (*Table A.11.2*) computed for a scene composed of a single tree viewed from sideways (*Figure A.11.2*).

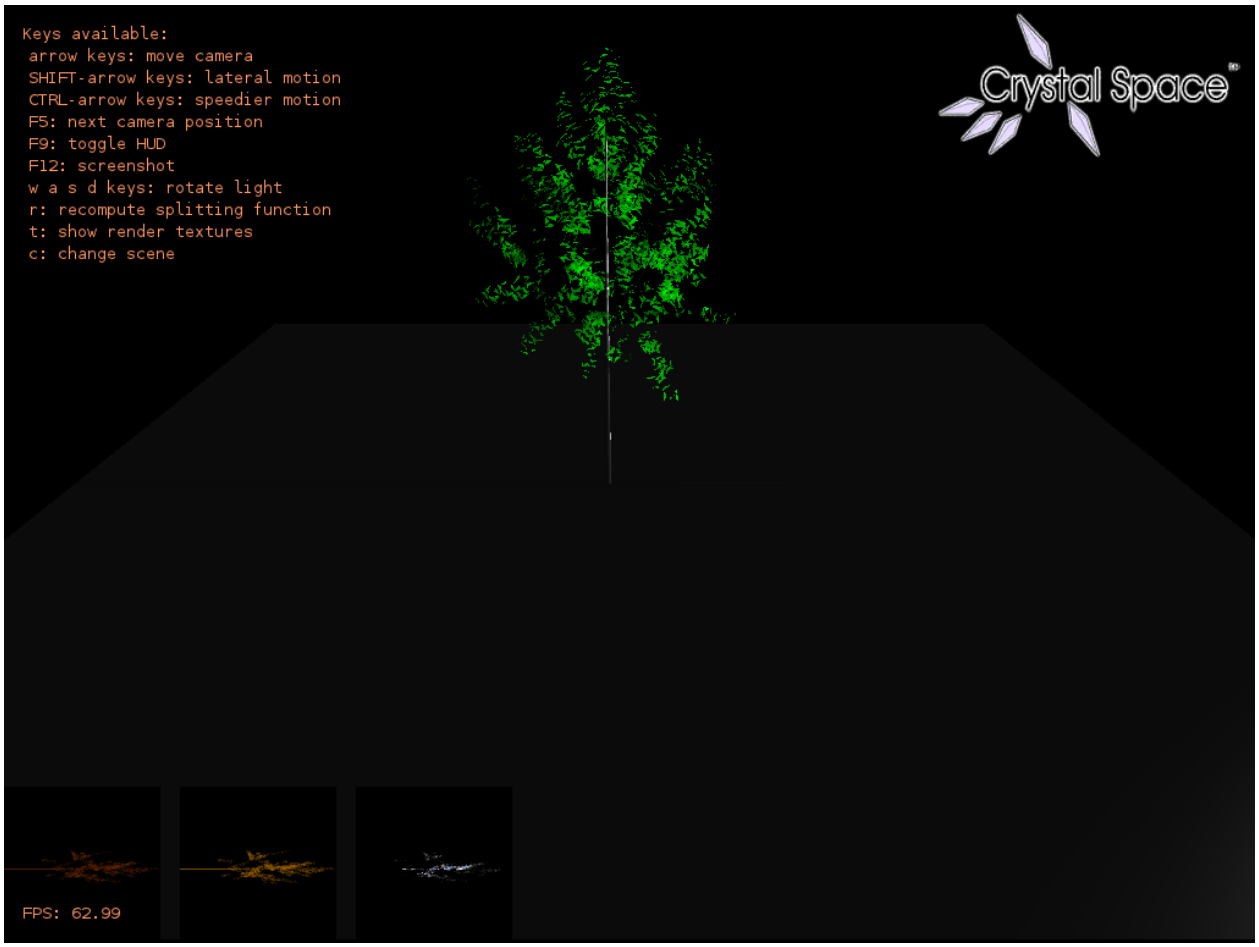

Figure A.11.2 Picture taken from Crystal Space

Split method \ Layers	1	16	32
Sum of absolute differences	1.0	0.6	0.2
Cross-correlation	0.7	0.3	0.1

Table A.11.2 Split ratio variation for the scene in *Figure A.11.2*

Split ratio (*Table A.11.3*) computed for a scene composed of a grass model viewed from above (*Figure A.11.3*).

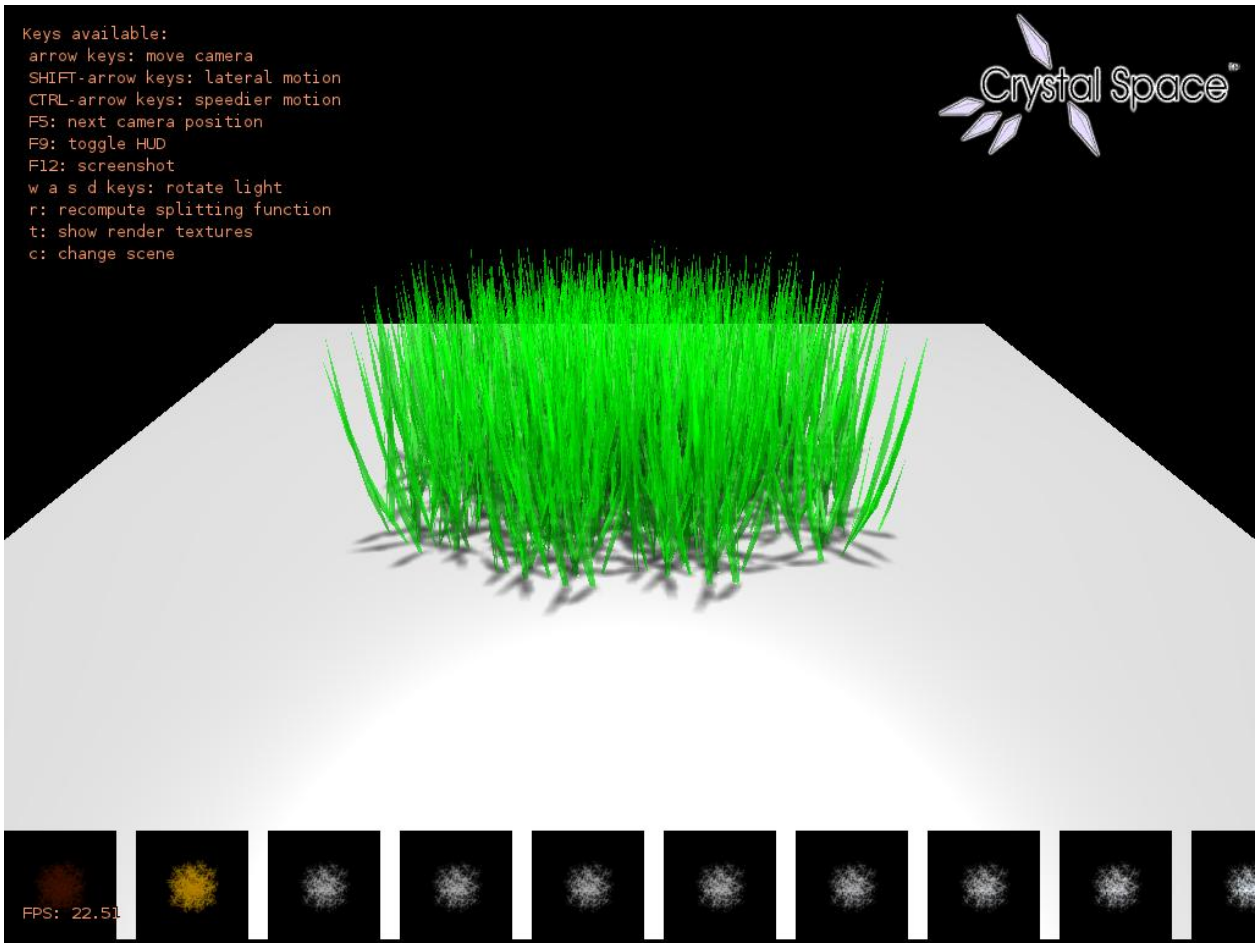

Figure A.11.3 Picture taken from Crystal Space

Layers / Split method	1	16	32
Sum of absolute differences	1.0	0.9	1.0
Cross-correlation	0.7	0.3	0.2

Table A.11.3 Split ratio variation for the scene in *Figure A.11.3*

Split ratio (*Table A.11.4*) computed for a scene composed of a grass model viewed from sideways (*Figure A.11.4*).

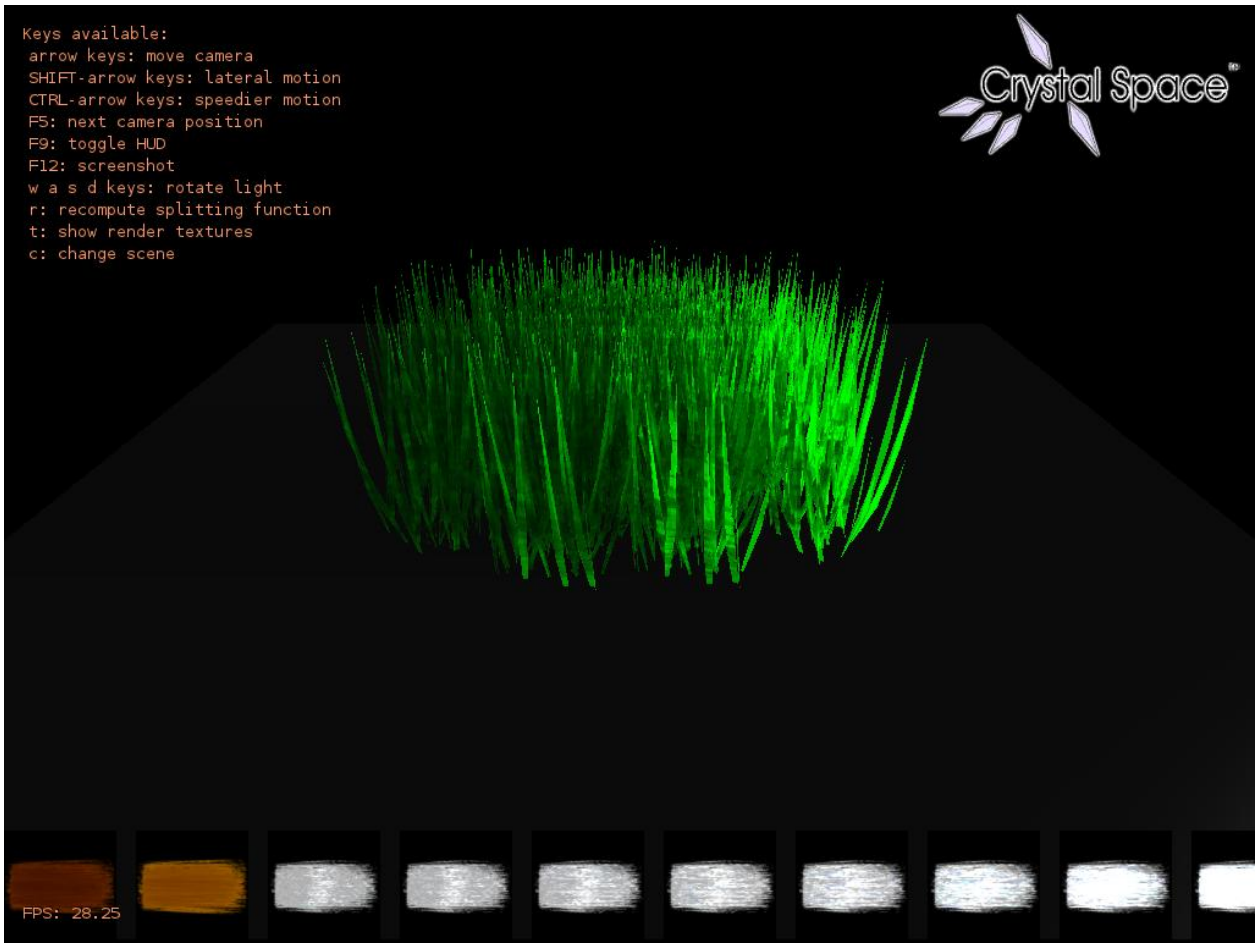

Figure A.11.4 Picture taken from Crystal Space

Split method \ Layers	1	16	32
Sum of absolute differences	1.0	1.0	1.0
Cross-correlation	1.0	0.8	0.7

Table A.11.4 Split ratio variation for the scene in *Figure A.11.4*

Split ratio (*Table A.11.5*) computed for a scene composed of a dense grass model viewed from above (*Figure A.11.5*).

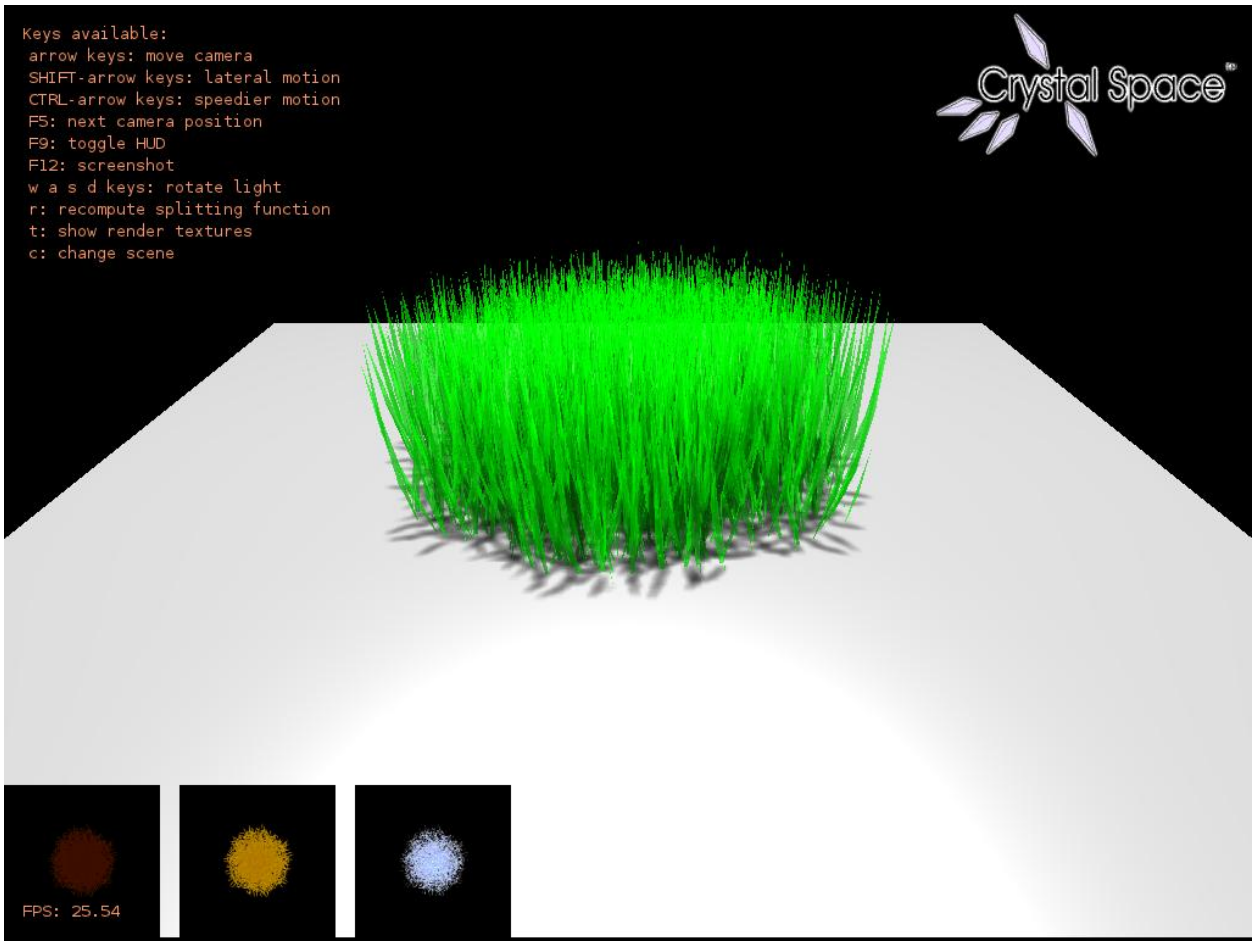

Figure A.11.5 Picture taken from Crystal Space

Layers / Split method	1	16	32
Sum of absolute differences	1.0	1.0	1.0
Cross-correlation	0.8	0.4	0.3

Table A.11.5 Split ratio variation for the scene in *Figure A.11.5*

Split ratio (*Table A.11.6*) computed for a scene composed of a dense grass model viewed from sideways (*Figure A.11.6*).

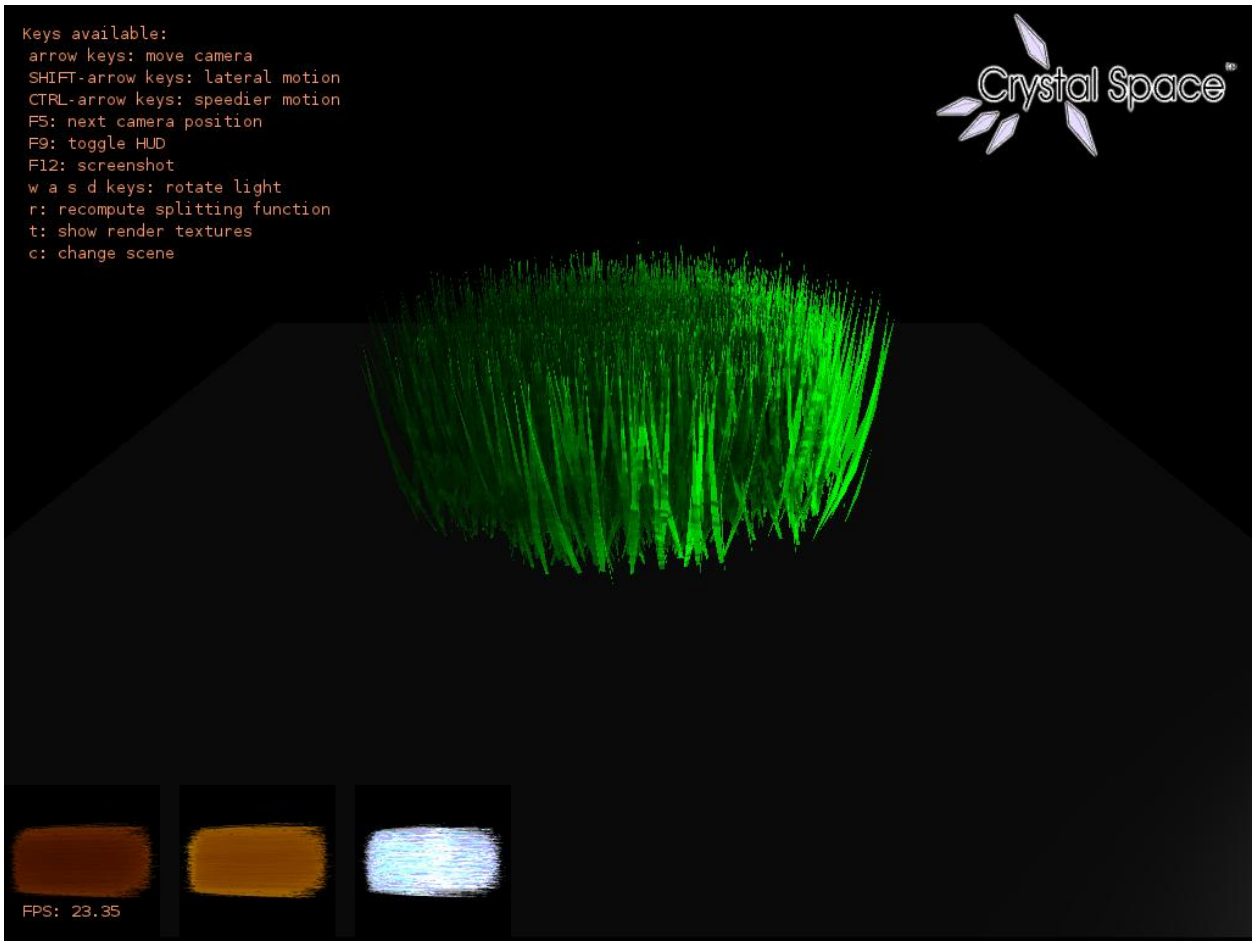

Figure A.11.6 Picture taken from Crystal Space

Layers / Split method	1	16	32
Sum of absolute differences	1.0	1.0	1.0
Cross-correlation	1.0	0.9	0.8

Table A.11.6 Split ratio variation for the scene in *Figure A.11.6*

Section 12

Excerpt from the code computing the cross-correlation coefficient using only one iteration over the input images:

```
// compute correlation coefficient:
double *means = new double[4 * persist.mrt];
double *squareSum = new double[4 * persist.mrt];
double *xy = new double[4 * persist.mrt];
double *correlations = new double[4 * persist.mrt];

for (int i = 0 ; i < 4 * persist.mrt ; i ++)
{
  means[i] = 0;
  squareSum[i] = 0;
  xy[i] = 0;
}

for (int layer = 0 ; layer < persist.mrt ; layer ++)
  for (int i = 0 ; i < persist.shadowMapRes ; i ++)
    for (int j = 0 ; j < persist.shadowMapRes ; j ++)
      for (int k = 0 ; k < 4 ; k ++)
      {
        uint8 x = data[layer][4 * (i + j * persist.shadowMapRes)+ k];

        means[4 * layer + k] += x;
        squareSum[4 * layer + k] += (x * x);

        if (4 * layer + k < 4 * persist.mrt - 1)
        {
          uint8 y = data[layer + (k + 1) / 4]
            [4 * (i + j * persist.shadowMapRes) + (k + 1) % 4];
          xy[4 * layer + k] += (x * y);
        }
      }

int n = persist.shadowMapRes * persist.shadowMapRes;
for (int i = 0 ; i < 4 * persist.mrt ; i ++)
  means[i] /= n;

for (int i = 0 ; i < 4 * persist.mrt - 1 ; i ++)
{
  correlations[i] = (xy[i] - means[i] * means[i + 1] * n) /
    sqrt( (squareSum[i] - means[i] * means[i] * n) *
    (squareSum[i + 1] - means[i + 1] * means[i + 1] * n));
}
```