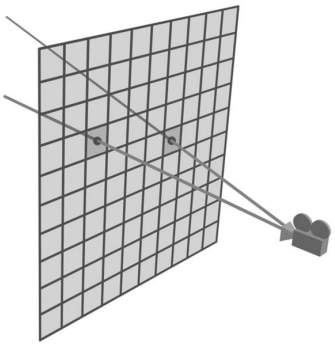
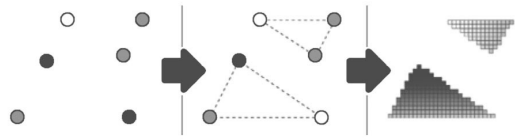
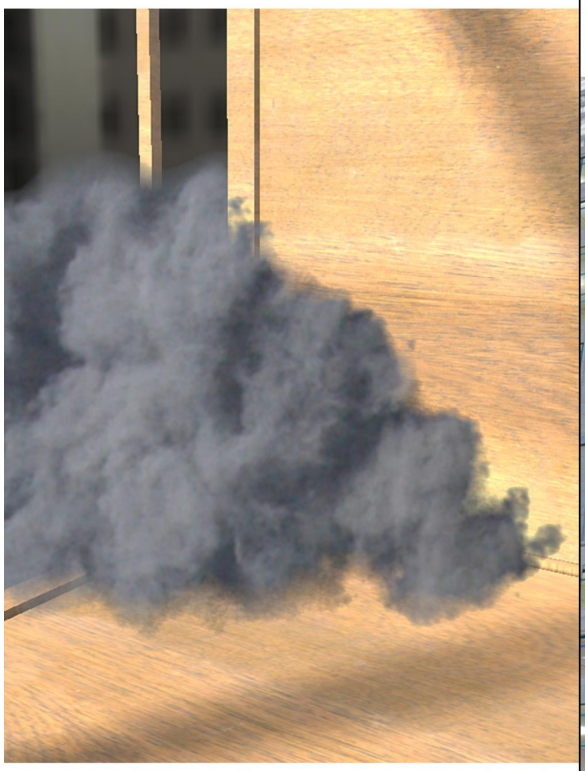


Using raycasting in GPU shaders to enhance real-time graphics

Joost van Dongen

Master's Thesis number INF/SCR-08-67
Student number: 0215821
Supervisor: Prof. Dr. Mark H. Overmars
Game & Media Technology
Utrecht University
4 June 2009





Universiteit Utrecht

Table of contents

Foreword

I would like to thank the Roniboys for turning this thesis into an epic struggle. Well, epic on the puny scale of my life, of course, but epic nevertheless. Even though I was full of ideas throughout the course of creating this thesis, even though I got at least something small done every time I sat down to work on it, even though the topic was entirely my taste in icecream (either banana or vanilla): this thesis was a struggle to work on.

The problems began when I graduated from the Utrecht School of the Arts. Instead of using the next year to finish this Master's thesis, together with six other graduates we founded our own little company (Ronimo Games) and started working full-time on our own games. After a full week of working at Ronimo Games, it is difficult to just sit down and spend the entire weekend working on my thesis. And indeed I rarely did. My thesis was finished at a snail's pace at best, causing me to take two years to finish it. All thanks to the Roniboys!

I would, however, also like to thank these Roniboys (Fabian, Gijs, Jasper, Martijn, Olivier and Ralph) for the great time we had and still have at Ronimo Games. We work on awesome games and bad jokes and I am really happy to be in it for the ride. Around the same time this thesis was finally finished, our first game was also released: Swords & Soldiers for Nintendo WiiWare. Right now I do not yet know whether it is going to be a success, but I consider it an awesome game that I can be proud of anyway! Ralph deserves an extra thank you: his remarks sparkled the idea that would eventually lead to the Interior Mapping technique that is central to this thesis.

Another thanks goes to my thesis supervisor, Prof. Dr. Mark H. Overmars. With sharp questions and critique, he helped me focus my mind and ideas, discover new paths and write this thesis. I would also like to thank him for founding the Game & Media Technology Master, which was great fun to study and taught me a lot.

Interior Mapping, the new technique that I came up with for this thesis, has been published at the Computer Graphics International conference in Istanbul in 2008. Getting parts of my Master´s thesis published is something I am really proud of, but this would not have been possible without the demolishing critiques I received from Eurographics before that. I had sent in an earlier version of the article to the Eurographics conference and the anonymous judges correctly saw how my article was seriously lacking in its scientific groundwork. However, they did like the idea of Interior Mapping so much that they gave me an overwhelming amount of references and ideas to improve the quality of the paper. This I did, and the new and revisited version was accepted for the Computer Graphics International conference. This would not have been possible without the very constructive comments of these judges, so anonymous as they may be, I would like to thank them for that!

The demo applications that come with this thesis were created using the open source 3D engine Ogre 3D and I would like to thank Ogre's development team for the flexible and easy to use engine that they have created and are still creating. Also note that Ogre 3D is not just an engine: it is a community as well. Therefore I would like to thank the visitors of the Ogre forums for their input and ideas on the topics of my thesis, and for running my applications to provide me with benchmarking results.

Finally I would like to thank the algorithm-monster called Thomas van Dijk and my brother Erik van Dongen for proof reading my thesis and providing me with valuable input on both content and grammar. Of course I also want to thank the rest of my family for general support.

To all of you: thanks!

1. Introduction

The success of specialised graphics rendering hardware is both a blessing and a curse. Computer chips that have to do only very specific things can be much faster than general processing units that have to be fit for every possible program that might be executed on them. GPUs are highly optimised to render triangles to the screen and can thus do so at speeds that are impossible to achieve on a CPU. This is a blessing for applications that use real-time 3D graphics, such as games: more complex and realistic scenes can be rendered at real-time framerates.

However, GPUs can only render graphics in one specific way, while several other ways exist as well; raytracing and voxel based rendering are more awkward to achieve on modern graphics rendering hardware and performance is orders of magnitude lower than when rendering through the rasterisation of textured and lit triangles. In a sense, this is a curse: in many specific situations, rasterised triangles may not be the best approach at all, but because the GPU can render them so quickly, they are used nevertheless.

An example of this problem is the computer game Outcast, which was released in 1999. For the time, it featured beautifully rendered landscapes. These were rendered through the use of voxels [58], a technique that was not supported by the early day GPUs that were in use then. Therefore the CPU handled the voxel rendering, but this had to be done at a low resolution, because otherwise, the CPU was not fast enough to achieve high framerates. Gamers complained about the low resolution and voxel rendering was hardly used afterwards, even though at the time for such landscapes it was a more fitting technique than rendering rasterised triangles. Additionally, because Outcast was rendered on the CPU, different techniques could be mixed within the same image: while the landscape was rendered through voxels, characters and objects were rendered through polygons, as can be seen in figure 1.1. Such flexibility is impossible to achieve on the GPU at real-time framerates and with high geometrical complexity for the landscape. Because rendering on the GPU is so much faster than on the CPU, and because of these complaints by gamers, most virtual worlds and games have since been rendered solely on the GPU, using only a single rendering technique.

Since then, GPUs have become faster and more complex, and have added a little bit of the CPU's flexibility to their rendering process: through the use of programmable vertex and pixel shaders, increasingly more changes to the standard rendering method have become possible, although always embedded in the same rasterisation technique. It is even possible to work around the standard pipeline and run any other application on the GPU, including raytraced and voxel based rendering. However, as soon as this is done, the complexity of the kind of scene that can be rendered at real-time framerates strongly diminishes.

This thesis focuses on a blend of these two: the scene is rendered through rasterisation, which is exactly what the GPU does best, so that complex scenes can be rendered at high framerates. Certain effects that are not possible to render through standard rasterisation are added by using raytracing

in programmable shaders. This way a number of effects can be achieved, including reflections, interiors of buildings in large cities, highly detailed surfaces, and volumetric smoke. The cases discussed in this thesis combine the advantages of rasterisation and raytracing, and effectively use the power of the GPU to render complex scenes. Only a limited set of problems is fit for this approach. However, some of these effects are very useful in practice.

Figure 1.1. A screenshot from Outcast (1999), developed by Appeal. The character and buildings are rendered using polygon rasterisation, while the landscape is rendered through the use of voxels.

Interior Mapping is a technique that is introduced in this thesis and that utilises this combination of rasterisation and raycasting. It is a new real-time rendering technique that can be used to render the interiors of buildings when looking at them from the outside. Through the windows of a building, perspectively correct rooms can be seen, with textures to add extra detail. However, these rooms are not actually modelled or stored, so the geometry of the buildings can remain simple, as can be seen in figure 1.2. A pixel shader that is applied to the polygons of the exterior of the building performs raycasting to render the interiors. The number of rooms rendered does not influence the framerate or performance at all and very few extra assets are needed for the interiors. With Interior Mapping, large cities can be rendered with transparent windows through which the interiors can be seen at only a small performance cost, something that would require too much rendering time otherwise. Interior Mapping was first presented in Istanbul at the Computer Graphics International 2008 Conference, and published in the accompanying proceedings [15].

The other major contribution that can be found in this thesis is an in-depth comparison of various ways to render volumetric effects, such as smoke and clouds, at high framerates. GPUs can only efficiently render polygons, which are flat, infinitely thin surfaces. Therefore, volumetric effects are problematic to render on the GPU. However, under the name of *soft particles*, a way to render relatively simple volumes has recently come in use in real-time graphics applications. An example of a possible result of this technique can be seen in figure 1.3. This technique works by first rendering

the scene to a depth texture that stores the distance of each pixel to the camera, and then raytracing the volumes and combining them with the depth texture. Although this technique already existed, the kind of comparison that will be presented here was yet to be conducted. Also, to compare this technique with traditional polygon rendering techniques, a technique using slices of the volume has been implemented and refined to show that the raytracing approach is indeed a good one.

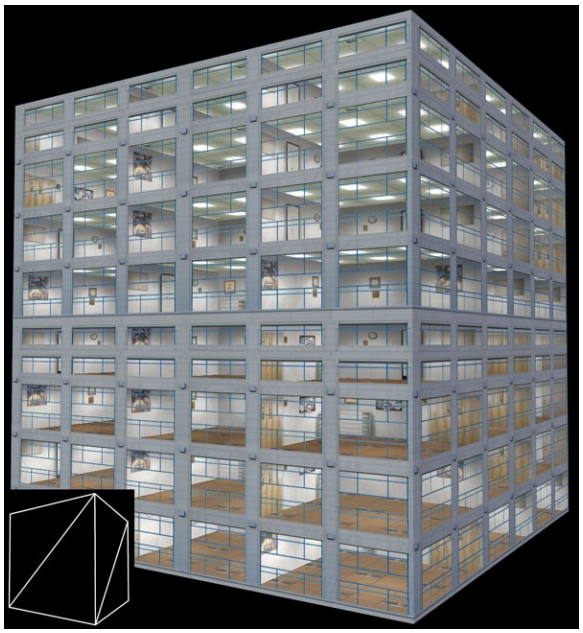


Figure 1.2. Here an example of the effect of Interior Mapping is shown. Through the windows of this building, perspectively correct rooms can be seen, even though these rooms do not exist in geometry, as can be seen in the wireframe inset of the same building.

Figure 1.3. This image shows an example of rendering volumetric smoke. Note how the wood smoothly disappears into the smoke, as the smoke is rendered as a real volume.

This thesis is organised as follows. Chapters 2 and 3 discuss the two techniques that are to be combined in this thesis: chapter 2 analyses the pipeline of GPU rendering and explains the possibilities and limitations of programmable shaders, while chapter 3 delves into raycasting techniques. Chapter 4 then combines these two approaches and discusses what is required of a raycasting technique to be fit for usage on the GPU. After that it proceeds to explain a number of existing techniques that already use this concept. Subsequently, chapter 5 introduces Interior Mapping and chapter 6 discusses volume rendering. Finally, chapter 7 presents the conclusion of this thesis.

2. Rasterisation and programmable GPU shaders

Why is it interesting to study the use of raytracing to enhance GPU graphics at all? To understand this, it is necessary to first have a good understanding of how GPUs work. It is interesting to see what is possible, but probably even more important to see what is *not* possible: GPUs work in a very specific way and many raycasting techniques are not easily combined with that, especially if real-time framerates are required.

GPUs are able to achieve high framerates because they are designed to render only in one specific way, with some variations. Although these variations have grown into a large number of options (from shape, colour and texture, to lighting and programmable shading), each step in the process has strict boundaries and often cannot be skipped, allowing optimisations to the rendering process that would not be possible if more freedom was given to each of the steps.

This chapter will review the rendering pipeline on modern GPUs. Special emphasis is placed on programmable shaders, because that is where raycasting techniques can be implemented. A wide range of different GPUs is available on the market and each one of those has slightly different specifications, so to be able to actually use any of the methods in this thesis in practice, it would also be necessary to explain the differences in specifications. Luckily, standards like *DirectX* and *OpenGL* have grouped the different feature sets together, so there is no need to analyse long lists of different hardware specifications. Also, graphics hardware is generally backwards compatible, so it is often not necessary to make different versions of a technique for each different hardware configuration. The first part of this chapter analyses the pipeline itself. Then some common example uses of shaders are discussed, followed by the different shader versions that one might target. This is concluded by a short discussion on the future of GPUs.

2.1 The fixed function graphics pipeline

The graphics pipeline describes the order in which the GPU processes the triangles that are being rendered, from vertex coordinates to pixels on the screen. For ease of explanation, the graphics pipeline is first discussed as it is when no programmable shaders are used. Vertex and pixel shaders are then added a little further in the text to show what parts of the pipeline they can replace. The graphics pipeline without shaders is called the *fixed function* pipeline. This text is mainly based on [21] and [53].

Figure 2.1 shows an overview of how the GPU renders a triangle. The first step is to define what is to be rendered. This is usually done through the use of either DirectX or OpenGL. The object itself is defined in object space (sometimes called model space). The *transformation matrix* defines how

the object is placed in the 3D world. This way, if several instances of the same object are to be rendered, the same vertices can be used each time and only the transformation matrix needs to be changed to render the object at a different position and orientation. To be able to render the object to the screen, it is also necessary to define the camera. Along with this geometric data, options concerning how to render the object are also set in this phase: what colour should the object have, what textures should be applied to it, how is the object lit, is the object partly transparent, etcetera.

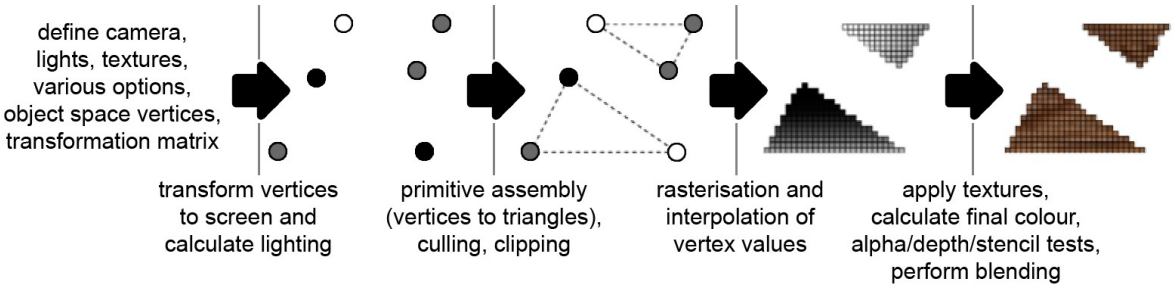


Figure 2.1. An overview of the rendering pipeline.

At this point the GPU has all the information it needs and can start rendering. The CPU does not have to wait for this and can proceed with other tasks (such as defining the next object to be rendered) while the GPU performs its work.

The next step is to transform the vertices from object space to screen space. First the object's transformation matrix is applied to the vertices to position them in the world, and then the camera and projection matrices are used to get the positions of the vertices on the screen. Lighting is calculated per vertex at the beginning of this phase.

Subsequently, the vertices are combined to form triangles. This is called *primitive assembly* and although this usually means forming triangles, it might instead form points, lines or polygons, depending on what the user has instructed the GPU to do. Here, *culling* and *clipping* also occurs. In this phase, culling involves throwing away triangles that have their back towards the camera (*backface culling*). Clipping is used to remove the parts of the polygons that are not actually on the screen. To this end the *camera frustum* is used: the area in the 3D world that is potentially visible to the camera. This is visualised in figure 2.2. The camera frustum is not just limited by the sides of the screen, but also by the near clipping plane and the far clipping plane, which limit the distances at which objects are visible: triangles which are very close to or far away from the camera are ignored.

Now that the triangles are on the screen, they have to be transformed into actual pixels. This step is called rasterisation and it is here that GPU rendering truly shines: in a simple double loop, the triangle can efficiently be turned into pixels. This step is the reason why GPUs can render complex scenes as fast as they do, especially if the number of polygons is low relative to the number of pixels on the screen: the double for-loop quickly finds all the pixels that are covered by a polygon and these pixels can be rendered at a high speed by processing them in parallel. In the same step, values from the vertices are interpolated to get values per pixel. These values can represent anything, but

are usually texture coordinates or the lighting that was calculated at the vertices. Here some extra calculations are needed: because of perspective the values cannot just be linearly interpolated in screen space.

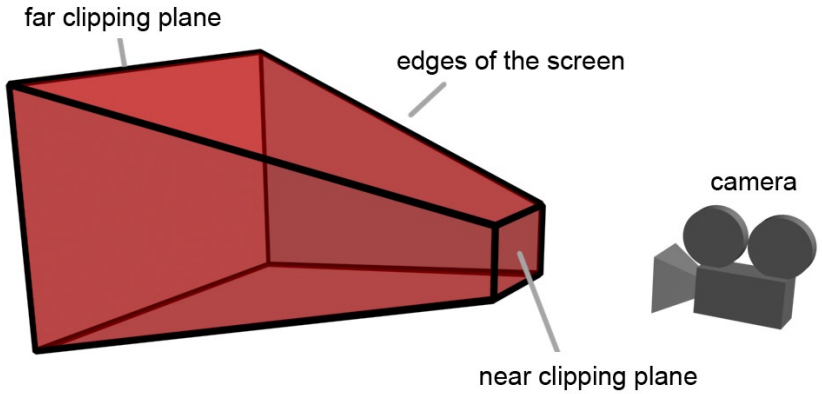


Figure 2.2. This image shows the camera frustum, as it is bounded by the four edges of the screen and the near and far clipping planes.

Next the actual colour of each pixel as it is to appear on the screen needs to be calculated. This is done by combining the colours from textures and interpolated values. The textures are read using the interpolated texture coordinates. The exact way in which this combination occurs, depends on how this was set before rendering began. What also needs doing here, is the decision whether the pixel is actually to be rendered. Stencil, depth and alpha tests are performed and if the pixel fails any of these tests, it is not rendered. Finally, the blending function is used to calculate the colour on the screen as a combination of the colour that was previously on the screen and the newly calculated pixel colour. In most cases the object is fully opaque, so the previously rendered colour is just overwritten by the newly calculated pixel colour. For transparent objects, some form of blending is needed here.

The aforementioned depth test requires some extra explanation. The GPU has no concept of whether an object is in front of another object, but when rendering the scene, it will have to somehow show the object closest to the camera. Instead of sorting objects, the GPU has a *depth buffer* (often called the *Z-buffer*). For each pixel on the screen, the depth buffer stores the distance of that pixel to the camera. If no object was rendered to that pixel, then the value in the depth buffer is infinity. Now if a new pixel is rendered, its distance to the camera is calculated. If this distance is closer than the distance that was already in the depth buffer, then the pixel is in front of what was already drawn there and thus is drawn to the screen. The value in the depth buffer is subsequently overwritten with the distance of the newly rendered pixel. On the other hand, if the distance of the new pixel is larger than what is in the depth buffer, then apparently the new pixel is behind what was already rendered there and thus the pixel is discarded. The exact functioning of the depth test can be manipulated by the user to achieve other effects.

Determining visibility through the use of the depth buffer does not work well with transparency, as a pixel can only have a single value in the depth buffer and therefore cannot handle pixels if their colour is a blend of the colours of several transparent objects. Because of this, applications usually handle transparent objects as follows. First all the fully opaque objects in the scene are rendered, using the depth buffer to determine the visibility of the objects. Then all the transparent objects are sorted from back to front and rendered, so that the objects closest to the camera are rendered last. This way, it can never happen that an object needs to be rendered behind a transparent object, as the transparent object would be rendered afterwards. However, this introduces the limitation that intersecting transparent polygons cannot be rendered correctly. Also, sorting the objects might place a heavy burden on the framerate.

An extra complication in the rendering pipeline is *anti-aliasing*. Normally when triangles are rendered, each pixel will either get the triangle's colour, or not. If the edge of the triangle is diagonal on the screen, then this results in the aliasing-effect called *staircasing*: when one zooms in at the edge of the triangle, a staircase becomes visible, even though the edge of the triangle should be a line. An example of this can be seen to the left in figure 2.3. The cause of this problem is that pixels at the edge of the triangle are only partly covered by the triangle and should therefore be coloured with a mix of the colour from the triangle and the colour from the background. Creating such a mix is called anti-aliasing. The effect of anti-aliasing looks very much like a blur, as can be seen to the right in figure 2.3, even though anti-aliasing is usually achieved through super-sampling. The general method to achieve anti-aliasing, is by rendering several smaller *sub-pixels* and averaging their colours to get the colour of the full pixel. To do this correctly, the depth buffer should also be aware of these sub-pixels. Here the GPU has a great advantage: anti-aliasing is only needed near the edges of a triangle and during rasterisation the GPU knows whether a pixel is close to the triangle's edge, so it can relatively easily only apply the costly anti-aliasing process to the pixels that actually need it, thus achieving high framerates even when anti-aliasing is turned on. However, it should be noted that even though anti-aliasing is implemented efficiently in modern GPUs, it will still decrease the framerate to the degree that it cannot always be turned on.

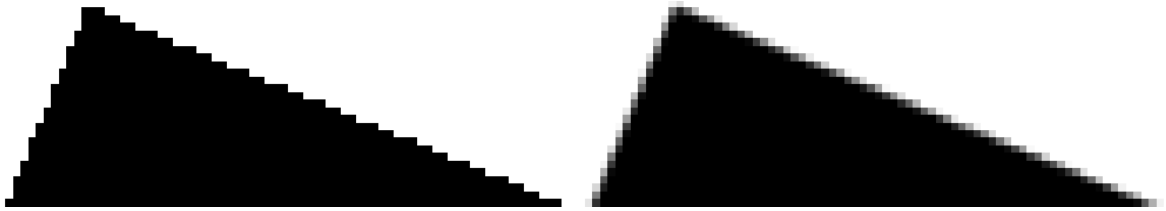


Figure 2.3. The left image shows the staircasing effect when a triangle is rasterised. To the right the same triangle is shown, but with anti aliasing applied, thus strongly reducing the staircasing effect.

In the pipeline above, each triangle is consecutively rendered to the screen. This suggests that the screen starts empty and is filled as the user is watching. As this gradual build-up of the image would look odd, the triangles are instead rendered to the back buffer. When the entire image is done, the back buffer is switched with the screen (the front buffer) and the user sees the new image. Thus,

instead of having a screen that builds up incrementally, the user only sees the resulting complete image.

The fixed function pipeline consists of a number of clearly defined steps. Although many settings can be applied to modify the exact workings of these steps, the core concepts of each step always remain the same. This makes rendering complex scenes at high framerates possible, but also strongly limits the rendering possibilities. As no real computations can be done per pixel, except for the combination of the interpolated colours from the vertices and the colours that are read from the textures, it is for example not possible to cast a ray into the scene from the position of each pixel, or to perform more complex lighting calculations, such as per pixel specular lighting. Another limitation is that the GPU is only capable of rendering points, straight lines and flat polygons. This means that to render, for example, a sphere, the only option is to use many polygons that together form a seemingly smooth curved surface. Also, objects never have any real volume and can only be made to look solid by rendering the complete hull around them, without any transparency.

A more complete and more theoretical introduction to the fixed function pipeline can be found in [54], while an introduction into how to control it through OpenGL can be found in [53].

2.2 Programmable shaders

Vertex and pixel shaders add programmability to certain parts of the pipeline. The steps remain the same, with the exception that now some of them can be programmed to function exactly as the user requires. This is a big step up from the fixed function pipeline, which only allowed a limited amount of settings to be combined. With programmable shaders, much more is possible. This removes many of the limitations of the fixed function pipeline, but as the main steps are still the same, many other limitations remain firmly in place.

2.2.1 How vertex and pixel shaders work

Figure 2.4 shows where shaders operate within the pipeline. The vertex shader replaces the transformation of vertices from object space to screen space, as well as the calculation of lighting. Therefore, the user will usually have to program these things in the vertex shader. This time, however, different behaviour can be created as well. For example, when creating waving grass, the vertex shader can move the top vertices of the grass a little bit every frame and thus create an animation. Without vertex shaders, this effect could only be achieved by actually changing the mesh in memory every frame. This is a time consuming process for the CPU, while the vertex shader can handle this at hardly any additional cost in performance.

The pixel shader calculates the actual colour of the pixel. This includes reading colours from textures and using these to calculate the pixel colour. An example of a common application of pixel shaders is to calculate lighting per pixel. Without shaders, the fixed function pipeline will only

calculate lighting per vertex and interpolate this value over the triangle. This results in imprecise lighting, especially when sharp specular lights are used. Using a pixel shader, this can be fixed. Additionally, as the pixel shader is fully programmable, the user is not limited to standard diffuse and specular lighting, but can use any other lighting formula. Even though the pixel shader outputs a colour, this colour can be interpreted as any other type of value as well. Examples of this are discussed in section 2.3.2.

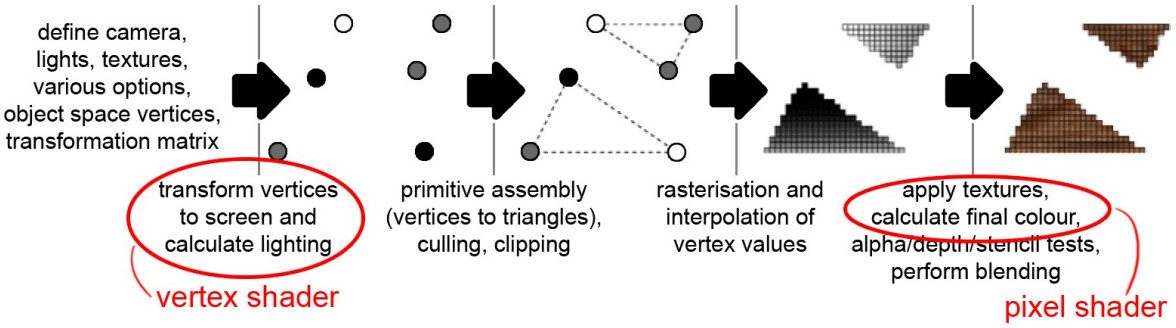


Figure 2.4. The position of the vertex and pixel shader in the pipeline.

To communicate between the vertex shader and the pixel shader, interpolated values are used, as is illustrated in figure 2.5. These values are called *interpolants*. The vertex shader outputs a number of variables, for example texture coordinates or the direction from the vertex towards a light. These values are then interpolated over the triangle and the pixel shader receives the resulting values. This way the texture coordinates vary smoothly over the polygon and can be used to look up the colour in a texture.

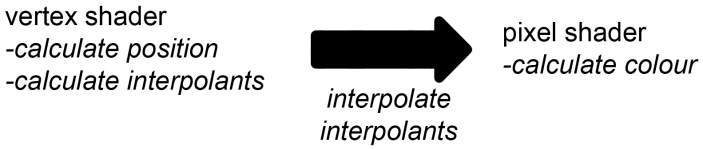


Figure 2.5. The flow from vertex shader to pixel shader.

An important thing to note here is that both the vertex shader and the pixel shader are limited in their scope. The vertex shader only receives information about the vertex itself. It does not receive any data about neighbouring vertices, or even about how many triangles use this vertex. The same kind of limitation applies to the pixel shader: it can only output the colour of its own pixel and cannot look up the colours of neighbouring pixels or the colour that was already in the frame buffer before this pixel was being rendered. So although shaders allow the user to freely program them, there are strong limitations as well. Because of these limitations, the GPU can very efficiently execute the shaders and keep real-time framerates. Nowadays, dozens of shaders can be executed in parallel, and because shaders cannot communicate between each other, there is no need to manage the parallelism. This removes any necessity for vertex shaders to wait for each other to access shared data, as the shaders cannot share any data they output and can only read from textures.

Shaders receive certain types of input. Interpolants and textures have already been discussed; another form of input is *uniform parameters*. These are fixed values that are set before rendering by the user and can then be used in the shaders. The most common uniform parameter is the matrix that transforms vertices from object space to screen space, as this matrix is needed in the vertex shader to perform this transformation. Another example of a uniform parameter is related to the waving grass that was mentioned before: to be able to actually animate the waving, the vertex shader will somehow need to know how large the wave displacement should be at any specific moment in time. The size of the displacement can be sent to the vertex shader by the user through the use of a uniform parameter. Other examples of uniform parameters are the positions and colours of lights and the position of the camera. These are mainly needed for lighting calculations in the vertex and pixel shaders.

The vertex shader also receives data from the vertex itself. Usually this includes the position of the vertex, its normal, and its texture coordinates. Other vertex data that might be used are the surface tangent (required for normal maps), vertex colours, and bone weights (required for skinning). The user can also define any other values here, but the total number of variables per vertex is limited.

The limitations of vertex and pixel shaders lie mainly in their positioning within the rendering pipeline of the GPU. A vertex shader can only output position data and interpolants, while a pixel shader can only output the colour of the pixel. Any other forms of output are impossible; for example, one cannot sort a list and output the entire sorted list in a single shader execution. Direct communication between different vertex shaders or pixel shaders is also impossible. Another limitation is that both shader types receive no information about the objects in their vicinity, so a vertex shader cannot do any computations based on the number of triangles that use it, and a pixel shader cannot use the colour that was previously rendered to the frame buffer. However, because the colours that are calculated by pixel shaders can be stored in textures and because other pixel shaders can read from these textures again, many of the limitations can be worked around. This does often come at a huge decrease in performance, so whether this is actually feasible depends on the complexity of the technique being executed in this way. Depending on the shader version, there are also limitations to the number of instructions that can be used in a single shader program, but this problem only exists in older GPUs.

2.2.2 An example of a vertex shader and a pixel shader

The best way to really understand how shaders work, is to look at a practical example. The example shown here calculates per pixel diffuse lighting, something that cannot be done on the GPU without programmable shaders. This is an extremely simple example of how shaders work: most shaders perform more calculations to achieve more complex effects. The code of the vertex shader is shown in figure 2.6 and the code of the pixel shader is shown in figure 2.7.

Shaders are written in a specific language and this example is given in Cg, a language based on C, but designed specifically to write shaders. A complete reference to the workings of Cg can be found in [29], while a step by step explanation of how to create shaders in Cg can be found in [21].

```
1.  void vertexShader(float4 position           : POSITION,
2.                    float3 normal             : NORMAL,
3.                    float2 textureCoordinates : TEXCOORD0,
4.
5.                    out float4 outPosition           : POSITION,
6.                    out float2 outTextureCoordinates : TEXCOORD0,
7.                    out float3 outLightDirection     : TEXCOORD1,
8.                    out float3 outNormal             : TEXCOORD2,
9.
10.                   uniform float3   lightPosition,
11.                   uniform float4x4 transformationMatrix
12.                  )
13. {
14.           outPosition = mul(transformationMatrix, position);
15.           outTextureCoordinates = textureCoordinates;
16.           outLightDirection = normalize(lightPosition - position.xyz);
17.           outNormal = normal;
18. }
```

Figure 2.6. Example code of a vertex shader, written in Cg.

```
1.  void pixelShader(float2 textureCoordinates : TEXCOORD0,
2.                   float3 lightDirection     : TEXCOORD1,
3.                   float3 normal             : TEXCOORD2,
4.
5.                   out float4 outColour : COLOR,
6.
7.                   uniform sampler2D texture
8.                  )
9.  {
10.           float4 textureColour = tex2D(texture, textureCoordinates);
11.
12.           normal = normalize(normal);
13.           lightDirection = normalize(lightDirection);
14.           float diffuseLighting = dot(normal, lightDirection);
15.
16.           outColour = textureColour * diffuseLighting;
17. }
```

Figure 2.7. Example code of a pixel shader that calculate per pixel diffuse lighting, written in Cg.

Although a detailed explanation of the syntax of Cg is not in order here, a brief explanation of how this code works, might make the workings of shaders in general clearer. The first thing to note here,

is that both the pixel shader and the vertex shader start with their input, their output and their uniform parameters. In the vertex shader, these are defined in lines 1 to 11, and in the pixel shader in lines 1 to 7.

Both in the pixel shader and the vertex shader, lines 1, 2 and 3 define the input of the shader. In the vertex shader the input consists of the position, normal and texture coordinates of the vertex. In the pixel shader, the input consists of interpolants that were generated by the vertex shader. The vertex shader defines that it outputs the texture coordinates of the vertex, the direction towards the light and the surface normal in lines 6, 7 and 8. From this, the pixel shader receives interpolated values, i.e. a combination of the results that were outputted by the vertex shaders of each of the three vertices that together form a triangle. For this reason, the pixel shader needs to normalise the normal in line 12, even though what the vertex shader outputted was already normalised. This is necessary, because an interpolation of three normalised vectors is itself not necessarily also a normalised vector.

While lines 6, 7 and 8 of the vertex shader specify output that goes to the pixel shader, line 5 specifies the required output towards the GPU: the position of the vertex on the screen. To get this position, the vertex shader multiplies the transformation matrix of the object with the position of the vertex in line 14. This transformation matrix is a uniform parameter, i.e. a variable that is the same for the entire object being rendered, and is specified in line 11 of the vertex shader.

The output of the pixel shader is the colour of the actual pixel. It is defined in line 5 and calculated in line 16 as the multiplication of the diffuse lighting and the colour of the texture. The diffuse lighting is calculated through a dot product of the normal and the direction towards the light in line 14 of the pixel shader. The colour of the texture is read in line 10 and the texture itself is again a uniform parameter. So for the entire object, the same texture is used, which is defined in line 7 of the pixel shader.

A major issue that has not been discussed so far, is how the input and output variables are connected to values outside the shader itself. In the case of the vertex shader, the keywords POSITION, NORMAL and TEXCOORD0 define what the GPU should put there. For the interpolants, the keywords TEXCOORD0, TEXCOORD1 and TEXCOORD2 link the output of the vertex shader to the input of the pixel shader. This also explains why the names of the variables can be different: only the keywords behind the semicolons are important there. Finally, the uniform parameters are linked to variables in code by the order in which they are defined in the shader and the index they receive when they are set in code.

One thing that should have become clear after looking at this example, is that the vertex shader and the pixel shader are tightly coupled. The vertex shader must prepare the interpolants that the pixel shader uses. This means that vertex shaders and pixel shaders often come in pairs that can not necessarily be used in other combinations.

2.3 Common applications of shaders

The most interesting aspect of shaders is that, since they are programmable, many different things can be done with them. For this reason, it only makes sense to give an overview here of some of the most used applications of shaders. This overview does not contain any applications that specifically make use of raycasting in shaders: examples of such techniques are discussed in chapter 4. Techniques discussed here are either commonly used, or are considered to be good examples of the wide range of possibilities of shaders.

2.3.1 Per pixel diffuse and specular lighting with normal maps

The most used shader application probably is to render objects with diffuse and specular lighting, with added details through the use of normal maps. Without shaders, lighting can only be done per vertex, not per pixel. This is not accurate enough for sharp specular lighting and results in diffuse lighting of less quality than when it is calculated in a pixel shader, as can be seen in figure 2.8.

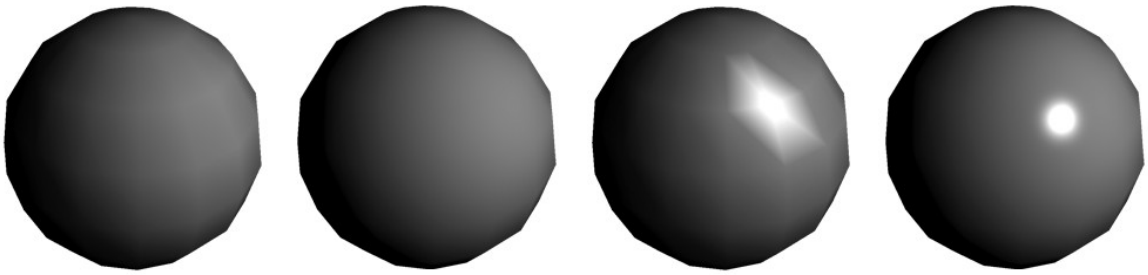


Figure 2.8. This image shows the quality difference between lighting per vertex and per pixel. From left to right: per vertex diffuse lighting, per pixel diffuse lighting, per vertex diffuse and specular lighting and finally per pixel diffuse and specular lighting. The difference between the first two images is quite subtle, although it is possible to see that the lighting in the image on the far left contains subtle banding. However, when sharp specular highlights are calculated per vertex, the lighting becomes heavily distorted, as can be seen in the third image. Finally, the fourth image shows that this sphere can have smooth specular highlights when lit per pixel in a pixel shader.

To calculate lighting in a shader, the positions of lights and of the camera are stored in uniform parameters. The vectors from the position of the pixel in the world towards the camera and towards each of the lights are calculated and then used to calculate diffuse and specular lighting. The colours and other parameters, such as attenuation, are also stored in uniform parameters for each light and taken into account in the pixel shader.

This technique becomes especially interesting when combined with a *normal map*. A normal map is a texture that stores surface normals. These surface normals are then used in the pixel shader to calculate lighting, instead of using the interpolated normals from the vertices. The benefit of this is

that the object can be lit as if its surface is extremely detailed, while the actual number of triangles can remain low. An example of this can be seen in figure 2.9.

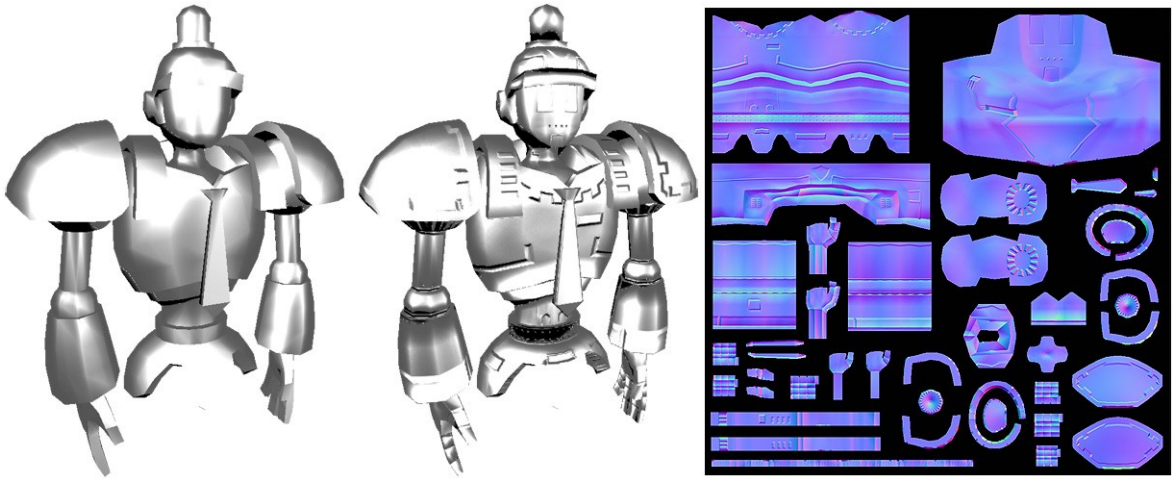


Figure 2.9. To the left an object is shown without a normal map. In the middle, the same object is shown with a normal map. To the right the normal map itself can be seen, in which the colours represent the directions of the surface vectors.

Normal maps are a good example of the use of textures to store data other than the colour of a surface. The RGB components of the texture usually represent colours, but in the case of normal maps, they represent normalised vectors; RGB is interpreted as XYZ coordinates here. The values in a texture can represent many different things and this fact is often used in shaders. Other examples are the specular map, which stores the strength of specular lighting for each part of the surface, and the self-illumination map, which stores how much light the surface emits (and thus how bright it is when no other lights shine on it).

2.3.2 Post effects

Although the most obvious application of vertex and pixel shaders is to just apply them to objects, it is also possible to first render the entire scene to a texture, and then render this texture to the screen while a pixel shader is being applied to it. This results in the special class of techniques called *post effects*.

Many different effects can be achieved using post effects; some of the most common ones include glow and heat distortion. In the case of glow, the brightest parts of the screen are smeared out over surrounding darker parts, resulting in an effect that gives the impression of blinding the player. In the case of heat distortion, the screen is distorted around fire. This is done by using an offset from the original rendering position wherever the heat is. Examples of both effects can be found in figure 2.10.

Whereas glow and heat distortion simply use a render of the scene to calculate their effects on, it is also possible to render other data to a texture and use that for the effects. An example of this is to render the distance of every pixel to the camera. This results in a texture that basically contains the same data as the depth buffer. However, the depth buffer is not readable from a shader and therefore not usable for post effects, unlike this texture, with its extra rendering of the distance to the scene. This depth data can then be used for certain effects, for example depth of field blur (also known as focal blur) [49]. Here, the image is blurred based on the distance to the camera, achieving the same effect as when a real photo camera makes a photo of objects that are out of focus. An example of this can be seen in figure 2.10.

Figure 2.10. Examples of post effects. From left to right: glow in Gears Of War by Epic Games (2007), heat distortion in Monolith Production's F.E.A.R. (2005) and depth of field blur on the weapon in Guerrilla's Killzone 2 (2009).

Rendering scene data to a texture is taken a step further in the case of deferred shading [46]. Here, not just depth, but many other forms of data are rendered to textures. In the final step a pixel shader takes in all these textures and uses them to calculate the final colours for the scene. One of the benefits of doing this, is that complex lighting calculations only need to be done once per pixel, as objects behind other objects are not visible in the textures and are therefore never lit. Figure 2.11 shows an example of rendering various kinds of data to textures. The pixel shader that calculates the final image then has enough data to perform lighting calculations. For example, a pixel shader that calculates only diffuse lighting on textured objects would require the position in the world of the pixel that is being rendered, the normal of the surface, and the diffuse colour of the texture. These are all rendered to textures. The position of the light is passed to the pixel shader as a uniform parameter. This gives the pixel shader enough data to calculate the colours of the final image.

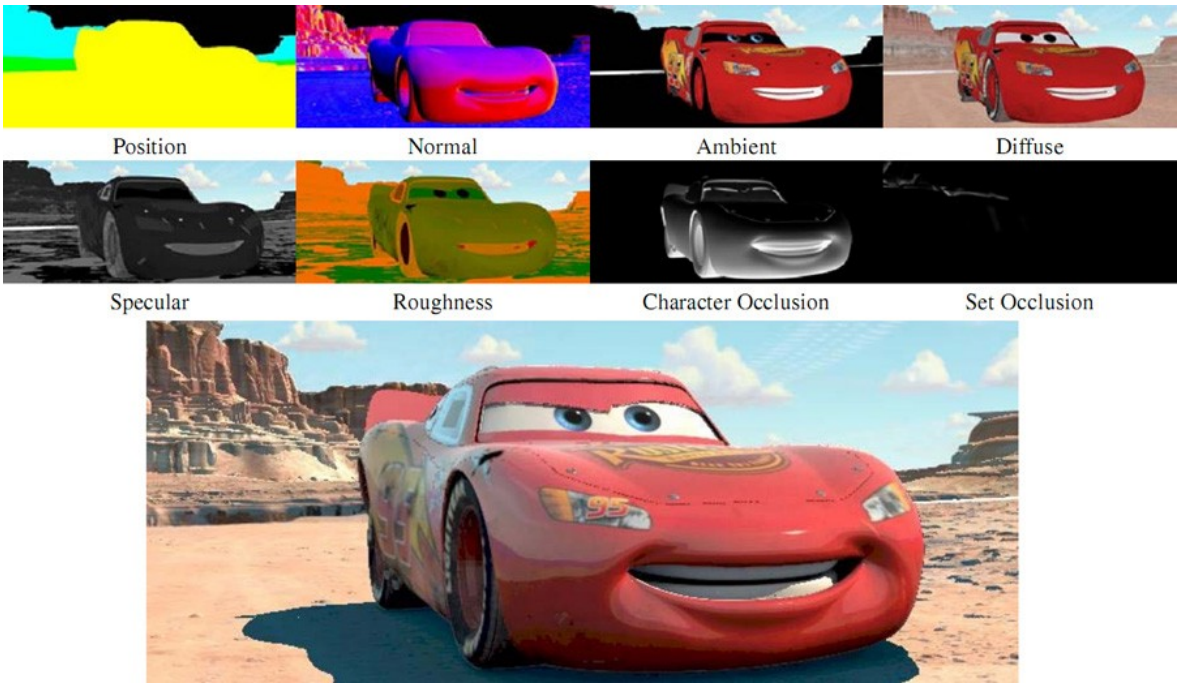


Figure 2.11. The two rows of images at the top show how various kinds of scene data have been rendered to textures for deferred shading. The bottom image shows the result of combining these images. Images from [46].

2.3.3 General programming on the GPU (GPGPU)

Most applications of shaders actually render graphics, but since they are programmable, shaders can also be used to let the GPU solve many other problems as well. Textures can be used to store data of any type and pixel shaders can take in this data, do some calculations and then output the result to other textures. Finally, the resulting textures can be read by the CPU to extract the calculated data. Doing this is called General Processing on the GPU (*GPGPU*). An often cited example use of this is to perform physics simulations on the GPU. In figure 2.12, an example of physics on smoke is shown. In this demo, the physics simulation is performed fully on the GPU.

It seems odd to perform this kind of calculations on the GPU, as the CPU seems more fit for tasks that are not related to rendering. However, modern GPUs have dozens of parallel processors, while CPUs only have a few. The rendering pipeline limits the processors of the GPU in what they can, but if a way can be found to efficiently translate a problem into something that the GPU can handle, then the GPU might be much faster than the CPU. Problems that are fit for this usually contain many floating point calculations and vector math and are highly parallel, so that all the shader processors on the GPU can work on the problem without having to wait for each other.

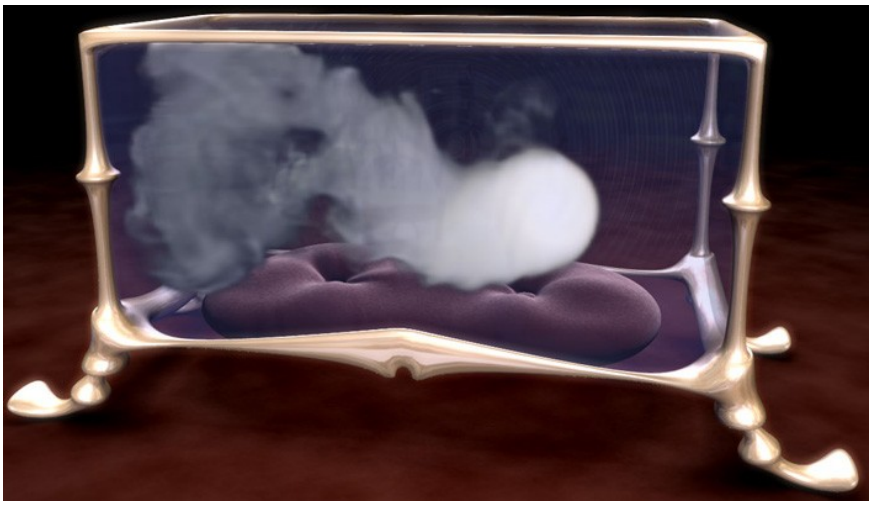


Figure 2.12. This image shows a screenshot from "Box of Smoke" (2006), a technology demonstration by Nvidia.

2.4 Shader model versions

Different GPUs support different feature sets and this is no different for shaders. Various definitions of what shaders can do have been released, each expanding on earlier versions. More features become available every year, along with an increase of processing power on the GPU, allowing more complex effects and models to be rendered in real-time.

The downside of this, is that the programmer needs to keep in mind the different hardware versions that the target audience might be using. A benefit here is that in general, GPUs are backwards compatible: if a shader worked on a GPU with an older specification, then it will also run on a newer GPU. Nevertheless, if the newest shader techniques are required, then either older GPUs cannot be supported, or simplified implementations will have to be made for the older GPUs. Implementing these simplified shaders takes extra time. This problem of course does not occur when working for a fixed platform like the Playstation 3 or Xbox 360. These machines always have the exact same GPU and to make things even easier, the Playstation 3 and the Xbox 360 support almost the exact same features in their GPUs. The main differences between these two are in the speeds of specific operations on the GPU and in the amounts of RAM available for textures: the Playstation 3 has 256mb of video memory and 256mb of general memory, while the Xbox 360 has 512mb of shared memory, which can be used either as video memory or as general memory [52]. In practice, this often means that the Xbox 360 has more memory available for textures, as many games do not require 256mb of general memory.

Regardless of whether a single GPU or several GPUs are targeted, it is always necessary to understand the limitations of the target hardware when writing shaders. For older cards, these limitations are mainly in the number of features and instructions, while some newer cards might support all the basic requirements, but might not always be fast enough to run the most complex of shaders at real-time framerates. Limitations in processing speed are difficult to specify, since

whether they actually apply to a specific shader technique heavily depends on the complexity of the rest of the scene, the resolution of the screen, and the desired framerate. This section therefore only gives an overview of the limitations in features and instructions that can be found on different GPUs, and does not discuss the processing power of the various types of GPUs any further.

The specifications are grouped in so-called *shader models*, each increasing the feature set of the predecessor. Understanding the limitations of the various shader models is not only important when actually writing shaders, but also for the rest of this thesis: when combining raycasting with rasterised graphics, many straightforward techniques turn out to be impossible because of the limitations of the various shader models. Also note here that although shader model 4 has already been released, applications that require it are still rare: at the moment many consumers do not yet possess a GPU that supports shader model 4 and none of the current gaming consoles support it.

2.4.1 Shader model versions 1, 2 and 3

In shader models 1, 2 and 3, the vertex and pixel shaders have the same role, only with more features per version. The number of instructions allowed in a single shader increases with each new version, as does the number of texture look-ups allowed. An overview of these shader model versions is found in figures 2.13 and 2.14.

Not only increases the number of instructions allowed with each shader model, but several other features as well. These are mentioned in the notes in the tables. One of these limitations is that in shader model 1 and 2, it was not possible to read from a texture in the vertex shader. This makes it impossible to implement a vertex shader that processes a height map to displace the surface.

Also, in shader models 1 and 2 it is not possible to use control structures like if/then statements, while-loops and for-loops, as this would make the number of instructions in the shader variable. Loops with a fixed number of iterations are allowed, as these can be expanded to get the same code, but without the loops. However, many of the features that require if-statements can be implemented by executing both the if-part and the else-part, and then choosing the right result using a function like the step function. This returns either 0 or 1, depending on which one of two numbers is the largest. This is not very efficient, though, as it makes that all code is always executed.

In shader model 1, the number of instructions allowed in the pixel shader is so small that hardly any effects seem achievable. However, it is often possible to divide a calculation into several smaller pixel shaders and combine the results in the frame buffer. This way, relatively complex effects can be achieved, such as normal mapping with specular lighting from several light sources.

Note that several sub-versions of shader models also exist, for example shader models 1.2, 1.3, 1.4 and 2.x. These models lie between the major versions that are listed in tables 2.1 and 2.2 and each add some new features. For example, shader model 2.x allows twice as many instructions in the pixel shader as shader model 2.0.

Vertex shader version	Hardware	Limitations	Notes
vs 1.1	Geforce 4 Radeon 8xxx	128 instructions	
vs 2.0	Geforce FX Radeon 9xxx	256 instructions	
vs 3.0	Geforce 6 Radeon X1xxx	∞ instructions	Controlling program flow using if/then, for-loops and while-loops is now possible. Also, from here on textures can be read from the vertex shader.

Pixel shader version	Hardware	Limitations	Notes
ps 1.1	Geforce 4 Radeon 8xxx	8 instructions + 4 texture reads	Hardly any standard functions are available. For example no square root or sine.
ps 2.0	Geforce FX Radeon 9xxx	64 instructions + 32 texture reads	It is now possible to calculate the texture coordinates for a texture-read from the result of another texture read. This is called a *dependent read*. Functions like sine and square root are now available.
ps 3.0	Geforce 6 Radeon X1xxx	∞ instructions	Controlling program flow using if/then, for-loops and while-loops is now possible

Figures 2.13 and 2.14. An overview of vertex and pixel shader versions and the hardware where these shader versions first became available.

The following examples give an idea of the kinds of effects that can be achieved in different shader model versions. In shader model 1.1, per pixel lighting and normal mapping can be applied. Specular mapping is also possible, but if normal mapping is used, the power to which the specular can be taken is limited. If these techniques are applied, then a single shader can only calculate the lighting from one light and several passes are needed when more lights are used. In shader model 2, more lights can be calculated in the same shader. Dependent texture reads now make it possible to let the normal map influence reflections, and to apply parallax mapping. Some form of displacement mapping in the pixel shader is also possible, but to a limited extend. In shader model 3, displacement mapping becomes possible at any desired quality. Displacement mapping can now also be applied to the mesh itself in the vertex shader, as the vertex shader can now read the height of the terrain from a height texture and use this to displace the actual geometry.

2.4.2 DirectX 10 and shader model 4

In shader model 4 (which was introduced together with DirectX 10 in Windows Vista) a completely new type of shader was introduced: the *geometry shader* [36]. This shader is executed right after the vertex shader. It takes as its input a whole rendering primitive, usually a triangle, including the interpolants of each of the three vertices of the triangle, which were calculated by the vertex shader. The geometry shader outputs zero or more primitives (again, usually triangles), which are then each processed in the rest of the pipeline, so that each of these new primitives is rasterised and the generated pixels are fed into the pixel shader. The uses of geometry shaders are slightly more obscure than those of vertex and pixel shaders, but many useful applications of geometry shaders exist nevertheless. The best known example is probably to use the geometry shader to dynamically change the quality of a curve by using a varying number of lines to represent it. Another application is to generate procedural models, as is shown in figure 2.15.

Figure 2.15. Screenshots from Cascades, a DirectX 10 demo by Nvidia. The procedural landscape is generated in a geometry shader, as are the waterfalls.

2.4.3 The future: DirectX 11 and shader model 5

Although DirectX 11 and shader model 5 have not yet been released, their feature sets have been announced [8] and are briefly discussed here. However, the features discussed here are at the time of writing still work in progress, so they may still change before DirectX 11 is actually released.

The most notable new features of DirectX 11 are the introduction of several new stages into the pipeline, including several new shaders. The standard pipeline gets three new steps between the vertex shader and the geometry shader. These three steps are intended to tessellate the mesh on the fly. The idea is that a model with few polygons is sent to the GPU. The vertex shader then processes the vertices and sends these on to the *hull shader,* which decides how many polygons should be generated. The *tessellator* subsequently generates the requested number of polygons. Unlike the hull shader, the tessellator is a fixed function and therefore cannot be programmed, only configured. The next step is the *domain shader*, which takes the vertices generated by the tessellator and calculates their actual positions. The triangles are then sent on to the geometry shader and from

there on the pipeline continues as usual. The addition of the hull shader, tessellator and domain shader allow the GPU to dynamically add extra polygons. At first glance, this seems to be very much like the geometry shader, but here the resulting performance is potentially much higher.

Another main feature of DirectX 11 is the addition of the *compute shader*. This shader exists outside the standard pipeline and is intended to be used for the aforementioned GPGPU: the processing power of the GPU is used for things other than standard rasterised rendering. Without compute shaders, if for example a physics simulation is performed on the GPU, this must still be done in vertex and pixel shaders, although vertices and pixels really have no meaning in this context. The compute shader allows such programs to run on the GPU, but outside the standard pipeline.

2.4.4 The far future: will GPUs cease to exist?

In the discussion above a common trend is visible: with every new shader model, the capacities of the GPU become more general and less specific for one particular way of rendering, especially with the introduction of the compute shader in DirectX 11. So the GPU is becoming more and more like a CPU, only with dozens of parallel processes (shaders). At the same time, CPUs are becoming increasingly more parallel. The Xbox 360 has three processing cores, many new consumer computers have a single CPU with four cores inside and the Playstation 3 even has a main core with seven smaller cores (SPUs) attached to it. The future increase in CPU performance mainly seems to lie in continuing to add more cores.

If there will be no changes to this process, the CPU and GPU would become increasingly similar in the future, causing the GPU to become ultimately redundant and disappear. In fact, for the Playstation 3 Sony already experimented with this: originally the Playstation 3 was planned to have twice as many CPU cores and no GPU at all. Their cell processors turned out to not be as fast at graphics as a standard GPU, so they ended up with a traditional GPU anyway, but ten years from now this may be a valid approach nevertheless. Gabe Newell, co-founder and management director of game developer Valve (mostly known for the Half-Life series of games) explained in an interview how this change might occur:

"Right now, we have the CPU, API, and GPU, and then people are trying to argue that there should be a PPU [physics processing unit], an A.I. accelerator, and these other kinds of things. We think it's going to go the other way, whether it's somebody who figures out how to generalize GPU cores or somebody who makes multicore CPUs that also handle rendering. The heterogeneous environment's going to go away. Scalability will stop being a hardware problem and will become a software problem -- how well can we take advantage of all these cores? It's good news, as it means the same performance gains that we've seen in graphics since 3DFX will now apply to every aspect of game engines as well." [19]

Of course, this kind of long term prediction is always a wild guess, but in the context of the current study this would be an important development, as this thesis discusses how to combine rasterised

GPU graphics with raycasting, within the limits of the GPU. If in the future the GPU disappears, then these limits automatically also disappear and any combination between different rendering techniques can be made, depending on what works for the specific kind of scene being rendered.

3. Raycasting

Just like the GPU based rendering in the previous chapter, rendering through raycasting is a way to turn a 3D scene into a 2D image. However, in many ways raycasting can be considered to be the opposite of rasterisation. Raycasting is often used to deliver high quality images with detailed and realistic lighting, but this goes at the cost of very long rendering times: waiting several hours for an image to be finished is not rare. Rasterisation on the other hand usually renders highly simplified scenes with simplified or faked lighting, but does so at real-time framerates. Raycasting performs relatively well when there are millions of polygons, while rasterisation performs well when there are few polygons and each polygon occupies lots of pixels on the screen.

These opposites come from the approach that is taken towards rendering. In rasterisation, each polygon is rendered to the screen: the flow is from polygon to pixel. In raycasting the flow is exactly the opposite: from the pixel to the polygon. This results in different benefits and different possibilities, as will be explained in the rest of this chapter. After discussing some of the applications of raycasted rendering, the technique itself and some of the things that can be done with it are discussed. Because raycasting is a very slow process, some optimisations are discussed as well, including using the GPU's processing power for pure raycasting. This chapter is mainly based on the books [54] and [18].

Raycasting is used for almost all applications that use rendering that does not have to be real-time. It can produce images of higher quality than rasterisation, so if time is not an issue, then raycasting is used. One of the most well known applications of raycasting based rendering are feature length 3D animation movies like Shrek, Toy Story and Finding Nemo. These are large projects where around ninety minutes of high resolution images is needed. Very similar to films like these are the special effects in films with real characters: backdrops might be changed, and some characters and stunts might be made with 3D animation instead of with real people. Everything that is either impossible or too expensive to do convincingly with real people and film sets can be done through 3D graphics, and this usually makes use of raycasting based rendering. Many other applications exist as well: for example early visualisation of architecture, computer art and cut-scenes for computer games. In short: raycasting based rendering is in general use for the creation of high quality 3D images and animations. Some examples of this can be seen in figure 3.1.

3.1 How raycasting works

The basic idea of raycasting is simple and straightforward: for each pixel on the screen, a ray is created from the camera through the pixel. This ray is shot into the scene and the colour of the first object that the ray hits is used as the colour for the pixel. This process is illustrated in figure 3.2. To be able to shoot a ray through a pixel, the screen is considered to be a plane in front the camera, so

that the pixel has a position p in the 3D world. If the camera is at position c, then the ray is defined as all points v_{xyz} defined by this formula:

$$v_{xyz} = c_{xyz} + \alpha(p_{xyz} - c_{xyz})$$

In this formula, α can have any value. Shooting the ray into the scene is done by taking the formula for the ray, and then iterating over all triangles in the scene. For each triangle, the triangle's formula is solved with the ray's formula. If there is no solution, then the ray does not hit that specific triangle. If there is a solution, then the distance from the camera to the intersection is calculated. Of all the intersections that are found, the closest one is used for the pixel.

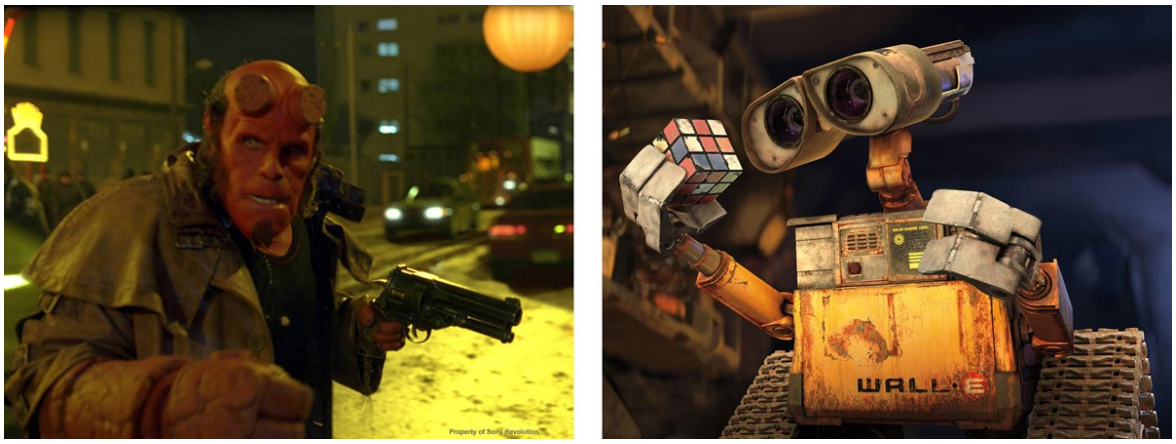


Figure 3.1. These images show examples of scenes that were rendered using raytracing. To the left a screenshot from the movie Hellboy (2004) is shown, in which the cars in the background where rendered using the raytracing renderer Brazil, and then combined with the filmed character in the foreground [55]. The right image shows a screenshot from Pixar's Wall-E (2008), a fully 3D rendered movie.

This simple algorithm does not only work with triangles, but with any 3D object that can be described by a formula and that can be solved with the ray's formula. Examples of such objects are infinite planes, spheres and even fractals. To add functionality to render such objects to a raycasting based renderer, only the algorithm to solve the formula for the shape with the ray needs to be added, so it is very easy to add new types of objects to the renderer. Figure 3.3 shows some examples of objects that can be rendered with raycasting.

So far it has only been described how the intersection of the ray with an object can be found. But how is the colour of the pixel calculated from this? If the object is unlit and has a flat colour, the pixel can simply use the object's colour. However, if lighting is required, then the incoming light at the intersection point is used to calculate the lighting at that point, using whatever formula the user wishes to apply to calculate the lighting there. So unlike the standard rasterisation pipeline, rendering based on raycasting always calculates lighting per pixel instead of per vertex. Standard texturing can also be added at this point. Usually texture coordinates are stored per vertex, so the

texture coordinates at the intersection point can be calculated through an interpolation of the vertices' texture coordinates. These coordinates can then be used to look up the colour in the texture.

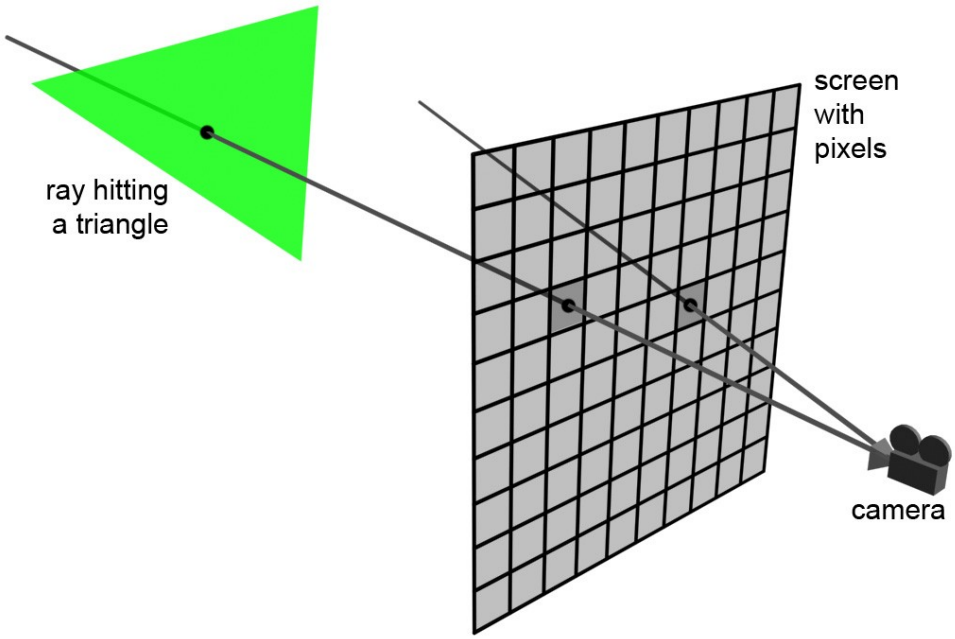


Figure 3.2. Casting rays from the camera through pixels on the screen and intersecting them with a triangle.

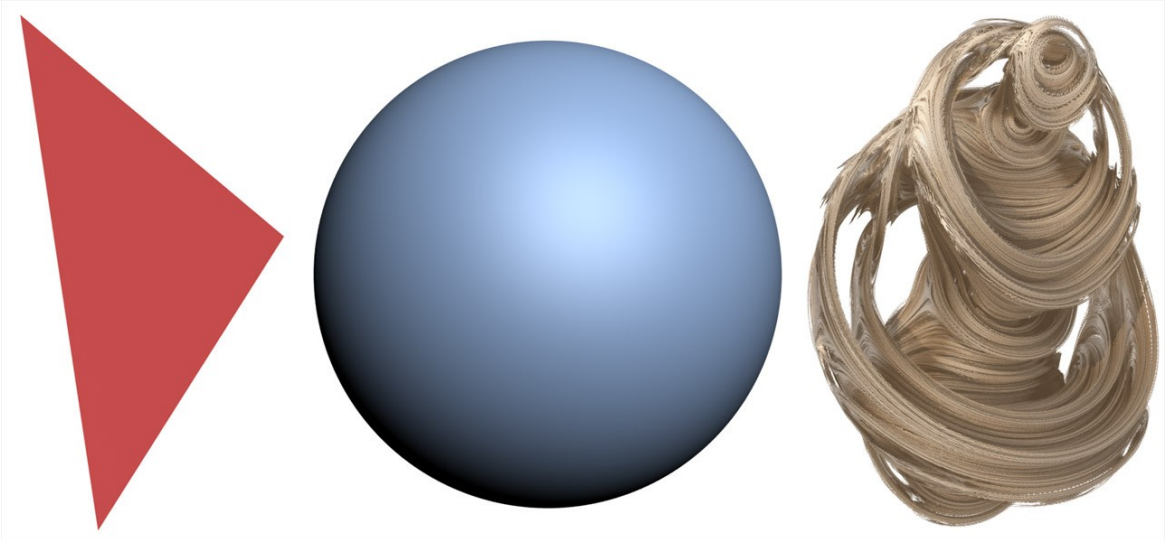


Figure 3.3. A triangle, a sphere and a julia quaternion fractal object rendered using raycasting. The latter image was taken from [5].

One of the greatest benefits of using raycasting is that rays do not need to start at the camera or go through a pixel. The same raycasting algorithm can be used for any ray, starting from any point in any direction. This is useful for many different applications; an example is rendering shadows. To calculate whether a point is in shadow, a ray is shot from this point towards the light source. If this

ray hits an object, then the point being rendered is in shadow. If the ray can freely reach the light, then the point is lit. This process is illustrated in figure 3.4. Rendering reflections is equally simple: the reflected ray is calculated and shot into the scene. This ray returns the colour of the reflection. If a reflected ray hits another reflecting object, then it will spawn another ray for that reflection, as can be seen in figure 3.5. As a path of rays is thus traced through the scene, techniques like these are called *raytracing* techniques (as opposed to ray*casting*).

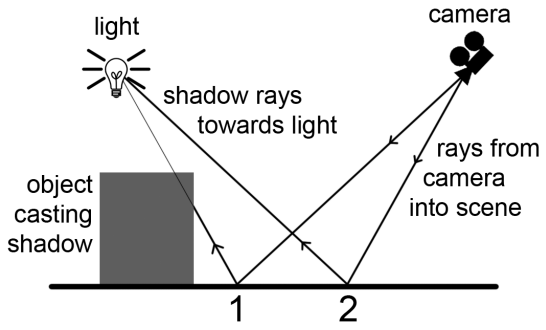


Figure 3.4. This image shows how shadows are calculated. Two rays are shot from the camera into the scene. Where a ray hits something, a new ray is created towards the light source. If this shadow ray hits an object, then that point is in shadow. In this image, point 1 is in shadow and point 2 is lit.

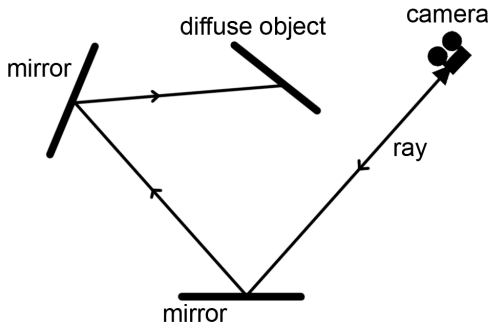


Figure 3.5. Here a ray is shown that is shot from the camera and then reflected twice as it hits two mirrors. After this, a diffuse object is hit, which does not reflect the ray any further. To calculate this, each reflection spawns a new ray that is again shot into the scene from the point of the reflection.

Basic shadows and reflections each require only a single extra ray. By shooting rays in several directions, more advanced effects can also be created, as is shown in the following example. In rendering, lights are often considered to be infinitely small points. This results in razor sharp shadows, which is not very realistic. The alternative is to render area lights, which have an actual surface. These produce blurred shadows, as can be seen in figure 3.6. However, it is difficult to determine exactly for a point in the scene what percentage of the area light is obscured by other objects, i.e. for how much the point is in shadow. By shooting lots of rays towards different points on the surface of the area light and determining for each ray whether it can reach the light without hitting other objects, the percentage of the light that is visible can be determined. However, to find an exact answer this way, an infinite number of rays would be required. This is of course not possible, so instead a large number of rays is cast and the error is accepted. If the number of rays is too low, then this error shows in the final image as noise.

Another difficult problem that can be solved with relative ease using raytracing, is to calculate global illumination. In real-life, when light hits an object that is not perfectly black, the light is reflected further into the scene and contributes to the lighting of the rest of the scene, where the light is again bounced further. An example of this can be seen in figure 3.7. With each bounce, the light loses some of its energy, so it does not keep bouncing indefinitely. In the case of a pure specular object,

like a mirror, the light is reflected into a single direction. However, most objects in the real world diffuse the light and spread it into all directions. To achieve realistic lighting, global illumination is very important. However, it is also difficult to model this effect when rendering. With raytracing a simple solution exists to calculate this kind of complex lighting for a point in the scene: shoot many rays from this point into the scene to find the light that is contributed by other parts of the scene. The sum of all the light gathered thus is the global illumination. To simulate more light bounces, each ray should in its own turn spawn more rays from the point where it hits the scene. Some threshold should be added to limit the number of bounces of the rays.

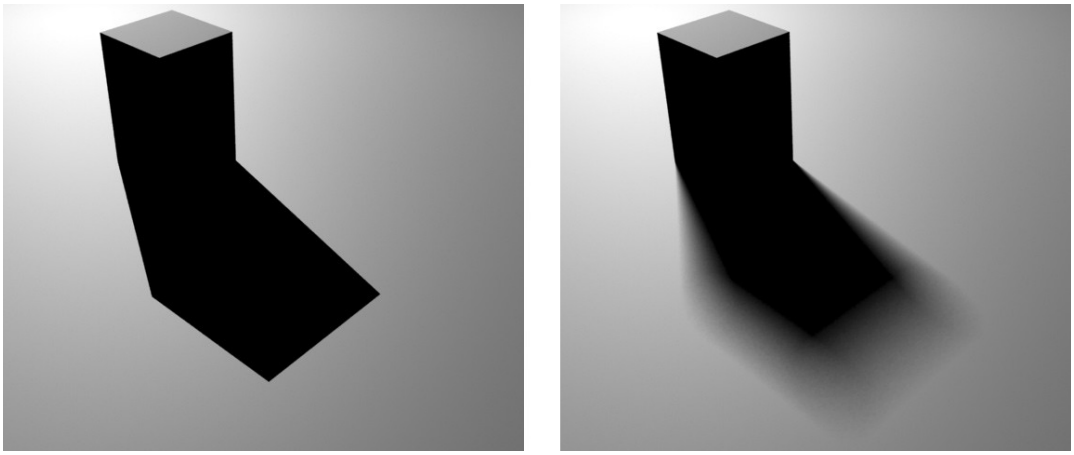


Figure 3.6. To the left the sharp shadows cast by a point light. To the right the more blurry and realistic shadows cast by an area light. Both images were rendered using raycasting.

Similar to the situation with area lights, the result is only exact if an infinite number of rays is used. This is impossible, so again by shooting a limited number of rays, raytracing introduces an error into the rendering here. This error is smaller if more rays are used. Here an important problem arises: for most scenes the number of rays required for an acceptable image quality is so large that the user might have to wait several hours or even days for a single image to be rendered. This raises the challenge of reducing rendering times, which is discussed further on in this chapter.

When doing raycasting, anti-aliasing is another problem that is solved by simply casting more rays. Instead of casting a single ray per pixel, the pixel is considered to be a small area, with several rays being shot through different parts of that area. The final colour of a pixel is the average of the colours of all the rays through that pixel. Depending on the desired quality, more or less rays can be shot per pixel.

Because casting an infinite number of rays is impossible, a limited set must be chosen. But which rays should be shot to get a good representation of the scene? Just randomly shooting rays in any direction is called *monte carlo* raytracing. If the number of rays increases, then the resulting solution diverges towards the actual solution, so this is mathematically correct. However, shooting in random directions results in very visible noise, which tends to only go away at infeasible numbers of rays. A number of solutions exist for this. One observation is that the directions from which most

light is coming, are most important to the final result. If the global direction of the light is known, then the random rays can be biased to tend more into that direction. This reduces noise, but the biasing makes the result incorrect, which is in turn compensated by dividing the results by the bias. Although this might seem a good solution, it is often not very useful: the directions from which most light is coming are not known, so a good biasing function is usually not available. A better solution is to shoot the rays in fixed directions, for example in a grid, thus making sure that rays are shot evenly in all directions. This, however, causes patterns in the lighting. It turns out that in practice a blend of these two techniques works best: shoot the rays in a grid, but randomly displace each ray within its grid cell. This way the number of rays can be kept relatively low, while still achieving good results.

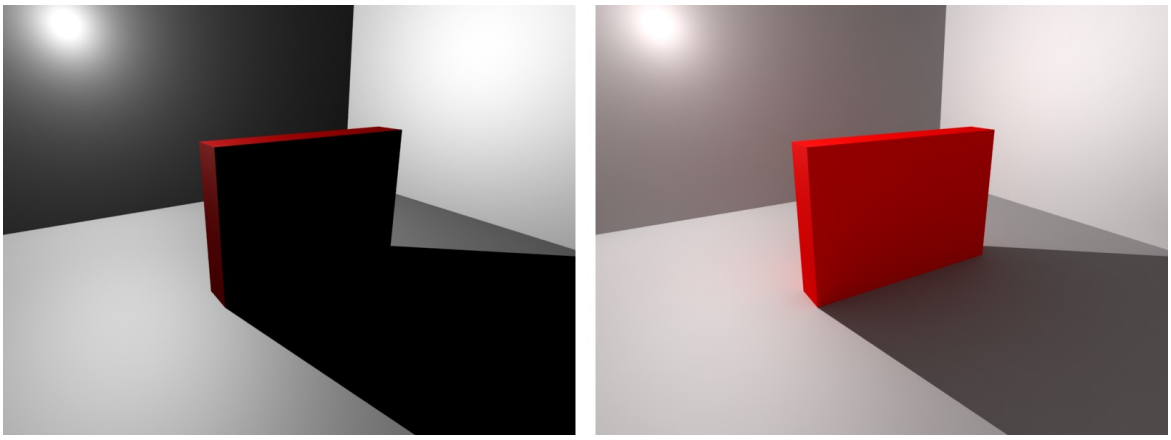


Figure 3.7. The same scene, rendered to the left without and to the right with global illumination. The only light in the scene is located in the top left corner. Note how without global illumination, the front of the red object is not lit by the light at all, while with global illumination, it receives light via the walls around it.

3.2 Optimisation

Raycasting is a great technique to easily render various complex effects, but its downside lies in its slowness. If a screen of 1024 x 768 pixels needs to be rendered, then this already requires 786,432 rays. To add anti-aliasing, each pixel requires at least four rays. To add modest quality global illumination, each ray will often easily require a further one hundred extra rays to find lighting coming from all directions, or even more if numerous light bounces are to be taken into account. This already brings the number of rays cast for a single image to over three hundred million. If the scene contains one million triangles, then a collision check is required for each of these rays with each of these one million triangles. Obviously, calculation times explode here, so one of the most important research areas in raycasting has been optimisation techniques that can decrease the rendering time per image. Here some of these optimisations are described.

3.2.1 Algorithmic optimisations

The time it takes to find the closest intersection of a single ray can significantly be reduced by using smart data structures to store the scene in. Structures like octrees, BSP trees, bounding volumes and kD trees all group triangles that are close to each other into single volumes. If the ray does not hit such a volume as a whole, then surely it does not hit the triangles inside it. This way, after a single intersection calculation, a large group of triangles can be skipped. In an ideal scene, with `n` being the number of triangles in the scene, such data structures might decrease the complexity of casting a single ray from `O(n)` to `O(log n)`. In the worst case, if all triangles intersect with each other, then such data structures would not have any effect at all, because they cannot be split into groups. In practice, triangles do not overlap that much, so usage of a good data structure would strongly decrease the time required for raycasting. An overview of a large number of different types of data structures can be found in [3].

Although the edges of polygons and shadows are sharp, many other lighting effects are very smooth. Some optimisation techniques are based on this: smooth values are calculated for a limited number of points and then interpolated for the areas in between them. This way the costly calculation of global illumination can, for example, be done once for every 16 pixels, while all pixels in between can use interpolated values. This greatly decreases rendering times, but does not work in every case: if an object boundary is exactly in between two global illumination samples, then these values cannot be interpolated. In this case the renderer needs to detect this and sample more border points. Criteria to add more sample points are not only object borders: sharp curves in the object, or strong changes in lighting between samples might also mean more samples are needed. Smart choices are required here: if the renderer takes too many samples, the rendering time decreases. If the renderer takes too few samples, then values that are not smooth might get interpolated incorrectly, resulting in images of lower quality. An example of this can be seen in figure 3.8, while an example of how to progressively improve the quality of anti-aliasing this way can be found in [45].

When rendering an animation of a camera flying through a static scene, the diffuse lighting remains exactly the same throughout the animation. So far the lighting calculations have not made use of this knowledge: global illumination was just calculated for every pixel that was seen from the camera's point of view. Therefore, during an animation, the lighting of the same point will probably be calculated again and again for many frames. This allows for a strong performance increase, if one first stores any lighting values that are calculated in the scene, and when rendering a pixel, first looks for values in the vicinity to see whether there is already enough lighting information there. Especially further on in the animation this will be a strong form of optimisation, as increasingly more values will have already been calculated and stored in the scene.

This might even be taken a step further, if, instead of calculating lighting from the point of view of the camera, one shoots lighting energy into the scene from the light itself before rendering. This makes more sense, as it is now sure that every packet of light sent into the scene actually contains a

significant amount of lighting energy. A single packet of lighting energy is called a *photon*. After sending the photons into the scene, the *gathering phase* starts: the image is rendered and for each pixel the relevant photons are gathered and taken into account. A downside of this technique is that large amounts of photons are required to get good lighting, which not only requires extra calculations, but also extra storage. Another problem is that often many photons will be stored in parts of the scene that are never seen by the camera and are therefore useless. However, if global illumination is required to have many bounces, then shooting photons often results in better performance than rendering purely from the camera's point of view. An overview of rendering methods that use variations of this technique can be found in [18].

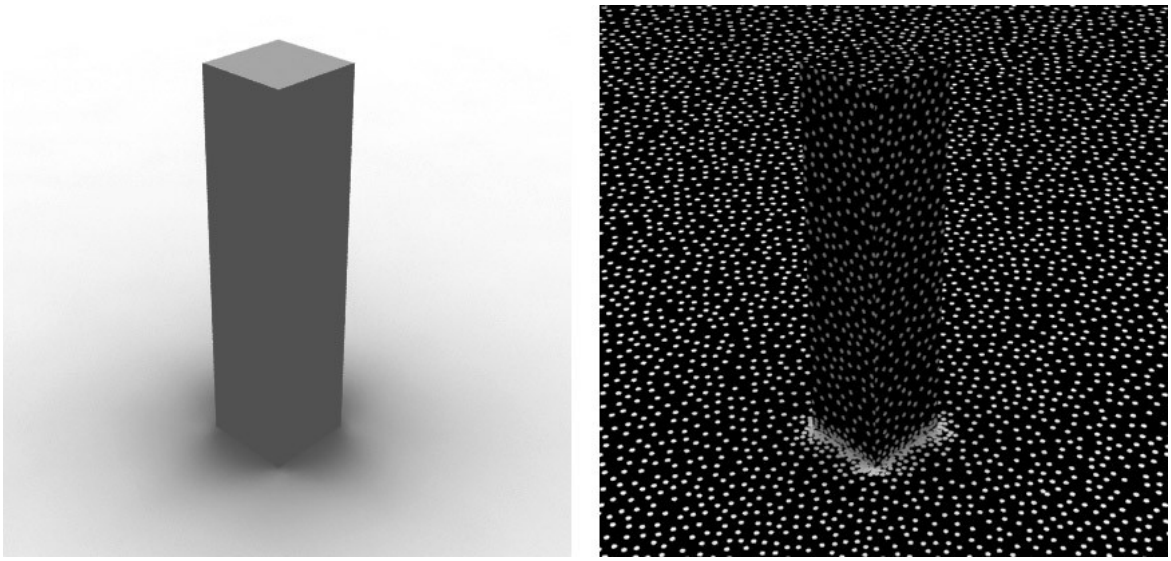


Figure 3.8. The left image shows an example of calculating global illumination only for a limited number of points and filling in the other pixels through interpolation. To the right a visualisation is shown of the samples that were actually calculated. As can be seen, the number of samples is larger where the lighting has more variation. These renders have been created using Vray 1.5 in 3D Studio MAX 9.

3.2.2 Hardware optimisations

The optimisations discussed so far sought to decrease the rendering times through smarter algorithms. A totally different approach is to use the hardware more optimally. Raycasting can very easily be calculated in parallel, as every raycast is a totally independent calculation and can thus be done on a different processor, even in a different computer. Many modern computers have two or four processor cores, so this can double or quadruple performance. *Rendering farms* take this a step further: here hundreds or even thousands of computers are linked together and all calculate part of the image. This is an expensive solution, but is often used for large productions, such as 3D animation for feature films. Another optimisation through hardware is to use specialised raytracing hardware. As in the case of a GPU, this hardware only needs to perform a specific type of calculation, in this case raycasting, and can thus be optimised to do this much faster than a CPU.

Although actual specialised raytracing hardware is rare, in March 2009 Caustic Graphics has announced that it will release a raytracing add-on card [12].

A hardware optimisation that deserves special attention in the context of this thesis is performing raycasting on the GPU. Although GPUs are not made specifically to do raycasting, they do often contain dozens of parallel shader processors. If raycasting can be implemented in such a way that it can be performed in shaders, then using all these parallel processors in the GPU becomes a very strong optimisation. Although slightly awkward to implement, it is indeed possible to implement raycasting on the GPU. All the data about the scene and the rays to be cast needs to be stored in textures. The colour of a pixel might for example represent the 3D coordinates of a vertex. The shader can then perform texture reads to get the scene data and use this to calculate intersections between the ray and the scene. Results are then output as pixel colours. This way the programmer is working around the standard graphics pipeline. Figure 3.9 shows an example of how such a shader based renderer may be structured.

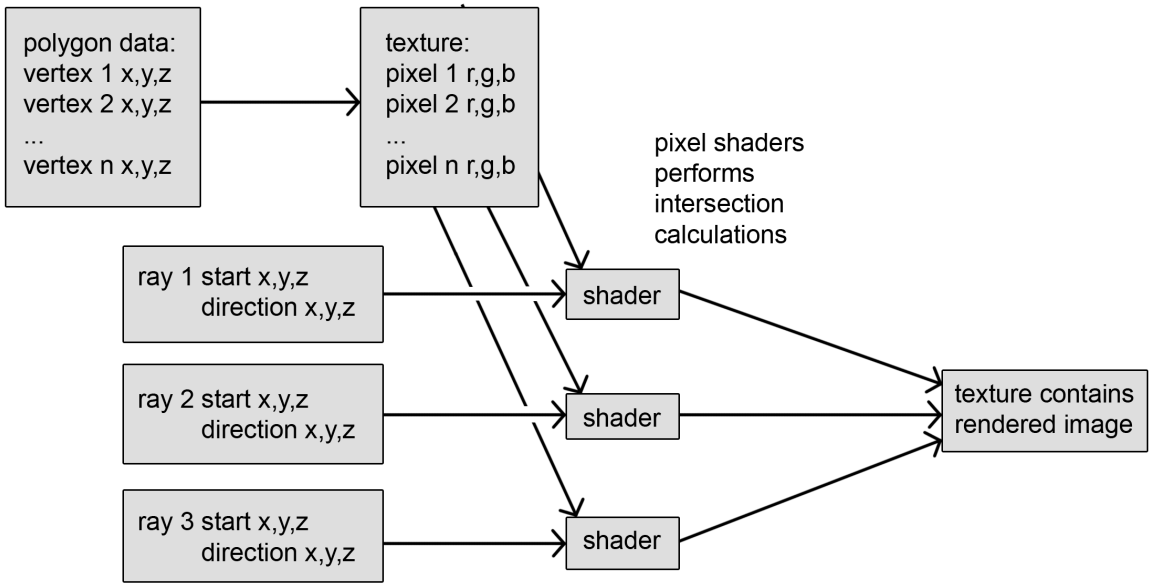


Figure 3.9. This image shows a simplified example of how GPGPU raycasting might be implemented. Vertex coordinates of the polygons are stored as rgb colours in a texture. For each pixel of the final image, a pixel shader calculates the intersections of the ray with all the polygons.

To make this development process more streamlined, programming languages such as CUDA [41] have been developed, which offer a more general programming environment than standard shaders. The field of general programming on GPUs is called GPGPU, as was already briefly mentioned in section 2.3.3. Production renderers that use raycasting on the GPU have already been developed and are currently in use [42], proving that this field of research is a valid approach to optimising renderers that use raycasting.

The results of optimisations through faster algorithms and hardware are much lower rendering times than without them. However, no matter how many optimisations are applied, there is

currently no way to apply raycasting to achieve the same framerates as rasterised rendering for scenes of equal complexity. A technique such as GPGPU might decrease rendering times by a factor ten, but going from one hour per frame to six minutes per frame is still nowhere near the 0.02 seconds per frame that can quite easily be achieved through rasterisation. For this reason, rasterisation is still the technique of choice for anything that requires real-time framerates, while raycasting is used for high quality graphics that do not have to be rendered in real-time.

4. Existing shader techniques that use raycasting

As has been shown in chapter three, raycasting is used for many different aspects of high quality rendering. From figuring out what is seen through a camera, to calculating light that bounces from object to object and light scattering through volumes. In many cases, raycasting only delivers an approximation of the exact values being calculated, and other approaches exist as well, but raycasting is often both the simplest and the most efficient solution. This makes it obvious to consider using raycasting to achieve various graphical effects in real-time graphics. However, in practice these solutions are often not feasible.

The first problem with raycasting is that it is too slow for real-time graphics. Polygons can be rendered very efficiently using rasterisation, especially with rendering hardware that uses specialised hardware to calculate the colour of each pixel of a polygon on the screen. Raycasting on the other hand is much slower and only rarely achieves real-time framerates. A lot of research has been done on the subject of decreasing the rendering times of raycasted graphics, and although some of these projects actually did achieve real-time framerates, this often required either specialised hardware that is not widely available on consumer computers [51], or heavy simplifications of the complexity of the scene that was being rendered, or letting go of many of the more complex and realistic lighting calculations that enable raycasted graphics to generate their much more realistic images [50]. Because of this lack of speed, raycasting is rarely used for real-time graphics, apart from in research projects.

Rasterised graphics on the other hand are significantly limited in the effects that can be achieved with them. Using programmable shaders, many visual effects are possible in real-time, but these are often either very rough approximations of the physical effect that is being simulated, or they are too slow to be used in the complex scenes that modern games and virtual environments require. To achieve more realistic and more complex visual effects, it seems interesting to combine rasterised graphics with raycasting: the basic polygons are rendered quickly through rasterisation, but certain effects are added to the rendered polygons through raycasting, thus allowing for a wider range of effects to be implemented.

Combining rasterisation with raycasting in programmable shaders seems to be a very promising approach, but due to the limitations of graphics hardware, it is much more complex than might appear at first glance. The first problem is that calculating what a ray hits, often requires a lot of calculations: a data structure that stores the polygons needs to be traversed, and the ray must be intersected with these polygons or with other primitives. This requires a lot of instructions. Older shader models have quite severe limitations on the number of instructions that can be performed, making raycasting very difficult to achieve. Newer consumer graphics hardware, however, has removed these limitations, but this is only a partial solution: these days, large numbers of

calculations can be performed in a single shader, but if the number of instructions becomes really large, this will simply take too much time to still achieve real-time framerates. For this reason, at this moment only those raycasting techniques that require relatively few instructions are feasible in real-time.

Another problem is memory. Shaders have only a limited amount of memory that they can access easily [29], so it is not possible to simply store for example a million polygons in memory and let the shader access them. In fact, it is often not even possible to easily store a hundred polygons. The only way to feed the shader large amounts of data, is by storing it in textures. Although this is an awkward thing to do, because it has little to do with what a texture conceptually is supposed to be, it does work. However, reading the data from textures only brings back the problem of performance: a single texture read is not a very expensive operation to perform on modern graphics hardware, but doing dozens of texture reads in a pixel shader would be much too slow for real-time. The conclusion of this is that shaders that use raycasting must either access just small amounts of data, or be performed on small numbers of pixels only.

Because of these severe limitations on the number of instructions in the shader and amount of data available, raycasting is not used as much in programmable shaders as might be expected. Many of the most interesting techniques, such as sub surface scattering or global illumination, are just not feasible on current hardware, and this is not expected to change in the coming years. The most important techniques that are currently in use, are discussed in this chapter. The only technique that is really widely used, is to apply environment maps to approximate reflection and, sometimes, refraction. A related technique, but currently still experimental, is caustics, i.e. rendering the focussed light that is refracted by a lens or reflected by a mirror. A technique that is not widely used yet, but is coming up quickly, is displacement mapping, which adds detailed and perspectively correct relief to flat polygons. The next technique is screen-space ambient occlusion, which approximates light coming from the sky from all directions. Similar in approach to this technique is the one used to render light shafts. The last effect that is discussed in this chapter is rendering the refraction that is caused by the eye's lens. This technique is rarely used in real-time virtual worlds, but uses an interesting approach to the combination of rasterisation and raycasting that is not seen in any of the other techniques discussed in this chapter. Finally, although the subject of voxel grids is related to the other topics treated here, it is instead discussed in section 6.2.2, as voxels are used to render volumetric effects.

A topic that might be expected in this chapter is full GPU raytracing, as was already mentioned in section 3.2.2. In this class of techniques, a traditional raycasting renderer is executed through pixel shaders. Although this is an interesting and promising technique, it will not feature again here: this thesis is about improving rasterised graphics through raycasting, not about replacing rasterised graphics with raycasting altogether.

4.1 Reflection and refraction through environment maps

The single most common application of raycasting to enhance rasterised graphics, is rendering reflections. Many real-world materials, like metal, glass and polished surfaces, contain reflections. In the case of glass, there are also refractions. Because pure reflections are relatively sharp, they are very noticeable to the human eye. Therefore, rendering reflections in real-time graphics is an important technique that can often be applied. Refractions occur less often, but are a related topic, and as the solution to render refraction is so similar to the solution to rendering reflections, refraction is also discussed here. Examples of both techniques can be seen in figure 4.1.

Figure 4.1. These two photos show examples of reflection and refraction. To the left the reflection in a television screen is shown. The right image shows a magnifying glass.

Although polygons can easily be rasterised for direct rendering, this becomes much more difficult when these polygons are reflected in a window. If the window is flat, than the polygons can be transformed to a mirrored location and rendered there, but in the case of a curved mirror, the entire scene would have to be re-rendered in a mirrored position for each polygon, which is far too slow to actually do. If the mirror itself also has a normal map that contains small details, such as waves in water, this technique becomes impossible altogether. Therefore, pure rasterisation cannot render reflections. In raycasting however, reflections are very simple: when a ray from the camera hits a polygon, it is reflected and cast into the scene again from the point of reflection. Conceptually, combining raycasting and rasterising is simple in this situation: polygons that are directly visible from the point of view of the camera are rendered through rasterisation, while reflections on those polygons are rendered through raycasting. To calculate the reflection ray, the ray from the camera to the polygon is needed, but this can easily be calculated for each individual pixel that is rendered by the rasteriser.

Although the basic idea of this combination of rasterisation and raycasting is simple, the execution is practically impossible: casting a reflection ray and intersecting it with all the geometry in the scene is not feasible on the GPU at real-time framerates. To solve this, an approximation of what the ray hits is used. First the scene around a reflecting object is rendered to a special texture (usually

either a sphere map or a cube map), which is often called an environment map. Then the environment map is used to look up what the ray would hit in a certain direction. The environment map stores the colour that would be found in any given direction. Thus, calculating the colour of the object that the reflection ray hits requires only a single look-up in the environment map, instead of intersecting the ray with all the objects in the scene to find out what it hits.

For this texture look-up to return the correct value, the environment map must be five dimensional: the ray can have any direction and begin from any point. Direction is 2D and position is 3D, so in total, that makes a 5D texture. There is a simple optimisation, though: as reflection rays can only begin on the surface of the reflecting object, the origin of the ray is always on its surface and is therefore not 3D, but only 2D, reducing the dimensionality of the entire environment map from 5D to 4D. However, 4D textures at high enough resolutions to allow for sharp reflections are still too large to fit in memory. Also, recalculating the entire 4D texture every frame would take too much time to achieve real-time framerates. Instead, the environment map would have to be pre-calculated and used for every frame, so it could still not be correct for dynamic scenes. Current GPUs, finally, do not support 4D textures, so the texture would somehow have to be compressed to fit in a 3D texture.

To reduce the dimensionality of the environment map, a simplification is used. Instead of calculating the colour of the scene in any direction from any point on the surface, it is only calculated for the centre of the object. What remains to be stored, is the resulting colour for rays in any direction. This makes the problem only 2D, which can easily fit in memory and can be recalculated often for dynamic scenes. The drawback of this approximation is that the reflection is rendered as if the scene is infinitely far away from the reflecting object, so the closer the rest of the scene is to the reflecting object, the more incorrect the reflection becomes. Another disadvantage is that inter-reflections of the object are not possible any more, so an object cannot reflect other parts of itself. Nonetheless, a great advantage is that, through the technique called *render to texture*, the scene can quickly be rendered to the environment map by the GPU itself, after which the GPU can use this environment map to render the reflections.

As environment maps are 2D, they can easily be stored in a texture. However, they are indexed through a direction instead of through standard texture coordinates, and are therefore also stored slightly differently. Two approaches are widely in use: sphere maps and cube maps. In the case of a sphere map, the reflection rays are considered to be on a sphere around the centre. The resulting texture leaves the corners of the texture empty, because the top and bottom of the sphere have a smaller radius than the centre. A cube map considers the directions of the rays to be on a cube around the centre. The cube has six faces, so the environment map is stored as six different textures, one for each side of the cube, as can be seen in figure 4.2. Unlike sphere maps, cube maps have no unused texture space, but this comes at a trade-off: the amount of detail of the cube map is lower at the corners, and if the cube map has a very low resolution, then its seams are often visible because of the lack of bilinear filtering between the different faces at the seam.

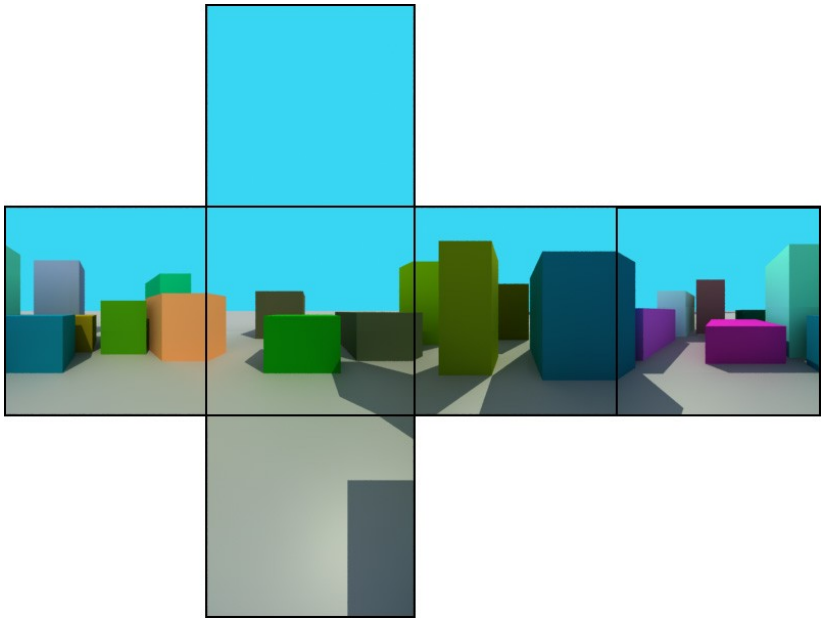


Figure 4.2. An example of a cube map. The six textures together span all directions.

Because reflections are used for so many different materials, cube maps and sphere maps both have specialised support on GPUs. It is possible to access these textures using a 3D vector, instead of through 2D texture coordinates. This greatly simplifies shaders that feature reflections, because the reflection ray can now be used directly and there is no need to map the reflection vector to 2D texture coordinates. Rendering reflections can even be done without using programmable shaders at all: the fixed function pipeline of the GPU can be set to render reflections using an environment map. Because environment maps are fully supported in hardware, they can be rendered very fast. Consequently, they are often used in virtual worlds. As can be seen in figure 4.3, 2D environment maps can produce quite convincing reflections, even though the results are not actually correct.

Figure 4.3. Examples of reflections using environment maps in games. To the left reflections in a car in a screenshot from Electronic Arts' game Need For Speed Undercover (2008). To the right reflections in a weapon in Serious Sam II (2005) by Croteam.

The major drawback of environment maps is that generating the map itself is an expensive operation, as it requires rendering the entire scene to a texture. In the case of a cube map, it even requires rendering the entire scene six times to the six different sides of the cube. Sphere maps

cannot be rendered directly by the GPU at all because of the spherical distortion. Extra renderings of the entire scene for every frame would strongly decrease the framerate, so a number of optimisations are in general use. The most common one is rendering the scene in a pre-processing phase, storing it in a texture and then just loading it when rendering the virtual world. This makes the environment map completely static, meaning that it cannot react to changes in the environment and cannot reflect completely dynamic objects like moving cars or characters. However, because this method has hardly any overhead while running the virtual world, it is the solution that is used most often. More performance-heavy solutions are to render only one face of the cube map each frame (thus decreasing the framerate of the cube map itself), rendering the cube map at a very low resolution (thus making the reflection more blurry), and rendering only parts of the scene to the cube map. This last solution often works very well, because many details in the reflection will hardly be noticed by the player, while objects that stand out to the viewer are still visible this way. An example of this, is a situation in which characters are only rendered to the cube map if they are near the reflecting object, and are otherwise ignored for the reflection.

A slightly more complex alternative is to analyse the surroundings of the reflecting object and only re-render the environment map once the surroundings have changed enough. In a mainly static environment, this will greatly reduce the number of times the environment map needs to be rendered. Also, in the case of a cube map, only the faces that feature the changes in the scene need to be re-rendered.

Because standard cube maps are used as if they are placed infinitely far away, they introduce perspective distortion when the ray's origin is further from the cube map's origin. One solution is to store the distance to the origin of every pixel in the cube map [56]. The result is a cube map with a depth map applied to it, which can then be raytraced in the same way as is done with displacement mapping. This is further discussed in this chapter in section 4.3. Visually, the results are very good, as can be seen in figure 4.4. This is not fully correct, though: a depth map can only store what is visible from the point of view of the cube map's origin. Consequently, not everything can be rendered correctly this way, as parts of the scene might be occluded when looking at them from the centre of the cube map, while they should have been visible from other points on the object's surface.

Using environment maps, not only reflection can be rendered, but also refraction, for example by a water surface or by a glass object, such as a lens. In the pixel shader, the normal of the surface and the ray from the camera to the surface can be used to calculate the refracted ray. This calculation is not supported in the fixed function pipeline and can therefore only be done in a programmable shader. The refracted ray can then be used to look-up the colour of the scene in that direction. An example of this technique can been seen in figure 4.5. Nonetheless, this technique does present a huge drawback: it only takes into account the refraction where the ray enters an object and ignores the refraction where the ray leaves the object. The error this introduces is clarified in figure 4.6. Visually, this is not necessarily a big problem: the user might not be aware that the refraction that is shown is in fact incorrect. However, if a depth cube map of the object itself is made, then this can be

used to calculate the place where the ray leaves the object [59]. By also storing the normals of the object along with the depth cube map, all information is available that is needed to calculate the refraction of the ray when leaving the object. This process is further discussed in the context of caustics in section 4.2. As this uses a depth cube map, this is only fully correct for convex objects, as rays might otherwise pass through different parts of the object several times, which cannot be modelled with a single depth cube map.

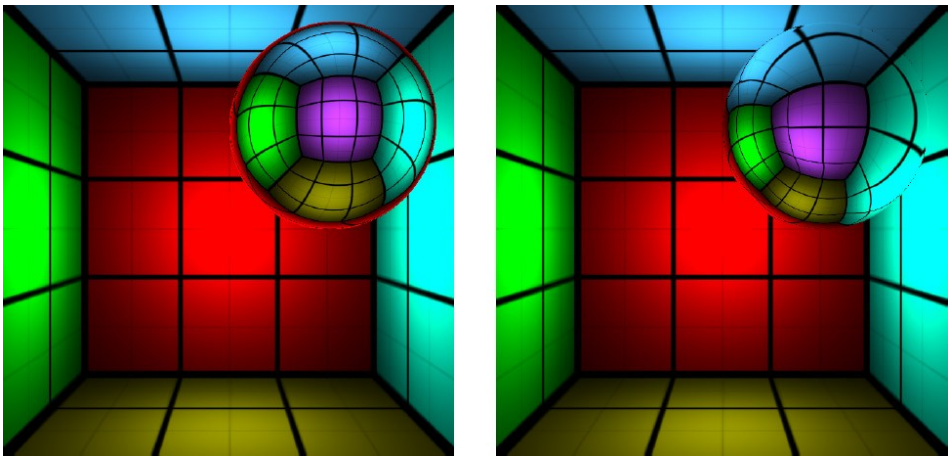


Figure 4.4. To the right an example of rendering reflections using a cube map with an added depth map. To show the improvement, a render of the same scene without the depth map is shown to the left. Both images from [56].

For the specific case of reflection on a plane, a solution that is simpler than an environment map can be used. Reflections on a single plane are a common problem, for example in the case of a mirror or a water surface. The approach here is to mirror the camera in the surface of the mirror and then render the scene from that point of view, as can be seen in figure 4.7. Objects that are beneath the surface are removed from this rendering by using the mirror's surface as an extra clipping plane. Now when the scene is rendered from its normal viewpoint, the render from beneath the mirror is mapped to the plane in such a way that it makes for a correct reflection [14]. This technique does not support curved surfaces, but smaller distortions are possible, for example in the case of ripples in the water, as can be seen in figure 4.8. An important drawback of this technique is that the render from beneath the mirror cannot be reused when the camera has moved, so it must be re-rendered every frame.

Regardless of whether cube maps or plane reflections are used, today's graphics hardware is well capable of rendering real-time reflections. Re-rendering the environment map every frame remains an expensive operation, but graphics hardware is fast enough to do this and still be able to deliver detailed graphics in the rest of the scene. Of course, the fact remains that if no reflections were present, more other effects could have been rendered while achieving the same framerate.

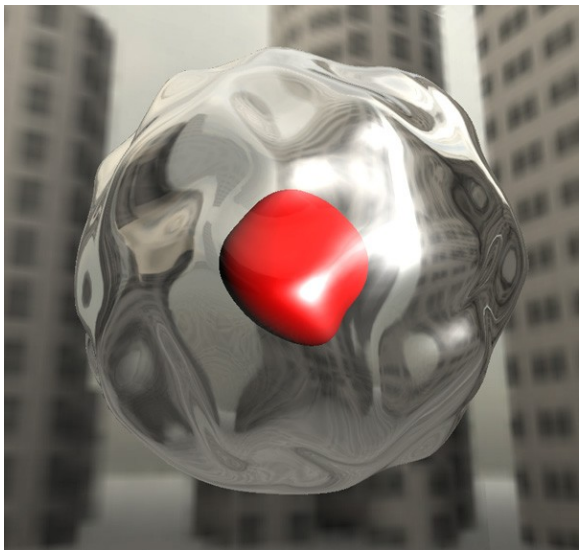


Figure 4.5. This image shows an example of real-time refraction rendered through the use of an environment map and a pixel shader to calculate the refraction ray. In this case, the ray is only refracted where it enters the object and not when leaving it.

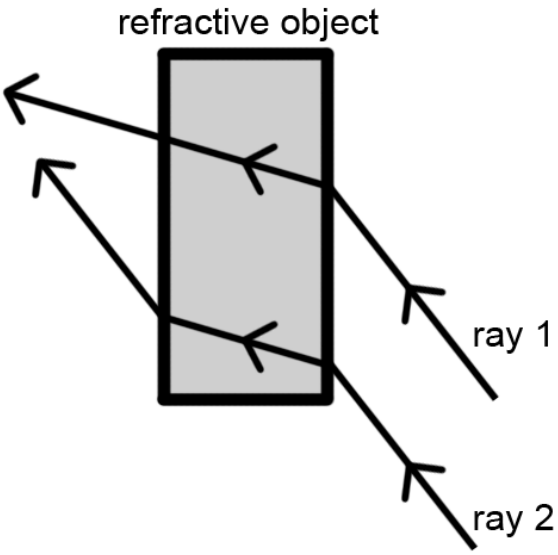


Figure 4.6. In this image, ray 2 is correctly refracted: it changes direction both when it enters the object, and when it leaves it. Ray 1 is how refraction is often calculated in real-time applications: it is only refracted where it enters the object, but ignores the object when leaving it.

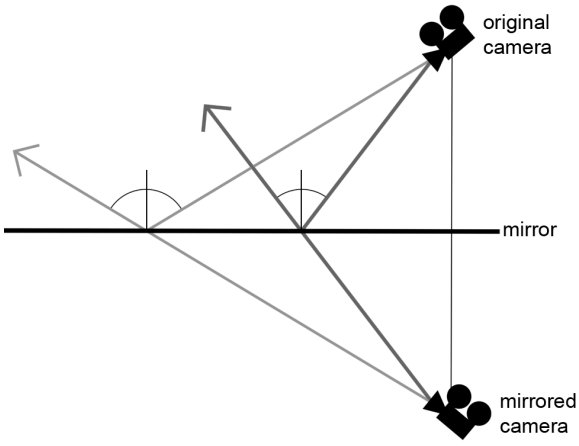


Figure 4.7. Mirroring the camera in the water's surface to render the reflection yields the same results as actually mirroring the rays from the camera.

Figure 4.8. In Crytech's game Far Cry (2004), water reflects the environment, including the distortion caused by ripples.

4.2 Caustics

The previous section discussed how to render the colour of an object that features reflection and refraction. However, such an object also influences the lighting of its environment: any light that illuminates the object will be reflected or refracted unto other objects. The effect of directly reflected or refracted light unto the surroundings of an object is called caustics. A well-known example is a lens, which can be used to concentrate light into a small bright spot. Another common example is a glass of water, as can be seen in figure 4.9. Caustics are only visible when caused by highly reflective or refractive objects, but in these cases, they are also a very noticeable visual effect. Therefore, in the case of glass objects, it greatly increases the visual realism if caustics are rendered.

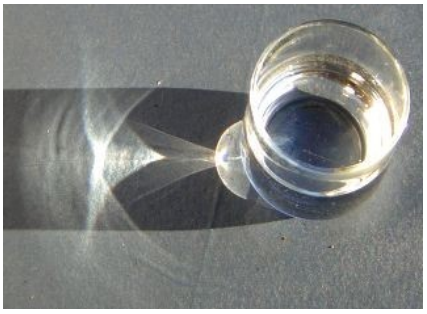


Figure 4.9: This photograph shows the caustics created on the ground by a glass of water.

The most common approach in non-real-time renderers to rendering caustics, is to shoot a large number of photons from a light source towards the reflecting or refracting object, calculate how they are reflected or refracted, and then store where these photons landed on the environment. This is done before the actual rendering takes place. After that, the scene is rendered from the point of view of the camera. Now, when a point on a certain object is being rendered, its environment is searched to find any photons that landed very near to that point. The energy of these photons is then used to calculate their contribution to the lighting at that point [1]. The steps of this method are illustrated in figure 4.10.

The above approach includes a number of steps that cannot efficiently be implemented on the GPU in real-time. Especially shooting photons into the scene through raycasting and finding nearby photons while rendering objects are not feasible in a shader: they take too long to achieve real-time framerates and require too much data to be used in the pixel shader. Therefore, these steps will somehow have to be simplified to be workable on the GPU in real-time.

In [59], a technique called *caustics triangles* is introduced. This paper uses the following approach. The first step is to render the object that is causing the caustics into a depth cube map, including the normals at the rendered points. This is done from the centre of this object. Next, the rest of the scene is rendered into a depth cube map as well, again from the centre of the object. Then the object is rendered from the point of view of the light, with a special pixel shader for the object. Each pixel is

considered to be a photon that is shot into the scene. For each photon, the ray from the light to the point on the object is refracted. The ray is now inside the object and the next step is to use the depth cube map of the object itself to find the place where the ray leaves the object again. At this point, the ray is once more refracted, using the normal that was stored with the depth cube map. The ray is now outside the object and is traced into the scene using the depth cube map of the scene. This results in the point in the scene where the photon hits a diffuse object. Finally, the direction from this point towards the centre of the depth cube map is stored in the pixel that was being rendered from the point of view of the light.

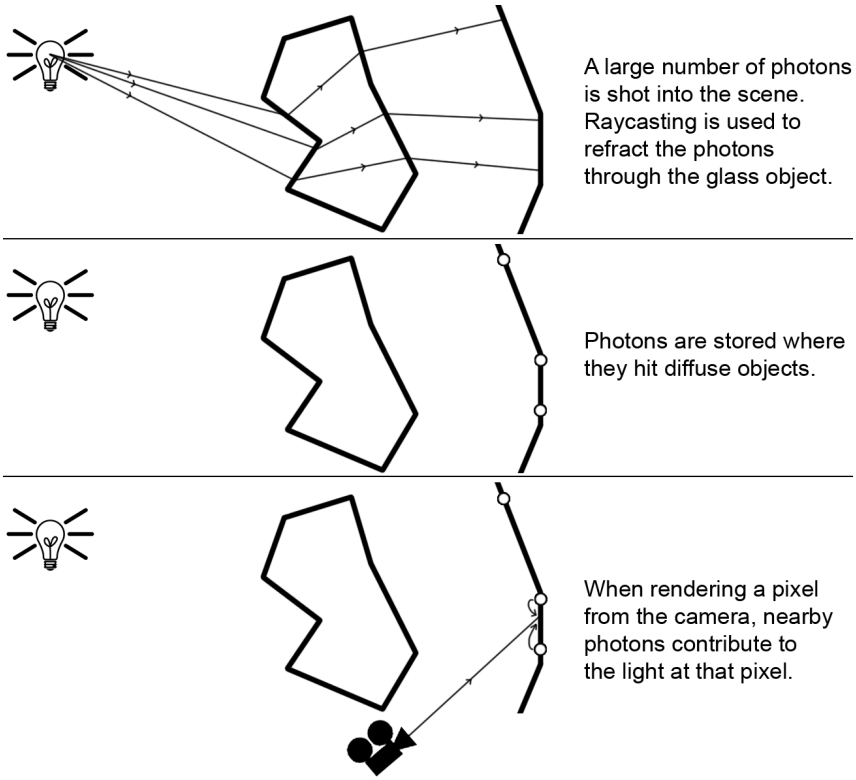


Figure 4.10: The steps of the traditional photon shooting approach to rendering caustics. The centre object is made of glass and thus refracts the photons.

At this point it is known where all the photons hit the scene, but this information is not ready to be used during rendering yet. In the next step, the directions that were stored are rendered to another cube map using *caustics triangles*. Each pixel that was rendered to a texture in the previous step represented a photon, and each photon is now rendered using a separate triangle. This conversion from pixels in the texture to triangle coordinates can fully be done on the GPU by letting the vertex shader of the triangle read from the texture and generate the position of the triangle from that.

Now everything is ready to finally render the scene. For each pixel of the scene that is rendered, a look-up is done into the cube map to which the photon triangles were rendered. As photons represent light energy, the value that is found in the cube map is just considered to be another light

source and is added to any direct light also coming in. A breakdown of the entire technique can be found in figure 4.11.

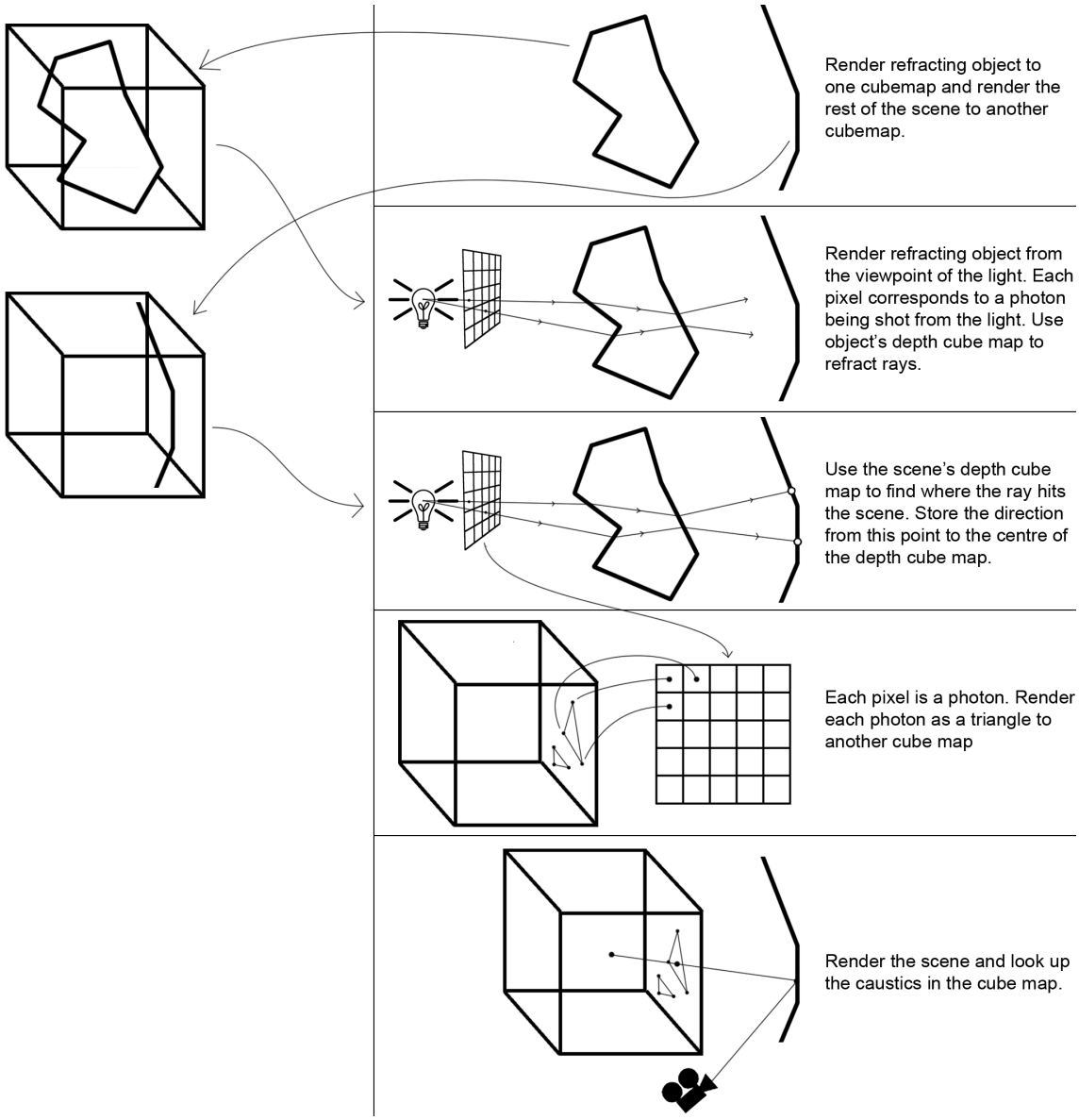


Figure 4.11: The steps in the caustic triangles *technique.*

The above technique is only correct if the object that is causing the caustics is convex, because it is represented using a depth cube map. For concave objects, one can choose to split the object into convex parts and create a separate depth cube map for each, or to simply accept the errors. In most cases, the caustics look very convincing even though these errors are introduced, as can be seen in figure 4.12.

The technique presented here is complex and requires a number of additional rendering steps. Because of this, it places a heavy burden on the framerate. However, as all steps are implemented on

the GPU, real-time framerates can still be achieved. In [59], a scene with a single room achieves a framerate of 60 frames per second on an Nvidia 6800GT GPU.

Figure 4.12: An image from [59]. The caustics are generated using the caustics triangles technique.

4.3 Displacement mapping

Whereas environment maps and caustics are used to render a lighting effects, *displacement mapping* is a technique of a totally different type: it adds geometrical detail to an otherwise flat polygon, but without adding any extra triangles. Instead, a depth texture or displacement map is used. This is a texture in which the brightness of the texels corresponds to the height of the surface. So if the surface that is being rendered is a landscape, then white in the texture corresponds to the tops of hills, while black corresponds to valleys.

Using displacement mapping has several benefits. The first of these, is that highly detailed surfaces can be rendered using only a single polygon. This saves performance, especially if the surface is occluded by another object: the number of triangles that needs to be transformed and rasterised remains low. Another benefit is that a small, repeating texture can be used. This greatly reduces the storage required for objects with repeating patterns, like a brick wall.

To render an object with displacement mapping, a pixel shader is used. This pixel shader applies ray-marching to find the intersection of the ray from the camera with the displaced surface [44]. To make this work, the displacement mapping can only be applied inwards. When a pixel on the polygon is being rendered, the pixel shader constructs the ray from the camera to the pixel. This ray is then used to step through the depth texture. At each pixel, the height of the surface at that point is read from the depth texture and this is intersected with the ray. If the ray hits the surface, this point is used to render the object. Subsequently, the diffuse texture is used to get the colour at that point. To calculate lighting, the normal of the displaced surface is needed. This could be calculated by sampling the heights around the intersection, but it is faster to just store the normals per pixel in a

normal map [22], although this does cost extra performance. An example of displacement mapping can be seen in figure 4.13.

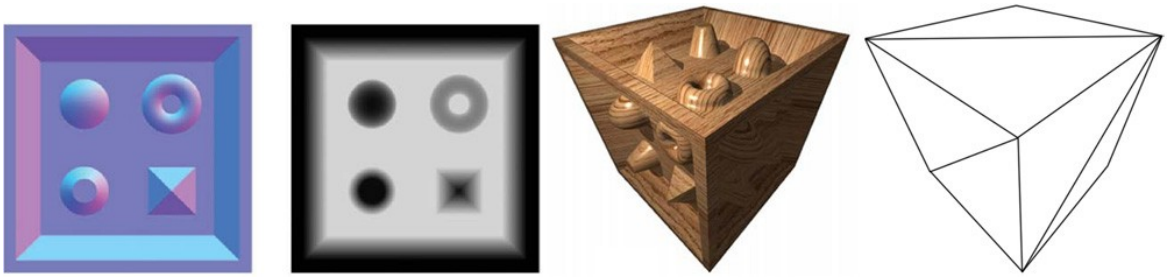


Figure 4.13. An example of displacement mapping from [17]. From left to right: the normal map, the depth texture, a cube rendered with displacement mapping, and finally the wireframe of the same object.

The downside of displacement mapping is that, if it has to take a lot of steps through the depth texture before the intersection between the ray and the displaced surface is found, the performance will greatly decrease. The simplest solution to this, is to take large steps through the depth texture and only start doing finer steps to find the exact point of the intersection when an intersection has been found. However, this adds the risk of missing intersections altogether: small details in the displacement map, such as thin spikes, might be skipped over altogether and thus missed, resulting in incorrect results.

A solution to this is to use *cone step mapping* [17]. Here, an extra value is stored along with the height in the depth texture: the angle of the largest cone that could be placed on that texel, such that the cone does not intersect with the displaced surface. Now, when performing the ray march in the pixel shader, for each step the cone is looked up and the ray is immediately stepped to the boundary of the cone. There the depth texture is checked again to see whether the ray hits the surface. If it does not, then the cone at that point is used to perform another step. This way, large steps can be taken through the depth texture, without missing any details. The benefits of cone step mapping are clear: if there is a lot of empty space, then the ray marching algorithm can perform a powerful empty space skipping algorithm at the cost of a relatively small amount of extra memory, thus greatly increasing the performance of the displacement mapping. To increase the size of the steps that can be taken, a variation called *relaxed cone step mapping* [48] can be used. Here the cones can sometimes intersect the displaced surface, but in such a way that no rendering errors are introduced. This results in broader cones, allowing the ray to perform larger steps over the displacement map.

To use displacement mapping, a depth texture is needed, as well as, depending on which algorithm is being used, a normal map and a cone step map or a relaxed cone step map. The depth texture is usually obtained in one of two ways. One can draw it by hand in a 2D drawing program. This is simple and fast, but it is difficult for the user to really understand how the brightness he draws corresponds to a displaced surface in 3D. Therefore, obtaining a good result is hard and requires an

experienced artist, especially when more complex shapes are required. The normal map and relaxed cone step map can automatically be generated from the depth texture.

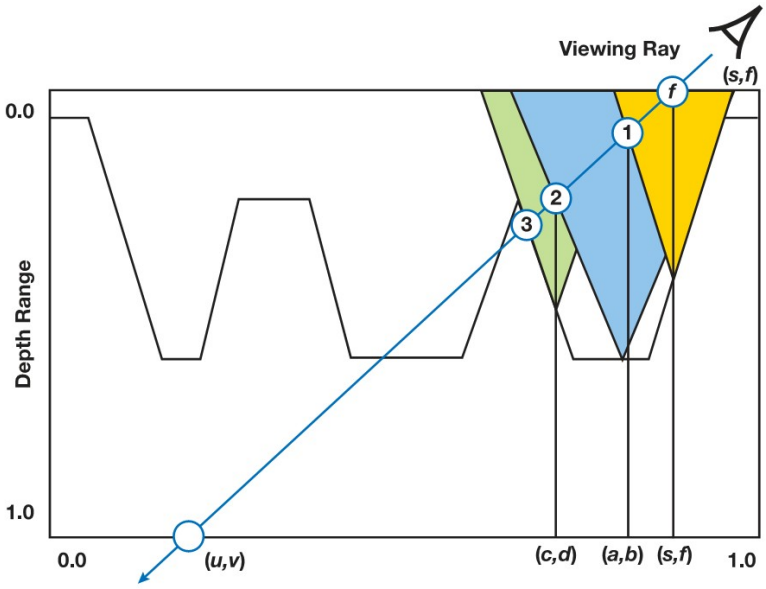


Figure 4.14. An example from [48] of how cone step mapping works. f is the pixel being rendered. The yellow cone at f is used to immediately step to 1. The next cone allows a step to 2 and then to 3, where the ray hits the surface.

Alternatively, the artist can work directly with a displaced surface in a 3D modelling tool. In this tool, the artist can make the surface using any technique available in the tool and using as many polygons as desired. After the surface is made, an algorithm is run that calculates the depth of the surface and stores this in a depth texture. Also, the normals for the normal map can be obtained directly from the 3D surface, which usually results in better normal maps. The benefit of this technique is that the artist has more control over the result and works with the surface directly, instead of manipulating brightness to manipulate a surface. The downside is that it usually takes more time to create a 3D surface like this.

So far, a depth texture has been used to define the displaced surface. Because of this, the surface can only be a height field: an overhanging piece rock and a cave in a landscape are both not possible. Alternatively, a voxel grid can be used instead of a depth texture. In this grid, each voxel defines whether that point is empty or solid. The grid is wrapped over the polygons, so it is only a thin shell around the surface. The ray is marched through the voxel grid to find the intersection with the first solid voxel. To still be able to do empty space skipping, the cone step map can be replaced with a *sphere step map* [31]: a map that stores the size of a sphere for each point in the voxel grid, so that this sphere is the largest sphere that can be placed at that point without having any intersections with opaque voxels.

Displacement mapping requires a ray march inside the pixel shader, which is a very expensive operation. Also, the number of steps needed to find the surface depends on the complexity of the depth texture and the viewing angle. Especially cone step mapping is very dependent on the actual contents of the depth texture for its performance. For very complex surfaces, a lot of steps might be required. This either decreases performance, or introduces errors if the algorithm is stopped after a maximum number of steps. Using *generalised displacement mapping* [62], this problem can be solved: with this technique, ray marching is no longer necessary and a constant time algorithm is used instead. Also, the surfaces rendered do not have to be height fields and can be of any complexity, as can be seen in figure 4.15. To achieve this, a 5D texture is created. For each point in the space where the generalised displacement map is applied, this stores for every direction the distance to a surface. When rendering a pixel, the position of the pixel and the direction of the ray from the camera to the pixel are used to do a look-up into this map, to find the place where the ray hits the surface. This requires only a single look-up. The map is 5D, because direction is 2D and position is 3D, so storing the distance for every point in every direction is 5D. However, storing a 5D dataset requires an enormous amount of space, and graphics hardware only supports up to 3D textures. To solve this problem, the 5D dataset is compressed into a number of 2D and 3D textures, using singular value decomposition. These can be used to reconstruct a given value in the 5D map in the pixel shader. However, depending on the strength of the compression, high frequency details might be lost.

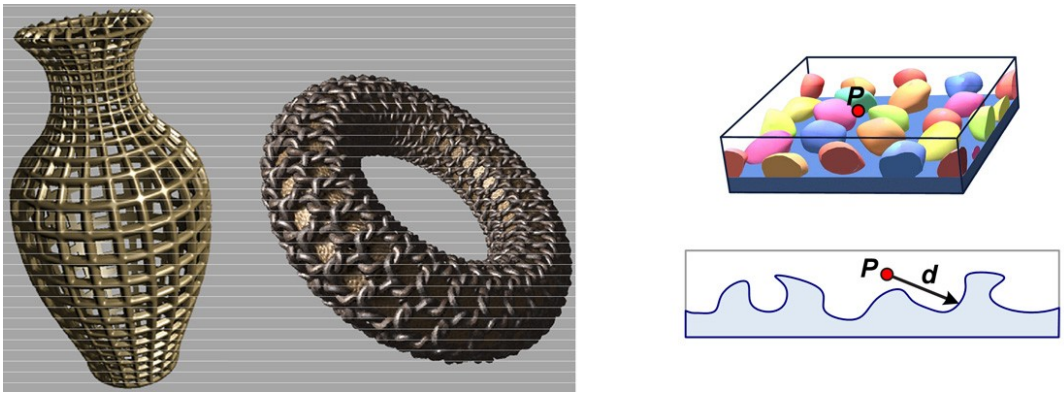


Figure 4.15. To the left an example from [62] of a result of generalised displacement mapping. Note how non-height field and even partly transparent surfaces are possible with this technique. To the right an example of the kind of data that can be stored in the 5D dataset.

So far, generalised displacement mapping seems to be a very useful idea. However, it has a number of downsides that make it infeasible in practice. The most important one, is that the textures that together store the compressed version of the 5D map still take a lot of space. With the compression strength used in [62], an area with a resolution of 128x128x16 already uses 4mb of texture space. For such a low resolution, that is too much to be feasible in practice. Also, the decompression algorithm requires a lot of processing, so the actual performance of generalised displacement mapping is low.

4.4 Real-time screen-space ambient occlusion

To the human eye, the most noticeable shadow effect is sharp shadows, cast directly from a light source. However, to achieve visual realism, more subtle lighting effects should be rendered as well. For example, the light reflected by the environment around an object, contributes to the appearance of that object. A brightly lit red surface will reflect light and thus cast a subtle red glow on all nearby objects. Similarly, since part of the light coming from the sun is refracted by the atmosphere, the entire sky can be considered a large source of light. This kind of lighting is called *global illumination* and has already been discussed in section 3.1 of this thesis. Not rendering global illumination results in an obviously fake image, even though many viewers might not be able to identify what is wrong. Therefore, it is necessary to somehow represent this environmental lighting. Such lighting effects are subtle, but required to achieve visual realism.

In traditional raycasting renderers, lighting from the environment on a specific spot is rendered by shooting lots of rays into the scene and calculating how much light is contributed from the directions of these rays. This solution is extremely costly, easily resulting in rendering times of minutes or even hours per frame. Also, shooting rays into the scene is not feasible in a pixel shader. To simplify the problem, the lighting from the environment is often represented by *ambient occlusion*: the actual amount of light coming from different directions is ignored and only how much of the environment is occluded by nearby objects is taken into account. If there is little occlusion, then the environment is considered to contribute more light to the surface. Although this might not be a completely correct solution, the subtle lighting caused by ambient occlusion already greatly increases the realism and visual quality of an image. An example of ambient occlusion can be seen in figure 4.16.

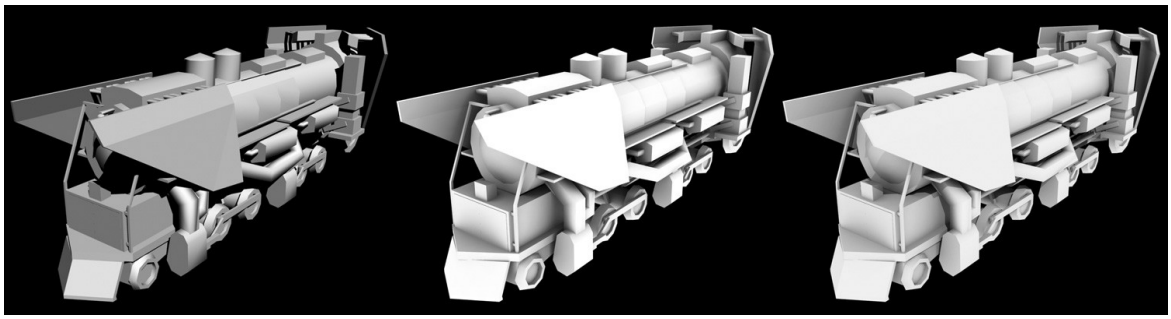


Figure 4.16. To the left this model is rendered using a single light that casts shadow. In the centre is the same model, but rendered using raycasted ambient occlusion. Note the more subtle shading. Ambient occlusion and a standard light are often combined. An example of this is shown to the right.

The standard solution to calculating ambient occlusion is the same as for calculating lighting for the environment in general: shoot lots of rays into the scene and see what they hit. As doing this is not possible in real-time, one solution that is often used is to calculate the ambient occlusion in a pre-

processing step and statically store this in a texture. When the image is rendered in real-time, the ambient occlusion is simply looked up in the texture. This way, the pre-processing can take as long as required, while the game itself can still achieve real-time framerates. This technique has two downsides, though: storing the textures requires extra storage, and the lighting is static. If the scene changes, then the ambient occlusion remains the same.

Screen space ambient occlusion is an approach to calculating ambient occlusion on dynamic scenes in real-time. It has been made popular by the game development company Crytek, who used it in their game Crysis (2007). Although it was only presented very briefly in [38], the idea is clear and simple and works as follows:

- First the scene is rendered to a depth texture.
- When the scene is being rendered to the screen, the pixel shader of each object uses the depth texture to calculate ambient occlusion. It takes a number of samples from the depth texture in the area around the pixel.
- A sample is only considered to be occluding the surface if the sample lies in front of the surface. Thus the number of samples that are occluding is a measure for the amount of ambient occlusion.

This technique is simple to implement and is feasible in real-time, although taking a lot of samples strongly decreases the framerate. The main problem, however, is that ambient occlusion is only calculated in screen-space, so any occluding objects that are not visible from the point of view of the camera are ignored. Also, it requires a lot of samples to get a smooth result. Instead, the ambient occlusion can afterwards be blurred in screen-space, where the depth texture is used to keep the blur from spreading over object edges. Interesting in this technique, is that it approximates shooting rays in random directions by simply taking samples in screen space. The result is that the actual ray directions follow from where the samples that are taken are positioned in the world.

Figure 4.17. This image from [38] shows the effect of screen space ambient occlusion. To get the final image, this result would be combined with other light sources and texturing.

In terms of performance, the number of samples taken per pixel is the main limiting factor of the screen space ambient occlusion presented in [38]. To increase the performance, [13] proposes a different sampling method: *vicinity occlusion maps*, of which an example can be seen in figure 4.18. Although this technique does not use any form of raycasting at all and is therefore not part of the topic of this thesis, this technique is explained here to be able to compare it to [38].

When using vicinity occlusion maps, instead of counting the number of occluding samples, the average depth of the samples around a pixel is calculated. If the average depth around a pixel is larger than the depth at that pixel, this is considered to mean that there are few objects in front of the object and thus few occluders. The benefit of calculating averages instead of counting samples based on whether they are in front of the object, is that averages can be calculated using a 2D convolution. The result is that a small number of vertical samples in the first pass and a small number of horizontal samples in the second pass is enough to calculate the average of a much larger number of samples. This way, a smooth result can be achieved while maintaining a high framerate. The downside of averaging is that objects that are at a great distance from the surface for which the occlusion is required, strongly influence the average. This happens even though an object that is far away is probably not very relevant for the occlusion. In [38], such samples could be discarded or considered not to be occluding, while with vicinity occlusion maps, such samples become very dominant. In [13] this is not a great problem, though: it only calculates ambient occlusion for a single character without any environment, so objects that are very far from the surface never occur.

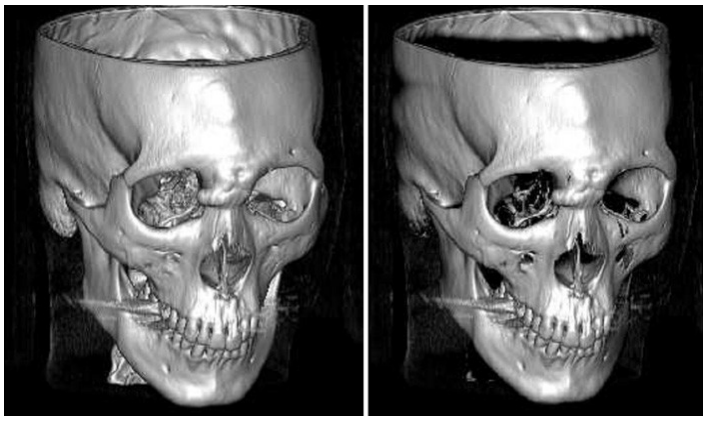


Figure 4.18: Two renders from [13] show the same model, rendered to the left without and to the right with Vicinity Occlusion Maps.

In comparison, vicinity occlusion maps result in more fluent ambient occlusion than [38], and does so while using less samples. However, because of the dominance of far away samples in vicinity occlusion maps, and because the normal of the surface cannot be taken into account with vicinity occlusion maps, the results of [38] are more correct. Both techniques suffer from a lack of information about objects that are not visible from the point of view of the camera.

4.5 Light shafts

When the air contains a lot of dust particles or aerosols, some of the light that passes through it is scattered. This results in *light shafts*, also known as *Jacob's ladders*, *God rays* or *volumetric light*. An example of this effect can be seen in figure 4.19. Although light shafts are not very common in everyday life, they are often considered to be rather spectacular effects, making it interesting to look for a way to render these effects in real-time virtual worlds.

Figure 4.19. This photograph shows an example of light shafts.

The standard raycasting approach to rendering light shafts is simple, effective and incredibly slow. When a ray is shot from the camera into the scene, sample points are taken along this ray. For each of these sample points, a ray is shot towards the light source that causes the light shafts. If this ray is not occluded, then this sample point adds a little bit of light for the light shaft, otherwise it is ignored. This process is further shown in figure 4.20.

To achieve smooth light shafts this way, large numbers of samples must be taken. Since each sample requires a ray to be shot towards the light, the number of rays that need to be cast for this technique is very high and thus it is too slow to be performed in real-time. Also, doing full raycasting on the GPU is very difficult and very slow, so this technique is not applicable to real-time graphics.

In [37], an alternative is presented that can be performed in real-time and might not be correct, but looks convincing and smooth and achieves real-time framerates. The method used here is to sample over a ray from the pixel being rendered to the light source, and to do so in screen space. This is done through the following steps:

- The scene is first rendered with the light source as a bright area and the occluding objects in front of the light source in black. An example of this can be seen in figure 4.21a.
- After this, the light shafts are rendered. Figure 4.21b shows how a single pixel is rendered. In view space, the colours of the samples from the pixel to the light source are accumulated.

The resulting light shafts can be seen in figure 4.21c.

- Finally, the rest of the scene is rendered normally and the light shafts are added to this, resulting in the image that can be seen in figure 4.21d.

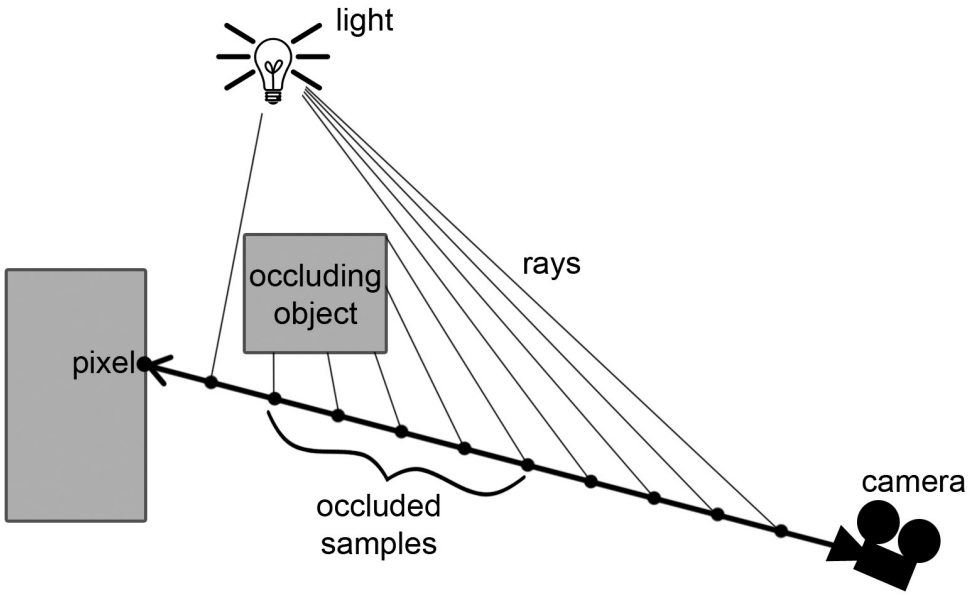


Figure 4.20. To render light shafts using raycasting, the method shown in this image is used. When a pixel is being rendered, a number of samples is taken over the ray from the camera to the pixel. For each of these samples, a ray is cast towards the light to see whether there is an occluding object between the sample and the light. The sample attributes to the volume light only if the ray is not occluded. In the example given here, five of the ten samples are not occluded.

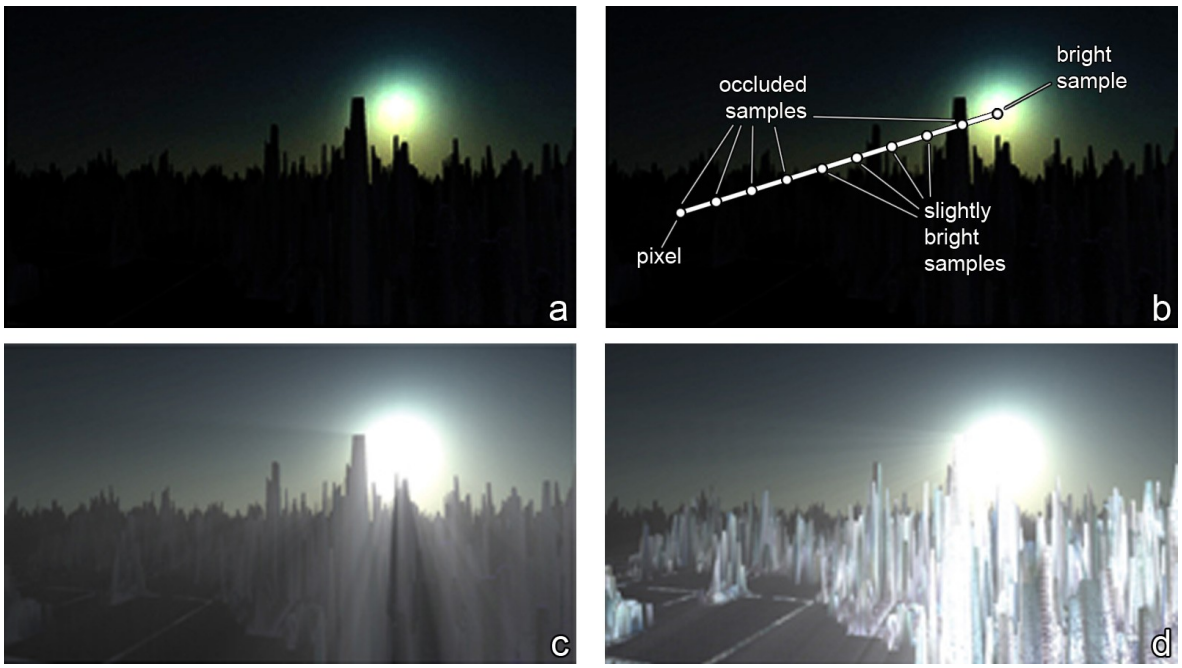


Figure 4.21. This figure shows the process of rendering real-time light shafts. These images are a modified version of images from [37].

Whereas in the original raycasting technique a new ray was shot for many samples for the pixel, here only a single ray has been used. This method is in no way correct and cannot easily render, for example, light shafts that are caused by a light that is behind the viewer. In the real world, such light shafts might exist and the full raycasting technique is capable of rendering such effects. However, the approximation of [37] looks convincing and is fast enough for actual usage in real-time and is therefore very useful in practice. What is unique in this technique in comparison to the other techniques that are discussed in this thesis, is that this is the only one that casts rays in screen space.

4.6 Nvidia eye-demo

The final technique to be discussed here is is Nvidia's “Ray Traced Eye" demo [31]. It uses a very different approach from the other uses of raycasting in this chapter, one which is much more similar to the techniques discussed in chapters 5 and 6 of this thesis. So even though the application of this technique is very limited, it is interesting to discuss it here.

The Ray Traced Eye technique renders an eye while taking into account the refraction of light by the lens, which distorts how the iris is seen. As can be seen in the image below, this is an effect that is only visible when zoomed in very closely to the eye, which rarely happens in imagery and is therefore only remotely useful.

This technique uses a smooth polygonal sphere to render the eye. In the pixel shader, it is checked whether the ray from the camera to the pixel hits the eye's lens. If it does, then the ray is refracted. The refracted ray is then intersected with a sphere for the iris, as can be seen in figure 4.22. This second sphere does not actually exist in geometry: it is only used in the pixel shader. The intersection with the iris is used to calculate lighting and perform a texture look-up to calculate the colour of the iris.

To achieve a convincing rendering of an eye, a number of smoothing, texturing and lighting techniques are used in [31] as well. However, as these techniques are not related to the topic of this thesis, they are not discussed here.

A benefit of this technique is that, because the sphere for the iris is not actually stored, using this technique does not increase memory usage. Also, because the effect is calculated per pixel, its performance cost is mainly dependent on the size of the eye on the screen. Therefore, if many characters are on the screen, this technique can still be used, as the eyes of each character will be small on the screen and thus take up only a small number of pixels.

What makes this technique specifically interesting, is that it uses a sphere primitive to render the iris. In general, triangles are most efficiently rendered using rasterisation, while spheres are a type of primitive that is very easily rendered by raycasting, but cannot be rasterised. Here these different

primitives are rendered using different techniques to combine them in a single render, which is an approach that has not been seen in any of the other techniques discussed in this chapter.

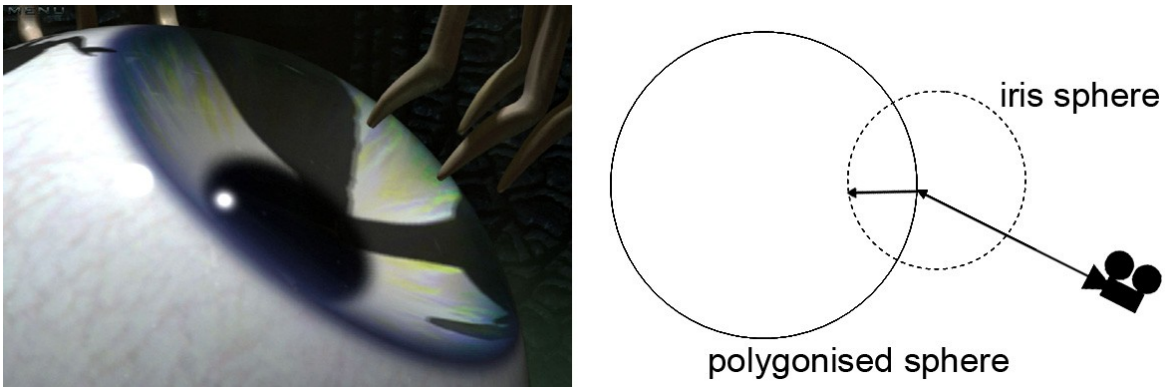


Figure 4.22. To the left the result of Nvidia Raytraced Eye Demo [31]. To the right a scheme of how it works: when the polygonised sphere is rendered, the pixel shader constructs a ray from the camera to the eye's surface. If the ray hits the lens, it is refracted into the eye and intersected with the sphere for the iris.

5. Interior Mapping:
A new technique for rendering realistic buildings

In the previous chapter, a number of existing techniques that combine raycasting and rasterisation have been discussed. Here, a new one is introduced: Interior Mapping. Interior Mapping is a novel real-time shader technique that renders the interior of a building when looking at it from the outside, without the need to actually model or store this interior. With Interior Mapping, raycasting in the pixel shader is used to calculate the positions of floors and walls behind the windows. Buildings are modelled in the same way as without Interior Mapping and are rendered on the GPU. The number of rooms rendered does not influence the framerate or memory usage. The rooms are lit and textured and can have furniture and animated characters. The interiors require very little additional asset creation and no extra memory. Interior Mapping is especially useful for adding more depth and detail to buildings in games and other applications that are situated in large virtual cities. An example of what can be achieved with Interior Mapping can be seen in figure 5.1.

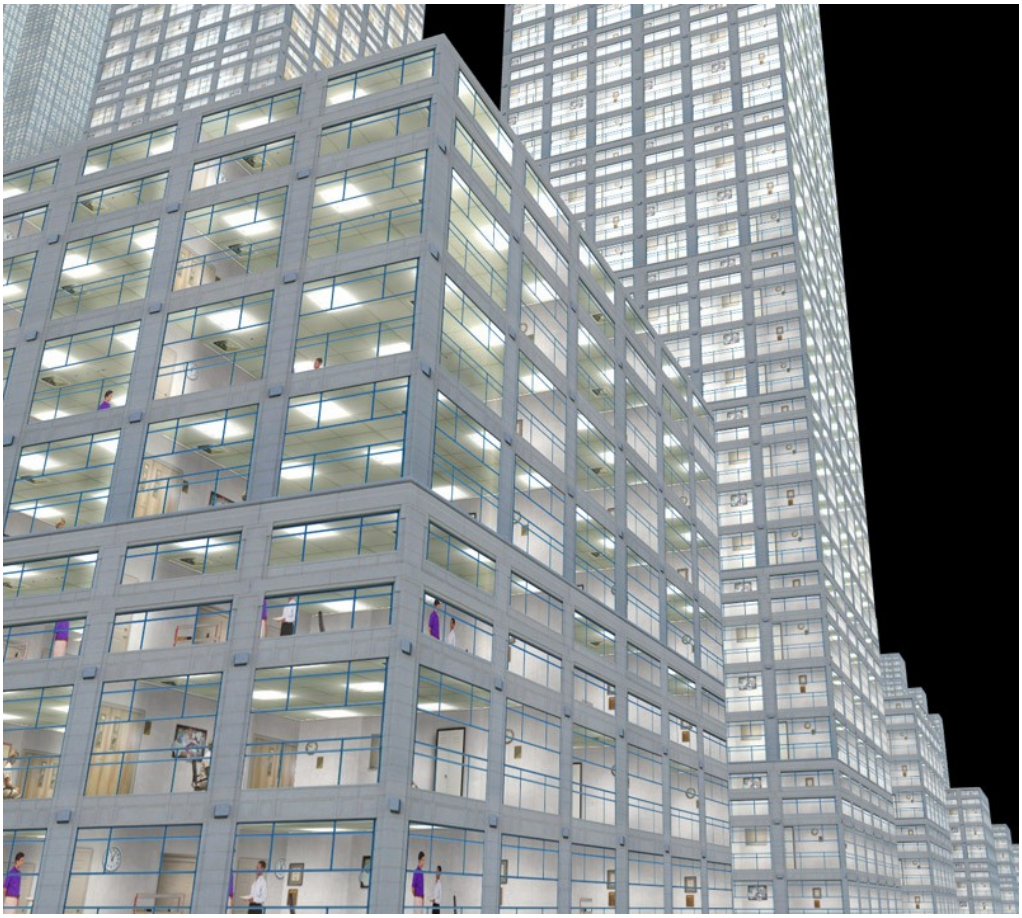


Figure 5.1. This image shows buildings with their interiors rendered with Interior Mapping. The geometry of the buildings consists of simple cubes only. Note how the interiors are textured and drawn perspectively correct.

5.1 Introduction

Many new games and virtual worlds feature large cities through which the player can move. Take for example games like the Grand Theft Auto-series by Rockstar Games, Crackdown (2007) by Realtime Worlds, and Superman Returns: The Video Game (2006) by Electronic Arts. Other examples can be seen in the use of virtual 3D cities in online communities, simulations and virtual tourism. For a significant part, the graphical quality of these games is determined by how the buildings are rendered. These can have a certain amount of polygonal detail, depending on the desired size of the city and the dynamic loading system used, but the aforementioned applications all have in common that the cities are too large to model detailed buildings using geometry and still achieve real-time framerates while doing so. An example of this can be seen in figure 5.2. To deal with this, this chapter introduces a new technique that significantly increases the graphical quality of buildings in real-time applications.

Figure 5.2. This screenshot from Superman Returns: The Video Game shows how simple the geometry of the buildings often is in games that play in cities.

Through the years, many techniques have been developed and used to visualise buildings as realistically and detailed as possible while still keeping framerates real-time. Geometry is often kept relatively simple, while details are added through the use of textures [9]. Originally, textures were only used to change the colours of surface elements. When adding details, this tends to look as if pictures of details have been glued on to a flat surface, instead of as if the details are really there. An example of using textures to add detail to buildings can be seen in figure 5.3. One technique that deals with the shading of details is bump mapping [4]. Unlike in the case of standard textures, with bump mapping, details are correctly lit with respect to the lighting in the scene. This can be extended through horizon mapping [34], to show the shadows of these details. In addition to bump mapping, normal mapping [22] is also often used, which stores the normals of each texture element, instead of their height. The effect of normal mapping can be seen when comparing figure 5.3 to figure 5.2: the windows in figure 5.2 have normal maps and thus seem to have a lot more depth than those in figure 5.3. All the aforementioned techniques have in common that they do not correctly

visualise the parallax effect when perspective changes. Displacement mapping, which was already discussed in section 4.3, fixes this by rendering height-field surface details perspectively correct.

Figure 5.3. This screenshot from Grand Theft Auto: Vice City (2002) shows how windows and curtains can be created by applying textures to otherwise flat walls.

The texturing techniques mentioned above can be used to add details to buildings. There are also a number of techniques that have been developed specifically to quickly render more distant buildings and objects, as they are smaller in view and can thus be simplified. Displaying simpler geometry through level of detail [11] and lower resolution textures through mipmaps [64] are commonly used techniques for this. More rigorous is replacing distant objects by imposters [33] or turning complete groups of buildings into single blocks with height field shapes inside, which are rendered through the use of an extension to displacement mapping called block maps [10]. This way, hardly any polygons are needed, while relatively complex shapes can still be rendered with reduced detail.

None of the techniques above makes it possible to render the interiors of buildings. However, interiors are important for the graphical quality of a virtual city, as its buildings contain many windows through which interiors should be visible. But as interiors are often not required for gameplay, they are usually left out entirely, except in those instances where the interior can be entered by the player (see for example Grand Theft Auto: Vice City). Adding the interiors of many buildings would require an enormous amount of polygons, which is not feasible when real-time framerates are required. Therefore, an other solution is required to render windows. Two simple techniques are used in most games and interactive applications today: either windows are fully reflective, not allowing a look inside (as can be seen in figure 5.2), or some details very close to the window are drawn into the diffuse texture, while the rest of the interior is left black (as can be seen in figure 5.3).

One way to add perspectively correct interiors would be through the use of displacement mapping. If the displacement map is calculated separately from the exterior texture, it could handle geometrically correct rooms. However, because displacement maps do not support texturing on surfaces that are perpendicular to the original polygonal surface, this would leave all walls with stretched coloured lines, except for the back wall, which could still have a real texture.

To fix this, textures can be added to all the walls of the interior through the use of indices from the displacement map into an extra texture, as is done with blockmaps [10]. An example of rendering buildings (not interiors) with this technique can be seen in figure 5.4. Blockmaps cannot render furniture or characters inside the room, though, because they are limited to height field geometry. Also, if the viewer watches a room at the corner of a building, she would see a different room through the windows on the different walls. This would be awkward, as it should in fact be the same room, only seen through different windows. Finally, to vary the textures on the walls per room, knowledge of which room is being rendered would be required and this is not readily available with block maps.

Figure 5.4. This image from [10] shows how geometry can be rendered through the use of blockmaps. The buildings are not in actual geometry and are rendered through an algorithm similar to displacement mapping, but with the addition of textures for the walls, which are perpendicular to the height map and thus would not be textured when using standard displacement mapping.

Another technique that could render interiors, is generalised displacement mapping, which was already discussed in more detail in section 4.3. This technique could render interiors of any complexity, but has a number of drawbacks that make it ultimately unsuitable. The most important problem is that generalised displacement maps use too much memory. After heavy compression, a map with a resolution of 128x128 pixels still requires 4mb of memory, while a standard texture of the same dimension, as can be used with Interior Mapping, requires only 48kb of memory, or 8kb if DXT1 texture compression [6] is used. Also, with generalized displacement mapping it is not possible to separately animate a single character or object in the interior and walls cannot be varied independently of each other.

Here, Interior Mapping is introduced to deal with the problem of rendering the interiors of buildings. This new technique solves the aforementioned issues with block maps and generalised displacement maps by directly rendering virtual walls through raycasting. The Interior Mapping

algorithm knows which wall in which room it is rendering and can use this information to vary lighting and textures per room. Furniture and characters can be added through the use of furniture planes.

An added benefit of Interior Mapping is that it can easily be coupled with procedurally generated buildings, such as those in [39] and [65]. Most procedural city and building generation systems do not take interiors into account. Interior Mapping could be added to the buildings that these systems generate, and even works well with buildings that have curved walls.

Although Interior Mapping is a technique that renders actual interiors, it does not need to model or store them in geometry. The walls of the rooms of the interior only exist as virtual geometry in the shader. For each pixel, the ray from the camera to the point on the building that is being rendered is intersected with the walls of the interior. Because the interior walls are regularly spaced, the ray can be collided in constant time, regardless of the number of rooms in a building or the number of buildings. Because it is shader-based, it is easy to turn Interior Mapping off for level of detail or lower quality rendering on older GPUs. Models do not need to be changed to make Interior Mapping possible, so the effect can quickly be added to virtual cities that have already been created or are near completion.

Because the raycasting is done for each individual pixel that is rendered and takes the camera direction into account, the result is that perspectively correct rooms are rendered inside the building when looking at it from the outside. This can then be combined with a texture that stores where windows are, so that the interiors are only visible through the windows. By blending the interior's colour with a reflection map, both the interior and the reflections in the windows can be shown. See figure 5.1 for an example of what can be achieved using Interior Mapping. Note that the buildings here are modelled as single blocks with no additional geometry.

Interior Mapping uses raycasting on the GPU, a research field that has quickly evolved since the introduction of programmable shaders. Here, a major trend is to turn the GPU into a full raytrace renderer [7], which means that the rasterising functionality of the GPU is worked around to create a renderer that utilises the vector processing power of the GPU, but does not use the GPU's approach to rendering. This was already discussed in more depth in section 3.2.2. Contrary to this approach however, Interior Mapping does not seek to *replace* rasterised GPU rendering, but actually to *enhance* it. Thus, it is more closely connected to techniques such as displacement mapping, and papers on topics like ambient occlusion [47]. Nonetheless, using the regularity of internal structures in the way that Interior Mapping does, does seem to be a novelty among existing GPU raycasting techniques.

Interior Mapping can be implemented in such a way, that the number of shader instructions is small enough to be able to calculate the effect within the 64 instructions allowed in pixel shader model 2.0 [29], and also add a diffuse and reflection map in the same pass. More effects can be added to the basic algorithm to create additional details.

The remainder of this chapter is structured as follows: section 2 explains how Interior Mapping works. Section 3 analyses the performance of Interior Mapping and section 4 looks into a number of extensions to the basic technique.

5.2 The algorithm

For ease of understanding, the simplest case is considered first: only ceilings and floors are being rendered. Walls, lighting and exterior textures will be added later.

5.2.1 Ceilings

For the Interior Mapping algorithm, the 3D space is considered to have ceilings at regular distances. Each ceiling is an infinite plane parallel to the XZ-plane. Intersecting a ray with a horizontal plane is a very simple thing to do, but the main point to Interior Mapping is to calculate efficiently which plane to use for a particular pixel.

When the pixel shader renders a pixel of a polygon to the screen, this pixel also has a position in the 3D world. This position is calculated in object space and the position of the camera is transformed into object space as well. When handling a ray from the camera to this pixel with a camera that is tilted upwards, the ceiling just above the pixel needs to be found. This can be done by taking the ceiling-function of the y-position of the pixel, divided by the distance d between ceilings. Afterwards, the height that has been found should be multiplied with d again to get the actual height of the ceiling. In Cg, this simply comes down to `ceil(y / d) · d`. When the camera is tilted downwards, the ceiling (or actually the floor) that is one position lower must be used, so in that case the height is at `(ceil(y / d) - 1) · d`.

Now that the height of the ceiling is known, the ray from the camera to the pixel can be intersected with the ceiling to find the position where the ray hits the ceiling. This is visualised in figure 5.5.

The result is the position of the intersection, but what is needed is a colour for the pixel. This can easily be obtained by using the x- and z-coordinates of the intersection as the uv-coordinates for a texture read for the ceiling or the floor. To scale the texture to the desired size, the coordinates can be multiplied by some constant.

By performing all the calculations in object space, the walls can be rotated by rotating the space of the object itself. This way the walls in different buildings do not have to be parallel.

Note that an exactly horizontal ray will result in a division by zero. However, GPUs do not crash on this and the visual artefacts this results in are so rare in practice that this problem is negligible.

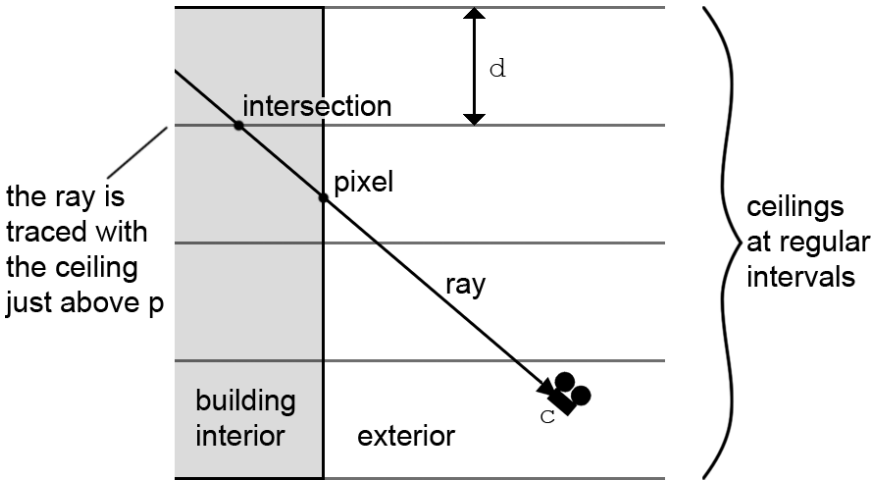


Figure 5.5. Visualisation in 2D of the variables in the formula above.

5.2.2 Walls

So far only horizontal planes have been considered. However, real interiors do not only have floors and ceilings, but also walls. These can be added by using exactly the same calculations as were used for the ceilings, but now with XY- and YZ-planes, instead of XZ-planes. The intersection of the ray is thus calculated with three different planes.

Of the resulting three intersections, the one closest to the camera will be used. To be able to use different textures for the ceilings, floors and walls, the texture corresponding to the closest intersecting plane is used. This allows for different textures to be used for walls and ceilings, as can be seen in figure 5.6. The interior thus calculated works with curved geometry as well, as is exemplified in figure 5.7.

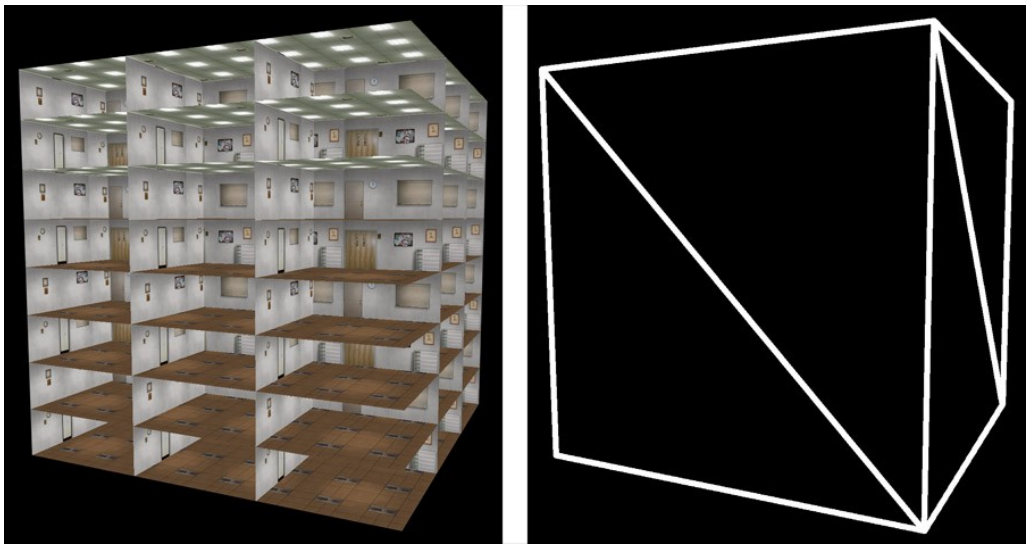


Figure 5.6. The result of calculating ceilings, floors and walls with Interior Mapping. The geometry of this building consists of a single cube, as is shown in the wireframe image of the same building to the right.

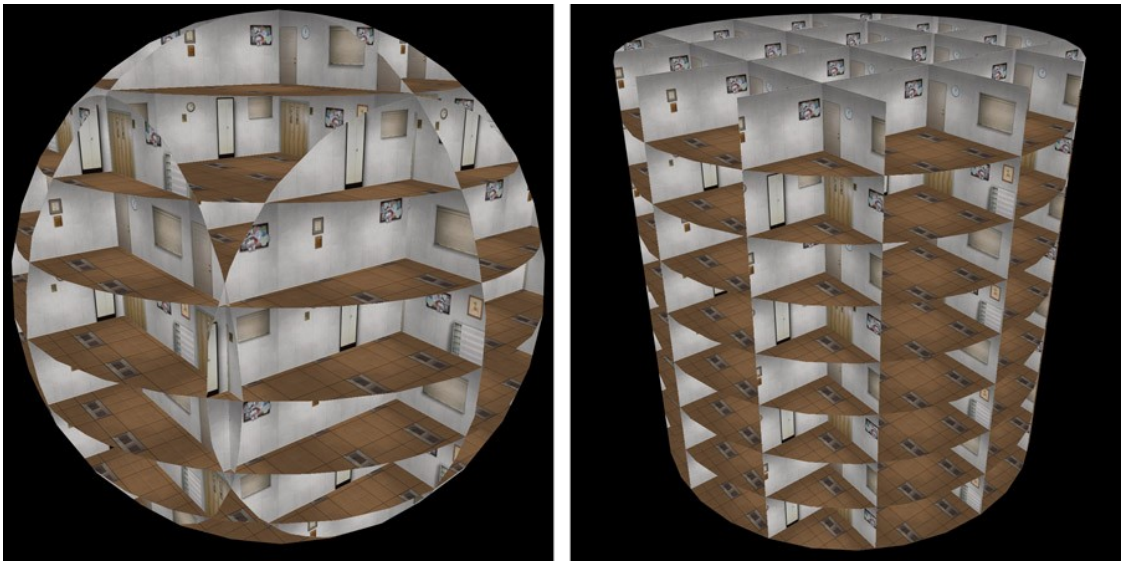


Figure 5.7. Interior Mapping applied to a sphere and a cylinder.

5.2.3 Combining with exterior textures

The basic Interior Mapping algorithm is now complete, but usually, it will be combined with a texture for the exterior of the building. This can be done by creating a diffuse texture that uses the alpha channel to store where windows are. If the alpha-value is one, then the diffuse texture will be used; if it is zero, then the colour calculated by the Interior Mapping algorithm will be used. This can be further refined by adding a reflection to the windows. The colour of the reflection will then be combined with the colour from the Interior Mapping, as can be seen in figure 5.8.

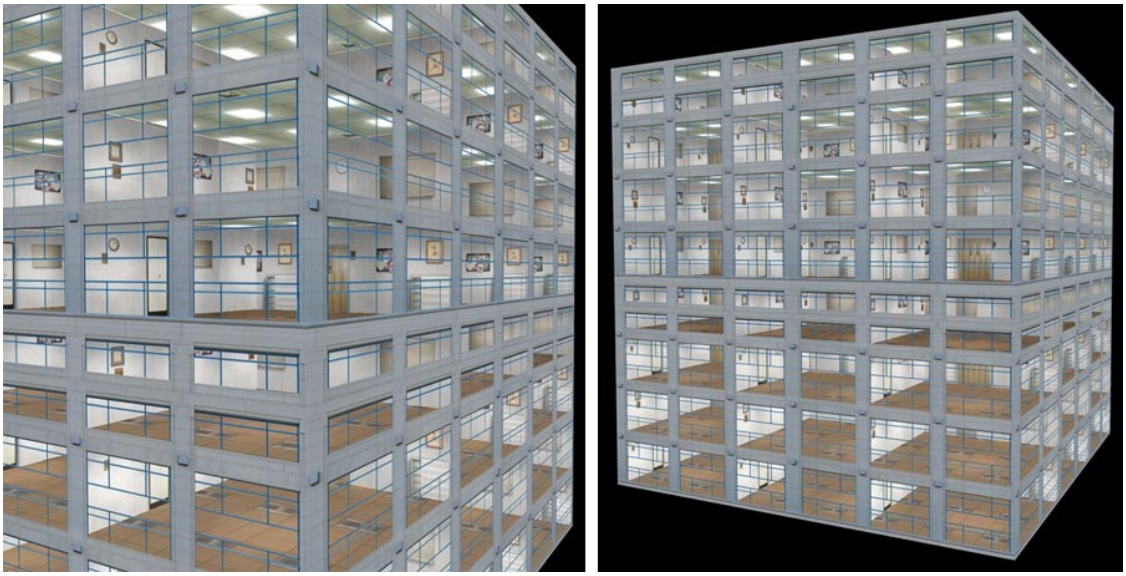


Figure 5.8. Interior Mapping combined with an exterior texture and a reflection map. The reflection is made very subtle here to emphasise the effect of the Interior Mapping.

5.2.4 Furniture and animated characters

So far, the rooms in the interior have been completely empty. This works well when looking from a larger distance, but adding furniture and even animated characters to the interior would greatly improve their liveliness. Adding actual geometry to the interior is not possible without raycasting against large numbers of objects, so a different solution is needed here.

A technique that can add details inside the rooms is to add an extra plane that is parallel to the surface of the building, but displaced a fixed distance to the inside. Here, this plane is called the *furniture plane*. The furniture plane does not actually exist in the geometry and is defined in the pixel shader. It is intersected with the ray from the camera to the pixel and if the intersection is closer than any of the intersections of the interior walls, then the furniture is shown. How this works is illustrated in figure 5.9.

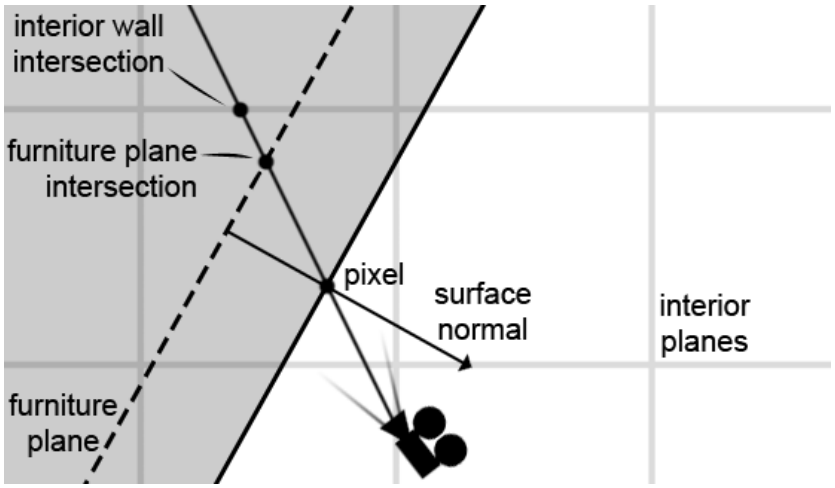


Figure 5.9. The furniture plane runs parallel to the actual surface of the object. The ray from the camera is intersected with both the furniture plane and the interior planes. In this example, the intersection of the ray with the furniture plane is closer than the intersection of the ray with the interior wall, so the furniture plane is visible.

The furniture plane is intended to show things like furniture and characters, not to be a solid wall inside the building. Therefore, the alpha channel of the texture that is used for the furniture plane determines whether the colour of the furniture plane is discarded or not. This way, an object can be seen standing in the middle of a room, as if it were a billboard or sprite inside the building, as can be seen in figure 5.10.

By using an animated texture, the objects on the furniture plane can actually move. Animated textures usually require a lot of storage, so only short animations can be done this way. An example of this is a character reading a newspaper in a chair, turning over the pages once in a while. Through the use of render to texture [25], more complex animations could be shown on the furniture plane.

This would require separately rendering an animated 3D character to a texture each frame and using this texture for the furniture plane.

Figure 5.10. An example of characters inside a room, made with a furniture plane.

Finally, note that, unlike the interior planes, the furniture plane is not geometrically correct. If the building's surface is curved or has a corner, then the furniture plane will show distortions and even seams. See figure 5.11 for an example of this problem. The stronger the curvature and the further into the interior the furniture plane is, the stronger the distortion will be. For this reason, it is not advisable to add furniture planes to highly curved objects. Around the corners of buildings the solution is much simpler: avoid transparency in the exterior texture exactly at the corner. This also makes sense from an architectural point of view: most buildings do not have windows that span the building's corners anyway.

Figure 5.11. In the left image, there is a seam at the corner of the building. This shows in that the centre character is cut-off. In the right image, this seam is hidden by a piece of exterior wall at the corner of the building, so that the seam can never be seen. Note that this seam appears here because the various sides of the building are oriented differently and thus have different furniture planes.

5.3 Experiments

To evaluate the effect of Interior Mapping for use in games and other real-time graphics applications, a test application has been written. This has been done in C++ using the Ogre 3D engine (`http://www.ogre3d.org`). All the images in this paper were produced using this application. The reader can download the test application from the author's website (`http://interiormapping.oogst3d.net`).

A number of different versions of Interior Mapping have been implemented, so that they can be compared visually and regarding performance: with and without textures, with and without exteriors, with only ceilings and no walls, and with and without some of the extensions listed below. Through this test application, the different effects were analysed and their performance was measured, to see which implementation results in the best ratio between graphical quality and performance cost. The results are given below.

The other goal of this test application is to compare Interior Mapping to traditional polygonised buildings. Interior Mapping was compared to two different settings here: to the use of no interiors at all by making the windows only reflect, and to the use of polygons to create the interiors.

The test application measures performance by observing how many frames can be rendered in five seconds at a resolution of 1024 by 768 pixels. It runs automatically and writes the resulting frame-counts and the properties of the computer it ran on to a text-file. The test application was run on a computer with Windows XP, an AMD Athlon 2500+ processor, 1024mb RAM and an Nvidia 6600 GT GPU with 128mb video memory. A problem is that, while running a test, Windows might decide to run some other process in the background. To keep this from cluttering the test results, the same test was run five times and any occurrences of oddly high numbers were not used in the results presented here. Furthermore, to verify that the test results are representative of other GPUs as well, the test was also run on thirteen other computers with various configurations. The results of these computers were similar to the main test computer, although faster GPUs of course achieved better framerates on all tests. Nonetheless, the proportions between the framerates were roughly the same on all configurations. The complete test results of all fourteen computers can be found on the author's website (`http://interiormapping.oogst3d.net`).

5.3.1 Interior Mapping in comparison to polygonised interiors

It was expected that, if the scene is limited to a few buildings, Interior Mapping is slower than rendering interiors using actual geometry, because Interior Mapping uses a fairly complex pixel shader. When the number of buildings in the scene increases, then the performance of Interior Mapping should quickly overtake the polygonised interiors. In the test, an apartment building with 31 floors and 6 windows wide and long was used. The Interior Mapped version of this building is only 10 polygons, while the polygonised version takes 158 polygons. The Interior Mapped building

can be rendered in a single draw call, while the polygonised version takes five draw calls. For the polygonised version, floors, ceilings and walls each use a different tiling texture and can therefore not be rendered in a single draw call. After that the reflection in the windows is first drawn, followed by the the exterior walls. The version of Interior Mapping that was used here has four different textures for walls, ceilings and floors, while no lighting is calculated on the interiors.

For the Interior Mapped buildings, z-cull [28] is used. This is a technique where objects are first rendered only to the z-buffer and after that are rendered normally. Rendering to the z-buffer can be done very quickly, because it does not require the calculation of lighting, colour or complex effects like Interior Mapping. On the second pass, the z-buffer has already been filled with correct depths, so only the actually visible pixels are rendered and there is no overdraw at all. This greatly increases the performance of Interior Mapping, because overdraw means that Interior Mapping is performed several times on the same pixel on the screen. With z-cull, this never occurs. The actual performance increase of adding z-cull can vary widely from situation to situation, because it depends on the scene, the rendering order of the polygons and the camera angle. A test was performed with a single mesh that contained lots of buildings in a grid and a camera rotating around this scene. The results of this test show that in this specific case, the number of frames rendered in five seconds increases from 881 to 1265 when z-cull is used, an increase of 44%. This demonstrates that the performance of Interior Mapping can indeed greatly benefit from the use of z-cull. Because the framerate of the polygonised buildings is mainly dependent on the processing of the polygons, z-cull does not help there at all. In fact, all polygons need to be processed twice, so it even results in worse performance.

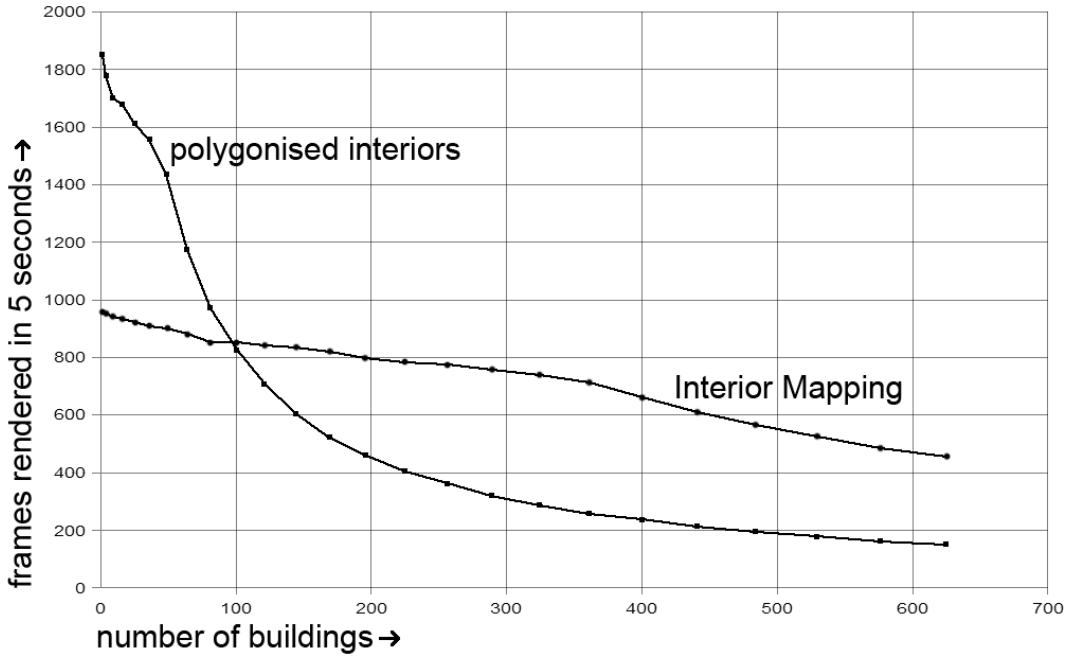


Figure 5.12. This graph shows a comparison between Interior Mapped buildings and polygonised buildings with interiors. The graph shows the number of frames rendered in 5 seconds for different numbers of buildings in the scene. More frames means a higher framerate and is thus better.

The graph in figure 5.12 shows how the number of frames rendered in five seconds changes when the number of buildings in the scene increases. As can be seen, the Interior Mapped buildings quickly overtake the polygonised ones. At 100 buildings, Interior Mapping already achieves a better framerate. This shows that the performance of the polygonised buildings is mainly dependent on the number of buildings, or in fact on the number of triangles, while the performance of Interior Mapping mainly depends on the number of Interior Mapped pixels that are actually drawn. The number of triangles is hardly relevant for Interior Mapping, because the polygon count is so low, that this is a negligible factor. At 500 buildings, Interior Mapping still only requires 5,000 polygons, whereas the polygonised buildings require 79,000 polygons. The reason why the Interior Mapped buildings do show a decrease in performance, is because the higher number of buildings still requires a higher number of draw calls. A draw call for a single object is an expensive operation, even if the object contains only 10 triangles and has no visible pixels.

5.3.2 Performance of the various versions of Interior Mapping

The previous paragraph showed that from a performance standpoint, Interior Mapping is better for large cities than creating interiors using polygons. However, there are a number of choices that have to be made when implementing Interior Mapping. Should it use only a single texture for the ceilings and leave the floors and walls blank, or should it use separate textures for the ceilings, floors and walls? And what is the effect of turning off Interior Mapping altogether?

Experiments show that using four separate textures for the floors, the ceilings and the two wall orientations, results in the same performance as only texturing the ceilings and calculating lighting for the walls and floor. The performance of reading the colours of all walls and ceilings from a single texture was even worse. Apparently, the extra instructions required to calculate which of the texture coordinates to use to read from the single texture are less efficient than just reading from four different textures using more easily calculated texture coordinates. This is in fact an advantage, as using four different textures allows artists to create much more graphically interesting rooms. The conclusion is that using four different textures is not only the best choice when graphical quality is required, but also when performance is required.

As expected, Interior Mapping is much more expensive than buildings that only have reflecting windows. A material that has a diffuse texture and reflections in the windows, which is the basic material for traditional buildings, renders 5,100 frames in 5 seconds. The same material with Interior Mapping renders only 999 frames. However, Interior Mapping adds a lot of detail to the buildings and is still easily fast enough to be applied in real-time applications.

Another important conclusion is that the resolution of the textures for the interiors does not influence performance much. When using four textures with a resolution of 256 by 256 pixels, 999 frames were rendered in five seconds. When switching to textures of 64 by 64 pixels, 1032 frames were rendered in the same time and at 16 by 16 pixels, 1053 frames were rendered. Decreasing the

texture resolution this way therefore only resulted in a 5% performance increase, while the detail inside the rooms significantly decreased.

Another test analysed the performance with different numbers of rooms in a single building. The same test was run several times, while the number of rooms in the building gradually increased from 1000 to 4,000,000. As expected, this does not influence the performance at all. The framerate did not decrease and only showed a negligible random variation between tests.

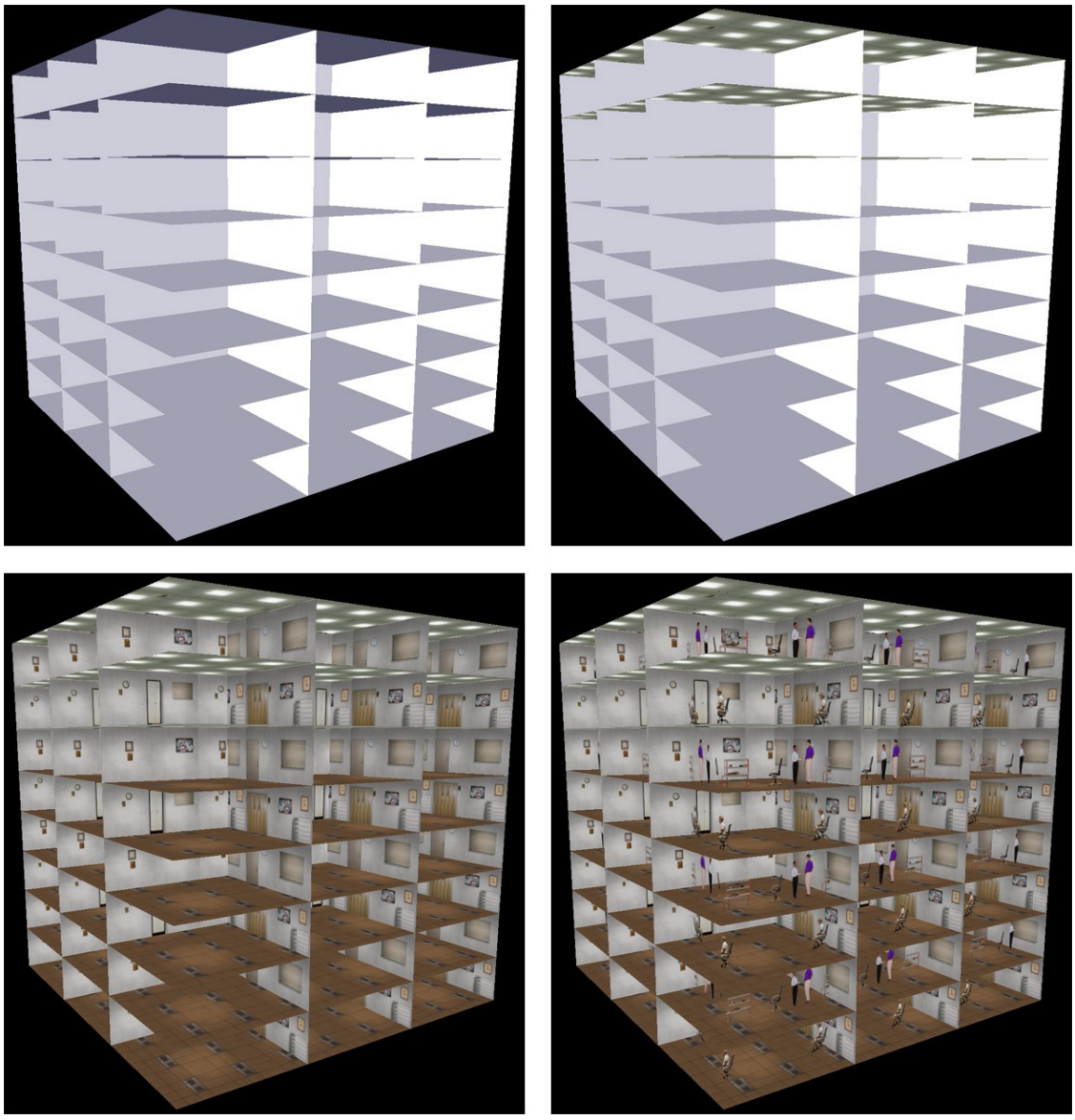


Figure 5.13. This image shows the results of various different variations of Interior Mapping. In the top left, lighting is applied to the walls, but no textures. In the top right, textures are applied only to the ceilings. In the bottom left, four different textures are used (one for each of the walls, one for the ceilings and one for the floors), and no lighting is calculated. In the bottom right, characters are added to the interiors.

5.4 Extensions

Although Interior Mapping adds a lot of detail to a building, the interiors are very repetitive. In this section, three extensions are discussed that add more variety to the different rooms.

5.4.1 Varying lighting per room

In a real city at night, some rooms will have the lights turned on, while other rooms are in the dark. This effect can be added to the basic Interior Mapping algorithm through the use of a 3D noise function. The building has a variable `c` between 0 and 1 that stores the probability that the lights in a room are on. For each room, a 3D noise-function is used to retrieve a random variable `r` between 0 and 1. The room is only lit if `r < c`. This way a room is either lit or unlit. If `c` is slowly increased, then gradually more and more rooms in the city will become lit, as can be seen in figure 5.14.

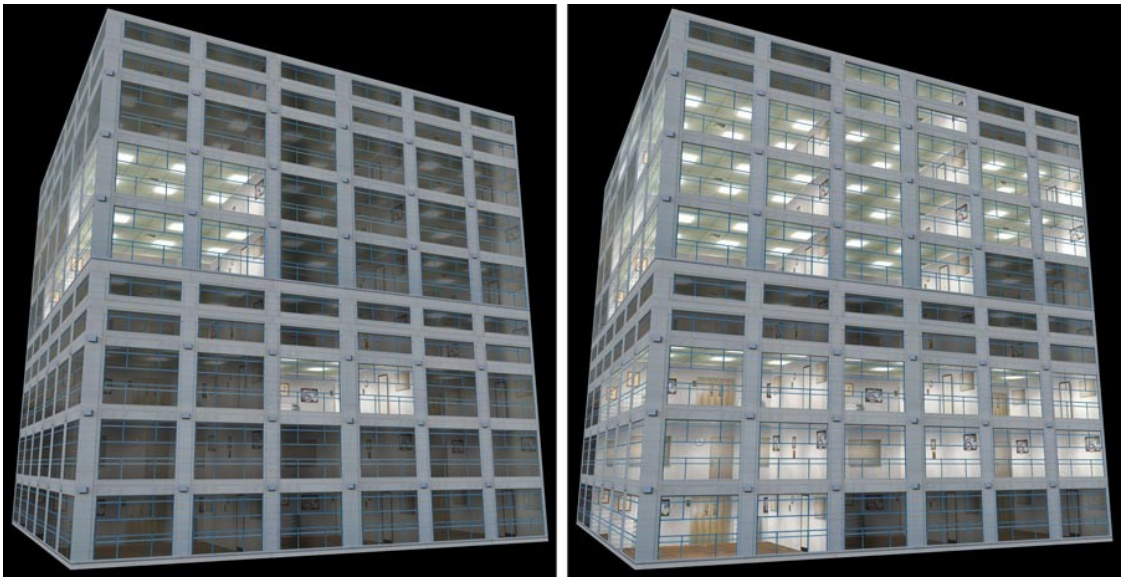


Figure 5.14. The variable `c` determines the probability that the lights in a room are on. By increasing this variable, the interior of the building goes from completely dark to completely light.

5.4.2 Varying textures per room

In the simplest approach, the texture for the interior walls will contain an image of the wall of a single room. This image is used for all the rooms, so they all look exactly the same. However, through the use of a texture atlas [43], it is possible to have different textures for each room. A texture atlas is a single texture that contains several different textures, in this case several variations to a room texture. For each room, one texture is randomly chosen from the texture atlas. An example of using a texture atlas is shown in figure 5.15.

Figure 5.15. The top image shows a texture atlas containing four different room textures. Below it the result of randomly choosing one texture for each wall of each room is shown. Note that since the atlas contains only four variations, the rooms in the bottom image still show a lot of repetition. The repetition can be decreased by adding more variations to the atlas.

5.4.3 Varying room sizes

So far, all rooms have been rendered with the exact same size. Although buildings with such a floor plan do exist, many buildings feature more variation in the sizes of their rooms. Unfortunately, if walls can have any position, it is unclear how to determine the interior wall closest to a pixel in constant time without a large space overhead. However, we can still achieve constant time if we limit how far the walls can be displaced from their position in the grid. In figure 5.16 an example is shown where walls can only be displaced from their original position by a maximum distance. Because walls can never be displaced further than the size of a room, only two walls need to be checked to find the wall closest to a certain pixel. This way, performance remains high, while more variation can be added to the rooms.

The algorithm that achieves this, works as follows:

- When the pixel shader is calculating the positions of the three walls that the ray needs to be intersected with, the positions of these walls are calculated in the same way as was done before.

- Next, the positions of the walls just before these three are also calculated. Normally, these would not be visible, as they are in front of the pixel.
- Subsequently, each of the walls is displaced with the displacement at that wall.
- This displacement might put the previous wall beyond the pixel and thus make it visible. If this is the case, then the previous wall is used for the intersection calculation and the normal wall is ignored. If after editing the displacement the previous wall is still in front of the pixel, then the normal wall is used for the intersection.

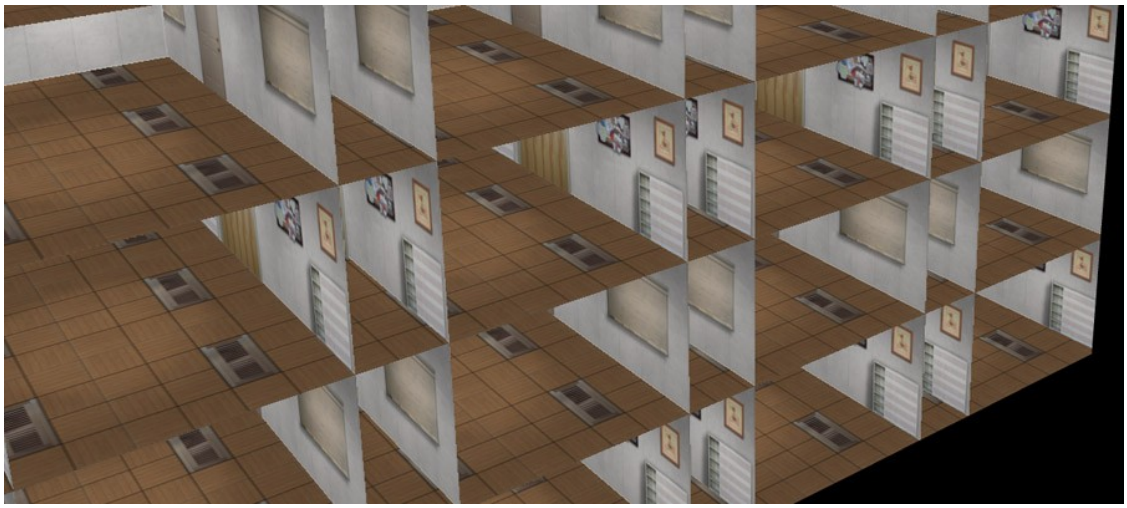


Figure 5.16. Walls with varying distances between each other are possible with Interior Mapping as well.

This algorithm is further illustrated in figure 5.17. As long as the displacement is always smaller than the standard distance between the walls, checking just the current wall and the previous wall is enough. Therefore the algorithm maintains a constant time complexity.

5.5 Conclusion

Interior Mapping has great potential for usage in the coming generation of computer games and other applications situated in virtual cities. Buildings can gain a lot of depth by adding interiors to them. Interior Mapping requires little extra work from the artists and only the extra storage space needed for the textures. It is efficient enough to be applied to games and virtual worlds for the Xbox 360, Playstation 3 and current and coming generations of PC games. Interiors made with Interior Mapping have complexity linear in the number of pixels on the screen, but constant in the number of buildings, windows, ceilings, floors and walls. As distant buildings occupy less pixels on the screen, this generates an automatic form of level of detail that greatly reduces the cost of rendering a building in the distance, whereas polygonised interiors would require the creation of more low-polygon versions of the same building to create level of detail. Interior Mapping is a great addition to any computer game that features large numbers of buildings.

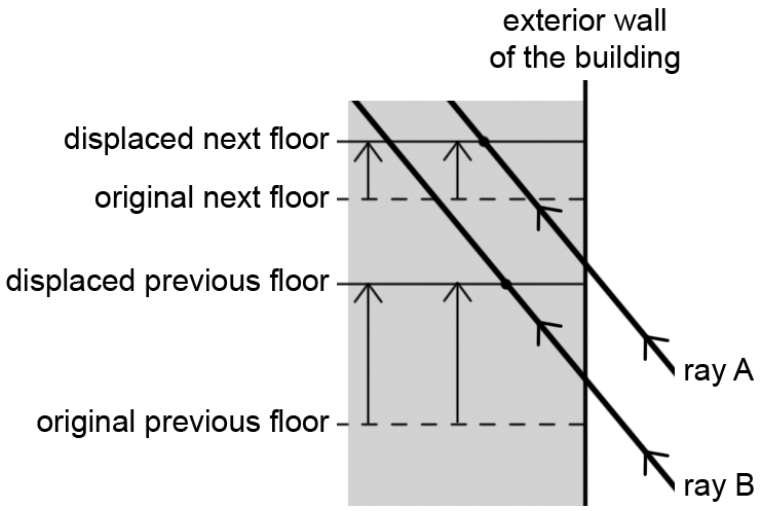


Figure 5.17. This image shows how displaced walls are made with Interior Mapping. Two rays are being shows for two different pixels: ray A and ray B. When the pixel with which ray B corresponds is being rendered, the position of the next floor ("original next floor") is calculated, as was already done in Interior Mapping without varying room sizes. The position of the floor before that ("original previous floor") is also calculated, even though this ray is already beyond that floor. Now both floors are displaced, resulting in "displaced next floor" and "displaced previous floor". After the displacement, the previous floor is beyond the point where ray B entered the building, so instead of raycasting with the next floor, the previous floor is used. The same goes for ray A, except that ray A still hits the next floor, since the previous floor was not displaced far enough to go beyond ray A. The rest of the Interior Mapping algorithm remains the same as it was without varying room sizes.

6. Comparison of rendering techniques for simple volumes

In this chapter, real-time volume rendering is achieved by combining rasterisation with raycasting in a pixel shader. Normally, rasterisation is all about rendering polygons: infinitely thin surfaces that the GPU can render very quickly. However, in the real world everything has volume. Nothing is infinitely thin. Graphics applications compensate for this by combining several polygons into a hull. There is nothing inside this hull, but because the hull is completely opaque, the user cannot see this and thus the object seems to be solid.

For most real-world objects this is a perfectly fine way to render them, but in some cases this does not work well. Transparent 'objects' like fog, slightly muddy water and smoke should be rendered as if they have real volume. In the case of an omnipresent fog that is visible everywhere, GPUs are still able to easily render those by simply changing the colours of pixels based on their distance to the camera. Because fog occurs so often, options to render it are implemented in all modern rendering hardware.

However, more complex cases like smoke are not so simple to recreate on a GPU when only surfaces are being used. The traditional approach to rendering such effects on the GPU is by placing a picture of smoke on a billboard. A billboard is a polygon that always faces the camera. Consequently, as the camera moves around, it rotates so that the camera always sees its front. An example of a billboard can be seen in figure 6.1. The textures used for this are made partially transparent, so that the smoke does not appear to be completely opaque. To animate the effect, the texture might be animated, or the effect might consist of a number of such billboards that each move and animate separately. This way a cloud of smoke coming from a chimney can be made. Since the smoke particles are not fully opaque, the z-buffer cannot be used to determine which particle is in front. Therefore, the particles need to be rendered from back to front, requiring them to be sorted before rendering.

The benefits of using billboards, are that they can give good looking results without costing too much performance, and that the storage required is small: a single texture can be used for many different billboards to still create a larger and more varied cloud of smoke. There are a number of major downsides to this technique as well. One is that the billboards rotate along with the camera. This is no problem if the camera remains in the same position, but if the camera moves around the smoke quickly, then this rotating becomes very obvious to the viewer. Another downside is that if large numbers of particles are used, they all have to be animated and rendered. This can especially decrease the framerate when large transparent particles are used, as these will often overlap visually from the point of view of the camera, thus requiring the same pixel to be rendered several times (once for the background and once for each billboard).

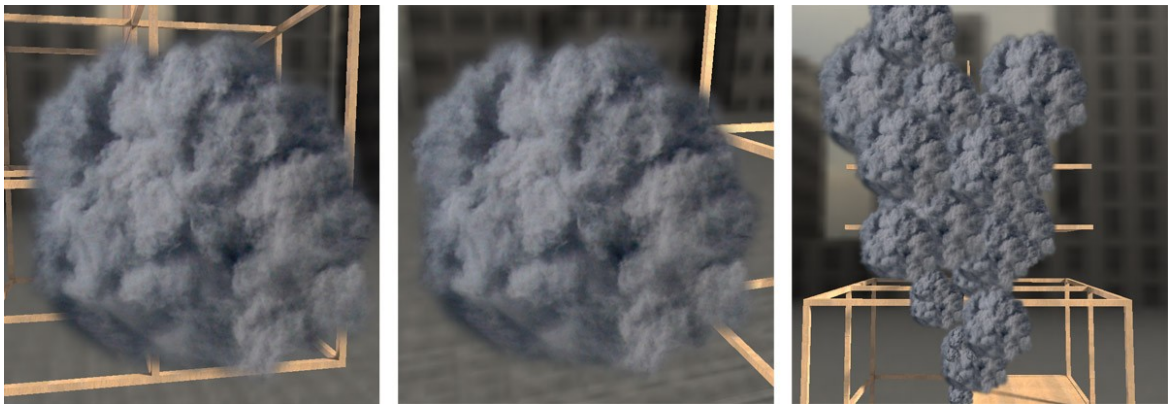


Figure 6.1. These images show smoke rendered with the use of billboards. To the left a single billboard with smoke is shown. In the middle the same scene can be seen, but rendered from a different angle. Note that the billboard still faces the camera, so the smoke looks exactly the same as in the left image. The right images shows a cloud of smoke rendered through the use of a lot of billboards that together form a particle system.

A third problem with rendering smoke through the use of standard billboards, is that they are polygons and have no real volume. If an object sticks through the billboard, then each pixel on the object will either be fully in front of the billboard, or fully behind the billboard. At the intersection of the billboard with the object, a hard edge will be visible, as can be seen in figure 6.2. If the billboard would have had real volume, then the object would have gradually faded into the smoke.

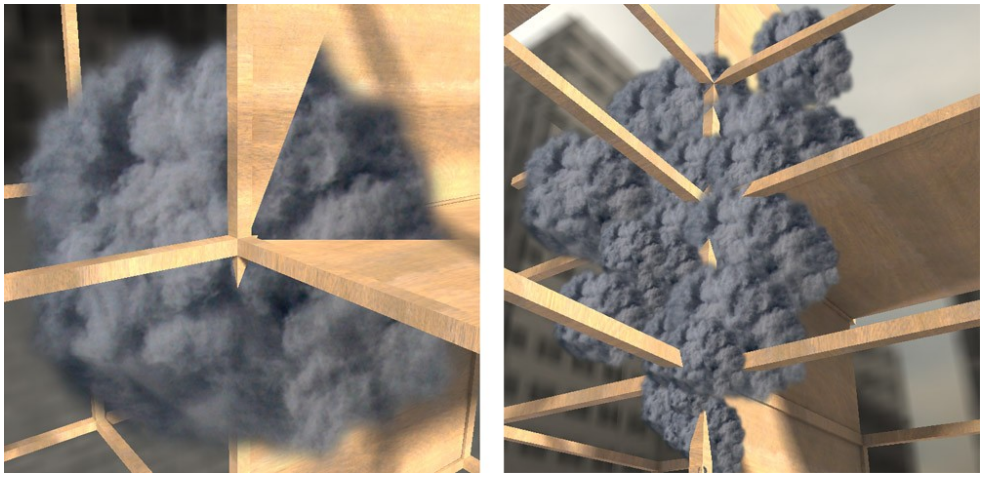


Figure 6.2. Two examples of objects intersecting with billboards. Note the lack of real volume in the smoke and the hard edges at the intersections.

This chapter mainly deals with solving this last problem. It looks into how to render volumes quickly on the GPU, and compares a number of different techniques for this, both in terms of quality and performance. Although various forms of volume rendering are discussed here, only volumes with constant density throughout the volume have been implemented and compared. This is a simplification of the real world, as for example a real cloud of smoke has variations in density and composition throughout its volume. However, rendering non-uniform volumes is significantly

more complex and takes much more processing time, so to limit the scope of this chapter and to be able to focus on a single problem, only uniform volumes have been implemented. By assuming uniform density, the techniques in this chapter can use simpler algorithms and can thus achieve very high framerates. This makes them more fit for direct use in the games and virtual worlds of the near future.

6.1 Applications

Rendering volumes has many applications in games, virtual worlds and applications that feature 3D graphics. Figure 6.3 shows a number of applications of volume rendering, which are explained in the following paragraphs.

Figure 6.3a shows smoke. Smoke might for example be emitted by a chimney, fire, an engine or a smoke grenade. It is usually a slowly evolving effect, so the user often gets the time to have a good look at it. In the case of a smoke grenade, the smoke will also often be very close to the player's viewpoint. This means that rendering smoke at a low quality will be easily noticeable to the player. Smoke frequently occurs in large volumes and close to other objects, so rendering the volume correctly is important to make it seem to interact with its environment correctly.

In figure 6.3b an explosion is shown. Although explosions might be rare in the real world, they appear in computer games all the time, so it is important to render them convincingly. Also, explosions are often located close to the camera and are specifically emphasised visually to the viewer. Therefore, high quality is important here. On the other hand, the effect changes and ends so quickly, that many rendering errors might not be noticeable during actual gameplay.

An often occurring backdrop is clouds. In most games, these are very far from the camera and can thus be rendered using static background drawings or with some simple movement in them, making volume not very relevant. However, in the case of flight simulators, the player might fly his plane right through a cloud, forcing even the camera to move inside the volume. In this case proper rendering of the volume of the clouds is very important. Figure 6.3c shows an image of a plane flying just above the clouds.

Another common application is rendering volume light. This effect occurs when bright light shines through air that contains dust. Some of the light reflects off the dust, causing the volume through which the light moves to become visible. This might for example occur when rays of sunlight break through the clouds (these rays are often called "Jacob's ladders" or "god rays"), when the beam of a flashlight is visible in the fog, or when light shines into a dark room through a window, an example of which is shown in figure 6.3d. A technique that renders this effect was already discussed in section 4.5 of this thesis.

Figure 6.3. Examples of the kind of effects that can be achieved or improved with volume rendering:
(a) Smoke in Infinity Ward's "Call of Duty 4: Modern Warfare" (2007)
(b) An explosion in Guerrilla's "Killzone 2" (2009)
(c) Clouds in Microsoft Game Studios' "Flight Simulator X" (2006)
(d) Volume light in Valve's "Half Life 2: The Lost Coast" (2005)
(e) A screenshot from Nvidia's technology demonstration "Box of Smoke" (2006)
(f) Medical visualisation of a child's head [16].

Liquids and gasses can also be visualised through volume rendering. Water is often slightly muddy, so things that are further away in the water should be less visible. The same goes for gasses. Both gasses and water are often combined with a physics simulation, the result of which should then be rendered, as can be seen in figure 6.3e. However, these effects need not always be realistic: special effects in movies and games might apply all kinds of animation and visual effects to volumes of gas.

A different class of problems where volume rendering is also applied, is medical applications. Through an MRI scan the density of each part of a human body can be retrieved, which can then be

visualised through volume rendering. The most common use of this technique is to find illnesses and to prepare for surgery. By showing cross-sections of the body to a doctor, he can assess the situation. A specific requirement in this field is often that the renderings should not be realistic, but informative. This means that such colours should be chosen that features are easily visible, and that rendering might have to adapt interactively to what a doctor is looking for specifically.

6.2 Volume rendering methods

Through the years, various approaches to rendering volumetric effects have been researched. Even though the rest of this chapter will delve deeper into only a few of these techniques, a more complete overview of approaches to rendering volume is given here.

6.2.1 Billboards and slices

The simplest and most frequently applied solution is to use billboards, as was already explained at the beginning of this chapter. Since these billboards by themselves do not have any depth, it may seem odd to mention this rendering technique as a volume rendering technique at all. However, by using larger numbers of more transparent particles, the whole will look more and more like a real volume. An object that sticks through a cloud of particles will not fade out immediately where it is behind a particle, but rather become less visible with each particle that is in front of it. So in a sense, a particle system can be considered as a form of volume rendering.

This approach can also be used when rendering a single object. By creating a volume that consists of several polygons in front of each other, each with a lower transparency, an object can be seen to gradually disappear into the volume. Each polygon is considered to be a slice of the total volume. To achieve smooth results, many slices are needed. This is not feasible in terms of framerate, as is shown further on in this chapter, but even when using only a few polygons, the results are already much more fluent than with a single billboard. An example of this can be seen in figure 6.4, where several large polygons are used to visualise smoke.

In [35] the slices approach is taken a step further by applying a different texture to each slice, so that all the slices together might form more complex geometry. This is shown in figure 6.5. In [35] this is used to render opaque objects that would otherwise require lots of polygons, but it can also be used to render complex non-uniform volumes. A problem here is that, as objects become less like spheres, it also becomes more problematic to simply rotate the object along with the camera, as is normally done with billboards. However, when the object does not rotate with the camera at all, then the camera might look at the object from the side, thus exposing the empty space between the slices. This can be seen in figure 6.6. The solution that is used in [35] to solve this, is to use several sets of slices, each with polygons oriented at a different angle.

Figure 6.4. Smoke made volumetric with slices in this screenshot from the game "F.E.A.R." (2005) by Monolith Productions.

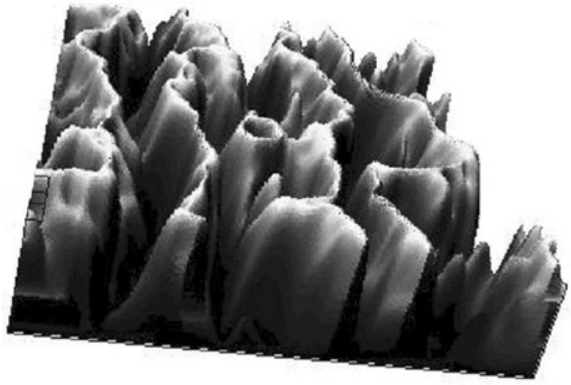


Figure 6.5. Complex terrain rendered with a large number of horizontal slices in [35].

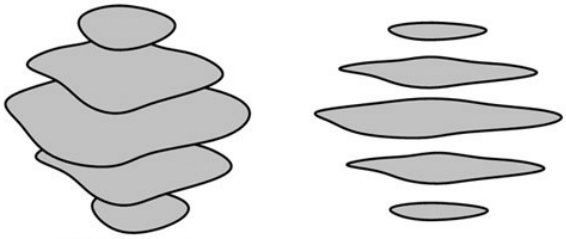


Figure 6.6. This images from [35] shows an example of how slices might be arranged. With this specific arrangement, looking at these slices from the side exposes the space between them.

Rendering volumetric effects through slices is relatively easy to implement, since the GPU simply renders a large number of textured polygons and does not need to do any extra effects or shaders on these. However, no matter how many slices are used, slices always remain a large collection of flat polygons; slice based rendering is not a real volume rendering technique. Also, as the number of slices increases, the number of pixels that the GPU needs to render quickly increases, resulting in low framerates.

6.2.2 Voxels

When such a large number of textures is stacked in slices on top of each other, this is very much like a 3D texture, as that is also a stack of 2D textures. The main difference is that, when using slices,

each slice itself is still an infinitely thin polygon. When using a 3D texture, it is possible to let each *texel* ("*TEXture ELement*") have a small 3D volume, so that all the texels together fill an area of space. This approach is called *voxel rendering* and in this case texels are called *voxels* ("*VOLume ELements*"). However, voxels are not polygons, so some way of letting the GPU render them is needed. The simplest approach is to use raytracing: as a camera ray enters the area of the 3D texture, it is traced from voxel to voxel through the volume to calculate the colour. Raytracing is straightforward to do on the CPU, but using a pixel shader, it can also be done on the GPU. Polygons are used to create a box around the entire volume. This box is rendered using the pixel shader that does the raycasting through the volume itself. The rest of the scene can be rendered as is normally done on the GPU. Some examples of the usage of this approach to rendering voxels can be found in [57] and [23].

Performing a ray-march through the voxels of a 3D texture is an expensive operation, especially if the resolution of the 3D texture is high, or if the number of pixels on the screen that require such a ray-march is large. This puts such a heavy burden on the framerate, that voxels are for example rarely used in current games. A lot of research has been done to do this more efficiently. Empty space skipping is one well known optimisation for voxel rendering: similar to cone step mapping in [17], extra data per voxel stores the size of the empty space around it, so that the ray-march can skip as much as possible. For a 3D voxel grid, this would require a sphere map that stores for each voxel the radius of the largest sphere that can be placed on that voxel's location without intersecting anything. This is especially useful if the 3D texture contains a lot of empty space.

In [26], the voxels are rendered by generating slices based on where the camera is and using the 3D texture as a texture for the slices. The benefit of rendering this way, is that the slices always face the camera. As a result, unlike in [35], it is not necessary to have several sets of slices for different viewing angles.

Another problem of using 3D textures is their size. A texture of 256 x 256 x 256 voxels already contains 16 million voxels. In the case of soft effects like smoke, this can easily be solved by storing the smoke on a low resolution. However, many other applications require sharper graphics. Often parts of the volume are empty or contain less detail, and could therefore be stored at a lower resolution. Using adaptive texture maps [30], a suitable resolution can be chosen for each section of the 3D texture, thus decreasing the total memory required for the texture. This also adds the benefit that the ray-march takes less time for the parts that are stored at a lower resolution. Figure 6.7 shows an example of this technique.

In [23], datasets are used that are too large to fit in video memory at all. Instead, the 3D texture is divided into regions and each region is loaded at a different resolution based on what the camera is currently looking at. If the camera moves, this requires constantly changing what data is in video memory, so that the parts closest to the camera are always at the highest resolution.

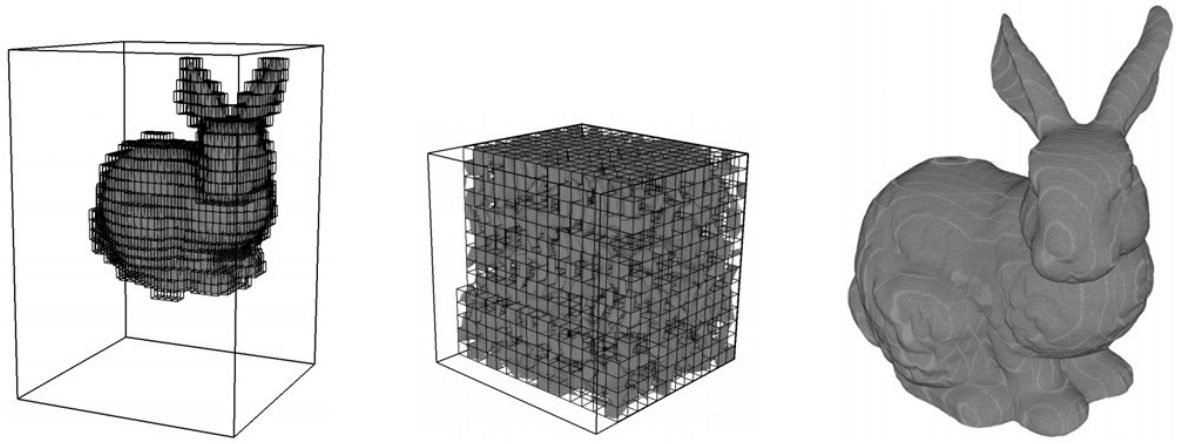


Figure 6.7. This image from [30] shows an example of using adaptive texture maps for volume rendering. The left image shows the original data. The blocks that are not completely empty are shown. These blocks are put together efficiently in a single 3D texture, which is shown in the centre image. The image to the right shows the resulting render.

Another solution for the storage would be to compress the 3D texture, as was done with the texture data in [62], which has already been discussed in section 4.3 of this thesis. However, during the ray-march, each voxel would first have to be decompressed before its contribution could be evaluated, so this would considerably decrease the performance.

An added benefit of using voxels for rendering, is that voxels are very well fit for the simulation of fluids and gasses. Each voxel can store the density and velocity of that area and this data can then be used to simulate a fluid or gas over time. This is for example done in [57], [20] and [67].

A very different application of using volume rendering with voxels can be found in [27]. Here, the voxel volume is shaped around geometry to represent much more complex geometry. In the case of [27], the complex geometry represents the fur of a toy, as can be seen in figure 6.8. Although it is disputable whether this is a form of volume rendering at all, the techniques that can be used here are similar to what has been discussed in the previous paragraphs.

A technique that is a blend between the 3D texture of voxels and the large group of particles to represent the volume, is splatting, as found in [66]. When splatting is applied to voxel rendering, each voxel is rendered as a separate small polygon, a splat. Rendering all the voxels this way obviously requires a lot of polygons, but a benefit of this technique is that it does not require shaders at all.

A common problem that all these voxel rendering techniques share, is that they are relatively slow to execute. Therefore, voxels are rarely used in 3D games. In medical applications the long rendering times are less of a problem, because these applications often require only a single object to be rendered and this can be achieved in real-time. However, if a complex scene is also being rendered around the voxel object, then rendering techniques bases on voxels are in most cases too slow to use.

Figure 6.8. This image from [27] shows the result of rendering a teddy bear by wrapping a volumetric texture around a polygonal hull.

6.2.3 Geometry

Instead of representing the volume through large amounts of particles or voxels, it is also possible to only render the geometry of the hull around the volume. A number of post-effects can then be applied to suggest volume, or the thickness of the geometry can be calculated to render really volumetric effects.

An approach that ignores volume rendering, but gives a good idea of how to render an explosion using geometry, can be found in [32]. Here geometry is used to create the shape of the explosion, while procedural noise is used to generate the colours of the explosion. To add volume to this explosion, the thickness of the geometry of the explosion would have to be calculated. This can be done in the way that is explained in [24], where first the distance from the camera to the back of all the polygons is rendered, and then the distance to the front of all the polygons. By calculating the difference between these two distances, the thickness of the object can be calculated. In [24] this is applied to a creature, as can be seen in figure 6.9, but the same could be used for explosions as well. To render real volume, the opacity of the explosion could then be decreased at the screen pixels where the thickness is low. The downside of this technique is that it can only render convex objects correctly. If several objects (or parts of objects) overlap from the viewpoint of the camera, they would have to be rendered separately and combined afterwards.

Suggesting volume by applying post effects is also possible. In [2], polygons are used to render spheres to a separate buffer. These spheres represent the smoke or explosion and can be lit and even shadowed correctly. The scene is first rendered without the smoke, after which these spheres are rendered to a separate buffer, while the z-buffer of the rest of the scene is used to cull the smoke spheres correctly. Now several post effects are applied to the buffer with the smoke, usually blur and

fractal noise. The noise makes the spheres look like smoke, while the blur blends them smoothly with their environment. Although no real volume is rendered, the hard edges where the volume intersects the environment are indeed removed here, making this technique an alternative to using some form of volume rendering. An example of this can be seen in figure 6.10.

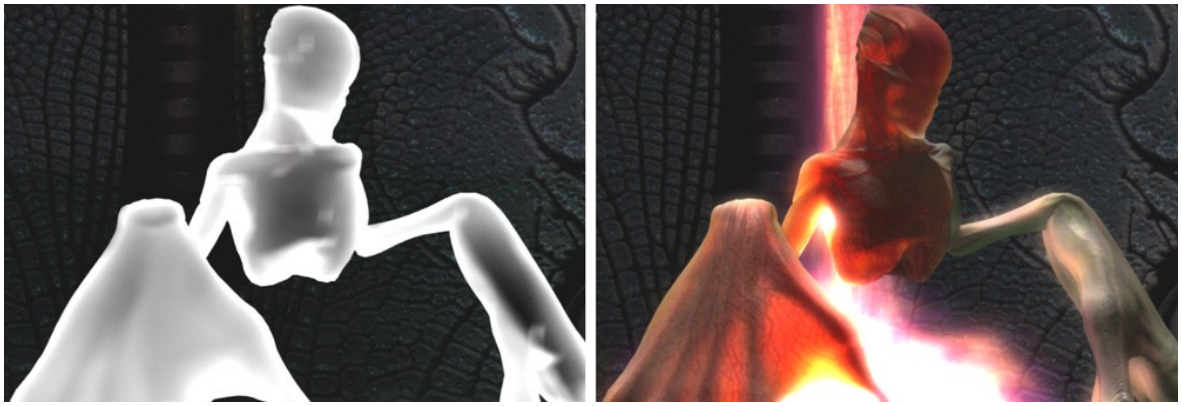


Figure 6.9. These images from [24] show how the thickness of a polygonal model can be used to render a translucent creature. The left image shows the thickness as rendered to a texture, while the right image shows the final result.

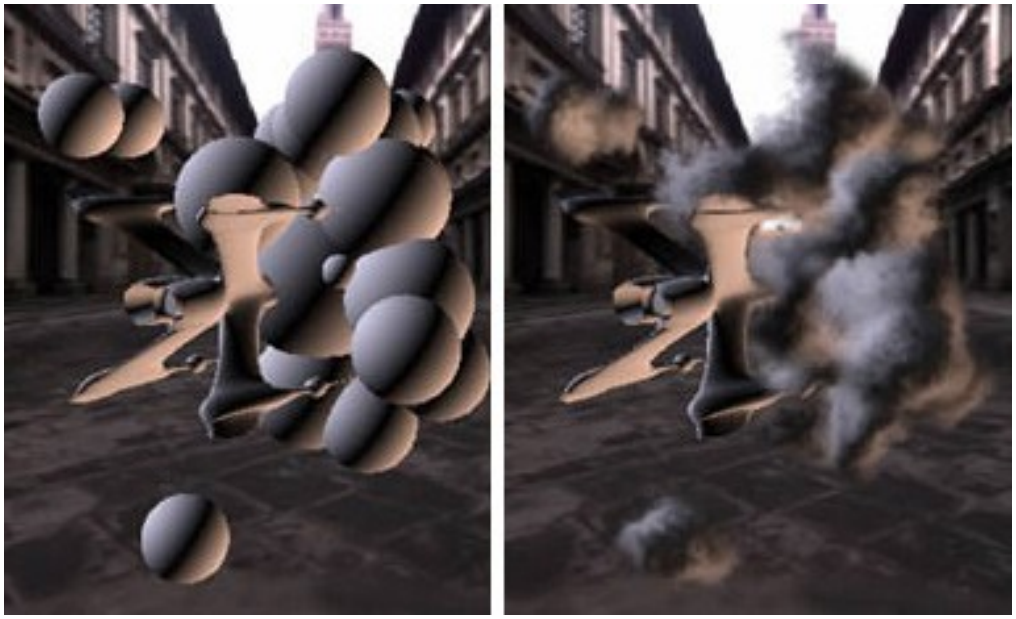


Figure 6.10. These images from [2] show how smoke can be rendered by first rendering polygonal spheres (shown in the left image) and then applying effects to these to produce the final effect (shown in the right image).

6.2.4 Raycasting and depth textures

Another approach to rendering volume is to render the scene to a depth texture and then render the volume by raycasting with a geometrical primitive like a sphere [61]. This is the main approach that has been implemented and tested further on in this chapter. Variations of this technique are in

common use in recent computer games, with particles that are rendered this way often being called *soft particles*.

Volume rendering through this technique works as follows:

- First the entire scene, except for the smoke, is rendered to a depth texture, in which the brightness of each pixel represents its distance to the camera.
- Subsequently, the scene is rendered normally.
- Now the smoke billboards are rendered. For this the z-buffer is turned off. Instead, the pixel shader uses the depth texture of the scene to look up the distance of the scene to the camera at each pixel.
- The ray from the camera to the billboard's current pixel is intersected with the volume at the billboard to find the front and back of the volume.
- The distance to the scene is then taken into account to calculate how much of the volume is in front of the scene and how much is behind it. The opacity of the volume at that pixel is calculated from this.

This process is further explained in figure 6.11. An example of the results can be seen in figure 6.12.

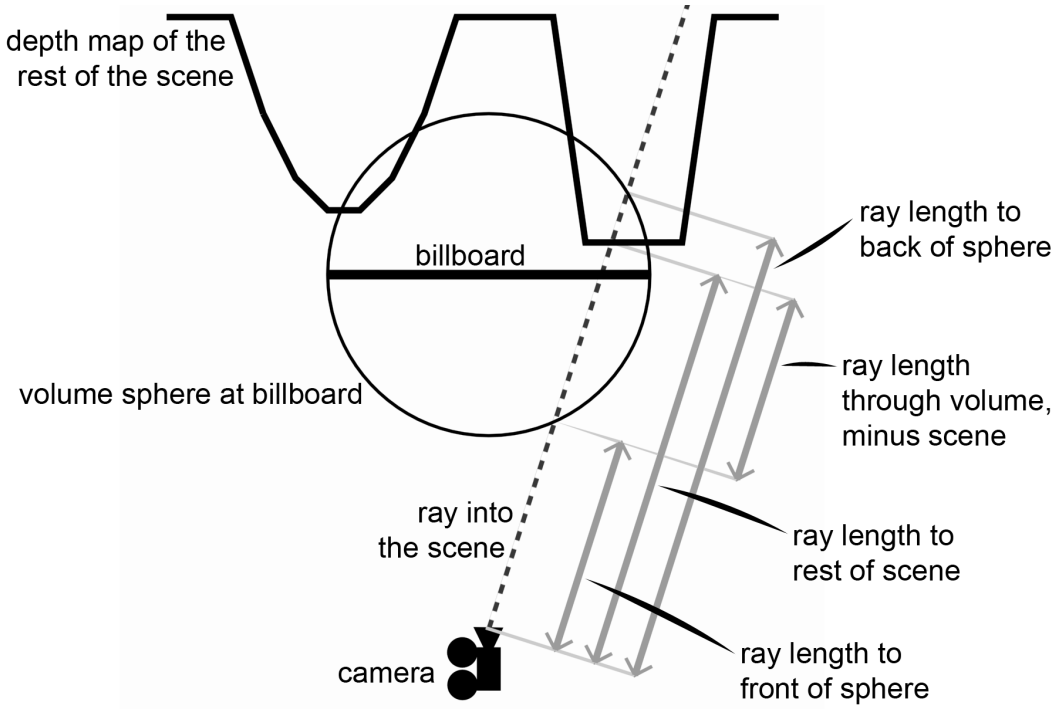


Figure 6.11. How a pixel of the flat billboard is being rendered. A sphere is considered to be at the position of the billboard. The distance of the ray towards the front and towards the back of the sphere through the billboard is calculated. The distance to the rest of the scene is read from the already rendered depth map. These are combined to get the actual length of the ray through the volume sphere, minus the part of the ray that is occluded by the rest of the scene.

The result is that a standard particle system can be used, with a texture as usual, while its volume is correctly rendered: objects that are inside the volume gradually fade out as there is more of the volume between the object and the camera. As a result, the hard edges that were discussed at the beginning of this chapter are removed. However, unlike when voxels are used, the volume is still rendered through the use of a billboard that rotates to always face the camera. The shape of the volume being used can be anything, as long as it is possible to efficiently intersect it with a ray. As can be seen in figure 6.13, not only spheres, but also boxes and ellipsoids can be used.

Figure 6.12. These two images from [61] show an explosion rendered with billboards, to the right with and to the left without correct volume rendering.

Figure 6.13. Here two images from [63] are shown. To the left the volume of the fog is an ellipsoid, to the right it is a box.

Instead of just using a raytraceable volume for the billboard, it is also possible to use a texture to store the thickness at each pixel. This way more complex shapes than boxes and spheres can be rendered efficiently. In [60], a block of particles is rendered together as a single billboard. A texture stores the distance to the front and back of the group at each pixel, plus the accumulated opacity there, as can be seen in figure 6.14. A billboard is then rendered in the same way as was done in the previous paragraph, only this time the thickness at a pixel is not calculated by raycasting with a volume, but by looking up the front and back distance at that pixel in the texture. This allows for the same smooth intersections with the environment, but also allows more complex shapes for the volumes that are being rendered. In [60] the main goal of this was to reduce the number of particles by grouping large numbers of particles together in a single billboard, but this technique is also fit to be used on a single particle.

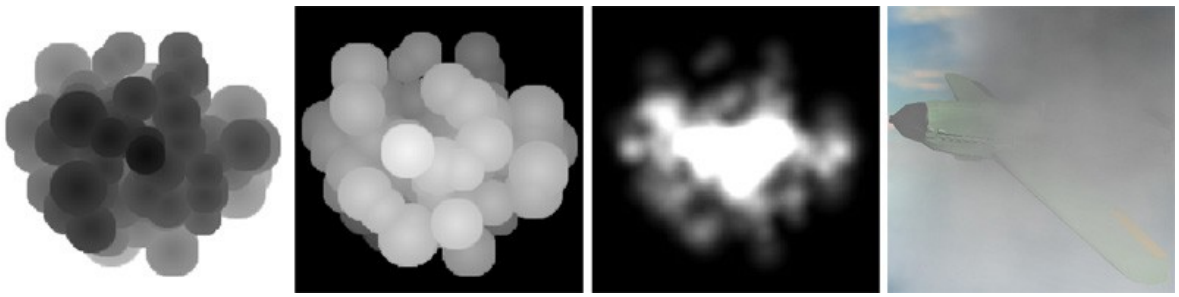


Figure 6.14. From left to right: the front depth, back depth and accumulated opacity of a billboard, stored in a texture. Fully to the right a plane flying through a cloud is shown. Note how smoothly the plane disappears into the cloud. Images from [60].

This technique has two main disadvantages. The first is that it requires a depth texture of the scene to be rendered before the actual volumes can be rendered. This is not a problem if this depth texture is already available for other effects, like deferred shading or depth of field blur. However, if the depth texture has to be rendered specifically for this effect, then this technique has a significant impact on the framerate. The other downside is that if the volume is textured, then it needs to rotate to always face the camera. This might look odd when the user quickly moves around the volume.

6.3 Description of implemented techniques

As the topic of this thesis is combining GPU rasterisation with raycasting, the most interesting techniques to analyse here are the ones using depth textures, the idea of which was explained in section 6.2.4. Although a number of different variations to this technique have already been described in various papers, a good comparison of the performance and quality of different implementations was still lacking. Therefore, a number of these techniques have been implemented and tested. To be able to get a clear view of the qualities of these techniques, this has also been done for simple billboards and slices. Techniques based on voxels have not been implemented, as these techniques seem to be so much slower, that they can be considered to solve a different problem: that of rendering non-uniform volumes. Therefore, a comparison with them is not necessary for a complete picture. Techniques based purely on post effects have not been implemented either, because they do not render real volume. To be able to compare anything, the techniques should render approximately the same effect. Therefore techniques like the geometric explosion in [32] have been ignored as well, because they are not very applicable to particle systems, which is something that the remaining techniques are very well fit for.

These techniques have been implemented in C++, using the 3D engine Ogre 3D, version 1.4.4. More information on Ogre can be found at `http://www.ogre3d.org`. The testing application uses Direct3D 9. The shaders have been implemented using Cg [29]. The source code of the implementations can be found at `http://volumes.oogst3d.net`. The implementation features both a simple, large particle, and a particle system that consists of a group of particles. The first allows a good analyses of what exactly the results of a technique are, while the latter is a form in which many applications might use these techniques. To clearly show how the rendering of the volume is done, all techniques also feature an implementation without the diffuse texture and alpha.

The rest of this section describes the techniques that have been implemented.

6.3.1 Simple billboards

This is how particles and volumes are traditionally rendered in 3D games: just a polygon with a texture that includes an alpha channel to determine its transparency. This technique performs no real volume rendering at all.

6.3.2 Slices

With this technique, a number of parallel slices together form the volume. Each slice is like the simple billboards of 6.3.1, but much more transparent, so that only all the slices together achieve the full opacity of the volume. The more slices there are, the worse the performance is, but also the smoother the intersection of the volume with another object is. Therefore, versions of this technique with 5, 9, 19, 39 and 79 slices have been created, so that the trade-off between quality and performance can be analysed.

Using slices introduces two technical problems: banding decrease the texture detail and perspective distorts the texture. The banding problem occurs because the standard frame buffer can store only 256 different colour values. If 90 slices each have to add a small portion of the total colour, then only 1 / 79th of the total colour is contributed by a single slice. This introduces a large rounding error, which is clearly visible in the render, as is shown in figure 6.15. The solution to this is to use a floating point frame buffer, which can store 2^{16} or 2^{32} different colours, depending on the bit depth of the frame buffer. In this case sixteen bits is enough. Although floating point frame buffers have been supported for a couple of years now, GPUs are still much slower at using them than at using fixed point colours. Therefore, the slices have been implemented both with and without the floating point frame buffer.

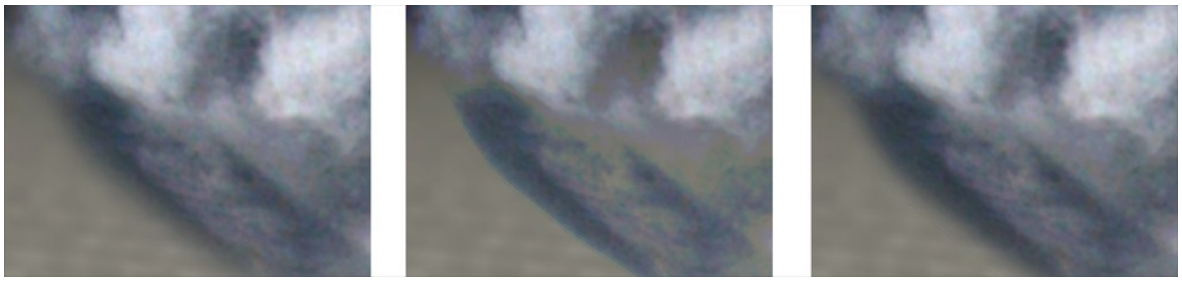


Figure 6.15. To the left a part of a volume, rendered with a single billboard. In the middle the same volume is rendered through the use of 79 slices. Note how rounding errors cause hard edges and holes. To the right the same 79 slices, but rendered to a floating point buffer and thus smooth again.

The other problem of using slices is perspective distortion, as can be seen in figure 6.16. Because the slices are in front of each other, moving the camera closer to or further from them changes the perspective of how the slices lie in front of each other. If the camera is very close to the object, then the front slice will appear much larger than the other slices. To solve this problem, the texture coordinates must somehow be calculated in such a way that the texture does not show the stretching caused by perspective.

A new technique is introduced here to solve this problem. The solution is to calculate new texture coordinates in the vertex shader. The ray from the camera to the vertex is intersected with the plane through the centre slice. The intersection point is then used as the texture coordinate for the vertex.

This technique is visualised in figure 6.17. The result is that no matter how close to or far from the slices the camera is, they will always get texture coordinates as if they are all on the centre slice, thus solving the problem of the perspective distortion[1].

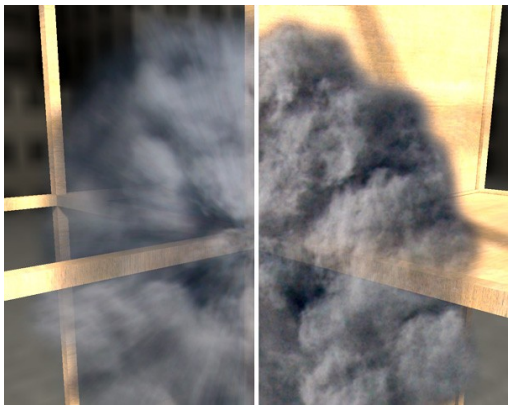


Figure 6.16. This image shows the effect of perspective distortion when working with slices. The left half of the image is distorted, while the right half shows the undistorted texture. Note how in the left half of the image, the texture is stretched in the direction of the centre of the cloud.

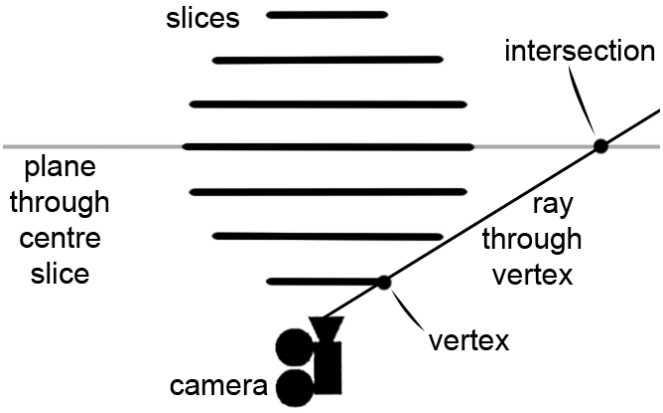


Figure 6.17. Raycasting is used to solve the perspective distortion. For each corner vertex of each slice, the UV coordinates are calculated by casting a ray through the vertex and intersecting it with the plane through the centre slice. Note how the slices are scaled such that together they form a sphere, as the smoke texture itself is roughly spherical as well.

6.3.3 Raytraced volumes

In this approach a real volume is intersected with the ray from the camera to the pixel. This yields a distance to where the ray enters the volume and one to where the ray leaves the volume, or the result might be that the ray did not intersect the volume at all. In the latter case, the object is not visible at that pixel. If the ray does intersect it, then the length of the ray inside the volume is calculated to get the opacity at that pixel.

The rest of the scene is not rendered through the use of raytracing, so to combine the raycasted volume with objects that might be in front of the volume, behind it, or intersecting with it, the scene is first rendered to a depth texture. For every pixel of the screen, this depth texture stores the distance from the camera to the object rendered there. Since a standard depth texture can contain only 256 different values, a 16 bit floating point texture is used here. This way the depth texture can

1 This is an example of using raycasting to enhance rasterised graphics and finding it here comes as a surprise: the slices were initially added to the implementation only to compare the raytraced volumes with them, not because the slices themselves were thought to be interesting in the context of this thesis. While solving the perspective distortion issue, it turned out that the slices themselves were an application of this thesis' topic as well. This is also one of the rare cases where the raycasting is done in the vertex shader instead of in the pixel shader.

be stored with enough detail to accurately calculate the intersection point of the volume with the scene. However, this does increase the rendering time of the depth texture and the time it takes to look up depth values in the pixel shader. Also, the memory usage of the depth texture is doubled.

Next, the whole is rendered normally through rasterisation, but without the volumes. Finally, the volumes are rendered on top of that. This is done by rendering billboards, but with the z-buffer turned off. Consequently, the billboard is rendered regardless of whether another object might be in front of it. The billboards are rendered with a pixel shader that performs the raycasting and calculates where the ray for a pixel enters and leaves the volume. These values are combined with the depth at that pixel in the depth texture. If the scene is in front of where the ray enters the volume, the volume is not rendered at that pixel at all, by setting the opacity to zero. If the depth is fully behind where the ray leaves the volume, the volume is entirely rendered. If the depth is inside the volume, then the visible part of the ray is calculated by taking the difference between where the ray enters the object and the depth at the pixel. The resulting length is then used as the opacity of the pixel.

So far only the transparency has been calculated. For the colour, a diffuse texture is used, as would be applied to a standard billboard. The result is that the billboard looks like a normal billboard, but when other objects intersect with it, they enter the volume smoothly, thus removing the hard edges at the intersections.

This technique can be applied with different volume primitives, such as boxes and spheres. It is interesting to see which primitive works best, especially as other articles, like for example [61], [63] and [60], do not make a good comparison on this topic. The primitives that have been implemented are shown in figure 6.18.

In 6.18a the thickness of the volume is constant. Unlike what might be expected, the result is not a box: because all the rays are cast from the camera's position, the actual volume is slightly tapered. The further the camera is from the volume, the smaller this effect is. To make the volume more like a sphere, it is possible to linearly decrease the thickness towards the edges of the billboard, as is shown in figure 6.18b. Again, the resulting volume is slightly distorted and is therefore not a real rhombus.

As the billboard already has a texture, it is also possible to use this texture to store the thickness of the volume. This is done in figure 6.18c. This allows an artist to create any shape he or she likes within the limits of the technique. The mayor limitations are that the thickness of the volume is the same in front of the billboard as it is behind, that the volume is distorted because of perspective and that the volume still always rotates towards the camera. However, defining the thickness in the texture allows the artist to fit the volume to the specific diffuse texture being used and to make dents and holes in the volume.

If a sphere is the goal, then it is of course also possible to simply intersect the ray with a real sphere, as is shown in figure 6.18d. Unlike the other shapes that have been implemented, the sphere is not distorted by perspective. Also, the other three shapes rotate with the camera to always face it, while the sphere remains the same regardless of the position or angle of the camera. Note that this is the case even though the sphere's billboard does rotate towards the camera. Another aspect that only applies to the sphere, is that the part of the ray that is inside the volume does not necessarily have the exact same length in the front of the billboard as behind the billboard. A downside of using a sphere is that raycasting it requires more instructions in the pixel shader, including a square root.

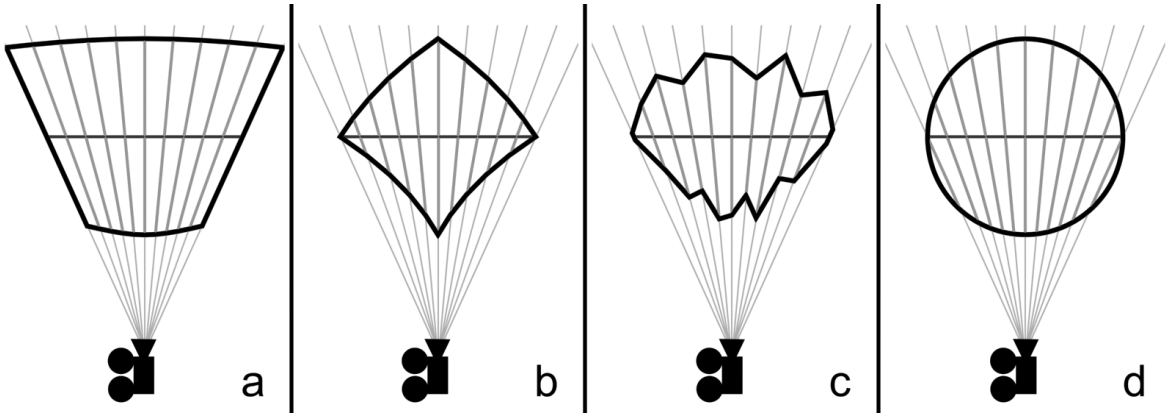


Figure 6.18. This images shows different primitives for raytraced volumes. The horizontal lines are the actual billboards.
a. Constant thickness.
b. Linearly decreasing thickness.
c. Thickness stored in a thickness texture.
d. Sphere.

Two important problems with the implementation when using a sphere as the volume primitive should be noted. The first is that if the ray does not intersect the sphere at all, the opacity will be zero. This should be taken into account when combining the sphere with the alpha of a texture. If the alpha value is positive too near the edges, the border of the sphere might be visible at some points, as can be seen in figure 6.19.

Figure 6.19. Here the texture is cut off by the border of the sphere, which is rendered as in figure 6.18d, thus creating a hard edge.

The second implementation note is that, if the sphere is scaled such that it exactly reaches the borders of the billboard, then the sphere will be visually cut off at the edges. This can be seen in figure 6.18d, where the outer rays go through the sphere, instead of being tangent to the sphere's surface. This can be repaired by making the sphere slightly smaller than the billboard. However, no matter how small the sphere is made, the camera can always be positioned closer to the sphere, thus again making the outer rays go through the sphere. A definite solution to this problem would be to use billboards that are so large that their edges are always outside the view frustum, but this would require rendering many more pixels, thus greatly decreasing the framerate. Therefore, the sphere is made only slightly smaller than the billboard here, and the cut off when the camera is very close is accepted, as the camera is expected to only be that close rarely.

Figure 6.20 shows these four volume primitives in a 3D scene. To emphasise the features of the shapes, they are rendered without a diffuse texture and with a very high density. Their features are thus much more visible than they would be in an actual application with a prominent diffuse texture, making judging their quality based on these images not very interesting. In section 6.4.1, these volumes will be combined with a diffuse texture to allow for a proper comparison of their quality. The white versions shown here do allow for an analysis of what is seen. The volume with constant thickness in figure 6.20a has hard edges where it does not intersect with other geometry, because the thickness does not decrease near the edges. Also, note that the horizontal line where the volume intersects the other geometry is slightly bend due to the perspective distortion. The curvature of the line is quite subtle, even though the camera is already very close in this image. This shows that the perspective distortion is probably not noticeable in real applications, where the volume would be combined with a diffuse texture.

The volume which has linearly decreasing thickness towards the edges is shown in figure 6.20b. The corners in the volume are caused by that the volume is like a pyramid with its top towards the viewer (plus the same pyramid again at the back). Also note that in the top left of figure 4.20b, the entire volume is visible, while in the bottom right, it is obscured by the wood in front of it. This makes the volume look oddly asymmetrical.

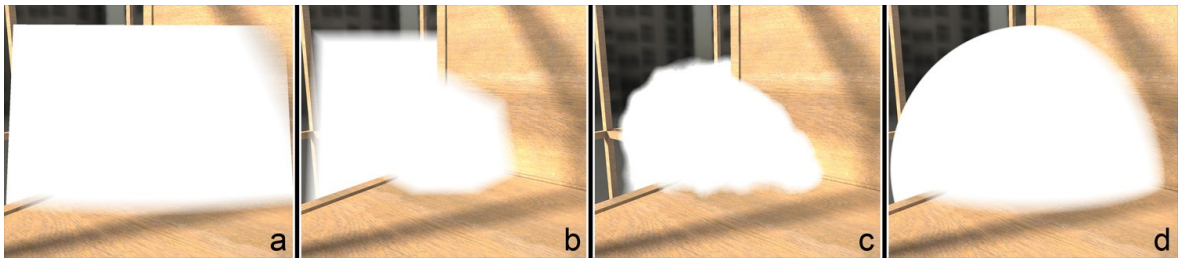


Figure 6.20. The same volume as in figure 6.13, rendered without a diffuse texture and with a high density.

A much more smooth shape can be found in figure 6.20c. The shape depends on the thickness texture and here it has been set to feature lots of noisy bumps. Finally, figure 6.20d shows the sphere.

Not only the choice of primitive is important for the end result: the function used to map the visible thickness of the volume to transparency is important as well. A linear function seems to make most sense, but using a logarithmic function or a square root might look better. The graphical effects of these three functions are shown in figure 6.21.

Figure 6.21. When the length of the ray through the volume is known, the actual alpha value can be calculated from that in various different ways. From left to right: a linear function, a logarithmic function and a square root. Note the difference in how the other objects fade into the fog.

6.4 Results

The various implementations can be compared in terms of performance and in terms of quality. From these, a best choice of which technique should be used in practical applications can be made .

6.4.1 Quality comparison

Figure 6.22 shows results of all the relevant techniques. In 6.22a the standard technique is shown: a simple billboard with a texture. As no volume is rendered at all, the plane is either fully in front of an object, or fully behind it. Even though the large volume is exactly at the centre of the scene, the billboard is rotated towards the camera in such a way that most of it is not visible. If the camera is rotated slightly, then suddenly the rest of the billboard might appear into view. Both the seams and the sudden popping into view are very unrealistic and harsh effects, while smoke like this should really be smooth.

Images b to f in figure 6.22 show the result of rendering the volume through the use of slices, using respectively 5, 9, 19, 39 and 79 slices. The more slices are used, the more gradually objects that intersect with the fog fade into it. When using 5, 9 or 19 slices, the intersections of the slices with the environment are still clearly visible, while they are already much more subtle when using 39 slices and are even hardly visible when using 79 slices. However, even at this very high slice count, the intersections are still easily distinguishable on a computer monitor, even though they might be too subtle to clearly make out in print. The slices themselves are laid out to together form a sphere, with the slices furthest to the front and back being much smaller than the centre slice, as was already shown in figure 6.17. In figure 6.22f this results in a nice curve at the bottom of the image, as if a

real sphere were intersecting with the geometry. However, when using less slices, this results in artefacts: in the bottom left and bottom right corners of figure 6.22e staircasing is visible at the intersections.

The more slices are used, the worse the previously discussed rounding errors become. This is clearly visible in the edges of the smoke at the top of figure 6.22f, which are much sharper than those of figure 6.22c. This problem is so visible that, if such high numbers of slices are used, it must really be taken care off and can not simply be ignored. Rendering the results to a floating point buffer solves this, as can be seen in figure 6.22g.

When raytracing volumes, several different primitives can be used. In figure 6.22, these are shown in images h (constant volumes), i (linear volume), j (spherical volume) and m (thickness map volume). What is immediately visible, is that the linear volume of image i shows very clear corners in the volume, making this technique useless for cases like this. The differences between the constant volume of image h and the spherical volume of image j are much more subtle. A careful observer will notice that the line at the bottom of these images, where the smoke intersects with the geometry, is nearly straight in image h and much more curved in image j, as would be expected. Surprisingly, the difference is really quite subtle. When looked at under certain angles, it becomes more obvious, but it does not seem very relevant.

The difference with the thickness texture technique of image m is much more obvious, mainly because the thickness map used here has lots of detail. Depending on which shapes are described by the thickness texture, it might also achieve exactly the same results as the constant volume, and almost exactly the same result as the spherical volume. To show this, image 6.22n shows an almost spherical thickness texture.

In figure 6.22, images j, k and l each use a spherical volume, but use different functions to calculate the actual alpha for a pixel from the thickness of the smoke at that point. These are the same functions as in figure 6.21, only here combined with a diffuse texture. In figure 6.21 the differences between these techniques were already very subtle, but now they are nearly invisible. The differences are so small, that it is difficult to draw any conclusions. In figure 6.21c the square root function seems less gradual, so that one seems to be the worst to use. Of the other two, the author prefers the logarithmic function, but this is based on personal preference rather than on any relevant argument.

Of the techniques shown in figure 6.22, the raytraced volumes seem to be the best choice. Even when using 79 slices, the edges of the polygons still remain visible, while the raytraced volumes are perfectly smooth. Of the volume primitives the one using the thickness texture appears to be the best choice, as it offers far more flexibility than the other techniques, and if needed, the thickness texture can be tweaked by an artist to get almost exactly the same results as the other techniques.

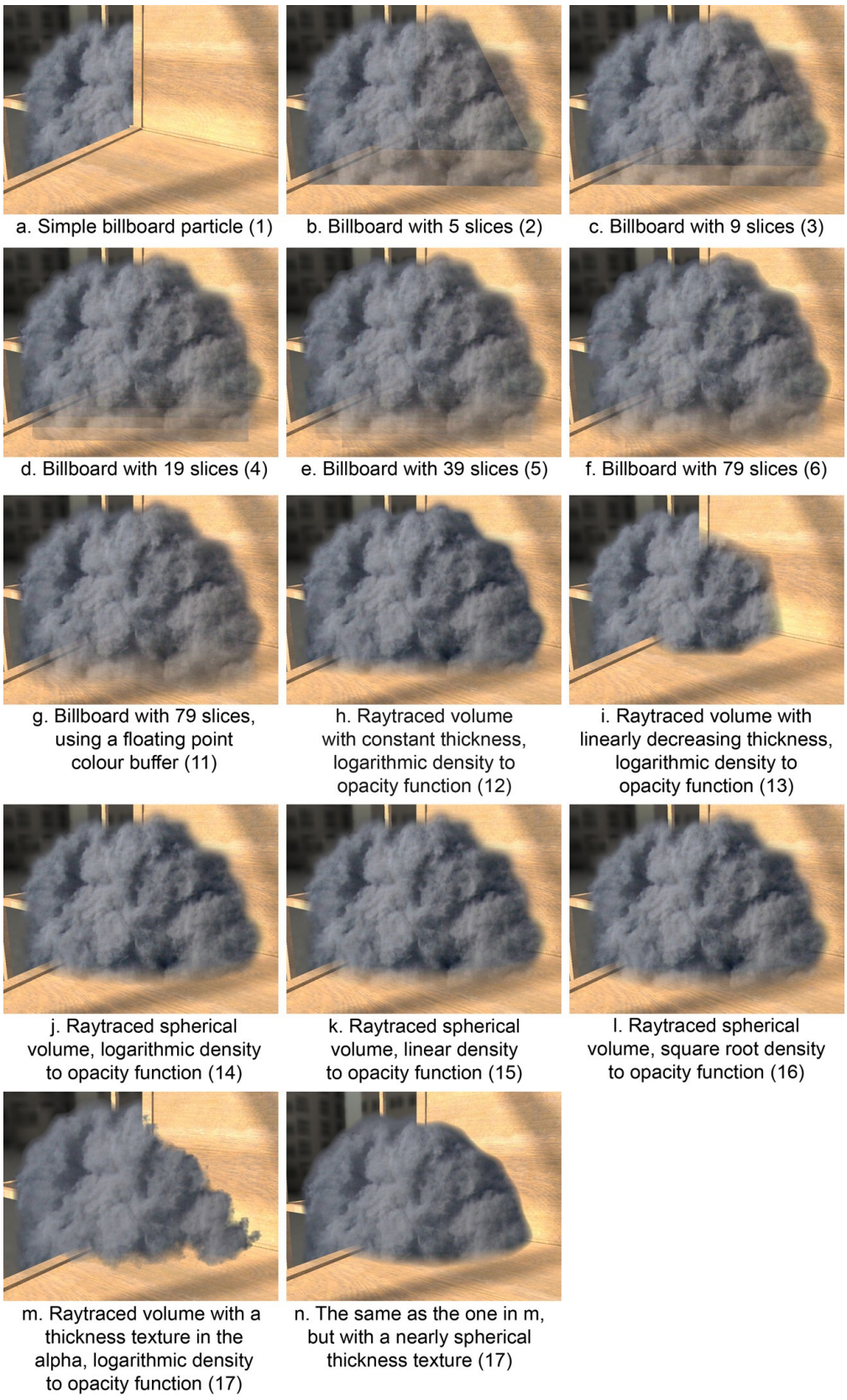


Figure 6.22. This image shows the results of the implemented techniques. The numbers correspond to those in figure 6.23.

6.4.2 Performance comparison

Performance can be looked at from several angles. In real-time graphics, framerate is generally most important, but first memory usage and development time are looked at briefly. In terms of memory, these techniques are very close to each other. They all have a diffuse texture with an alpha. The depth map approach to raytraced volumes requires an extra thickness map, but this can be combined with the alpha of the diffuse texture, so it is not necessary to store the thickness map separately. When using slices, the mesh of the particles has more polygons and vertices, which have to be stored. However, the amount of memory used for this is usually negligible: using large numbers of slices per particle is not practical in terms of framerate (as will be shown further on in this section), and when smaller numbers of slices are used, then the amount of memory used for the diffuse texture is several orders of magnitude larger than the amount of memory used for the extra vertices and polygons of the slices.

The only really relevant extra usage of memory in any of these techniques, is the depth texture that must be rendered for the raytraced volume primitives. For best quality, this texture should be rendered at the same resolution as the screen itself. Each pixel should be stored as a single 16 bit greyscale floating point value. Depending on the resolution, this might require several megabytes of memory.

The development time to actually create an effect is often ignored in research, but is very important when creating an actual product. Luckily, none of these techniques require a lot of time to implement, especially since the source code can be re-used and taken from the freely available demo's that were made for this thesis. All these techniques require a programmer to implement them, but the raytraced volume with a thickness map also requires an artist to create the thickness map itself. However, this is not a very complex job and will probably not require a lot of time to create.

To analyse the framerate, benchmarking functionality has been implemented into the testing application. A total of 41 different variations of simple particles, slices and raytraced volumes have been implemented. When benchmarking, each variation is run consecutively and the number of frames rendered in five seconds is counted. To make sure the swapping between different implementations itself does not influence the resulting frame counts, each technique is first run for one second without counting the number of frames. The five seconds of counting start only after that.

As modern computers have numerous different processes running at any given time, the results might be influenced by these other processes, for example if the virus scanner suddenly decides to check for an update online while the benchmark is running. To make sure such issues do not influence the results, all the tests have been run four times on the main testing computer. The numbers used here are the averages of these tests. It turned out that the variation from the averages presented here was never more than 0.7%.

The standard computer to run the tests on was an AMD Athlon XP 2500+ at 1.47ghz, with 1024mb RAM, running Windows XP Professional with Service Pack 3 installed. The GPU was an Nvidia 6600 GT with 128mb video memory, running Forceware driver version 163.16.

As different GPUs have different strengths and weaknesses, the tests have also been run on a number of other computers. Of course these tests showed different framerates, but the general order of the techniques regarding their speed was similar to the results on the main testing computer, so only the latter results are discussed here. The complete set of results can be found at `http://volumes.oogst3d.net`.

The relevant results can be seen below in figure 6.23. The first thing to notice here is that the ordinary billboards without any volume rendering at number 1 are much faster than any of the other techniques discussed here.

	Test description	Average number of frames rendered
1	Simple billboard particle	3573
2	Billboard with 5 slices	2324
3	Billboard with 9 slices	1691
4	Billboard with 19 slices	1066
5	Billboard with 39 slices	620
6	Billboard with 79 slices	338
7	Billboard with 5 slices, using a floating point colour buffer	754
8	Billboard with 9 slices, using a floating point colour buffer	575
9	Billboard with 19 slices, using a floating point colour buffer	381
10	Billboard with 39 slices, using a floating point colour buffer	229
11	Billboard with 79 slices, using a floating point colour buffer	129
12	Raytraced volume with constant thickness, logarithmic density to opacity function	1141
13	Raytraced volume with linearly decreasing thickness, logarithmic density to opacity function	1088
14	Raytraced spherical volume, logarithmic density to opacity function	988
15	Raytraced spherical volume, linear density to opacity function	1012
16	Raytraced spherical volume, square root density to opacity function	906
17	Raytraced volume with a thickness texture in the alpha, logarithmic density to opacity function	1138

Figure 6.23. The number of frames rendered during five seconds in each test, averaged over four executions of the same test on the same computer.

As would be expected, using large numbers of slices for the volume strongly decreases the framerate due to the enormous overdraw (i.e. the number of times a single pixel on the screen is rendered). Using more than 19 slices is not feasible if high framerates are required. The same goes for using a floating point buffer to fix rounding errors for the slices: the framerate is around three times as high when the floating buffer is not used. As the floating buffer is only needed when high numbers of slices cause rounding errors, the floating buffer techniques tested here are practically useless: their framerates are already very low with few slices, while they are only useful with many of them.

Tests 12, 13, 14 and 17 show that the choice of which type of volume to use in the case of raytraced volumes can change performance with up to 15%. The sphere is by far the slowest to render, which only makes sense: it uses the most complex algorithm, including a square root operation, which is very expensive to perform in a pixel shader. It might be a surprise that using a thickness texture yields almost exactly the same framerate as using constant thickness. The cause for this, is that the thickness texture replaces the alpha of the diffuse texture and since the diffuse texture is already being read, no extra operations are required to read the thickness texture.

Which function is used to calculate the actual alpha from the thickness of the volume is also relevant to the performance, affecting it by up to 9%. Unlike the square root, the logarithmic function is very close to the linear function, so apparently calling log() in a pixel shader costs much less processing time than calling sqrt().

When comparing slices with raytraced volumes, the outcome depends on the number of slices. It turns out that the raytraced volumes are only faster when the number of slices is at least 19.

6.4.3 Overall comparison

In conclusion, the best quality is achieved by the raytraced volume with a thickness texture, as was shown in figures 6.22m and 6.23.17. Other techniques are only relevant if they achieve higher framerates. Using nine slices and using five slices are the only volume rendering techniques that achieve a better framerate than the raytraced volume, so they are also valid options to use. The quality of using nine slices might also be quite acceptable for some applications, making it a useful choice. Finally, if performance is much more important than quality, then just using a billboard without any form of volume rendering is of course the fastest solution.

An aspect that has so far been ignored in this comparison, is that the depth texture of the scene might already have been available for other effects, like deferred shading or depth of field blur. To research the effect of this, tests have also been run where the volume was rendered through slices, but the depth texture of the scene was renderer nevertheless. It turns out that when using only five slices, rendering the depth texture for the scene decreases the framerate by no less than 48%. When the number of slices increases, the impact of the depth texture decreases: at 19 slices, rendering a depth map causes a decrease of 30%. The reason for this decrease in impact is that the slices themselves are partly transparent and therefore not rendered to the depth texture. As the rendering of the slices takes more time, the depth map becomes relatively less important. When the framerates are compared in this case, then rendering slices is only faster than raytraced volumes when only five slices are being used. At nine slices, the raytraced volumes are already faster. This means that when the depth texture of the scene is already available, using slices is not really an option, since the raycasted volumes achieve a much higher quality.

6.5 Conclusion and future work

Rendering volumes on the GPU is problematic, because the GPU specialises in rendering polygons and those are always infinitely thin. However, as has been shown in this chapter, there are many applications of volume rendering and many different techniques to render them on the GPU. Of these techniques, those that can render uniform volumes at high framerates have been implemented and compared, concluding that an improvement in quality can be achieved at acceptable framerates. The best technique for this turned out to be raytraced volumes using a thickness texture and a depth render of the scene. If performance is more important than quality, then using a low number of slices to render the volume is also an option.

In this chapter, a number of other interesting techniques have also been discussed. However, they have not been implemented and compared in terms of performance and quality. For future work it might be interesting to implement voxel rendering (section 6.2.2) and see what its presumably poor performance in terms of framerate really is, and what kind of resolution is needed for the voxels to achieve good quality. It is expected that using voxels will turn out to result in such low framerates, that using them is not an option for real-time virtual worlds and games in the near future, but this should be validated experimentally.

Another interesting technique is representing the volume by surface geometry, as was discussed in section 6.2.3 of this chapter, and calculating the thickness from the distance of the front faces to the back faces. It is expected that when rendering the kind of effects that were implemented here, this will result in lower framerates, because representing the volume through polygons requires a lot of extra polygons. Also, it will probably be a lot slower to render the distance to the front and to the back of the volume and then combining these to get the thickness of the volume. However, the types of objects that can be rendered this way are more diverse, so it would be interesting to see what framerates this technique really achieves.

7. Conclusion

Raytracing and triangle rasterisation are opposite techniques. Awkward to combine as they may be, many useful and interesting rendering techniques can be found in their collaboration. A few of these are already in use: displacement mapping, voxel rendering, reflection and refraction, caustics, ambient occlusion and light shafts, as well as more obscure techniques like Nvidia's eye demo. Many of these techniques require pixel shaders with often quite complex algorithms to achieve them, while others require rendering the scene several times from different perspectives, or with different settings, to be able to combine these renderings into the final image. This type of complexity does make sense: rasterisation is so different from raytracing, that applying raytracing to improve rasterised graphics can only be done with significant extra effort.

One of the already existing techniques has been researched further in this thesis: a careful comparison has been made of the performance and quality of rendering volumes through a shader that raytraces geometrical primitives and combines this with a depth map render of the scene. This produces volumes that blend smoothly with any intersecting geometry. Although the technique itself is not new and is already being applied in many games and virtual worlds (often being called “soft particles” there), a proper analysis of the best way to implement it was still missing. It led to the conclusion that this raycasting technique does indeed deliver smooth volume rendering and that in most cases it also achieves a better framerate than rendering the volume through polygonal slices.

This thesis has also introduced a completely new combination of rasterisation and raytracing: Interior Mapping. With this technique, it is possible to render the interiors of buildings as seen through windows, even if the number of buildings in the city, and thus the number of rooms in the interiors, is very high. With Interior Mapping this can be done with a relatively small decrease in performance and no extra memory usage per room or building.

As most combinations of rasterisation and raytracing are relatively complex to execute and often require a lot of processing power, many of these techniques are hardly used in current virtual worlds. However, as GPUs become even faster and the complexity of the shaders they can run increases, more of these techniques will become feasible to run in real-time. Therefore it is expected that in the near future raytracing will be applied more and more to improve the quality of rasterised real-time graphics. Of these techniques, displacement mapping is probably the technique to gain the most ground, as it can be applied to many different types of objects to improve their detail and visual quality.

One might argue that in the long run, raytracing will make rasterisation obsolete altogether. Raytracing might be more fit to render complex and realistic scenes, but it does not seem probable that this will actually happen in the foreseeable future. Rasterisation is much faster to perform and the performance gap between raytracing and rasterisation is so enormous, that it will take a long time before the extra quality that can be achieved with raytracing outweighs the extra scene

complexity that can be rendered with the speed of rasterisation. Therefore, the topic of this thesis will continue to be relevant: rasterisation cannot render certain effects, while raytracing can, so combining the two will continue to be a valuable option in the coming decades.

Appendix: Interior Mapping source code

Various variations to Interior Mapping can be implemented, for example with or without randomised textures and with or without furniture and characters inside the rooms. The code presented here gives the basic implementation of Interior Mapping, without any extra effects, written in the shader language Cg. The source code for all the different versions that have been implemented for this thesis can be found in the Interior Mapping demo, which can be found at `http://interiormapping.oogst3d.net`.

The vertex shader

```
void InteriorMapping_VertexShader
      (
            // Data from the vertex.
            float4 position      : POSITION,
            float3 normal        : NORMAL,
            float2 uv            : TEXCOORD0,

            // Output position on the screen for the GPU.
            out float4 oPosition           : POSITION,

            // Data for the pixel shader.
            out float4 oLighting           : COLOR0,
            out float2 oUv                 : TEXCOORD0,
            out float3 oCopiedPosition     : TEXCOORD1,          // In object space.
            out float3 oReflection         : TEXCOORD2,

            // The matrix that transforms from object space to screen space.
            uniform float4x4     worldViewProjMatrix,

            // The matrix that transforms from object space to world space.
            uniform float4x4     worldMatrix,

            // The position of the camera, in world space.
            uniform float3       cameraPosition,

            // A variable that is used to modify the scale of the exterior's texture.
            uniform float        uvMultiplier
      )
{
      // Transform the position of the vertex from object space to screen space.
      oPosition = mul(worldViewProjMatrix, position);
```

```
    // Just copy this data for the pixel shader.
    oUv = uv * uvMultiplier;
    oCopiedPosition = position;

    // The direction of the reflection (needed for reflections in the windows)
    // is calculated in world space to keep it from rotating with the object.
    float4 worldPosition = mul(worldMatrix, position);
    float3 worldNormal = mul(float3x3(worldMatrix), normal);
    oReflection = reflect(worldPosition - cameraPosition, worldNormal);

    // Calculate lighting on the exterior of the building with a hard-coded directed light.
    float lightStrength = dot(normal, float3(0.5, 0.33166, 0.8));
    oLighting = saturate(lightStrength) * float4(1, 1, 0.9, 1);

    // Add some ambient lighting.
    Olighting += float4(0.3, 0.3, 0.4, 1);
}
```

The pixel shader

```
void InteriorMapping_PixelShader
    (
        // Interpolated input from the vertex shader.
        float4 lighting      : COLOR0,
        float2 uv            : TEXCOORD0,
        float3 position      : TEXCOORD1,          // In object space.
        float3 reflection    : TEXCOORD2,          // In world space.

        // The output colour of the pixel.
        out float4 oColour   : COLOR,

        // Textures for the walls, the exterior and the reflections in the windows.
        uniform sampler2D    ceilingTexture,
        uniform sampler2D    floorTexture,
        uniform sampler2D    wallXYTexture,
        uniform sampler2D    wallZYTexture,
        uniform sampler2D    diffuseTexture,
        uniform samplerCUBE  cubeTexture,

        // The number of walls per unit (higher numbers means smaller rooms),
        // for the X, Y and Z directions, packed together in a single float3.
        uniform float3       wallFrequencies,
        uniform float3       cameraPosition        // In object space.
    )
```

```
{
    // Calculate the vector from the camera to the surface.
    float3 direction = position - cameraPosition;

    // Scale the pixel's position into a space where each room is of unit size.
    float3 walls = position * wallFrequencies;

    // Round down to get the positions of the walls.
    walls = floor(walls);

    // When looking in the positive direction, add one to get the next wall.
    walls += step(float3(0, 0, 0), direction);

    // Scale back into object space for the correct sizes of the rooms.
    walls /= wallFrequences;

    // Calculate how much of the ray is needed to get from the cameraPosition to each of the walls.
    // This results in three separate values, one for each of the walls.
    // These are calculated at once and stored in the three components of rayFractions.
    // Calculating them at once like this saves instructions, as the GPU performs operations on
    // a float3 in exactly the same time as operations on a single float value.
    float3 rayFractions = (float3(walls.x, walls.y,walls.z) - cameraPosition)
                            / direction;

    // Calculate the texture-coordinates of the intersections with each of the walls.
    float2 intersectionXY = (cameraPosition + rayFractions.z * direction).xy;
    float2 intersectionXZ = (cameraPosition + rayFractions.y * direction).xz;
    float2 intersectionZY = (cameraPosition + rayFractions.x * direction).zy;

    // Look-up the colours of the walls, the floor and the ceiling, in their textures.
    float4 ceilingColour = tex2D(ceilingTexture, intersectionXZ);
    float4 floorColour = tex2D(floorTexture, intersectionXZ);
    float4 wallXYColour = tex2D(wallXYTexture, intersectionXY);
    float4 wallZYColour = tex2D(wallZYTexture, intersectionZY);

    // Choose either the colour of the ceiling or the colour of the floor,
    // based on whether we are looking up or down at this pixel.
    float lookingUp = step(0, direction.y);
    float4 verticalColour = lerp(floorColour, ceilingColour, lookingUp);

    // Choose the closest of the walls for the colour of the pixel. The closest wall is the
    // one with the lowest value in rayFractions, as rayFractions stores the fraction of
    // the ray that is needed to get to the wall. Some smart step-usage is needed here
    // to compensate for the lack of an if-statement in ps_2_0.
    float xVSz = step(rayFractions.x, rayFractions.z);
    float4 interiorColour = lerp(wallXYColour, wallZYColour, xVSz);
```

```
    float rayFraction_xVSz = lerp(rayFractions.z, rayFractions.x, xVSz);
    float xzVSy = step(rayFraction_xVSz, rayFractions.y);
    interiorColour = lerp(verticalColour, interiorColour, xzVSy);

    // Look-up the colour for the exterior of the building.
    float4 diffuseColour = tex2D(diffuseTexture, uv);

    // Calculate the colour of the wall by combining the texture's colour
    // with the lighting that was calculated in the vertex shader.
    float4 wallColour = diffuseColour * lighting;

    // Look-up the colour for the reflections in the building's windows.
    float4 cubeColour = texCUBE(cubeTexture, reflection);

    // What is seen at the windows is a combination of reflection and the interior.
    float4 windowColour = cubeColour + interiorColour;

    // The alpha of diffuseColour stores whether this is a window or not.
    // Choose either the colour of the wall or of the window based on this.
    oColour = lerp(wallColour, windowColour, diffuseColour.a);
}
```

References

[1] J. Arvo, **Backward ray tracing**, in Developments In Ray Tracing, SIGGRAPH '86 course notes, volume 12, 1986

[2] H. Bahnassi, W. Bahnassi, **Volumetric clouds and mega particles**, in ShaderX5: Advanced Rendering Techniques, chapter 5.3, ISBN 978-1584504993, 2006

[3] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, **Computational geometry - Algorithms and applications**, second edition, Springer-Verlag, ISBN 3-540-65620-0, 2000

[4] J. F. Blinn, **Simulation of wrinkled surfaces**, ACM SIGGRAPH Computer Graphics, volume 12, issue 3 p286-292, 1978

[5] P. Bourke, **Quaternion Julia fractals**, `http://ozviz.wasp.uwa.edu.au/~pbourke/fractals/quatjulia/`, 2001

[6] P. Brown, **S3 texture compression**, NVIDIA Corporation, November, 2001

[7] N.A. Carr, J.D. Hall, J.C. Hart, **The ray engine**, ACM SIGGRAPH/Eurographics Conference on Graphics Hardware, Saarbrucken, Germany, p37-46, 2002

[8] L. Case, **DirectX 11: Sooner than you think**, article in Extreme Tech, `http://www.extremetech.com/article2/0,2845,2329314,00.asp`, 2008

[9] E. E. Catmull, **Computer display of curved surfaces**, IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure, Los Angeles, May, p11-17, 1975

[10] P. Cignoni, M. Di Benedetto, F. Ganovelli, E. Gobbetti, F. Marton, R. Scopigno, **Raycasted blockmaps for large urban model visualisation**, Computer Graphics Forum, volume 26, number 3, p405-413, 2007

[11] J. Clark, **Hierarchical geometric models for visible surface algorithms**, Communications of the ACM 19(10), p547-554, 1976

[12] D. Connors, **Caustic promises 200x boost in raytracing by 2010**, news report on Tom's Hardware, `http://www.tomshardware.com/news/Caustic-Graphics-Raytracing,7240.html`, March 10, 2009

[13] J. Díaz, H. Yela, P.-P. Vázques, **Vicinity occlusion maps**, in Computer Graphics International proceedings, p56-63, 2008

[14] P. J. Diefenbach, N. I. Badler, **Pipeline rendering: interactive refractions, reflections and shadows**, Displays: Special Issue on Interactive Graphics, 15(3), p173-180, 1994

[15] J. van Dongen, **Interior Mapping - A new technique for rendering realistic buildings**, in Computer Graphics International proceedings, p170-177, 2008

[16] R. A. Drebin, L. Carpenter, P. Hanrahan, **Volume rendering**, in ACM SIGGRAPH Computer Graphics, volume 22 issue 4, p65-74, 1988

[17] J. Dummer, **Cone step mapping: An iterative rayheightfield intersection algorithm**, `http://www.mganin.com/lonesock/ConeStepMapping.pdf`, 2006

[18] P. Dutré, P. Bekaert, K. Bala, **Advanced global illumination**, A K Peters, ISBN 1-56881-177-2, 2003

[19] S. Elliot, **Beyond the box: Orange Box afterthoughts and the future of Valve**, interview with Gabe Newell, `http://www.1up.com/do/feature?pager.offset=0&cId=3165930`, Games For Windows Magazine #13, December, 2007

[20] R. Fedkiw, J. Stam, H. W. Jensen, **Visual simulation of smoke**, Proceedings of the 28th annual conference on Computer graphics and interactive techniques, p15-22, August 2001

[21] R. Fernando, M. J. Kilgard, **The Cg tutorial: The definitive guide to programmable real-time graphics**, Addison-Wesley Professiona, ISBN 978-0321194961, 2003

[22] A. Fournier, **Normal distribution functions and multiple surfaces**, Graphics Interface '92 Workshop on Local Illumination, May, p45-52, 1992

[23] E. Gobbetti, F. Marton, J. A. I. Guitián, **A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets**, The Visual Computer: International Journal of Computer Graphics, volume 24 issue 7, p797-806, 2008

[24] S. Green, **GPU programming exposed - The naked truth behind NVIDIA's demos**, presented by Nvidia at SIGGRAPH, 2005

[25] [RenderToTexture95] S. Green, **The OpenGL framebuffer object extension**, Game Developers Conference, 2005

[26] M. Ikits, J. Kniss, A. Lefohn, C. Hansen, **Volume rendering techniques**, GPU Gems, Chapter 39, 2004

[27] J. T. Kajiya, T. L. Kay, **Rendering fur with three dimensional textures**, ACM SIGGRAPH Computer Graphics, volume 23 number 3, p271-280, July 1989

[28] [Geforce605] E. Kilgariff, R. Fernando, **The GeForce 6 series GPU architecture**, GPU Gems 2, p471-491, 2005

[29] D. Kirk, et al., **Cg Toolkit user's manual - A developer's guide to programmable graphics**, Nvidia, `http://developer.nvidia.com/object/cg_toolkit.html`, 2005

[30] M. Kraus, T. Ertl, **Adaptive texture maps**, Proceedings of the ACM SIGGRAPH/Eurographics conference on Graphics hardware, Saarbrucken, Germany, September 1-2, 2002

[31] B. Langsdorf, **GPU programming exposed - The naked truth behind NVIDIA's demos**, presented at SIGGRAPH sponsored exhibitor sessions, 2005

[32] T. Lorach, **Bigger bang with fewer sprites**, presented at Game Developers Conference Europe, 2003

[33] P.W.C. Maciel, P. Shirley, **Visual navigation of large environments using textured clusters**, SI3D, p95-102, 211, 1995

[34] N. L. Max, **Shadows for bump-mapped surfaces**, Proceedings of Computer Graphics Tokyo '86 on Advanced Computer Graphics, Tokyo, Japan, p145-156, 1986

[35] A. Meyer, F. Neyret, **Interactive volumetric textures,** in Rendering Techniques'98, Eurographics Rendering Workshop, p157-168, 1998

[36] Microsoft, **Shader Stages (Direct3D 10)**, MSDN Direct3D programming guide, `http://msdn.microsoft.com/en-us/library/bb205146(VS.85).aspx`, 2008

[37] K. Mitchell, **Volumetric light scattering as a post-process**, in GPU Gems 3, ISBN 978-0321515261, Chapter 13, 2007

[38] M Mittring, **Finding next gen - CryEngine 2**, Advanced Real-Time Rendering in 3D Graphics and Games Course, SIGGRAPH, 2007

[39] P. Muller, P. Wonka, S. Haegler, A. Ulmer, L. van Gool, **Procedural modeling of buildings**, ACM Transactions on Graphics, volume 25, issue 3, p614-623, 2006

[40] Nvidia, **Cascades**, technology demo, `http://www.nzone.com/object/nzone_cascades_home.html`, 2007

[41] Nvidia, **CUDA**, http://www.nvidia.com/cuda, last accessed on March 10th, 2009

[42] Nvidia, **Gelato**, http://www.nvidia.com/gelatozone, last accessed on March 10th, 2009

[43] NVIDIA, **Improve batching using texture atlases**, SDK Whitepaper, NVIDIA, july, 2004

[44] M. M. Oliveira, G. Bishop, D. McAllister, **Relief texture mapping**, Proceedings of the 27th annual conference on Computer graphics and interactive techniques, p359-368, 2000

[45] J. Painter , K. Sloan, **Antialiased ray tracing by adaptive progressive refinement**, ACM SIGGRAPH Computer Graphics, v.23 n.3, p281-288, July 1989

[46] F. Pellacini, K. Vidimce, A. Lefohn, A. Mohr, M. Leone, J. Warren, **Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography**, ACM Transactions on Graphics 24, 3 (August), p464-470, 2005

[47] M. Pharr, S. Green, **Ambient occlusion**, in book: GPU Gems, Addison-Wesley Professional, ISBN 978-0321228321, 2004

[48] F. Policarpo, M. M. Oliveira, **Relaxed cone stepping for relief mapping**, in GPU Gems 3, ISBN 978-0321515261, Chapter 18, 2007

[49] T. Scheuermann, **Advanced depth of field**, presented at Game Developers Conference, http://ati.amd.com/developer/gdc/Scheuermann_DepthOfField.pdf, 2004

[50] J. Schmittler, D. Pohl, T. Dahmen, C. Vogelgesang, P. Slusallek, **Realtime ray tracing for current and future games**, ACM SIGGRAPH courses, 2005

[51] J. Schmittler, I. Wald, P. Slusallek, **SaarCOR: a hardware architecture for ray tracing**, Proceedings of the ACM SIGGRAPH/Eurographics conference on Graphics hardware, September 01-02, 2002

[52] A. L. Shimpi, D. Wilson, **Microsoft's Xbox 360, Sony's PS3 - A hardware discussion**, article on AnandTech, http://www.anandtech.com/video/showdoc.aspx?i=2453&p=1, 2005

[53] D. Shreiner, M. Woo, J. Neider, T. Davis, **OpenGL programming guide: The official guide to learning OpenGL**, Addison-Wesley Professional, ISBN 978-0321481009, 2007

[54] M. Slater, A. Steed, Y. Chrysanthou, **Computer graphics and virtual environments, from realism to real-time**, Addison Wesley, ISBN 0201624206, 2002

[55] SplutterFish, **Artist spotlight: The Orphanage**, article on the website of SplutterFish, http://splutterfish.com/sf/WebContent/ArtistSpotlightTheOrphanage, April 26, 2004

[56] L. Szirmay-Kalos, B. Aszódi, I. Lazányi, M. Premecz, **Approximate ray-tracing on the GPU with distance impostors**, proceedings of Eurographics, volume 24, number 3, 2005

[57] S. Tariq, I. Llamass, **Real-time volumetric smoke using Direct 3D 10**, presented at Game Developer's Conference, San Francisco, 2007

[58] B. Turner, **Real-time dynamic level of detail terrain rendering with ROAM**, Gamasutra.com feature article, http://www.gamasutra.com/view/feature/3188/realtime_dynamic_level_of_detail_.php?page=3, 2000

[59] T. Umenhoffer, G. Patow, L. Szirmay-Kalos, **Caustic triangles on the GPU**, in Computer Graphics International proceedings, p222-228, 2008

[60] T. Umenhoffer, L. Szirmay-Kalos, **Real-Time Rendering of Cloudy Natural Phenomena with Hierarchical Depth Impostors**, Proceedings of Eurographics Conference, short papers, 2005

[61] T. Umenhoffer, L. Szirmay-Kalos, G. Szijártó, **Spherical Billboards and their Application to Rendering Explosions**, Proceedings of Graphics Interface 2006, Quebec, Canada, June 79, 2006

[62] X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, H.-Y. Shum, **Generalized displacement maps**, in Eurographics Symposium on Rendering, p227-233, 2004

[63] C. Wenzel, **Real-time Atmospheric Effects in Games**, ACM SIGGRAPH 2006 Courses, Boston, Massachusetts, July 30-August 3, 2006

[64] L. Williams, **Pyramidal parametrics**, in Peter Tanner (ed.), Computer Graphics (SIGGRAPH 83 Conference Proceedings), volume 17, p1-11, 1983

[65] P. Wonka, M. Wimmer, F. Sillion, W. Ribarsky, **Instant architecture**, ACM Transactions On Graphics, volume 22, issue 3, p669-677, 2003

[66] D. Xue, R. Crawfis, **Efficient splatting using modern graphics hardware**, Journal of Graphics Tools, 8(3):1-21, 2004

[67] G. D. Yngve, J. F. O'Brien, J. K. Hodgins, **Animating explosions**, Proceedings of the 27th annual conference on Computer graphics and interactive techniques, p29-36, July 2000