Efficient Multi-Bounce Lightmap Creation Using GPU Forward Mapping

Doctoral Dissertation submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Randolf Schärfig

under the supervision of

Prof. Kai Hormann

co-supervised by

Prof. Marc Stamminger

October 2016

Dissertation Committee

Prof. Marc Langheinrich	Università della Svizzera Italiana, Switzerland
Prof. Evanthia Papadopoulou	Università della Svizzera Italiana, Switzerland
Prof. Marco Tarini	Università degli Studi dell'Insubria, Varese, Italy
Prof. Matthias Zwicker	Universität Bern, Switzerland

Dissertation accepted on 18 October 2016

Prof. Kai Hormann
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Marc Stamminger
Research Co-Advisor
Universität Erlangen, Germany

Prof. Michael Bronstein
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Randolf Schärfig
Lugano, 18 October 2016

Abstract

Computer graphics can nowadays produce images in realtime that are hard to distinguish from photos of a real scene. One of the most important aspects to achieve this is the interaction of light with materials in the virtual scene. The lighting computation can be separated in two different parts. The first part is concerned with the direct illumination that is applied to all surfaces lit by a light source; algorithms related to this have been greatly improved over the last decades and together with the improvements of the graphics hardware can now produce realistic effects. The second aspect is about the indirect illumination which describes the multiple reflections of light from each surface. In reality, light that hits a surface is never fully absorbed, but instead reflected back into the scene. And even this reflected light is then reflected again and again until its energy is depleted. These multiple reflections make indirect illumination very computationally expensive. The first problem regarding indirect illumination is therefore, how it can be simplified to compute it faster.

Another question concerning indirect illumination is, where to compute it. It can either be computed in the fixed image that is created when rendering the scene or it can be stored in a light map. The drawback of the first approach is, that the results need to be recomputed for every frame in which the camera changed. The second approach, on the other hand, is already used for a long time. Once a static scene has been set up, the lighting situation is computed regardless of the time it takes and the result is then stored into a light map. This is a texture atlas for the scene in which each surface point in the virtual scene has exactly one surface point in the 2D texture atlas. When displaying the scene with this approach, the indirect illumination does not need to be recomputed, but is simply sampled from the light map.

The main contribution of this thesis is the development of a technique that computes the indirect illumination solution for a scene at interactive rates and stores the result into a light atlas for visualizing it. To achieve this, we overcome two main obstacles.

First, we need to be able to quickly project data from any given camera configuration into the parts of the texture that are currently used for visualizing the 3D scene. Since our approach for computing and storing indirect illumination requires a huge amount of these projections, it needs to be as fast as possible. Therefore, we introduce a technique that does this projection entirely on the graphics card with a single draw call.

Second, the reflections of light into the scene need to be computed quickly. Therefore, we separate the computation into two steps, one that quickly approximates the spreading of the light into the scene and a second one that computes the visually smooth final result using the aforementioned projection technique.

The final technique computes the indirect illumination at interactive rates even for big scenes. It is furthermore very flexible to let the user choose between high quality results or fast computations. This allows the method to be used for quickly editing the lighting situation with high speed previews and then computing the final result in perfect quality at still interactive rates.

The technique introduced for projecting data into the texture atlas is in itself highly flexible and also allows for fast painting onto objects and projecting data onto it, considering all perspective distortions and self-occlusions.

Acknowledgements

First and foremost, I would like thank my advisor Prof. Kai Hormann for his support and guidance through the PhD-process. I am very grateful for his advice, his constructive criticism, and his encouragement and sincerity.

Furthermore, I would like to thank my external advisor Prof. Marc Stamminger from the University of Erlangen for his suggestions, insights and comments about my research.

I would also like to thank my internal committee members Prof. Evanthia Papadopoulou and Prof. Marc Langheinrich as well as my external committee members Prof. Marco Tarini and Prof. Matthias Zwicker for reading and evaluating this thesis.

I am also grateful to my colleagues here at USI with whom I had a great working relationship and many interesting talks. Especially Sandeep Kumar Dey and Dmitry Anisimov became very close friends during the time I spent here and helped me a lot in different situations. You guys made my life here very enjoyable even in days and weeks of extreme stress. I would also like to mention Ämin and Ali for the interesting talks we had about different topics.

Most certainly, I am indebted to my family, who laid the foundation for all I have and will accomplish in my life.

And most importantly, thank you Lilla for being part of my life and making me enjoy every minute of it to the fullest. You always gave me strength to work hard on this and lifted my spirit with your positive attitude towards everything. You managed to motivate me even during the hardest times.

Contents

Figures

Tables

Chapter 1

Introduction

Computer generated images for movies are nowadays often indistinguishable from real footage. Since there is no time limit for the computation of these scenes, everything is computed with very high precision to model reality as best as possible. Here the main focus is on following light rays or photons into the scene, which makes these techniques slow and computationally expensive.

But also interactive computer graphics has advanced to the point where it is possible to render scenes, objects, plants and characters in a way that makes the result hard to distinguish from a photography. All of the effects necessary to produce these results – e.g. dynamic lighting, subsurface and atmospheric scattering, depth of field, lens flares, high dynamic range lighting and more – are computed in real-time and react to current lighting conditions, material properties and other dynamic effects. Even dynamic reflections on arbitrary rough surfaces can be handled convincingly. Figure 1.1 shows examples from current computer games, rendered in realtime using the above-mentioned effects.

The techniques used in modern computer games include high detail textures giving materials a very natural look, tone mapping that helps to make the colors and contrast more realistic, and image space reflections. Shaders allow each material to have entirely different visual appearance and can be used to approximate physical effects like subsurface scattering, diffuse and specular lighting effects for different BRDFs and so on. Normal mapping makes low-tessellated surfaces look like they have different roughness or impurities on them. Due to some simplifications of the physically-based formulas and the enormous parallel computational power of GPUs all of this can be handled in realtime and yet it gives incredibly realistic results as Figure 1.1 demonstrates.

As long as the virtual scene is rendered in the same style making every piece fit into the whole image, the user is inclined to take it for real. This is an important factor that makes it different to CG-effects in movies, where the rendered object needs to fit into the environment captured by the camera, because otherwise the viewer would immediately

Figure 1.1. The left image shows a realistic looking scene with atmospheric light scattering from the game *Crysis 3* rendered in realtime. The middle image shows screen-space reflections, while the last image shows a scene from *Call of Duty: Advanced Warfare*, demonstrating the realism of scanned human actors rendered in realtime with *depth of field*.

and intuitively spot the rendered objects which would destroy the immersion that a movie is aiming at.

Although all the above-mentioned techniques have been optimized for high speed computation as well as best possible looks, indirect illumination is still in a very simplistic state. In most cases it is modelled by artists or precomputed and the result is then used in the dynamic rendering process. This holds even for the *Unreal Development Kit* [16], which is considered the most advanced engine currently available. Although that gives good results, it restricts the rendering to rather static lighting conditions, while dynamic lights do not affect the scene ambience. Other approaches simplify the computation, allowing it to be computed at interactive rates, but the light distribution is very restricted. This approach becomes more and more popular recently and gives good results in generating a certain ambient, but since it is strictly bound in the number of possible light bounces, it does not correctly illuminate scenes with complicated structures lit by only a few light sources. So if the user, for example, opens a door separating a brightly lit room from a completely dark corridor, the direct light – if any is directed towards the door – would shine into the corridor and the door frames would create shadows. But the surrounding of the door would still stay entirely dark. The effect is even worse when the light inside the room is not directed towards the door. Then the corridor would stay entirely unchanged no matter if the door is opened or closed.

Therefore we propose a novel idea for computing indirect illumination that can be computed very fast and provides very convincing results. It uses a forward mapping approach that we introduce to create a smooth high-resolution *light atlas* for the scene.

The light atlas allows us to use this results as long as the lighting computation in the scene does not change, resulting in very high rendering times. It allows for a fast recomputation of the lighting in case of only a few lights moving inside the scene.

Our proposed technique aims mainly at static scenes with mostly static light sources, yet allowing for moving objects and light sources, too. The technique is also highly scalable allowing to replace the real-time computation of the results by a more time consuming computation and therefore more precise result.

1.1 Research questions

The main goal of this thesis is to create an algorithm that can quickly and precisely compute the indirect illumination solution for a scene and store the results smoothly and visually appealing. To achieve this goal, we have to answer two main questions first:

RQ1: How can indirect illumination be computed efficiently and flexible on the GPU?

RQ2: How can the computation be simplified and speeded up without loss of physical correctness?

There exist many techniques for handling these questions. Most of them fall into the categories of being either very fast and improving the overall visual quality of the computed scene but being not even close to the correct solution, while other methods compute the full physically-based solution at the cost of hours of computation time. Now we need to answer the question:

RQ3: How can the data be stored efficiently for visualizing the scene of interest?

To answer this question, I decided to store the lighting solution within the texture atlas of the scene for which we compute the indirect illumination. This lightmap approach is already very established, but here we present a physically correct and fast method to create it. To that end we need to figure out:

RQ4: How to efficiently project data from the scene into its texture atlas?

While texturing is standard nowadays for all kinds of graphical applications, the mapping of the object into the texture atlas is usually computed once and the texture is created accordingly. When projecting data from an arbitrary view onto an object, it will most likely overlap multiple seams. A seam is created by a mesh that is continuous in 3D, but separated into multiple patches within the texture atlas. That gives rise to the question:

RQ5: How can we project contiguously from a continuous mesh surface into the different related parts of the texture atlas?

When all the above questions are answered we need to ask the final questions:

RQ6: How to compute the illumination and then project the result into the texture atlas?

RQ7: How to create smooth solution with soft shadows?

1.2 Contributions

We present a novel technique that creates visually appealing indirect illumination, which is flexible enough to switch between high quality results that take a couple of seconds to compute but can be compared with state-of-the-art results and very fast previews that can be computed in almost realtime. Despite the very low quality of these previews, they still give a very good approximation of the overall lighting condition within a scene, thus providing artists with fast feedback during the light editing phase. The contributions in more detail are:

- We present an approach to project data from the surface of a given scene into its texture atlas. The technique is very flexible and can be adapted and optimized towards different goals. It is easy to implement into an existing rendering engine and takes full advantage of the GPU's extreme parallel computational power, making it extremely fast. In contrast to most other techniques, our solution is not fragment-bound and works efficiently for multiple high-resolution texture atlases.
- To implement the projection technique, the main obstacle to overcome is the presence of seams on the 3D-mesh and to correctly project the data over them. In this thesis we present three approaches to handle that without much overhead in both memory and computational time.
- We present a technique for painting onto a 3D-mesh as a proof of concept of the above mentioned projection technique.
- Using the above technique in combination with a many-light approach we present a method that can efficiently compute and store indirect illumination for a given scene at interactive rates. It is almost entirely implemented on the GPU to guarantee the short computational time and computes enough light bounces to come very close to the real solution for that scene.
- The method can also quickly recompute the entire solution for a scene in case a single light changes. Since our technique is deterministic in contrast to stochastical methods like photonmapping, we can simply undo the effect of a single light and then recompute it with its new configuration. This restricts the computation to only those parts of the scene, where the change in the lighting situation actually takes effect.
- The technique is very flexible and can compute physically realistic images with smooth shadows in seconds as well as previews in realtime. While these previews have harder shadows and do not look as visually appealing as the results created by the longer processing, the physical correctness is preserved. This makes the preview optimal for editing the lighting situation in virtual scenes.

1.3 Outline of the thesis

We start by giving an overview of the general physics of light in Chapter 2. This chapter also descibes the difference between the computation of direct and indirect illumination in virtual scenes. We end that chapter with an overview of the state of the art in indirect illumination computation.

We begin Chapter 3 by describing the history of the graphics hardware as well as the APIs that allow for programming them. Based on that we show how this led to different techniques that allow drawing onto or projecting data into the texture atlas of a virtual object and how these techniques became much more effective with the advancements in the underlying hardware.

In Chapter 4 we describe the idea of forward mapping that allows us to project data from screen space into the parts of an objects texture atlas visible on the screen. This mapping is not fragment-bound since it draws only into the necessary parts of the – possibly many different – texture atlases. This allows for fast computation of results even for multiple high-resolution texture atlases.

In Chapter 5 we present an application that uses forward mapping to allow artists to view a scene or an object on screen and then paint directly into the texture atlases of this object or scene.

In Chapter 6 we introduce a new approach to compute indirect illumination within a scene that strongly depends on the forward mapping to create and store the final results. We explain our multi-light approach for distributing the light and how we adapted the forward-mapping approach to suit the needs of this application.

We conclude the thesis with Chapter 7 by putting the current work into the context of the field and by describing how the research questions are answered to achieve the presented work. We furthermore give an outlook on how the work can be improved and optimized further.

1.4 Publications

The work presented here led to two publications. The first paper is a proof of concept for the efficiency of the forward mapping approach, using mesh painting as a demonstration. The second paper describes our approach to the problem of efficiently computing and storing indirect illumination results.

- Chapter 5 is based on the forward mapping technique that was presented at the VMV 2010 conference and published in the proceedings *Vision, Modeling & Visualization* [47] with the title “Hardware Accelerated 3D Mesh Painting”.

- Chapter 6 is based on the work that was published in the journal *Computers and Graphics* [48] with the title “Creating Light Atlases with Multi-bounce Indirect Illumination”.

Chapter 2

Basics of Light and Lighting

In this chapter we give an overview of the physics of light and its interaction with objects in Section 2.1 and the perception of light and color in Section 2.2. With these basics we then describe to mathematical descriptions that capture this light interaction and can be used to model lighting in virtual scenes in Section 2.3.

In Section 2.4 we describe different models for direct lighting and then go into details about indirect illumination and why these two models need to be separated in Section 2.5. In Section 2.6 we describe the current state of the art in computing global lighting conditions with close to realtime frame rates and also describe techniques aiming for perfect results, where the computational time is irrelevant.

2.1 Physical properties of light

Let us start by describing how light interacts with surfaces and capturing devices as well as the human eye. The lighting condition in an arbitrary surrounding is given by the photons that are created and emitted from any direct light source and are then reflected and refracted from any surface – and partially even by atmospheric effects like fog – in the scene. What we perceive visually is the combination of all of these interactions with the photons that end up in our eye.

With our eyes or any capturing device like e.g. a camera we see the world as a sum of the photons, that are ultimately reflected into the visual sensor. There the final image is created as a measurement of the number of photons giving the brightness in every perceived point and the wavelength giving the color of that point.

The interaction of the photons with the sensor are described in Section 2.2. The brightness of a point on an object or light source is given by the number of photons that arrive at the sensor from this given point (for more details see Section 2.2.1) and is measured in Lumen [Lm].

The color of an object is given by the wavelength of the photons being reflected by this objects surface properties. Each photon is an electro-magnetic wave with par-

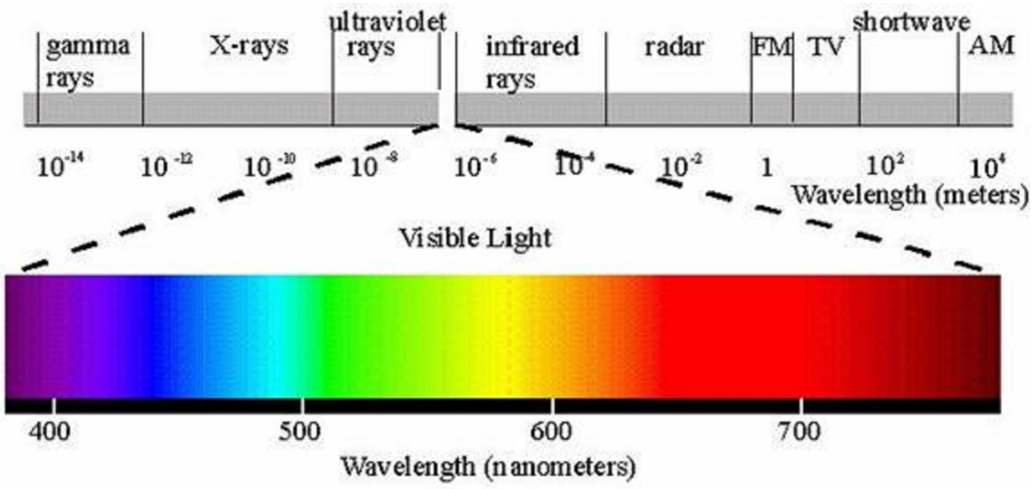


Figure 2.1. The lightspectrum going from blue on the left side (short wavelength) to red on the right side (long wavelength). Note that the visible part is just a small fraction of the continuous electro-magnetic spectrum.[1]

ticle properties moving with the speed of light c in a certain direction with a given wavelength λ related to the photon's energy E through $E = \frac{hc}{\lambda}$, where h is the Planck constant. The photon's wavelength is what we perceive as color. The continuous color-spectrum is depicted in Figure 2.1.

The surface of an object might reflect photons with one wavelength while absorbing photons with another one. The wavelength of the photons that get reflected, determine the color of the object (more details about that can be found in Section 2.2 and the mathematical model that describes the light interaction in Section 2.3).

In the absence of extraordinary strong gravitational forces, a photon will travel on a ray from its point of origin, until it interacts with the surface of an object, at which point the direction will change – that is, if the photon does not get absorbed – and it will travel along the new ray.

2.2 Perception of light

Human eyes perceive light in two different ways. In dark environments the eyes rely on the rod cells, that react to light around 500 nm wavelength (blue-green). Since all rod cells react to this same wavelength, they do not allow us to differentiate colors, but due to the high concentration of Rhodopsin, they enable us to see in dark environments, since they are around 1000 times more sensitive than the cone cells, that allow us to perceive colors.

[1]Image taken from: `https://en.wikipedia.org/wiki/Light/media/File:EM_spectrum.svg`

For bright environments the eye possesses three different types of cone cells, one type for each of the colors red (500-700 nm), green(450-630 nm) and blue (400-500 nm). They use slightly different variations of the molecule Photopsin, that reacts to different wavelengths. Incoming photons of a certain wavelength therefore create a reaction in the corresponding cells. The signal created by the three cone cells is then combined by other cells through the step-wise adaptive blending $r + g + ((r + b) + (g + b))$ into a single color signal that is send to the brain together with a signal encoding the overall brightness.

In monitors the same composition method is used to create the different colors on displays. Each pixel has a mask for the three colors red, green and blue which are used to dim the corresponding portion of the white background illumination. Each mask is controlled by a byte that sets the amount of light that can pass through that color-channel with 1 meaning all light can pass and 0 that meaning the mask absorbs all the amount of the specific color. This way we end up having a 24-bit color range, leading to $2^{24} = 16777216$ different colors (Note that it also contains the same color in different brightness shades).

Since the number of photons define the brightness, the mask does not only control the color itself, but also the brightness that is perceived, ranging from the brightness of the background illumination to the amount that can still pass through the color-mask set to full absorption. In the optimal case this would be completely black, meaning no photons passed, but due to technical restrictions this value will be higher than that.

2.2.1 High-Dynamic-Range

As mentioned before, the brightness is measured in Lumen, where 1 Lm at a wavelength of 555 nm is equivalent to a photon rate of 4.11×10^{15} photons per second.

Each photon creates a single chemical reaction, triggering a signal to the brain, and the signal strength is given by the amount of reactions within a given time. Let aside the color-perception described in Section 2.2 and looking at just one arbitrary receptor-cell, the brightness that can be perceived is given by the maximum number of signals that the cell can create in a given time-interval for a single measurement.

Since the brightness varies strongly between bright daylight and a night with nothing but starlight, the human eye adapts to the current brightness level through changes in the pupil dilation – adapting the number of photons entering the eye – and by changing the sensitivity of the photo-receptor-cells – changing the signal strength from the retina to the brain.

Since the human eye is additionally made up of rod and cone cells that operate at different brightness-levels, the overall visual range is further increased. This extreme range is called *High-Dynamic-Range*(HDR) in computer graphics due to the strong contrast of the brightness range of the monitor (1:1000) which is now usually labelled as being Low-Dynamic-Range.

Currently the problem is solved by compressing the brightness range of images with a *tone mapping*. This technique applies non-linear curves on the input colors to compute output colors that match the contrast and luminosity as best as possible over the whole image.

With HDR-monitors becoming widely available, the correct computation of indirect lighting that illuminates every part of the environment – and be the quantity ever so small – becomes more and more important. In contrast to the Low-Dynamic-Range screens, where an approximation of the brightness is enough to give a feeling of realism, since the image is compressed to 8 bit per color channel with tone-mapping, with HDR-screens the brightness needs to be computed with high fidelity to achieve the sense of looking at a realistic scene.

2.3 Lighting in virtual environments

In computer graphics we deal with virtual scenes or objects mainly given as triangular meshes. Other representations like point-clouds and implicit functions describing the surface have also been experimented with, but triangle meshes proofed to be the most efficient (see Chapter 3). To give these scenes and objects a natural look when being rendered, they need to be correctly lit, since their interaction with light is what we perceive (see Section 2.2).

To achieve this, we need to understand the physical basis of lighting to compute these effects in the virtual environments.

As mentioned before, the brightness is given by the number of photons. These photons originate from a light source (*direct light*), but might also have bounced off of surfaces in the scene (*indirect illumination*, more about this in Section 2.5). Figure 2.2 shows a comparison. The further away we get from any light source (direct or indirect), the fewer photons we register on the same area. This number falls off with a factor of $\frac{1}{r^2}$, where r is the distance to the object, since the photons are distributed on the surface of the sphere with the radius r, which grows by r^2.

Now that we have covered the basics of the light origin, we need to look at the interaction of the photons with objects. Note that the following three types of interactions describe what can happen to a single photon. In reality one surface can – and most likely will – show different types of interaction. That is, each photon interacts in a single random way with the surface, but the probability of the exact interaction of the photon is given per every material. Furthermore, these probability distribution differs for every wavelength, leading to this materials reflection coefficients ρ, that describe the probability distribution for each color:

- **Absorption:** its energy is transformed into heat and no further interactions can occur with this photon.
- **Translucency:** the photon passes right through the object, changing its trajectory due to the refraction index of the material.

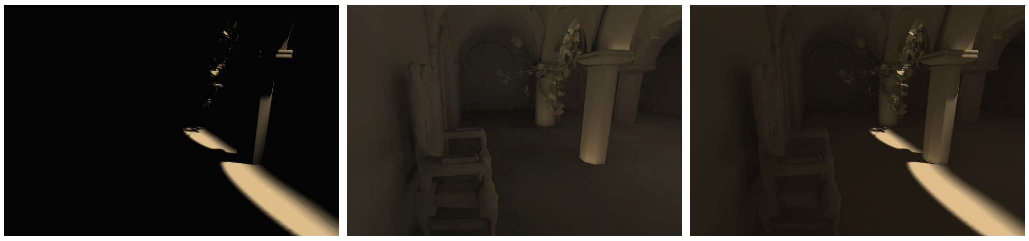


Figure 2.2. Difference between direct light (left) and indirect illumination (middle). The final solution is composed of both parts (right).[2]

- **Reflections:** Materials reflect light in different directions with different intensities due to microscopic inter-reflections on rough surfaces and other effects. Each single reflection follows the law of reflection (specular), but for rough or weakly translucent surfaces a number of inter-reflections occur leading to diffuse reflections or subsurface scattering respectively.
  - **Specular:** the photon is reflected along the reflection vector. Details about this can be found in Section 2.3.2.
  - **Diffuse:** the photon is reflected into a random direction away from the surface. Details about this are described in Section 2.3.1.
  - **Subsurface-Scattering:** the photon enters the material and gets reflected many times before leaving the surface in a random direction close to the entry point.

The interactions that make up most parts for most materials are the diffuse and specular reflection as well as the absorption of the incoming light. Therefore we look into this more closely now. We will start with the diffusion term in Section 2.3.1 and then look into the specular reflection in Section 2.3.2.

2.3.1 Diffuse reflection

The diffuse reflection is created by light being reflected in the top layer of a surface one or more times. Due to the possibly high amount of inter-reflections, the reflected light can be assumed to be reflected equally in all directions away from the surface, although the exact BRDF might be slightly different. To determine the brightness of a fully diffuse surface it is therefore enough to compute the incoming radiosity, since the reflected light is in all directions a certain fraction of this value.

We describe the diffusion term now by looking at an exemplary interaction of light with a plane surface. First we look at a light ray R with a certain width w. The amount n of photons in the cross-sectional area of this ray is given by the brightness of the light source from which it originates (see Figure 2.3). For simplicity we ignore the reflection

[2]Image taken from: `https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/`

coefficient of the plane material for now and assume that all photons are reflected diffuse ($\rho_s = 0$ and $\rho_d = 1$). The perceived brightness of a surface area A is then given as a fraction of the amount of photons that impact in A.

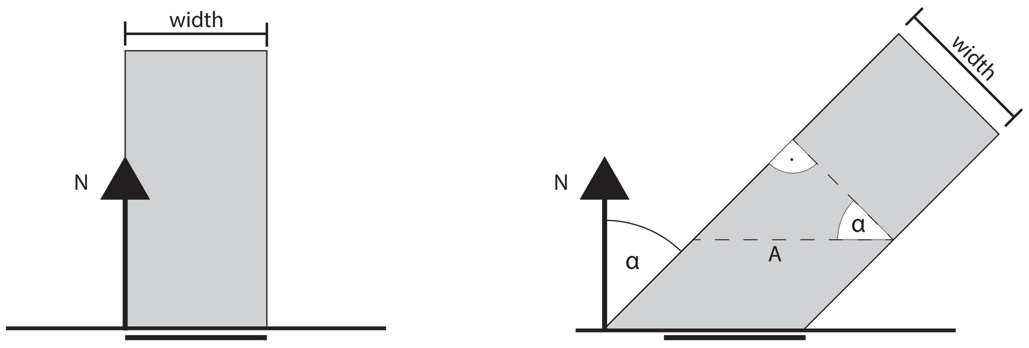


Figure 2.3. When the light shines perpendicular onto a surface, the lit area has the same size of the incident light ray (left). If the light hits the surface with an angle α bigger then zero, the lit area becomes bigger then the with of the light ray. Therefore the amount of light energy is distributed over a bigger area which makes each point within this area darker, the bigger the angle α gets.

If the light ray intersects the plane straight from the top, the illuminated area A has exactly the width w as seen on the left side of Figure 2.3. Therefore all n photons contained in the cross-section of R impact the area A. If the incident angle α between the normal N and the light ray R is bigger then 0°, the area will be bigger then w. Using trigonometry (image on the right side of Figure 2.3), we see that we have a 90° triangle given by the light ray R. Since the sum of all angles in a triangle sum up to 180° and we have a 90° angle between R and w, we know that $\alpha + \beta = 90°$. The angle α occurs also in the triangle where w is adjacent to α.

The lit area A' is now given as $\frac{w}{\cos(\alpha)}$ since it is the hypotenuse of the triangle with w adjacent to it. Let the number of photons in the cross-section of R be n which leads to a photon density of $\frac{n}{w}$. With R having an angle of α with the normal, the photon density now becomes $\frac{n}{\frac{w}{\cos(\alpha)}} = n\cos(\alpha)\frac{1}{w}$. Since the width is infinitesimally small and the number n of photons is set for the light source, we see that the amount of photons hitting the surface can be computed by multiplying the brightness of the light source with the cosine of the angle between the normal N of the surface and the light-ray R.

2.3.2 Specular reflection

Perfect specular reflection describes the effect that is given by a mirror. A light ray R hits a surface with a certain incoming angle γ measured between R and the surface normal N and are reflected of the surface in the direction with the same angle to the normal in the other direction within the plane created by the vectors R and N (see Figure 2.5 left).

The explanation for that is the electro-magnetic-wave-character of light. When the light wave hits the object, the Huygens-principal creates a wave-front with the same angle to the surface, as the incoming angle but in the opposite direction.

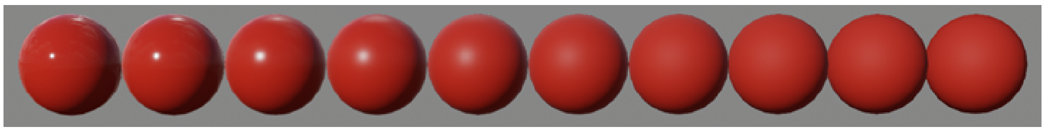


Figure 2.4. Here different values for the roughness of a material are shown. With growing roughness, the reflection coefficients change from high specular and low diffuse to low specular and high diffuse (from left to right).[3]

The reflection vector can be computed as $R' = 2\langle N, R\rangle N - R$. An approximation for that is to use the half-vector as suggested by Blinn [3]; it uses the fact, that $\frac{R}{||R||} + \frac{R'}{||R'||}$ will create a vector that is completely aligned with N. If the vector C from the point to the camera is equal to R', the reflection is perfect and the camera will receive the full reflected intensity. If the vector from the point to the camera deviates, the intensity should decrease. It can be computed as $\langle N, (R+C)\rangle^{\rho_s 4}$ as the brightness given by the specular reflection. This is 1 for $C = R'$, while the value quickly decreases the further C deviates from R'. Computing this is faster and gives results very close to using the ones computed with the true reflection vector. Figure 2.4 shows examples of different values for the specularity of an object.

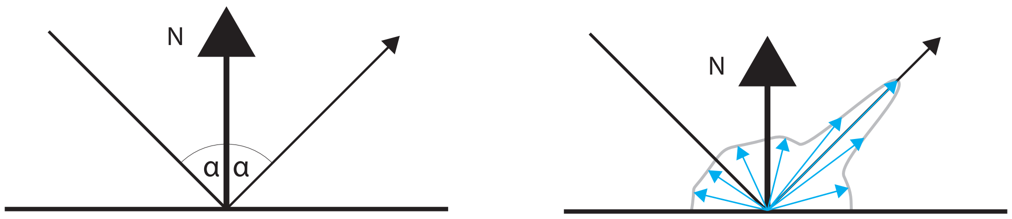


Figure 2.5. The incident angle is equal to the reflected angle for ideal specular reflection (left). For different materials the light flux is highest around the reflection vector and falls off quickly with increasing divergence from it (lenght of blue arrows on the right).

[3]Image taken from: `https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/PhysicallyBased/`

2.4 Direct illumination

For efficient rendering we differentiate between direct lighting and indirect illumination (see Section 2.5) which is visualized in Figure 2.2. Direct lighting – also refereed to as local illumination – describes the effects created by light being emitted from a direct light source, hitting at most one surface and then being reflected directly into the camera where they contribute to the color and intensity of a single pixel. In the beginning of computer graphics, objects were not illuminated at all and were simply assigned a fixed color. Then simplified lighting models were created, that approximated realistic lighting effects fast enough to be computed. The first of these techniques was flat-shading, where the color of a triangle was determined by the cosine of the angle between its normal and the light-ray. In 1971 Henri Gouraud [18] improved this by computing the lighting at every vertex of the triangle and interpolate the resulting colors over the triangles surface. Since this still created artefact when dealing with specular reflections, Bui Tuong Phong [41] proposed in 1975 to interpolate the normals across the triangles surface and then compute the lighting per visible pixel with this normals.

These and other techniques are based on the Phong lighting model [41], that takes into account the self-emitted light of an object, approximates the indirect illumination by a simple constant color (omnidirectional) and then sums up the effects of all direct lights in the scene. This is done by splitting them into a diffuse and a specular component. It furthermore takes the materials reflection coefficients into account, which describe how much of the incoming light is absorbed. The whole computation as well as the reflection coefficients are usually given in the RGB-color-space (see Section 2.2).

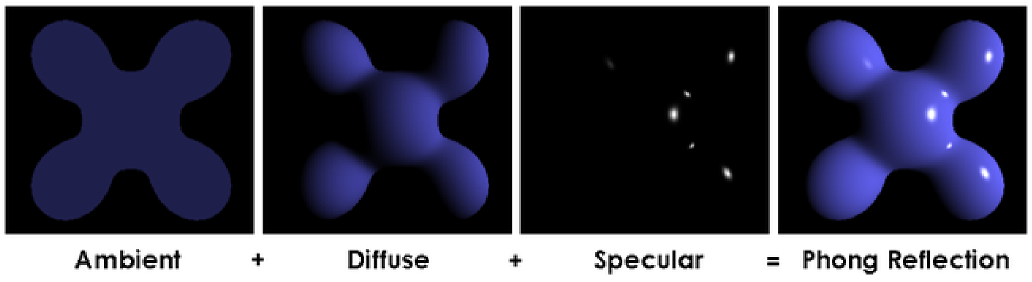


Figure 2.6. The different components for Phong-shaded objects. In this shading, the global lighting situation is simplified to be a constant color.[4]

Since all lighting computations are based on this equation we describe it in detail here:

$$R = E + \rho_a A + \sum_{i=0}^{L} I_i(\rho_d D_i + \rho_s S_i)V(i) \tag{2.1}$$

[4]Image taken from: `https://en.wikipedia.org/wiki/Phong_shading/media/File:Phong_components_version_4.png`

where R is the color of the current pixel, given as the sum of the self-emitted light E, the overall ambient color A and the sum over all L light sources with intensities I_i interacting with this point. The interaction between the light and the surface itself is separated into a diffuse interaction D_i describing the reflection of the light equally in all directions (see Section 2.3.1) and a specular term S_i that describes the light reflection according to the law of reflection (see Section 2.3.2).

The amount of light reflected from the surface point is a property of the material at that position and given by the reflection coefficients ρ which are material parameters given in the RGB-color-model describing which light waves are absorbed and which get reflected. The parameters are usually given for ambient ρ_a and diffuse ρ_d reflections. The specular reflection coefficient ρ_s should have the same values across the colors for dielectric materials since they reflect all wavelength equally, while they are material color dependent for other materials such as metals.

These terms are evaluated as

$$D_i = \langle L_i, N \rangle \tag{2.2}$$

where L_i is the normalized vector from the point of interaction to the light source i and N is the normalized normal at the point of interaction. The specular term is evaluated as

$$S_i = (\langle 2\langle L_i, N\rangle N - L_i, C\rangle)^s \tag{2.3}$$

with C being the vector from the point of intersection towards the camera. The exponent s is a surface property describing the shininess of the surface. As mentioned in Section 2.3.2 this can be simplified by the use of the half vector leading to similar results. An example that separates the results of the different terms is visualized in Figure 2.6.

The function $V(l,i)$ describes the visibility of point i from the light source l and determines if the point is shadowed ($V(l,i) = 1$) or lit ($V(l,i) = 0$). In modern interactive visualizers this is determined through shadow mapping, which is hardware accelerated on all modern 3D-accelerators. Another technique that was quite popular for a while was to use the stencil-test for shadow-volumes, but shadow mapping has proven to be superior and is nowadays the standard.

2.5 Indirect illumination

Indirect illumination in contrast to local lighting as described in Section 2.4 describes the illumination created by light that was reflected off of at least two surfaces. When light hits a surface, it can be refracted leaving the object at another point than where it entered the object. Another possibility is that the energy is absorbed by the material it hit; in this case the photon's energy is transformed into heat. The third case is, that the light bounces off the surface either in a random direction (diffuse reflection) or it is reflected (specular).

In real world materials all of the above might happen at once. A good example for that are thin leafs on a tree. They let a fraction of the light through (refraction), absorb another part (heat and photo-synthesis) and reflect the rest both diffusely and specularly.

Even black materials reflect at least a small fraction of the incoming light which makes indirect illumination hard to handle. This is due to the fact, that light bounces off of every surface, and though the intensity might have been reduced by a huge amount due to the reflection-coefficient of the surface material, it will now serve as a weak light source itself, illuminating objects in its vicinity.

This effect occurs wherever light hits a surface, and every photon is reflected until it is absorbed. To compute the global lighting means therefore to follow all possible light paths ad infinitum in contrast to direct illumination, where we just compute the lighting situation for every visible point on the screen.

In Section 2.5.1 we describe the history and basic idea of global illumination, classify different approaches to that problem in Section 2.6 and then describe state-of-the-art ideas in Section 2.7.

2.5.1 Basics for implementation

When the importance of indirect illumination in rendered scenes became obvious, the first approach of computing it was through the radiosity equation formulated by [5]. The authors' radiosity method describes an energy equilibrium (see Eq. (2.6)) within a closed surface assuming that all emissions and reflections are ideal diffuse. The authors further partition the whole scene into a finite number of n small patches. These patches are associated with a position x_i – the center of patch i – and the normal of this point. With this assumption and simplification they reformulate the rendering equation introduced by [27] to the following:

$$B_i = E_i + p_i \sum_{j=0}^{n} B_j F_{ij} \tag{2.4}$$

where B_i is the radiosity of the current patch i (the amount of light leaving), E_i is its self emission, p_i is the percentage of light that gets reflected from patch i and the sum over all the n patches that make up the environment takes into account the amount of light reaching the current patch through light that is reflected or emitted from other patches in the scene. Therefore B_j describes the amount of light that leaves the patch j and the form-factor F_{ij} describes the geometric relationship between the patches i and j and therefore the percentage of the light leaving patch j and reaching patch i. The F_{ij} for patches with areas A_i and A_j is defined as:

$$F_{ij} = \frac{1}{A_i} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} V_{ij} \tag{2.5}$$

were the angles θ_i and θ_j in the cosine terms describe the angle between the direct line between the two patches i and j and the corresponding normal vector for these patches, and r is the distance between the patch i and patch j.

In case of partial occlusion between two patches the authors suggest to subdivide them to get a better result. The authors also mention that for a finite area the form-factor is equivalent to the fraction of a circle covered by the projection of the area first onto the hemisphere and then orthographically further down onto the circle that is the basis of this hemisphere. By subdividing the hemisphere into smaller patches itself and computing the projection of other patches on them and only considering the closest patch that gets projected to each element, it also takes care of occlusion. The solution found for these small partitions of the hemisphere – called delta form-factors by the authors – for one distant patch can then be summed up to get the form-factor between the current patch and this distant one.

With these form-factors and the initial emission values $E = (E_1, \dots, E_n)$ for all patches the authors solve the following equation system to receive the final radiosity $B = (B_1, \dots, B_n)$ for all the patches in the scene,

$$\begin{pmatrix} 1-p_1F_{11} & -p_1F_{12} & \cdots & -p_1F_{1n} \\ -p_2F_{21} & 1-p_2F_{22} & \cdots & -p_2F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -p_nF_{n1} & -p_nF_{n2} & \cdots & 1-p_nF_{nn} \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix} \tag{2.6}$$

The authors use scenes made up of polygons that are also used as the patches. After they have solved the linear equation system and get the radiosity of each patch, they render the scene using bilinear interpolation on the vertices to get a color out of the polygons/patches surrounding this vertex. This way the resulting image looks smooth, since there are no sudden jumps in color or brightness. We described this technique in much detail, since it introduces many tricks that are standard for computing radiosity nowadays.

The idea of solving the equation system to compute the final radiance for the scene is the basis for modern shooting or gathering techniques. These techniques take advantage of the fact that in the first step of the algorithm only the light sources possess energy that needs to be projected into the scene, and even in the next few steps there are some elements that do not contribute to the light distribution because at this stage they still did not receive any light. The first approach to use this fact was only computing the lines in the matrix that have a value greater then zero on the right side. Later, shooting and gathering approaches were introduced and even ported onto the graphics card, as described in the work of [7], for example. Their technique stores for all patches in the scene the energy that they distribute in a sorted list where elements with the highest energy come first. Then the elements shoot their energy into the scene in the given order and are set to have zero energy afterwards. This process permanently introduces new elements into the list and updates elements that are already stored, since their energy level increases. This way the whole list is processed until the next element in the list has a radiosity that lies under a certain threshold, which terminates the computation, since the energy is distributed well enough.

Since the field of indirect illumination is large and diverse, we start this partitioning the field into smaller sub-fields (e.g. interactive vs. non-interactive) in Section 2.6. This is important, since the computational cost of indirect illumination is very high, and one needs to choose between perfect results and real-time computation. For light-editing a trade-off between these two might also be desirable. The comparison of existing techniques only makes sense within one of these classes. We then give an overview of the techniques that are state of the art in each field in Section 2.7, concentrating on the field of this work being close to real-time computation of the indirect illumination solution.

2.6 Classification of indirect illumination techniques

Since the beginning of indirect illumination computation, different approaches with deviating aims were developed for its computation, so we first distinguish between these varying techniques and create clusters. The first approaches date back into the 1970ies and since then one goal was to make the computation of indirect illumination ever more precise. As computers got faster and the results for diffuse surfaces led to good results, the focus shifted from making it more and more realistic to making indirect illumination computation also faster, since achieving interactive frame rates became a realistic goal. When the first 3D-accelerators were introduced in 1996 and newer versions supported hardware-accelerated Transforming & Lighting in 1999, researchers focused on this goal. The research was now focused on the different path of either increasing the realism of the results by taking into account more and more physical interactions of materials with light into account while others the quality and instead tried to compute visually appealing but not physically correct results in real-time.

If we take a look at the varying techniques that exist today it is convenient to separate them into different classes. There are techniques that compute indirect illumination almost in real-time – something around 30 *frames per second* (FPS). These ideas make strong simplifications or compute only one or two light bounces to achieve this. Within this class of techniques another criterion for further distinguishing is the number of bounces the algorithms can compute. Another class of techniques do not aim for high framerates and instead aim at increasing the realism of the computed images. These techniques have to compute all light bounces – meaning they have to follow the light until its energy falls below a certain threshold.

To simplify the distinction, we concentrate on the class of real-time algorithms. Most compute only a single bounce of light and the reason for this is twofold: On the one hand, it might be sufficient in small and simple scenes to just compute one bounce while computing more would just be a waste of computing time. On the other hand, it is much faster to compute only one inter-reflection, since this is the most time consuming part of indirect lighting computation after all (even if pre-computed light transport functions are available).

So for simplicity let's say we have the following classes:

- Interactive Frame Rates
  - One bounce reflections
  - Multi bounce reflections
- Non-Interactive Frame Rates

Now that we have partitioned the existing algorithms into these groups, we can also distinguish between the following main ideas of how to scatter the light:

- Computationally
  - Discretizing the scene (Radiosity)
  - Many-light approaches
- Statistically
  - Ray-tracing
  - Photon-mapping

While the computational method via the form-factors is very time consuming, it is also very precise. It partitions the scene until the discrete parts (patches) are small enough to give good results. Modern methods also partition the scene depending on the already computed results and their quality, to only improve the computed results where necessary, speeding up the computation time considerably. But still, doing the visibility test between each pair of surfaces is very expensive.

The other idea is to use the *Graphics Processing Unit* (GPU) to handle this visibility test via rendering the scene from the current light source's point of view. This can be done very efficiently, because it uses the graphics card for its main purpose and the extreme parallelism is optimally used. But then other problems arise, like for example getting all necessary data for a distant surface that is interacting with the current one. One reason for this is that the graphics card does not allow random access to its memory.

Another idea using the GPU is to create new light sources and use the shadow mapping algorithm implemented in modern graphics cards to do this computation as if it was done for a direct light source. This gives good results but is still quite expensive. While both techniques can be used for interactive computation, it is mainly the latter that is actually used nowadays.

The other two techniques – ray-tracing and photon-mapping – are mainly used for non-interactive computation of indirect lighting, since they are very precise but also computationally expensive. Both methods do not calculate the lighting distribution based entirely on a physical model but instead use a stochastic approach for scattering light from every point that has gathered light in a previous step.

Another differentiation that is needed is concerned with the material properties that are supported. Most techniques do not allow for glossy materials, since the appearance of the light also depends on the current viewing direction. Other techniques do compute one bounce (the last one of the indirect lighting computation) to consider glossy materials. But this almost only holds for interactive techniques, because the non-interactive techniques can easily take care of this, since the additional computational cost is irrelevant and negligible.

So if we want to distinguish the different techniques, the most realistic partitioning is:

- Interactive Frame Rates
  - One bounce reflections
    - glossy reflections
    - diffuse materials only
  - Multi bounce reflections
    - glossy reflections
    - diffuse materials only
- Non-Interactive Frame Rates (mainly Photon-mapping or ray-tracing)
  - multiple bounces
- Full Solution
  - very many bounces
  - complex material-light-interaction with
    - diffuse and glossy reflections
    - BRDF
    - subsurface scattering
    - refraction

2.7 Overview of existing techniques

There are many techniques based on the before mentioned light distribution model. We will now give an overview of the state-of-the-art techniques. We start with the historically first implementations and there direct improvements in Section 2.7.1. We then look at techniques that achieve interactive rates in Section 2.7.2 and finally describe methods that aim for photo-realistic results in Section 2.7.3.

2.7.1 Non-Interactive approaches

In this section we discuss techniques that improve the computational time by optimizing other algorithms through parallelization and/or porting them to the GPU without achieving real-time-speeds.

The first approach of computing indirect illumination was through the radiosity equation formulated by Cohen and Greenberg [5]. This radiosity method describes an energy equilibrium within a closed scene assuming that all emissions and reflections are ideally diffuse and the light distribution is computed iteratively over the discretized surface of the scene.

Then shooting and gathering approaches were introduced and even ported on the graphics card by Coombe et al. [7]. This technique uses the aforementioned approach for the coarse distribution of light within the scene, followed by a highly precise method for storing the lighting results, therefore changing the geometry of the scene and creating a higher tessellation. In our approach we only use the shooting method in the beginning of the light distribution and then switch to a scene texture atlas for storing the high resolution results. Therefore, our approach does neither depend on nor does it change the scene geometry. Szécsi et al. [55] propose to precompute the expensive integrals needed for indirect illumination for all but one variable, the light position, and store them in a texture atlas of the scene. During rendering, the radiance of the visible scene pixels is determined by evaluating this precomputed data for the given light positions. Although the results look very promising, the amount of memory needed is rather big, even for scenes without complex occluder configurations. Our technique can handle complex scenes using only a light atlas, which is required anyway.

Arikan et al.[2] accelerate the final gather step of global illumination algorithms through approximation of the light transport by decomposing the radiance field near a surface into separate near- and far-fields which then get approximated differently. They rely mainly on the assumption that radiance will exhibit low spatial and angular variation due to distant objects, and that the visibility test between close surfaces can be reasonably predicted by simple location- and orientation-based heuristics. They use scattered-data interpolation with spherical harmonics to represent the spatial and angular variance for the far-field, while the near-field scheme employs an aggressively simple visibility heuristic.

Other techniques [20, 23] use the extreme parallel capability of modern GPUs to accelerate ray-tracing approaches. They cluster different light rays according to their direction and then use the GPU to compute the visibility for all these bundles simultaneously. This makes the computation much faster than on the CPU, but still does not achieve interactive frame rates. Our technique creates similar quality, but in shorter time.

Luksch at al. [36] propose a technique for computing the light atlas of a scene at almost interactive speed. They partition the scene into polygons and distribute the light energy among virtual light sources, one for each polygon. In a final gathering step, they render the light atlas and collect the radiosity for each texel from the direct and virtual light sources. Although this technique is similar to ours, it is limited to two light bounces and therefore produces results with a lower quality.

2.7.2 Interactive approaches

An important technique considering indirect illumination was introduced by Greger et al. [19]. Instead of calculating the indirect lighting situation within a given scene, it rather tries to compute the illumination introduced by the scene on a static object moving through the scene. That means that the scene lighting needs to be already computed, for example by the previously described radiosity method. Under the assumption that this is given, the idea behind *irradiance volumes* (IV) is that instead of storing the amount of light leaving each surface point within the scene, it rather pre-computes for certain points in space the incoming light under certain angles. To do this, the technique first fills the bounding box of the scene with virtual spheres – the irradiance volumes – that are distributed through the scene in a way that best samples the underlying geometry and its complexity. Then each sphere samples the space around with a specific step size, resulting in the projection of the scene around the sphere on its surface with the pre-computed brightness and color. That means that every sphere is “lit” by its environment. The authors then store these color values for every sphere and use them for the objects moving within the scene. Therefore, a moving object is first associated with the closest irradiance volumes and the object’s surface color is then computed by taking into account the object’s normals and the distance to the spheres using bilinear interpolation of the color values determined. This technique is only able to compute indirect lighting for moving objects within a scene for which the indirect illumination was already computed. This means that this technique cannot handle changes of light sources within the scene and is therefore bound to statically lit environment.

Sloan et al. [52] introduce a technique that extends the idea of IV through *pre-computed radiance transfer* (PRT). The authors describe a technique that can handle the different lighting effects introduced in [19], but also takes into account inter-object reflection and shadow casting. This is done by a pre-computed spherical harmonic basis that describes how an object scatters light on itself and the surrounding space. Like in the previous idea of IV, the lighting environment is assumed to be given and being infinitely far away, which makes the illumination the computation of a cosine-weighted integral for each surface point of the object if it is convex. For concave objects the integral is multiplied by a value that describes visibility along each direction to account for self-shadowing. The authors pre-compute these values and the integrals and represent it through spherical harmonics. Through the linearity of this representation the integral of the light transport function becomes a dot product between the pre-computed environment light and the pre-computed coefficient vectors that can be handled by the graphics hardware in real-time. In the same way the technique also handles glossy surfaces, inter-reflection, and soft-shadows.

Sloan et al. [53] extend the paper [52] to account for deformable objects in a local PRT considering the inter-surface reflections and shading of a surface point by nearby surroundings. This is done by switching from the pre-computed spherical harmonics basis to the zonal harmonics basis, which allows for fast computations of rotations. So whenever a part of the object is deformed, the local basis is updated to fit this deformation. Afterwards the transformed basis is used to compute the resulting lighting for this object.

Kristensen et al. [29] presented an algorithm that is able to indirectly relight scenes in real-time. This is achieved by extending the idea of precomputed radiance transfer and introducing the idea of unstructured light clouds to account for local lighting. The indirect illumination of a static scene is precomputed for a very dense light cloud, which is then compressed to contain just a small number of lights. While rendering the scene, its indirect illumination is approximated by interpolating the precomputed radiosity of the closest lights in the light cloud. This requires a relatively high number of values stored for every vertex of the scene and can only compute the lighting at these scene vertices. Therefore, the quality of the resulting image depends strongly on the tessellation of the scene. The technique gives nice results in static scenes with moving lights, but it cannot handle scene changes or large moving objects that strongly change the scene's lighting situation. Instead, our technique uses a high resolution texture to store the indirect shadows, which allows us to handle even high frequency lighting details. Moreover, our technique does not require any further computations once the light distribution is stored, which in turn reduces the time needed for rendering every frame.

Another approach was introduced by Walter et al. [57]. The authors suggest an algorithm that can handle all kinds of light, e.g. environment maps and indirect illumination. The main idea is to simplify huge amounts of light sources by grouping them together and only to compute the illumination effects for the light sources created through this combination process. To speed up this merge and make it as accurate as possible, the authors introduce a binary light tree combined with a perceptual metric. While the former helps in finding appropriate light clusters, the latter helps to stop clustering if the introduced error exceeds a certain threshold. This way the rendering time can be reduced, while at the same time a certain quality for the final result can be guaranteed. For indirect lighting the authors simply use their algorithm – which they claim to be able to reduce hundreds of thousands virtual lights to only few hundred shadow rays – to reduce the expanse in computation. To calculate indirect illumination they create new virtual light sources wherever a direct light source enlightens a surface as described in [28].

A technique that reaches interactive results for single-bounce indirect illumination was introduced by Dachsbacher and Stamminger [9]. The authors use *reflective shadow maps* generated from the direct light source and a certain number of pixels in this shadow map act as new point light sources to illuminate the scene. This is done in the screen space of the final scene rendering, using the geometry information of each visible pixel as well as the information given for all the point light sources. Due to the nature of this technique it can only compute one-bounce indirect lighting and it does not compute the visibility. Hence, it cannot create indirect shadows, while our technique creates such soft shadows naturally. Prutkin et al. [42] suggest to speed up global illumination through reflective shadow maps by clustering the VPLs and treat them as single area lights. Lensing and Broll [32] apply reflective shadow maps, but reduce the number of computations by first clustering visible points, depending on their geometric properties and then computing only one radiosity value for all similar pixels. Dong et al. [13] compute indirect illumination through clustered visibility. VPLs are clustered into area lights and soft-shadow techniques are used to compute the illumination of visible screen pixels. The above techniques introduce colour bleeding and soft shadows, but only from those points which are directly illuminated. They are fast because they use a very low number of VPLs, which in turn might be problematic in situations where many direct lights illuminate non-overlapping parts of the scene.

Dachsbacher et al. [10] presented the idea of anti-radiance to overcome the costly visibility test when computing indirect illumination. The authors compute the indirect lighting by reformulating the rendering equation (2.6) to avoid the term that handles the visibility within the scene. Instead they distribute the light into the scene as if every patch is not occluded and then they use a second pass that reverts the energy propagation onto patches that are occluded by reducing the energy that this patches received in the first pass by the amount of light blocked.

A technique that combines the idea in [29] with the concept of light-cuts is proposed by Ritschel et al. [44]. They compute indirect lighting on glossy surfaces by storing coherent surface shadow maps which allows for fast visibility tests within a scene even with moving objects. For indirect lighting, the authors use the idea of virtual point lights and light cuts to compute the illumination distribution within the scene. Although this technique produces good results, it is useful only in scenes that are nicely illuminated by a few (one or two) light bounces, while our technique targets at scenes that have many occlusions and need a high number of light bounces to produce a realistic result.

Lehtinen et al. [31] describe a method for computing indirect illumination that is decoupled from the underlying geometry, since it only computes and stores the light transport for randomly scattered points in the scene. For later evaluating other points, they use scattered data approximation. The authors also define a hierarchy over these points and store the difference between the current and the next level. This technique is similar to ours regarding the initial light distribution, but we use a texture atlas for the final visible result to capture even small lighting details.

A different approach for indirect lighting computation is based on deep frame buffers. These techniques [17, 40, 22] aim at computing many effects of lighting – reflections, caustics, multi bounce indirect illumination – in real-time for changing light sources, but they rely not only on a static scene, but also on a fixed camera position. The algorithm introduced by [22] computes a deep frame-buffer once the camera is positioned in the scene. That allows to do all the re-lighting computations in real-time, but the pre-computation that is necessary to create the deep frame-buffer takes a long time. This restriction makes the technique useless for interactive applications where the camera is also moved.

Ritschel et al.[46] describe a technique that uses final gathering performed on the graphics card. The scene itself is partitioned into many small discs that approximate its geometry. For the final gather step the authors render the scene into many "micro-buffers" by traversing a hierarchical point-based representation of it. Each micro-buffer contains a projective mapping of the scene, and the convolution of the incident light is computed by summing up the contents of the micro-buffers. One thing to mention here is that an interactive walk-through of an indirectly lit scene needs a pre-computation through photon-mapping, since this technique only does the final gathering of the photon-mapping approach. That means that this technique does not allow indirect lighting for changing light sources or non-static scenes.

Dong et al. [13] use an approach similar to PRT by clustering different virtual point lights within the scene together and then using these as *Virtual Area Lights* as proposed in [44]. Therefore the number of shadow tests can be reduced drastically allowing for interactive computation of the indirect lighting.

The technique proposed by McGuire and Luebke [37] uses a photon mapping approach to compute indirect lighting. The authors describe an algorithm that first uses the graphics card to distribute initial photons from a light source using the GPU-projection for the expensive visibility tests. The data is then read back to the CPU where it is processed with standard ray-tracing techniques and the results are recorded in an octree structure that stores the radiance. This data structure is then sent to the graphics card again for computing the final gathering. While this technique creates a reasonable ambience that is adequate for computer games, the results look rather uniform and lack highly detailed shadow effects, due to the low number of photons and the wide spread of the corresponding photon-volumes.

An approach that computes double-bounce indirect lighting was introduced by Crassin et al. [8] and can be seen as an extension of reflective shadow maps. Instead of a shadow map, this method uses a sparse octree to store the radiosity from direct light sources. Then the radiosity is gathered for every visible pixel in the final image from this octree. That makes this technique effectively a double-bounce algorithm with rather coarse approximations in the gathering and the light distribution. Yet, this algorithm is able to handle indirect shadows in contrast to reflective shadow maps [9]. Nevertheless, this technique can only handle up to two light bounces, while our technique distributes the light with multiple bounces until the distribution is physically plausible.

2.7.3 Full solution

This section handles techniques that aim at perfectly realistic results without considering short computational times. These techniques are mainly based on following light deep into the scene in a stochastic manor to approximate a physically based result.

Jensen and Christensen [26] introduce the idea of photon mapping, which is a stochastic approach to indirect illumination. Algorithms based on photon mapping shoot energy into the scene, starting at the light sources. Whenever one of the particles (photons) hits a surface element in the scene, the energy that the photon transports is stored at this point in a *photon map* and this surface point is treated as a new light source: it shoots photons into the scene, which carry the amount of light that is reflected at this point. After the light distribution is finished, that is, when all new photons carry an amount of energy below a certain threshold, the photon map is evaluated from the viewpoint of the camera, giving each surface element a certain brightness that depends on the radiosity stored in the photon map at this position. While this technique is still state of the art in terms of quality, the time required for shooting the high amount of photons that is necessary for good and realistic results is very high. Furthermore, this method, as well as other stochastic approaches [20, 23, 25], requires a noise filter, since the results are randomly distributed and therefore scattered points. Our technique distributes the light in a smooth way that has a lot of similarity with photon mapping, but it creates results much faster. Due to the different approach in storing the final result, our technique is also very flexible, allowing the user to trade quality for shorter rendering times on a wide range. With our technique, previews that give a good and physically plausible idea of the final scene illumination can be created in the order of seconds or minutes.

All techniques that use stochastic approaches (e.g. [23, 20, 26, 25]) need a second pass that takes the noisy – due to the fact that photons or light rays where set randomly – radiosity values and smoothes them to give the final image a more natural look. Different techniques have been proposed to efficiently do that, for example the recent one by Dammertz et al. [12].

2.7.4 Current unsolved problems

So far only few techniques store the computed results (e.g. [23]), and therefore most of the algorithms have to compute them for every frame again even if the lighting conditions have not changed between the last frame to the current. Since in realistic scenes the light sources are static, this is a waste of computational capacity. Other techniques like *Imperfect Shadow Maps* by [45] store pre-computed results, but still have to do computationally expensive calculations in every frame. Another drawback of this technique is that it uses a lot of memory on the graphics card even though the data is already compressed. Other techniques rely on highly tesselated geometry for good results, like [39]. Also these techniques compute good results in real-time, but they use rather artificial scenes without texture maps. If the tessellation algorithm needed to compute texture coordinates on the subdivided surfaces instead of just subdividing the geometry, the computation time would increase again, making it hard to still achieve real-time framerates. This holds especially, since multi-textured environments are common nowadays; this means that most if not all vertices of the scene have two or more different texture coordinates.

Also almost all techniques only concentrate on computing indirect lighting and do not consider other things necessary for realistic images that are well established in the field of computer graphics, for example direct lighting with normal-mapping, transparent objects, etc. That means that these techniques create good results in real-time, but do not leave enough computational power for computing these effects that are also important for realistic images.

Another important issue is that most techniques that produce almost photo-realistic results were only tested for very simple scenes. The walls in the example scenes might have a lot of detail and there might be highly complex objects within the scenes, but realistic scenes with many rooms connected by corridors and stairways are missing. One reason for this is that the light transport can be pre-computed more easily in a single room. Another important reason is that in a single room the light distribution becomes very similar between the second iteration and the following, because in step two almost all surfaces already received light and so the further distributions do not change much on the visual outcome except the overall brightness. When the rooms become more complex, this changes and therefore these scenes require (much) more then only one or two light bounces.

2.8 Summary

In this chapter we gave an overview of the physics of light and its interaction with different materials. We also mentioned how the light is perceived by the human eye and how monitors create different colors. With that we described the basics of the direct lighting model that is used to illuminate virtual scenes and objects in the field of computer graphics to then differentiate between direct and indirect illumination of a scene.

The main part of the chapter then described the beginning of indirect lighting computations and how the field advanced. We introduced three different classes to partition the big field of related work by the time it takes to compute the solution. This reaches from realtime techniques to algorithms that compute the full solution.

We furthermore pointed out some of the drawbacks of the existing techniques and how we overcome that in our own technique.

Chapter 3

Technical Background

In this chapter we describe how virtual scenes can be rendered and how current graphics hardware is designed to speed up this process. We furthermore analyse the advancements in graphics hardware and talk about how this enabled fast and efficient techniques in the section of rendering into textures.

We start by giving an overview of how the rendering of 3D-models works in Section 3.1 and go into details about the hardware that was developed to efficiently perform the required operations in Section 3.2. We then describe the APIs that allow the programming of the hardware in Section 3.3 and describe shaders in Section 3.4 and texturing in Section 3.5, since the techniques presented later rely heavily on both techniques and needed to overcome some of the existing pitfalls. We end this chapter with an overview of techniques used to draw onto 3D-objects and project that data into the object's textures atlas in Section 3.6.

3.1 Rendering virtual objects and scenes

A virtual object given in a 3D-description, needs to be projected into a 2D-image on the camera plane for displaying. In principle there are two mainstream rendering techniques.

One is *raytracing* (see Figure 3.1 left) that was developed in the late 60's, in which a ray is shot from the camera into the scene for every screen pixel. If that ray intersects an object, the color of the pixel is computed as a combination of the material properties and all the lightrays that intersect this point on the object's surface. The light rays are created from the initial point of intersection and connect it with all the light sources in the scene. If a light ray intersects another object before reaching the light source, the original surface point is in the shadow of the light source and therefore no lighting is computed for it with respect to the currently tested light source. If the light ray is not intersecting any other object, the lighting computation is done with the light ray and the ray coming from the pixel that is currently tested, using the lighting computation

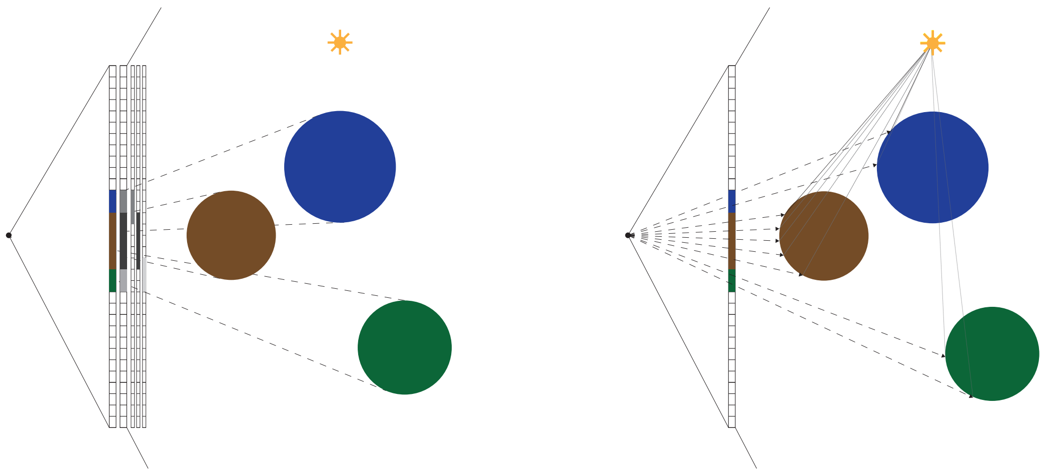


Figure 3.1. Comparison between rasterization (left) and raytracing (right). While raytracing follows a ray from the camera into the scene, rasterization projects each object onto the screen, where it influences the color, if it passes the depth test. Here the depth values are symbolized as gray scale values in the raster behind the colors. The three small lines to the right show the values that each object is writing. In case of an overlap the closer object writes both the new depth value as well as the color.

introduced in Section 2.4. This technique requires to test every ray against the whole geometry of the scene and is therefore very computationally expensive.

The second and mostly used technique in realtime graphics is *rasterization* (see Figure 3.1 right). Here the object's coordinates are multiplied by a 4×4 Modelview-Projection-Matrix that projects the vertices of the object given in homogeneous 3D-coordinates – with 1 in the fourth component – into the 2D-coordinates on the screen. The third component – the depth – is written into the *depth-buffer* (or *z-Buffer*), that was introduced in 1974 independently by Edwin Catmull [4] and Wolfgang Straßer [54]. This buffer contains one depth value for every screen pixel and is essential for the rendering of 3D-scenes, since it takes care of the occlusion of far objects by objects that are closer to the camera. When an object is rendered and overlaps parts of the screen, the overlapped parts of the z-buffer are tested against the depth values computed for the current object. A pixel is only changed by the newly rendered object, if it has a depth value that is closer to the camera then the currently stored value in the z-buffer. In that case the pixel color is changed according to the properties of the current object and the value in the depth buffer is replaced. This technique does not require any knowledge about the geometry of the scene when computing a single pixel, but it creates a certain overhead by possibly rendering a single pixel multiple times, if the objects are not ordered from front to back. It obviously requires furthermore a buffer in the dimensions of the screen for storing the depth values. Nowadays this is implemented in every 3D-accelerator.

As mentioned before, in realtime applications like games, the technique used is rasterization applying the z-Buffer. Here the geometry of the scene does not matter and the rendering can be done very fast. If speed is not the main focus, but instead the requirement is a close to real-life image, then raytracing or its successors – e.g. photon-mapping (see Section 2.7.3) – are used. Since these techniques follow a ray into the scene for an arbitrary number of reflections and refractions, almost all real world effects can be captured, including caustics, that are especially hard to handle in rasterization. Photon-mapping as mentioned before is also able to handle global illumination, but all this comes at the cost of high computation times.

Note that because of the high speed and fill rate of modern GPUs most effects can be convincingly approximated or faked. One example for that is reflection. Since real-time 3D-applications focus on fast changing environments (like for example computer games), the reflection does not need to be perfect, but should be convincing. To achieve that, one of the first attempts that was also supported by the graphics hardware was cube-mapping. When an object is rendered that has full or partial mirroring properties, the reflection ray was computed for each rendered pixel, but instead of testing this ray against the real scene geometry to find the color reflected in this surface point, the reflection ray was used to access a cube surrounding the reflecting object. This cube has a 2D-texture on each of its six faces, and the intersection point of the ray with the cube determined both the texture to be sampled as well as the position where to sample the texture. The textures for the cube could be either predefined images or even rendered samples from the real scene, that contains the object. This of course is just an approximation and it fails when a mirroring floor has an object standing right on top of it. But for most effects it does a convincing job even nowadays. But today other tricks are mainly used for faking reflections, like *Screen-Space-Reflection* by McGuire and Mara [38] or computing the intersection of the reflection vector with a simplified version of the 3D-scene stored in a 3D-texture as proposed by Crassin et al. [8].

3.1.1 3D-object descriptions

Objects can be given in a number of fashions. One is for example an implicit function describing the surface of the object. This is sometimes used in raytracing to draw spheres, since they have a very simple function to describe them. For complex objects, that is not feasible. In that case the objects can be given as a point cloud/voxels or as a mesh consisting of polygons.

Nowadays almost all objects are given as triangular meshes for multiple reasons. Having a fixed geometry shape to render, both hardware and software can be optimized for this specific type in regard to property interpolation (e.g. color, texture coordinates, etc.), intersection tests, clipping, and many more. Furthermore triangles are the most simplistic 3D-shape for approximating 3D-surfaces and they can not be filled incorrectly, in contrast to rectangles for example, that can form two triangles if two opposing edges cross. This allows for some simplifications and optimizations in the hardware that fills the pixels overlapped by the 2D-projection of the triangle. An additional advantage of triangles is, that the three vertices are entirely connected either clockwise or counter-clockwise. This fact was used from early on in 3D-acceleration for clipping non-visible faces.

The clipping orientation can be set to either one of the two. When rendering a mesh that is guaranteed to have all triangles oriented in one way, when looking at each of them from atop, then a triangle that switches orientation while being drawn onto the screen, can safely be discarded, since it is known, that it is viewed from the backside and will therefore be covered by other triangles of the object, that are actually facing towards the camera, if we assume all objects to be closed.

3.2 Hardware design

The development taking place in the area of realtime computer graphics is largely made possible by the advances in the graphics hardware. Everything started in 1961, when William Fetter created an isometric view of a human to optimize the layout of cockpits for Boeing and termed it “computer graphics”. Rather simple software techniques for rendering followed, until in 1981 the company *Silicon Graphics International* (SGI) was founded and started to produce graphic terminals in 1982. These machines were especially designed for this kind of computations and had certain computations directly hardwired into them. SGI also initiated the graphic programming API *OpenGL* (see Section 3.3) which strongly simplified the programming of 3D-visualization tools by setting an industrial standard for both hard- and software to rely on. In these first years of computer graphics, the necessary computational power was way to expensive to be employed outside of industry or universities and was exclusively used in these fields.

Then in the 1990’s 3D-accelerators like the *3dfx Voodoo* were created to be used as extensions for average personal computers and were available for a reasonable price (making 3D rendering available for everyone). The card implemented certain algorithms for 3D-computations in hardware and was meant as an addition to the regular graphics card. At the end of the 90’ies the *3dfx Voodoo Banshee* combined these two into a single hardware component increasing the fill rate and introducing the Transform&Lighting-Design (*T&L)* that implemented and parallelized the basic 3D-operations in hardware, like the multiplication of 4D-vectors and 4×4 matrices and the computation of the dot-product essential for the shading of 3D-surfaces (see Section 2.4). Since the rendering consisted only of a certain amount of steps that were then repeated for all the primitives that are drawn, this fact was used to design a hardware doing this job with a few necessary things to be setup for the scene to be drawn. The rest was then just streaming in the mesh data and applying the operations set in the fixed function pipeline depicted in Figure 3.2.

Note that here an explicit programming was not yet possible, and the developer was restricted to just setting matrices, textures, materials and other parameters, that were then used on the primitives being rendered in contrast to the much more flexible pipeline that allows complex programs to be executed on the input nowadays (compare Figures 3.2 and 3.4).

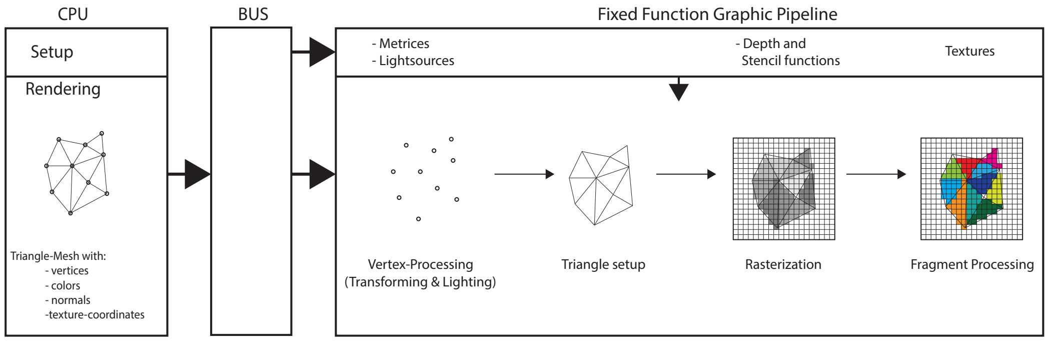
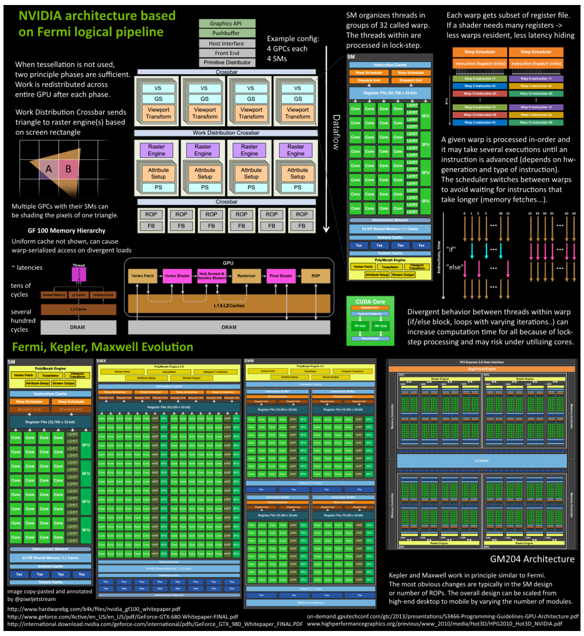


Figure 3.2. The fixed function OpenGL pipeline from the early days of hardware acceleration. The matrices for the transformations, light sources, material parameters and textures could be set, but the pipeline itself was completely static.

The main purpose of modern 3D-accelerators is to render scenes given as 3D-triangle-meshes. As mentioned before, this requires to project the objects onto the screen. It furthermore computes the translations, rotations and scaling of the objects. To do that efficiently, the object vertices are given in homogeneous coordinates as described before. This way the whole transformation and projection can be done by multiplying each 4D-vertex with a single 4×4 matrix, usually referred to as the *Modelview-Projection-Matrix* (MVP). This matrix is created by iteratively multiplying all the transformation matrices and the projection matrix together once per object and then sending the resulting matrix to the GPU.

As mentioned before, the main tasks here are 4D-vector and matrix operations, that are all implemented in hardware. This design is also known as *single-instruction-multiple-data* (SIMD), and allows for computing these operations with a single command, accessing the elements of the target vectors and matrices automatically. Furthermore it allows for using these fixed operations (that are after all equal for all the vertices, normals, texture coordinates and colors of a given object and in the second stage for all the created fragments) on all the data for a given object. Therefore the data can be streamed in as fast as possible, without the need for any explicit load operation that would be issued by the processor when finishing the operations on one vertex.

The fact that the operations for different vertices (or fragments in the later stage) are identical and more importantly do not rely on the output of any other data, the process can be strongly parallelized. Every available processor can work on its data, output it to the next stage and directly fetch the next data-element in the queue, without having to wait for any other task to finish. While this holds for the processors in one stage, it does not hold for the interaction between the two stages. Fragments can only be processed, once a full triangle is computed and rasterized. To speed this up, graphics cards have an intermediate step that gathers all vertices for a triangle and computes the overlapped region and linearly interpolates the data given at the vertices over the area of the triangle. Figure 3.3 shows the design of the NVidia-Fermi-architecture and demonstrates the amount of parallel processors as well as their connection to fast caches, to minimize the latency introduced by the low accessing speed of random access memories (see Section 3.2.1).



When tessellation is not used, two principle phases are sufficient. Work is redistributed across entire GPU after each phase.

Work Distribution Crossbar sends triangle to raster engine(s) based on screen rectangle

Multiple GPCs with their SMs can be shading the pixels of one triangle.

Uniform cache not shown, can cause warp-serialized access on divergent loads

SM organizes threads in groups of 32 called warp. The threads within are processed in lock-step.

Each warp gets subset of register file. If a shader needs many registers -> less warps resident, less latency hiding

A given warp is processed in-order and it may take several executions until an instruction is advanced (depends on hw-generation and type of instruction). The scheduler switches between warps to avoid waiting for instructions that take longer (memory fetches...).

Divergent behavior between threads within warp (if/else block, loops with varying iterations..) can increase computation time for all because of lock-step processing and may risk under utilizing cores.

Kepler and Maxwell work in principle similar to Fermi. The most obvious changes are typically in the SM design or number of ROPs. The overall design can be scaled from high-end desktop to mobile by varying the number of modules.

image copy-pasted and annotated by @pixeljetstream

http://www.hardwarebg.com/b4k/files/nvidia_gf100_whitepaper.pdf
http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf
http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF
on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf
www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_NVIDIA.pdf

Figure 3.3. Design and evolution of NVidia GPUs.[1]

Nowadays graphical processing units (*GPUs*) are much faster then current CPUs due to the design of performing many mathematical operations in parallel. Since these operations come from a mainly linear program, the complicated and necessary technology required in CPUs to keep the pipeline filled, is not necessary in GPUs, since there should be only few switches in the program path introduced by conditional jumps which were not even possible in early shaders.

In the beginning, the hardware was providing a fixed function pipeline (see Figure 3.2), that allowed for the MVP-matrix to be set and the vertices – with some additional attributes like color, normal and texture coordinates – to be streamed to it. Less then a decade later this was replaced by a pipeline that consisted of two programmable parts: the *Vertex-Processing* and the *Fragment-Processing*.

[1]Image taken from: `https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline`

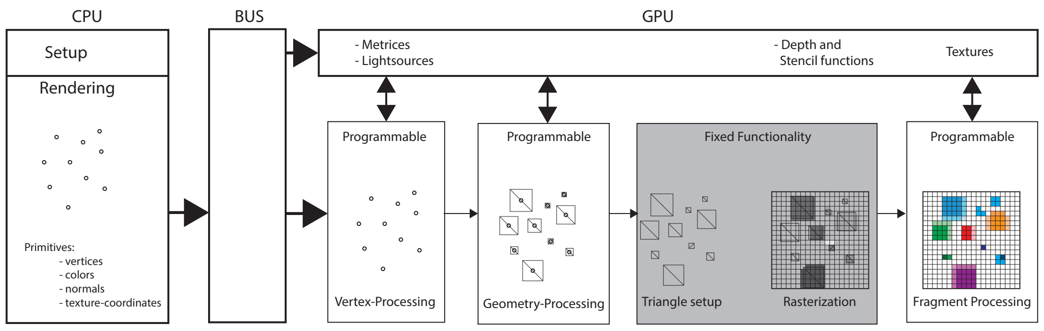


Figure 3.4. Later design of the graphics pipeline. Here the different stages are programmable and much more flexible. Furthermore the primitive type can be changed in the geometry processing and additional geometry can be created, which can partially help to overcome the bottle neck introduced by the low throughput of random access memory.

The first stage gets as an input a single vertex with all additional attributes (e.g. texture coordinates, normal and color) and runs the currently bound vertex program on this data. It can perform many 4D-mathematical operations with the provided data (mainly used for computing the lighting). The computed results are then handed down the pipeline to the fixed function rasterization processors.

The resulting fragments are then handed over to the programmable last stage: the Fragment-Processing. Here the incoming data is used to access textures and other informations to combine them into a single color for that fragment. Furthermore the depth of the fragment can also be changed here or a fragment could be discarded entirely, not writing any output.

In 2008 the hardware was strongly improved by changing the paradigm from dedicated processors for each vertex and fragment processing to an architecture that has general processors that are assigned to currently available tasks by a scheduler. It was first introduced by *NVidia* with their *GeForce 8* series. In addition another programmable stage was introduced into the rendering pipeline, namely the *geometry-processing*. This stage takes an input primitive – either point, line or triangle – and outputs a certain amount of output primitives. This allows for example to render a number of particles by sending single points to the GPU and then create billboards around the input vertices. Another possible application is to refine a given input geometry by taking an input triangle, subdivide it and change the position of the resulting vertices. Figure 3.4 shows the graphic pipeline at this stage of the hardware development.

3.2.1 Data throughput

The massive increase in the computational capability of GPUs made it necessary to further increase the throughput of the interface between the motherboard and the graphics card to keep the GPU busy by providing new data fast enough. While this started with the ISA-Bus that had a throughput of around 16 MB/s, it was increased with EISA to around 20 MB/s. Afterwards the PCI was introduced and improved from 133 MB/s in 1993 to 533 MB/s in 2004. Until 2006 existed another standard (*AGP*) that reached 2.1 GB/s in its final state. But since this state could not be improved anymore due to technical limitations, the newest standard (*PCIe*) was introduced in 2003 with 250 MB/s and is now developed to 6.1 GB/s. But despite the enormous amount of throughput, it would not be enough to keep the GPU busy, since the data would not come in fast enough.

For efficient GPU-programming it is therefore necessary to store as much data in the GPU-RAM as possible. To that end, the producers of GPUs have introduced new features that allow to store objects to be drawn in the graphics memory as vertex-buffer-objects (*VBOs*) and to render them with a single command sent from the CPU, therefore reducing the necessary communications between CPU and GPU dramatically compared to the beginning of the 3D-accelerators, when the whole mesh had to be sent to the GPU one primitive at a time. But even though it is now possible to store most if not all data in GPU-RAM, the fetching is still quite time-consuming. Although current GDDR5-RAM does allow a transfer of 20 GB/s, since during the rendering of a scene a high number of texture-lookups as well as geometry retrieval is required, the memory-access as well as the communication between CPU and GPU remain the bottleneck in most situations that occur in graphic programs.

3.3 Programming APIs

The first general API for programming the graphics card and 3D-accelerators was Glide developed by 3dfx to get computer game developer to make extensive use of their hardware capabilities. Another standard was OpenGL, that was developed to program the industrial graphic workstations created by SGI. When the general consumer market grew, OpenGL became the general standard for programming 3D-hardware. With Win95 Microsoft tried to establish their own programming API called *DirectX*. This is a more general programming-API but also contains direct3D that supports hardware accelerated 3D-programming, since it was introduced into DirectX in June 1996.

The development of these two APIs usually follows the advancements of the graphics hardware closely. Nowadays both APIs are fully supported and regularly extended. A big advantage of OpenGL over Direct3D is the fast development and extensibility. While DirectX only comes after major changes in a new Version, OpenGL allows for the use of extensions. That allows GPU-vendors to create new technologies – either in hardware and/or in software – to be used directly after the release of their drivers. Therefore developers have directly access to the newest techniques and standards. Figure 3.5 shows the different development speeds of the two APIs.

Both APIs and the underlying hardware work as state-machines, letting the user set certain parameters and keeping them until explicitly changed. Reasons for that are, among others, to avoid copying data over the bus, since it is rather slow compared to the extreme computation speed of the GPU and should be avoided and the synchronization of the driver running on the CPU and the GPU execution. These reasons also led to the introduction of many *buffer-objects* that are permanently stored in the VRAM – given that there is enough space for them – and are then executed for drawing or accessed without the need of communication between CPU and GPU.

In the following sections we will have a closer look at the techniques that this work is mainly based on, give a brief history of their development, showing that until recently, the idea presented in this work could not have been implemented.

3.4 Shaders

While the first graphics cards allowed for very fast computations of 3D graphics, the functionality was fixed to the computation of the transformation of objects with the current modelview-matrix and performing fixed lighting computations per vertex for a very limited number of light sources. Each primitive could furthermore only be colored by a single texture. This changed in 1998 with the introduction of the first ARB-extension (*ARB_Multitexture*) that allowed the lookup in multiple textures per fragment. Shortly afterwards NVidia introduced the *NV_Register_Combiners* that allowed to do limited computations with the fetched texel-data and even combine the results from different textures in a predefined way. For more details about the techniques mentioned here we refer the reader to the OpenGL specification [49].

True programmability was introduced only with the NVidia-GeForce 3 though, that introduced the *NV_Vertex_Program* that allowed the programming of the vertex processor, controlling how incoming vertices are transformed and colored. Then ATI introduced the *ATI_Fragment_Shader* that gave a great deal of flexibility to programming the resulting color of each fragment.

To make this more standardised, 3D Labs started to reorganize OpenGL into Version 2.0 in 2004. The main goal was to introduce a general shader language that was

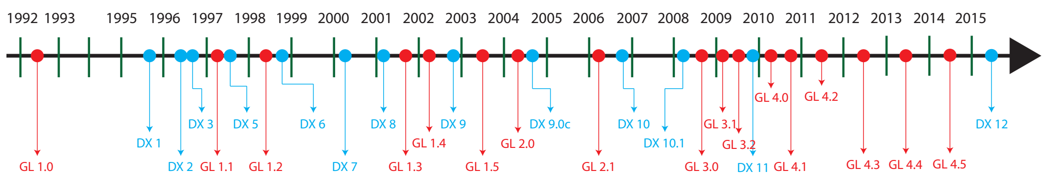


Figure 3.5. The timeline shows the major improvements of both OpenGL and DirectX. While in the beginning even minor changes took long, major versions of both APIs are released in much shorter succession now, due to the fast development of the underlying hardware.

supported by NVidia as well as ATI. This OpenGL-Shading-Language is a C-Style language, that is identical for programming both the vertex and the fragment processing unit, since the introduction of unified shaders in 2008.

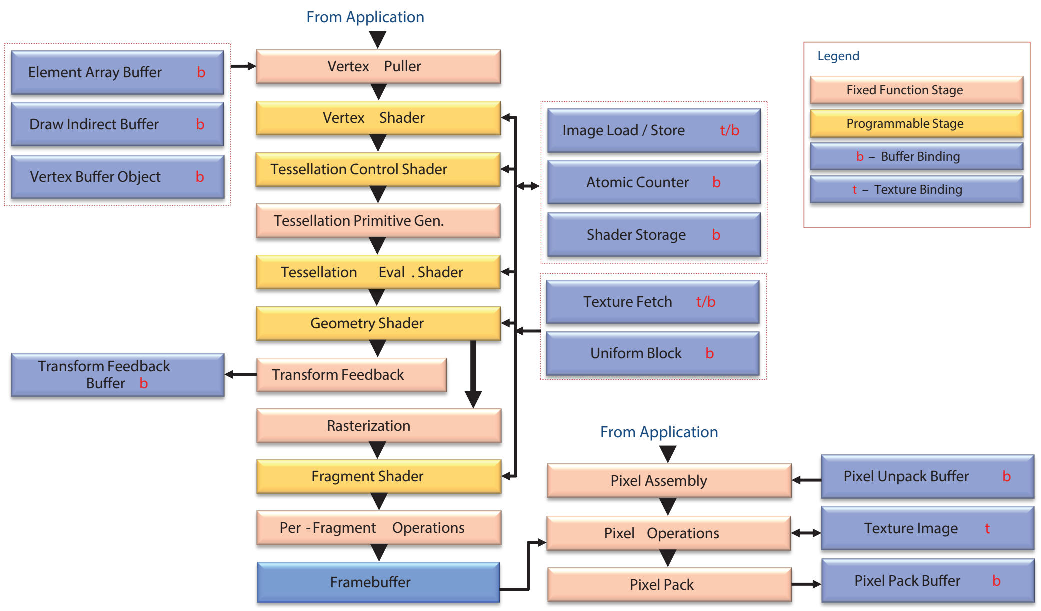


Figure 3.6. The rendering pipeline of OpenGL as of Version 4.5. Almost all stages of the pipeline are programmable nowadays. User defined operations can be performed on the vertices of incoming primitives, the primitives can be used to create new primitives (even of a different kind) and they can be tessellated by the hardware. Furthermore the final stage of coloring the output of the rendering stage can be defined by the user.[2]

With the increased flexibility of the fragment shader, it became more and more reasonable to render an image – for further use as a texture – directly on the graphics card itself in realtime. The first attempt to efficiently allow this was through the PBuffer in 2000. It allows to render a scene offscreen into a texture and then use it in subsequent draw calls as a texture. The drawback of this approach is, that it requires time consuming context switches. To overcome that, *the Frame-Buffer-Object* (FBO) was introduced and implemented with a very high flexibility in 2008. FBOs allow for rendering into multiple textures as well as into multiple layers of one texture simultaneously. This allows for a huge amount of data being processed or generated and stored on the GPU.

Other important techniques that were introduced and allowed for a much higher flexibility in the programming of the graphics card were *NV_Geometry_Shader* and *NV_TransformFeedback_Buffer* (TFB). The former technique allowed for a shader taking a standard primitive as input and creating a certain amount (around 32 on current

[2]Image taken from: `https://www.opengl.org/registry/doc/glspec45.core.pdf`

hardware) of new primitives, thus increasing efficiency by allowing to handle more data than is directly read from VRAM minimizing this bottleneck (see Section 3.2.1) and to adapt data to certain conditions at runtime allowing for higher flexibility (e.g. computing LODs depending on the distance to the camera). TFBs allow for arbitrary data to be computed on the GPU and recorded in VRAM directly. Unlike the FBO-extension, the TransformFeedback-Buffer allows the storage of arbitrary large data, and has no restrictions in the amount of data that is to be recorded from a single shader-pass. Another difference between FBOs and TFBs is that FBOs are filled by the fragment-shader, while TFBs are filled by either the vertex- or the geometry-shader (see Figure 3.6).

The last shader for the graphics pipeline that was introduced so far is the *Tesselation-Shader*. It is located between the vertex- and the geometry-shader-stage in the graphics pipeline and can be used to tessellate given primitives with a specified behaviour. It is more powerful then the geometry shader in that it does not have any restrictions on the number of created primitives and the tessellation is done automatically by the hardware. Another shader that was introduced recently is the *compute shader* that allows easy usage of the graphics card for general purpose computing is very general and therefore will not be described here in detail.

While we make extensive use of the previously mentioned techniques and extensions, the tessellation shader and the compute shader are not used within the course of this thesis and are only mentioned here for completeness and to show the direction that the improvements of graphic cards is currently taking.

3.5 Textures

When 3D-accelerators were introduced, their main purpose was to do the transforming and lighting of 3D objects and projecting the result onto a screen for 2D-visualization but the hardware also allowed for texturing the drawn objects. To that end the pixels on the screen receive an interpolated texture coordinate which is used to look up the corresponding data – back then almost entirely colors – from a texture (see Figure 3.7). The fixed function pipeline back then looked as depicted in Figure 3.2.

From that days until today, rendered scenes often use a texture called *light map* first used in the computer game Quake that contains the precomputed lighting situation within the scene, stored as an image. The final image on screen was then rendered with the diffuse texture and the light map content added on top of it (see Figure 3.8). In the beginning of interactive 3D graphics, when even simple lighting was rather expensive, it was a means to simulate indoor-lighting that was either pre-computed or created by artists. Nowadays it is still used (like in the Unreal-Engine [16]) to store precomputed indirect illumination or other static lighting effects.

Efficient rendering **into** a texture in contrast was problematic and time consuming, since it required expensive context switches or even copying the data over the bus since the hardware was not meant to be used in that direction (see Section 3.2).

With the advancement of the graphics hardware it became more and more flexible and in 2008 the graphics hardware introduced Frame-Buffer-Objects as rendering targets. They allow for drawing into a texture and using it for subsequent rendering without the need of copying back and forth between the CPU- and the GPU-memory or switching the rendering context, as was the case with the earlier PBuffers.

Furthermore the shaders that allowed to program the GPU since 2001 – starting with DirectX 9.0 or OpenGL 2.0 (see Section 3.3) – where just improved to Version 3.0 allowing for a much greater flexibility.

This combination of new technologies enables us now to implement an efficient forward mapping from the current view of a 3D-object on screen into this objects texture atlas (see Chapter 4).

3.5.1 Texturing basics

When rendering scenes or objects, the surface details are usually added through the use of textures (see Figure 3.7). Nowadays most applications use many different layers for each surface, such as diffuse-color textures, specularity textures, normal-maps, light-maps and many more (see Figure 3.9). This mapping from the texture onto the objects surface is done through interpolated texture coordinates. These coordinates are given at the vertices of the mesh – usually by UV-mapping or generic functions – and are computed for each visible fragment via barycentric coordinates.

When drawing a triangle with a texture, the texels used for each fragment on the

[3] Model and textures are courtesy of Crytex GmbH

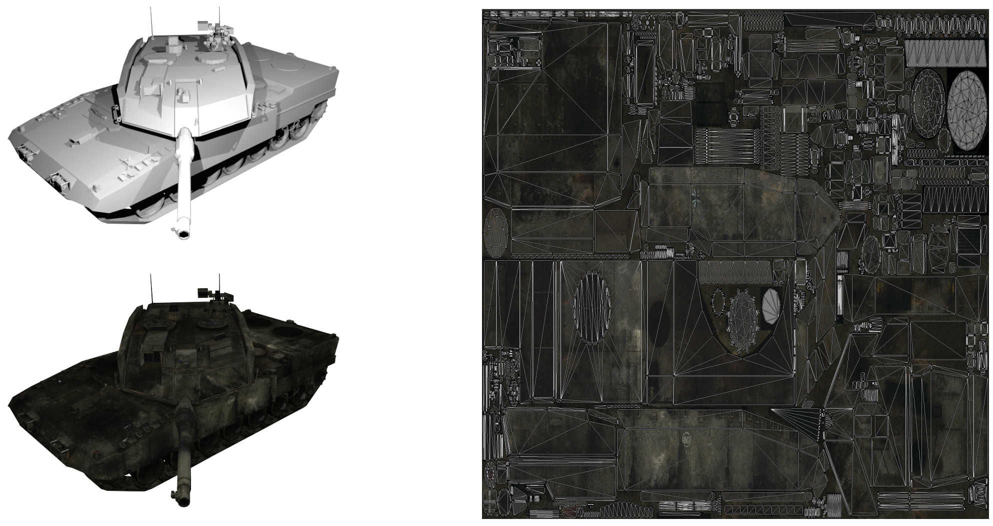


Figure 3.7. A 3D triangle mesh on the left top. Bellow is the same model with a texture. The UV-Mapping for the texturing is shown on the right overlaying the texture.

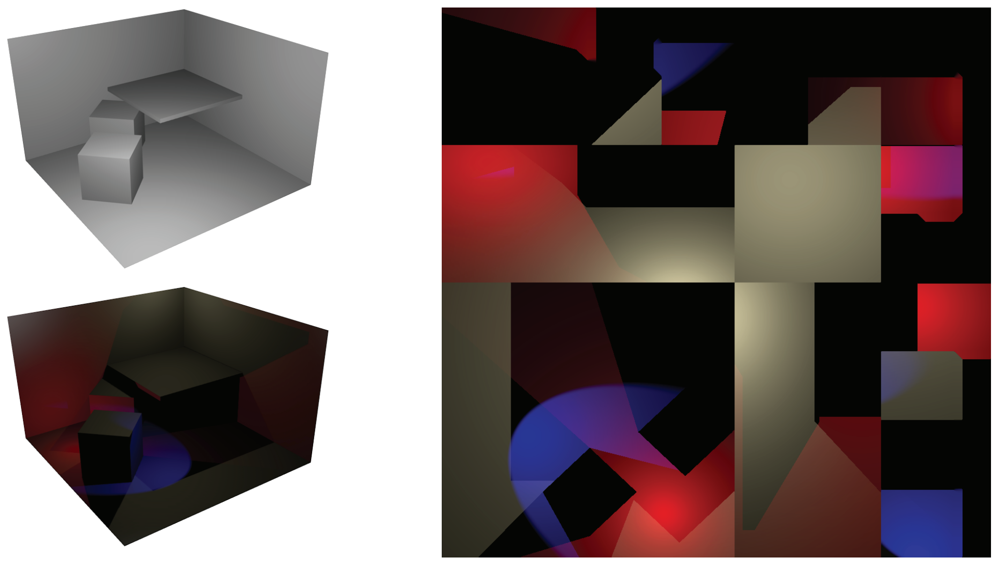


Figure 3.8. A scene using lightmaps to fake complex illumination (left) and the texture that created the effect (right).

screen, are interpolated or mip-mapped (see Section 3.5.2), so that the resulting rendering appears smooth, independent of the relationship between the resolution of the target area of the triangle on the screen and the resolution of the area of the image used as a texture.

3.5.2 Mip-mapping and (bi-)linear interpolation

When rendering a textured object on the screen, the resolution (R_T) of the texture will most likely be different then the resolution (R_S) that it covers on the screen. This leads to the problem that a screen pixel can not be correlated with a unique texel in the texture image. A screen pixel might either overlap multiple texels, or it may fall just occupy a small fraction of a texel. This problems need to be addressed to ensure a smooth and pleasing final image.

If the textured area of the object is smaller on screen, the graphics card will use *mip-mapping* for texturing. When using mip-mapping, the graphics card driver creates for every texture that is loaded, a whole series of images – each one half the size of the previous – until the smallest images has a resolution of 1×1. The texturing itself then uses the image which is closest to the one of the area that is textured on screen.

When rendering a textured triangle that overlaps fewer pixels on the screen then it has texels, the graphics card chooses the image whose resolution is closest to the resolution it overlaps on screen. Furthermore it will even choose the two closest images and linearly interpolate between them when trilinear interpolation is enabled. This way an object that moves away from the camera – therefore getting smaller and smaller –

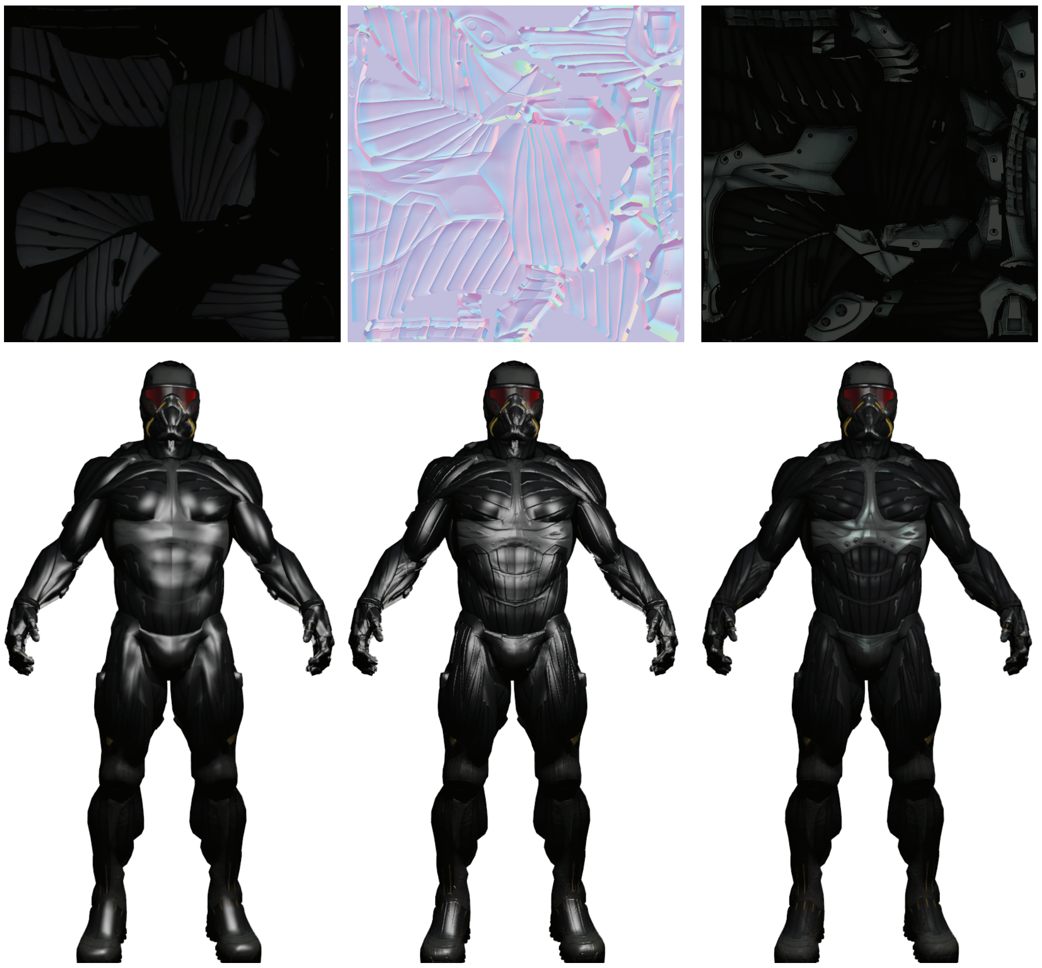


Figure 3.9. A model with just the diffuse color texture (left). Adding a normal-map (middle) adds more details to it and simulates complex structures that effect the lighting. The specular texture changes the behaviour or the reflectiveness of the object.[3]

will show a smooth and recognizable image and the colors will continuously change, while the object is fading into the background.

If the textured object instead moves closer to the camera – getting permanently larger on screen, therefore overlapping more pixels on the screen then the texture provides – the graphics card uses linear interpolation to access the texture. The linearly interpolated UV-coordinates used to access the position within the texture, will end up being between multiple texels. The graphics card will then blend between the colors of the four neighboring texels using the distance to their centers as weights in the bilinear interpolation. This effects are visualized in Figure 3.10.

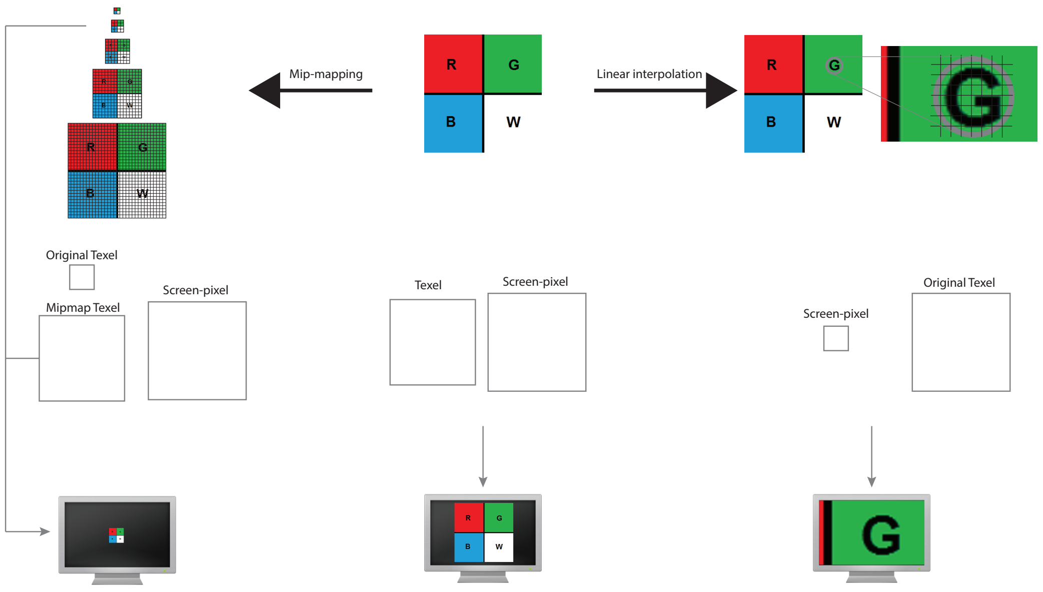


Figure 3.10. The texture (top center) has a certain resolution. When the resolution on the screen (bottom row) is equal, the texture is simply sampled without changes. If the object has a smaller resolution on the screen than the original texture, mip-mapping is applied (left side). If the object on screen has a higher resolution, linear interpolation is applied, smoothing the color for the screen pixels by interpolating the closest matching texels (right side).

3.5.3 Perspective effects

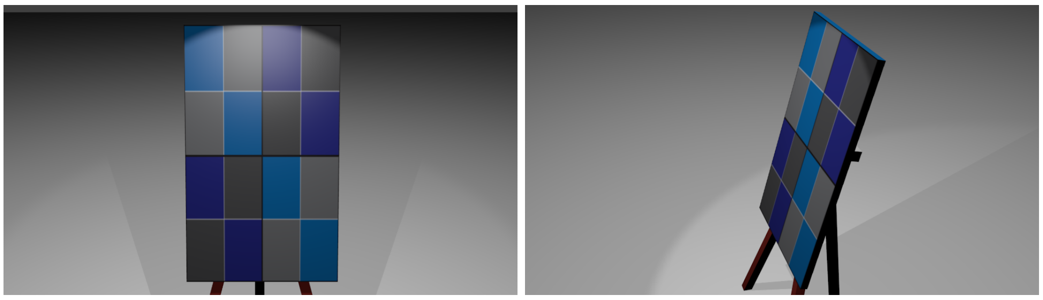


Figure 3.11. The camera faces the textured canvas head on (left) and therefore no distortion occurs. If the camera is placed to look at the canvas from a very steep angle (right), the perspective deformation is applied to the texture.

Another effect occurring in rendering textured objects is, that the image gets distorted by the viewing angle as visualized in Figure 3.11. This distortion is taken care of by the GPU by applying a perspective correction when interpolating the texture coordinates over a given surface. So whenever something is projected from the current view into the texture atlas of an object, we also have to take care of this perspective deformation.

3.6 Creating textures on the fly

In this section we give an overview of techniques that allow for drawing onto an object, storing the data into the object's texture atlas. We also point out how the advancements in graphics hardware allows for ever more efficient drawing using the GPU directly.

When looking at the advancements of the graphics hardware, the rendering with textures had a long history of improvements – mip-maping, bi- and tri-linear interpolation, anisotropic filtering, etc. – but the rendering into textures in realtime became a topic only around 2000 with the introduction of the rather expensive PBuffer. Only in 2008 with the introduction of FBOs the programmer had the chance to quickly and flexibly render into a texture.

This process can also be seen when looking into the state of the art in this area. Following is a description of techniques that draw onto a mesh and store the result into a texture atlas of the rendered object. While early techniques required all the computations and the writing to the texture on the CPU, later attempts utilised the GPU for its fast computational power. But even these first techniques that ran on the GPU could not use the efficient framebuffer object, since it was introduced only later.

The first approach to mesh painting was presented by Hanrahan and Haeberli [21]. They simply sample the brush once for each vertex of the mesh and store the sampled values as vertex colours. This technique was introduced before the rise of textures and has the disadvantage that the mesh has to be very densely tessellated for visibly appealing results. A similar approach was later presented by Agrawala et al. [1].

The recent paper by Fu and Chen [15] proposes to draw directly to the mesh triangles and even sub-sample the mesh if it is not sufficiently tessellated for good results. This approach does not follow the design of modern graphics hardware, which provides methods for reading such detail from a texture. Other GPU-based techniques sample the whole texture area, which is very expensive. Although this might work at interactive rates with a small texture atlas, it becomes slower with increasing texture sizes or more than one image per texture atlas if the mesh is very complex.

Other approaches simply process all texels of the whole texture atlas by drawing all mesh triangles into the texture atlas and sampling the corresponding screen triangles. If for some texel the corresponding screen pixel is overlain by the current brush, then the pixel is set to the colour of the corresponding brush pixel, otherwise the texel is discarded. This is completely done within the fragment program and therefore quite expensive since it operates on many pixels that are actually not get drawn to.

Igarashi and Cosgrove [24] follow this idea but introduce an intermediate step for storing the colour from a painting session in a frame buffer object that covers the whole screen. This is sufficient as long as the camera does not change and appears to the user as if the paint had already been copied into the texture atlas and texture-mapped back on the mesh surface. But the colour is actually only copied into the texture atlas of the object (by the method described above) whenever the camera moves. The main drawback of this approach is that it is bound to screen resolution. When the brush resolution exceeds the screen resolution, it can therefore not be copied without loss into the texture atlas, even if the resolution of the latter allows for the brush to be stored in full resolution.

The technique described by Lefebvre et al [30] is designed to draw on meshes that do not possess a parameterization. Therefore the paint information is not stored in a texture atlas but instead in a 3D texture. For painting as well as for rasterization this method uses an octree which is handled entirely on the GPU to guarantee fast access to the texture. The advantage is that it does not require any precomputed parameterization into a texture atlas. On the other hand, a 3D texture requires a lot of space in the graphics card memory. Another drawback of this approach is the limitation of the maximum texture resolution. This limitation does not apply to our method because it could easily be extended to work with multiple 2D textures per model, and then the overall texture resolution would be virtually unlimited.

Another paper that uses the GPU for rendering was presented by Ritschel et al. [43]. They propose to store geometry images in a texture atlas which is then used to render the object and allows for interactive surface changes by painting on the mesh. But this paper is restricted to Catmull–Clark subdivision surfaces, because it relies on the specific connectivity information that is induced by the hierarchy of these surfaces.

3.6.1 Current unsolved problems

The issues with all these approaches is speed. Many of the techniques rely on expensive computations to ensure that the results are of a high quality. But this reduces the speed of the technique. When creating a texture atlas for a scene once, this might not be of much importance, but when the creation needs to be done very often, it becomes a bottleneck for realtime rendering.

In the following chapters we will consider this problem and devise a technique that will project data onto the surface of a 3D-object into its texture atlas using the advancements made in the field of graphics hardware.

3.7 Summary

In this chapter we gave an overview of the advancements in graphics hardware and pointed out how the APIs openGL and directX followed this evolution to allow developers to make use of the hardware. We also described the techniques that are directly required for the technique introduced in the following chapters in more detail.

We also explained why the early hardware did not allow for an efficient implementation of that technique. Efficient implementation of this technique become only recently available through a combination of all the available technology developed for the graphic cards, most importantly the introduction of framebuffer objects as well as flexible shaders.

We then gave an overview of older techniques that can be used to paint onto a 3D-object and project this painting interactions into the texture map of that object. We furthermore pointed out problems that these techniques are facing.

Chapter 4

Forward Mapping

Texturing is nowadays a common approach to improve the fidelity of 3D-scenes and objects. The textures are accessed using texture coordinates given for every vertex of the mesh. When rendering a triangle, these coordinates are interpolated over the area of the triangle on screen and then used to access the related texture to read the necessary data for each screen pixel. In this chapter we introduce a flexible approach for projecting data into the texture atlas of an object from any chosen direction, handling all the occlusions and projection effects introduced by viewing the object from that specific view point (Figure 4.1 visualizes this process). This projection can be seen as a way of drawing into the texture atlas of a given object from a certain view and projecting the data from the object into its texture. This can be used for painting onto a mesh (see Chapter 5), but it can also be used in non-trivial applications like storing the illumination of light sources into a light map (see Chapter 6). To accomplish this projection, we need to:

- determine the object's current projection,
- check which texels are currently visible,
- and smoothly project the data into these texels.

Another important goal for using this technique in the creation of a whole texture map for a virtual object or scene is, to do this projection as fast as possible. Therefore we design the technique to make excessive use of the GPU, harnessing its extreme parallel computation power, rasterization and texturing hardware.

As described in Section 3.5, rendering into a texture using the graphics card is only efficiently possible since the hardware became flexible enough to be programmed and the introduction of FBOs (see Section 3.3). In this chapter we describe an approach to use the graphic hardware to update a virtual object's appearance by projecting data onto its surface and storing it in this object's texture atlas. We start by explaining the overall goal in Section 4.1 and describe general problems that have to be overcome. We

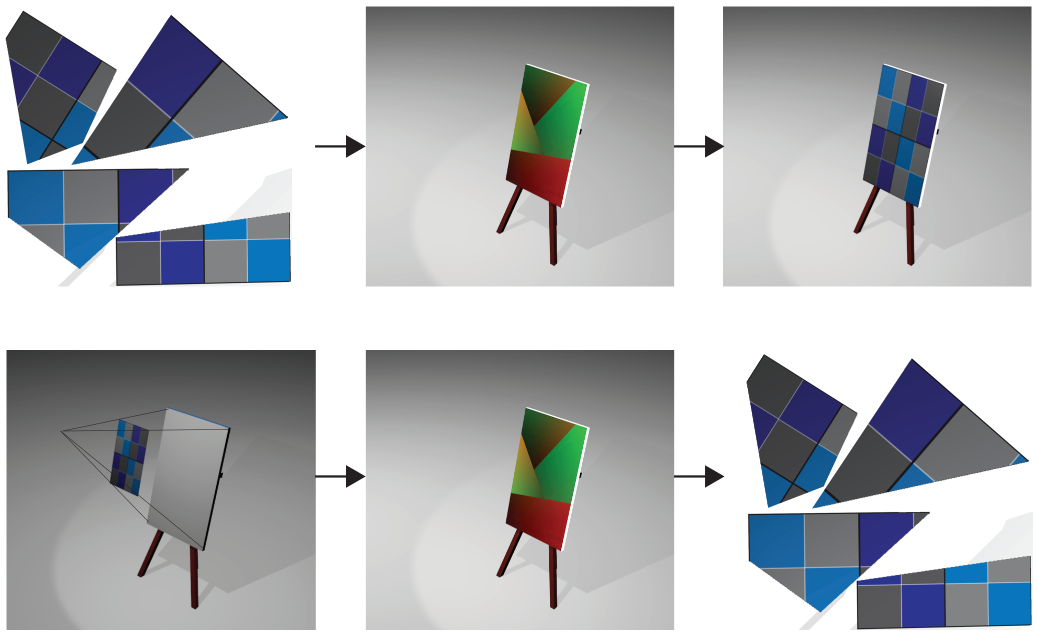


Figure 4.1. The top row shows the standard graphic pipeline. It takes a texture (left) and a model with UV- coordinates (center) and renders the final image. The bottom row shows the forward mapping idea. We have a mesh and chose data to be projected onto it (left). We then sample the UV-coordinates (center) and create the appropriate texture atlas with all the distortions introduced by the given UV-mapping and viewpoint.

then describe naive approaches in Section 4.2 and discuss problems that arises when using them.

This serves to give a general idea of the overall problem. In Section 4.3 we proceed by introducing a basic idea that is efficient and fast and can handle the most common cases and explain it in more detail in Section 4.4. In Section 4.5 we make the idea robust enough to handle all possible cases.

4.1 Basic idea

Assume there is a virtual 3D-object – with a given image texture in its texture atlas – visible on the screen. Now data should be projected onto it from the given current view. After the data is projected into the texture atlas, the rendered object should look the same on screen even without the data overlapping it, since the projection will take care of all the perspective corrections and resolution problems (see Sections 3.5.2, 3.5.3 and Figure 3.11). The method we present here will only change that parts of the object's texture atlas that are actually overlapped by data on the screen.

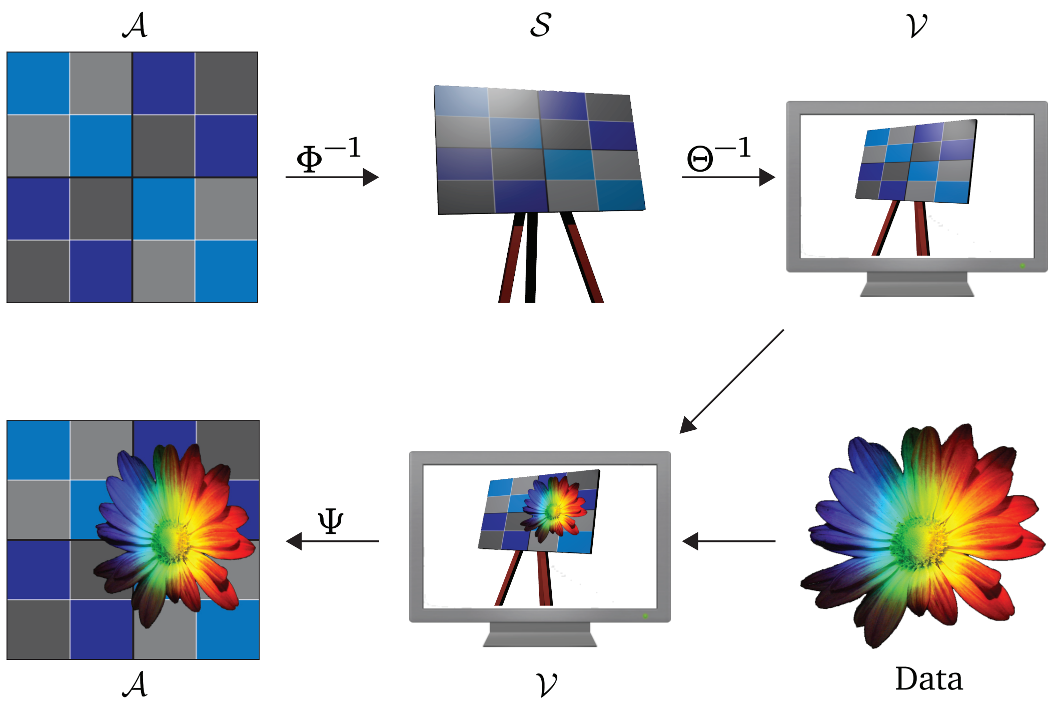


Figure 4.2. The top row shows the standard rendering of a textured 3D-object. On the left we have the texture atlas $\mathcal{A}$ with a simple pattern that is mapped by Φ^{-1} onto the 3D-object shown in the center. After applying Θ to the object $\mathcal{S}$, it is projected into the view plane $\mathcal{V}$ visible on screen shown on the right side. The bottom row shows the raw data that is to be projected on the right side. The center shows $\mathcal{V}$ with the data overlapping parts of $\mathcal{S}$. The left side shows the resulting $\mathcal{A}$, after the data is projected into it via Ψ.

4.1.1 Problem description

Let's say we have a view of a 3D-scene $\mathcal{S}$, textured with an atlas $\mathcal{A}$, from an arbitrary angle and then project data given in the projection plane $\mathcal{V}$ of the current view. At any time you want to be able to project the data into the corresponding part of $\mathcal{A}$, that is, exactly into that section of $\mathcal{A}$ that was used to texture the parts of $\mathcal{S}$ that are currently visible on $\mathcal{V}$. Figure 4.2 visualizes this setup.

This mapping $\Psi\colon \mathcal{V} \to \mathcal{A}$ from the projection plane into the texture atlas depends on the current camera parameters used to draw the scene onto the screen. It is not trivial to define this mapping and it would involve a lot of computations to perform it – like with raytracing for example.

But such a mapping does not easily overcome the problems of the different resolutions R_S for the screen and R_T the data. So instead we achieve this by separating the projection Ψ into two independent mappings Φ and Θ so that $\Psi = \Phi \circ \Theta$. Here Θ^{-1} is the mapping from the 3D-object into $\mathcal{V}$ while Φ is the parametrization of the 3D-object into $\mathcal{A}$.

We now describe the general idea that we will refer to as *forward-mapping* for realizing this mapping $\Psi\colon \mathcal{V} \rightarrow \mathcal{A}$ that projects the data, given as an image, into the 2D-texture-atlas $\mathcal{A}$ of a given 3D-scene efficiently on the graphics card.

So instead of going entirely from $\mathcal{V}$ to $\mathcal{A}$ directly, we rather store the data needed for Ψ into $\mathcal{V}$. Now projecting the data from $\mathcal{V}$ becomes a simple look-up of the results of the previous mapping, containing for every pixel on the screen the target in $\mathcal{A}$, that is, the texture coordinates sampled for drawing this specific pixel on the screen.

4.2 Naive approaches

The obvious approach for this projection would be to sample every pixel's UV-coordinate and draw on the corresponding position in $\mathcal{A}$. This can be easily done by checking for each pixel, which texture coordinate it uses for rendering and then drawing at this coordinate in $\mathcal{A}$. This would introduce two problems though:

- In case of R_T being higher then R_S, only one of the texels that are drawn onto the same screen pixel would receive a color, while the others would remain untouched.

- If R_T is lower then R_S, all texels will be drawn, but there is no linear interpolation of the data that is to be projected.

Similar problems would arise if we were to loop over all texels in $\mathcal{A}$ and project them onto the screen for sampling the data that is to be projected. In this case it is guaranteed that all texels will receive data, but depending on the relationship between the resolutions R_T and R_S, the result in $\mathcal{A}$ might be very blocky (if $R_T > R_S$), or the resulting projection will be non-continuous, since the texels fetch data too far apart on screen without interpolating it (if $R_T < R_S$). It is worth mentioning here that it is not reasonable to loop over all texels in the scenario of projecting data from screen, especially since there might be multiple texture atlases used for big and complex objects. In that case only few of the texels – the few actually visible on screen – will receive data but the projection of the texels into the space of the data that is to be projected has to be done for all the texels, making this very expensive.

4.3 Our approach

We consider the first approach, going from screen space into the texture atlas for performance reasons. To avoid the previously mentioned problems, we project from $\mathcal{V}$ into the texture atlas $\mathcal{A}$ using a continuous projection mesh $\mathcal{M}$, therefore touching only the texels of interest. This is especially important for huge scenes having possibly dozens of texture atlases; looping over all texels in this scenario would be very costly.

To achieve this projection, we first have a look at how the scene is rendered: For each textured triangle, the fragment on the screen gets the interpolated color from $\mathcal{A}$, determined by the texture coordinates computed for that fragment.

Since these texture coordinates on screen are required to get the target position of the projection into $\mathcal{A}$, we store them in an offscreen geometry-buffer $\mathcal{G}$ that exactly overlaps $\mathcal{V}$ and has the same resolution as the viewport. Whenever $\mathcal{V}$ or an object within the view moves, we need to refresh the content of $\mathcal{G}$, otherwise it can be reused for every projection onto the mesh. Therefore this projection technique requires only a single rendering step – the projection of the data into $\mathcal{A}$ – once $\mathcal{G}$ is computed.

Processing $\mathcal{G}$ per fragment of interest would not be sufficient for a good result though. Painting all the overlapped texels in $\mathcal{A}$ would lead to an uneven drawing, since the two resolutions – $\mathcal{G}$ and the overlapped region in $\mathcal{A}$ – will have different resolutions as mentioned in Section 4.2.

We achieve continuous and smooth results with the acceleration techniques already implemented in the graphics hardware. We overlap $\mathcal{V}$ – or the area of interest which might be much smaller – with a continuous 2D-triangle mesh $\mathcal{M}$ (see Figure 4.3). Then we use a shader to sample $\mathcal{G}$ at the triangle vertices and draw $\mathcal{M}$ with these new coordinates into $\mathcal{A}$. Since vertices that are shared by multiple triangles of $\mathcal{M}$ sample the same coordinate, the resulting projected mesh in $\mathcal{A}$ will also be continuous.

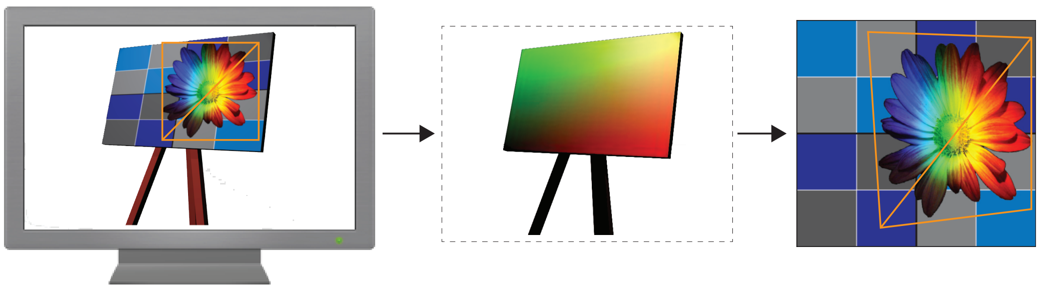


Figure 4.3. The image shows $\mathcal{V}$ from Figure 4.2 with the data overlapping it (left). It also visualizes the mesh $\mathcal{M}$ that is used for the projection of the data into $\mathcal{A}$ in orange. The overall mapping Ψ is achieved by using the content of $\mathcal{G}$ (center), to sample for every vertex v of $\mathcal{M}$ its position within $\mathcal{A}$. The resulting $\mathcal{A}$ with the data projected into it (right) furthermore visualizes the distorted mesh $\mathcal{M}$.

We therefore do not rely on sampling in either of the spaces, to project the data from $\mathcal{V}$ into $\mathcal{A}$, but instead we let the graphics card sample as good as possible in both via texture sampling and rasterization. To speed up this process, we implement the projection and the sampling using the graphic hardware's build-in capabilities. The mesh $\mathcal{M}$ accesses the data that is to be projected via a texture. The projection into $\mathcal{A}$ is done by sampling the texture coordinates t_c of the object seen on screen under each of the vertices v_i of mesh $\mathcal{M}$. The sampled t_c will then be used to draw $\mathcal{M}$ into $\mathcal{A}$ using the graphics cards texturing implementation to store in each texel the closest fitting data, using mip-mapping or linear interpolation for accessing the texture that contains the data that is to be projected.

Section 4.4 describes the technique in detail and gives an example of how each step is implemented. This projection only relies on sampling and no computation is needed. Furthermore all necessary operations are implemented very efficiently using the graphics hardware in the way it is intended to by using sampling, rasterization and vertex transformations.

4.4 Implementation details

After the camera is positioned and $\mathcal{V}$ is fixed, we first render the scene into $\mathcal{G}$, which is implemented as a Frame-Buffer-Object (FBO), which gives us the mapping Θ. Note that this FBO has 16 bits per color component per pixel instead of the regular 8 bits, since otherwise the quality would be insufficient. Instead of rendering the object onto the screen, we use a shader that stores its pixelwise linearly interpolated UV-coordinates (u, v) for each visible fragment $P(x, y)$ into the red and green channel of $\mathcal{G}$. This render step effectively performs the first part of the projection Ψ, albeit in the inverse direction. That is, the scene is projected onto $\mathcal{M}$ rather than the other way around (see Figure 4.3). Note that $\mathcal{G}$ has the same resolution as the screen used for displaying the object.

We then create a uniform 2D triangle mesh $\mathcal{M}$ covering $\mathcal{V}$ as a VBO and send it to the graphics card. When projecting data onto the mesh we use the shader to transform it to the position of interest and simply render it. This drawing of $\mathcal{M}$ uses a shader that samples $\mathcal{G}$ at the coordinates (x, y) of each mesh vertex and replaces the latter with $\mathcal{G}(x, y) = (u, v)$, which is the mapping Ψ. Furthermore we also send the original (x, y) coordinates in $\mathcal{V}$ down the pipeline, since this information is needed to sample the data that we want to project into $\mathcal{A}$. Now every mesh triangle is turned into a triangle in $\mathcal{A}$ – implemented as an FBO – and drawn there. Since each triangle also contains the texture coordinates (x, y), each texel in $\mathcal{A}$ gets the best fitting data since the graphics hardware is using mip-mapping or linear interpolation on the data, according to the current relation between the resolutions of the data and the area that it overlaps in $\mathcal{A}$.

Due to the fact, that $\mathcal{M}$ is continuous in $\mathcal{V}$, it will also be continuous in $\mathcal{A}$. If multiple vertices of $\mathcal{M}$ sample the same (u, v) and get therefore projected to the same texel in $\mathcal{A}$, the neighboring triangles of $\mathcal{M}$ draw from this point on.

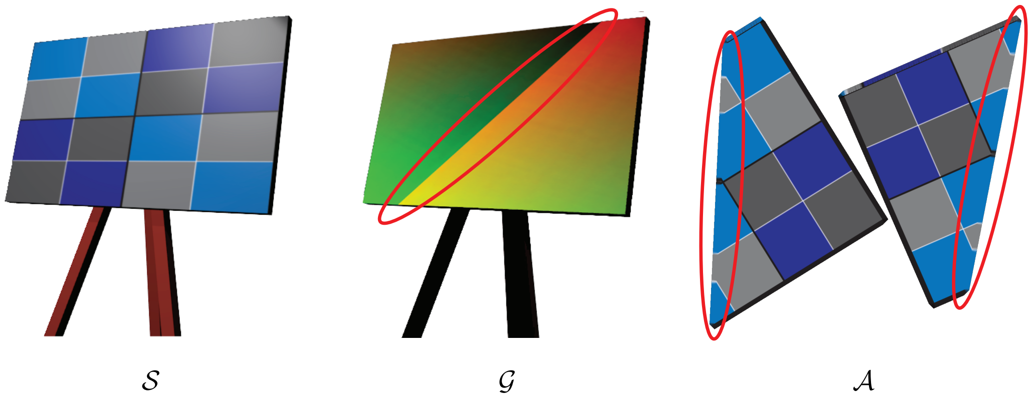


Figure 4.4. The scene $\mathcal{S}$ as seen in $\mathcal{V}$ (left) is identical to the situation in Figure 4.3. The visualization of the content of $\mathcal{G}$ (center) and the texture atlas $\mathcal{A}$ (right) show however, that this objects texture is divided into two independent charts, making $\mathcal{A}$ no longer continuous.

4.5 Seams

What has been described so far works nicely if the target of the projection is itself continuous. But this can not be guaranteed for the texture atlases of most scenes. In almost all cases the result of the parametrization creates multiple charts $\mathcal{C}_i$ in the texture domain $\mathcal{T}$. On the surface of the mesh these charts are separated by *seams* as shown in Figure 4.4. When trying to project over a seam with the coordinates sampled from $\mathcal{G}$, the following problem arises: Since the texture is not continuous in $\mathcal{T}$, the vertices of $\mathcal{M}$ are projected into unrelated parts within $\mathcal{T}$, each in the chart that the coordinates in $\mathcal{G}$ were sampled from. Though each individual vertex of the triangle of $\mathcal{M}$ is now in the correct place within $\mathcal{A}$, the resulting triangle covers an unrelated area within $\mathcal{A}$. Figure 4.5 visualizes the effect.

Sampling $\mathcal{G}$ under each vertex works nicely in absence of seams, since it projects the vertex onto the exact point in $\mathcal{A}$ that was used to render it on screen. But it requires $\mathcal{G}$ to contain texture coordinates for all the triangle vertices. This is not guaranteed, since $\mathcal{M}$ can be partially outside the object (see Figure 4.6). In this case one or more of the vertices of the triangle can not be projected, since no target coordinate is known. Therefore the projected triangles has one or two correct vertices in $\mathcal{A}$ – the ones that actually sampled texture coordinates – while the other points get the coordinate that is the result of sampling the background of $\mathcal{G}$.

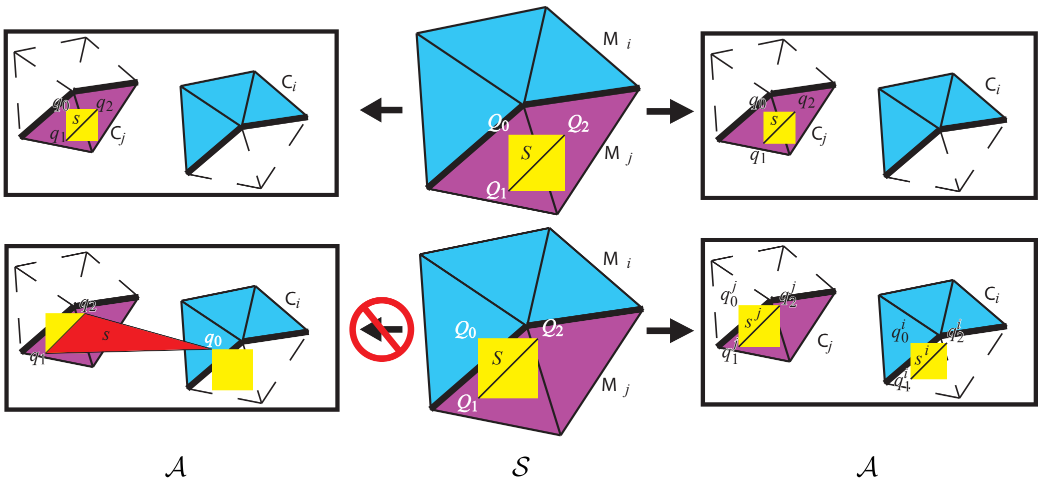


Figure 4.5. The top row shows the projection as described before. Since the triangle S overlaps only the part $\mathcal{M}_j$ of the 3D-mesh that uses the chart $\mathcal{C}_j$ the projection can be carried out without problems by just sampling the underlying texture coordinates of $\mathcal{M}$. The bottom row shows the result of simply using the sampled coordinates on the left side (resulting in the wrong red triangle), while the right side shows the expected result.

4.6 Solving the seam-problem

The two previously introduced problems can be solved in different ways that depend on the main goal of the application. Here we introduce a straight forward approach that is easy to implement. The solutions is to draw arbitrary close to the seam. This can be achieved by iteratively refining the projection mesh $\mathcal{M}$ until each triangle overlaps at most two pixels. We implement this using a geometry program and two transform-feedback-buffers ($\mathbf{T}_1$ and $\mathbf{T}_2$). To refine the mesh we draw the initial mesh $\mathcal{M}_0$ while the shader tests for each triangle if it overlaps a seam.

To detect that, we update $\mathcal{G}$ to contain a chart ID within its blue channel for all visible pixels of $\mathcal{S}$ in $\mathcal{A}$. These chart IDs are computed together with the parameterization of the mesh and handed over as a third component of the texture coordinates when rendering $\mathcal{S}$ into $\mathcal{G}$.

Note that there could be seams within one chart, where for example the left side and the right side of the texture mapping meet. This case cannot be handled by the forward-mapping approach and needs to be identified in a preprocessing step. By splitting up such a chart into two, therefore creating an additional seem, the problem can be overcome. This does not interfere with the process of the mapping though, and does not create any problems during the projection step.

In the refinement step the shader reads these chart IDs from the blue channel of $\mathcal{G}$

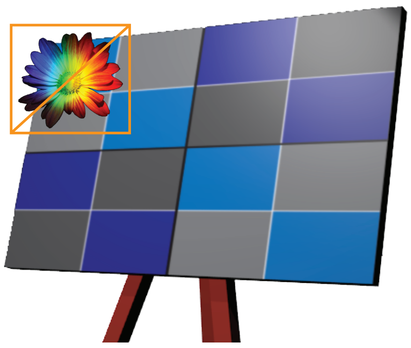


Figure 4.6. Two triangles of the mesh $\mathcal{M}$ are partially outside of the target object $\mathcal{S}$. Only one of the vertices of $\mathcal{M}$ would sample a valid UV-coordinate, while the others would not sample any. This makes a projection of these two triangles impossible. Note the huge amount of information that is discarded, since the data that is to be projected is simply getting lost, when these triangles are ignored.

and compares all of them. If even a single one has a different ID than the others, the triangle needs to be subdivided into smaller ones. The subdivision scheme used by the shader is visualized in Figure 4.7. It creates four smaller triangles for the original bigger triangle if a subdivision is necessary while writing the original input triangle otherwise. Furthermore, triangles that lie entirely outside the mesh are completely discarded. We chose this subdivision scheme because it ensures that all the triangle vertices lie within the center of a screen pixel, hence within the center of a texel in $\mathcal{G}$.

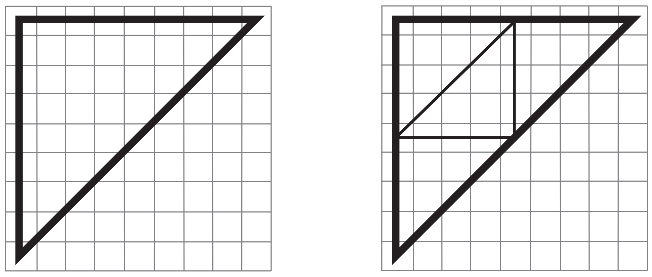


Figure 4.7. The triangle that has to be refined overlaying a pixel grid (left side). This triangle is then subdivided into four new triangles. The new vertices lie again in the center of a texel of $\mathcal{G}$ (right side).

To achieve a good sampling and find every seam, the shader samples $\mathcal{G}$ not just at the triangle vertices, but also at different locations inside the triangle. The number of sample points within the triangle depends on the current resolution of the mesh. While the big triangles of the initial mesh $\mathcal{M}_0$ need to have a high number of internal sample points, this number can be decreased for later meshes due to their smaller triangle

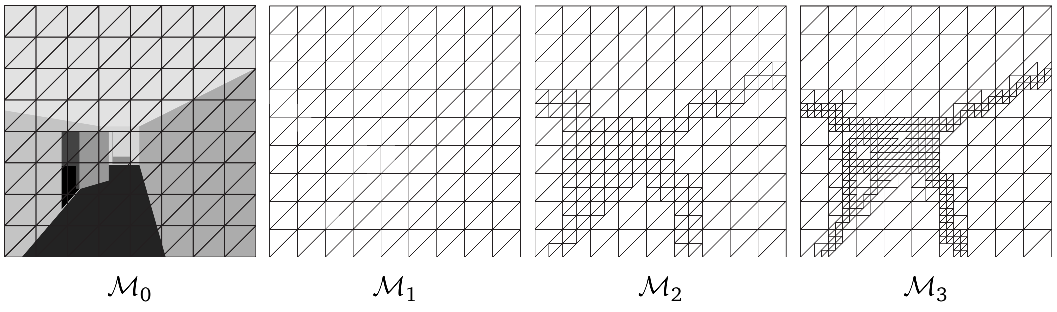


Figure 4.8. We adaptively tessellate the initial mesh by iteratively refining the triangles that overlap a seam (from left to right).

areas. Since the sampling of textures is rather expensive, this optimization is quite important.

4.6.1 Subdividing $\mathcal{M}$

Our first approach was implemented by passing the current tessellation level as a parameter to a single shader. Then within the shader we tried to execute different functions with different numbers of samplings of $\mathcal{G}$. This proved to be impossible however, since texture accesses have to be fixed at compilation time of the shader. That forced us to use a multi-shader approach, in which we implemented the different numbers of samplings in different shaders and then loaded the appropriate shader, when switching between the render targets $\mathbf{T}_1$ and $\mathbf{T}_2$ after each refinement step. The iterative refinement process can be seen in Figure 4.8.

The process of subdividing $\mathcal{M}$ starts with rendering the initial mesh $\mathcal{M}_0$ into $\mathbf{T}_1$. In each of these rendering steps, the appropriate shader – that is, a shader with a good number of samples for the current resolution of the smallest triangle in the currently refined $\mathcal{M}$ – checks each triangle and subdivides it if this is necessary. This step is now repeated by rendering the resulting mesh from the first step ($\mathcal{M}_1$) stored in $\mathbf{T}_1$ into $\mathbf{T}_2$ after selecting the right shader with fewer sample points to refine the mesh further. This iteration is done f times – always switching $\mathbf{T}_1$ and $\mathbf{T}_2$ as being the render target in one step and providing the current subdivision of the triangle mesh in the next – until the smallest triangles (that is, triangles that have been refined continuously in every step) overlap exactly two pixels in $\mathcal{G}$. The final mesh $\mathcal{M}_f$ is now so highly tessellated around – and only around – the seam (see Figure 4.8), that triangles still overlapping the seam can be discarded without having strong visible effects in the resulting projection.

4.6.2 Computational cost

In case the whole view $\mathcal{V}$ with a resolution of 1024×1024 is overlapped with an initial triangle mesh of $4 \times 4 \times 2$ triangles, the tessellation will be sufficient after 8 iterations.

While this approach leads to acceptable results, it needs multiple iterative steps on the GPU that need to be started by the CPU. This communication will decrease the computational speed of the otherwise entirely GPU-based technique. Note that a read-back into the RAM is not needed, since the contents of transform-feedback-buffers can be used for drawing by simply defining them as a VBO and using its index for drawing its content.

Furthermore, the number of triangles that need to be drawn is usually much smaller than the number of triangles that cover $\mathcal{V}$ with a size of 2×2 pixels, since such a high resolution is usually only needed for a fraction of $\mathcal{V}$. But since the exact triangle count is not known in advance, the transform-feedback-buffers need enough space to house the worst case amount of triangles. This case occurs if every triangle needs a subdivision in every step.

In cases where a 3D-object or scene has more than just one texture atlas we would encounter a similar problem like with seams. Also in this case we can not simply use the sampled UV-coordinates, since they might lie in different atlases. In that case we need to draw into both related atlases. To handle this, we also store a unique ID for every chart into the blue channel of $\mathcal{G}$ and – in case of multiple atlases for the given scene – we additionally store the atlas-ID in the alpha channel of $\mathcal{G}$. Writing and sampling these additional information comes without an additional cost, since the texture fetching reads all four elements (red, green, blue and alpha) for every access to it regardless.

In the example above that would be $512 \times 512 \times 2 = 524288$ triangles. Each needs to store 3 vertices with 3 float (consuming 4 bytes) components for the coordinates, leading to a memory consumption of $524288 \times 9 \times 4 = 18874368$ bytes. Although 20 MB are not too much, requesting this amount in the limited GPU-memory should be avoided.

In the next chapters we introduce other approaches that are more tailored for their specific applications and are therefore more efficient in projecting data.

In case that the 3D-object has more than one texture atlas, our technique can handle that by checking not just the seam-IDs when refining the brush mesh $\mathcal{M}$, but also checking for the additional texture-atlas-ID. The drawing itself would require as many drawcalls to the mesh $\mathcal{M}$, as there are texture atlases used in the scene. Although geometry programs are able to target more then one FBO at a time, it has to write into **all** connected FBOs. So even if we bind all texture atlases for drawing, the restriction of having to draw to all restricts any optimization here. Note that the filling of the geometry buffer $\mathcal{G}$ has to be done only once in contrast.

4.7 Summary

In this chapter we introduced a theoretical idea that allows to continuously and smoothly project data from a given viewpoint for any object into its texture atlas. The data is stored in exactly the same way, as was visible from the viewpoint when projecting it downward. Given a reasonable resolution of the texture atlas for the object that is the target of the projection, no difference will be visible, unless the viewing onto the object changes, at which point the data will be apparent to be now on the object's surface, as is the main purpose of texturing.

We mentioned the problems that are mainly created by different resolutions of the texture atlas $\mathcal{A}$, the resolution of the projected 3D-scene $\mathcal{S}$ and the resolution of the data that is to be projected. We overcome these by projecting the texture coordinates of $\mathcal{S}$ into an off-screen geometry-buffer $\mathcal{G}$ and overlapping the projection plane $\mathcal{V}$ with a continuous triangle mesh $\mathcal{M}$. We then sample $\mathcal{G}$ for each vertex of $\mathcal{M}$ and render the triangle into $\mathcal{A}$ with these sampled coordinates. This way the graphics card is using the best possible sampling of both the data that is to be projected, as well as the resolution of $\mathcal{A}$.

We also mention the problem created by seams when using the above mentioned technique and introduce an adaptive tessellation as a solution to prevent any visual artefact in the resulting projection. The tessellation introduced here is done on the GPU and creates additional triangles only where it is necessary, therefore reducing the amount of computation needed to draw the resulting mesh $\mathcal{M}$.

The whole technique uses the graphic card in the way it is intended to – that is, transforming vertices, texturing and linear interpolation of data across triangle surfaces – by reducing the projection computation to a sampling of an off-screen buffer that contains the results of this mapping. Another important aspect of this approach is, that the continuous data in $\mathcal{V}$ is projected continuously into $\mathcal{A}$ without any unwanted distortions or holes.

Chapter 5

Mesh Painting

In this chapter we present a new algorithm that uses the Forward-Mapping-Idea introduced in Chapter 4 for interactively painting onto 3D meshes exploiting recent advances of GPU technology (see Section 3.3).

As the user moves a brush over one or more 3D objects, a selected paint pattern is projected onto the 3D geometry from the current viewing angle and copied to the corresponding region in the object's texture atlas. Both operations are realized on the GPU, with the advantage that all data resides in the fast GPU memory, which in turn leads to high frame rates. Here we introduce a better approach for handling seams allowing to draw arbitrarily close to them in Section 5.2.4

Whenever the brush overlaps two or more patches, this situation is detected and the paint pattern is copied correctly to the corresponding texture charts. Due to the forward-mapping idea, the operation of the projection into the texture atlas is reduced to texture lookups. The performance of this algorithm is independent of the resolution of both the brush and the objects texture atlas as well as the number of mesh triangles.

5.1 Introduction

A common way of storing data given on the surface of a 3D mesh is to write it to the object's texture atlas [35]. And as the texture resolution dictates the sampling density, current techniques usually fill the texture by sampling the surface values on the mesh once for each texel [36]. For large textures this can be very expensive and therefore undesirable, in particular if the data on the mesh surface changes frequently and only in small areas.

An example of the latter is mesh painting, where the user wants to draw with a brush onto the surface of a mesh and so its texture atlas must be updated correctly and at interactive speed. In this situation, a more natural approach is to work the other way around, that is, to project and copy the brush pattern into the relevant texels.

As mentioned in Chapter 4, the main difficulty of this approach is to correctly deal with texture atlases that contain more than one texture chart (see Section 4.5). In this situation, the 3D mesh is split into several patches, each with its own texture chart, which is generally unavoidable for complex meshes, both for topological and practical reasons regarding parametric distortion [33].

Now, if the projection of the brush $\mathcal{B}$ onto the 3D-object $\mathcal{S}$ is a contiguous region on the mesh surface that spans across $k \geq 2$ patches $\mathcal{C}_i, i \in 0 \ldots k$, its projection into $\mathcal{A}$ is no longer contiguous as it lies in different texture charts (see Section 4.5 and Figure 4.5). Note that the notation changed slightly from Chapter 4. We replace $\mathcal{V}$ that described the whole viewplane with $\mathcal{B}$ that denotes a brush that can be – and in most cases is – much smaller than $\mathcal{V}$.

We handle this situation by mapping the brush into each of the separate texture charts that correspond to the k patches which intersect with $\mathcal{B}$. In order to realize this idea, we must enlarge the charts by computing additional *virtual texture coordinates* VTC for the mesh vertices near the patch boundaries (see Section 5.2.5). A nice consequence of using enlarged charts is that the texture data is replicated in corresponding regions near the chart boundaries in the texture atlas, which in turn helps to avoid texture bleeding if bilinear texture filtering is turned on, leading to texel-reads slightly outside of the current texture chart. Figures 5.1 and 5.7 show some examples of our approach.

Our technique is designed to nicely follow the flow of the graphics pipeline and exploits the capabilities of every unit: vertex processor, geometry processor, rasterizer, and fragment processor. Once fed with the relevant data, it computes the mapping $\Psi \colon \mathcal{B} \to \mathcal{A}$ that projects the brush into the texture atlas, purely on the GPU, without needing to read any data back from the graphics card, and it requires only a single drawing step for any painting that is done. Furthermore it does not require the multi-step adaptive tessellation presented in Section 4.6 that was reducing the computational speed due to the communication between the GPU and CPU and the necessary synchronization.

As mentioned in Section 4.3 a notable feature of our technique is that it treats the brush as a contiguous object and does not simply project each single pixel individually into the texture atlas. The latter approach would create holes in the texture if the texture has a higher resolution than the brush, but our approach does not suffer from such sampling artefacts. Moreover, the runtime is independent of the mesh complexity.

5.2 The algorithm

Let us start by fixing the notation used in the description of our algorithm. The given 3D-object-mesh $\mathcal{S}$ consists of vertices $\boldsymbol{V} \subset \mathbb{R}^3$ and triangles $\boldsymbol{T}$, and usually each vertex $v \in \boldsymbol{V}$ has a unique associated texture coordinate $u \in \mathcal{T}$ in the 2D-texture-atlas $\mathcal{A} \subset \mathbb{R}^2$. These texture coordinates can be computed with any standard parameterization method and we assume them to be given. In our examples, we used the method

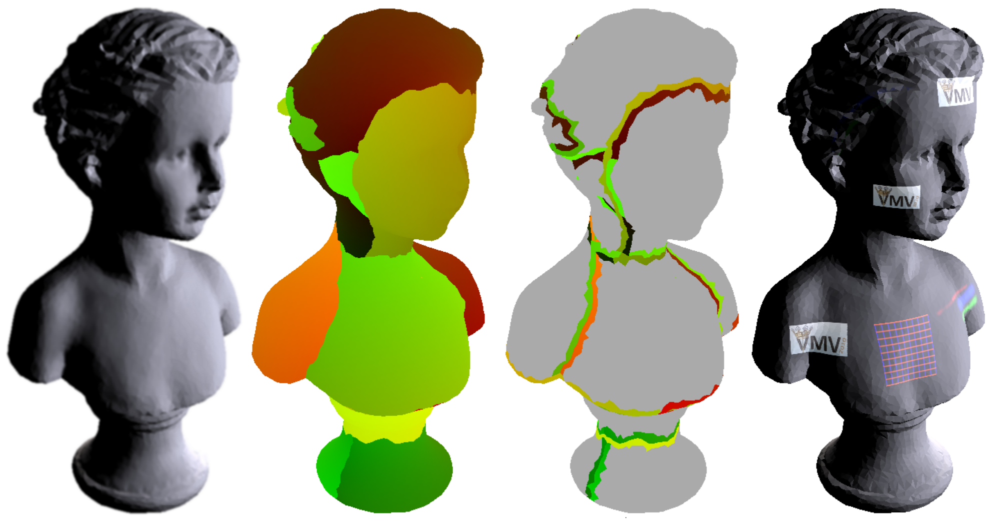


Figure 5.1. Overview of our method (from left to right): The first image shows the 3D mesh $\mathcal{S}$ that the user wants to paint onto. Note that it is segmented into several patches as becomes clear from the next image, which shows the content of the geometry buffer $\mathcal{G}$, i.e. the interpolated texture coordinates for $\mathcal{S}$, color-coded in the red and green components. The third image shows the virtual texture coordinates of triangles near the seams. The color coding is as for the second image and it can be observed that they continue the texture coordinates of each patch across the seams into the neighbouring patches. The rightmost image finally shows the result of a painting session; the corresponding texture atlas is shown in Figure 5.2.

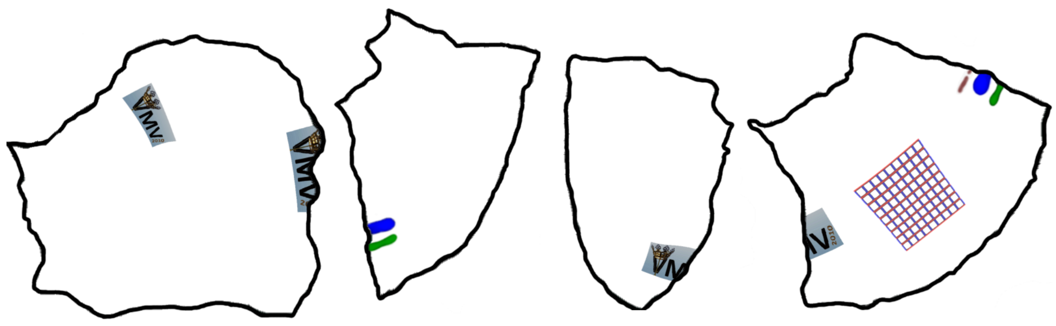


Figure 5.2. A part of the texture atlas used for the rightmost image in Figure 5.1. The thick black lines show the borders of the texture charts.

of Lévy et al. [33]. They induce the parameterization $\Phi\colon \mathcal{S} \to \mathcal{A}$, which linearly maps from each mesh triangle $T = [v_0, v_1, v_2] \in \boldsymbol{T}$ to the corresponding texture triangle $t = [u_0, u_1, u_2] \subset \mathcal{A}$.

If the mesh is split into $k > 1$ patches $\mathcal{S} = \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_k$, each with its own texture chart $\mathcal{C}_i$, then we occasionally add the chart index i to texture coordinates and triangles

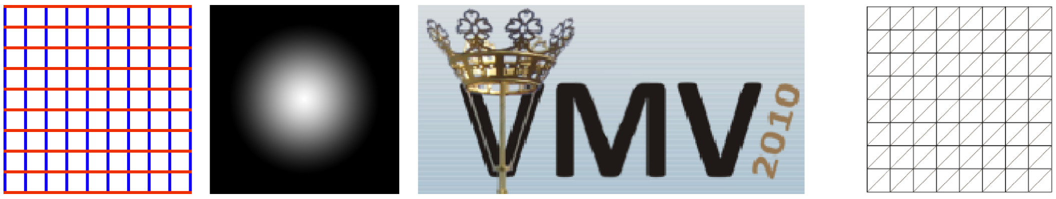


Figure 5.3. A small collection of brushes used for the example in Figure 5.1 and brush geometry for $n = 8$ (rightmost image).

in order to emphasize to which chart they belong (e.g., $u^i \in \mathcal{A}$ or $t^j \subset \mathcal{A}$). Moreover, any mesh vertex v on the common boundary of a pair of neighbouring patches $\mathcal{S}_i$ and $\mathcal{S}_j$ has two texture coordinates, u^i and u^j, one in each corresponding chart, and likewise for the few vertices where three (or even more) patches meet.

The brush $\mathcal{B}$ that is used for painting onto the object consists of the *brush texture* (a general 2D image) and the *brush geometry* $\mathcal{M}$, which is a simple regular 2D mesh with $(n+1)^2$ *brush vertices* $\boldsymbol{Q}$ and $2n^2$ *brush triangles* $\boldsymbol{M}$ (see Figure 5.3 right). The brush tessellation n depends on the distance between the camera and the object and is chosen such that size of the brush triangles is similar to the size of the mesh triangles in screen space (see Section 5.2.4 for details). Note that this changes only once after each repositioning of the camera and it does not require an adaptive tessellation of the mesh every time when a draw stroke is invoked.

More formally, we let the brush be the unit square $\mathcal{B} = [0,1] \times [0,1]$ and sample the brush texture at the given texture coordinates for any point $b \in \mathcal{B}$. Moreover, the brush vertices are distributed on a regular grid over $\mathcal{B}$, i.e. $\boldsymbol{Q} = \{(i/n, j/n) : 0 \leq i, j \leq n\}$, and so the whole brush is covered by brush triangles, $\mathcal{B} = \bigcup_{M \in \boldsymbol{M}} S$.

Now the main goal of our method is to efficiently implement the mapping $\Psi : \mathcal{B} \to \mathcal{A}$ on the graphics card and use it to copy the content of the brush texture into the the corresponding parts of $\mathcal{A}$ as described in Section 4.3. This is done by first projecting each brush vertex $q \in \boldsymbol{Q}$ onto $\mathcal{S}$ and then using the given parameterization Φ to determine the associated texture coordinate $q = \Psi(Q) = (\Phi \circ \Theta)(Q)$ of the projected point $\Theta(Q) \in \mathcal{S}$. Finally, we extend the mapping Ψ from the brush vertices to the brush triangles, i.e. for each brush triangle $M = [Q_0, Q_1, Q_2] \in \boldsymbol{M}$ we linearly map the brush texture to the corresponding triangle $m = [q_0, q_1, q_2]$ in $\mathcal{A}$.

5.2.1 Overview

We distinguish two possible user interactions: either the user changes the camera position or the viewing angle, or uses the brush to draw onto the mesh. In the first case, we

- draw the mesh on screen with texturing and lighting turned on, so that the user sees what he is interacting with;

- draw the object again into the *Geometry-Buffer* $\mathcal{G}$, which is a FrameBufferObject (FBO) that stores the interpolated texture coordinates of the mesh (see Section 5.2.2);
- adapt the resolution n of $\mathcal{M}$.

On the other hand, when drawing onto the mesh, the user moves a textured brush (see Figure 5.3) with the mouse over the screen and this texture needs to be copied (or alpha-blended) into the corresponding regions of the texture atlas, either continuously (painting) or when the mouse button is pressed (stamping). In order to do so, we

- draw the brush triangles on the screen and texture them with the currently chosen brush texture to give the user the feedback necessary for drawing onto the object;
- read the corresponding texture coordinates for the brush vertices from $\mathcal{G}$ and draw the brush triangles again, this time into $\mathcal{A}$, using the texture coordinates retrieved in the previous step (see Section 5.2.3);
- draw the mesh on screen with texturing and lighting turned on (as above) with the new texture information created in the previous step, so as to give the user direct feedback of his drawing action. Note that we do not need to re-compute the contents of $\mathcal{G}$.

Note that all steps can be done at interactive rates and depend neither on the complexity of the mesh (number of vertices and triangles) nor on the resolution of both the brush texture and $\mathcal{A}$.

5.2.2 Initialization

At the start of the program, both the object $\mathcal{S}$ (including texture coordinates) and its texture atlas $\mathcal{A}$ are loaded. Since we need to have write access to the texture on the graphics card, we store $\mathcal{A}$ as an FBO. The texture can either be an existing texture image or just an empty bitmap, set to a user-specified background color. An important feature of using an FBO as texture is, that FBOs allow to write negative values, and hence support additive and subtractive image manipulation.

We further instantiate $\mathcal{G}$ as a second FBO, whose resolution is identical to the view screen $\mathcal{V}$ in which the 3D-object is displayed. This $\mathcal{G}$ is used to store the interpolated texture coordinates of the object and it is filled in a second rendering step whenever the user has changed the camera settings. For each mesh triangle $T = [v_0, v_1, v_2]$ that is rendered into $\mathcal{G}$, we let the rasterizer linearly interpolate the texture coordinates u_0, u_1, u_2 of its vertices, and let the fragment program write the interpolated texture coordinate $(u, v) \in \mathcal{A}$ into the red and green component of $\mathcal{G}$'s pixels (see Figure 5.1). If the mesh consists of two or more patches, then we further use the blue component to store the index of the chart that each triangle belongs to (see Section 5.2.4 for details).

Thus, $\mathcal{G}$ provides an efficient evaluation of the parametrization $\Phi\colon \mathcal{S} \to \mathcal{A}$ on the GPU: for any surface point $O \in \mathcal{S}$ that is visible from the current camera position and hence has an associated screen coordinate $(x, y) \in \mathbb{N}^2$ in $\mathcal{V}$, we can simply get its texture coordinate $\Phi(O)$ by reading it from $\mathcal{G}$ at the coordinates (x, y).

5.2.3 Painting

During a painting session, the goal is to quickly transfer the brush texture into the texture atlas, and this essential part of the program must be as fast as possible, because it is carried out for every paint stroke. We implement this operation on the GPU by rendering the brush geometry in the following way.

Whenever a paint event is evoked, each brush vertex $Q \in \mathbf{Q}$ has a certain screen coordinate (x, y) that depends on the current position, orientation, and size of the brush. By associating with Q the surface point $O \in \mathcal{S}$ that is visible at pixel (x, y) in the geometry buffer, we effectively define a projection $\Theta\colon \mathcal{B} \to \mathcal{S}$ for each brush vertex with $\pi(Q) = O$. Note that this merely describes the underlying concept, but does not involve any computations. The real action happens in the vertex program, where we read $\mathcal{G}$ at (x, y) to retrieve the texture coordinate $\Phi(O)$ of the surface point O and replace the coordinate (x, y) of Q by $\Phi(O)$ before sending this brush vertex down the rest of the graphics pipeline. We additionally send the brush texture coordinate $(i/n, j/n) \in \mathcal{B}$ along with the other data, to retrieve the brush color value by sampling the brush texture for every fragment b that gets drawn into $\mathcal{A}$ with the best fitting coordinates.

As $\Phi(O) = \Phi(\Theta(Q)) = \Psi(Q)$, this modification essentially converts each brush triangle $M = [Q_0, Q_1, Q_2]$ into the corresponding texture triangle $m = [q_0, q_1, q_2]$, where $q_i = \Psi(Q_i)$. By now rendering this triangle into the FBO that contains the texture atlas $\mathcal{A}$, with texturing turned on, we effectively copy the brush texture that is given for each brush triangle M into the correct portion of the texture atlas.

We should emphasize here that the brush triangles form a contiguous cover of the brush, and so this way of implementing the (piecewise linear) map $\Psi\colon \mathcal{B} \to \mathcal{A}$ is guaranteed to create a contiguous copy of the brush texture in the texture atlas. I.e., even if the resolution of the texture atlas is much higher than that of the brush texture, it does not leave any relevant pixels in $\mathcal{A}$ unpainted, as it could happen if we would only splat the individual brush texture pixels into $\mathcal{A}$.

5.2.4 Seams

While the method explained in the Section 5.2.3 works well if the mesh consists of a single patch, a slightly more involved process is required for handling multiple patches, which is the usual situation for any non-trivial mesh. It can then happen that the brush overlaps two or more patches and so the brush texture must be split and copied into two or more disjoint regions in the texture atlas. We resolve this problem on the level of brush triangles.

First of all – as mentioned in Section 5.2.2 – we use $\mathcal{G}$ to also store the chart indices of the visible mesh triangles. So when looking up the texture coordinate q for some brush vertex Q in the vertex program, we also get the more detailed information that it belongs to some chart $\mathcal{C}_i$, i.e. $q = q^i$. After the primitive assembly, we then use a geometry program to check if the current triangle $m = [q_0^i, q_1^j, q_2^k]$ is contained in a single patch or spans across a seam by comparing the chart indices i, j, k. If they are all the same then we just proceed as usual (Figure 4.5, top), but if two of them differ, say $i \neq j = k$, then rasterizing m as it is would copy the brush texture for this triangle to the wrong region of the texture atlas (Figure 4.5, bottom left).

Instead, the correct solution in this situation is to create two instances $m^i = [q_0^i, q_1^i, q_2^i]$ and $m^j = [q_0^j, q_1^j, q_2^j]$ of m and to render them into the charts $\mathcal{C}_i$ and $\mathcal{C}_j$, respectively (Figure 4.5, bottom right). But how do we get the missing texture coordinates q_0^j, q_1^i, q_2^i for setting up m^i and m^j?

A straightforward solution is to simply enlarge each patch $\mathcal{S}_i$ by adding a ring of triangles to its boundary before computing the corresponding chart $\mathcal{C}_i$. Thus, if $\mathcal{S}_i$ and $\mathcal{S}_j$ are neighbouring patches and T is a triangle in $\mathcal{S}_i$ with at least one vertex on the seam, then it gets two corresponding texture triangles $t^i \subset \mathcal{A}_i$ and $t^j \subset \mathcal{A}_j$. While only the primary texture triangle t^i is used for texturing T when the mesh $\mathcal{S}$ is displayed, we need the secondary t^j to provide the missing texture coordinates above. More precisely, when initializing $\mathcal{G}$, we store the interpolated texture coordinates from t^i and the index i as RGB values as described in Section 5.2.2, but additionally store the interpolated texture coordinates from t^j and the index j as RGB values in a second color attachment of the FBO of $\mathcal{G}$.

Then, if the brush triangle vertex Q_0 is projected into this triangle, $\Theta(Q_0) \in T$, we first fetch its primary texture coordinate q_0^i from the $\mathcal{G}$ in the vertex program as in Section 5.2.2. If the geometry program later detects a seam overlap, we look up the secondary texture coordinate q_0^j in the same way from the second color attachment of the $\mathcal{G}$ in order to correctly set up the triangle m^j. Of course, the same strategy is applied to get the secondary texture coordinates q_1^i and q_2^i of the other two brush triangle vertices which are needed to specify m^i. Note that all this can be realized in the geometry program with just a single conditional branch and is thus very efficient. Figure 5.4 gives an overview of the described algorithm.

Near a mesh vertex v where three mesh patches meet, it can even happen that the vertices of a single brush triangle are mapped into three different texture charts. In this case, we need to create three instances of m, which in turn requires to store another secondary set of texture coordinates plus index for all mesh triangles in the one-ring around v, and we simply use a third color attachment to $\mathcal{G}$ for storing and accessing this data.

Even with this method of texture chart enlargement, it may occur that the secondary texture coordinates, which are needed to specify the several instances of a seam-overlapping brush triangle, are not available in $\mathcal{G}$. For example, this can happen

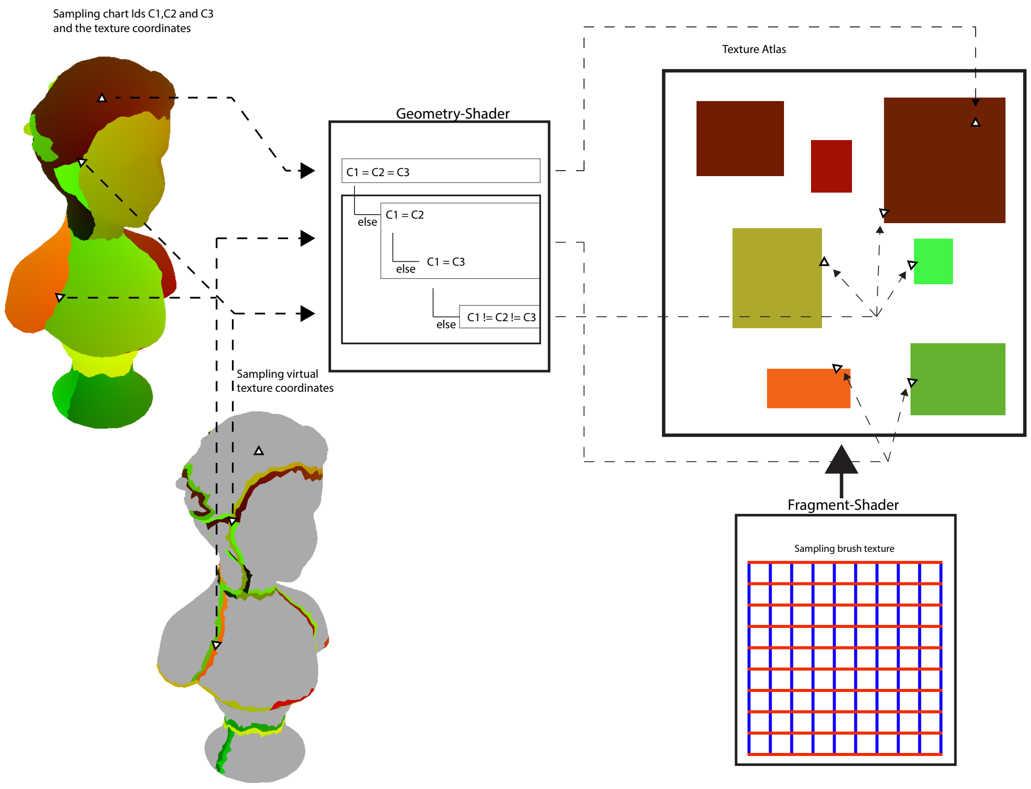


Figure 5.4. Overview of the painting algorithm. The geometry-shader receives the brush geometry and checks the chart-ids under each of the three triangle vertices. Depending on the result, it creates up to three triangles (using the sampled VTC) in the overlapped charts. The fragment-shader then samples the brush texture with the given texture coordinates of that brush triangle – its the same coordinates for all created triangles – and draws the result into $\mathcal{A}$ with the chosen blending function.

when the brush triangles are relatively big compared to the mesh triangles (in screen space) and then one of its vertices may end up being projected into a mesh triangle that is not adjacent to the seam and hence has no secondary texture coordinates in the neighbouring patch (compare Figure 5.1).

Our solution to this problem is twofold: first, we enlarge the patches not only by a single ring of triangles, but rather add two or even three rings around the boundary of each patch; second, we adapt the brush resolution n so that brush triangles and mesh triangles are of similar size. E.g., if the mesh is far from the camera, we need a high resolution of the brush, while a brush with two triangles is sufficient if the user has zoomed very close to the mesh.

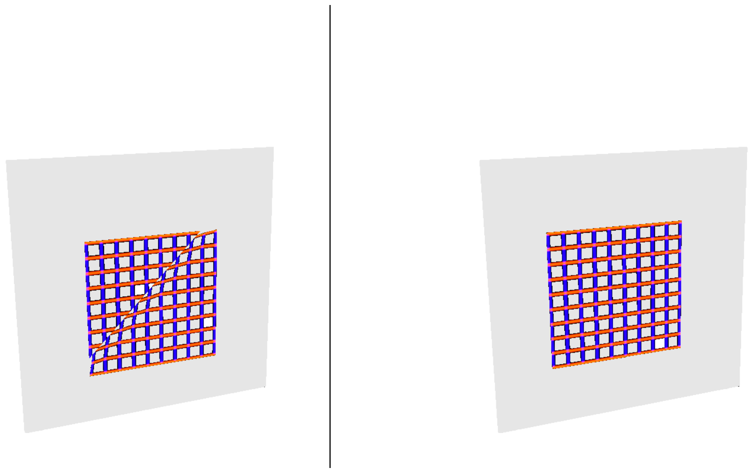


Figure 5.5. It is important to calculate the VTC very carefully. If the parametric distortion to both sides of the seam is not compatible, then the painted pattern gets strongly distorted when mapped back to the mesh (left), otherwise it works out nicely (right).

5.2.5 Virtual texture coordinates

Although the idea of providing secondary texture coordinates by enlarging and parameterizing the mesh patches as described in the previous section works conceptually, it has two major disadvantages:

1. It is often the case that a parameterization is given and can or should not be changed, for example, when a user wants to modify an already existing texture atlas.

2. By parameterizing the enlarged patches individually, it can happen that distortion across a seam edge between patches $\mathcal{S}_i$ and $\mathcal{S}_j$ is different in the corresponding charts $\mathcal{C}_i$ and $\mathcal{C}_j$, and this can yield a severe texture mismatch on both sides of the seam when mapping the texture back to the mesh as shown in Figure 5.5.

We overcome both disadvantages by enlarging not the patches, but rather the charts of a given parameterization (without modifying them) and by taking care of maintaining the same parametric distortion around corresponding chart boundaries. In order to distinguish the secondary texture coordinates computed by our method from the ones obtained by simply parameterizing enlarged patches, we call them *virtual texture coordinates* VTC. While computed differently, VTC are utilized by our method exactly as explained in the previous section.

Suppose $\mathcal{S}_i$ and $\mathcal{S}_j$ are neighbouring patches and that $[v_1, v_2]$ is one of the seam edges on their common boundary. Adjacent to this edge are the two triangles$T_1 = [v_0, v_1, v_2] \subset \mathcal{S}_i$ and $T_2 = [v_3, v_2, v_1] \subset \mathcal{S}_j$, with corresponding texture tri-

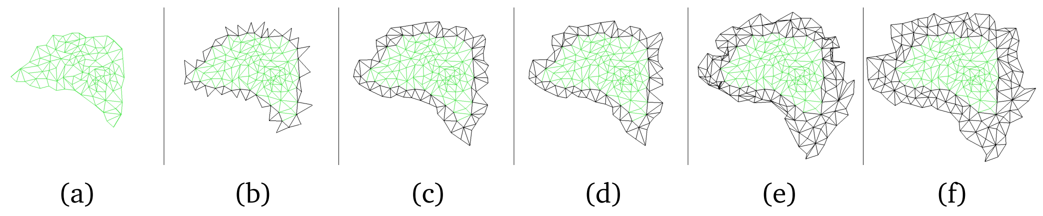


Figure 5.6. From left to right: chart of a mesh patch (a); VTC of neighbouring triangles with one seam edge (b); initial VTC of remaining vertices in the one-ring (c); result of optimizing the one-ring (d); initial VTC of vertices in the two-ring (e); result of optimizing the two-ring (f). Note how the optimization untangles the triangles and reduces the distortion.

angles $t_1^i = [u_0^i, u_1^i, u_2^i] \subset \mathcal{C}_i$ and $t_2^i = [u_3^j, u_2^j, u_1^j] \subset \mathcal{C}_j$, according to the given parameterization. In order to compute the VTC u_0^j of v_0 in $\mathcal{C}_j$ we

1. rotate T_1 about the common edge $[v_1, v_2]$ so that the rotated vertex $\tilde{v}_0$ and T_2 lie in the same plane;

2. determine the barycentric coordinates of $\tilde{v}_0$ with respect to T_2, i.e. we compute $\lambda_1, \lambda_2, \lambda_3$ such that $\tilde{v}_0 = \sum_{k=1}^{3} \lambda_k P_k$ and $\sum_{k=1}^{3} \lambda_k = 1$;

3. set $u_0^j = \sum_{k=1}^{3} \lambda_k u_k^j$.

In this way, the quadrilateral $\Diamond_j = [u_0^j, u_1^j, u_3^j, u_2^j]$ is an affine image of the quadrilateral $[\tilde{v}_0, v_1, v_3, v_2]$, and by computing the VTC u_3^i of v_3 in $\mathcal{C}_i$ analogously, we guarantee that the parametric distortions across the two corresponding chart boundaries $[u_1^i, u_2^i]$ and $[u_1^j, u_2^j]$ are compatible. That is, if we copy the brush texture into both quadrilaterals $\Diamond_i = [u_0^i, u_1^i, u_3^i, u_2^i]$ and $\Diamond_j$ as described in Section 5.2.4 and illustrated in Figure 4.5, and texture the mesh triangles T_1 and T_2 with the texture information stored in t_1^i and t_2^j, then these fit perfectly together along the common edge $[v_1, v_2]$, because $\Diamond_i$ and $\Diamond_j$ are just affine images of each other (see Figure 5.5).

While this fixes the VTC for the vertices of all triangles with one edge on a seam (see Figure 5.6 b), there usually remain a few more vertices in the one-ring around each patch boundary for which we still need to specify a VTC. For example, if $[v_0, v_1]$ and $[v_1, v_2]$ are two successive seam edges with adjacent triangles $[v_0, v_1, v_3]$ and $[v_1, v_2, v_4]$ and two triangles $[v_1, v_5, v_3]$, $[v_1, v_4, v_5]$ in between (all in the same patch and on the same side of the two edges), then the previous algorithms determines VTC u_3 and u_4 for v_3 and v_4, but not for v_5.

We first initialize this missing VTC by a simple linear interpolation $u_5 = (u_3 + u_4)/2$, and similarly if there should be more missing VTC between u_3 and u_4 (see Figure 5.6 c). We then minimize the parametric distortion for the affected triangles $[v_1, v_5, v_3]$ and $[v_1, v_4, v_5]$ by applying a few iterations (10 to 15) of the ARAP method [34] to get an optimized VTC u_5 (see Figure 5.6 d).

The whole procedure can be repeated to add more rings of VTC to each chart (see Figure 5.6 e,f), with the only difference, that from the second ring on, all added VTC can be optimized by the ARAP method, as only the VTC that were computed in the very first step above are constrained to remain unmodified so as to guarantee compatible distortion across the seam edges. The only case in which we deviate from this condition is when some of the initially computed virtual texture triangles overlap each other, which happens very rarely (less than 1%). Then we include the VTC that cause the overlap into the ARAP optimization so as to get rid of the overlap.

5.3 Summary

In this chapter we presented a practical implementation of the forward-mapping-idea presented in Section 4. It demonstrates that the data projection is fast, precise and without discontinuities even in the presence of seams. The technique smoothly draws over seams in the mesh by extending the data slightly outside of the current texture chart so that it can be used without errors when visualizing the object with bilinear texture interpolation.

The mesh painting algorithm stands out for three reasons: first, the underlying brush geometry guarantees that the brush texture is copied correctly into the texture atlas, regardless of the resolution of both the brush and the atlas.

Second, our method nicely handles the problem of seam-overlapping and provides a simple way of projecting a contiguous area from screen space correctly into the texture atlas, even if this area spans across several patches and thus needs to be mapped to several charts at separate locations in the texture atlas.

Third, all steps of our technique are implemented exclusively on the GPU, including all data access, which avoids expensive read back operations of data into the RAM. It exploits the natural flow of the graphics pipeline (vertex → geometry → fragment program) and needs to write only into those pixels of the texture atlas that are affected in each paint event, instead of testing for all texels whether the corresponding surface point is below the brush or not.

All this leads to interactive frame rates (more than 60 fps) on a modest Nvidia 9600 GT Mobile graphics card and is basically independent of the mesh complexity (number of vertices and triangles) and both the size of the brush texture and the texture atlas.

A nice feature of our method is that we can also handle the case where the user wants to draw onto more than one mesh, each with its own texture atlas. Using the *MultiDrawBuffer*-extension of FBOs, we can easily have the texture FBO contain more than one color buffer, and the fragment program that manages the copying of the brush texture into the texture atlas can decide into which texture to map, depending on the mesh which is currently being painted. It receives this information from the geometry program.

Despite the advantages of our method, it also has two limitations. So far, we do

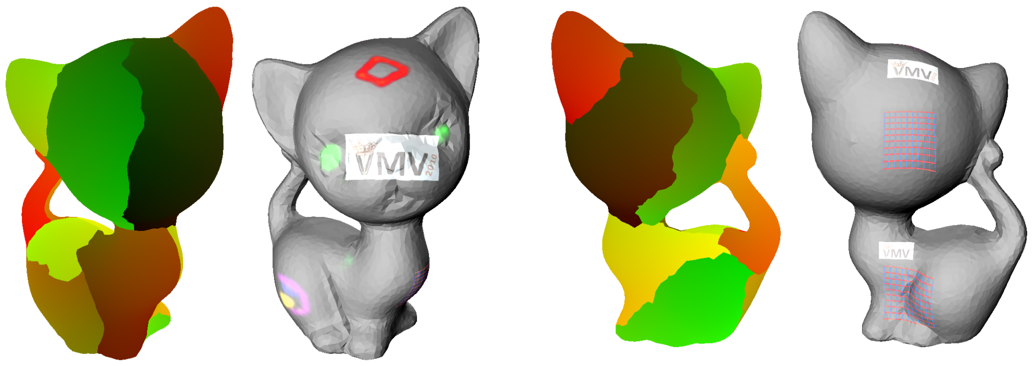


Figure 5.7. Two views (top and bottom row) of a mesh that has been painted with our method. The left side shows the content of the first layer of the TexBuffer, the right side shows the textured model as displayed to the user.

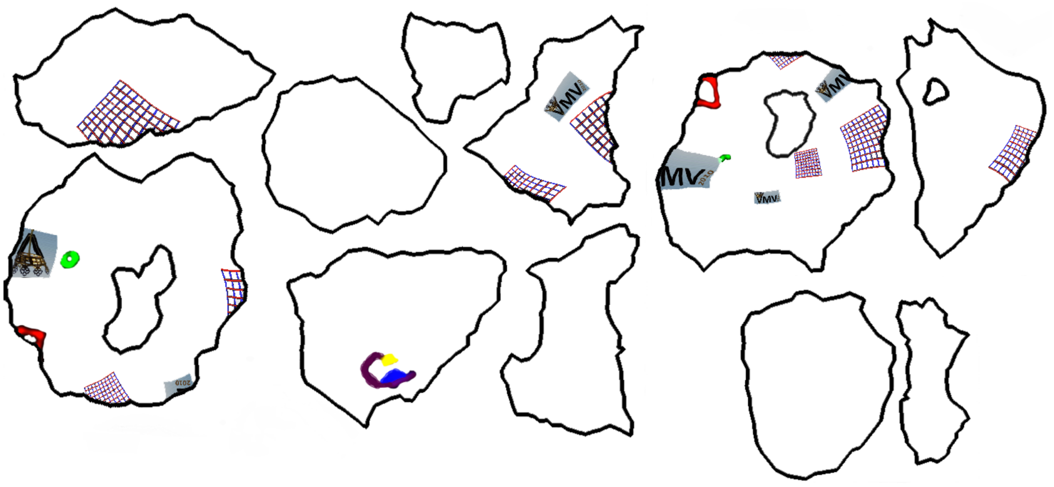


Figure 5.8. Texture atlas for the mesh in Figure 5.7.

not handle the situation where more than three mesh patches meet in a common mesh vertex. This could in principle be handled by adding further color attachments to the $\mathcal{G}$, but would require a much more complex (and slower) geometry program for distinguishing all the different situations that can occur in the case that a brush triangle overlaps this common mesh vertex. Moreover, our method of choosing the brush resolution adaptively may fail, if the mesh triangles are very non-uniform in size, so that a cluster of very small triangles resides next to a very large triangle on opposite sides of a seam. Then, adding any fixed number of VTC rings around the chart boundaries might not be enough to ensure that every brush vertex of a seam-overlapping brush triangle can read the required VTC from $\mathcal{G}$.

Chapter 6

Indirect Illumination

Although indirect illumination is essential for realistically rendering virtual scenes, it is rarely computed on the fly in interactive computer graphics, due to the expensive computations that are required. Instead, indirect illumination is usually either precomputed or created by artists and stored in a special texture map called *light map*. This resembles indirect lighting well enough to improve the visual quality of the scene but cannot react to changes in the lighting conditions. Although there are techniques that can be used to illuminate objects in real time by indirect light, they require the indirect lighting situation of the surrounding scene to be known. An extensive overview of the state-of-the-art is given in Chapter 2.

Most of these techniques only provide a rough approximation of the global illumination in order to improve the visual quality of rendered scenes, but they are not able to react to changes in the lighting situation. Other techniques, that achieve interactive rates, can approximate indirect illumination in a scene by computing only one or two light bounces, but this is insufficient for complex scenes, like the one in Figure 6.10, because it prevents the light to spread out far enough.

In general, most indirect illumination techniques perform the rather costly lighting computations in every frame instead of storing the results and reusing them for subsequent frames, which wastes valuable computational power, in particular if the scene is static and has little changes in the lighting conditions.

In this chapter we present a novel approach for approximating the indirect illumination of a virtual scene with multiple light bounces on the GPU and storing it in the *light atlas* $\mathcal{A}$ of the scene. This approach is not intended to compute direct lighting effects, which can be generated more efficiently with better quality using other standard techniques. Furthermore we restrict the computation to purely diffuse surfaces. The multi-bounce method that we present in this chapter captures the phenomenological properties of the exact diffuse light distribution with high quality in a couple of seconds up to a few minutes, depending on scene complexity and the desired quality of the result.

We achieve this by splitting the lighting computation into 2 separate parts (see Section 6.1). In the first step we compute an approximation to the overall light distribution within the scene (see Section 6.2). We then use the results of this computation to compute the final visible result in high quality (see Section 6.3). Our technique allows for fast computations of small changes in the lighting conditions by computing only the changed lighting conditions while reusing the unchanged previous results (see Section 6.4). In Section 6.5 we give an overview of the visual result.

The main contributions of this technique are:

- approximating a physically correct low-resolution light distribution with *multiple bounces*;
- converting the result into a high-resolution light distribution and transferring it to a *light atlas*;
- recomputing only necessary parts of the light distribution in case a single direct light source changes.

6.1 Basic idea

Since many scenes require significantly more than just one or two light bounces to produce realistically looking results (see Figure 6.10), we separate the basic light distribution from the gathering of these results into the final visible result.

So on one hand, the light distribution from a light source deep into the scene – around corners and obstacles – is an important part of giving a realistic appearance to a scene, but it is hard for an observer to judge the correct light distribution by looking at an image. The aspect of soft shadows created on visible surfaces on the other hand is very important and can create or destroy the illusion of looking at a real scene depending on the quality and consistency of the computed results.

Therefore we split the lighting computation into two separate steps with different goals. The first step concentrates on distributing the light energy roughly but physically plausible over the scene. To achieve this, we use pre-computed *virtual point lights* VPLs similar to the technique described by Lehtinen et al. [31]. In the light distribution these VPLs are used in a way similarly to patches in classical shooting-gathering approaches like [6, 7]. As the results computed in this first step are only intermediate and never visible to the user, an approximation is perfectly fine.

This fist step is a good approximation for the overall light distribution, but does not give a visually appealing result so far. To achieve that, we introduce a second step, that creates visually appealing and smooth results, capturing all the phenomenological effects of soft shadows, brightness gradients and color bleeding. This is implemented by rendering the lighting of all the VPLs into $\mathcal{A}$ which is then used for rendering the scene with the indirect illumination. Storing the computed lighting results allows us to reuse it as long as the scene configuration does not change.

In case that only few of the direct lights change at a certain point, we can also undo their – and only their – previous light contribution in $\mathcal{A}$. This is done by recomputing the first step for the light in question this time with the negative amount of energy, exactly undoing the effect that the changed lights had on the scene radiosity. We then compute their contribution with their new configuration – either a different direction, color, brightness, etc. – and check for the changes in the VPLs. We finish the computation by redrawing only these VPLs that changed with the energy that is the difference between the previous and the current radiosity.

Since we can also render with negative values into $\mathcal{A}$, the final result after these three computations will contain the correct light solution for the scene, without taking the unchanged lights into consideration. Note that the light atlas does not store the direct lighting effects and in all our examples we consider the indirect illumination only, since the direct lighting can be computed more efficiently with better results using existing rendering techniques.

The drawing into $\mathcal{A}$ is based on the Forward-Mapping-approach introduced in Chapter 4 but needs to be tailored to this specific problem.

Combining these two techniques, we achieve short computation times by distributing the light only coarsely in step one and get a high quality by switching to a high resolution rendering in step two when computing the results that are used to finally display the scene (Section 6.5). Furthermore, the rendering could even be performed in the background while the user is navigating the scene. That enables an early judgement of the lighting situation and the user can cancel the computation early on and change the lighting to better fit the intended effect. Using a low number of VPLs for a rough first decreases the necessary computation time further and still allows for judging the overall setting of the direct light sources involved.

Our algorithm, which is restricted to diffuse surfaces, computes realistically looking results for complex scenes. The results are smooth and the render time does not depend on the number of direct lights, since the initial distribution from these lights take only a fraction of the overall computation time (see Section 6.5). Possible applications, where these conditions apply include architectural light design, where most surfaces are diffuse and quick previews help speeding up the process of placing and configuring lights.

6.2 Coarse light distribution

We assume that the scene consists of diffuse surfaces and is illuminated by a number of direct light sources. The goal of the first step of our method is to quickly approximate the indirect light distribution over the scene in a physically correct manner using multiple bounces. We achieve this by discretizing the scene with uniformly distributed

VPLs (Section 6.2.1) in a precomputing step, where each VPL represents a small surface patch. The irradiance per VPL is then determined in two steps, first by distributing the power from the direct light sources to all VPLs with a standard shooting method [51] (Section 6.2.2) and then by iteratively distributing the irradiance amongst the VPLs (Section 6.2.3). The visibility between two patches is determined by using sample points associated with each VPL to get a more detailed distribution at low costs. The main part of the algorithm is parallelized and runs on the GPU, but the priority queue, which is needed for choosing the next VPL during the light distribution stage, is implemented on the CPU.

6.2.1 Scene discretization

We start by uniformly distributing n VPLs $V_1, \dots, V_n$ over the scene with the desired density (see Figure 6.1) and storing the normals $N_1, \dots, N_n$ and the reflection coefficients $\rho_1, \dots, \rho_n$ of the patches P_j represented by each VPL V_j. The distribution of VPLs is done in a pre-processing step that divides the scene into almost planar patches. Each patch is then triangulated with a pre-determined triangle area size, using the *triangle* tool [50]. The interior vertices of this triangulation are then used as VPL positions. This guarantees each VPL to have a certain offset to nearby corners, which could otherwise lead to artefacts in the light drawing stage. Furthermore, we create sample points associated to each VPL, where each V_j has m_j sample points S_{jk}, $k = 1, \dots, m_j$. The sample points are generated in the same way as the VPLs, but with higher density. The latter ensures that we can approximate the area A_j of patch P_j by the number of sample points,

$$A_j \approx m_j C, \tag{6.1}$$

$$C = \frac{A}{\sum_{j=1}^{n} m_j},$$

with A denoting the overall surface area of the scene.

6.2.2 Initial direct light distribution

We now assume that the scene is illuminated by k direct light sources $L_1, \dots, L_k$ with emitting powers $\tilde{E}_1, \dots, \tilde{E}_k$, each represented by a light patch with area $\tilde{A}_i$. Note that we use the tilde accent in this section to distinguish quantities that correspond to the direct light sources L_i from those that belong to the patches P_j, which are represented by the VPLs V_j. To distribute the light of the L_i to the VPLs, we compute the initial reflected power of P_j as

$$E_j = \rho_j \sum_{i=1}^{k} \tilde{F}_{ij} \tilde{E}_i, \qquad j = 1, \dots, n, \tag{6.2}$$

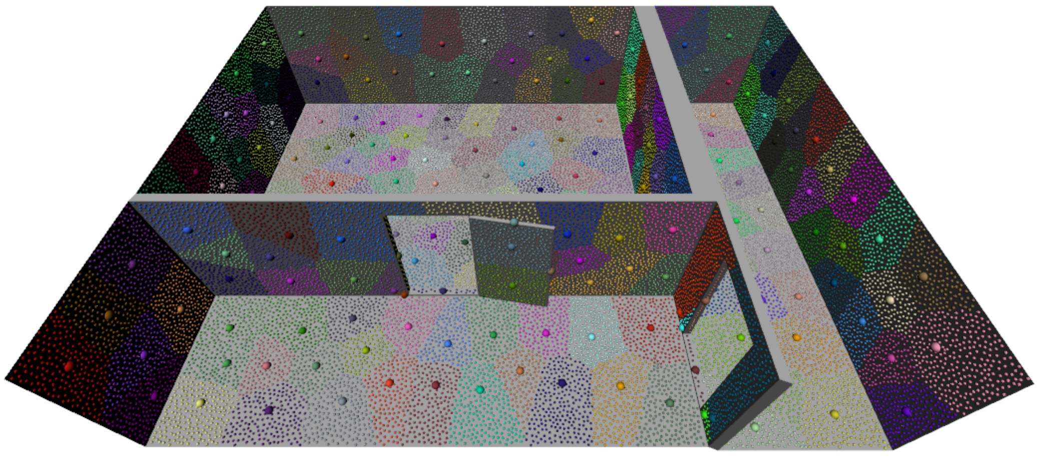


Figure 6.1. Distribution of VPLs (big dots) and associated sample points (small dots). In this example, the density of sample points is 30 times higher than the density of VPLs, so that the patches represented by each VPL become visible. In practice, it is sufficient to use about 10 sample points per VPL.

where $\tilde{F}_{ij}$ denotes the form factor that describes the geometric relationship between L_i and P_j. Under the classical assumption that the patches are small and not too close to the light sources, it can be approximated by

$$\tilde{F}_{ij} = A_j \frac{\langle l_{ij}, n_i \rangle \cdot \langle -l_{ij}, N_j \rangle}{\pi {d_{ij}}^2} \cdot \mathbf{s}_{ij}, \tag{6.3}$$

where n_i is the direction of light source L_i, N_j is the normal vector of patch P_j, l_{ij} is the normalized light vector from L_i to V_j, d_{ij} is the distance between L_i and V_j, and $\mathbf{s}_{ij}$ is a shadow or visibility factor that describes what percentage of the patch represented by V_j receive light from L_i. Note that for point lights the term $\langle l_{ij}, n_i \rangle$ is simply set to one, since there is no specific light direction.

Equation (6.2) is derived from the standard radiosity equation

$$B_j = \rho_j \sum_{i=1}^{k} \tilde{F}_{ji} \tilde{B}_i, \qquad j = 1, \ldots, n,$$

by replacing the emitted radiosities $\tilde{B}_i$ of the light sources and the reflected radiosities B_j of the patches with the respective powers $\tilde{E}_i = \tilde{B}_i \tilde{A}_i$ and $E_j = B_j A_j$, and by exploiting the reciprocity of the form factors, $\tilde{F}_{ij} \tilde{A}_i = \tilde{F}_{ji} A_j$.

In order to implement this strategy on the GPU, we store the data in two VBOs. The *patch VBO* contains the list of VPLs and stores for each V_j its position in world coordinates, the normal N_j, the number m_j of associated sample points, and an offset o_j into the sample VBO. This *sample VBO* is just the list of sample point positions, grouped by VPLs, so that the position of S_{jk} can be accessed with the index $o_j + k$. In addition, we use a *transform feedback buffer* that contains the *incoming* radiosities or *irradiances*

$$I_j = \frac{B_j}{\rho_j} = \frac{E_j}{\rho_j A_j} \tag{6.4}$$

of the patches P_j and is initialized with zero values. Note that we use the irradiance I_j instead of the power E_j or the radiosity B_j for efficiency reasons. Dividing both sides of (6.2) by $\rho_j A_j$, we get

$$I_j = \sum_{i=1}^{k} \tilde{F}'_{ij} \tilde{E}_i, \qquad j = 1, \ldots, n, \tag{6.5}$$

with the scaled form factor

$$\tilde{F}'_{ij} = \frac{\tilde{F}_{ij}}{A_j} = \frac{\langle l_{ij}, n_i \rangle \cdot \langle -l_{ij}, N_j \rangle}{\pi {d_{ij}}^2} \cdot \mathbf{s}_{ij}. \tag{6.6}$$

The advantage of computing (6.5) instead of (6.2) is that we do not need to access ρ_j. Moreover, we see that I_j is independent of the specific light patch areas $\tilde{A}_i$. Therefore, we do not need to specify the actual light patches and can describe each light source solely by its position, direction, and emitting power.

The direct light distribution is now implemented as a shooting method by iterating over the L_i. For each L_i we first render the scene from the light source's point of view and store the depth values in the FBO D. We then handle all VPLs in parallel by processing the patch VBO with a vertex shader. This shader determines the contribution of L_i to I_j and accumulates these values in the TFB, thus computing (6.5). While most terms of the form factor $\tilde{F}'_{ij}$ in (6.6) can be derived from the information associated with L_i and V_j, the visibility factor needs to be determined on the fly to account for changes in the scene configuration, like moving objects. To this end, we adapt the hemicube algorithm [5] and project the sample points S_{jk} of V_j, which are looked up in the sample VBO, into the scene, by multiplying them with the current modelview-projection-matrix. We then test the transformed z-components z_{jk} against the values in D, and compute $\mathbf{s}_{ij}$ as

$$\mathbf{s}_{ij} = \frac{1}{m_j} \sum_{k=1}^{m_j} \delta_k, \qquad \delta_k = \begin{cases} 1, & \text{if } z_{jk} \le \text{depth in } D, \\ 0, & \text{if } z_{jk} > \text{depth in } D, \end{cases}$$

where δ_k is essentially shadow mapping for each of the sample points.

6.2.3 Iterative light distribution

We finally distribute these initial radiosities iteratively amongst the VPLs with an adapted version of the classical progressive refinement algorithm [6]. We start by setting for each V_i the unshot irradiance ΔI_i to I_i from (6.5) and creating a priority queue

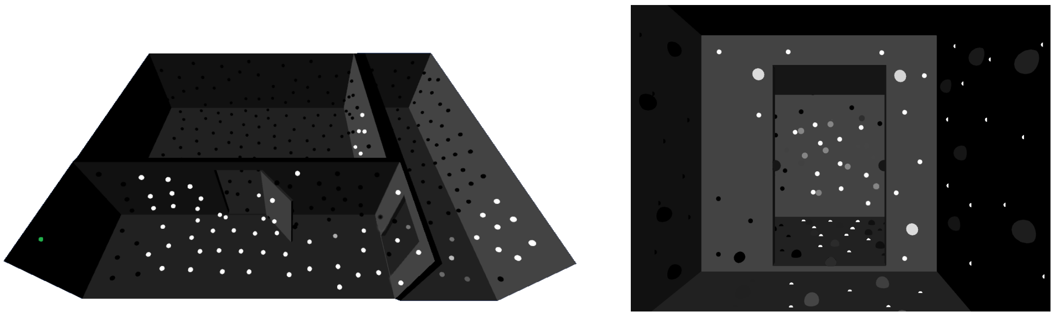


Figure 6.2. Light distribution from the active VPL V_i (green dot) to all other VPLs. The left image visualizes the VPLs that are occluded as black dots, and the brightness of the other dots corresponds to the amount of radiosity that each VPL receives from V_i. The right image shows a close-up to the door on the right, with the sample points (small dots) in addition to the VPLs. Sample points appear in black or white, depending on their visibility. Note how the ratio of visible and occluded sample points influences the brightness of the corresponding VPL.

of VPLs, sorted by the VPLs unshot power ΔE_i. According to (6.1) and (6.4), these values can be approximated as

$$\Delta E_i \approx \rho_i m_i \Delta I_i C, \tag{6.7}$$

and we compute and update them on the CPU after each shooting step. We then shoot the unshot irradiance of the first element V_i of the queue into the scene (see Figure 6.2) by computing

$$\begin{aligned} R &\leftarrow \rho_i F_{ji} \Delta I_i, \\ \Delta I_j &\leftarrow \Delta I_j + R, \\ I_j &\leftarrow I_j + R, \end{aligned} \tag{6.8}$$

for $j = 1, \dots, n$, set ΔI_i to zero, and update the priority queue. This process is iterated until the largest ΔE_i is smaller than some threshold. Here, the form factor F_{ij} describes the percentage of the power exchange from P_i to P_j and can be approximated similar as above by

$$F_{ij} = F'_{ij} A_j, \qquad F'_{ij} = \frac{\langle l_{ij}, N_i \rangle \cdot \langle -l_{ij}, N_j \rangle}{\pi {d_{ij}}^2} \cdot \mathbf{s}_{ij},$$

where l_{ij} is the normalized vector from V_i to V_j and d_{ij} is the distance between these two VPLs. Note that the scaled form factors F'_{ij} are symmetric in i and j, so that the right hand side in (6.8) simplifies to

$$\rho_i F_{ji} \Delta I_i = F'_{ji} \rho_i A_i \Delta I_i = F'_{ij} \Delta E_i.$$

The unshot power ΔE_i is taken from the priority queue, and the scaled form factors F'_{ij} and the visibility factor $\mathbf{s}_{ij}$ are computed as in Section 6.2.2 using the sample points S_{jk} associated with the receiving VPL V_j. The only difference is that we now iterate only over VPLs and no longer over direct light sources.

As in Section 6.2.2, the whole computation is done in parallel on the GPU and therefore very fast (see Table 6.1). The CPU is involved only for updating ΔE_i, sorting the priority queue and selecting the next VPL for distributing its unshot radiosity. Since the form factors are computed on the fly, we avoid the $O(n^2)$ storage overhead.

6.3 Filling the light atlas

After having distributed the radiosity amongst the VPLs, the goal of the second step of our method is to create a smooth high resolution light distribution and store it in the light atlas $\mathcal{A}$. We achieve this by shooting the light from each VPL into the scene and accumulating the irradiance for all texels in $\mathcal{A}$. The main idea of this algorithm is simple (Section 6.3.1), but special attention needs to be paid to seams and shadow edges (Section 6.3.2). We then improve both quality and timings by adding a per-texel soft-shadow algorithm (Section 6.3.3).

6.3.1 Final shooting step

For each texel τ of the light atlas $\mathcal{A}$ with coordinates (u,v) we compute its irradiance as

$$\mathcal{A}(u,v) = \sum_{i=1}^{n} F'_i(u,v)E_i, \tag{6.9}$$

where $F'_i(u,v)$ is the scaled form factor between V_i and the scene point $P(u,v)$ which corresponds to the texel τ. As in the first step of the algorithm (Section 6.2) we implement the computation of the irradiance in (6.9) as a shooting method by looping over the VPLs and accumulating their contributions in $\mathcal{A}$, but with one important difference regarding the visibility test.

During the distribution of light amongst the VPLs, the number of light receivers is relatively small and it is efficient to determine the visibility for each VPL. In this final step, however, the resolution of the light atlas is usually high, hence testing each texel for visibility is costly. Therefore, we propose a different strategy for finding the texels that are visible from the current VPL V_i. We overlap the scene, as seen from V_i, with a triangle mesh and project each triangle first onto the scene geometry and from there into $\mathcal{A}$ (see Figure 6.3). The projected triangles thus cover exactly that part of $\mathcal{A}$ which contains the texels that receive light from V_i.

In order to implement this strategy on the GPU, we first place a camera at the position of V_i, looking in the direction of N_i. We then create a uniform 2D triangle mesh $\mathcal{M}$ that covers the camera's viewport and render the scene into a geometry buffer

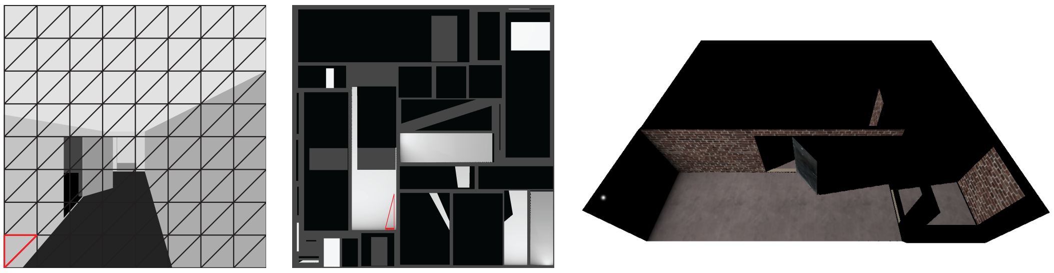


Figure 6.3. The light is distributed from a VPL into the scene by first rendering the scene from the viewpoint of the VPL and overlapping it with a 2D triangle mesh (left). Each triangle is then projected into the light atlas $\mathcal{A}$ (centre), and the light distribution is computed for all covered texels (right).

$\mathcal{G} = [\mathcal{T}, N, D]$ with three layers. These layers store the texture coordinates, the normals with respect to the local coordinate system of V_i, and the distance to the camera of the underlying scene geometry, respectively. This render step effectively performs the first part of the projection mentioned above, albeit in the inverse direction. That is, the scene is projected onto $\mathcal{M}$ rather than the other way around. We now draw $\mathcal{M}$ using a vertex shader that samples the texture coordinate buffer $\mathcal{T}$ at the coordinates (x, y) of each mesh vertex and replaces the latter with $\mathcal{T}(x, y)$. This turns each mesh triangle into a triangle in $\mathcal{A}$ and completes the projection process. We also sample the normal buffer N and the distance buffer D at (x, y) and send these values as vertex attributes further down the pipeline. In the fragment shader we therefore have all the relevant information for computing and accumulating the contributions $F_i'(u, v)E_i$ of each V_i to the texel at (u, v), except for the light vector l. Since the light vectors are constant with respect to the local coordinate systems of the VPLs, we precompute them for each pixel of a texture with the same resolution as $\mathcal{G}$, sample this texture at (x, y) in the vertex shader, and introduce this value as an additional vertex attribute. Note that this procedure automatically resolves the visibility test, because only visible texels are considered.

In order to correctly distribute the light of the VPL into the whole scene, we would have to set up the camera with a large opening angle, such that all visible parts of the scene are covered. But as this induces high distortions in the projection, we consider instead a hemicube to cover the 180° view from the VPL's position and use the geometry shader to replicate each triangle of $\mathcal{M}$ five times, once for each side of the hemicube.

6.3.2 Handling seams and shadow edges

The two-step projection process described in the previous section works as long as a triangle does not straddle a seam or a shadow edge. In these cases, the vertices of the triangle get projected into different charts in $\mathcal{A}$ so that the resulting texture triangle is wrong (see Figure 6.4).

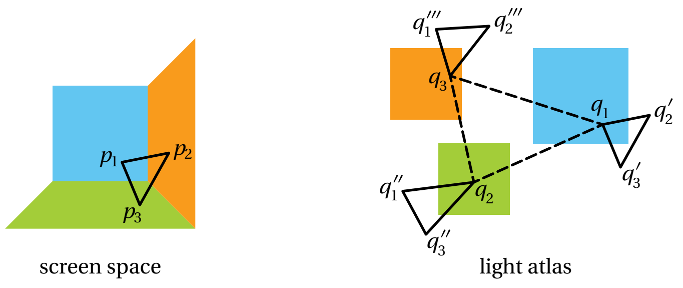


Figure 6.4. The vertices of a triangle that overlaps a seam (left) get projected to texture coordinates in different charts of the light atlas. As the charts may be packed arbitrarily, the resulting texture triangle (dashed) does not cover the correct texels, which in turn are the ones inside the three black triangles (right). A similar situation occurs at shadow edges.

To resolve this problem, we propose to replicate and project those triangles into each chart involved by reconstructing the missing coordinates. So far this requires an additional layer I in $\mathcal{G}$ which stores in each pixel the chart ID of the underlying scene geometry.

The challenge here is that not all the vertex coordinates of the projected triangles can be read from the geometry buffer $\mathcal{G}$. For example, in the situation shown in Figure 6.4, the copy of the mesh triangle $S = [Q_1, Q_2, Q_3]$, which is projected into the first chart, is missing the vertices q_2' and q_3'.

We resolve this problem by using Taylor's theorem to approximate these missing vertices as

$$q_k' \approx q_1 + \nabla\mathcal{T}(Q_1)(Q_k - Q_1), \qquad k = 2, 3.$$

However, this approximation is correct only if $\mathcal{T}$ varies linearly, but due to the perspective correction, which is applied to $\mathcal{T}$ by the GPU when the scene is rendered into $\mathcal{G}$, this is not the case. Therefore, we replace the layer $\mathcal{T}$ of $\mathcal{G}$ with the layer U, which stores the texture coordinates divided by the homogeneous w-coordinate. We turn off the perspective correction for this layer and instruct the GPU to also create the layer ∇U with the screen-space derivatives of U. Moreover, we add the layer W, which contains the reciprocal w-coordinates without perspective correction, and its derivatives ∇W to G. With these layers at hand, we can now compute the projections $q_k = \mathcal{T}(Q_k)$ of the mesh vertices Q_k as

$$q_k = \frac{U(Q_k)}{W(Q_k)}, \qquad k = 1, 2, 3, \tag{6.10}$$

and the missing vertices of the projected triangles as

$$q_k' = \frac{U(Q_1) + \nabla U(Q_1)(Q_k - Q_1)}{W(Q_1) + \nabla W(Q_1)(Q_k - Q_1)}, \qquad k = 2, 3, \tag{6.11}$$

and similarly for q_k'' and q_k'''.

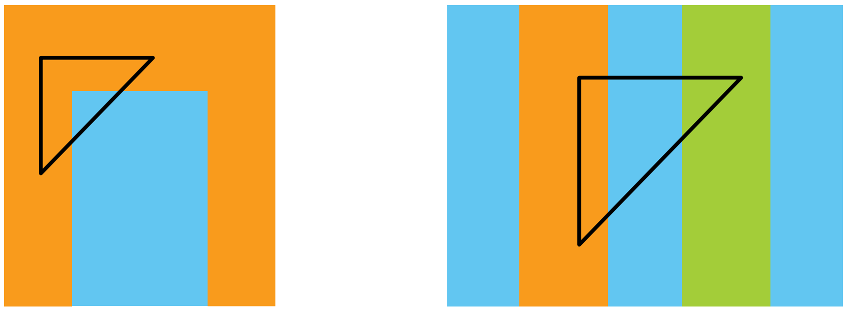


Figure 6.5. It may happen that a triangle of $\mathcal{M}$ overlaps parts of the scene (blue) that belong to charts that are different to the ones corresponding to the triangle vertices. This can happen, for example, at the corner of a door frame (left) or in the case of two pillars in front of a wall (right). In these cases, the overlapped texels in the blue chart do not receive any light.

The multiple projection strategy is implemented with a geometry shader, which processes the triangles of $\mathcal{M}$ in parallel. For each triangle $S = [Q_1, Q_2, Q_3]$ of $\mathcal{M}$, the shader first samples I to get the chart IDs of the vertices. If the chart IDs are all identical, then we replace the coordinates Q_k with q_k in (6.10) and handle the projected triangle $[q_1, q_2, q_3]$ as described in Section 6.3.1. Otherwise, we create as many projected triangles as there are different chart IDs and reconstruct the missing vertices using (6.11). Since these texture triangles also cover parts of the light atlas outside the corresponding chart (see Figure 6.4), we add a stencil test in the fragment shader which checks if the fragment is contained in the correct chart and otherwise discards it.

This stencil test relies on a precomputed version of $\mathcal{A}$. To create this stencil map $\mathcal{S}$, we render the scene into a texture, using the triangles (u, v) coordinates as targets for the resulting position. Each triangle is then rasterized with a single color that contains its chartID in the red-component. To allow drawing slightly over the chart boundaries for correct sampling results with enabled linear interpolation when rendering the scene, we extend this map in a second step. This second step checks each texel of the current map. If the red value is not equal to zero, no change is made. In case that the value is zero, we check the eight neigboring texels and color it with the ID that was encountered most often. If this texel is far between charts, it will still remain black, but if it lies on the outside of the boarder of an existing chart, it will simply take this charts ID value. Since a good UV-mapping should leave some distance between charts to avoid unwanted color bleeding, this simple technique is good enough to give an extended $\mathcal{S}$. Note that this map has to be computed only once and is valid as long as the scene patches do not change their IDs, which should never happen in a scene that is considered ready for light configuration.

The stencil test itself is then implemented by accessing this map in the fragment shader and comparing the sampled value – accessed at the target position in $\mathcal{A}$ with nearest filtering – with the ID sampled from layer I in $\mathcal{G}$. If the values are equal, the texel is written with the computed brightness, otherwise it gets discarded.

The main reason for implementing this complex repairing technique here instead of using one of the previously introduced is two-fold. We first tried adaptive tessellation, which lead to acceptable quality but was too slow due to the continues communication between the CPU and the GPU (see Section 4.6.2).

Using the precomputed *virtual texture coordinates* introduced in Chapter 5 is problematic in the case of scenes with very low tessellation, since here almost each edge is a border to a chart. Therefore each vertex is incident to at least three different charts. Therefore implementing this approach here would be too costly in sense of necessary $\mathcal{G}$ layer as well as sampling them for the necessary triangle repairs and extensions.

Although the described technique is very stable, it can lead to artefacts in certain situations. For example, even if all vertices of a triangle of $\mathcal{M}$ lie in the same chart, the triangle may still overlap a part of the scene which corresponds to a different chart (see Figure 6.5), and then this part does not receive any light. A similar situation can occur if the vertices of the triangle lie in different charts. In our experiments we observed that this happens rarely and does not lead to recognizable artefacts in the final indirect illumination solution, because of all the other VPLs that contribute light to the problematic regions. Moreover, the likelihood of such situations can be reduced by increasing the tessellation of $\mathcal{M}$.

6.3.3 Creating soft shadows

Figure 6.6 shows an example of the overall light distribution computed with the technique described so far and illustrates that realistic shadows are achieved only if $\mathcal{M}$ consists of a large number of triangles. The reason for this behaviour is explained in Figure 6.8, where the triangle $S = [Q_1, Q_2, Q_3]$ of $\mathcal{M}$ straddles a shadow edge, and Q_1 lies on scene object O_1, which casts a shadow onto scene object O_2 that contains Q_2 and Q_3. In this situation, the technique from the previous section creates and distributes light into the two extrapolated, projected triangles $s_1 = [q_1, q_2', q_3']$ and $s_2 = [q_1'', q_2, q_3]$ in the light atlas $\mathcal{A}$, which correspond to the scene triangles $S_1 = [Q_1, Q_2', Q_3']$ and $S_2 = [Q_1'', Q_2, Q_3]$. The first triangle extends the part of S belonging to the shadow caster O_1, and it is clipped to the correct region by the stencil test, because the shadow edge is also a boundary edge of O_1's chart in $\mathcal{A}$. The second triangle extends the part of S belonging to the shadow receiver O_2 and ranges into the shadow of O_1, thus causing light to spread into a region, which is not supposed to be illuminated. Clearly, this effect diminishes and becomes negligible if the resolution of $\mathcal{M}$ is high so that the projected triangles are sufficiently small.

However, since the resolution of $\mathcal{M}$ has an impact on the overall timing (see Figure 6.6), we propose to adapt shadow mapping to create realistic shadows even if $\mathcal{M}$

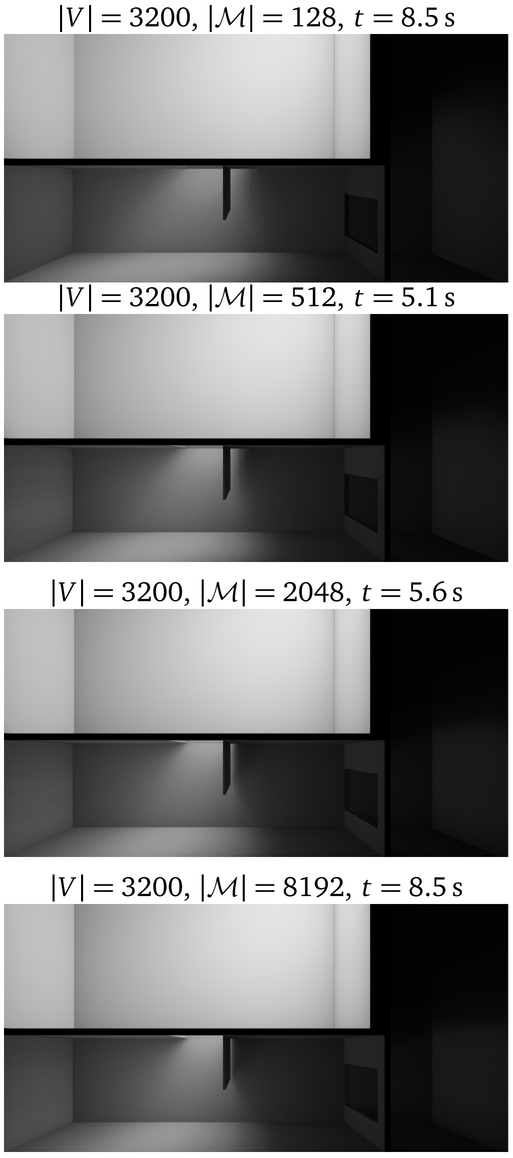

$|V| = 3200,\ |\mathcal{M}| = 128,\ t = 8.5\,\mathrm{s}$
$|V| = 3200,\ |\mathcal{M}| = 512,\ t = 5.1\,\mathrm{s}$
$|V| = 3200,\ |\mathcal{M}| = 2048,\ t = 5.6\,\mathrm{s}$
$|V| = 3200,\ |\mathcal{M}| = 8192,\ t = 8.5\,\mathrm{s}$

Figure 6.6. Light distribution using 3200 VPLs and $\mathcal{M}$ with 128, 512, 2048 and 8192 triangles (from top to bottom) for the projection step. Note that the shadow of the door appears correct only if the resolution of $\mathcal{M}$ is sufficiently high so that the projected triangles are small.

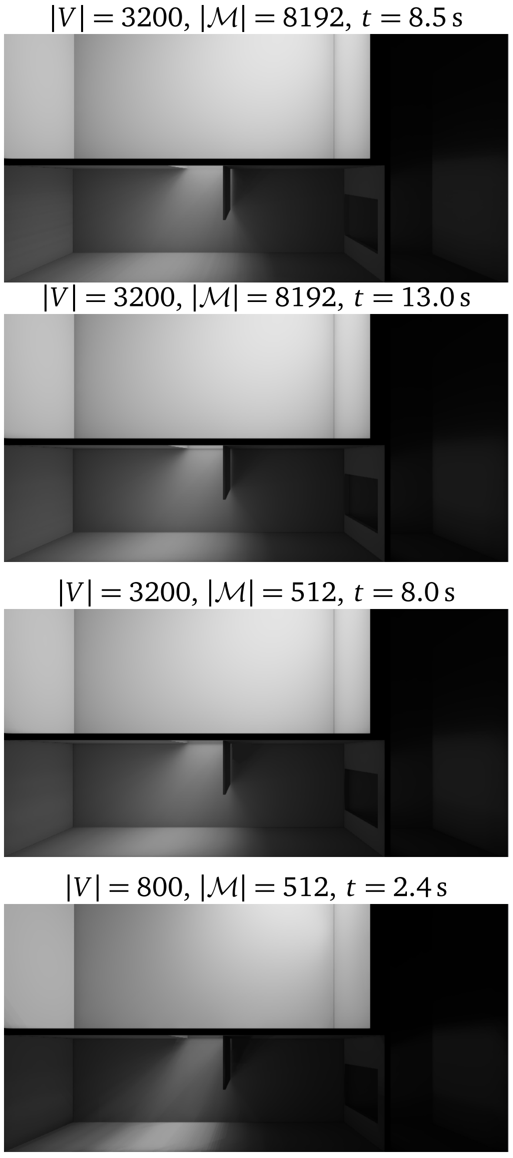

$|V| = 3200,\ |\mathcal{M}| = 8192,\ t = 8.5\,\mathrm{s}$
$|V| = 3200,\ |\mathcal{M}| = 8192,\ t = 13.0\,\mathrm{s}$
$|V| = 3200,\ |\mathcal{M}| = 512,\ t = 8.0\,\mathrm{s}$
$|V| = 800,\ |\mathcal{M}| = 512,\ t = 2.4\,\mathrm{s}$

Figure 6.7. Without soft shadows (top), a high resolution of $\mathcal{M}$ is required to create the correct shadow effect of the door (compare Figure 6.6). Using our soft shadow technique with $K = 5$ and the same resolution of $\mathcal{M}$ creates a computational overhead, but this can be counterbalanced by decreasing the resolution of $\mathcal{M}$ without sacrificing the quality of the result (centre). Moreover, with soft shadows we achieve the same quality with less VPLs, which in turn reduces the computation time (bottom).

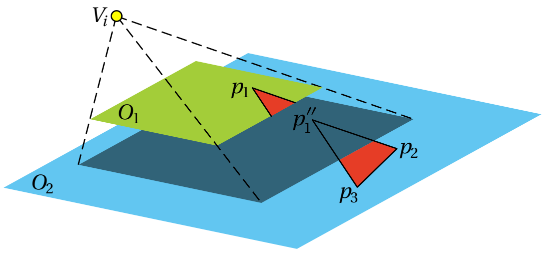


Figure 6.8. The red triangle of $\mathcal{M}$ is projected onto two different objects of the scene, and the part that lies in the blue object is extrapolated into the shadow of the green object.

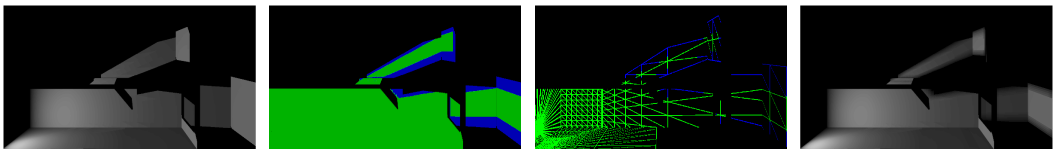


Figure 6.9. Light distribution for one VPL with the technique from Section 6.3.2 (left) and with soft shadows (right). The pictures in the centre show the projection of $\mathcal{M}$ with 512 triangles into the scene, where the green parts are illuminated correctly, while the blue parts are shadowed and receive light incorrectly due to the extrapolation process.

consists only of few triangles. To this end, remember that the fragment shader has access to the distance D between the camera and the scene point $P(u,v)$ that corresponds to the texel at (u,v). Note that for the texels in the extrapolated part of a projected triangle, this value may not be correct. For example, in Figure 6.8, the distance value D for the texel at q_1'' is not $\|V_i - p_1''\|$, but the smaller distance $\|V_i - p_1\|$. However, we can use this to our advantage, because it allows us to detect points that lie in the shadow of another object. All we need to do is to add another layer ∇D to the geometry buffer $\mathcal{G}$, which contains the derivatives of D, so that the depth value at q_1'' can be approximated as in (6.11) by the geometry shader. Hence, the texel is shadowed, if the extrapolated depth value is larger than D.

To further improve the quality, we use the approach of Fernando [14] to create soft shadows by considering the neighbourhood of $P(u,v)$. This requires us to introduce the coordinates (x,y) of the mesh vertices as additional vertex attributes, so that the fragment shader has access to the interpolated coordinates $(\bar{x},\bar{y})$. We then sample D at $(\bar{x},\bar{y})$ and the neighbouring coordinates and compare the values $D(\bar{x}+\delta_x, \bar{y}+\delta_y)$ for $\delta_x, \delta_y \in \{-K,\dots,K\}$, where K is the size of the sampling kernel, with the extrapolated depth value at $P(u,v)$, to find out what percentage of the neighbourhood of $P(u,v)$ are shadowed. The irradiance value for the current texel is then multiplied with this percentage before being drawn into the light atlas. Figure 6.9 illustrates this approach for one VPL, and Figure 6.7 shows the overall effect for the example in Figure 6.6.

6.4 Accelerating the computation for small changes

Realistic scenes contain many light sources that are mainly static, while only few might change over time. Our technique allows to quickly recompute the final lighting result in the case after small changes – e.g. a positional or directional change as well as color and brightness changes of a light source – by keeping the current lighting result and only updating the areas affected by the changed lighting condition. The idea behind this is to undo the effects of the changing light source with its old configuration and then to recompute the light contribution to the scene with the new setting.

The only additional memory necessary is a variable Δ for each VPL that keeps track of the difference of the power that each VPL will draw into $\mathcal{A}$ between the old and the new configuration of the currently changing light source L_i. Before starting a new recomputation step for L_i, all Δs are initialized with zero. Once this is done, we apply the first step of the indirect illumination computation (Section 6.2) for L_i only, but this time with the negative amount of emitted power that was used when previously computing the lighting results; all other parameters for L_i are kept.

Distributing this negative power over all VPLs exactly undoes the effect of L_i in its old configuration. This computed value is now stored in the newly added variable Δ for all VPLs. Note that all Δ-values should now either be zero, in case L_i had no effect on the VPL, or negative by exactly the value that L_i had distributed to that VPL in the previous lighting solution.

Once we have undone the old contribution, we change L_i's configuration and compute the first step again, adding the newly computed irradience values to the ones current stored in Δ. After the computation finished, each VPL has stored the difference between the old and new irradience values that L_i distributes in its Δ variable.

To give an example of the contents of Δ in two extreme cases: If L_i is simply turned off, all Δs contain the negative amount of energy that was drawn into $\mathcal{A}$ in L_i's original light distribution. In case L_i just got brighter, each Δ contains a slightly higher value than the one that was previously drawn into $\mathcal{A}$.

After the first step is done and the radiosity changes are computed for all VPLs, we now draw each VPL with a Δ value above a certain threshold into $\mathcal{A}$ as we did before in step two and update the distributed irradience from this VPL by the value of Δ during the following recomputation steps. Since we can also blend negative values into the FBO containing $\mathcal{A}$, the result is exactly the one we would have computed with the new light source position. To come back to the example of switching off a certain light, each VPL would draw exactly the negative amount of radiosity into $\mathcal{A}$ as was drawn with a positive sign in the original configuration, exactly undoing this lights contribution. This leaves the results from all other light sources unchanged. Details about the potential speedup of this approach are given in Section 6.5.2.

6.5 Results

All results mentioned in this section feature one spot light source and were computed on an Intel i5-3350P with 8GB RAM with an NVidia GeForce GTX 680. After first demonstrating the quality of our technique (Section 6.5.1), we discuss the timings and their dependence on various parameters (Section 6.5.2), and finally show that our approach can also be used for simulating the effect of big area lights (Section 6.5.4).

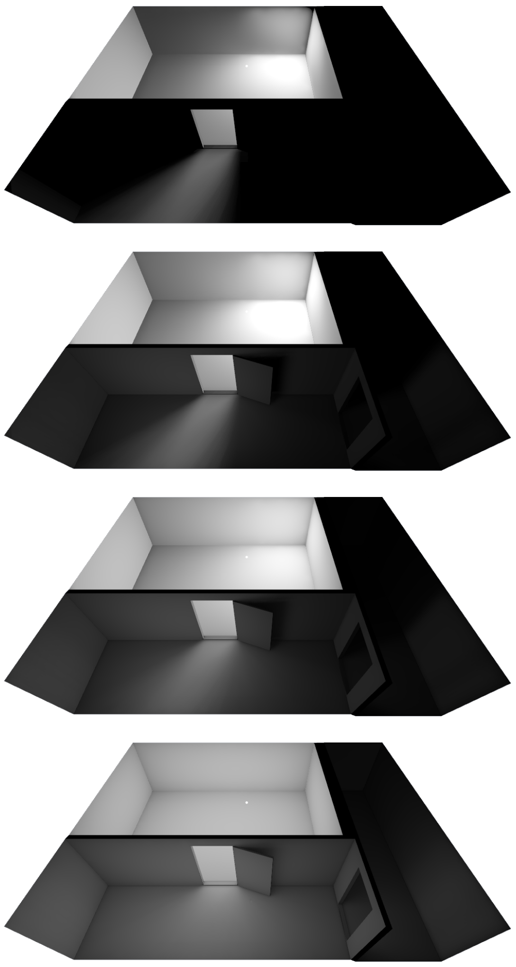


Figure 6.10. Indirect light distribution inside a test scene with 1, 2, 5, and 20 light bounces (from top to bottom) and 1000 VPLs.

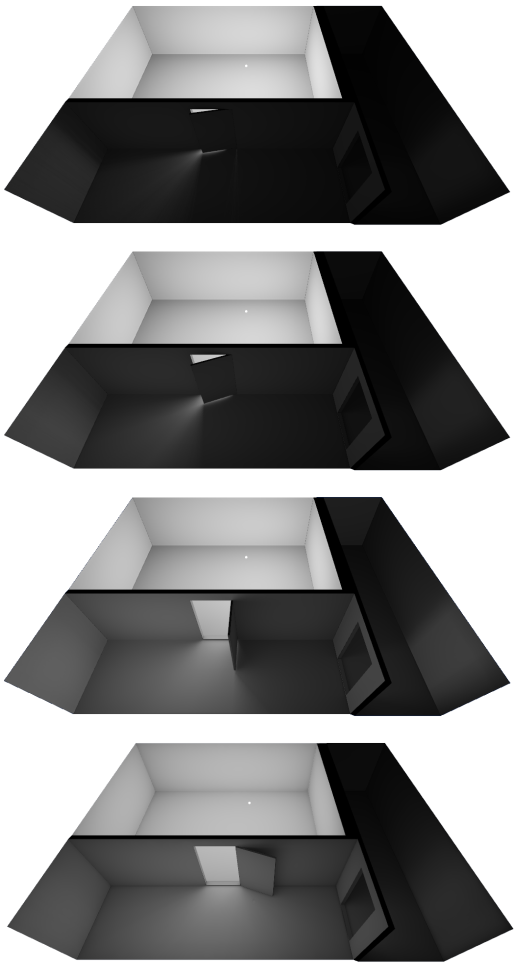
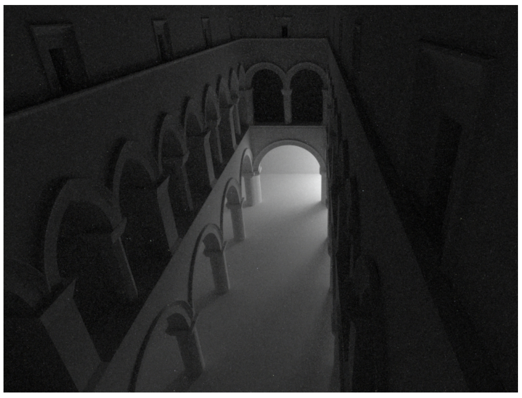
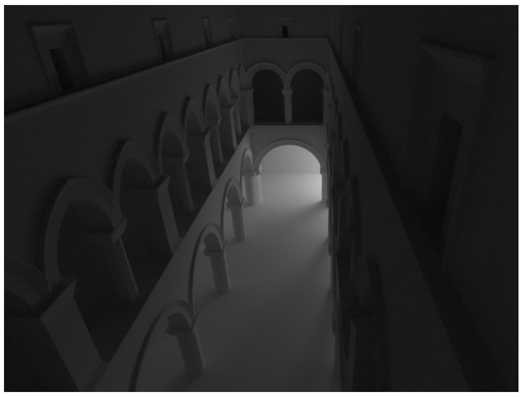
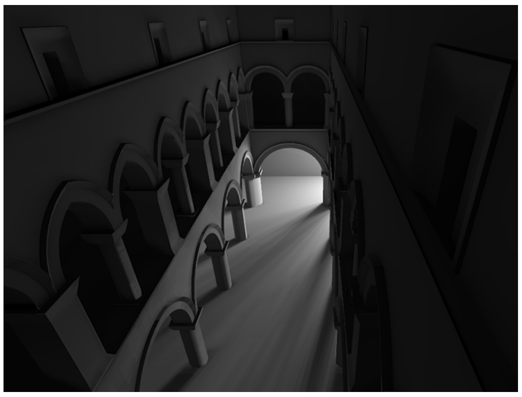
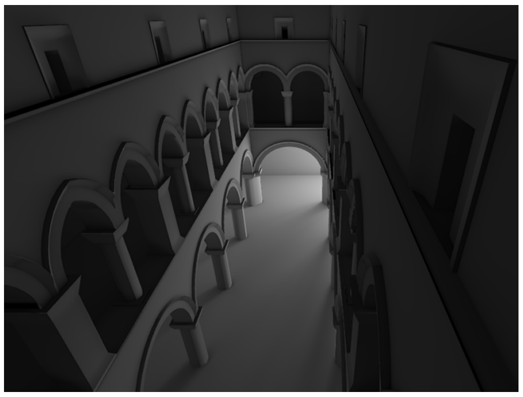


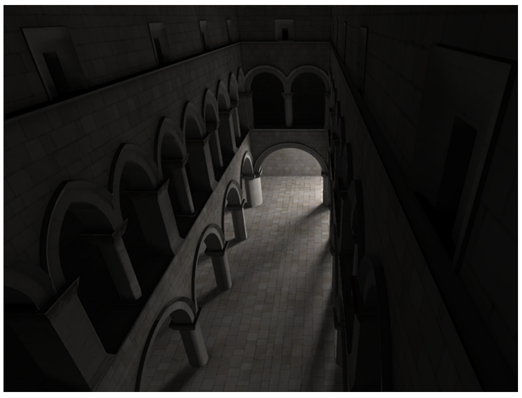
Figure 6.11. Indirect light distribution for the scene from Figure 6.10, with the door open by 10°, 20°, 90°, and 170° (from top to bottom).


$t = 228\,\mathrm{s}$


$t = 1.2\,\mathrm{h}$






$|V| = 6400,\ t = 29\,\mathrm{s}$

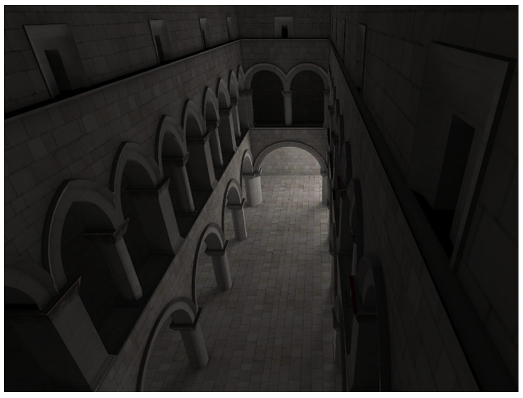

$|V| = 25600,\ t = 228\,\mathrm{s}$

Figure 6.12. The top row shows the Sponza scene rendered by *LuxRender* in about 4 minutes (left) and more than one hour (right). The images below show the results of our technique with different numbers of VPLs for comparison, and the last row shows the same results with textures.

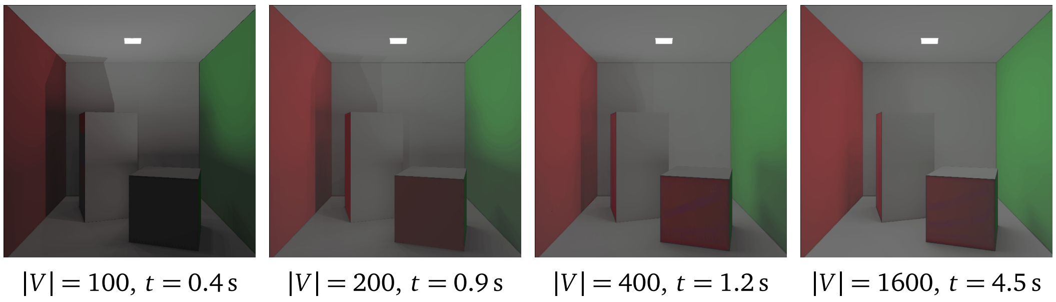

$|V| = 100,\ t = 0.4\,\mathrm{s}$
$|V| = 200,\ t = 0.9\,\mathrm{s}$
$|V| = 400,\ t = 1.2\,\mathrm{s}$
$|V| = 1600,\ t = 4.5\,\mathrm{s}$

Figure 6.13. Light distribution for the Cornell box with different numbers of VPLs.

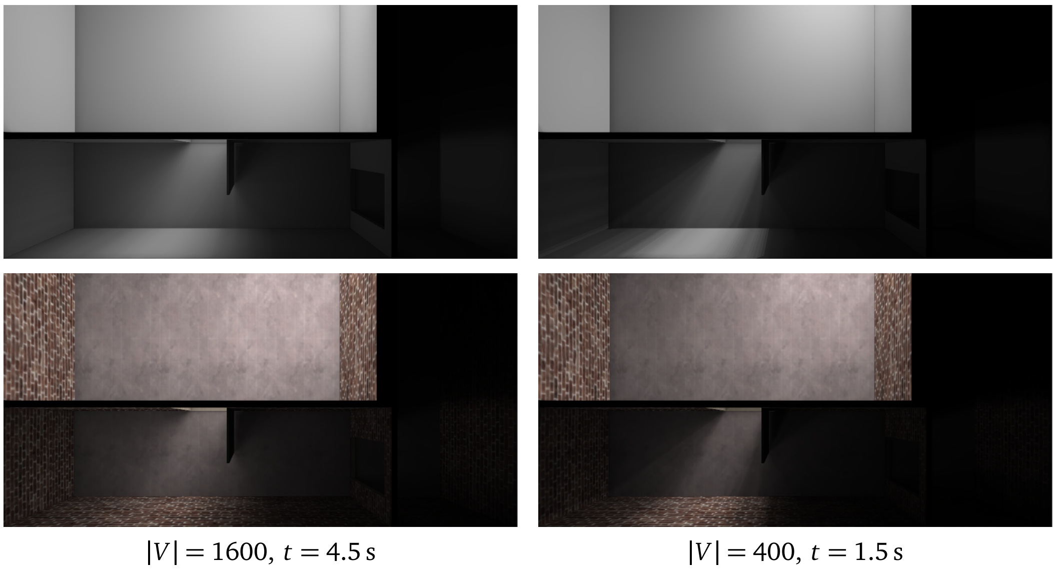

$|V| = 1600,\ t = 4.5\,\mathrm{s}$
$|V| = 400,\ t = 1.5\,\mathrm{s}$

Figure 6.14. Light distribution for the scene from Figure 6.10 with different numbers of VPLs. Using only 400 VPLs (right) creates visible artefacts in the light distribution, but they are barely noticeable once the scene is rendered with texture.

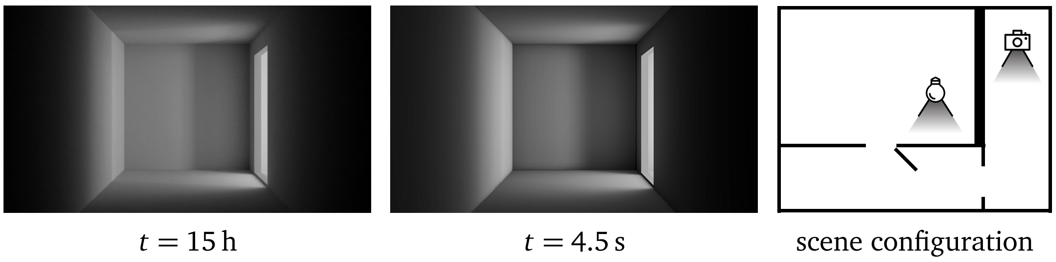


Figure 6.15. Comparison of the results using *LuxRender* (left) and our technique (middle) for the corridor scene, with a spot light source and the camera as shown in the sketch (right).

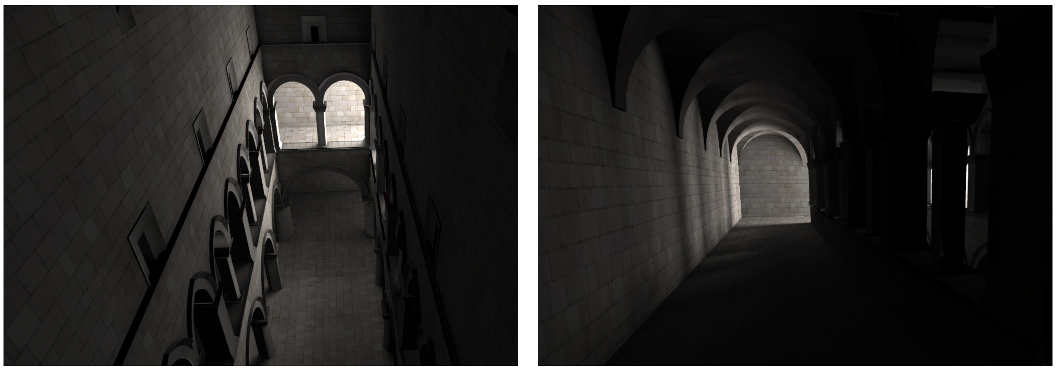


Figure 6.16. Indirect illumination of the Sponza scene as computed by our technique with 6400 VPLs and three light bounces in 29 seconds.

Note that all the images in this section are created by applying a tone-mapping to the results of the computation. This is necessary, since the overall brightness change between dark and bright areas is too big to be captured in the printed image or when being displayed on non-HDR-screens that have only 8-bit resolution per color channel. In interactive applications the visual quality can be further improved by compressing only the currently visible range of brightness into the LDR-image, which helps to preserve more details in the output.

6.5.1 Quality

Figure 6.10 demonstrates that the ability of our method to distribute light with more than two bounces significantly helps to create a realistic indirect illumination effect. Our tests suggest that five light bounces are a good compromise between quality and speed, and we use this setting in all our examples, unless stated otherwise. Figure 6.11 shows the same test scene, but with the door open at different angles. Note how our technique captures the lighting effect at the bottom of the door and how light is propagated by multiple bounces into the corridor on the right, even for small opening angles.

While the number of light bounces increases the realism of the result, it is the number of VPLs, which impacts the visual quality of the result, as shown in Figure 6.13. Even though the global lighting situation and colour bleeding appear to be correct for a small number of VPLs, using too few of them can lead to sharp, unnatural shadow edges and other small artefacts. Both are correctly smoothed out for a sufficiently large number of VPLs, depending on the scene geometry. In general, the more occluders a scene contains, the more VPLs are needed to achieve good results, and the same holds for the situation where light must to be distributed through a small hole or gap (see Figure 6.17).

Figures 6.14 and 6.12 further show that the number of VPLs can be decreased for textured scenes, because the texture tends to hide the apparent small artefacts of the

	timings								
	per VPL [ms]				for different numbers of VPLs [s]				
scene	fill D	distribute	fill $\mathcal{G}$	draw $\mathcal{A}$	100	400	1600	6400	25.6k
Corridor	0.0014	0.00011	0.28	5.05	0.70	**1.54**	4.53	16.98	159.63
Cornell	0.0013	0.00011	0.24	5.02	0.49	**1.21**	4.57	19.51	128.69
Sponza	0.0025	0.00011	2.10	8.04	—	—	7.73	**29.15**	228.25

Table 6.1. Timings for our test scenes. The left half of the table shows the average runtime per VPL for filling the depth buffer and distributing the light from one VPL to another (first step), as well as filling the geometry buffer and drawing into the light atlas (second step). The right half of the table shows the overall runtime of our algorithm, where the timing corresponding to the smallest number of VPLs that gives realistic results is marked in boldface.

indirect light distribution.

Figure 6.15 shows a comparison between our technique and the "ground truth", as computed by letting *LuxRender*'s path tracer run for 15 hours. The results are very similar, despite the fact that our approximation to the ground truth was computed in 4.5 seconds only. Note that in this example the light reaches the shown part of the scene only after three light bounces. Therefore, other methods with comparable timings, for example [9, 8, 44], would generate an entirely black image.

The closest competitor for our algorithm is the method by Luksch et al. [36], which also computes realistic results at interactive rates, but the advantages of our approach are threefold. First, we can handle more than two light bounces, which significantly improves the realism of the rendering result (see Figure 6.10). Second, our light distribution is decoupled from the scene geometry. This allows us to freely change the number of VPLs, which in turn gives us more flexibility to choose between speed and quality. Third, the creation of the light atlas is more efficient in our technique, because we only address those texels in the light atlas which receive light, instead of looping over all texels for each VPL.

While the static regularly distributed VPL positions have the advantage of allowing the recomputation of only changed light sources in contrast to stochastic distributed results as used in the photon mapper approach as shown in 6.4, it can also lead to shadow artefacts (see Figure 6.17). The reason for this is that only a small fraction of the uniformly distributed 400 VPLs is involved in distributing light through the door, especially when it is open by only 10°.

6.5.2 Timings

The overall runtime of our algorithm depends on a number of parameters: the number of light bounces and VPLs, the resolutions of the geometry buffer $\mathcal{G}$ and the 2D triangle mesh $\mathcal{M}$, and the scene complexity. The resolution of the light atlas $\mathcal{A}$ does not in-

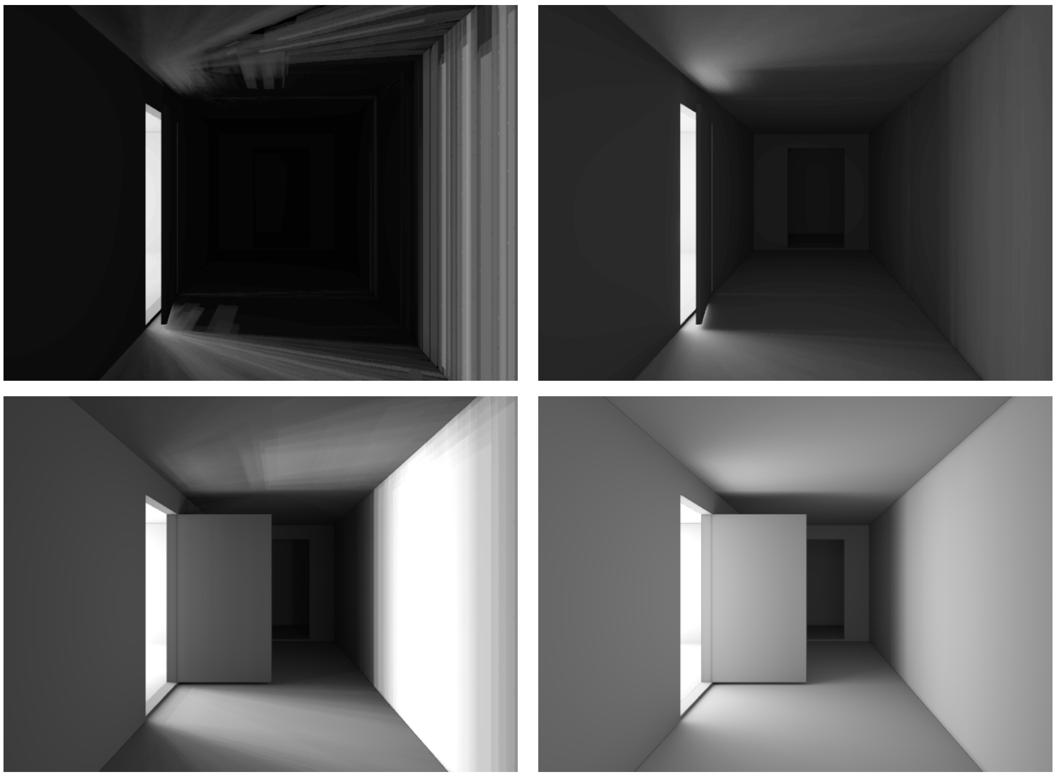


Figure 6.17. Using only 400 VPLs for the scene from Figure 6.11 with the door open by 10° (top) and 90° (bottom) leads to shadow artefacts (left), and it requires 10000 or 1600 VPLs, respectively, to get visually smooth results (right).

fluence the performance, because we draw only those texels of $\mathcal{A}$ which really receive light and this procedure is not fragment-bound.

The number of light bounces is relevant only for the first step of the algorithm and affects the runtime linearly. The resolutions of $\mathcal{G}$ and $\mathcal{M}$ are relevant only for the second step of the algorithm, and while the timings are almost independent of the resolution of $\mathcal{G}$, they depend linearly on the resolution of $\mathcal{M}$ (see Figure 6.6). In addition, both steps depend on the number of VPLs, and while the runtime of the first step scales quadratically in this parameter, the second step depends only linearly on the VPL count.

Figure 6.7 illustrates that our texel-accurate soft shadow technique effectually reduces the overall rendering time, because it leads to convincing results even for small numbers of VPLs and triangle meshes $\mathcal{M}$ with low resolution. In order for the soft shadow technique to be effective, the resolution of $\mathcal{G}$ needs to be higher than the resolution of $\mathcal{M}$. Our tests suggest that setting the resolution of $\mathcal{M}$ to $16 \times 16 \times 2 = 512$ triangles, the resolution of $\mathcal{G}$ to 256×256, and the kernel size K to 5 is a good compromise between quality and speed, and we used these settings in all examples, unless stated otherwise.

Table 6.1 shows that the average cost of the first step of our algorithm further depends on the complexity of the scene, because the visibility test requires to render the whole scene into a depth buffer for each VPL. The cost for the latter ranges from 0.0014 ms per VPL and per light bounce for the corridor scene in Figures 6.10 and 6.11 and the Cornell box in Figure 6.13 to 0.0026 ms for the Sponza scene in Figure 6.12. Instead, the time for distributing light from one VPL to another is constant for all scenes. However, while all other parts affect the overall runtime linearly, this part has a quadratic influence on the total computational cost. Filling the geometry buffer $\mathcal{G}$ in the second step of our algorithm also depends on the scene geometry. Fortunately, filling $\mathcal{G}$ is not very costly, and so this does not affect the overall runtime significantly. The latter is mainly determined by drawing the indirect illumination into the light atlas $\mathcal{A}$, and this part is independent of the scene complexity. An additional overhead is noticeable in both steps for the Sponza scene, because this scene comes with one light atlas for each of the three floors of the scene. Overall, the runtime per VPL for the Sponza scene is about twice as high, compared to the corridor scene and the Cornell box, and about 16 times as many VPLs are needed to achieve realistic results, because of the complexity of this scene.

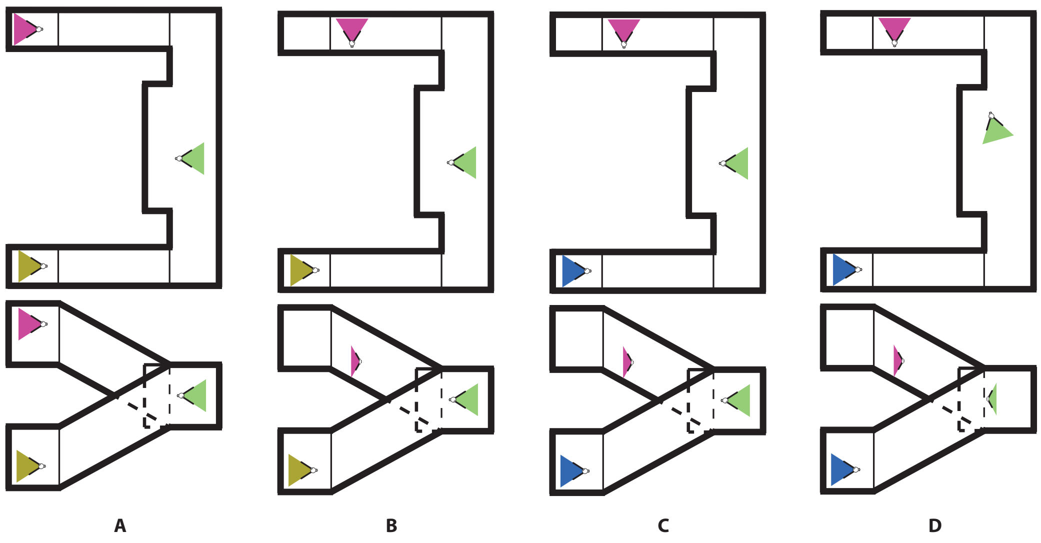


Figure 6.18. Sketch of the scene that is used for measuring the timings for the recomputation of a single light source. Scene A shows the lights in their starting position and direction. The next images show the change of each of the light sources at a time until the final configuration (D) is reached shown in the rightmost image.

The indirect illumination computation presented here can be easily tuned between very short computation times that lie in the range of seconds to perfect results (comparable to that of *LuxRender*) within a couple of minutes. The fast computation comes at the price of beautiful and smooth results, but the important thing is, that the results

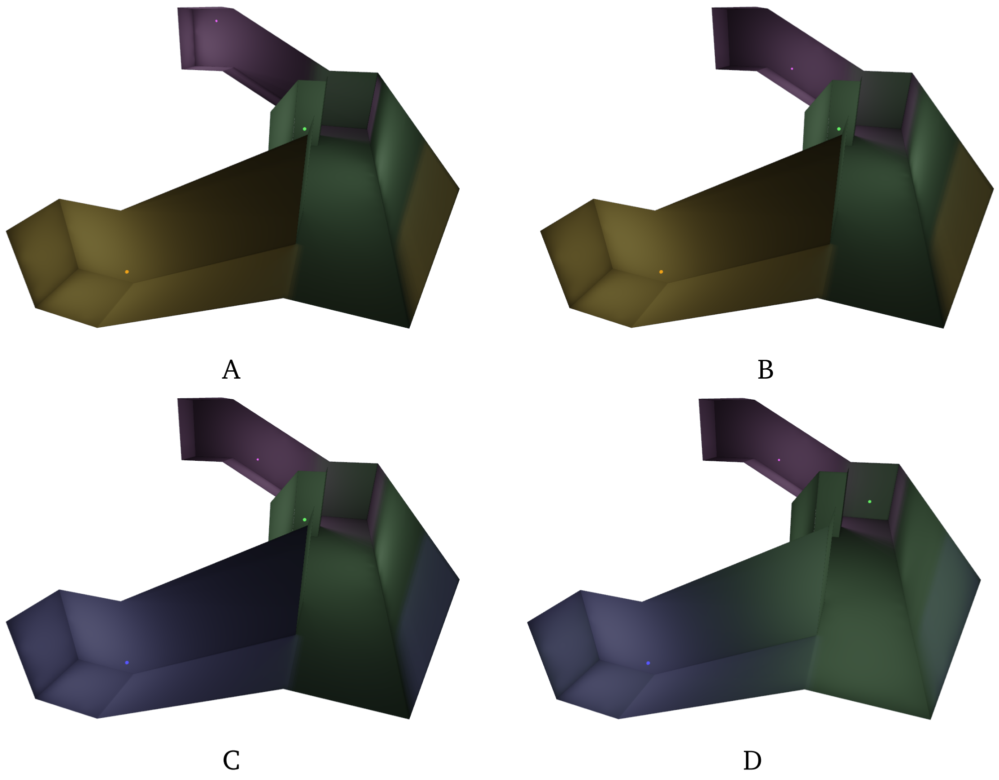


Figure 6.19. Visual results for the indirect illumination with the configurations A to D.

are never the less physically plausible, giving the user a good idea of the correct lighting situation. The high quality setting computes very similar results, but with perfectly smooth shadow edges. The high quality results are comparable to the ones created by state-of-the-art path-tracers, but are computed in a fraction of the time.

6.5.3 Recomputation timings

To test the efficiency of the recomputation of indirect illumination after the change of a single light source, we start with “scene A” shown in Figure 6.18 that contains three different light sources. We start by computing “scene A” from scratch, that is, no indirect illumination is computed and we run our algorithm to create the light map for the scene. To reach the other scene configurations, we only recompute the single light source that ahs changed from the previous step. From A to B the red light source is slightly moved away from the end of the corridor and turned by 90°. From B to C we change the yellow light source to now emit blue light and from C to D we change the position and rotation of the green light source.

The experiment performed with the scene shown in Figure 6.18 shows that even in a simple scene the indirect light contribution of different direct light sources overlap

configuration	active VPLs	distribution [s]	drawing [s]	overall
scene A	1008	0.356	1.579	**1.935**
scene B	1008	0.305	1.543	**1.848**
scene C	1008	0.414	1.546	**1.960**
scene D	1008	0.400	1.547	**1.947**
single light source change				
A→B	330	0.664	0.553	**1.207**
B→C	304	0.305	0.530	**0.835**
C→D	637	0.580	1.074	**1.654**
all 3 single steps	**1271**	**1.549**	**2.147**	**3.696**

Table 6.2. Timings for the scene used for the lighting recomputation test (see Figures 6.18 and 6.19). The rows labelled "scene A" to "scene D" show the data for computing the indirect illumination in this configuration from scratch. The rows labelled "$x \rightarrow y$" show the data for computing the result resulting scene y starting from scene x by just recomputing the indirect illumination that changed from configuration x to y.

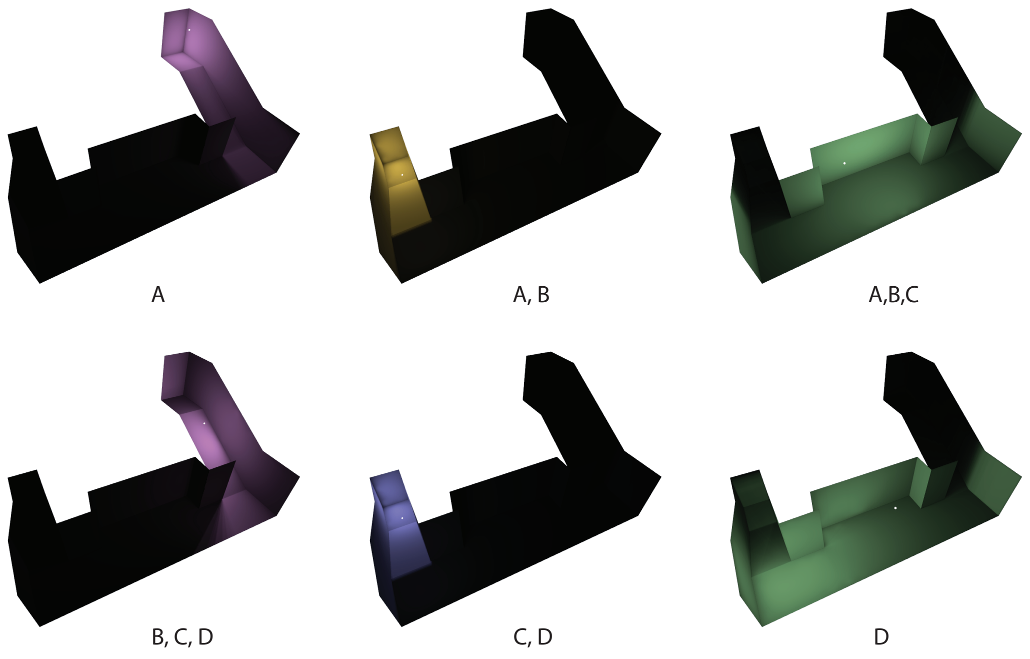


Figure 6.20. Results of the indirect illumination computation of the single lights in their initial configuration (top) and the changes created by setting them to their final configuration (bottom). Under each image is the scene in which the corresponding light configuration is active.

only partially. The results in Table 6.2 demonstrate that even in a simple scene like the one chosen for the experiment, the gain of recomputing a single light source instead of recomputing the whole lighting situation is significant. If in an editing case the user changes the three light sources one after the other as shown in Figure 6.18 from left to right, the overall time for the computation of the final result would take 7.6 seconds which results from recomputing the indirect illumination for the different configurations entirely. In case that only the changed lights are recomputed, the overall time is instead 3.6 seconds (see last row of Table 6.2). This is the result of drawing that fraction of the overall VPLs of the scene that have actually changed and drawing the value that results from subtracting the previously distributed radiosity from the currently distributed one.

The table shows that during the recomputation, the time for distributing the light over the VPLs increases by a factor of 2, as is to be expected due to the fact that we need to first compute the distribution of the light into the scene with a negativ value from its old configuration and to then recompute the distribution with positive values for the new configuration of the light source. But since the drawing of the radiosity into the light atlas is more costly than the light distribution, we achieve an overall reduction of the computation time that is significant even in this small scene. Note also that the complete recomputation of the radiosity can be reduced to a single step, if the light source only changes its color or intensity like in B→C. In this case the involved VPLs are completely the same and the light distribution can be done with the difference of the old lighting values and the newly chosen ones.

Applying this to a realistic scene with much more complex geometry and a much smaller overlap of indirectly illuminated areas for different direct light sources leads to a drastically shorter computation time. This in turn leads to a faster editing of lighting situations especially in huge scenes.

6.5.4 Area lights

Our technique for rendering VPLs into the light atlas can also be used to create realistic area light effects, as shown in Figure 6.21. We first distribute n VPLs uniformly over the surface of an area light with power E. Then we assign to each VPL the power E/n and fill the light atlas as explained in Section 6.3. In contrast to simple soft shadow algorithms, this method also correctly handles the shape of the area light, as well as the distances between the area light, the shadow caster, and the shadow receiver.

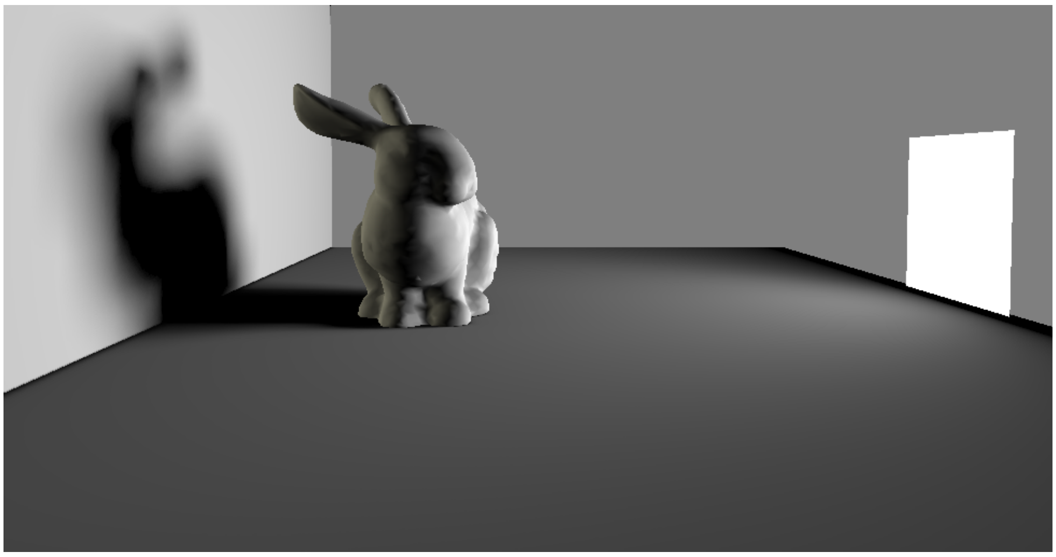


Figure 6.21. Area light effect, created with our technique using 100 VPLs in 0.6 seconds.

6.6 Summary

In this chapter we introduced a technique for computing light maps at almost interactive rates. We achieved this by separating the computation into a rather unprecise light distribution into the scene over virtual point lights, that captures the overall lighting situation correctly. In a second step we actually capture the light, by drawing the gathered radiance from each VPL into the light atlas. This second step is again heavily based on the idea of Forward-Mapping introduced in Chapter 4. Although the main idea is the same as in texture painting, a greatly different implementation had to be used due to the strongly different nature of the problem, e.g. more seams and separations of data given in screen-space due to geometric factors, as well as the very low tessellation of the scene geometry.

For the projection of the radiosity we decided to chose a mesh of a fixed tessellation and instead of just reading the position of triangles in the texture atlas, rather compute them. This needs a higher number of informations within $\mathcal{G}$, but the result is a fast computation with smooth results.

Note that we decided to store irradiance instead of radiosity in the light atlas, because we assume that the multiplication with the diffuse reflection coefficient is computed when the scene is rendered. In particular, this is the correct approach for textured scenes, where the texture itself provides a pixel-accurate description of the reflection coefficient.

Apart from the fact that our indirect illumination method handles multiple bounces and efficiently creates realistic results, it is worth noticing that it is particularly wellsuited for static scenes. In this situation, the light atlas is computed in a preprocessing step and can then be used to navigate the scene in real time.

Furthermore, our technique is fast enough to give an artist the possibility to create the desired overall lightning effect by interactively changing the numbers and positions of the light sources. In this setting, interactivity can be achieved by reducing the number of VPLs, and even though the rendering result with too few VPLs may exhibit artefacts, it still gives the user a realistic preview of how the result will look when a greater number of VPLs is used. It should even be possible to compute and display the result progressively with improving quality by increasing the number of VPLs on the fly, but it remains future work to further explore this idea.

Comparing the computation time and the quality of the results of the light map created by UDK and our technique – we could imagine this technique being used to improve the lighting design in architecture as well as in level-design for games by computing physically plausible results and giving almost immediate feedback to the user. The high quality setting can be used for creating light maps and in combination with the re-computation approach for small changes might be used in real-time applications, if the computation time can still be reduced by a little bit (especially considering that GPUs become faster at an incredible pace).

Chapter 7

Conclusion

In this thesis we present a novel approach for projecting data into a texture atlas entirely on the GPU and adapt it for a new method to compute indirect illumination that is very flexible and computes results that are close to the ground truth.

The first contribution is a novel forward mapping technique that runs entirely on the GPU and is therefore faster and more efficient than previous techniques in the field. With this method we answer RQ4, setting the basis for an efficient storing of radiosity.

The difficulty with this technique is foremost the problem of seams. We show three approaches to overcome the problem of projecting over seams that can all be easily implemented in the projection framework. All of these approaches are tailored to specific problems. This in turn demonstrates the flexibility of the projection method, thus answering RQ5.

With these solutions we decide to store the radiosity directly into a light map, which answers RQ3. While this is a standard approach for handling different complex effects in a virtual scene, our goal is to compute it at interactive or even realtime frame rates with a computed radiosity, that is as close as possible to the ground truth. To achieve this, we still need to answer how to compute the global illumination and how to store the radiosity.

We do this by adopting the standard shooting-gathering method and combining it with a multi-light approach. Both techniques are state of the art and can compute indirect illumination very precisely and fast. But multi-light approaches do usually not store the results in a way that can directly be visualized but instead compute the final result per frame using for example a light-cut approach, answering the first part of RQ6. With this data available we store it with an efficient projection of the data into the light atlas using the methods that answer RQ4 and RQ5. Therefore our algorithm does not require any more computations per frame, once the light atlas is filled, since during rendering this texture is sampled, which answers the second part of RQ6.

To increase the quality of the results, we choose to adopt one of many soft-shadow algorithms directly into the projection of the final results into the light map. The exact soft-shadow method can be replaced without affecting any other part of the algorithm, allowing for additional flexibility of our algorithm. This way we also answered RQ7.

The solutions above together form the answer to RQ1. But although the approach works fine, there is the always present RQ2: How can it be optimized. How can it be made faster or more realistic or more beautiful or at best all of the above together? While parts of the answer to this question remain for future work, we already optimized the approach. The main answer to RQ2 is to create good quality at a lower computational cost. While our first approach uses a huge number of VPLs to generate smooth results, answering RQ7 gives both a speedup as well as an increase in the quality. This is due to the fact, that we can use much fewer VPLs which reduces the computational time linear in their number, while only increasing the computational time per VPL by a small fraction necessary for creating the soft shadow effect. The important thing here is to find a perfect balance between computation time and quality and furthermore reduce the computation expenses that have only a marginal impact on the resulting quality.

Conceptually, our technique is somewhat similar to photon mapping. In the first part, the light is distributed with multiple bounces, but instead of shooting it randomly in any possible direction, the light exchange is restricted to the fixed VPL positions. In the second part, this discrete light distribution is extrapolated to the whole scene, but while a photon mapper requires a rather expensive averaging step, our projection strategy automatically generates a smooth light distribution in the light atlas. Consequently, our approach is more efficient and can illuminate scenes with high quality in a fraction of the time that it takes *LuxRender*, a state-of-the-art path-tracer. All phenomenological effects of the ground truth are present in our result despite the much shorter computation time.

Another technique that our method is obviously related to, is the idea of the many-lights-approach. While in the idea of many lights the lights are mainly distributed stochastically and then clustered, our idea is based on fixed VPLs, which makes that part much closer to the standard shooting-gathering-approaches. Furthermore, multi-light approaches compute the final result in each rendered frame, which is rather expensive even when using light-cut-techniques. Our technique, on the other hand, computes physically plausible results by carrying out the light distribution long enough and then only storing the final result once in a smooth and efficient way.

Overall, we present a technique that lies somewhere between the multi-light approaches and photon-mapper. It is able to compute very high quality physically correct results with multiple bounces – which most realtime techniques to not compute – in a fraction of the time that a photon mapper or path tracer would require to achieve results that are only slightly different from ours.

7.1 Future work

Although our technique computes results that are close to the ground truth at interactive rates, there is still room for improvements. We think that our technique can still be optimized and extended to reduce the computational time further, eventually even into the real-time range. Furthermore, the quality of the computation might be increased far enough to compute the results shown in this thesis with a much smaller number of VPLs, therefore decreasing computational time and memory consumption. Additionally, further effects like caustics and specular reflections might be implemented into the existing framework. Following are a few ideas that look promising for further investigation.

We choose to pre-compute and distribute the VPLs uniformly instead of creating randomly distributed VPLs on the fly [13], because we observed that the uniform distribution leads to smoother shadows and a higher overall rendering quality. Yet it would be interesting to explore the possibility of distributing the VPLs not uniformly but adaptively in the scene. As we show, the uniform distribution has the benefit of allowing for re-computation of single light changes in a multi-light environment, which is the most realistic scenario on one hand. But on the other hand, it might result in strong shadow artefacts in rare cases. One solution could be, to distribute the VPLs during the computation of the final result in a way that avoids these problems. Another approach that would still allow for static distribution and should be faster during light editing would be to analyse the source of the problem in a second pre-computing step, after the VPLs are uniformly distributed. While this would of course increase the time for the pre-computation, this would only have to be done once. In this step, additional VPLs could be created that take care of minimizing or entirely erasing any artefact in the light distribution later.

Another approach for improving the overall smoothness of the final result could be to increase the quality of the implemented soft-shadow algorithm. That could be done in the final rendering shader alone, and would only require to replace the currently used PCF method with any other technique. The tests done in this thesis show that there is a lot of potential for an increase of both speed and quality here.

It should furthermore be useful to apply the adaptive tessellation method introduced in Section 4.6.1 in a preprocessing step and store the result as a mesh per VPL. This might reduce the projection time for some of the VPLs that create huge – due to perspective distortion – triangles in $\mathcal{A}$ that are mainly clipped away by the stencil test.

Our technique could also be extended to render caustics and handle specular surfaces by combining it with the idea of *caustic triangles* [56]. The computation would then be done just once for the refractors and the caustics would be stored directly in the light atlas for rendering.

In principle, it would be possible to speed up our approach by clustering the VPLs [11], so that the light is distributed from a much smaller number of sources.

This would reduce the number of required rendering passes, but the small number of clusters would create artefacts as described above, even if soft shadow techniques are used to smooth out the shadow edges.

Another option for accelerating the first step of our method would be to pre-compute and store the form factors between VPLs with the visibility factor included. However, this would restrict the technique to work with static scenes only, whereas the current approach is flexible enough to also handle moving objects. The implementation of this approach is therefore strongly dependent on the overall usage of the method. In case that is light editing in static scenes, the implementation of this pre-computation will increase the speed significantly.

To allow an even faster feedback to the user during light editing, it might be interesting to implement the computation of the indirect illumination progressively. This would allow the user to interact with the virtual scene, while the computation is running in the background. The user would experience the improvements of the lighting condition mainly in the first few frames, while the later changes only affect the visual result slightly. With this approach real-time displaying of the results would be possible, while still giving physically correct results after a short computation time.

Appendix A

Notations and mappings

The following table gives an overview of the mappings and the notations used throughout this thesis. It contains the symbols and explanations for Chapter 4.

Name	abbreviation
3D-Scene	$\mathcal{S}$
Texture atlas	$\mathcal{A}$
2D-Brush	$\mathcal{B}$
Brush-Mesh	$\mathcal{M}$
Geometry-Buffer	$\mathcal{G}$
Projection view	$\mathcal{V}$
Screen-Resolution	R_S
Data-Resolution (later brush tex-res)	R_T
Transformfeedback-Buffers	$\mathbf{T}_1$ and $\mathbf{T}_2$
Chart	$\mathcal{C}$
Texture-Domain	$\mathcal{T}$
Mapping for drawing	$\Psi\colon \mathcal{V} \to \mathcal{A}$
Splitted mapping	$\Psi = \Phi^{-1} \circ \Theta$
Modelview-Projection	$\Theta\colon \mathcal{S} \to \mathcal{V}$
2D-parameterization	$\Phi\colon \mathcal{S} \to \mathcal{A}$
Texture-mapping	$\Phi^{-1}\colon \mathcal{A} \to \mathcal{S}$
Data-Projection	$\Pi\colon \mathcal{V} \to \mathcal{M}$

In Chapter 5 the notations are similar to the ones used in Chapter 4, but were updated and extended when it become necessary. The following table gives and overview.

Name	abbreviation
3D-Scene	$\mathcal{S}$
Texture atlas	$\mathcal{A}$
2D-Brush	$\mathcal{B}$
Brush-Mesh	$\mathcal{M}$
Geometry-Buffer	$\mathcal{G}$
Projection view	$\mathcal{V}$
All Scene-vertices	$\boldsymbol{V}$
Single triangle vertex	v
Number of patches and charts	k
Mesh-triangles	$\boldsymbol{T}$
Texture-coordinate	u
Brush vertices	$\boldsymbol{Q}$
Brush-Triangles	$\boldsymbol{M}$
Brush-triangles in $\mathcal{A}$	m

The following are the notations used in the Chapter 6. We added notations regarding the indirect illumination computation and updated the symbols for the forward mapping technique where it had to be adapted to the needs of storing indirect illumination.

Name	abbreviation
Brush-Mesh	$\mathcal{M}$
Projection view	$\mathcal{V}$
All Scene-vertices	$\boldsymbol{V}$
Single triangle vertex	v
Number of patches and charts	k
Mesh-triangles	$\boldsymbol{T}$
Texture-coordinate	u
Brush vertices	$\boldsymbol{Q}$
Brush-triangles	$\boldsymbol{M}$
Brush-triangles in $\mathcal{A}$	m
Shadow mapping kernel-size	K
Geometry-Buffer	$\mathcal{G}$
TexCoord-part of $\mathcal{G}$	$\mathcal{T}$
VPL for patch i	V_i
Sample point i for patch j	S_{ji}
Direct light source i	L_i
Emitting power of L_i	$\tilde{E}_i$
Direction of L_i	n_i
Normal of patch i	N_i
Formfactor (light i and patch j)	$\tilde{F}_{ij}$
Formfactor between patches i and j)	F_{ij}
Area of patch i	A_i
Visibility term between i and j	$\mathbf{s}_{ij}$

Bibliography

[1] Maneesh Agrawala, Andrew C. Beers, and Marc Levoy. 3D painting on scanned surfaces. In *I3D '95: Proceedings of the 1995 Symposium on Interactive 3D graphics*, pages 145–150, 1995.

[2] Okan Arikan, David A. Forsyth, and James F. O'Brien. Fast and detailed approximate global illumination by irradiance decomposition. *ACM Trans. Graph.*, 24(3):1108–1114, 2005.

[3] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977. ISSN 0097-8930.

[4] Edwin Earl Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, 1974.

[5] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: a radiosity solution for complex environments. *scg*, 19(3):31–40, July 1985. Proceedings of SIGGRAPH.

[6] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. *scg*, 22(4):75–84, August 1988. Proceedings of SIGGRAPH.

[7] Greg Coombe, Mark J. Harris, and Anselmo Lastra. Radiosity on graphics hardware. In *Proceedings of Graphics Interface*, pages 161–168, London, ON, May 2004.

[8] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. *cgf*, 30(7):1921–1930, September 2011. Proceedings of Pacific Graphics.

[9] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, pages 203–213, Washington, D.C., April 2005.

[10] Carsten Dachsbacher, Marc Stamminger, G. Drettakis, and F. Durand. Implicit visibility and antiradiance for interactive global illumination. volume 26, 2007. SIGGRAPH Procedings.

[11] Carsten Dachsbacher, Jaroslav Kr̂ivanek, Miloŝ Haŝan, Adam Arbree, Bruce Walter, and Jan Novak. Scalable realistic rendering with many-light methods. volume 33, 2014.

[12] Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik P. A. Lensch. Edge-avoiding À-trous wavelet transform for fast global illumination filtering. *Proceedings of the Conference on High Performance Graphics*, 2010.

[13] Zhao Dong, Thorsten Grosch, Tobias Ritschel, Jan Kautz, and Hans-Peter Seidel. Real-time indirect illumination with clustered visibility. In M. Magnor, B. Rosenhahn, and H. Theisel, editors, *Vision, Modeling & Visualization*, pages 211–218. Eurographics Association, November 2009.

[14] Randima Fernando. Percentage-closer soft shadows. In *Proceedings of SIGGRAPH*, Sketches, pages #35:1–1, Los Angeles, CA, July 2005.

[15] Yongxiao Fu and Yonghua Chen. Haptic 3D-mesh painting based on dynamic subdivision. *Computer-Aided Design and Applications*, 5:131–141, 2008.

[16] Epic Games. Unreal development kit. Technical report, Epic, 2014. URL `https://www.unrealengine.com/what-is-unreal-engine-4`.

[17] Reid Gershbein and Pat Hanrahan. A fast relighting engine for interactive cinematic lighting design. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 353–358, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[18] Henri Gouraud. *Computer Display of Curved Surfaces*. PhD thesis, 1971.

[19] Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18:32–43, March 1998.

[20] Toshiya Hachisuka. High-quality global illumination rendering using rasterization. In Matt Pharr, editor, *GPU Gems 2*, chapter 38, pages 615–633. Addison-Wesley, 2005.

[21] Pat Hanrahan and Paul Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. *SIGGRAPH Computer Graphics*, 24(4):215–223, 1990.

[22] Milos Hasan, Fabio Pellacini, and Kavita Bala. Direct-to-indirect transfer for cinematic relighting. *ACM Trans. Graph.*, 25(3):1089–1097, 2006.

[23] Jan Hermes, Niklas Henrich, Thorsten Grosch, and Stefan Müller. Global illumination using parallel global ray-bundles. In *Proceedings of Vision, Modeling, and Visualization*, pages 65–72, Siegen, Germany, November 2010.

[24] Takeo Igarashi and Dennis Cosgrove. Adaptive unwrapping for interactive texture painting. In *I3D '01: Proceedings of the 2001 Symposium on Interactive 3D graphics*, pages 209–216, 2001.

[25] Henrik Wann Jensen. Global illumination using photon maps. In Xavier Pueyo and Peter Schröder, editors, *Rendering Techniques '96*, pages 21–30. Springer, 1996.

[26] Henrik Wann Jensen and Niels Jørgen Christensen. Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *cag*, 19(2):215–224, March–April 1995.

[27] James T. Kajiya. The rendering equation. *ACM Trans. Graph. (Proceedings ACM SIGGRAPH '86)*, 20(4):143–150, 1986.

[28] Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[29] Anders W. Kristensen, Tomas Akenine-Möller, and Henrik W. Jensen. Precomputed local radiance transfer for real-time lighting design. *ACM Transactions on Graphics*, 24(3): 1208–1215, July 2005. Proceedings of SIGGRAPH.

[30] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. Octree textures on the GPU. In Matt Pharr, editor, *GPU Gems 2*, chapter 37, pages 595–613. Addison-Wesley, 2005.

[31] Jaakko Lehtinen, Matthias Zwicker, Emmanuel Turquin, Janne Kontkanen, Frédo Durand, François X. Sillion, and Timo Aila. A meshless hierarchical representation for light transport. *ACM Transactions on Graphics*, 27:#37:1–9, August 2008. Proceedings of SIGGRAPH.

[32] Philipp Lensing and Wolfgang Broll. Efficient shading of indirect illumination applying reflective shadow maps. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 95–102, New York, NY, USA, 2013. ACM.

[33] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérôme Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics*, 21(3):362–371, 2002.

[34] Ligang Liu, Lei Zhang, Yin Xu, Craig Gotsman, and Steven J. Gortler. A local/global approach to mesh parameterization. *Computer Graphics Forum*, 27(5):1495–1504, 2008.

[35] Kok-Lim Low. Simulated 3D painting. Technical Report TR01-022, Department of Computer Science, University of North Carolina at Chapel Hill, June 2001.

[36] Christian Luksch, Robert F. Tobler, Ralf Habel, Michael Schwärzler, and Michael Wimmer. Fast light-map computation with virtual polygon lights. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, pages 87–94, Orlando, FL, March 2013.

[37] Morgan McGuire and David Luebke. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of High Performance Graphics*, pages 77–89, New Orleans, LA, August 2009.

[38] Morgan McGuire and Michael Mara. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, December 2014. ISSN 2331-7418. URL `http://jcgt.org/published/0003/04/04/`.

[39] Quirin Meyer, C. Eisenacher, Marc Stamminger, and Carsten Dachsbacher. Data-parallel hierarchical link creation for radiosity. *In Proc. EGPGV*, pages 65–70, 2009.

[40] Fabio Pellacini, Kiril Vidimce, Aaron Lefohn, Alex Mohr, Mark Leone, and John Warren. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Trans. Graph.*, 24(3):464–470, 2005.

[41] Bui Tuong Phong. Illumination for computer generated pictures. In *Graphics and Image Processing*, volume 18, pages 311–317, June 1975.

[42] Roman Prutkin, Anton Kaplanyan, and Carsten Dachsbacher. Reflective shadow map clustering for real-time global illumination. In *Eurographics 2012 - Short Papers Proceedings*, pages 9–12, 2012.

[43] Tobias Ritschel, Mario Botsch, and Stefan Müller. Multiresolution GPU mesh painting. In *Eurographics 2006 Short Papers*, pages 17–20, September 2006.

[44] Tobias Ritschel, Thorsten Grosch, Jan Kautz, and Hans-Peter Seidel. Interactive global illumination based on coherent surface shadow maps. In *Proceedings of Graphics Interface*, pages 185–192, Windsor, ON, May 2008.

[45] Tobias Ritschel, Thorsten Grosch, Min H. Kim, Hans-Peter Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph.*, 27:129:1–129:8, December 2008.

[46] Tobias Ritschel, Thomas Engelhardt, Thorsten Grosch, Hans-Peter Seidel, Jan Kautz, and Carsten Dachsbacher. Micro-rendering for scalable, parallel final gathering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia 2009)*, 28(5), 2009.

[47] Randolf Schärfig and Kai Hormann. Hardware accelerated 3D mesh painting. In R. Koch, A. Kolb, and C. Rezk-Salama, editors, *Vision, Modeling & Visualization*, pages 211–218. Eurographics Association, November 2010.

[48] Randolf Schärfig, Marc Stamminger, and Kai Hormann. Creating light atlases with multi-bounce indirect illumination. *Comput. Graph.*, 55:97–107, April 2016.

[49] Mark Seagal and Kurt Akeley. The opengl® graphics system: A specification. Technical report, The Khronos Group Inc, 9450 SW Gemini Drive, 45043 Beaverton, OR 97008-6018 USA, March 2014. URL `https://www.opengl.org/registry/doc/glspec44.core.pdf`.

[50] Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer, 1996.

[51] François X. Sillion and Claude Puech. *Radiosity and Global Illumination*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann Publishers, 1994.

[52] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics*, 21:527–536, July 2002. Proceedings of SIGGRAPH.

[53] Peter-Pike Sloan, Ben Luna, and John Snyder. Local, deformable precomputed radiance transfer. *ACM Trans. Graph.*, 24(3):1216–1224, 2005.

[54] Wolfgang Straßer. *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten*. PhD thesis, 1974.

[55] László Szécsi, László Szirmay-Kalos, and Mateu Sbert. Light animation with precomputed light paths on the GPU. In *Proceedings of Graphics Interface*, pages 187–194, Quebec City, QC, June 2006.

[56] Tamás Umenhoffer, Gustavo Patow, and László Szirmay-kalos. Caustic triangles on the gpu. In *Proceedings of Computer Graphics International*, pages 222–227, Istanbul, Turkey, June 2008.

[57] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Lightcuts: A scalable approach to illumination. *ACM Trans. Graph.*, 24(3):1098–1107, 2005.

Index