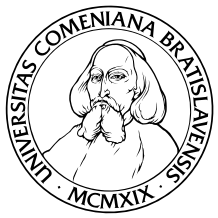
DEPARTMENT OF ALGEBRA, GEOMETRY AND DIDACTICS OF MATHEMATICS
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
COMENIUS UNIVERSITY, BRATISLAVA

REAL-TIME RENDERING OF PARAMETRIC SKIN MODEL

Bratislava, 2013

Stanislav Fecko



DEPARTMENT OF ALGEBRA, GEOMETRY AND DIDACTICS OF MATHEMATICS
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
COMENIUS UNIVERSITY, BRATISLAVA

REAL-TIME RENDERING OF PARAMETRIC SKIN MODEL

(Master thesis)

STANISLAV FECKO

Supervisor: RNDr. Martin Madaras

Study field: 1113 Mathematics

Study program: Computer graphics and geometry

Code: 41095156-3d98-4da7-9345-6fad6610d71f

Bratislava, 2013

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Stanislav Fecko
Študijný program: počítačová grafika a geometria (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.1.1. matematika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Real-time Rendering of Parametric Skin Model

Cieľ:
1) Naštudovanie kľúčovej literatúry k danej téme
2) Navrhnutie parametrického modelu pokožky
3) Implementácia modulu na renderovanie objektov s aplikovaním daného modelu pokožky

Literatúra:
- d'Eon, E., Luebke, D., and Enderton, E. 2007. A system for efficient rendering of human skin
- GPU Gems 3 by Hubert Nguyen

Vedúci: Mgr. Martin Madaras
Katedra: FMFI.KAI - Katedra aplikovanej informatiky

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 07.10.2010

Dátum schválenia:

prof. RNDr. Július Korbaš, CSc.
garant študijného programu

student

Vedúci

I hereby declare that I have written this thesis on my own, with the help of my supervisor, referenced literature and support from people mentioned in Acknowledgement.

Bratislava, May 6, 2013 .....

Acknowledgement

I would like to thank my supervisor RNDr. Martin Madaras for his guidance, useful suggestions and a lot of patience. Many thanks also go to doc. RNDr. Andrej Ferko, PhD, for support in the course of the work. I would also like to thank Luis Manuel Morillo for providing us the Head of Davy Jones model.

My entire family has, as ever, been the pillar on which i could always rely, for what i would like to express my sincere gratitude. Last but not least i must mention my friends, Mgr. Veronika Dullová, Mgr. Michal Ferko, Mgr. Ivana Uhlíková and Mgr. Matúš Ždanský, who pushed me in the right direction countless times, kept giving me the most valuable moral support and without whom this thesis would never see the light of day.

Abstrakt

Cieľom tejto práce bolo implementovať renderovací engine špecializovaný na vykresľovanie rôznych typov pokožky v reálnom čase. Opisujeme postupy a techniky ktoré sme použili.

Na začiatku sa venujeme základnejším pojmom, aby sme čitateľa uviedli do kontextu práce. Vysvetlíme pojmy ako bump-mapping a deferred shading. Potom sa zameriame na niektoré techniky, ktoré využívame na realistické zobrazovanie kože. Popíšeme postupy na real-time počítanie efektov ako Sub-surface Scattering, Ambient Occlusion a zrkadlových odrazov. Jednu po druhej metódy rozoberieme, spomenieme výhody i nevýhody daných prístupov a vyjadríme sa k niektorým zaujímavým detailom a vylepšeniam, ktoré by mohol čitateľ využiť.

No a keďže našou snahou je zobrazovať čo najširšiu paletu rôznych pokožiek, predstavíme možnosti jednoduchej a efektívnej manipulácie so vzhľadom kože v závislosti od niekoľkých nastaviteľných parametrov.

Kľúčové slová: Real-time rendering, Skin Rendering, Subsurface Scattering, Texture-space Diffusion, Deferred Shading, Screen-space Ambient Occlusion, Environment Mapping

Abstract

In this thesis we describe concepts and methods we used to implement a real-time renderer of humanoid skin surfaces.

In the first part we explain basic ideas of techniques such as bump-mapping or deferred shading. Then we focus on particular effects our rendering system uses to simulate realistic skin appearance. We describe methods for rendering sub-surface scattering, ambient occlusion and mirror reflections. Focusing on one at a time, we explain pros and cons of the approaches as well as notable details that might interest the reader.

Since the goal of this work is to render highly variable skin types, we also introduce certain ways to easily and efficiently modify the skin by a set of input parameters.

Keywords: Real-time rendering, Skin Rendering, Subsurface Scattering, Texture-space Diffusion, Deferred Shading, Screen-space Ambient Occlusion, Environment Mapping

Contents

List of Figures

1 Introduction

The goal of this thesis is to create a rendering system capable of realistic real-time rendering of various skin materials, including different non-human versions. Our engine uses the concept of the deferred shading, supports high dynamic range rendering and takes advantage of the methods like texture-space light diffusion, screen-space ambient occlusion or environment mapping technique. We describe the methods used in our pipeline as well as both their advantages and disadvantages. All the mentioned techniques are illustrated in the resulting images these allowed us to achieve.

First we explain the situation, why we need special rendering methods for skin and how we plan to achieve the results we want. Afterwards we explain some more basic concepts necessary to understand the following chapters. The reader who is familiar with these topics is encouraged to skip the section and focus their attention on more interesting content in other chapters. Then we take a closer look on the most notable techniques we have used and describe how they work. Finally, we present our results and provide some performance rating comparisons of different setting used while rendering.

When a ray of light enters a translucent material, it starts to move through an optically non-uniform environment, resulting in scattering of the light under the surface of the object. After traversing the material, the light eventually leaves the object. This happens not at the point where it entered though, but in slightly different. Visual effect called the sub-surface scattering is often visible on materials like marble or wax. Human skin is formed of multiple layers of different skin tissues. That is why effects, usually seen on other translucent materials also occur on the skin.

In section **??** the reader can find out more about the Deferred Shading concept, which we have been using in our program. It is widely used in various real-time applications to avoid a loss of performance on shading hidden pixels. The approach also allows the using of many local lights and directly supports numerous image-based post-processing effects as well as HDR rendering. The idea of the deferred shading is rather old already, first time presented (although not yet using the 'deferred shading' name) by Michael Deering [3]. The method's real-time usefulness has, hovewer, started only a couple years back, when modern graphics hardware became capable of

rendering into multiple arbitrary textures. These days the concept is utilized in virtually every real-time rendering system as, for example, computer games. The implementation in the Killzone 2 computer game, as well as many interesting details are described in [13]. The concept and its limitations are explained in a presentation by Hargreaves and Harris [12]. Koonce in his chapter in GPU Gems [9] describes different extensions and effects available when using deferred shading.

Chapter 2 describes the concept of sub-surface scattering on translucent objects. There are different ways these objects are being rendered. An offline method described in an article by Donner and Jensen [5] is based on the photon mapping technique, which takes about 15 minutes (on an Intel Core 2 Duo 2.4GHz) to generate the resulting image, but it is capable of producing such advanced effects as volumetric caustics, translucent inter-scattering between surfaces and volumetric shadows. A rapid hierarchical integration of irradiance computed at selected points on the surface is used in [7]. Depending on the model, it can deliver results in under one minute. The technique presented by D'Eon et al. [4] uses texture-space diffusion to compute the scattered lighting in real time. Shah, Konttinen, and Pattanaik [11] use dual light-camera space technique, computing the area integral via a splatting approach in image-space. Also GPU Gems have a chapter on rendering of this phenomenon [6]. The fast methods do not produce result of the same quality as the previously mentioned techniques, but their huge advantage is the speed making the methods suitable for real-time rendering applications.

Chapter 3 is about ambient occlusion, and its estimation in real-time. The text explains what ambient occlusion is, how we determine its amount in various applications. Then we focus on a fast real-time technique, also known as Screen-Space Ambient Occlusion (SSAO) that is being utilized in our rendering system. Mittring describes it in his article [10] about developing the computer game Crysis. We mention both advantages and shortcomings of this approach as well as some possible enhancements. Other screen-space implementations are described in an article by Bavoil and Sainz [1], where the authors face the precision issues of existing real-time ambient occlusion methods. Bavoil, Sainz and Dimitrov [2] introduce another similar technique for the estimation of the ambient occlusion based on image-space information.

A fast and easy method for estimation mirror reflections is described in Chapter 4. The

technique produces fake reflections and has various misperfections, but it is awesomely cheap in terms of computational time, while its quality is still rather satisfactory. That is why we incorporated the method in our system to enhance the resulting renders.

In Chapter 5, we present the parametric skin model we have been using. Each of them is described, its impact on the skin appearance explained and also illustrated on pictures.

And finally, Chapter 6 shows the results of our rendering system. You can find there comparison of quality of images and tables describing dependency of the framerate on particular settings.

1.1 About the thesis

Our aim in this thesis has been to extend the existing concepts also for the non-human skins. The texture-space diffusion technique gives the nice results for human skin, because it gives it the look of a translucent material. However, there are situations when we need to render also the non-human characters like aliens, undead, dragons and others, with highly realistic features. This is where we may need to extend the method to fit the particular type of skin.

The model used for the testing of our rendering techniques is the head of Davy Jones, a character from a well known movie Pirates of the Caribbean. The mesh and the texture were created and are the property of Luis Manuel Morillo[1]. This character model suits our needs because of its rather specific skin, which is pretty much like the skin of an octopus - therefore wet, slimy and oily. Since human skin rarely has such attributes, we had to slightly extend the method for human skin rendering to achieve the satisfactory results with our Davy Jones.

Our implementation of the skin-rendering pipeline is done in C++ and OpenGL. Because of the features we use to achieve our results we require OpenGL 3+ capable hardware. We are making use of the framebuffer objects, which allow us to easily take advantage of the deferred shading concept in all stages that require the lighting, HDR rendering and high-precision image manipulation, as well as the screen space ambient occlusion effect.

[1] http://luima.com

1.2 OpenGL

Our rendering system uses OpenGL for performing the graphical calculations. OpenGL, which is an abbreviation of Open Graphics Library, is a standardized, platform independent interface between application and underlying graphical hardware. Its main benefit is that it allows the programmer to take advantage of the great power of massively parallel processing units on the graphics card. First time introduced in 1992, the OpenGL standard has been following the steep progress in the area of electronic engineering, delivering additional functionality based on recent hardware's new capabilities.

OpenGL basically allows the programmer to render simple geometric objects like points, lines and polygons. To be displayed on the screen, these undertake a set of steps including transformations, lighting calculation, shadowing and many others.

1.3 Shaders

To allow for even higher flexibility of the system, OpenGL allows its users to define small programs which are then executed for each polygon/vertex/pixel. They are called shaders and we distinguish several types. Depending on what objects the shader manipulates with, we call them geometry shaders, vertex shaders and pixel shaders. The first mentioned receive as their input a single polygon (in vast majority of cases triangles are used) and their task is to, if need be, subdivide the polygon by creating additional vertices. Vertex shader takes in one vertex and performs on it the desired calculation. Usually this is the part where all the geometric transformations (camera projection, object translation) take place. The output of a vertex shader is a points transformed from world-space into screen-space. After this phase the polygon is rasterized, therefore broken into individual pixels that are covered by the transformed polygon. Now, pixel shaders (often also called fragment shaders) take it over. Their task is to determine the color that is to be assigned to the given screen pixel. Here most light and color based effects, like shading, texturing or shadowing, take place.

If approached properly, shaders give us outstandingly wide range of possibilities. We can do basic effects as well as perform complex and advanced routines that squeeze the very maximum

even from the most powerful graphics cards.

We will be using shaders in every single step of our rendering pipeline. Since the nature of the tasks we need to perform the effects we need does not require the usage of geometry shaders, we will not be working with these. Also vertex shaders used in our rendering system are going to be rather simple, usually performing only some basic camera projection. All the magic presented here will be done in pixel shaders.

1.4 Bump mapping

Bump mapping is a technique for adding high-frequency details to rendered surfaces. It is a process of altering normal vector in each pixel of the surface. We do not change geometry, only give the flat objects bumpy look. As an input we use a bump map, which is a grayscale texture describing how 'high above geometry' the apparent surface should be. This texture is then mapped onto triangles in the scene and based on local difference of the bump value the deviation of the normal vector is calculated. If the difference is zero, the pixel uses unchanged normal, therefore the one determined by geometry alone. The steeper the bump gradient is, the more the altered normal differs from the original (geometric) one.

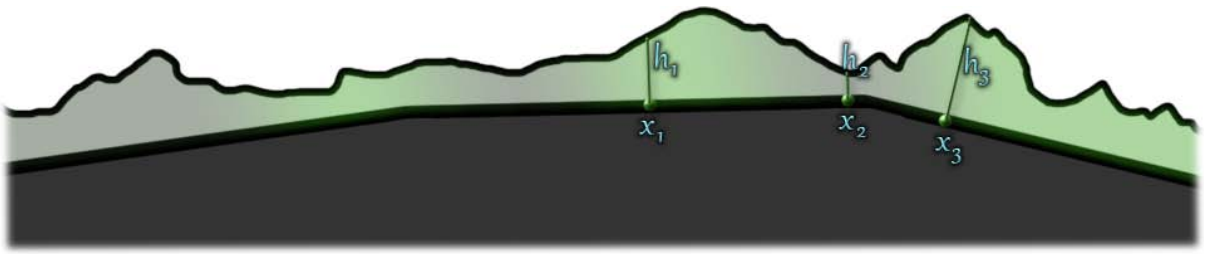


Figure 1: Concept of bump mapping. The dark area is the actual geometry of the rendered object. The light green area represents the volume "added" by the bump map. For each surface point x_i we consider the offset distance from polygon to seeming surface h_i.

Great advantage of this method is that it often enables us to use meshes with far less polygons to achieve results that are visually hardly distinguishable. The less polygons, the lower resources

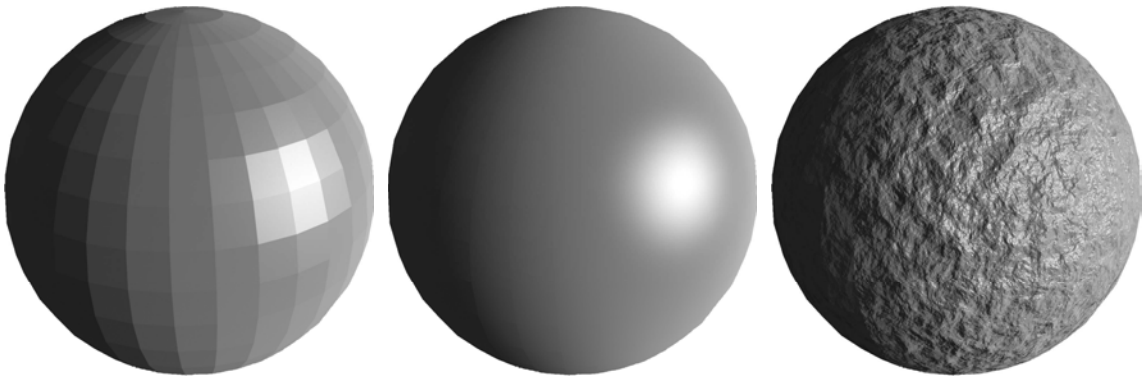


Figure 2: Bump mapping example. Each sphere consists of the very same triangles than the others. Left image shows flat shading (normal vector is equal for all points of given triangle), middle one uses smooth shading (bilinear interpolation used to determine the normal) and the right image uses the bump-mapping effect to add details.

it consumes and the higher the framerate, of course. Bump mapping is an effect that significantly increases visual quality of rendered objects while the computational its cost is rather low.

Figure 3: Another bump mapping example. From left to right: Flat shading, smooth shading, smooth shading with bump-mapping and finally also with albedo color. Note that the visual detail of the character is much higher in the image with bump-mapping, despite the fact the number of polygons used is the same. Used image is from computer game The Witcher.

1.5 Deferred shading

Deferred shading is a concept first described by Deering [3] in 1988. At first the method was not used for real-time rendering due to various hardware issues, but today it is widely used for its both easy implementation and great speed-increasing ability.

The main idea is to separate a geometry pass from shading. In the first phase geometry is being processed (geometry and vertex shading) and the results are written into textures. Only after the first part is done, we proceed to shading. In this pass data is being read from textures and processed in pixel shaders. Not only that the approach reduces number of shaded pixels (for complex scenes where many objects map to the same screen pixel and when we use complicated shaders, this reduction can be vital), but also makes it possible for HDR rendering and performing rather advanced post-processing effects

Default OpenGL pipeline outputs its results into the backbuffer, which is a memory segment used for collecting the pixel values of the final image. However, newer versions give the programmer the opportunity to change this default behavior of the pipeline and exchange the backbuffer with a special target called framebuffer object. This feature allows us to redirect the output of our rendering into textures. The same textures that we can later read from as from any other textures. Moreover, one can render into several (depends on system, but usually 8) textures at a time, meaning we rarely need more than one geometry pass. This functionality has a wide range of applications, amongst others also the HDR rendering, the deferred shading or the texture-space diffusion effect.

The framebuffer consists of a set of textures. Each texture can be written to (there are exceptions, as always, but usually it is possible) and read from. In first geometric, pass we render entire scene. Primitives and their vertices are transformed, attributes interpolated. Afterwards we write these interpolated values into separate textures. Design of the framebuffer strongly depends on particular task we seek to deal with, but usually one contains a texture for albedo color, texture for normal vector, texture for depth value, textures for screen-space normal, speed, reflectivity, specularity... Once these textures are filled with proper per-pixel data, we proceed to the shading pass.

Now all we do is render a few quads over the whole screen. Usually one quad for each effect.

All the computation is now done in pixel shaders, which read data from textures and evaluate them. It is good to understand that when using deferred shading, pixel shaders do not have only access to information about the very pixel they are operation on, but since all the data is in textures, a shader can read from any texel of any texture it needs. As a result, many techniques that determine pixel's value based on its local neighborhood are available to us, as well as some global methods, which take into account the whole image.

1.5.1 Killzone example

A very nice example of deferred shading is described in Michal Valient's presentation [**?**] on developing computer game KillZone 2. Figure 4 shows the content of different layers of the framebuffer and the final rendering result.

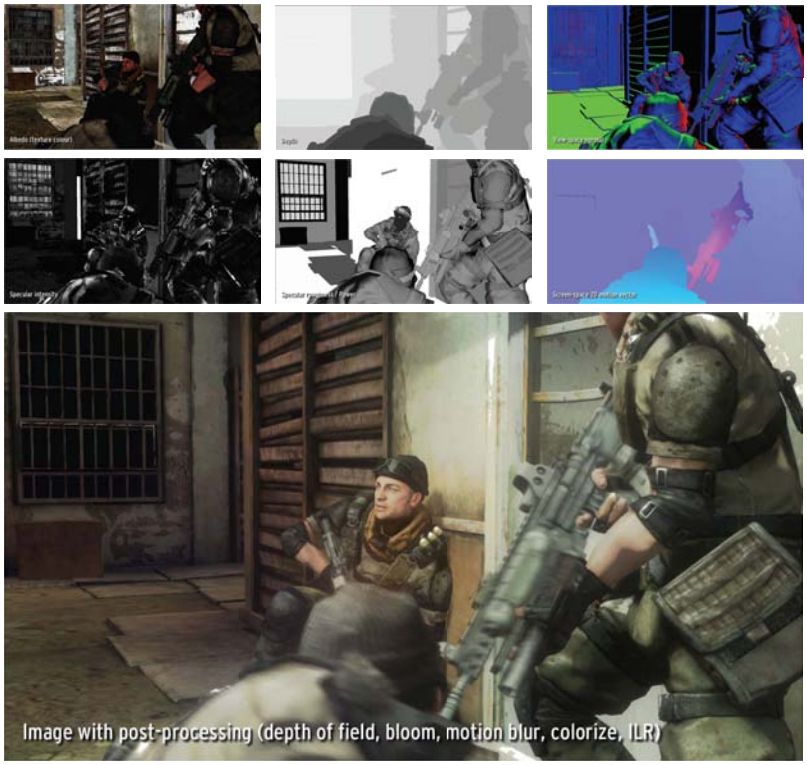


Figure 4: Deferred shading in Killzone. Images in the top two rows show the accumulated framebuffer data. From left to right and top to bottom: albedo color, scene depth (z-buffer), normal, specular intensity, specular roughness and screen-space speed.
Finally, the single bottom image is the final composition of data stored in framebuffer textures. Shading and various post-processing effects have been applied.

2 Sub-surface scattering

2.1 Concept

Sub-surface scattering (SSS) is a natural phenomenon observed on translucent materials. Light entering the volume of the object interacts with the material, scatters, and exits the volume at a different point and a different direction. This causes even areas that are not directly lit, to receive some amount of lighting. The effect is most notable in the scenes where an translucent object is lit by a back light. Despite the fact that a perfectly opaque object would be black from our point of view, the translucent object will have a bright silhouette due to the light that traversed through the material and left on the other side. The effect is very well visible on materials like wax, milk, marble or ice.

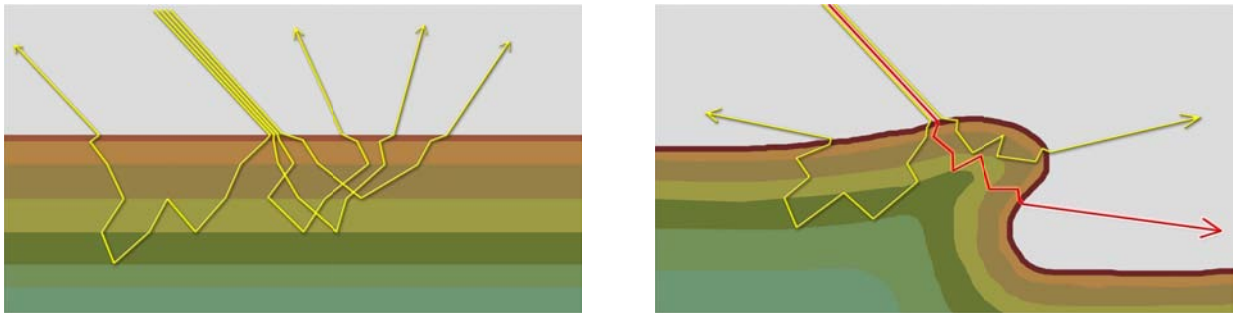


Figure 5: The concept of the sub-surface scattering. The ray entering the object scatters along its way in the material and finally leaves in a different position and with a different direction. Due to the sub-surface scattering effect also the spots that are not lit by the light source directly receive some light through the material (the red arrow in the right picture).

A human skin is a much more complex material than it may look. It consists of many thin layers with different optical properties. This is why the light penetrating the skin is scattered under the surface, resulting in the very same effect described earlier. Skin has no color of its own. Even the typical reddish color we usually observe when looking at our hands is caused by the skin's translucency and the red color of the blood beneath it. That is also why we can see our bones through the skin provided we have bright enough source behind it.

Figure 6: Sub-surface scattering on translucent materials. The first image is from [5], the second is from [6].

In computer graphics we need to compute the scattered lighting in order to get realistic looking translucent materials, including skin. There are various ways to do it, ranging from slow and precise methods (ray-tracing, photon mapping) all the way to fast and far less accurate ones. Since our interest here is to create a real-time renderer, we will focus on the fast techniques, which trade precision for speed. In our engine we have been using, with some adjustments, a method described in [4]. This approach lets us produce images of plausible quality, while the cost of the rendering remains rather low.

2.2 Sub-surface scattering via texture-space diffusion

The method proposed in [4] approximates the effect of the sub-surface scattering using techniques affordable in real-time rendering. The base idea is to render the entire scene into a off-screen buffer, process this data and then, in the second pass, use it to render the final image.

First phase of the algorithm is rendering of the object into the texture space. Each 3-dimensional vertex (x,y,z) from the world space is projected to its texture coordinates (u,v), therefore into the

2-dimensional texture space. Ambient and diffuse phong shading is computed for each texel. The result of this pass is a shaded texture, or rather a lightmap, of the entire object. It contains light information for every point on the surface of the model, including the areas that are not in a viewing volume of the camera, nor the light volume of the light source. Important note is that only the ambient and the diffuse components of the Phong's lighting model are used in this light calculation.

$$\underbrace{\begin{bmatrix} \text{Position} & (x,y,z,w) \\ \text{Color} & (r,g,b,a) \\ \text{TexCoords} & (u,v) \\ \text{Normal} & (n_x,n_y,n_z) \\ \text{Tangent} & (t_x,t_y,t_z) \\ & \vdots \end{bmatrix}}_{\text{Vertex data}} \quad \mapsto \quad (u,v) \tag{1}$$

The second phase is where the the data is processed, as mentioned earlier. This is actually the part where the simulation of the sub-surface diffusion takes place. We blur the light intensity values captured on the surface of the object, making the bright spots to bleed into the darker areas. This way also points with little direct light can be lit, provided that they are close to some bright ones.

In reality, the metric for this 'closeness' is a three-dimensional distance of two points. Our approach uses a two-dimensional distance in the texture space. This difference leads to different results, but the fact that we can compute this value very quickly outscores the precision drawback.

What is important here is the fact that all this blurring is done in the texture space. Therefore the result is also, as in the first stage, a lightmap.

The third phase requires another geometric pass. Here we render the whole scene again, but instead of computing every pixel's brightness, we use the blurred lightmap. So the diffuse and the ambient components of the pixel's illumination are read from the lightmap, and the specular component is added separately.

We compute specular highlights separately due to the fact that the specular component of

Figure 7: Image from [4] describing the phases of the sub-surface scattering rendering process. From left to right: Albedo; Diffuse light component; Blurred versions with different kernel size used; specular highlights and the final composition.

the Phong's illumination model represents the light reflected from the surface. That is why this light does not interact with the deeper material structure and is not scattered, merely mirrored off the topmost layer. The fact that we treat this component apart from the other two results in a soft translucent look of the rendered objects while the fine details on their surface are visually preserved thanks to the specular highlights.

2.3 Our approach

2.3.1 First phase

In the first phase we are going to map triangles from the world space into the texture space. Therefore our output framebuffer ought to be resized to the dimensions of the texture. Due to the performance load of working with high resolutions (in our case the size was 4096x4096 texels), we decided to use the framebuffer sized up to 2500x2500 pixels. This reduces the quality only slightly while significantly improving the speed of the rendering. The maximal size used for rendering can be dynamically set without need to perform any pre-computation, therefore the initial limit of 2500 texels squared does not negatively affect the method's maximum possible

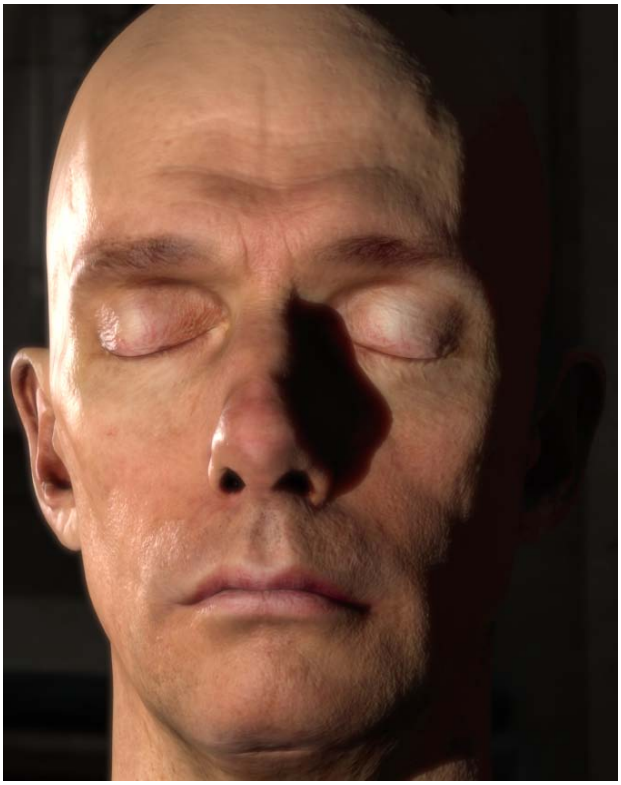


Figure 8: Image from [4] presenting a result image generated in real-time.

quality.

Since we are using the deferred shading approach, we do not render the intensity values into the buffer straight away. First we transform each vertex to its texture coordinates, while only its world-space position and normal vector are written into the buffer. Note that in this pass we do not need to capture per-pixel depth information (this saves time as well as memory requirements for the used framebuffer).

We are using the bump-mapping effect to enhance the surface of our model. We are performing this normal vector transformation in the first phase of the SSS algorithm. Our bump mapping calculation uses a greyscale texture as a heightmap in the tangent-space, from which the world-space normal is determined every frame. This way the method is open to future improvements that might change bump map dynamically.

The bump mapping process is controlled by a bumpiness parameter (see Figure 9) describing

Figure 9: Bumpiness parameter of the skin. The left image has the value 0, middle image 0.75 and the right one is rendered with the parameter set to 3.

how much the heightmap affects the final normal vector. The lowest valid parameter value of 0 corresponds to unchanged normal, therefore a smooth surface. The higher the value is, the more featured the bumps become. In all figures in this thesis, except for the Figure 9 and Figure 28, the value 1.0 was used.

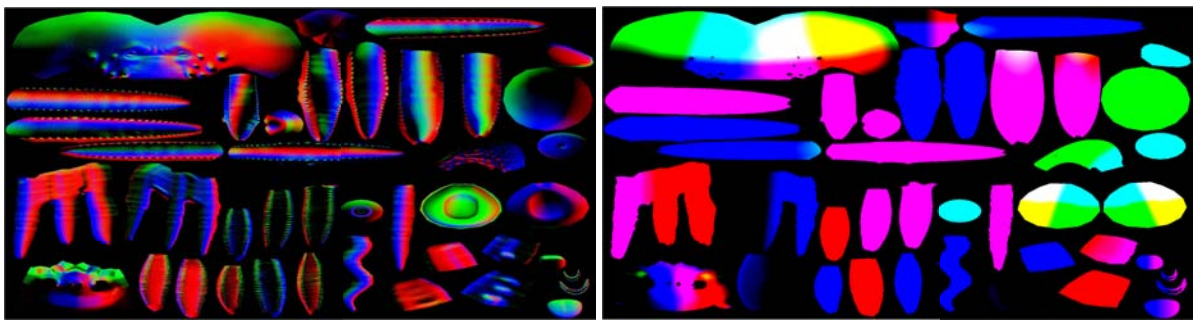


Figure 10: Content of the framebuffer textures. World-space normal is shown on the left and world-space position on the right image. Calculated lighting (ambient and diffuse) is left in Figure 11.

Only after the entire scene is rendered, we proceed to shading. For every light source one pass is required and the additive blending is used for merging the intermediate results of separate passes. The result of this pass is a texture containing the light intensity values for every texel of the original texture (see Figure 10).

2.3.2 Second phase

The method presented in the paper on which our effect bases uses a series of different gaussian blurs, which are afterwards blended together with some weights to get more physically correct model of diffusion. In our rendering engine, however, we use only one blurring sequence. This way the speed of the rendering is increased, while the accuracy of our results is only slightly lower.

This phase uses a separable gaussian blur to simulate the diffusion process. The amount of blurring depends on the parameters of the particular skin (for comparison see Figure 13). The more translucent look we desire, the wider kernel has to be used. Currently the skin is parameterized by one real number describing the radius of the blurring effect. After two shader passes (gaussian in u and v directions) we have the resulting blurry lightmap in one of the framebuffer's textures (see Figure 10).

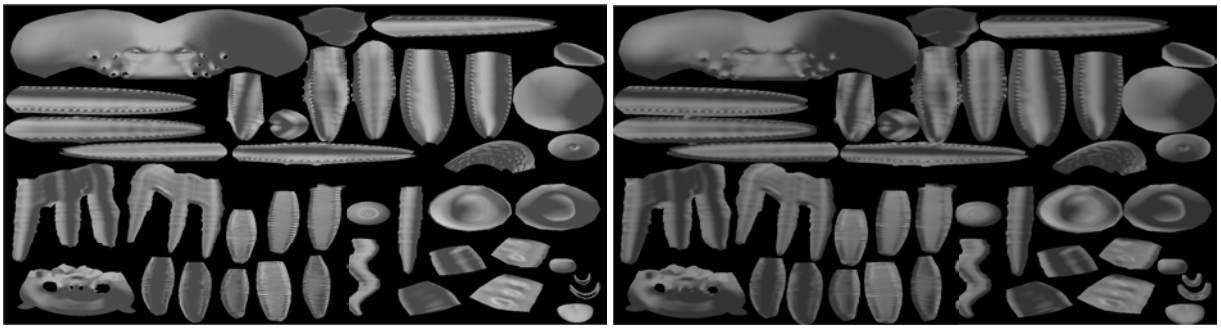


Figure 11: Result of the second phase of the method. Generated lightmap (left) is blurred by arbitrarily wide kernel. Note that the contours of the segments in the blurred texture (right) have inflated. This is best visible on holes in the segments, that have shrinked.

Here we had to face a problem with unwanted intensity leaking. When blurring the lightmap, the black area surrounding the non-black segments would leak into close-by texels. As a result, the borders between parts of the model that used different sectors of the texture became visible (Figure 12 on the left).

To solve this problem we added alpha testing into blurring passes. Each sample taken has

Figure 12: Problem with the diffusion process: visible leaks (left) of the background to the shaded areas. On the right image, the problem is solved.

its weight multiplied by the texel's alpha value. This value is computed for entire texture in real-time, without need for preprocessing as a side result of the first phase.

Before the first geometry pass, each texel's opacity is set to 0. Afterwards, as the geometry is being projected into the texture space, every written fragment overwrites current alpha value by 1. This way we do not need any additional pass to determine which texel is background (note we do not need constant background color) and which belongs to the object's surface. When the blurring takes place, all the background texels (in our case these are the black ones) are ignored. The lightmap blurred by the improved shader is in the Figure 11 on the right.

This texture can be, at the beginning of the third phase, bound to the texture unit and, the same way as any other texture, used as a source.

2.3.3 Third phase

Finally, in the last part of the algorithm we use the light intensity information contained in the processed texture to determine the ambient and the diffuse components of the pixel's illumination. In this phase we also use deferred shading model, therefore whole model is again rendered into separate data layers. Afterwards we proceed to shading, where only the specular component of the Phong's illumination model are evaluated and added, the ambient and the diffuse components are read from the blurred texture. After adding the specular highlights, the diffusion method is at an end and we proceed to other effects.

Figure 13: Amount of blurring and it's impact on the look of the surface. From left to right used kernel size: without blur, 7x7tx and 19x19tx. Note that specular highlights on all three images are identical.

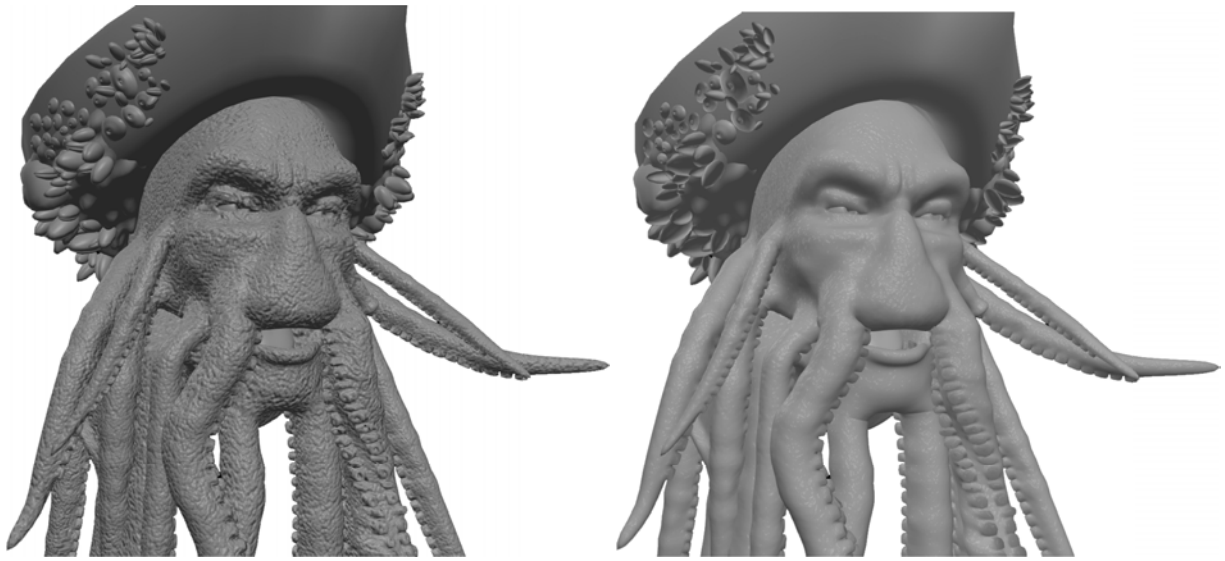


Figure 14: Final result of the sub-surface scattering effect. Only basic phong shading (left) is compared to scattered phong (right; blurring kernel 9x9tx), with no additional effects added.

3 Ambient occlusion

3.1 Concept

Ambient occlusion is a technique determining amount of ambient light at a given point of the scene. This occlusion is due to geometric obstacles in the point's neighborhood, blocking some fraction of the incoming ambient light from the surrounding environment. This term can range from 0 (total occlusion, no ambient light available, for example inside of a keyhole) to 1 (full ambient light, for example straight surface).

Proper ambient occlusion factor is view-independent, therefore it can be precomputed and stored with model, for example in separate texture. This approach allows for very high quality of resulting image, because we can afford to take very long time to generate the texture. On the other hand, if the model changes (animation, deformation, even moving within the scene), the occlusion factors change as well, and therefore the precomputed data cannot be used.

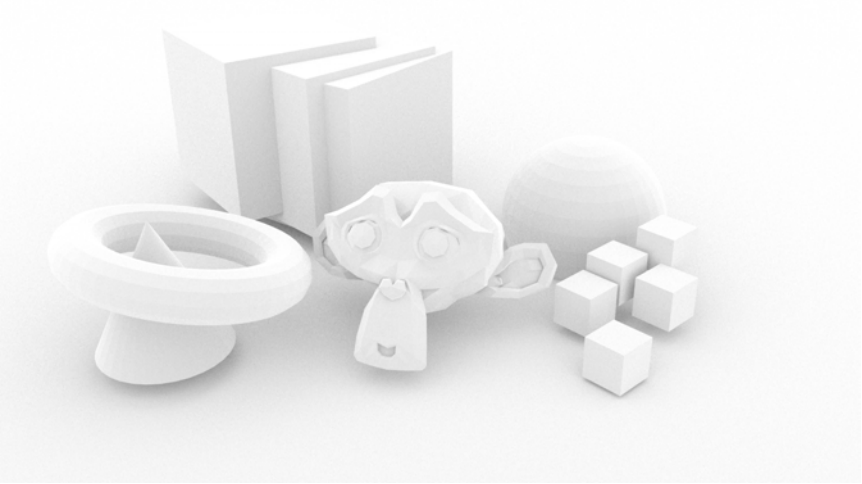


Figure 15: Raytraced ambient occlusion of a scene. The effect is nicely visible in areas where the incoming light is restrained due to surrounding geometry. Image rendered in Blender and the process took 78 seconds.

In many real-time applications, for example in computer games, a fusion of pre-rendered data with the real-time computed lighting is used. Baking ambient occlusion into a texture is very popular method of enhancing shading quality of real-time rendering. This approach is most valuable in cases when the object doesn't change much, as would be for example lighting of room interior. Even when used for shading of characters, the technique delivers notable improvement

of the output quality, despite the fact the characters tend to move in a scene a lot.

In our application, however, we want an approach that would be completely independent from any pre-procesing or pre-computation. Our aim is to create a renderer that could estimate the ambient occlusion in real time for any arbitrary scene. To achieve this, we are using Screen-Space Ambient Occlusion technique.

Figure 16: Ambient occlusion computed for our model of Davy Jones. Rendering in Blender took 71 seconds.

3.2 Screen-space ambient occlusion

Screen-space approach is a real-time method that trades accuracy for speed. Results produced by this technique are no longer view-independent, tend to be noisy and suffer from various aliasing-related artifacts. On the other hand, the method produces its results without any need for pre-computation and, on decent hardware, in matter of milliseconds. Ambient occlusion can now be produced in real-time, for any dynamic scene, regardless it's geometric complexity. With a few optimizations the results are produced both quickly and with rather sufficient quality.

The method consist of one full-screen shader pass where, for every pixel of the scene, defined number of samples are taken from pixel's neighborhood and based on these the occlusion factor is calculated. Approach uses depth values from z-buffer to determine relative positions of sampled point and the geometry of the scene.

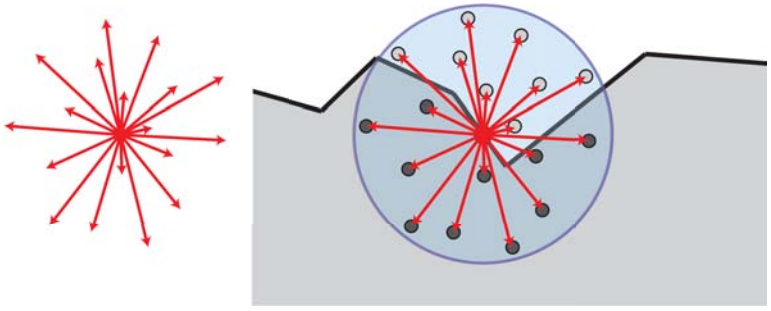


Figure 17: Left: static set of sample positions. Right: Positions applied on the examined pixel's surroundings.

3.2.1 Sampling

Figure 17 above shows the sampling process in two dimensions. Gray area with black borders is the geometry of the scene represented by depth values in z-buffer. The red dot is the point for which we are determining the occlusion factor. Now we take a number of samples from around the pixel (from inside the blue circle) and for every one of them we check whether it is or is not hidden behind geometry. Based on number of samples which fall behind geometry we determine amount of occlusion on the red dotted pixel.

The main idea of the method is sampling every screen pixel's neighborhood and by comparing the depth values of these samples we determine the amount of the occlusion the sampled

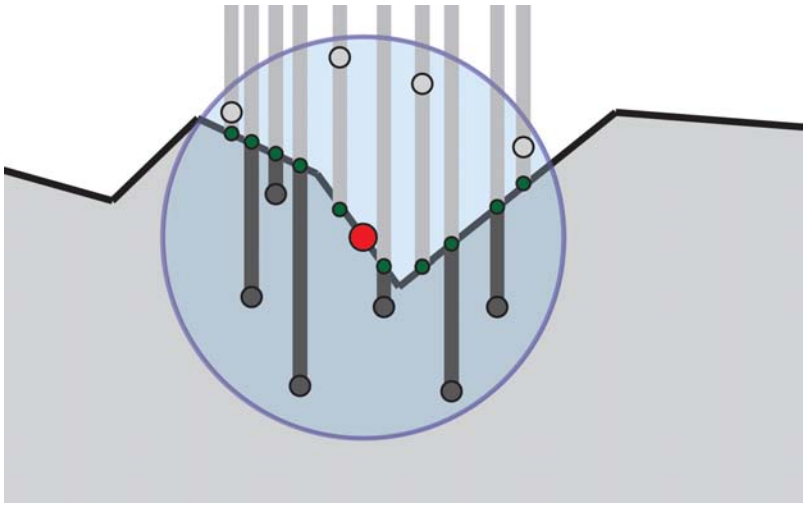


Figure 18: Sampling pixel's neighborhood and comparing the depths of the samples against the scene geometry. The dark circles contribute to the examined pixel's occlusion, while the bright circles decrease the amount of the occlusion. The green circles correspond to the values in the depth map.

points cause onto the examined pixel. Every sample that falls behind the geometry of the scene (its depth value is greater than the value in the depth map) contributes to the examined pixel's occlusion (pixel will be darker) and every sample closer than the depth map reduces the occlusion (brightness increases).

3.2.2 Hemisphere improvement

A technique taking advantage of the screen-space normals is often used to both speed up the ambient occlusion computation and make it more accurate. Provided that we know a normal vector in the examined pixel, we can choose to sample only the points inside the hemisphere in the normal's direction. This way we need merely half of the samples we would have to use otherwise, what means the algorithm provides results twice that fast. The normal used here has to be in screen-space, so we also project it by projection matrix in vertex shader (unlike we do in other cases, where we only use modelview matrix).

In addition to increased speed we actually get better results. Due to the fact that the normal vectors are projected into the screen-space after the normal/bump mapping takes place, the method generates the ambient occlusion also on flat surfaces enhanced by bump-mapping. Since the bump-mapping is computationally pretty cheap effect, this improvement of the ambient oc-

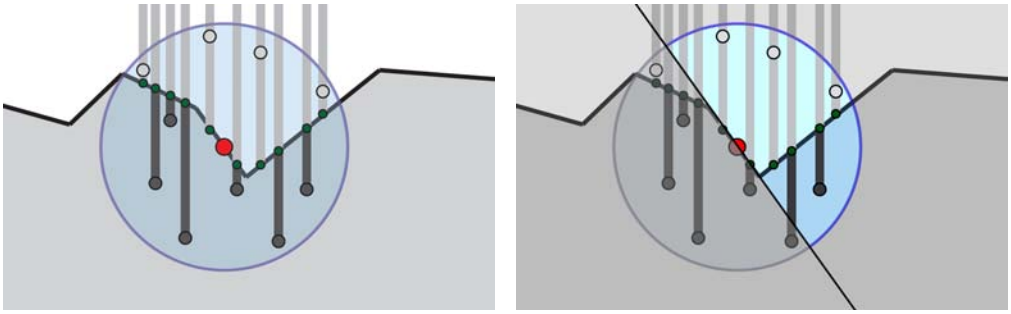


Figure 19: Sampling pattern in case that we know every pixel's normal. Only half of the samples that had to be compared without the improvement have to be sampled now, resulting in doubled speed of the method.

clusion estimation algorithm is particularly useful for applications requiring a high framerate. Also scenes containing rather low-polygon models are efficiently rendered with satisfactorily good results.

Figure 20: Screen-space ambient occlusion using only depth information (left) compared to the method using per-pixel screen-space normals (the other two). The right image is with bump-mapping on, the middle one is without the effect. In all cases the number of taken samples was the same.

3.2.3 Noise vs. Artifacts

It turns out that using the same sampling pattern for all pixels tends to cause various banding artifacts (Figure 21). The way around this problem is a tradeoff between banding and noise. We use a static set of sample positions, but to avoid artifacts, we rotate this set by different angle for each pixel. This way we trade artifacts and visible patterns over the image for a rather uniform noise. Afterwards we reduce the noise by blurring the occlusion map a little (gaussian with 3x3 kernel).

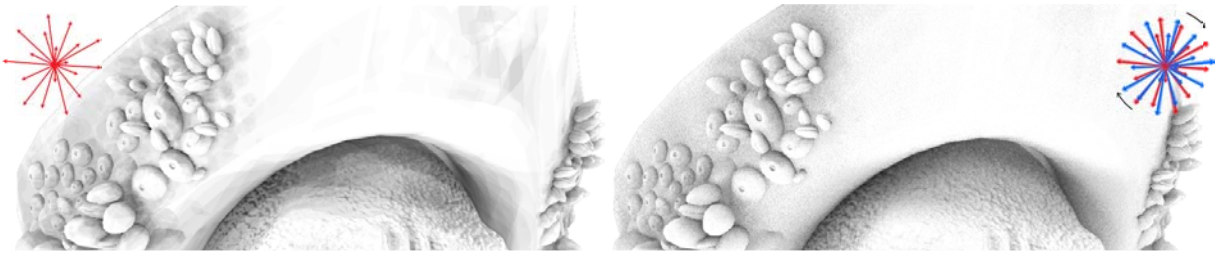


Figure 21: Trading artifacts (left) for noise (right).

3.2.4 Multiple scales of SSAO

To achieve the occlusion on both small details and larger areas, we use one sampling pattern for high-frequency features and a different one for low frequency. When combined together, these give a satisfactory quality of the occlusion, while the cost of the effect is affordable.

In our implementation of ambient occlusion we used different parameters for high, medium and low-frequency maps. Figure 22 shows the intermediate results. In the first case, 12 samples were taken from area of radius 12px. In the middle-frequency case, the number of samples was 16 and radius 60px. Finally in the low-frequency map we took 20 samples from area with radius 170px. After the sampling, all three textures were blurred by gaussian blur with 3x3 kernel size. The mixing formula we use for blending these maps is in Equation 2. We achieved visually best results with parameters $a = 1$, $b = 2.2$ and $c = 3.8$.

$$\text{SSAO} = \sqrt[a+b+c]{(\text{High})^a \cdot (\text{Medium})^b \cdot (\text{Low})^c} \tag{2}$$

Figure 22: From left to right: High, middle and low frequency ambient occlusion map. The last picture shows the same images, all in one.

To show how much the quality of the map depends on sampling settings, we provide Figure 24. In this image one can see how sampling radius affects the output (the higher radius, the lower occlusion frequency). Note the difference of noise intensity based on number of taken samples.

Figure 23: Final screen-space ambient occlusion map compared to raytraced image.

Figure 24: SSAO parameters comparison. First row shows high-frequency occlusion (radius 12px), second is middle (60px) and the bottom row displays low-frequency occlusion maps (radius 200px). From left to right samples count: 4,16 and 64. These images have not been blurred to show the difference in quality and noise intensity.

4 Environment mapping

4.1 Concept

To achieve wet or slimy look of the rendered surfaces, we added the effect called environment mapping. It approximates mirror reflections on the objects in the scene. The method requires an environment texture, which is usually a panoramatic photograph that describes the surroundings of the object or the entire scene. For each screen pixel, we read also its normal vector defining a tangent plane at the point. Then we reflect the viewer-to-pixel vector from the plane (reflection Equation 3, where $\vec{v_v}$ is pixel-to-viewer unit vector, $\vec{n}$ is surface normal and $\vec{v_r}$ is the reflected direction).

$$\vec{v_r} = 2\,(\vec{v_v} \cdot \vec{n})\,\vec{n} - \vec{v_v} \tag{3}$$

A ray defined by the reflected vector afterwards crosses the environment texture. This intersection point determines, which texel from the environment texture is to be used as the reflected color. The sampled color is afterwards then added to the current pixel's color. The method is in a combination with the deferred shading very fast, requires only one full-screen shader pass that reads only one filtered texel value for every screen pixel.

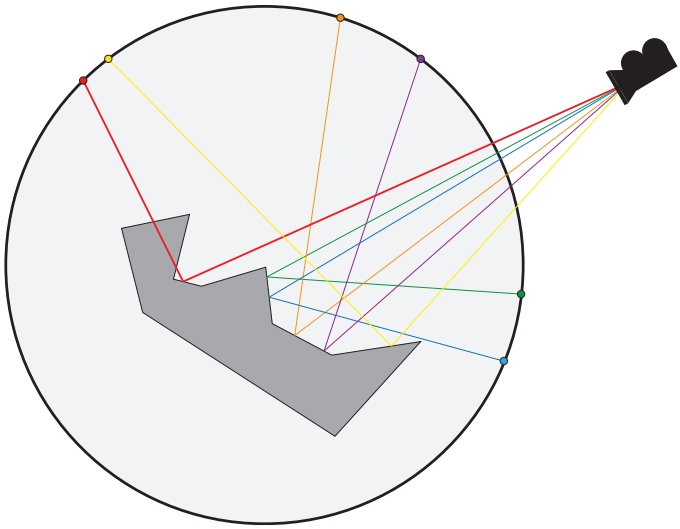


Figure 25: Concept of environment mapping. Rays from the camera bounce from the object and hit the background.

The mechanism of the effect is described in the Figure 25. The environment texture, which encircles the scene objects, is drawn here as a thick black circle. Each screen pixel is interpreted as a ray originating in camera and oriented towards the scene. If the ray hits an object (meaning the depth value in z-buffer is less than 1), it reflects from the object's surface. After that it hits the background at some point. From this point we sample the color from the environment texture. This color is multiplied by a reflectivity of the skin and added to the current pixel's value. We use the reflectivity parameter to control how much environment illuminance is reflected by the skin, therefore to determine how much (on scale 0 to 1) the surrounding scene affects the object's look. See Figure 29 to see examples how the reflectivity affects the final image.

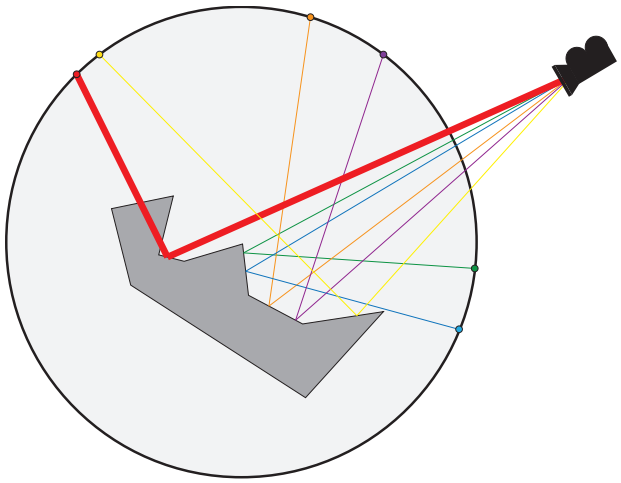


Figure 26: Environment mapping problem: multiple reflections. The red ray is not traced correctly by this method, resulting in non-realistic reflections.

Note the thick red ray, which after the reflection crosses the volume of the object on its way to the background. Properly handled ray would reflect a few more times before reaching the environment texture. However, the data we have does not allow us to trace the ray accurately, therefore these bounces are ignored. On the other hand, this approach makes the estimation of the reflection quite fast, making the method highly suitable for real-time rendering.

4.2 Self-reflections

Since this approach does not take self-reflections into consideration, it may produce rather unconvincing results when used on complex objects. And as our testing model could be considered somewhat complex due to the numerous tentacles, we had to deal with this problem. Our easy and fast solution uses the result of the SSAO procedure and modifies the amount of the reflection based on the amount of the occlusion. The idea behind the trick is that the more occluded the pixel is, the more reflections the ray has to undergo to reach it. Therefore, its energy is strongly reduced and that is why we decided to reduce its contribution. Of course, this approach doesn't produce accurate mirror reflections either, but the visual quality of the reflective object is more convincing.

Figure 27: Environment mapping enhanced by the SSAO term. Left is untreated, middle is with the improvement. Right image is the model raytraced (in Blender over less than 10 seconds).

5 Parametric skin model

As our aim is to render as wide range of different skins as possible, we provide a few parameters, that can be modified in order to change the skin's appearance. The parameters are bumpiness, reflectivity and the amount of sub-surface scattering present. All mentioned parameters can be changed in real-time, without any need for precomputation. Moreover, bumpiness and reflectivity do not even affect rendering time, as these are merely multipliers of intensity of particular effect. Translucency (here understood as the radius of the blur of lightmap during SSS computation) does affect the framerate, because for greater radius more texels have to be sampled. The higher the radius, the longer it takes to render one frame, and the framerate decrease should not be steeper than linear with the size of the blur.

5.1 Bumpiness

The bumpiness parameter controls how bump map affects the object's surface. The greater the value, the more featured the bumps are, as if stepping further in to the third dimension. For comparison of different values of this parameter, see the Figure 9 and Figure 28.

Figure 28: Comparison of the influence of the bumpiness parameter on the character's skin appearance. From left to right used values 0, 1 and 3. Images rendered with all effects on, except for albedo texture.

5.2 Reflectivity

Reflectivity parameter controls how much the environment affects the object's surface. It is used as a multiplier of reflected environment light intensity. The greater the value, the more intense reflections are added, therefore the overall brightness increases. For comparison of different values of this parameter, see the Figure 29.

Figure 29: Comparison of the influence of the reflectivity parameter on the character's skin appearance. From left to right at top to bottom used values 0.07, 0.18, 0.35 and 0.70.

5.3 Translucency

Translucency parameter controls how much the material scatters the light under its surface. The parameter directly means the radius of the blurring kernel used during the texture-space diffusion. The greater the value, the more the lightmap is blurred, therefore the diffuse lighting of the character will be softer. For comparison of different values of this parameter, see the Figure 30. Note, that there is much more visible difference between the lowest setting and the effect completely switched off, than the difference between small radius and huge radius. This is mainly due to high-frequency details on the surface (the very details bump-mapping added), which almost vanish even after small blurring.

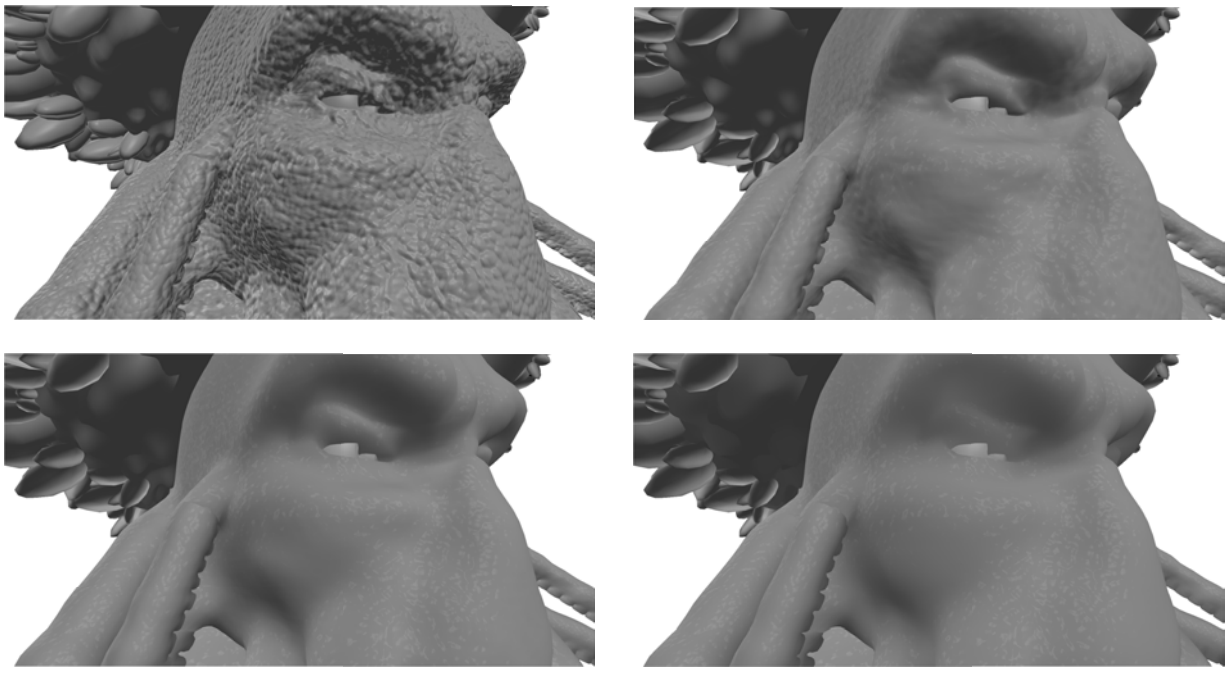


Figure 30: Comparison of the influence of the amount of lightmap blurring on the character's skin appearance. From left to right and top to bottom used values 0x0tx, 5x5tx, 13x13tx and 25x25tx.

6 Performance and results

6.1 Final rendering pipeline

In Equation 4 is described how the effects are combined into the final image composition. First, the line 1 takes place in the framebuffer (in texture-space; all the other lines are in the screen-space). During the second geometry pass, lines 2 and 3 are evaluated (in that order). Last two lines describe the application of the environment mapping and the final fusion of all the effects. Meaning of the High, Medium and Low, as well as a, b and c is explained in the Figure 22. Parameter Skin describes the properties of the particular type of skin. Ambient, Diffuse and Specular are the components of the widely used Phong's shading model. Values EnvMap and Albedo are color vectors read from textures.

$$\begin{aligned} \text{AmbDif} &= (\text{Ambient} \cdot \text{Skin}_{\text{amb}} + \text{Diffuse} \cdot \text{Skin}_{\text{dif}})_{\text{blur}} \\ \text{SSAO} &= \sqrt[a+b+c]{\text{High}^a \cdot \text{Medium}^b \cdot \text{Low}^c} \\ \text{SSS} &= \text{AmbDif} \cdot \text{SSAO} + \text{Specular} \cdot \text{Skin}_{\text{spec}} \\ \text{Reflection} &= \text{EnvMap} \cdot \text{Skin}_{\text{refl}} \cdot \text{SSAO} \\ \text{Final} &= \text{SSS} \cdot \text{Albedo} + \text{Reflection} \end{aligned} \tag{4}$$

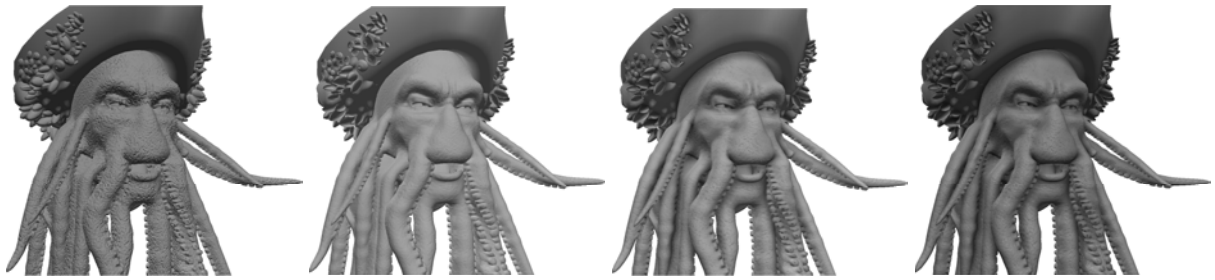


Figure 31: Sub-surface scattering process on our model. From left to right with additional effects: ambient and diffuse lighting; blurred lighting; with screen-space ambient occlusion; with specular highlights.

Figure 32: Composition of the effects. From left to right: Sub-surface scattering alone; SSS with environment mapping; SSS with screen-space ambient occlusion; SSS with both environment mapping and ambient occlusion.

6.2 Performance

Our skin rendering pipeline has been tested on a 64-bit Windows 7 Home Premium system with GeForce GTX560 graphics card, Intel Core i7-950 processor and 12GB of physical memory. When all effects are switched on and the application runs in 1920x1080 resolution with no multisampling, we achieve average pace of rendering above 20fps.

Top part of Table 1 presents the performance of the application based on three parameters: geometric complexity of the scene (number of triangles), blur size during SSS effect simulation (blur kernel size in texels) and the screen resolution (in pixels).

The bottom part of Table 1 shows how mesh complexity and screen resolution affect the time necessary for one geometric pass (in milliseconds) and SSAO time consumption (also in milliseconds). The SSS method consists of two geometric passes, but the table contains the measured time of the second one. This is simply because the latter took longer to run in most cases.

Since the pipeline consists of two geometric passes, we have to draw over half a million polygons per frame (with the more complex mesh), what also reflects on a lower framerate in scenes with the higher complexity of the geometry. As is also clear from both the table and the description of the used methods, time needed to render one frame depends on the used resolution. This is why cases with lower screen dimensions scored double the framerate of the scenes with

higher resolution. However, in all used scenarios we still reach real-time rates, therefore satisfying results. Even in the most complex scenes and the highest resolutions we rarely dropped below 20fps.

Mesh complexity	Screen resolution	SSS blur size	Framerate
65 K	640 x 480	5x5	60+
65 K	1280 x 720	5x5	42
65 K	1920 x 1080	5x5	31
65 K	640 x 480	17x17	43
65 K	1280 x 720	17x17	30
65 K	1920 x 1080	17x17	21
259 K	640 x 480	5x5	31
259 K	1280 x 720	5x5	30
259 K	1920 x 1080	5x5	31
259 K	640 x 480	17x17	30
259 K	1280 x 720	17x17	28
259 K	1920 x 1080	17x17	20

Mesh complexity	Screen resolution	Geometry pass	SSAO
65 K	640 x 480	2.5	2.0
65 K	1280 x 720	3.8	5.3
65 K	1920 x 1080	6.8	12.1
259 K	640 x 480	3.6	2.1
259 K	1280 x 720	5.4	5.6
259 K	1920 x 1080	9.0	13.2

Table 1: Performance of our solution.

6.3 Results

Finally, here we present the images our rendering system produced.

Figure 33: The final image output of our rendering system.

Figure 34: Another image.

Figure 35: And yet another image...

7 Conclusion

The result of our project is a C++ class dedicated to real-time skin rendering. It bases on the deferred shading concept utilizing the benefit of the HDR rendering. It implements the sub-surface scattering effect via texture-space diffusion technique, estimates the ambient occlusion using a fast, screen-space approach and simulates the surface reflections by the environment mapping method. On described testing system the engine delivers real-time results even for the complex model we are using.

There are also effects we have not implemented yet and features we would like to add to the project as a future work. Our plan is to create a procedural skin generator for an automatic production of various non-human skins. We are also searching for and experimenting with the implementation of other real-time methods that would improve the translucent and slimy look of the rendered character while retaining the high framerate.

The area we are now mainly focusing on is building a texture generator capable of producing different types of skin textures as well as bump or reflection maps. Our idea is to use various pre-made images and by blending these to create new ones. The prototype uses the framebuffer objects and OpenGL shaders to process the source images and output the final textures based on a given description. We aim to generate these skins in real-time. This way the application could be used for both designing and rendering of the characters.

Our system has certain flaws we haven't had enough time or expertise to remove. The way the texture is mapped onto the model separates the mesh into parts. Borders between these parts would become more and more featured with increased blur radius during the texture-space diffusion effect. As a future work, it would be pretty nice to solve this issue. We also encountered a problem with application of ambient occlusion. The best way to do it would be to apply the occlusion on the lightmap before blurring, not afterwards as it is currently done. This way the translucency of the skin would be better visible. The reason we haven't done this is that we would need texture-space depth information for each texel. Yet another possible improvement that would increase the realism of rendered characters is some volume-based translucency method. Our current approach uses texture-space diffusion and therefore cannot simulate the

light penetrating across the object, only through small local features. There are techniques that could solve this problem for the price of one more geometric pass. Experimenting with these methods could provide a reasonable trade-off between the computational cost and visual quality.

References

[1] Louis Bavoil and Miguel Sainz. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH Talks*. ACM, 2009.

[2] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, SIGGRAPH '08, pages 22:1–22:1, New York, NY, USA, 2008. ACM.

[3] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In Richard J. Beach, editor, *SIGGRAPH*, pages 21–30. ACM, 1988.

[4] Eugene d'Eon, David P. Luebke, and Eric Enderton. Efficient rendering of human skin. In Kautz and Pattanaik [8], pages 147–157.

[5] Craig Donner and Henrik Wann Jensen. Rendering translucent materials using photon diffusion. In Kautz and Pattanaik [8], pages 243–251.

[6] Simon Green. Real-time approximations to subsurface scattering. In Randima Fernando, editor, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 16. Pearson Higher Education, 2004.

[7] Henrik Wann Jensen and Juan Buhler. A rapid hierarchical rendering technique for translucent materials. *ACM Trans. Graph.*, 21(3):576–581, 2002.

[8] Jan Kautz and Sumanta N. Pattanaik, editors. *Proceedings of the Eurographics Symposium on Rendering Techniques, Grenoble, France, 2007*. Eurographics Association, 2007.

[9] Rusty Koonce. Deferred shading in tabula rasa. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 19, pages 429–457. Addison-Wesley, 2008.

[10] Martin Mittring. Finding next gen: Cryengine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 97–121, New York, NY, USA, 2007. ACM.

[11] Musawir A. Shah, Jaakko Konttinen, and Sumanta N. Pattanaik. Image-space subsurface scattering for interactive rendering of deformable translucent objects. *IEEE Computer Graphics and Applications*, 29(1):66–78, 2009.

[12] Mark Harris Shawn Hargreaves. Deferred shading. https://developer.nvidia.com/sites/-default/files/akamai/gamedev/docs/6800_Leagues_Deferred_Shading.pdf.

[13] Michal Valient. Deferred rendering in killzone 2. Online, accessed Feb. 20th, 2012, 2007. Develop Conference, http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf.