Screen-Space Subsurface Scattering
A Real-time Implementation Using Direct3D 11.1 Rendering API.

Dennis Andersen

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full-time studies.

Contact Information:
Author(s):
Dennis Andersen
E-mail:
dean11@student.bth.se

University advisor:
Ms.c. Stefan Petersson
Dept. Computer Science & Engineering

Faculty of Computing                  Internet  :  www.bth.se
Blekinge Institute of Technology      Phone    :  +46 455 38 50 00
SE–371 79 Karlskrona, Sweden          Fax      :  +46 455 38 50 57
Abstract

**Context.** Subsurface scattering - the effect of light scattering within a material. Lots of materials on earth possess translucent properties. It is therefore an important factor to consider when trying to render realistic images. Historically the effect has been used for offline rendering with ray tracers, but is now considered a real-time rendering technique and is done based on approximations of previous models. Early real-time methods approximates the effect in object texture space which does not scale well with real-time applications such as games. A relatively new approach makes it possible to apply the effect as a post processing effect using GPGPU capabilities, making this approach compatible with most modern rendering pipelines.

**Objectives.** The aim of this thesis is to explore the possibilities of a dynamic real-time solution to subsurface scattering with a modern rendering API to utilize GPGPU programming and modern data management, combined with previous techniques.

**Methods.** The proposed subsurface scattering technique is implemented in a delimited real-time graphics engine using a modern rendering API to evaluate the impact on performance by conducting several experiments with specific properties.

**Results.** The result obtained hints that by using a flexible solution to represent materials, execution time lands at an acceptable rate and could be used in real-time. These results shows that the execution time grows nearly linearly with consideration to the number of layers and the strength of the effect. Because the technique is performed in screen space, the performance scales with subsurface scattering screen coverage and screen resolution.

**Conclusions.** The technique could be used in real-time and could trivially be integrated to most existing rendering pipelines. Further research and testing should be done in order to determine how the effect scales in a complex 3D-game environment.

**Keywords:** Translucency, Subsurface scattering, Real-time, Compute shader
B Subsurface scattering code

B.1 Horizontal subsurface compute shader

B.2 Subsurface scattering function
### List of Figures

1.1 Different subsurface scattering scenarios with altering light conditions .......................... 3  
1.2 Light scattering model  .............................................. 5  
3.1 Direct3D 11.1 pipeline  .............................................. 10  
3.2 CPU and GPU comparison  ............................................ 11  
3.3 Compute Shader threading model  .................................... 12  
4.1 Light that penetrates human skin and a made-up material visualizing the light interaction ............................................. 14  
4.2 Rendered images from the first test  .................................. 16  
4.3 Subsurface scattering & Translucency effect results  .............. 17  
4.4 Rendered images from the second test  ................................ 19  
4.5 Subsurface scattering effect results using two different resolutions ............................. 20  
4.6 Translucency effect on the Stanford dragon, human head and BTH logo  ......................... 21  
4.7 Translucency effect results  ........................................... 22  
A.1 A shader program to compute translucency in a pixel shader  .............. 27  
B.1 A Subsurface shader program using the compute shader to acquire data .............................. 28  
B.2 A function in shader code that computes subsurface scattering  ...................... 29
List of Tables

2.1 Hardware used during the conduction of the experiments. . . . . 9
Chapter 1

Introduction

This thesis proposes a real-time approximation to achieve subsurface scattering in screen-space using translucent shadow maps and layered materials with consideration to artists and flexibility. Calculations of the effect will be based on previous algorithms using a modern rendering API. This chapter will give some background to the thesis and explain the techniques this work will be based on.
Chapter 1. Introduction

1.1 Background

(a) Front-lit off

(b) Front-lit on
Chapter 1. Introduction

Figure 1.1: Different subsurface scattering scenarios with altering light conditions.

In some 3D scenes, such as those in game applications, there will often be a wide variety of different objects depending on the scene and purpose. Many of these
objects is given various material and properties that is often used to calculate how light interacts with these objects. Some materials would probably possess some degree of translucency in the real world. This is known as subsurface scattering.

As shown in Figure 1.1, depending on how light hits an object and where the viewer is standing, one could perceive this as two different light interactions. In Figure 1.1a, the subsurface scattering effect is disabled and in Figure 1.1b the effect is enabled. This can be observed in the shadowed parts on the object, where the shadow is much smoother because of subsurface scattering. This effect is achieved by blurring an image, generated with material specific properties, to imitate the scattering of light when it hits the surface. In Figure 1.1d, where the object is illuminated from the back, one can clearly see the translucency effect enabled and how it differs from Figure 1.1c where the effect is disabled. Depending on the properties of a material, such as thickness and number of layers, different amount of light will penetrate an object. This effect is achieved with the use of translucent shadow maps, which is calculated by rendering the scene from the light and the viewers point of view to save depth information. This information is then used to compare the depths from the different shadow maps to determine object thickness.

Physically accurate simulations of subsurface scattering is not practical with modern hardware due to the heavy and time consuming calculations, not at all fit for real-time applications such as games. Because of the realism that subsurface scattering brings to a 3D scene, there has been lot of research on the subject with much focus on rendering realistic human skin for both online and offline rendering and several methods exists to approximate the effect [Barré-Brisebois and Bouchard, 2011; D’Eon and Irving, 2011; Jimenez et al., 2009; Munoz et al., 2011].

Realism is often something 3D game developers pursue using various rendering techniques. This effect is nowadays used in many modern movie productions to create realistic effects. Because the rendering in a movie production does not need to be in real-time, the effect of subsurface scattering would often be more accurate since the timespan to perform computations is greatly increased.

Historically there has been a lot of research on the subject of subsurface scattering in computer graphics, but nothing that was practical enough to use in real-time applications. Early research was mainly focusing on developing models for the bidirectional reflectance distribution function (BRDF), which is a simplification of bidirectional surface scattering distribution function (BSSRDF). It was not until Henrik Wan Jensen’s research on the BSSRDF model [Jensen et al., 2001] in the early 2000s the subsurface scattering effect became practical.
Chapter 1. Introduction

Figure 1.2: Light scattering model

http://lib.znate.ru/docs/index-40271.html

The BSSRDF model, shown in Figure 1.2b, describes how light travels through a material, taking properties such as surface reflection, subsurface scattering, absorption and light transmission, into account and not using the same exit and entry point while the BRDF, Figure 1.2a, assumes light hits and exits at the same point. Of course, the BSSRDF model is significantly more computational demanding compared to the BRDF model, but it is more efficient than than a Monte-Carlo simulation [Jensen, 2003; Mahadevan, 1997], which is a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Jensen's BSSRDF model was used in the production of numerous movies including but not limited to “The Matrix Reloaded” [Wachowski and Wachowski, 2003] and “Harry Potter and the Chamber of Secrets” [Columbus, 2002].

1.2 Related work

As there currently exists a lot of subsurface scattering techniques, those of most interest includes the ones that can be trivially integrated into modern real-time rendering pipelines such as those present in games. Since this thesis is aiming in that direction, these are the ones of most interest. Of course there are techniques that is not directly focusing on real-time or with any consideration to a modern rendering pipeline, but these are still interesting and should be taken into account when developing a technique. But for the sake of simplicity when conducting this thesis, related work will be limited to those of relevance.

Research done by [Jimenez et al., 2009] showed that subsurface scattering could be approximated in screen-space and still preserve a realistic look. To represent this phenomena calculations can be made in texture space, but according to Jimenez this method does not scale well with applications that tends to have a lot of different translucent materials. Another problem with this technique is the texture size, by using larger textures the time to compute also increases. Since
several texels rendered to a scene may end up on the same pixel, these calculations is unnecessary. In his research the selected translucent material is human skin. As Jimenez mentions, skin is a very interesting material to represent in computer graphics because of the many different properties and layers. An interesting aspect to his research is to apply the same technique to achieve a wide variety of translucent material instead of just a skin, making it more dynamic and artist friendly.

[Barré-Brisebois and Bouchard, 2011] spoke at the Game Developers Conference 2011 (GDC) and showed that a screen space method without the use of depth maps to calculate thickness could be used. By pre-calculating ambient occlusion with inverted normals to get an approximated surface thickness, the need to render depth maps is no longer needed making it a very fast approximation method and could be used for games.

The techniques mentioned above is compatible with many existing pipelines making them trivial to integrate, which makes these techniques interesting.

1.3 Aim

The aim of this thesis is to implement a technique to support subsurface scattering in real-time. The subsurface scattering technique will be divided into two separate rendering effects. Multiple translucent materials with non scientific values will be used per scene, where each material should support multiple layers. The scattering of light in the subsurface scattering effect will be implemented in the compute shader using previous techniques.

1.4 Objectives

1. Create a delimited real-time 3D rendering engine using the rendering API Direct3D 11.1 with C++ as the programming language.

2. Implement the proposed technique by utilizing previous work [Jimenez et al., 2009, 2010].

3. Test, evaluate, discuss and draw conclusions.

1.5 Research questions

RQ1: How can subsurface scattering, including translucency, with multiple user defined scene materials and multiple layers per material be computed in screen-space using shader model 5.0?

RQ2: How is performance affected when using multiple user defined translucent
materials, when computing subsurface scattering in screen-space and how does the technique scale with a higher resolution to render higher quality images?
Chapter 2

Methodology

In order to answer the research questions, an application with a suitable rendering pipeline is implemented to measure the rendering time and to generate images to see the visual results. To answer research question RQ1, previous subsurface scattering and translucency techniques must be studied, [Jensen et al., 2001; Jimenez et al., 2010; Li et al., 2009; Munoz et al., 2011] to understand which elements of translucency is of most importance and which properties that is needed in the implementation. Information and references was found using databases such as IEEE, ACM and Google scholar. Some keywords used was; Subsurface scattering, subsurface light transport and translucency. In order to answer research question RQ2, the experiment application must first be implemented in order to measure performance in different scenarios with various number of models, materials and resolutions.

2.1 Implementation

This thesis is focusing on real-time and games, which made C++ a suitable implementation language because it is a common language used in game development. Direct3D 11.1 is used to implement the real-time rendering engine, because this is one of the latest rendering API in the Direct3D series. The subsurface scattering effect is separated into two effects, the scattering of light when the object is front faced and translucency of an object when it is back lit. Some techniques such as the rendering of depth, is performed only once and used in both effects. The steps needed to achieve the effects will now be explained;

**Subsurface scattering, front lit objects/light scattering**

1. Set a single depth-map that is used with all different materials.
2. Draw all objects that share the same material to a separate render target and set the output image containing the objects color/information to the compute shader.
3. Apply a blur kernel with material specific values to the compute shader.
4. Compute the subsurface scattering effect using the subsurface scattering shader compute shader.
5. Repeat steps 2-4 for each different material.
6. Combine each image generated from step 5 into one, in a post-effect pass and add it to the default output image.

Subsurface scattering, back lit objects/translucency

1. Render all objects that should be affected by the effect to a depth map from the lights point of view. It is important that the depth saved is linear or the effect will not be rendered correct [Dunlop].

2. Apply all shadow-maps generated to the translucency pixel shader.

3. Send the current material to the pixel shader and store it as a structured buffer.

4. Draw all objects using the same material with the translucency shader.

5. Repeat step 3 and 4 for each different material.

2.2 Benchmarking

Three experiments is conducted. One experiment will measure performance when the subsurface scattering technique is applied as a single effect, meaning the front and back lit effect is present simultaneously. In the other two experiments the effects is measured separately to see how they perform individually. The performance is measured using D3D11 queries. To measure the performance in a scene the first operation to perform is to save the GPU time-stamp. Thereafter the GPU needs to finish executing the operations, and then the GPU time-stamp is saved again and then compared to the first time-stamp in order to calculate the rendering time.

2.3 Hardware

The experiments will be conducted on a computer with the following equipment.

<table>
<thead>
<tr>
<th></th>
<th>CPU Intel(R) Core(TM) i7-2630QM CPU @ 2.00 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Nvidia GeForce GT 555M</td>
</tr>
<tr>
<td>RAM</td>
<td>8GB DDR3</td>
</tr>
<tr>
<td>OS</td>
<td>Microsoft Windows 7</td>
</tr>
</tbody>
</table>

Table 2.1: Hardware used during the conduction of the experiments.
Chapter 3

GPGPU shader model 5.0

Shader model 5.0 was introduced when the 3D-rendering API, Direct3D 11, was released. New features such as structured buffers, compute shader and unordered access views makes it very flexible to create and manipulate user defined dynamic data on the graphics processing unit (GPU). A limited number of features used to develop the subsurface scattering test application will be explained in this chapter.

Figure 3.1: Direct3D 11.1 pipeline
Chapter 3. GPGPU shader model 5.0

3.1 Structured Buffers

Structured buffers [Microsoft, c] is a structure that can be accessed by index on the GPU. The structure allows customized data containers to be used on the GPU. This type of buffer can be bound to either read or write operations by binding it to a shader resource view or an unordered access view.

3.2 Unordered Access Views

Unordered Access View (UAV) [Microsoft, d], is a structure used on the GPU to allow binding of resources for arbitrary read or write operations. This structure is now available in every pipeline stage since Direct3D 11.1. The number of simultaneous views allowed at once is limited to 64.

3.3 Compute Shader

The compute shader [Microsoft, b] is a new programmable shader stage that allows general purpose programming on the GPU not bound to any other pipeline stages, unlike the vertex- and pixel shader who are connected, operating without interference from other stages, see Figure 3.1. The compute shader is programmable and can be controlled by using code written in High Level Shader Language (HLSL)[Microsoft, a]. HLSL was first introduced in Direct3D 9.0 as part of a programmable 3D pipeline. A compute shader can take advantage of the many processors available on the GPU in comparison to the CPU, see Figure 3.2. The compute shader allows memory sharing and thread synchronization which allows more efficient parallel programming. It is also possible to use a compute shader with hardware supporting Direct3D 10, but with some restrictions.

http://commons.wikimedia.org/wiki/File:CPU_and_GPU.png

Figure 3.2: CPU and GPU comparison
In the rendering API Direct3D 11, the compute shader is controlled mainly by the shader program but is started with the device context Dispatch method. This method starts a desired number of thread groups in three dimensions, see Figure 3.3. There are no restrictions in the compute shader that limits the usage to graphic applications, it is also possible to use for other computing purposes like physics, path-finding or other algorithms that can take advantage of parallelization. Texture sampling and filtering methods exists on the Compute shader but needs explicit instructions. New methods that is available in shader model 5.0 makes it possible to perform atomic operations and synchronization between threads within a thread group on the GPU.


Figure 3.3: Compute Shader threading model
Chapter 4

Subsurface scattering pipeline

4.1 Overview

To answer the research questions, the first step was to create a delimited real-time 3D rendering engine using C++ as the programming language that could serve the purpose of this experimentation. In order for the experiment to work, some basic rendering techniques need to be supported. The first and most important technique is shadow mapping, which is a technique used in computer graphics to achieve shadows by rendering a scene from the lights point of view. This will be used to achieve translucency and the depth needs to be linear [Dunlop]. In the experiment application a deferred rendering pipeline was implemented, but using a forward rendering pipeline instead should give the same visual result at the cost of performance. The implementation of subsurface scattering effect is treated as two separate effects. This is because the subsurface scattering effect relies on a blur pass and the translucency effect does not, making it more practical to divide into separate effects.
4.2 Layers

Materials in our world consists of layers. Human skin for example, consists of several layers. These layers variate in thickness and other properties and will affect how incoming light interacts with the material. As shown in Figure 4.1a the light penetrates an object transporting light within a material, which in this case represents skin. In Figure 4.1b the layers is exposed and we can see where the different layers in the material begins and ends. The material in this figure is not scientific, random properties is used to show the different layers. To represent layers in the experiment implementation, each layer consists of four floating point numbers where three will be weights to the subsurface scattering effect and when calculating translucency. The fourth value will describe variance in the layer. This can be seen in appendix A, where all layers are iterated and included in the process.

4.3 Translucency

This section describes how the translucency effect is computed meaning the object is back-lit.

Translucency is computed in screen-screen space with consideration to Jimenez [Jimenez et al., 2010] algorithm and is then rendered to a separate texture, taking advantage of a deferred pipeline. The computations are made in the pixel shader, making it easy to use per-mesh material giving a different translucent effect depending on the material. By using structured buffers, the switching of different material properties becomes relatively easy. The linear depth map obtained from
shadow mapping is used combined with the depth map from the cameras point of view to compute the thickness of the current mesh. The thickness is then used to determine how much translucency to apply to the current pixel.

4.4 Subsurface scattering

This section describes how the effect is applied when an object is front-lit and not in need of the translucency effect. Prior to Shader model 5, Gaussian blurring was mostly done in the pixel shader in a horizontal and a vertical blur pass [Thibieroz, 2009]. The number of input/output (IO) operations needed to perform this effect on the Pixel shader is far greater than if it was conducted on the Compute shader. In the experiment, the subsurface scattering effect is applied to a texture with objects possessing translucent material properties. A kernel using four floating point values for each sample is used where the three first values is used as weights to the texture color, and the fourth is used as an offset when sampling the input texture. These weights are calculated on the CPU with consideration to the material layer properties for example marble material properties will yield a different kernel than wax material properties. This means different kernels must be used for each material. The depth maps used in this stage will determine if to apply the effect to the current pixel. This is because computations is made in screen space. The depth map is then used to contain the effect to the current object when the samples goes out of range of the object.

4.5 Experiment and Result

The time unit used when measuring rendering speed is in milliseconds using Direct3D query interface. All calculations used in benchmarking is done with single point precision. Profiling was made randomly during the time the application was running using different presets. The start values chosen for the tests, such as blur radius and the number of material layers, were chosen randomly in the tests. The performance was measured until the rendering time was greater than 18 milliseconds or when 11 different values was tested. This was to limit the tests and to keep the frames per second at a reasonable number to consider the use in games. The performance was only measured once per experiment and test. Five different models where used in these tests with various quality with the total of 436508 triangles. The number of triangles present in the scene is shown for future reference, to compare to other scenes and performance measurements. The dimension of the depth maps used in all tests for shadow-mapping was 2048x2048 pixels, and the back buffer resolutions used was 1280x720 and 1920x1080. The aspect ratio of the screen used to render was 16:9, and the resolutions chosen fits the screen dimensions. The two resolutions was used to show how the performance differs between the two, and how heavy the calculations becomes when the resolution is increased. When using a shadow-map/depth map with low res-
olution, the visual effect will have poor quality, but if the resolution is higher the visual effect will have higher quality. The shadow-map resolution was chosen by testing different resolutions, and the result that was best suited to use was the one that had minor impact on performance and best visual effect when rendering.

4.5.1 Experiment 1

In the first experiment the subsurface scattering effect is applied to the scene as it would have been observed in the real world, as a single effect. This means that translucency and diffusion is combined to obtain the full subsurface scattering effect in the rendered scene, resulting in the images seen in Figure 4.2. In this test, performance were logged at 1, 3, 6 and 10 layers. The reason for not incrementing by one layer between the tests is because it would make no major difference performance wise, and we would still able to show how the performance is affected. The blur pass in this test used a static kernel with totally 11 weights giving a radius of 5. This was chosen as a static value since the amount of scattering was enough for the purpose. Shown in Figure 4.3 are the results from the first experiment.
Figure 4.3: Subsurface scattering & Translucency effect results
4.5.2 Experiment 2

(a) Human head rendered with no effect applied

(b) Human head rendered with the subsurface scattering effect
Figure 4.4: Rendered images from the second test.

(c) A dragon rendered with no effect applied

(d) A dragon rendered with the subsurface scattering effect
In this experiment the objects where only front-lit leaving translucency left out. Because of this, the material layers is only needed when computing the blur kernel. The same material was used throughout this test which gives a constant blur kernel that is applied to all objects. If we observe Figure 4.4d when the effect is applied and compare this to the other Figure 4.4c where the effect is absent, we can see how the effect is affecting the shadow edges making them smoother. There is quite a big difference in performance compared to the two resolutions since the surface to apply subsurface scattering on is greatly increased. We can also observe that there is a linear drop in performance regardless of the resolution as shown in Figure 4.5. When the radius is increased we can see that the execution time is increased with a 0.5 milliseconds step when the resolution is 1280x720 and with 1.0 millisecond using a resolution of 1920x1080. The blur radius starts at 3 since a radius lower that that would not be noticeable and have less or no visual effect in the test scene. The maximum blur radius used in the tests is 13 as seen in the graph in Figure 4.5, and the rendering time in milliseconds is closer to 18 giving an fps of 55 fps.

Figure 4.5: Subsurface scattering effect results using two different resolutions
4.5.3 Experiment 3

Figure 4.6: Translucency effect on the Stanford dragon, human head and BTH logo
In this experiment, only the translucency effect is applied. This means that blurring is not done leaving the blur kernel unused in this test since the objects is only back-lit and the front-lit scattering effect is unnecessary. Shown in Figure 4.7 are the results from the benchmarking. When increasing from seven to eight layers the rendering speed is increased. To understand why this happens, more tests on other hardware needs to be done in order to draw a conclusion and understand why this occurs. Beside the small increase in performance, the graph is almost linear regardless of the resolution. Shown in Figure 4.6 is images produced using only the translucency effect, and by modifying the layer properties the visual results are very clear.

![Figure 4.7: Translucency effect results](image)

4.5.4 Result

The result of the experiments shows that it is possible to maintain real-time rendering performance using the proposed techniques based on previous research. By using a modern rendering API, such as the one used in this research, GPU resources can be used to create a flexible subsurface scattering rendering pipeline that scales well with a lot of material and objects. By using the compute shader to achieve the scattering effect, performance is not lost in the same way that it would have if the scattering effect was to be calculated in the pixel shader. Because the effect is based on screen-space algorithms, the performance scales with the resolution. When a larger percentage of the screen is covered with the scattering effect, the rendering speed will drop significantly due to the large amount of area to cover. This could occur if an object is close to the camera. An average of several tests would be necessary to obtain from on each experiment to eliminate any errors that may occur when measuring performance.
Chapter 5
Conclusion, Discussion and Future Work

5.1 Conclusion and Discussion

A subsurface scattering technique has been developed based on previous screen-space algorithms using the rendering API Direct3D 11.1 with shader model 5.0. An experimentation application was developed to test performance of the proposed technique. To draw a conclusion performance was measured using various settings.

By using shader model 5.0 and utilizing features such as unordered access views, structured buffers and the compute shader, subsurface scattering including translucency can be achieved to support multiple user defined scene materials and multiple layers per material. To use multiple materials and multiple layers per material as stated in RQ1, unordered access views and structured buffers are used combined with previous algorithms to achieve translucency in real-time. To achieve the scattering effect, a blur-algorithm was implemented in the compute shader. Structured buffers was used for the blur kernel to have dynamic blur values, but this was later changed to a constant kernel in the shader to achieve better performance. This answers how performance is affected in RQ2, having a dynamic blur kernel will give the option to recalculate kernels without interference in performance compared to a static kernel defined in the shader that will give better performance without interference but with less flexibility. Looking at the translucency effect, there are no major performance loss when using multiple materials with multiple layers as seen in the graphs in experiments 1-3.

The technique works well with shader model 5.0 and there where no problems when utilizing previous work from older shader models and rendering API’s. The effect is trivial to use within modern real-time rendering pipelines using Direct3D 11.1. There are however an issue when moving from the pixel shader to the compute shader when applying the subsurface scattering effect. In order to take advantage of the shared memory in the compute shader when applying the scattering pass, the texture samples needs to be pre-fetched to avoid redundant texture sampling. If the texture is pre-fetched to shared memory and shared within the thread group, a problem arises when the offset from the blur kernel used to calculate the effect is computed. When the offset and the subsurface strength variable gets to big, the index goes out of range when fetching the pre-fetched samples giving the wrong value.

Applying the subsurface scattering effect was done in two different rendering passes. Because blurring in this case relies on an image containing all the objects that should retrieve the subsurface scattering effect, the effects was created in separate passes. This makes it possible to apply only one of the effect if necessary.
Performance is good enough to be able to use the technique in real-time applications. However, performance drops with an increased amount of translucent material that covers the screen. This is due to the nature of the algorithm, since it is performed in screen-space the more surface covered with the effect it scales with the rendering time. This is no major issue since the rendering speed is still feasible.

There are a lot of room for improvement in the experiment application. This is mostly due to the time limitations when developing the benchmarking application to this thesis. Optimizations to the parts running on the CPU side of the applications could be improved to avoid cache misses, redundant rendering API calls etc. which will give a performance boost.

Seen in Figures 4.3, Figure 4.5 and Figure 4.7 the increase in rendering time is almost linear in every test. Comparing performance between translucency and font-lit scattering shows that translucency is cheaper to achieve, which is good since translucency is easily picked up by the human visual system and will contribute to the realism in a scene. Human skin and other materials that possess a high degree of subsurface scattering would however need to apply the full subsurface scattering effect or the eye would notice the hard computer generated surface.

An interesting approach would be to explore the possibilities of using pre-calculated textures that could be used to apply the scattering effect. Since this is where the performance is drops when applying the subsurface scattering effect, this is where research should be done to maximize performance. If pre-calculated textures that represents the scattering effect could be used, the quality of the effect will most certainly be affected.

5.2 Future work

Since the experiment application was developed with limited time, optimization of the rendering pipeline was not prioritized. Better solutions could be used for resource management in the future to speed up rendering time.

Other methods could be tested to calculate thickness. Instead of using shadow mapping techniques, a local thickness map could be precomputed by techniques used in the Frostbite 2 engine [Barré-Brisebois and Bouchard, 2011]. By using a precomputed texture to represent thickness, extra depth maps does not need to be computed but an extra thickness texture per object is needed.


Appendix A

Translucency code

This code snippet determines how translucent an object is by calculating the transmittance using Jimenez’s formula [Jimenez et al., 2010] shown in Listing A.1. First the linear depth of the current shadow map is fetched by displacing the current vertex along the negative vertex normal using a scalar. The depth of the current vertex and the depth from the current shadow map is then used to calculate the transmittance. This procedure is repeated for each shadow map. These computations are done in the pixel shader stage of the pipeline.

```cpp
float ShadowDistance(float3 pos, float3 normal, const int i) {
    // shrink vertex in normal direction to avoid artifacts
    float4 pn = float4(pos - (0.5 * normal), 1.0); 
    // Transform shrunk vertex to light space:
    float4 lp = mul(pn, ShadMapData[i].viewProjection);
    // Fetch linear depth from the shadow map:
    float depth = ShadowMaps[i].SampleLevel(LinearSampler, lp.xy / lp.w, 0);
    // Scale depth with shadow range
    return abs(depth * ShadMapData[i].range - lp.z);
}

float3 Transmittance(float thickness) {
    float3 t = (float3)0;
    thickness = -thickness * thickness;
    for (int i = 0; i < numberOfMaterialLayers; i++) {
        // Sum the transmittance of all layers
        t += MaterialLayers[i].xyz * exp(thickness / MaterialLayers[i].w);
    }
    return t;
}

float4 main(pixIn p) : SV_Target0 {
    // See http://www.iryoku.com/translucency/
    if (sssEnabled == 1 || sssStrength == 0.0f)
        discard;
    float4 transmittance = float4(0, 0, 0, 1);
    // Sum of all shadowmap is the final transmittance.
    [unroll]
    for (int i = 0; i < shadowmapCount; i++)
        { float thickness = ShadowDistance(p.pssW, p.normal, i) / sssStrength;
            transmittance.xyz += Transmittance(thickness) * translucencyStrength;
        }
    return transmittance;
}
```

Figure A.1: A shader program to compute translucency in a pixel shader
Appendix B

Subsurface scattering code

B.1 Horizontal subsurface compute shader

Compute shader program that is used to gather pixels from a source image shown in Listing B.1, that will be used to compute subsurface scattering. Data is pre-fetched and saved to shared memory and thereafter used in several threads. This is done to reduce read and write operations per pixel.

```cpp
def main(uint3 DTid : SV_DispacthThreadID, uint3 GTid : SV_GroupThreadID)
{
    if (sssStrength == 0.0f)
    return;

    // Dimension of the source texture
    float2 colorDim = Source.Length.xy;
    // Dimension of the depth texture
    float2 depthDim = Depth.Length.xy;
    float2 dm = depthDim / colorDim;
    const uint off = BLUR_RADIUS;

    [branch]
    if (GTid.x < off)
    {
        float x = max(DTid.x - off, 0);
        blurCache[GTid.x].color = Source[int2(x, DTid.y)];
        blurCache[GTid.x].depth = Depth[int2(x, DTid.y) * dm].r;
    }
    else if (GTid.x >= THREAD_COUNT - off)
    {
        float x = min(DTid.x + off + SSS.MAX_KERNEL_OFFSET, colorDim.x - 1);
        blurCache[GTid.x + (2 * off)].color = Source[DTid.x, DTid.y];
        blurCache[GTid.x + (2 * off)].depth = Depth[DTid.x, DTid.y + dm].r;
    }
    blurCache[GTid.x + off].color = Source[DTid.xy, colorDim - 1];
    blurCache[GTid.x + off].depth = Depth[DTid.xy + dm, depthDim - 1].r;
    
    GroupMemoryBarrierWithGroupSync();

    // Call SubsurfaceBlur with the current direction.
    BlurDest[DTid.xy] = SubsurfaceBlur(float2(1, 0), DTid.xy, GTid.xy);
}
```

Figure B.1: A Subsurface shader program using the compute shader to acquire data.
B.2 Subsurface scattering function

This function, shown in Listing B.2, is part of the compute shader in Listing B.1 and calculates the actual subsurface effect. This is done by applying an effect that is similar to a Gaussian blur effect, with the current depth from the viewer to the pixel, a scalar to modify the strength of the effect and a blur kernel with four float components to blur various channels differently using one component as an offset when sampling the textures.

```cpp
float4 SubsurfaceBlur(in float2 direction, in uint2 DTid, in uint2 GTid)
{
    // See: http://www.iryoku.com/sss88/

    // Get index for blur cache
    float G = dot(GTid, direction) + BLUR_RADIUS + 1;
    float4 blurColor = float4(0, 0, 0, 1);
    float4 color = blurCache[G].color;
    float depth = blurCache[G].depth;

    if (color.a == 0) return (float4)0;

    float distance = 1.0f / tan(radians(55) * 0.5);
    float scale = (distance / depth);
    float step = (sssStrength * 2) * scale * color.a * (1.0f / BLUR_RADIUS);

    for (int i = -BLUR_RADIUS; i <= BLUR_RADIUS; i++)
    {
        // Calculate the blurCache offset
        float p = G + i - 1 + Kernel[i + BLUR_RADIUS].w * step;
        float3 colorTmp = blurCache[p].color.rgb;

        #ifdef SMOOTH_SURFACE
        float depthTmp = blurCache[p].depth;
        float s = saturate(300 * distance * sssStrength * abs(depth - depthTmp));
        // If the difference in depth is huge, lerp colorTmp back to color:
        colorTmp = lerp(colorTmp, color.rgb, s);
        #endif

        // Accumulate using the blur kernel
        blurColor.rgb += (Kernel[i + BLUR_RADIUS].rgb * colorTmp.rgb);
    }
}
```

Figure B.2: A function in shader code that computes subsurface scattering