# Application of Subsurface Scattering Techniques on Responsive Visualization of 3D Range Scans

Richard Tóth*
*Supervised by: Adam Riečický* †

Faculty of Mathematics, Physics and Informatics
Comenius University
in Bratislava

## Abstract

3D scanning nowadays is not a difficult task when the hardware is available for everyone and the quality is quite good. However, scans compared to real-world objects have still some noticeable differences in terms of realism. In this paper, we implement the screen-space subsurface scattering technique that approximates the light's behavior in translucent objects and how it affects the color of the object. This technique will be used for the enhancement of visual appearance during the rendering of translucent objects captured by 3D scanners. Our work takes scan as an input and uses hierarchical structures and screen space operators to create a virtual surface for point clouds in real-time. This virtual surface will be the base for the subsurface scattering algorithms. We are using the common approaches, where we fit 4-6 Gaussian functions into the laboratory-measured diffusion profile to achieve the effect. We are also implementing a custom solution based on a dipole model to calculate scattering coefficients using just the laser beam from the scanner.

**Keywords:** Rendering, Screen-space, Translucency, Scattering

## 1 Introduction

Nowadays we can capture the reality as no one had ever before. Even a mid-range smartphone has the proper computational power to perform advanced algorithms on images and with the help of neural networks, we can achieve impressive qualities. The goal here is to create a virtual representation (later model) of the object, for further use. The process of model obtaining can be very diverse, which gives us a wide range of available devices and methods to work with. These models can be used from archaeology through medicine, to gaming.

We introduce a flexible method for real time surface rendering, enhanced with subsurface scattering to add the feel of realism into the scans. This solution targets use cases with real time interaction. Live capturing and processing large point clouds with today's computational power is achievable, but the accurate surface reconstruction has significant impact on performance. Capturing massive amount of data, with drone shots or with long range LIDAR scanners is a relevant task, where we want fast visual representations, no need for accurate calculations. The captured data will be unaffected, we are modifying just the rendered result. It could be a good start to replace mesh representations one day, and use the cloud based representations in much larger scale.

The first stage to accomplish correct generation of 3D model with subsurface scattering is to scan the object itself. The most common and suitable representation for these scans is the point cloud. This structure is the general representation format for capturing 3D data based on the sampling of the object surface and creating small uniform sized points in space. This format allows us to manipulate every point independently but we lose topology information in the structure. This property makes the meshing algorithms challenging. The scans should be improved with various post processing techniques to improve the structure, geometry or remove the noise generated by scanners. The overall scan quality can be enhanced with simply using more samples, but the fast processing of these massive clouds produces many new problems to solve. More about this is in Section 2.1.

To add subsurface scattering, we have do deal with light calculations. Collecting all incoming light rays on the object surface is nearly impossible task, but using mathematical models and approximations the effect can be simulated. Another problem in this area is to access the material's physical properties dynamically. All these properties have huge influence in the final result and could increase computational cost.

This paper covers two different, significantly larger areas from point-based rendering. The first problem to deal with is surface reconstruction. This refers to the smooth and continuous solid surface, created from the raw point cloud data. The most widely used solution is to create a polygonal mesh. Even the direct method where we use the points as individual vertices and interpolate them requires

*rtoth94@gmail.com
†a.riecicky@gmail.com

it to iterate through every point. Almost every graphics hardware had been optimized to work well with the tasks like this, but in our case, the solution is very resource-heavy. We have to create a virtual surface and simulate the scattering effect simultaneously in real time.

The second part is the subsurface scattering effect. Every algorithm on the previously created surface, is performed using precomputed values, like the visibility information or the reconstructed normals from the previous phase. In this paper we present two different solutions for visualization. The first solution uses Gaussian's to approximate the object's diffusion profile. The second uses a mathematical model to approximate the material's scattering parameters with basic simplifications and precomputations, which is performed during runtime.

In Section 2, we describe the present state of methods used in the mentioned areas and we mention the older solutions as well with the problems that they revealed. In Section 3, we describe the whole rendering pipeline. In Section 4 we mention the core parts of the surface reconstruction part. In Section 5.1 we present the Gaussian subsurface scattering approach, and in Section 5.1.1 we take a look to the Single scattering approach, where we are trying to determine the scattering parameters. Section 6 presents the results what we got, and Section 7 presents some idea to improve our method.

# 2 Background

In this section we provide a brief overview of the most important methods and algorithms used in the surface creation field and in subsurface scattering calculations as well. Both techniques were already implemented in many different ways, but with new ideas, these solutions are often recreated to fulfill the new standards in computer graphics and requirements of the industry.

## 2.1 Surface reconstruction

The 3D scanners are creating a massive amount of point data most of the time. To visualize them as a solid object, we have to use this information for surface creation. There are direct approaches, using the scanned points as vertices and interpolating the values between captured points. A relatively new research [11] gathered all the important factors and problems with the common techniques.

The first technique [13] shows an useful hint for optimization: Screen space operators. They project the points from the current camera view into a buffer and performing a screen space triangulation on neighboring points. A minor disadvantage of screen space operators is the visibility artifacts at holes or overlapping points. To eliminate the problem, a temporal coherence can be implemented by getting the proper depth value from previous frame. For fast search, they subdivide the area around the point into equal sub-regions and finding the nearest point in every region. The lowest values are selected by the rasterizer, using the *OpenGL* blending. The final normal estimation and triangulation are performed on these nearest points. By limiting the angle of two adjacent points they avoiding flipped normals. They subdivide the area around the pixel into equal regions. This subdivision range is calculated dynamically according to the previously calculated frames to maintain precision and performance. One of the limitations is the temporal instability that happens when the motion is applied to camera.

The auto splatting method [12] uses the previous method as a base solution but adds improvements in key parts. They truly calculate the K nearest neighbors using information exchange between points in rendering pass. They share information between the rendered points and its neighbors in screen space in parallel. The furthest neighbor defines the splat radius for the rendered point. This modification gives much smoother and faster results. The number of neighboring points to take into account is given with a parameter, which indicates the local point density. They discarding the not visible points, so at normal calculations the lack of these information can cause artifacts.

The newest method is presented in [1]. As can be seen in previous methods, using screen space operators and fast algorithms to gather neighboring information is one of the key parts for creating smooth surfaces. The idea here is to use pyramidal neighborhood pattern to gather information around the point. Dividing the area around the point into 8 sectors and finding the lowest occlusion value solves the visibility problem in real-time. The next part of the pipeline uses a visibility map to perform screen space reconstruction with the Pull-Push algorithm. [9]. In this pass they build up a new pyramid from non-occluded pixels, to obtain the missing pixel information and perform the reconstruction of background points. Using this pyramidal structure, they calculate normal values through individual levels and adaptively select the smoothest according to the distance from the camera. The biggest drawback of this method is the larger VRAM usage and the artifacts under camera motion.

## 2.2 Subsurface Scattering

Trying to accomplish rendering that accounts with a subsurface scattering of light is not a new feature in computer graphics. The largest industrial companies like Unreal Engine had this feature in 2011 [10]. The largest impact of this feature is visible on materials like the human skin. The challenge here is to perform these calculations in real-time and on the virtual surface produced in the previous stage of surface reconstruction, in the most compatible way, preferably as a post-processing effect. In this paper, we refer to an older method [8] which was originally developed to enhance human skin renderings. This method uses Gaussian convolution to estimate the diffusion profile. They use numerical fitting to find the proper Gaussian values to adjust

the separable kernel for a laboratory-measured diffusion profile.

In 2001 a [6] mathematical model was presented, which covered multiple variations depending on the material. They designed a model with two major components. The single scattering component what follows a single photon through the object surface, and gives us an approximation how light bounced. The effect is much more dominant in materials like marble. The multiple scattering component diffusion approximation, suitable for multi-layered materials like human skin. This method is based on observation, that light after impact rapidly becomes isotropic. For highly scattering media like skin they use dipole model to calculate the diffusion approximation. Both methods need some basic precomputation for proper simulation.

# 3 Pipeline overview

Our goal is to create a pipeline for fast real-time rendering of 3D scans. We are focusing on translucent objects, where using the common rendering techniques to visualize them as a point cloud could be challenging. For scanning, we use a 3D camera that uses structured light patterns to obtain 3D information. Every stage of the designed pipeline is stored in a separate buffer. We illustrate the individual stages on Figure 1. This strategy allows us to use the calculated data in multiple phases easily, however with the cost of larger VRAM usage.

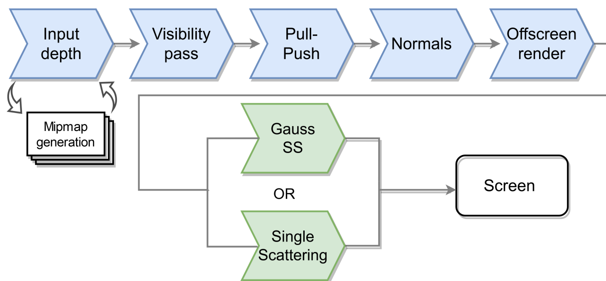


Figure 1: Rendering pipeline overview. The first part of the pipeline focuses on the solid rendering (Top, blue). Before rendering to the screen we apply the subsurface scattering. We offer 2 optional methods for that (Bottom, green).

As an input, we take the depth values from the scanner and render them into a texture. We take this texture and manually build the pyramid structure using mipmaps, and re-render them into the first input buffer. The visibility pass uses this pyramid structure to get the neighboring pixels and calculates the occlusion value for the given pixel. Using this visibility information we set initial weights for the Pull-Push algorithm. At the end of the reconstruction, we have several buffers to work with: original input buffer, visibility buffer, pull buffer, push buffer. In the next part, we take the fully reconstructed pull buffer and calculate normals on adjacent pixels on the adjacent mipmap levels. The final part takes the reconstructed surface with normals and renders them into a final offscreen buffer. Individual stages are presented in Figure 2.

As post-processing, we take the visibility buffer, the reconstructed result, and the depth buffer. We perform a Gaussian blur on the visible pixels. To smooth the effect on the edges of the object, we narrow the Gaussian blur kernel according to the depth differences. The Gaussian parameters are chosen from the precomputed table for the desired material.

The second optional rendering path aims for realistic visualization of the surface translucency, The whole visualization is preceded with a precomputation step, that obtains parameters of the surface, directly from the scan. It modifies the fragment color according to the photon tracing calculations implemented in Section 5.1.1.

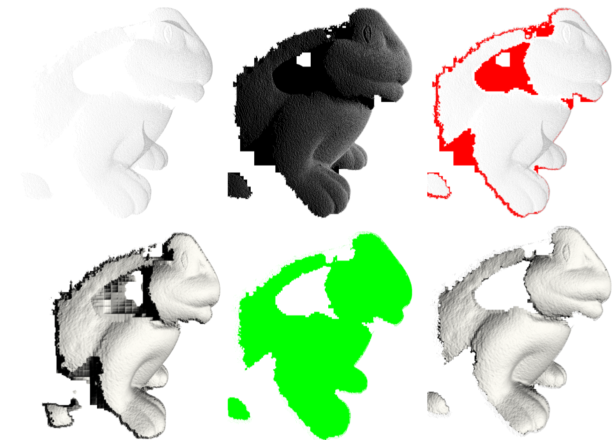

Figure 2: Rendering stages. Top left: input point cloud. Top center: hidden point removal. Top Right: fill removal. Bottom left: reconstruction without fill removal. Bottom center: Filled buffer (Push). Bottom right: reconstruction

# 4 Solid rendering

The idea behind solid rendering is to change the rendered image as if it was a solid surface, using screen space techniques. To render this virtual surface we only need the depth information from scans.

## Pyramid structure

Each buffer in the pipeline uses a pyramid structure created with custom OpenGL mipmap textures. Every pixel in mipmap level *R* contains minimum (depth) value from underlying pixels in the previous level *R-1*. Level 0 is the full resolution texture. We disabled OpenGL default filtering method and implemented our MIN_FILTERING in fragment shader, in Figure 3.

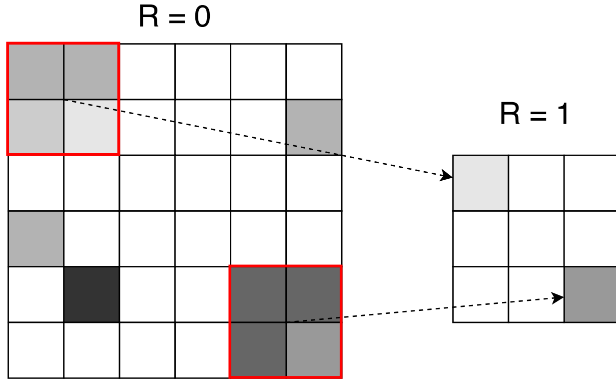


Figure 3: Mipmap generation with custom min filtering. Every MIP level is half the size as the previous one. We take 4 adjacent texels and select the one with the smallest value as the new pixel value for the next level.

## Hidden point removal

To avoid artifacts in the reconstruction phase, we discard the occluded points. With the help of the previously built pyramidal structure, we can effectively get the texel neighboring information.

We calculate occlusion values from every mipmap level, but we store only the smallest one for every neighbor, that in total gives 8 values. First, a pixel is back-projected to 3D using the depth value. For a neighbor, the dot product is calculated. Occlusion value here is the angle between the current point and the neighbor point. Occlusion value $\omega$ for every neighbor $y$ of a current pixel $x$ is expressed as:

$$\omega(x,y) = \left(1 - \frac{y-x}{||y-x||} \cdot \frac{-x}{||x||}\right)$$

The average of the 8 smallest neighbor occlusion values gives the final occlusion value for the pixel $x$. Comparing this average occlusion value to a threshold determines the visibility of the pixel. For every pixel we define a *visibility flag* and store it into the dedicated texture channel for later use.

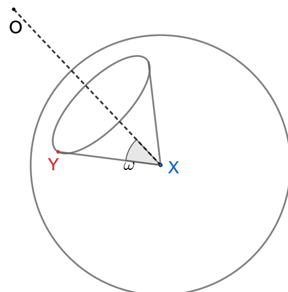


Figure 4: Occlusion value calculation ($\omega$) for point $x$. Point $x$ is back projected from the depth texture, while $y$ is one of the neighboring points. Here $o$ is the camera position.

## Pull Push

The pull-push algorithm interpolates the scattered data and reconstructs the missing information using the pyramid structure [9]. In our case, this information is the input depth map from scans.

In the pull phase, we rebuild the pyramid from the finest level to coarse using the data from the Hidden Point Removal - *HPR* stage. The algorithm uses weights - $w$ - to determine the importance of each point.

At the first level ($R = 0$), we set the initial weights according to the visibility flag, and initial depth according to the HPR buffer.

On the higher levels ($R > 0$), we determine the weights using a pull mask (eq. 1) and the weight values from previous level (eq. 2). The impact of the different pull masks is discussed in [9]. The depth values we get according equation 3., where we use clamped weights,depth values from the previous level and the pull mask.

$$h_{pull} = \left\{\frac{1}{3}, 1, 1, \frac{1}{3}\right\} \tag{1}$$

$$w_{[R+1]} = \sum_{k=0}^{3} h_{pull}[k] * w_{[R]} \tag{2}$$

$$d_{[R+1]} = \frac{Clamp(w_{[R]})}{w_{[R]}} \sum_{k=0}^{3} h_{pull}[k] * d_{[R]_{Pull}} \tag{3}$$

In the push phase, we create another pyramid structure from coarse to finest level. We initialize a push kernel (eq. 4) and a modulation value - $m$ - using the last level values from Pull phase.

$$h_{push} = \left\{\frac{1}{4}, \frac{3}{4}, \frac{3}{4}, \frac{1}{4}\right\} \tag{4}$$

$$m = \sum_{k=0}^{3} h_{push}[k] * d_{[R]} \tag{5}$$

$$d_{[R-1]} = m * (1 - Clamp(w)) + d_{[R]_{Pull}} \tag{6}$$

The value $d_{[R]_{Pull}}$ is the depth taken from the level equivalent to the level currently computed, but from the pull phase. The first level ($R = 0$) of the Push phase contains a fully reconstructed depth buffer.

## Normal Approximation

We calculate normals on every mip level from coarse to fine, as input we take the finest level of push phase, which is a reconstructed depth buffer. First, we back-project neighboring points into 3D space and construct the tangent vectors. Cross of these tangent vectors gives the normal vector per point on a given level. In the end, the coarsest level contains the smoothest normals. We apply normal smoothing with obtaining the current rendered mipmap level as a floating-point number $l$. The formula for this operation is presented in [1]. The 2 closest levels are linearly interpolated using the fractional part of $l$.

### Rendering

In the last phase, we use the previously rendered buffers to determine information for a successful render. We back-project from the depth buffer into 3D space and export positions for remaining points, additionally, we export color values into separate textures. At the end of the Pull-Push phase we get a fully reconstructed buffer, with background pixels included. With the help of the HPR mask, we filter out the points which do not belong to the object. A key part of that is the initial value of the HPR buffers before the whole pipeline. To get proper results we have to clear all buffers to the clipping plane distance.

#### Improvements

With camera motion, a small flickering artifact appears on object edges caused by the *HPR* (Hidden point removal) phase. To reduce this effect we added a custom edge detector. If the current pixel belongs to the edge we automatically set this pixel as visible, preventing from discarding. If 3 neighboring pixels in a row occlude the current point and there are at least 3 an additional pixels wherefrom it is visible, we tag this pixel as an edge point.

When the pull phase first creates the pyramid structure, weight information is added to every pixel. We use this information to filter out false-positive points and avoid filling in the push phase. To get the suitable mip level for filtering we obtain the current level,similarly to the normal approximations [1].

# 5 Applying subsurface scattering

After we managed to create a virtual surface from a point cloud scan, we apply the subsurface scattering as a post-processing effect. We offer two different approaches, to do that.

## 5.1 Gaussian subsurface scattering

Figure 5: Measurement of the material diffusion profile, with a single leaser beam in a dark laboratory environment (a). Visualizing the measured reflectance decline with the increasing radial distance (b) [4]

Subsurface scattering with Gaussian filters is based on the approximation of the diffusion profiles [3]. The diffusion profile describes how the light scatters under the surface of the object. The value depends on many physical properties of the given object and is usually modeled with standard BSSRDF function [2]. As we see on 5, the scattering effect behaves differently for every light wavelength.

We can calculate diffusion profiles, taking already measured scattering coefficients and the dipole model or with laboratory measurements [6]. With a known profile, we need to find N Gaussian functions, which sum matches these profiles as closely as can. We can find these by discretizing the known profile an using optimization tools to perform the numerical fitting. In our solution, we use the already measured Gaussian values for marble and skin from [7].

This method takes the color buffer and the depth-stencil buffer as an input. We apply the Gaussian as a separate horizontal and a vertical 1D convolution filter and a multiple render target workflow and OpenGL alpha blending render the 4 Gaussian passes into a single buffer. To be performance efficient, we apply the blur just on the color buffer only, further constrained with the depth-stencil buffer. Without this improvement, a slight halo effect appeared on edges.

The entire effect is implemented as a simple post-processing effect in a fragment shader. We used the *HPR* visibility flags to get the mask of the object and the reconstructed pull buffer. At the end of the pipeline, we added a specular color component.

### 5.1.1 Single scattering

In 2001 a research presented the dipole and multipole models, including the measurements for obtaining the diffusion profile of any material [6]. Later, another research presented a practical solution for implementing these mathematical models in a fragment shader [5]. We take this idea and implement a shader with a single scattering component only. The diffusion approximation which is the second part of the equation has a less significant effect in single layered materials.

Figure 6: Single scattering.

We trace the path of a single photon exiting a surface point, observed by a camera. To do so, we have to march along the direction opposite to vector path $T_o$ and fire out a ray in a direction to light source, to get the intersection point $P_i$. We are calculating the sum of the incoming light energy on the refracted outgoing ray $T_o$. We determine the incoming refracted vector $T_i$ in point $P_i$, and we assume that light has uniform direction and does not reflect. Using Fresnel equation we determine the reflectance and we define values for phase function [5]. The model can be seen in Figure 6.

Knowing that, the sample point $P_{samp}$ can be calculated as:

$$P_{samp} = P + T_o * s'_o \tag{7}$$

Mathematically what we want to achieve, is the radiance at observed point can be expressed as eq 12.

$$phs = (\vec{\omega}'_i \cdot \vec{\omega}'_o) \tag{8}$$

$$E_i = L_i(x_i, \vec{\omega}_i) \tag{9}$$

$$s'_i\text{-}f = e^{-s'_i \sigma_t(x_i)} \tag{10}$$

$$s'_o\text{-}f = e^{-s'_o \sigma_t(x_o)} \tag{11}$$

$$L_o(x_o, \vec{\omega}_o) = \left[ E_i \frac{\sigma_s(x_o) * F * phs}{\sigma_{tc}} s'_i\text{-}f \right] s'_o\text{-}f \tag{12}$$

The outgoing light energy depends on: the phase function (eq. 8), on the Fresnel term and on exponential falloffs (eq. 10, 11) and the the scattering coefficients. Section 5.2 covers how to obtain and work with scattering coefficients.

In research [5] they presented a code in pseudo shading language with a ray tracing component. We implemented a custom ray tracing method based on the screen space operators. We shoot a ray from every sample point on the refracted view ray into the light direction. We need to obtain the intersection between the ray and surface. As illustrated on a figure 6, our solution is to get a point $(A_1, A_n)$ on the ray vector, and project it back to the texture space. The depth of the point is compared to the surface depth stored in the depth map $(B_1, B_n)$. This iterative process is repeated until:

$$||A_n - B_n|| \approx 0 \tag{13}$$

For the best accuracy the distance between 2 adjacent points on the ray should project to the size of a single pixel.

### 5.2 Parameter estimation

To be able to calculate the single scattering as we presented in Section 5.1.1, we need to obtain the scattering properties of the material. We extract the scattering coefficient $\sigma_s$ and the extinction coefficient $\sigma_a$ from regular diffuse texture [5]. Assuming, that this texture is the result of the scattering events under laser beam of 3D camera projector. Approximation of the BSSRDF function with BRDF, (eq. 18) and fixing the $\eta$ value in Fresnel equation for refraction index allows us to build a table of discrete values from the equation (eq. 18). This table contains the reduced transport albedo eq. 14, which is the ratio of the scattering coefficients (eq. 14).

Detailed description of the equation parameters is presented in [6] on Figure 2.

$$\alpha' = \frac{\sigma'_s}{(\sigma'_t - \sigma'_s)} \tag{14}$$

$$\sigma'_t = \frac{1}{ld\sqrt{3(1-\alpha')}} \tag{15}$$

$$ld = \frac{1}{\sigma_{tr}} \tag{16}$$

$$\sigma'_s = \alpha' \sigma'_t \tag{17}$$

$$R_d = \frac{\alpha'}{2}(1 + e^{-\frac{4}{3}A\sqrt{3(1-\alpha')}})e^{-\sqrt{3(1-\alpha')}} \tag{18}$$

Getting the color of the pixel in every frame allows us to dynamically recalculate the table and get scattering coefficients. With the help of mentioned equations Eq. 15, 17, 16 we can substitute values in the Equation 12 and build the fragment shader.

## 6 Results

We tested our pipeline on a custom dataset, scanned with a Photoneo camera. For testing we used an Intel i3 - 8350K CPU with 4.4GHz clock speeds and a NVIDIA GTX 1060 GPU on a FullHD resolution.

| Dataset | Point count | AVG FPS | 1% low FPS |
|---|---|---|---|
| Spike | 3 457 357 | 32.4 | 26.0 |
| Armadillo | 172 974 | 33.9 | 30.9 |
| Dragon | 435 545 | 33.2 | 28.3 |

Table 1: Average and the low 1% FPS values in different datasets, while simulating user interaction with the view.

In performance tests we have simulated basic user interactions in the scene for 1.5 minutes, with disabled backface culling and with 4X MSAA. For capturing the results we used MSI Afterburner v4.6.2, and we run the tests on the 442.74 NVIDIA drivers.

The reconstruction results are presented on Figure 7. Subsurface scattering results are on Figure 8. The overall quality improves with higher point count but the FPS remains the same. We illustrate the reconstruction and the subsurface scattering results on a larger custom model on Figure 9. and on Figure 10.

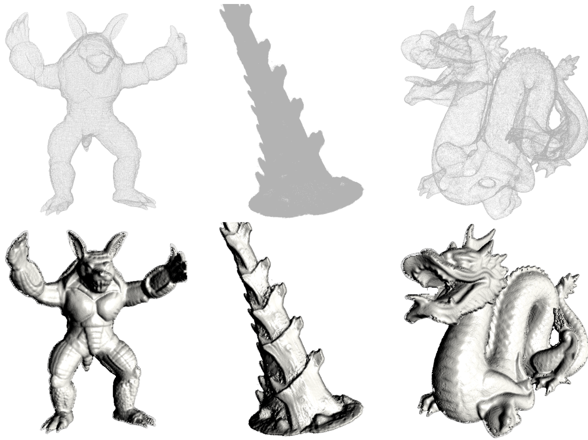

Figure 7: Reconstruction results. Top line: input scans. Bottom line: our reconstruction method

Figure 8: Subsurface scattering example. From left to right: our reconstruction, subsurface scattering with marble diffusion profile, subsurface scattering with skin diffusion profile.

Figure 9: Reconstruction result on a high quality custom scan with 3.8M points

## 7 Future work

Eliminate the flickering artifacts under camera motion with temporal coherence between frames should be a significant improvement in quality. Adding interface to calculate the numerical fitting to find the Gaussian sum should be useful to make the whole pipeline much more robust. After generating a massive amounts of data, using a neural network to determine the scattering coefficients may improve the general usability.

Figure 10: Gaussian subsurface scattering result on a high quality custom scan with 3.8M points

## References


[1] Hassan Bouchiba, Jean-Emmanuel Deschaud, and Francois Goulette. Raw point cloud deferred shading through screen space pyramidal operators. 2018.

[2] Eugene d'Eon and David Luebke. Advanced techniques for realistic real-time skin rendering. in nguyen (ed.), gpu gems 3, 2007.

[3] Eugene d'Eon, David Luebke, and Eric Enderton. Efficient rendering of human skin. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 147–157. Citeseer, 2007.

[4] E d'Eon and D Luebke. Chapter 14. advanced techniques for realistic real-time skin rendering. *GPU Gems*, 3.

[5] Christophe Hery. Implementing a skin bssrdf: (or several...). In *ACM SIGGRAPH 2005 Courses*, pages 4–es. 2005.

[6] Henrik Wann Jensen, Stephen R Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518, 2001.

[7] Jorge Jimenez and Diego Gutierrez. Screen-space subsurface scattering. *GPU Pro: Advanced Rendering Techniques*, pages 335–351, 2010.

[8] Jorge Jimenez, Veronica Sundstedt, and Diego Gutierrez. Screen-space perceptual rendering of human skin. *ACM Transactions on Applied Perception (TAP)*, 6(4):1–15, 2009.

[9] Martin Kraus. The pull-push algorithm revisited. *Proceedings GRAPP*, 2:3, 2009.

[10] Martin Mittring and Bryan Dudash. The technology behind the directx 11 unreal engine" samaritan" demo,". In *Game Developer Conference (GDC)*, 2011.

[11] Reinhold Preiner. *Dynamic and Probabilistic Point-Cloud Processing*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, October 2017.

[12] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. Auto splats: Dynamic point cloud visualization on the gpu. In H. Childs and T. Kuhlen, editors, *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 139–148. Eurographics Association 2012, May 2012.

[13] Reinhold Preiner and Michael Wimmer. Interactive screen-space triangulation for high-quality rendering of point clouds. Technical Report TR-186-2-12-01, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, April 2012. human contact: technical-report@cg.tuwien.ac.at.