UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Adrian Kirikal

Glyptics Portrait Generator – Improved Materials

Bachelor's Thesis (9 ECTS)

Supervisor: Raimond-Hendrik Tunnel, MSc

Tartu 2021

Glyptics Portrait Generator – Improved Materials

Abstract:

In this thesis improvements are done to Glyptics Portrait Generator, which is an application for rendering an engraved gem. A technique is implemented to simulate subsurface scattering in translucent materials, which is required to accurately render many materials used in glyptic art. The existing gem materials are replaced by more realistic marble and multilayered agate materials. This thesis also covers the usability testing, the feedback analysis gives solutions to some problems that arose during the testing.

Keywords:

Computer graphics, glyptic art, subsurface scattering, post-processing effect, rendering, procedural textures, metaballs, museum exhibit

CERCS: P170 Computer science, numerical analysis, systems, control

Gemmipildi generaator – täiustatud materjalid

Lühikokkuvõte:

Lõputöö käigus arendati edasi Gemmipildi generaatorit, mis on rakendus interaktiivse lõigatud kivi renderdamiseks. Rakendusse lisati meetod pinnaaluse hajupeegelduse simuleerimiseks, mis on vajalik, et õigesti renderdada kivilõikekunstis kasutatud poolläbipaistvaid materjale. Gemmipildi generaatoris olevad gemmi materjalid asendati realistlikumate marmori ja mitmekihilise agaadi materjalidega Lõputöös kirjeldatakse ka kasutajatestimist, analüüsitakse selle tagsisidet ning pakutakse lahendusi testimise käigus välja tulnud puudustele.

Võtmesõnad:

Arvutigraafika, kivilõikekunst, pinnaalune hajupeegeldus, järeltöötlemise effekt, renderdamine, protseduurilised tekstuurid, metapallid, muuseumi eksponaat

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1. Introduction

Glyptics is the art of engraving precious or semi-precious stones to resemble a portrait or some other image. According to the State Hermitage Museum [1], stones were either engraved with a raised relief (*cameo*) or with a negative image (*intaglio*). The first known examples of intaglios date back as far as 4000 BC, while cameos appeared around 300 BC in Hellenistic Egypt. Intaglios were most often crafted from single color transparent or semi-transparent stones like carnelian or sard. In contrast, cameos used stones with multiple layers, such as onyx or agate. This allowed the engravings to bring out the different coloristic properties of the layers for an increased level of detail.

Historian Megan Cooper [2] writes that throughout history, engraved stones remained quite popular as art pieces. Ancient Mediterranean cultures depicted mythological scenes and figures, and Romans used to portray significant political figures. However, their popularity peaked during the 19th century when most notably Napoleon Bonaparte and Queen Victoria were known to collect cameos.

Celebrating the 250th anniversary of art historian and glyptic art collector Johann Karl Simon Morgenstern, the University of Tartu Art Museum put together an exhibition[1] to showcase parts of his collection and glyptics in general. The exhibition also showcased Glyptics Portrait Generator (GPG), developed by Vladyslav Kupriienko for his Software Engineering Master's thesis in 2020 [3]. GPG is a program written in C++ which renders a 3D model of an engraved gem in real-time. The gem can be interactively rotated, and its material can be changed with a button. GPG also uses a connected webcam to search for facial landmarks of the user and morphs the gem's face to look like the user.

GPG could display the gem with three different materials: gold, marble, and amethyst. However, they did not look that realistic because their textures were not of sufficient quality and the library that GPG uses for rendering could not show translucency in the gem's material.

In this thesis, support for rendering translucent materials was added to GPG with the *subsurface scattering* effect. Two new procedural materials, marble and multilayered agate were also created to look more realistic than those available before. The new materials can

[1] https://www.kunstimuuseum.ut.ee/et/content/2020-aasta-n%C3%A4itused

be seen in Figure 1. Two new materials: marble on the left and agate on the right. Both use the subsurface scattering technique to achieve a better visual appearance.

Implementing subsurface scattering in GPG was one of the main objectives in this thesis. Chapter 2 gives an overview of how light behaves in semi-transparent materials and shows methods to achieve this effect digitally. Chapter 3 shows how GPG was first installed for development and then goes on to explain how subsurface scattering and the new materials were added to GPG. User testing was also performed to validate the results and the findings from that can be read in Chapter 4. Chapter 5 contains the conclusion and possibilities for future work. The Appendix includes the Glossary, testing materials, an overview of the accompanying files, and the License.

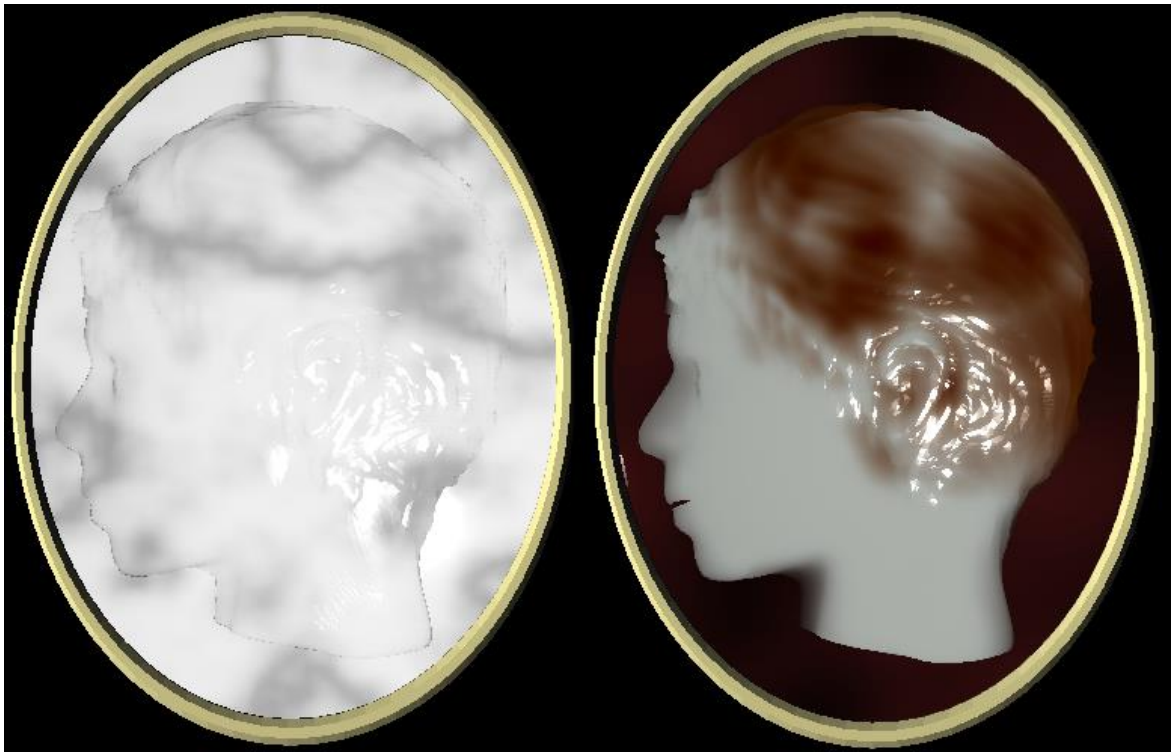

Figure 1. Two new materials: marble on the left and agate on the right.

2. Subsurface Scattering

Every material in the real world exhibits translucency or semi-transparency to some degree [4]. This is especially prevalent in materials such as wax (Figure 2) and skin (Figure 3). When creating a digital scene that needs to look as realistic as possible, translucency must be considered during rendering. This effect is achieved with a technique called subsurface scattering [4].

Figure 2. Light passes through the translucent candle wax.

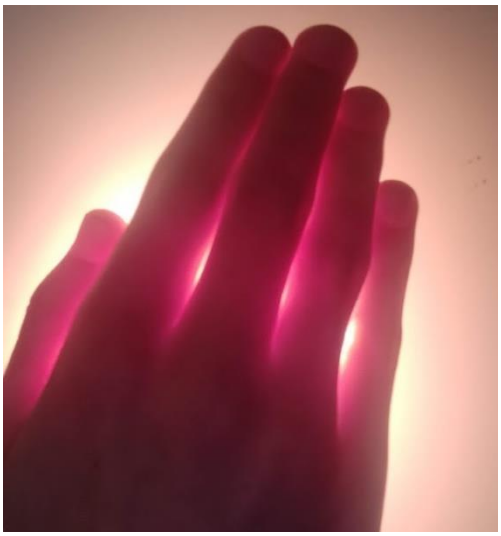

Figure 3. Skin translucency.

Pettineo writes [5] that when light hits a material, it either reflects off the surface or reflects into it. Inside the material, the light that does not get absorbed right away scatters around and might exit some other point on the surface. On exit, the light changes how an observer sees the exit point's color. How the light travels under a surface can be seen in Figure 4.

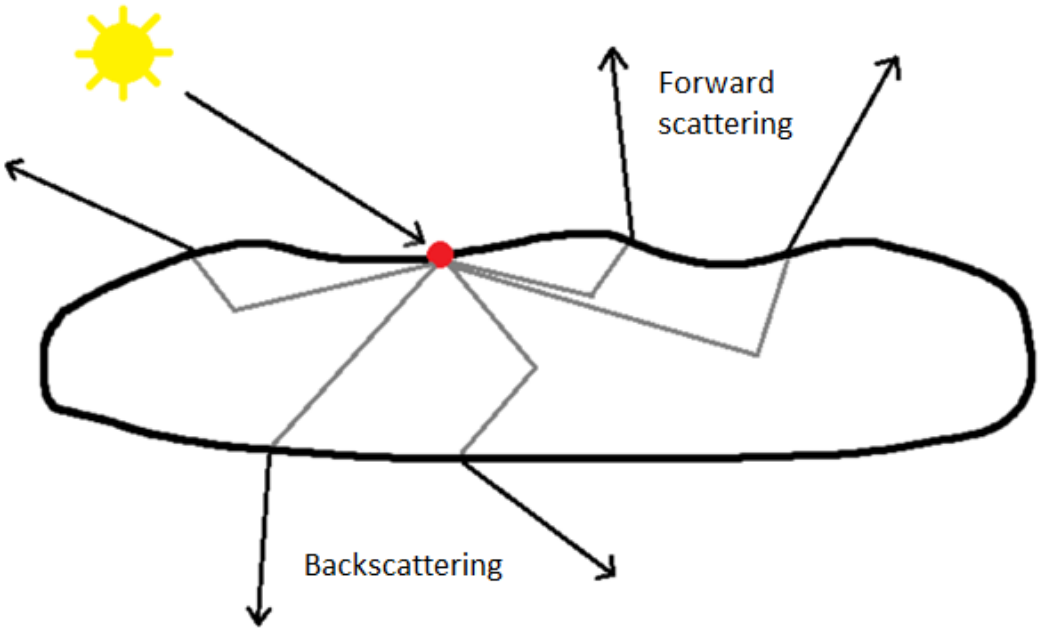

Figure 4. Light traveling inside a surface.

As shown in Figure 4, light in thin and translucent objects can scatter back towards the light source but it can also penetrate through it. These are called *forward scattering* and *backscattering*, respectively, and subsurface scattering can account for both of them [4].

2.1 Subsurface Scattering in Computer Graphics

When a material is not particularly translucent, shading each pixel during a render is simple. This is because the light that enters under the surface in an area of a pixel only ever exits the surface of the same pixel. In this case, the whole subsurface scattering can be simplified to light hitting only the center of the pixel and scattering back in different directions [5].

If some material is more translucent, this approach cannot be used because the light that enters the surface in some pixel might now exit from another pixel. This means that calculating the light exiting from a pixel requires that the light from every neighboring pixel is considered as well.

Traditionally, *volumetric path tracing* has been used to simulate subsurface scattering in renders with high precision [6]. With this method, many light rays are cast for every pixel of the image, and each ray's path through the material is computed separately. This produces realistic results but also requires very long render times. More recent path tracing methods have achieved increased accuracy of the subsurface scattering effect with render times in the order of seconds per frame [7].

When simulating subsurface scattering in real-time applications like GPG, path tracing cannot be used because of its high computational cost. In 2001, a method for simulating subsurface scattering was proposed, which compared the light scattering to a blurring function. This approach was used to approximate subsurface scattering by blurring a model's 2D texture with a Gaussian filter [6].

This texture space subsurface scattering was further improved by separating a 2D Gaussian blur into two 1D Gaussian blurs, which is much faster to calculate. This enabled the usage of subsurface scattering in real-time applications for the first time [6].

It was then found that computing subsurface scattering in screen space rather than texture space has a better performance, and instead of computing the scattering per texture, it can be all done as a single post-processing step [7].

When it comes to screen space subsurface scattering, *Separable Subsurface Scattering* (SSS) [6] is considered state-of-the-art [8]. It is widely used throughout the graphics industry, notably in the game engine Unity [5] and in the 3D graphics software Blender[2]. SSS has seen some improvements over the years, most notably LinSSS, which provides more accurate results at no additional frame time cost [8].

To achieve real-time subsurface scattering in GPG, SSS was chosen as the most suitable method. Although LinSSS gives more accurate results, SSS only needs one shader[3] and a relatively simple post-processing technique to use. Additionally, the authors of SSS provided the diffusion profile of marble in their paper's supplementary materials[4], which reduced the implementation effort considerably.

[2] https://developer.blender.org/diffusion/B/browse/master/source/blender/draw/engines/eevee/shaders/effect_subsurface_frag.glsl

[3] https://github.com/iryoku/separable-sss/blob/master/SeparableSSS.h

[4] http://www.iryoku.com/separable-sss/

3. The Glyptics Portrait Generator

Before any work on subsurface scattering or new materials could be started, GPG and its dependencies had to be installed. GPG's source code was available to download from the version control hosting provider GitHub[5]. The source code, however, did not contain information about what dependencies it needed to run.

3.1 Setup

Figuring out what libraries and what versions of those were needed took some effort. In GPG's source files was a file named CMakeLists.txt, which is used to describe the compilation process of the executable. It also contained the names of the libraries that were required during the compilation, so all of these were installed.

Some of these libraries were installed using vcpkg[6], a tool created by Microsoft for downloading and installing C++ libraries in Windows. Since the whole project was built as a 64-bit application, a command line argument *--triplet x64-windows* had to be specified when installing the packages with vcpkg. Table 1 shows which libraries were installed in this way, and contains information about how GPG uses them.

Table 1. Libraries installed through vcpkg and their usages in GPG.

Name	**Usage**
realsense2	Gathering depth information from an Intel RealSense camera, if one was available.
boost	Dependency for the library eos (see Table 2).
dlib	Detecting faces from images.
OpenCV	Capturing images from the camera.

Besides those dependencies that were installed using vcpkg, Table 2 shows those that would not work with this approach and thus were installed by cloning the libraries' Git repositories and building them manually.

[5] https://github.com/AllysanderStark/glypticsgenerator

[6] https://github.com/microsoft/vcpkg

Table 2. Manually installed dependencies, their versions, and usages in GPG.

Name	Version	Usage
Ogre	1.12.4	Handles the scene rendering and user interface.
eos	1.2.1	Detecting facial landmarks from an image and based on them morphing a model of a face to look like the face from the image.

For eos, no vcpkg package has been created, so there was no other option than to install it manually. Ogre, however, has a vcpkg package, but it does not include the capabilities to compile Cg shaders[7], which had to be turned on manually during Ogre compilation. Also, the specific version of Ogre was chosen because it is the last version that includes the HLMS[8] component, which is used in GPG.

When starting up GPG, it first loads the facial landmark data to enable face detection and model morphing. All of this data is over 400MB, so the time the application spends reading and processing it before the engraved gem is visible is around 30 seconds. This meant that every little change in application code required waiting for the application to start up again, which quickly amounted to quite a lot of time just waiting around.

To alleviate this problem, a C++ compiler directive was added, which told the compiler to ignore the face detection code when this directive was enabled. This successfully reduced the startup time to around one second and made the development process a lot faster. Since the code for manipulating the face model was ignored, then the gem lacked a face in this case, as seen in Figure 5. However, most of the new features added to GPG did not need the face to be present on the gem, so during most of the practical work, this directive was kept turned on.

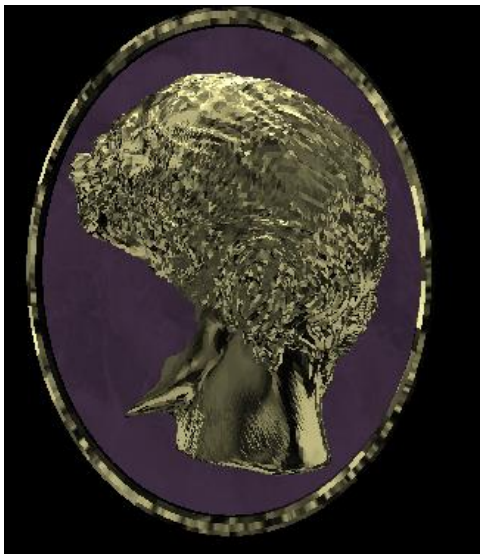

Figure 5. Gem without a face.

[7] A high level shading language developed by NVIDIA

[8] High Level Material System

3.2 Subsurface Scattering

To achieve subsurface scattering in GPG, it was first necessary to find out whether such an effect already exists for Ogre. Implementing a new method would be considerably more time consuming and any existing method would probably be already thoroughly tested and thus the effect would look better.

For the Ogre rendering engine, there has been only one method for calculating subsurface scattering[9], created by Ogre developer Matías Goldberg[10]. Other than that, there have been some discussions in Ogre forums [11] regarding Separable Subsurface Scattering [6]. However, no one has used that in Ogre yet.

Goldberg's method was first tested as a suitable method for achieving subsurface scattering in GPG. Because since it was already created for Ogre, it only required modifying some existing material definitions to get it to work. In the end, the process took quite a while due to the author's inexperience with the Ogre framework. However, this saved a lot of time when implementing the new materials and the final method of subsurface scattering later on.

Figure 6 shows the result of Goldberg's method. This picture was taken early in the development process when the face model did not yet have a texture applied to it. The effect is visible on the rest of the head. This method dimmed the thicker parts of the model as seen in the center and lightened up the edges where the model was thinner.

[9] https://forums.ogre3d.org/viewtopic.php?t=40352

[10] https://twitter.com/matiasgoldberg

[11] https://forums.ogre3d.org/viewtopic.php?t=68805

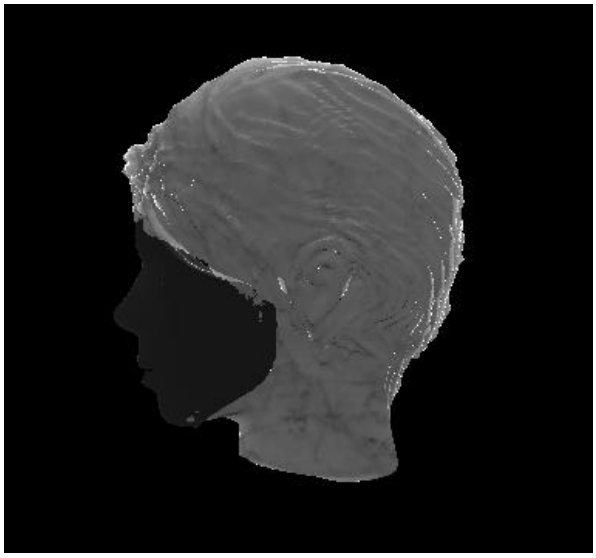

Figure 6. Result of Goldberg's subsurface scattering.

The method rendered the scene four times. First, it rendered the scene normally, saving the resulting image in a color buffer. Then it did two additional renders, one where only the faces towards the camera were visible and the other where only the faces that pointed away from the camera were visible. In both cases, the scene depth information was saved. saving the scene depth information in both cases. The depth information from these two renders was then used in the fourth render to calculate the model's thickness at any point. Based on the thickness, the image in the color buffer was then modified to dim the thicker sections of the model while making the thinner parts more visible.

As this method only simulates the backscattering part of subsurface scattering while ignoring forward scattering completely, the result is not that great. In the scene, the light is in front of the model, but it looks like there is a bright light on the other side. The fact that the result does not look as good as true subsurface scattering would, is also something Goldberg mentioned[12]. Because of the unsatisfactory results, it was decided that another approach was needed to achieve realistic subsurface scattering.

[12] https://forums.ogre3d.org/viewtopic.php?p=380899#p380899

3.2.1 Separable Subsurface Scattering

SSS works as a post-processing filter. After the scene has been rendered normally, the method then goes over the image twice, first blurring it horizontally and then vertically. During a pass for a single pixel, a fixed number of neighboring pixels are also sampled from the image, and then averaged together to produce a blurred pixel.

How far from the original pixel the other pixels are sampled from determines the visual strength of the blurring effect. This can be seen from Figure 7, where in the left, the blurring strength is low, whereas in the right, the strength is high.

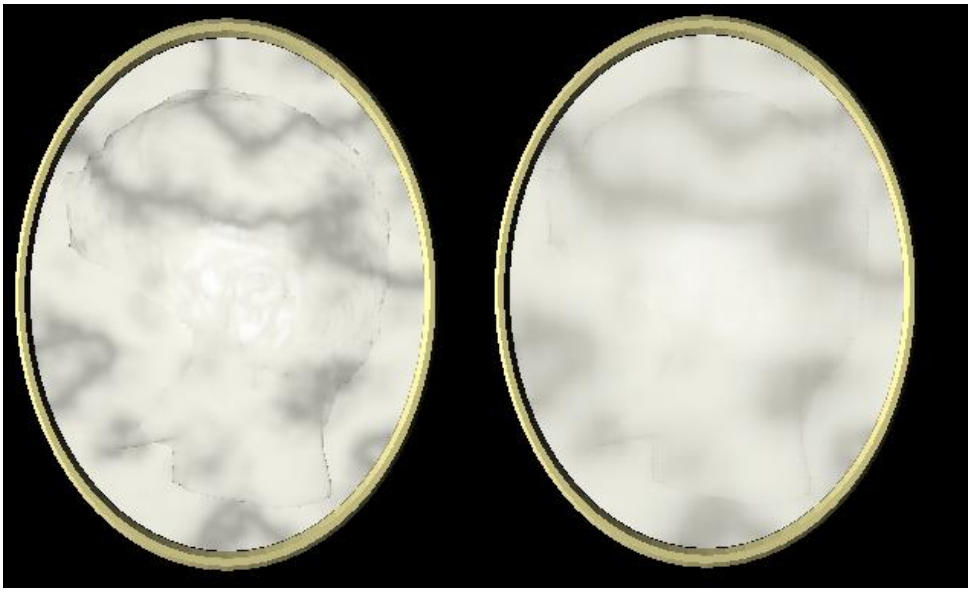

Figure 7. Blurring with different strengths.

SSS assumes that when projecting a 3D scene onto a 2D plane, then points close to each other on the model get translated to neighboring pixels on the screen [5]. However, when the model has a sharp difference in height somewhere, the previous assumption might be false. When, for example, looking straight down at a corner of a room, then points in 3D that might be far away from each other can be very close on the rendered image. This is illustrated in Figure 8, where the two spheres are not near each other, but in the rendered image on the right they are touching.

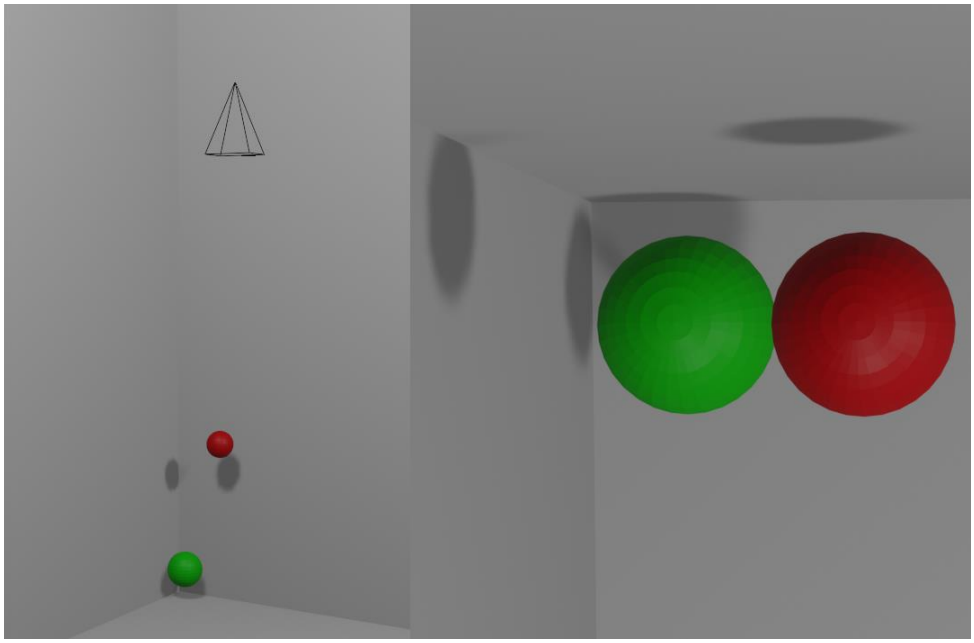

Figure 8. Separate objects are touching after rendering.

When ignoring this effect during blurring, then details provided by the model's geometry are lost and the model looks hazy. To compensate for this, the scene depth is sampled from the neighboring pixels' locations, and when the difference to the original pixel's depth is too large, then the neighboring pixel is ignored. A comparison can be seen in Figure 9, where on the right the compensation is turned off and many details are lost.

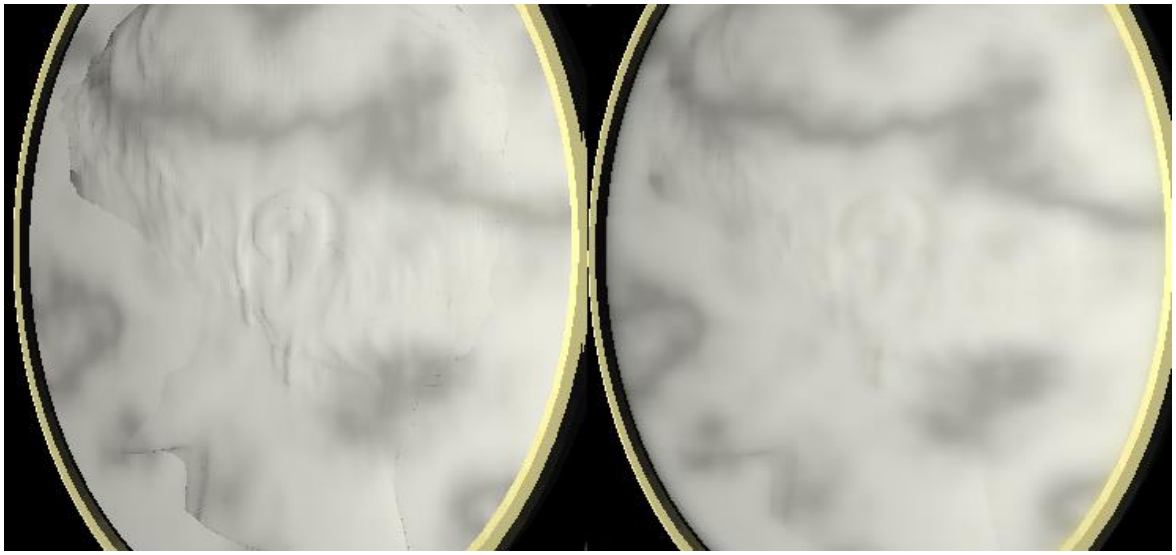

Figure 9. Depth compensation.

Implementing SSS in GPG was relatively straightforward. Ogre supports post-processing filters natively with its *compositor framework*, which allows the definition of *compositors* (or rendering pipelines) using *compositor scripts*. The compositor scripts are text files that tell Ogre how to achieve the desired post-processing effect. They are easily reused and modified because they do not require the usage of the Ogre API, which is why SSS was ported into Ogre as a compositor.

The SSS compositor first took in the original rendered image, which Ogre automatically provides to each compositor. The image was then passed as an input to the SSS shader, and a blur pass was performed on the vertical axis. The vertically blurred image went through the shader once more, this time with the blur applied horizontally. Both passes also got the scene depth buffer as an input to compensate against the loss of fine detail. See Figure 10 for an illustration of the compositor's rendering pipeline.

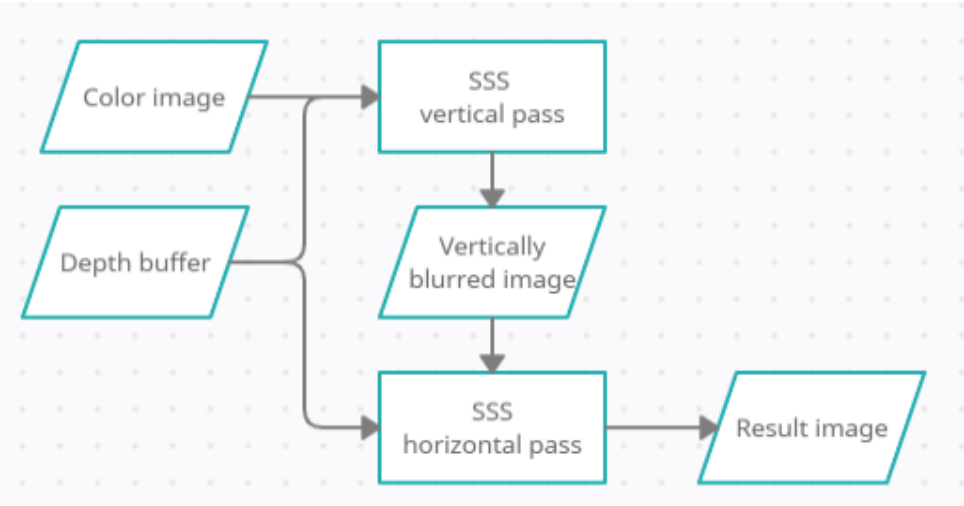

Figure 10. Pipeline of the SSS compositor.

When taking the average of neighboring pixels in each blur pass, the operation was not just a simple arithmetic average. The blurring was done using a pre-computed Gaussian kernel shown by Jimenez, J. et al. to give physically accurate results. In the case of a Gaussian kernel, the pixels near the center have a much higher weight in the average than those further away.

The authors of the SSS provide pre-computed kernels for several different materials in their paper's supplementary materials. Included is also a kernel specifically tuned to match real

marble's diffusion profile as closely as possible. The kernel was provided as a Matlab[13] file which was converted to a CSV file using a free Matlab alternative GNU Octave. The benefit of using a CSV was that it is easily readable line by line.

The kernel had a sample size of 440, which meant that if it were to be used in GPG without any modifications, then for each pixel in the rendered image, 440 other pixels had to be sampled. This amount of computation would have caused the render times for each frame to get too long, so the sample size had to be reduced.

The supplementary materials of the SSS paper also included a demo application. It had a method for converting the large kernel into a smaller one, which was imported into GPG. This method can be used by running GPG with a command line argument *-kernel x*, where *x* is the desired sample size. The downsized kernel is then printed to the console and can be used in the SSS shader. After some experimentation, a sample size of 21 seemed to give the best balance between the computational complexity and the visual quality of the effect.

The user interface (UI) of GPG was changed to allow SSS to be turned off and on with a button. Additionally, a slider was added which changed the strength of the effect and a button that reset the slider to a known value. The new UI elements can be seen in Figure 11.

Figure 11. New UI elements in GPG: toggle button (bottom left), slider (top right), and buttons for toggling the slider and resetting its position (top left).

[13] A programming and numeric computing application.

A bug in the interaction of the Ogre UI framework and the camera controls was found where the camera still moved while dragging on the slider. This proved quite distracting while trying to observe the visual changes that the slider's movement produced. As a workaround, the slider is only available after toggling the "Adjust SS" button, and while it is toggled on, the camera is locked in position.

3.3 Materials

GPG was able to show the engraved gem in three different colors: gold, amethyst, and marble. They can be seen in the same order from left to right Figure 12. It was decided with the exhibition's curator Jaanika Anderson that GPG needed a proper marble material and a way to display multilayered materials.

Figure 12. Existing materials in GPG.

The virtual engraved gem in GPG consisted of five different 3D models, as seen in Figure 13. The material for the base edge and the base back models was kept as a gold color. The majority of the efforts went towards creating the textures for the base center, head, and face models.

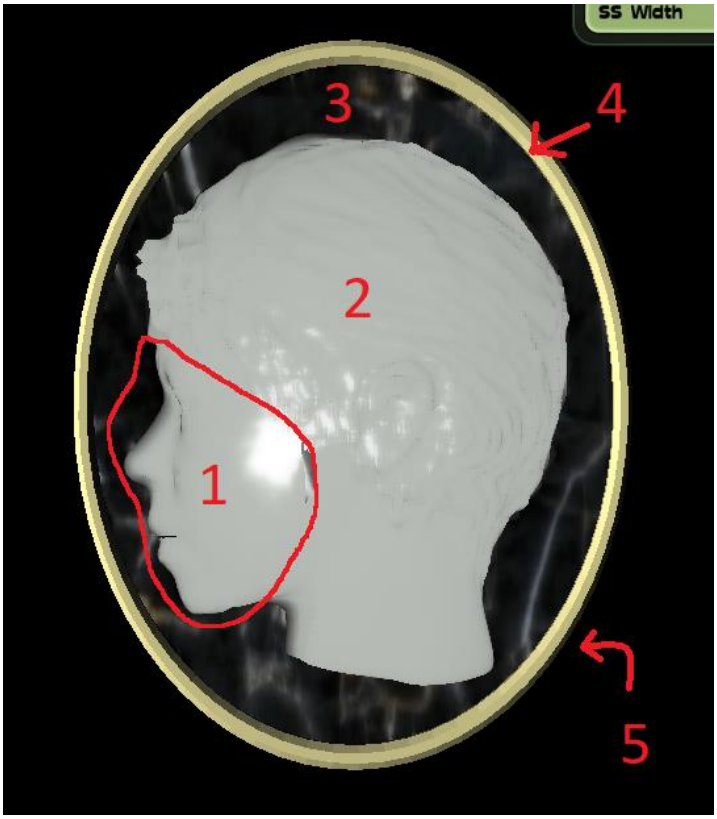

Figure 13. Different 3D models of the gem: face (1), head (2), base center (3), base edge (4), base back (5, not visible from the image).

The simplest way to create new textures for the models would have been to use image textures and configure Ogre to use those. This was tricky because the face model lacked *texture coordinates*[14], so there was no easy way to map an image to the face model's geometry. Also, it would have been difficult to align the texture of the face and the head so that the texture would appear continuous.

For these reasons, it was decided that the textures had to be procedural. This means that the color values at some point are not looked up from an image but instead calculated by some function. With this approach, the problems listed above were easy to fix. Additionally, procedural textures are easily modified with parameters, which when changed, instantly create new variations of the same texture. However, as a downside, the creation of decent-looking textures would be more complicated than just downloading and using a realistic image.

[14] A set of coordinates associated with each vertex that show which point on the image corresponds to that vertex.

Procedural textures use a function to determine a pixel's color, so it was important to decide what value the function would use as input. Often, texture coordinates are used to procedurally calculate the colors[15], but as mentioned above, those were not available for the face model. The choice was then between object coordinates, where the origin point is different for every object, and world coordinates, where each object has the same origin point. After some experimentation, it was decided that world coordinates are the best choice for using as input to the procedural textures. With them, the texture continuity between the face and the head models was not a problem.

The downside of using world coordinates in procedural textures is that when the model's location changes, then the generated texture changes as well. In this use case, it is not a problem because when the gem is rotated or scaled by the user in the application, then the models themselves do not move, instead the camera orbits around the world's origin.

Each procedural texture was realized as a pair of OpenGL shaders: one vertex shader and one fragment shader. They are small programs written in the OpenGL shading language that tell the GPU how to process model vertices and which colors to show for every pixel on the screen, respectively. It is important to note that data can be sent from a vertex shader to a fragment shader when some data is associated with a model's vertices that the fragment shader needs access to. In the case of the created procedural textures, the vertex shaders sent the current vertex world coordinates to the fragment shader, which used it to calculate the color for every pixel.

3.3.1 Marble

One of the first examples of procedurally generated marble texture was used to decorate a marble vase [9], seen in Figure 14. It was created with *gradient noise*, a type of functions widely used in computer graphics that assign a pseudo-random vector to a point in space [9]. Additionally, when two points are close to each other, then the resulting values are also similar. In effect, this makes a gradient noise function output locally smooth random numbers. The specific version of gradient noise that K. Perlin used to texture the vase is called *Perlin noise*. In 2001 Perlin released *Simplex noise* which is an improved version of

[15] https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/procedural-texturing

Perlin noise. Figure 15 shows the output values of two-dimensional Simplex noise at each point on a plane.

Figure 14. Marble vase by K. Perlin [9].

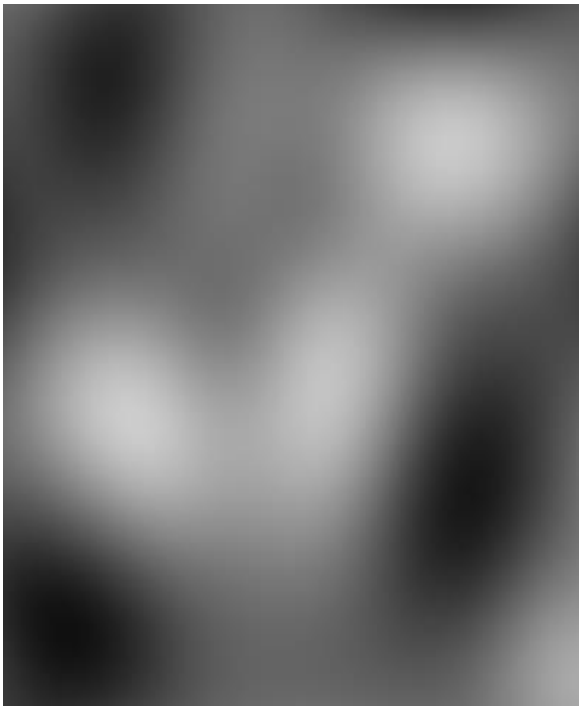

Figure 15. 2D Simplex noise.

Lagae, A. et al. describe the following function for generating a marble texture:

$$C\big(x + N(x, y, z)\big), \tag{1}$$

where x, y and z are the spatial coordinates, N is a function to generate turbulence using gradient noise, and C is a combination of a *colormap* and a periodic function that generates parallel vertical lines across the surface. Figure 16 shows the procedural marble texture that Lagae, A. et al. achieved and how their intermediate functions' outputs look on the vase.Figure 16. Lagae, A. et al.'s

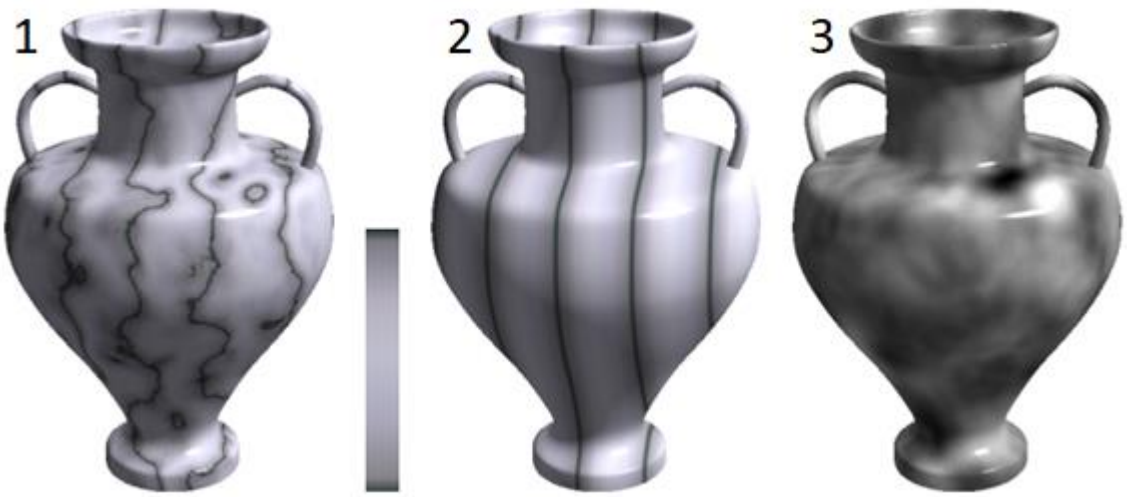

Figure 16. Lagae, A. et al.'s procedural marble texture (1), function *C* (2), and turbulence function N (3) [9].

To construct the function C, the colormap function was taken from a JavaScript game and rendering engine BabylonJS[16] which used a slightly modified version of Lagae, A. et al.'s method for a procedural marble texture. This colormap took a number in the range of [-1, 1] as an input. When the value was very close to -1, the output was a dark black color, but then as the input value increased, the color quickly turned to marble white.

The *x* coordinate, which was used as an input to the colormap function, was first passed through a sine function so that it was both in the correct range and periodically approached zero, where it produces a dark stripe. The resulting combined function was thus:

$$C(x) = colormap(\sin(x)) \tag{2}$$

The visual output of this function came out to be very similar to that shown by Lagae, A. et al. [9], so it was taken into use in GPG without any modifications. How it looks can be seen in Figure 17.

[16] https://github.com/BabylonJS/Babylon.js/blob/master/proceduralTexturesLibrary/src/marble/marbleProceduralTexture.fragment.fx

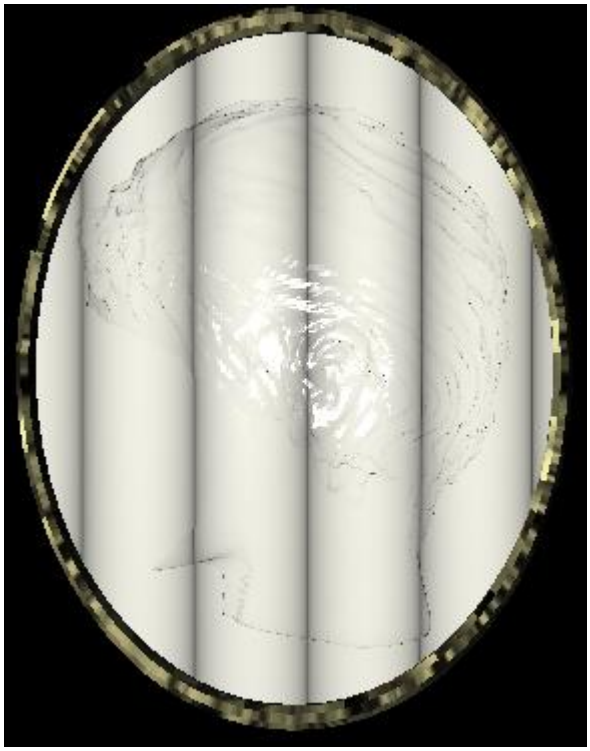

Figure 17. Function *C*.

To distort the black lines to look more like real life marble, a turbulence factor was added to the x coordinate (function N in equation (1)). The turbulence was achieved by averaging multiple layers of three-dimensional noise, each one with double the frequency than that of the previous layer [9]. The method for generating noise was chosen to be Simplex noise mainly because of its lower computational complexity when compared to traditional Perlin noise.

How each additional layer affected the resulting noise can be seen in Figure 18, where the number in the top left of each gem shows how many increasingly detailed layers of Perlin noise were used in that instance. It was visually determined that five layers of noise was sufficient to bring out small enough details in the marble texture.

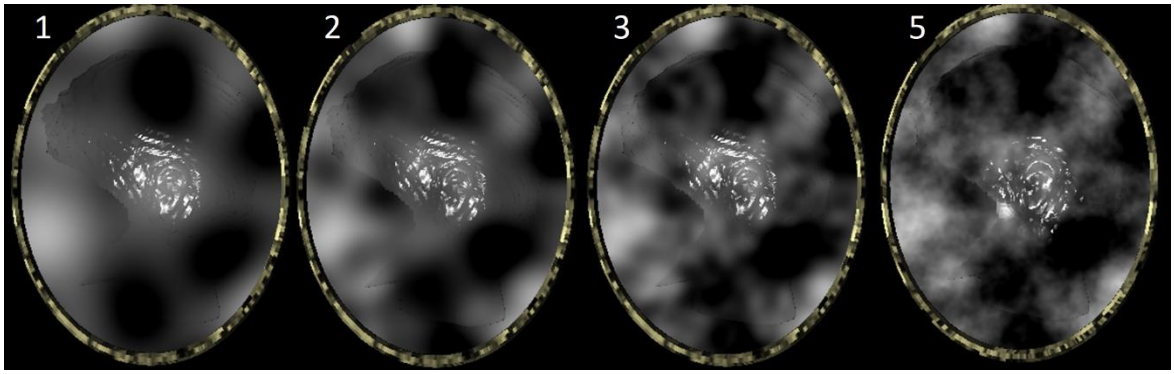

Figure 18. Different amounts of layers of Perlin noise.

The turbulence was then combined with the function C to produce the first version of a marble texture which is comparable to that shown by Lagae, A. et al. [9] and it can be seen in Figure 19.

Figure 19. First version of procedural marble texture.

When comparing the result to images of real-life marble, like that in Figure 20, then it was noticed that usually, the darkest parts of the lines are not that dark like they came out in this case. This was solved by multiplying the input of the colormap function by 0.95, so the highest input value never reaches exactly -1, and so the output color near the lowest values is not that dark.

Also, in real marble, the lines usually break up often and sometimes form island-like patches. To create this effect, the x variable in Equation (1) was experimentally multiplied with different parameters, which each distorted the dark lines in some way. The square of the y coordinate was chosen, which gave the result shown in Figure 21.

Figure 20. Example of the texture of real marble[17].

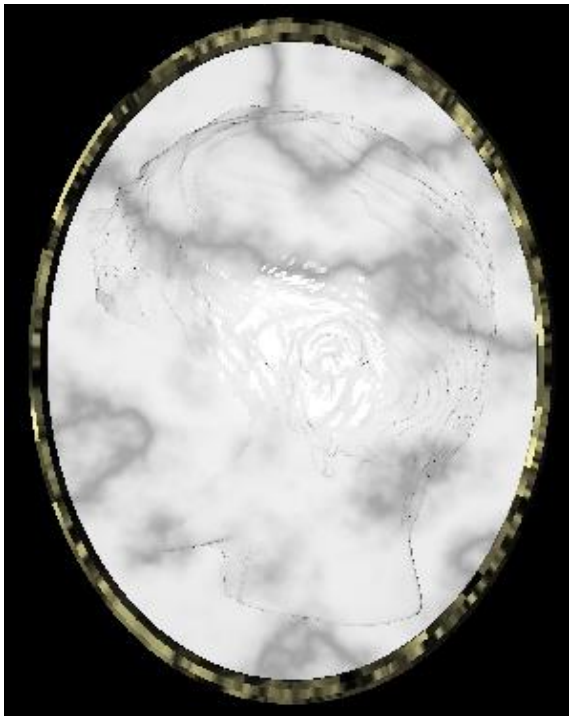

Figure 21. Marble texture after changes.

After the described changes, the function to generate a color value for each pixel looks like this:

$$color = colormap\big(0.95 \cdot \sin\big(y^2 \cdot x + N\ (x, y, z)\big)\big) \quad (3)$$

3.3.2 Multilayered material

Since cameos have been created with materials that have multiple layers, then it was decided with Jaanika Anderson that support for rendering these was going to be added to GPG in

[17] https://lorenacanals.com/blog/wp-content/uploads/2017/09/marble-2371776_1920.jpg

addition to previously mentioned marble. A good example of a multilayered engraved gem can be seen in Figure 22, which is made of agate and separates the background from the cameo with different colored layers. Additionally, there is a third darker layer on top of the cameo, which brings out the decorations on the head and near the chest. All these visual effects were tried to emulate in GPG.

Figure 22. Agate cameo[18].

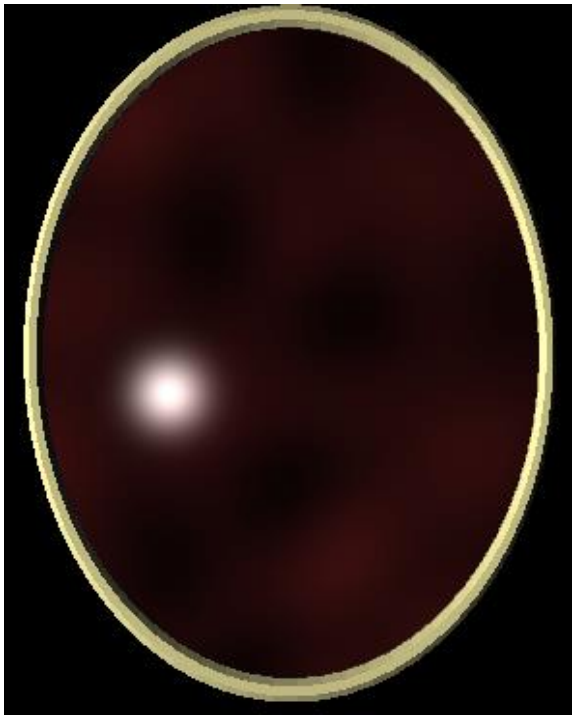

Figure 23. Colored background.

To have the background be a different color from the head was simple. The background was given a different material from the rest of the head, which could be separately colored. The background was given a dark red color, sampled from the edge of the cameo in Figure 22.

With only one color in the background, it did not look that natural, so Simplex noise was used to add some variation to the background's color. The noise value at every point on the surface of the background was used to multiply the dark red color, so where the noise value was small, the color became darker as a result. Figure 23 shows how this background color modulated with Simplex noise looks.

In some cases, the morphed face does not exactly line up with the rest of the head, and there is a gap between the two models. The gap shows the background beneath, and when the rest

[18] https://www.rct.uk/collection/65238/emperor-claudius

of the head is white, this gap is very visible. As a workaround, the background was made white right under the gap, which successfully hid it. Figure 24 shows how the visible gap on the left is hidden on the right, because the background is the same color as the head near the gap.

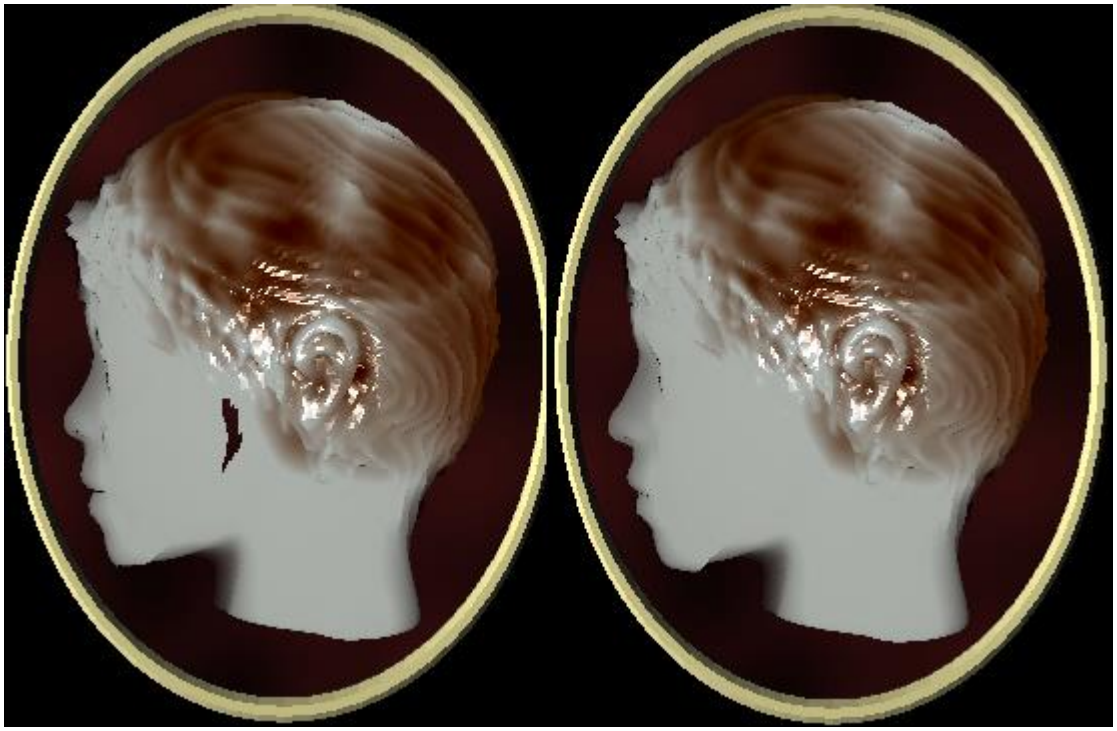

Figure 24. Visible gap on the left and hidden gap on the right.

Coloring the head was more complicated. The whole head was given a white base color that was also sampled from the image of the same cameo in Figure 22 that the background's color was taken from.

To create a multilayer effect, first, a small fade was given to the edges of the face, where the thickness of the white layer was very small. In real cameos, the white layer in thin places would show the layer beneath and thus be darker than the rest of the face.

This was achieved by using a colormap function with the scene *z* coordinate. When a pixel on the face was close to the background, it was a similar dark red color to it, and when moving away from the background, the color faded to white. The result of this is shown in Figure 25.

This approach worked quite well in most cases. However, where the edge did not meet the background smoothly, the effect was not visible, as is the case with the bottom of the neck in Figure 25.

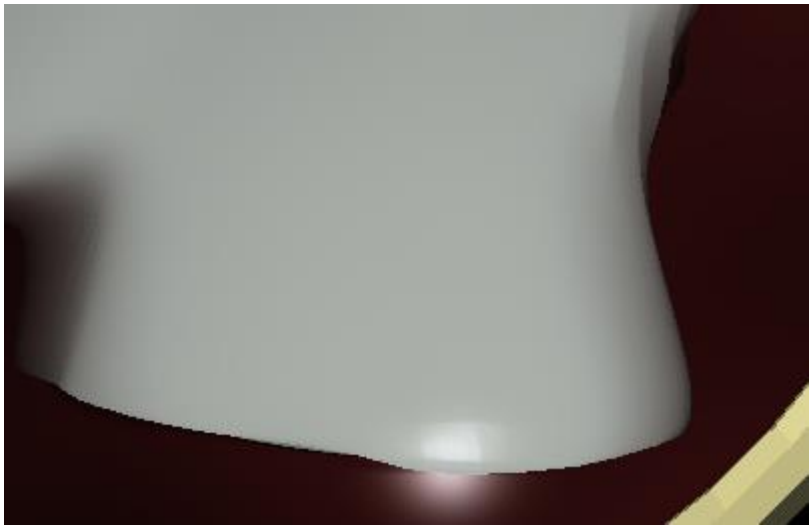

Figure 25. Faded edges of the face model.

In the example of a real-life multilayer material in Figure 22, the decoration on the hair is a separate layer and thus darker. In GPG's case, there was no such decoration, so the hair itself was colored as if it was in a different layer.

At first, an attempt was made to achieve this using the same colormap approach as the fading of the edges. However, the model was not the right shape for this, so when applying the colormap, the color was darker even where there was no hair. Also, it became hard to distinguish between different pieces of hair when they were a solid dark color. Because of these problems, it was decided that the hair would be colored a different way.

The model's hair approximates a dome shape with a rough surface. To paint it a different color than the rest of the head without having to manually do the painting, an approach with *metaballs* was chosen. This means that under the model's surface, a few invisible balls were placed so that the distance from the ball to a piece of hair was shorter than the distance to other parts of the head. An illustration of this approach can be seen in Figure 26, which shows a cross-section of the gem.

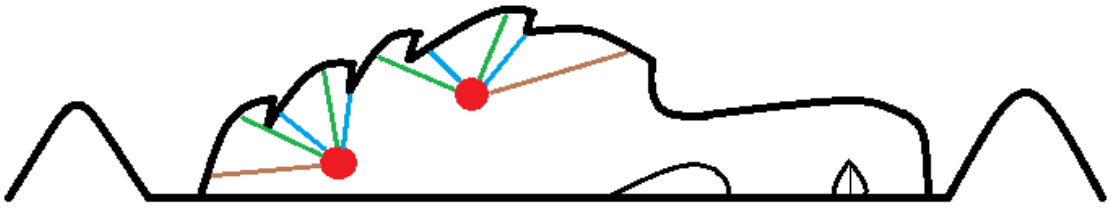

Figure 26. Cross-section with metaballs to color the hair.

When the distance from a point of the model's surface is smaller (blue lines in the figure), the hair was colored the same dark red as the background. Then as the distance got longer, the color linearly faded to white, becoming completely white at a fixed distance point (green lines in the figure). Points even further than that, shown with brown lines in Figure 26, were also set to white.

In total, three such balls were placed under the surface so that most of the hair is in the distance range of at least one ball. When a point was in the distance range of two different balls, the smallest distance was chosen for color calculation. Figure 27 shows the result of this approach, which is also the finished version of the multilayer material.

Figure 27. Multilayer material.

4. Usability Testing

To best assess the quality of the new features added to GPG, a test had to be conducted for that purpose. The best-case scenario for testing the improvements would have been to set it up in the exhibition and ask the visitors to rate the various features. Since they would have been right in the context of glyptic art, their feedback on whether the gem looked more realistic with the additions would have been the most useful.

Unfortunately, the exhibition closed before the testing could be seen through, so as an alternative, it was chosen that the testing was going to be performed with the museum's staff instead. The museum regularly asks the staff's opinion on other exhibits as well, so they already had the experience of evaluating new art pieces.

4.1 Test Setup

The testing was performed on three different aspects of GPG: the subsurface scattering filter, the procedural marble texture, and the multilayer material texture. A questionnaire was created (see Appendix) in which the testers were asked to first rate the quality of the aspects and then, for each one, write any comments that they might have. The questionnaire was created in Estonian because most of the museum's employees who took part in the testing only spoke Estonian.

In addition to the questionnaire, a user guide was created for GPG, also available in Appendix. It detailed all the available features and gave step-by-step instructions on how to use each of them.

To run GPG, an HP x2 210 G2 laptop was provided by the museum, which was the same model that was used in the original exhibition.

Table 3 shows the laptop's specifications, taken from HP's website[19]. The GPU in the laptop was not the fastest, however thanks to the relatively fast SSS, GPG still ran around 30 frames per second with the filter enabled. The touch-sensitive screen allowed the testers to interact with the application without having to use the touchpad.

[19] https://support.hp.com/us-en/product/hp-x2-210-g2-detachable-pc/11572362/document/c05235377

Table 3. HP x2 210 G2 specifications.

Component	**Details**
CPU	Intel Atom x5 Z8350, 1.44GHz, 4 cores
GPU	Intel HD Graphics 400
Display	10.1 in, 1280 x 800 touchscreen
Memory	4GB
Other	Webcam, trackpad

At the start of the test, the testers were asked to sit behind the test setup (shown in Figure 28) and to read through a short introduction in the questionnaire. It gave an overview of what GPG is, what improvements were made, and what subsurface scattering is. The latter was explained to help the testers more accurately assess the SSS filter's effect.

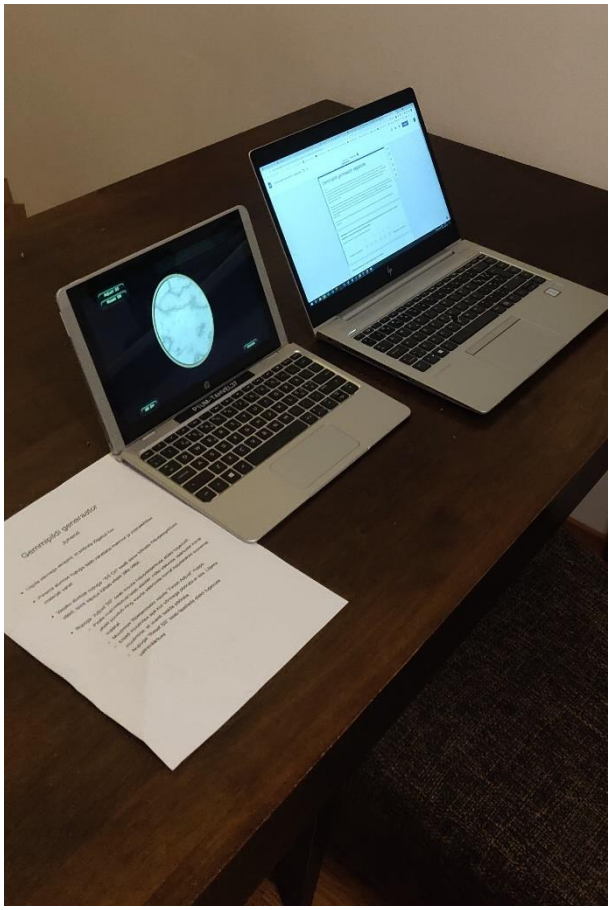

Figure 28. Testing setup. Left: user guide; Middle: GPG; Right: questionnaire

After reading the introduction, the tester was asked to get acquainted with the different features of GPG with the help of the user guide. The tester was also given help in case they had questions about a feature or needed some other help.

The main part of the testing took place so that the tester first read a question in the questionnaire and then used the specific feature in a way described in the question. Then they rated the appearance or other aspect on a scale from 0 to 5. The lowest value was chosen to be zero instead of one because this

way, the testers could not choose the neutral answer in the middle and had to pick a specific side.

They also had the opportunity to write some comments if they had any. The written feedback was needed because the rating alone would not be enough to make meaningful improvements.

During all the steps, notes were also taken about how the testers used the application, how quickly they learned to use it and what feedback they gave besides what they wrote in the comment boxes.

4.2 Results

The first thing that the testers experienced when using the application was its user interface. Rotating the model with the touch-sensitive screen worked quite well, however, in other aspects, the touching proved to be a bit difficult. The buttons were a bit small and did not have visual feedback on whether they were pressed or not. Combining both of these resulted in testers pressing buttons multiple times because they did not know if the button press was successful. In addition to the buttons, the slider often did not move when sliding on it with a finger.

Because of these problems with the UI, the testers were instructed to use the touchpad for interacting with it instead of the touchscreen. After this, the testers had fewer problems with clicking the buttons and using the slider.

The testers also reported that the button that turned the SSS filter on and off was mislabeled. When the filter was disabled, the button's label was "SS On" to indicate that the next press would turn the filter on and when the filter was enabled then the button's label was "SS Off". This, however, confused the testers, who mostly assumed that the button's label told them in which state the filter currently is.

4.2.1 Subsurface Scattering

The testers were asked to select the marble material, toggle the SSS filter on and off, and rate how much the filter made the marble material look more realistic. The rating scale went from 0, meaning that without the filter, it looked better, and 5, meaning that the filter made it seem more like actual marble. Figure 29 shows the testers' responses to this question.

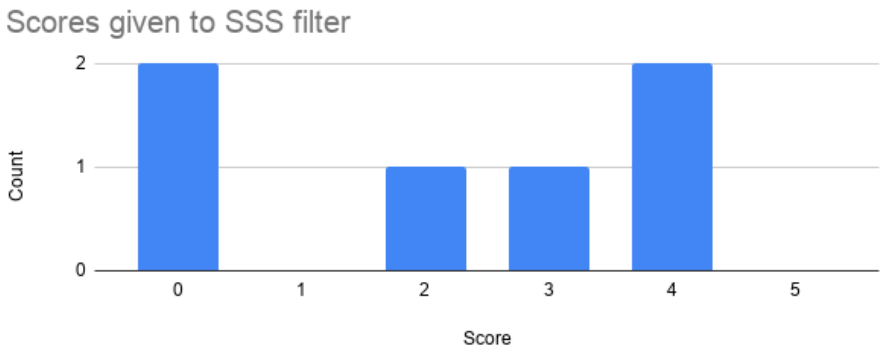

Figure 29. Scores given to SSS filter.

From the figure, half of the testers think that the filter makes the marble look better, and the other half thinks otherwise. The ones that rated the effect negatively wrote that it made the gem look too hazy and commented how the details get lost too much and because of that, the gem does not look very realistic. One tester that rated the effect positively was of the opinion that the filter does make it look more like real marble. They also mentioned that the lesser shininess compared to the gem without the filter, makes the effect less natural than it otherwise would be.

From the feedback, it was clear that SSS needs to be improved so that more people using GPG would agree that the filter makes the marble look better.

4.2.2 Procedural Textures

Next, the testers were asked to turn off the SSS filter and assess whether the procedural marble texture looks like real marble or not, and the result can be seen in Figure 30. In general, the response was quite positive. One tester gave it a slightly negative score because they questioned whether an artist would use a piece of marble with that exact texture in real life. Others said that the marble looked realistic and liked its look, but one suggested that the lines on the marble could be a little bit softer and lighter.

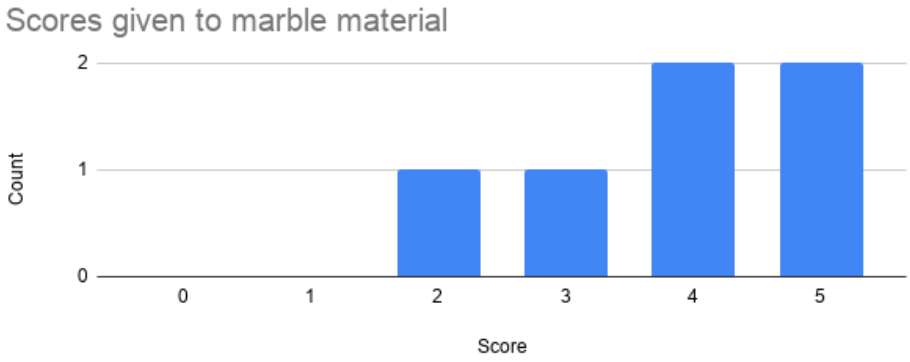

Figure 30. How much the procedural marble texture looks like real marble according to the testers.

Finally, there was a question about the multilayer agate material. To give an example of how materials with multiple layers were used in real life, there was an image of such a gem under the question. The testers then had to again give a rating whether the rendered gem seems to be made from different layers or are the layers not apparent. Figure 31 shows the given ratings to that question.

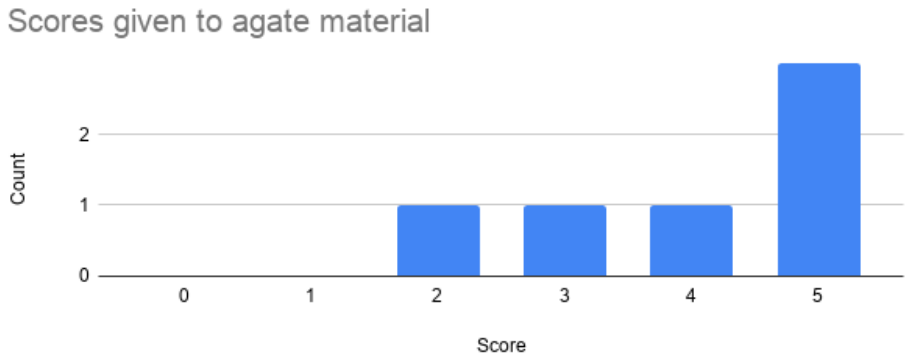

Figure 31. Tester's ratings to the agate material.

The testers liked the multilayer material quite a lot. During the first test, there was a visual bug in its fragment shader, where the gap between the face and the head was still visible (as in Figure 24). Because of this, the first tester gave the material a score of two and said that this distracted them from experiencing the multilayer effect properly. The bug was fixed for the other testers, which all rated the visual look positively, with half of the testers giving it a maximum score. The ones that gave the material a high rating commented that the multilayer effect is clearly visible and that there are three distinct layers visible. This meant, that the efforts to create a multilayer material was successful.

4.3 Improvements

Since there was some negative feedback, mostly about the SSS filter and some about the UI, these aspects of GPG were improved after the testing.

4.3.1 User Interface

The main complaints about the UI were that the interactable elements were hard to press accurately when using the touchscreen and gave no feedback whether they were pressed or not. There is, however, little that can be done about that. The slider and the buttons are provided by Ogre's OgreBites module, and changing their behavior is not possible.

When the testers tried to use the slider, its height was too small to accurately select the sliding dot with a finger on the touchscreen, and they had to try sliding it multiple times before it moved. The buttons had a similar issue where touching them with a finger did not register the press sometimes because the touch missed the button.

The problems with both the buttons and the slider could possibly be fixed if they were made taller. When creating the UI elements then their width can be controlled with a parameter. However, their height is fixed, so stretching them very wide does not give the desired effect of making them more usable.

The only issue with the UI that was fixed in the end was the toggle button for the SSS filter, which was confusing to the testers. It now has a label "SS On" when the filter is turned on and vice versa, which the testers expected when using the button.

4.3.2 Separable Subsurface Scattering Filter

Since most of the testers mentioned that the filter made the marble look too blurry and hazy, some additional work had to be done regarding that problem.

First, the depth correction mentioned in Chapter 3.2.2 was modified to interpolate back more aggressively to the pixel's original color instead of blurring it when the sampled neighboring pixel was further away from the current pixel. This did not improve the result that much, so the version of the depth correction created by the method's authors was sufficient for the task.

Jimenez, J. et al. mention in their work that when using this screen space approach, the rendered image cannot be blurred as a whole. Instead, the diffuse and specular components must be separated, and the blurring must be done on the diffuse layer only. After that, the

two components can be combined back together to produce the final result [6]. However, the approach initially used in GPG blurred both the diffuse and specular components. This is most likely why the testers mentioned that the image looks too blurry because the specular reflection was blurred when it should not have been.

To fix this problem, the fragment shaders of all the materials were changed to not include the specular component in their color output. The specular calculation was moved to a new render pass in the compositor to render the specular reflections to a separate texture. The texture was then combined with the blurred diffuse layer during the second, horizontal blur pass. The modified compositor pipeline can be seen in Figure 32.

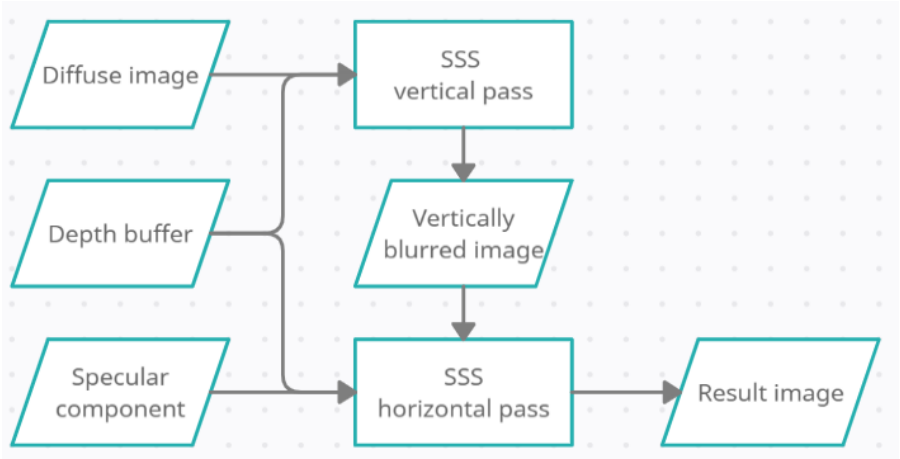

Figure 32. Improved pipeline of the SSS compositor.

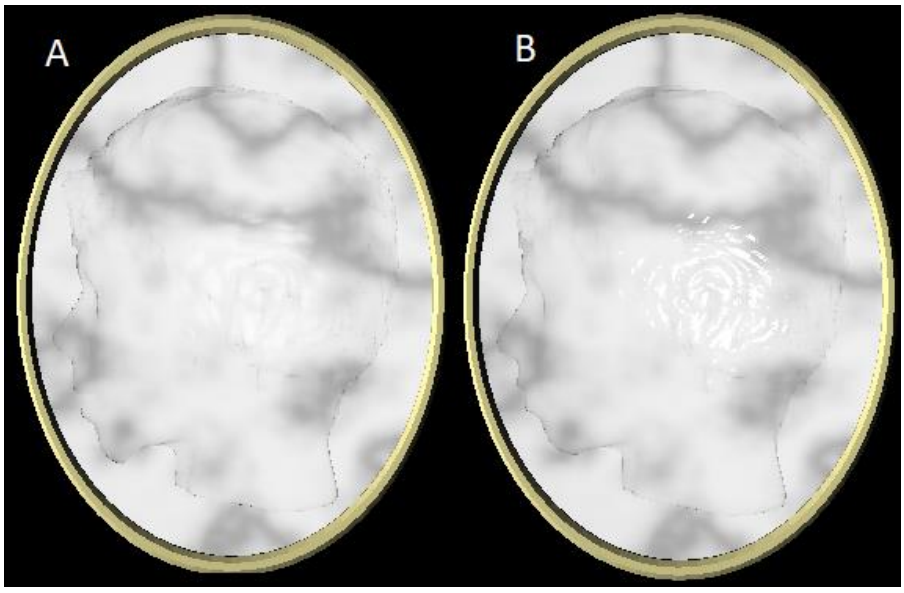

Figure 33. Gem with the original SSS filter (A); Gem with the fixed SSS filter (B)

With the new fixed pipeline, the SSS effect now worked like Jimenez, J et al. [6] described – only the diffuse layer is blurred with a Gaussian filter and the specular reflections are untouched. Figure 33 shows the original version of the filter on the left, where the reflections are practically invisible due to the blurring. As a contrast, the reflections are clear and very visible on the right. With the new version, the haziness that the testers reported is thus eliminated, which means that the materials should now look more realistic with the filter than without it. A more detailed comparison of the old and fixed versions is available on YouTube[20].

[20] https://www.youtube.com/watch?v=wYLjnx4BEBY

5. Conclusion

In this thesis, the Glyptics Portrait Generator was improved to display materials that more closely resemble those used to create glyptic art in real life. The Separable Subsurface Scattering method was implemented to simulate subsurface light transport. It functioned as a post-processing filter that blurred the diffuse layer of the original rendered image in two separate passes and combined it with the specular component to produce the result.

Two new marble textures were created for the engraved gem. The first was marble, where the color value at each point was calculated procedurally using the world coordinates. The texture was created with a repeating vertical line pattern which was displaced using Simplex noise. The second was a multilayer material, where the gem background was a single color modulated with Simplex noise. The edges of the head were blended in with the background with a colormap function. Finally, the hair was colored using an approach with metaballs.

The new features were user-tested with the staff of the University of Tartu Art Museum. Their response was generally positive, especially regarding the new materials. However, some shortcomings were also noticed, and some of them were also fixed during this thesis.

In future work, face morphing could be improved. Currently, it does not exactly align with the rest of the head, so there are holes in some places in between them and visible intersections in other places. Additionally, the gem model could be specifically modelled to look like a real cameo, making coloring it simpler, and the result would look better. If the model is redesigned with a higher polygon count, then the specular reflections would look smoother as an additional benefit.

References

[1] "Glyptics: Past and Present.," [Online]. Available: https://www.hermitagemuseum.org/wps/portal/hermitage/what-s-on/temp_exh/2019/glyptics/. [Accessed 09 12 2020].

[2] M. Cooper, "A Brief History of Cameo Jewelry and How It's Still Popular Today," 2020. [Online]. Available: https://mymodernmet.com/history-of-cameo-jewelry/. [Accessed 09 12 2020].

[3] V. Kupriienko, Glyptics Portrait Generator. Master's Thesis, Tartu: University of Tartu, Institute of Computer Science, 2020.

[4] Pluralslight, "Understanding Subsurface Scattering - Capturing the Appearance of Translucent Materials," 7 7 2014. [Online]. Available: https://www.pluralsight.com/blog/film-games/understanding-subsurface-scattering-capturing-appearance-translucent-materials. [Accessed 10 12 2020].

[5] M. Pettineo, "An Introduction To Real-Time Subsurface Scattering," 2019. [Online]. Available: https://therealmjp.github.io/posts/sss-intro. [Accessed 01 03 2021].

[6] J. Jimenez et al., "Separable Subsurface Scattering," *Computer Graphics Forum*, vol. 34, no. 6, p. 188–197, 2015.

[7] J. Jimenez and D. Gutierrez, "Screen-Space Subsurface Scattering," in *GPU Pro: Advanced Rendering Techniques*, A.K. Peters, 2010, p. 335–351.

[8] T. Yatagawa et al., "LinSSS: linear decomposition of heterogeneous subsurface scattering for real-time screen-space rendering," *Visual Computer*, vol. 36, no. 10–12, p. 1979–1992, 2020.

[9] A. Lagae, S. Lefebvre, T. DeRose and R. Cook, "A Survey of Procedural Noise Functions," *Computer Graphics Forum*, vol. 29, no. 8, p. 2579–2600, 2010.

Appendix

I. Glossary

Rendering – process of generating an image from a 3D model with a computer program.

Shader – a type of program that tells the computer how to render each pixel.

Volumetric path tracing – a rendering technique, in which the path of many light rays is computed in the 3D scene.

Gaussian filter – A mathematical function[21] that is used to multiply the output of some other function.

Texture space – the coordinate space of a 3D model's 2D texture[22].

Screen space – the coordinate space of a rendered image[22].

Object space – the coordinate space where every point's position is relative to some object.

World space – the coordinate space where every point's position is relative to the world origin at coordinates (0, 0, 0).

Compiler directive – a programming language construct that has a special meaning to the compiler.

Scene – the collection of 3D models and their textures, which is transformed into a 2D image during the rendering process.

Buffer – an intermediate image used in the rendering process.

Vertex – a point in space.

Face – a single flat polygon made up of vertices, usually a triangle. Not to be confused with the face of the head that is engraved on a gem.

Geometry – a collection of vertices and faces that makes up a 3D model.

Colormap – a function which outputs a color based on a numeric input value.

Vertex shader – a program that tells the GPU, how to modify vertices.

---

[21] https://en.wikipedia.org/wiki/Gaussian_function

[22] https://en.wikipedia.org/wiki/Glossary_of_computer_graphics

Fragment shader – a program that tells the GPU, what color is each pixel on the screen.

Metaballs – in the context of this thesis, a set of balls in 3D space, where the sum of distances from each of them to a pixel can be used to calculate a color for said pixel. A more formal definition can be found in Wikipedia[23].

[23] https://en.wikipedia.org/wiki/Metaballs

II. Accompanying Files

The source code of GPG can be found on GitHub[24].

The GPG comes with accompanying files that contain:

1. A distribution of GPG that can be run on a Windows PC.
2. The questionnaire that was presented to the testers.
3. The user guide for GPG.
4. Testers' responses as a CSV file.

Instructions on how to run the GPG distribution can be found in the README.md file in the distribution folder.

[24] https://github.com/lakies/glypticsgenerator

III. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Adrian Kirikal**

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Glyptics Portrait Generator – Improved Materials,

supervised by **Raimond-Hendrik Tunnel.**

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Adrian Kirikal

07/05/2021