UNIVERSITY OF CRETE

COMPUTER SCIENCE DEPARTMENT

Physically-principled character lighting and shading in real-time

BY

Papanikolaou Petros

Master's Thesis

HERAKLION, JUNE 2013

Abstract

In many applications, user interacts in a 3D virtual world rendered with physically-based light. In order to enhance user's experience, the graphics community is continuously working on enriching these worlds with realistic depiction of light interaction with objects. The controversy between accuracy and rendering speed is always constant, especially for dynamic surfaces in order to achieve at least 25 frames per second.

In this thesis, we analyze the physically-principled lighting and shading effects that occur on human skin in an area and point light illuminated 3D space. Some of these effects are the following: Ambient Occlusion, Shadow Mapping, Image-Based Lighting, Specular Surface Reflectance, Subsurface Scattering and Environment Mapping. There are systems that support a combination of them, but only in isolated character parts (e.g. human head) in non-hierarchical and non-animated surfaces. The problems that we are tackling are not only limited on how to apply these effects on a fully, skeleton-based, deformable and animated virtual character, but also on how to calculate the most robust physically-based approximations (hence physically-principled) in order to achieve high visual accuracy along with efficient real-time execution in modern commodity graphics h/w.

We have thus created a high-fidelity, physically-principled real-time rendering framework for articulated characters. It features a series of effects that can be applied on the human model either simultaneously or separated. The unification was implemented in screen space (as opposed to standard object-space methods), achieving that way both scalability and efficient real-time execution. Multiple rendering fragment shader passes are created and handled within our system. The output image rendered by any pass can be used as input to the subsequent pass that follows ensuring fast and efficient reusability between the common and essential data for the different effects. For instance, our real-time subsurface scattering method is achieved in just two rendering passes, applying horizontal and vertical convolutions using an intermediate texture as output to the first pass and input to the next. The implementation of our rendering framework is based on widely available open-source tools and techniques such as OpenGL, C++ and GLSL so that it can be easily integrated in modern rendering engines and scene graphs via commodity graphics h/w.

Περίληψη

Σε πολλές εφαρμογές, ο χρήστης αλληλεπιδρά σε ένα τρισδιάστατο εικονικό κόσμο με προσομοιωμένο σύμφωνα με τους νόμους της φυσικής φωτισμό. Προκειμένου να βελτιωθεί η εμπειρία του χρήστη, η υπάρχουσα έρευνα στην περιοχή των γραφικών βρίσκεται σε μία συνεχή διαδικασία εμπλουτισμού των κόσμων αυτών με ρεαλιστική απεικόνιση της αλληλεπίδρασης του φωτός με τα αντικείμενα. Η διαμάχη μεταξύ ακρίβειας και ταχύτητας εκτέλεσης είναι πάντα παρούσα, ειδικά στις περιπτώσεις που θέλουμε να απεικονίσουμε δυναμικές επιφάνειες 25 φορές το δευτερόλεπτο.

Σε αυτήν την εργασία, αναλύουμε τα φυσικά φαινόμενα φωτισμού και σκίασης που παρατηρούνται στο προσομοιωμένο ανθρώπινο δέρμα ενός εικονικού χαρακτήρα σε ένα τρισδιάστατο χώρο φωτιζόμενο τόσο με σημειακά φώτα όσο και φώτα χώρου. Μερικά από αυτά τα φαινόμενα που το σύστημά μας υλοποιεί περιλαμβάνουν: Ambient Occlusion, Shadow Mapping, Image-Based Lighting, Specular Surface Reflectance, Subsurface Scattering και Environment Mapping. Υπάρχουν συστήματα που υποστηρίζουν συνδυασμό ενός η περισσοτέρων από αυτά, αλλά μόνο σε απομονωμένα τμήματα του εικονικούχαρακτήρα (π.χ. ανθρώπινο κεφάλι) σε μη-ιεραρχικές και στατικές επιφάνειες. Το πρόβλημα που καλούμαστε να λύσουμε είναι όχι μόνο η εκτέλεση των φαινομένων αυτών φωτισμού σε ένα παραμορφώσιμο, κινούμενο, αρθρωτό ανθρώπινο σώμα βασισμένο σε σκελετό, αλλά επίσης να κάνουμε τις σωστές προσεγγίσεις ώστε να πετύχουμε υψηλή ακρίβεια μαζί με πραγματικό χρόνο εκτέλεσης σε μοντέρνες εύκολα προσβάσιμες κάρτες γραφικών.

Έχουμε δημιουργήσει έτσι ένα ολοκληρωμένο σύστημα απεικόνισης αρθρωτών εικονικών χαρακτήρων με υψηλή ακρίβεια, πιστότητα και ικανό να προσεγγίσει τον φυσικό τρόπο φωτισμού σε πραγματικό χρόνο. Περιλαμβάνει μία σειρά από φαινόμενα, τα οποία μπορούν να εφαρμοστούν σε έναν εικονικό ανθρώπινο μοντέλο είτε ταυτόχρονα είτε ξεχωριστά. Η ένωση αυτών υλοποιήθηκε σε Χώρο Εικόνας (Screen Space) (σε αντίθεση με το χώρο αντικειμένου), επιτυγχάνοντας έτσι επεκτασιμότητα και αποτελεσματική εκτέλεση σε πραγματικό χρόνο. Πολλαπλά στάδια απεικόνισης δημιουργούνται και διαχειρίζονται εντός του συστήματός μας. Η παραγόμενη εικόνα του κάθε σταδίου μπορεί να χρησιμοποιηθεί ως είσοδος σε κάποιο επακόλουθο στάδιο, σιγουρεύοντας έτσι γρήγορη και αποτελεσματική επαναχρησιμοποίηση σε κοινά δεδομένα των διαφορετικών φαινομένων. Για παράδειγμα, το φαινόμενα διασποράς υποεπιφάνειας (subsurface scattering), που είναι εκτελέσιμο σε πραγματικό χρόνο, επιτυγχάνεται με τη χρήση μόνο δύο σταδίων απεικόνισης, εφαρμόζοντας οριζόντια και κάθετη συνέλιξη με τη χρήση μιας ενδιάμεσης εικόνας ως έξοδο από το πρώτο στάδιο και είσοδο στο δεύτερο. Η υλοποίηση όλου του συστήματός μας είναι βασισμένη σε ευρέως διαθέσιμα πρότυπα και ανοιχτό λογισμικό, όπως OpenGL, C++ και GLSL, έτσι ώστε να μπορεί να ενσωματωθεί σε μοντέρνες μηχανές γραφικών και γράφους σκηνής με τη χρήση εύκολα προσβάσιμων καρτών γραφικών.

Contents

List of Figures

List of Tables

1 State of the Art

1.1 Background

Twenty five years ago, the procedure of creating virtual human characters using only computers was introduced (3D Modeling) in an offline manner. The difficulty in this procedure can be encountered not only in the depiction of the complicated human characteristics, but also in the creation of animation. Nevertheless, since then, a lot of tools have been created, which allow for automation in this procedure. Tools like 3DSMax, Maya or Blender continuously are being evolved to cover this need. The last few years, a significant part of computer graphics community is taking action in order to achieve the depiction on the screen of the most challenging effects that constitute our world in real time. The most demanding part in this is the application of these effects in virtual human body and face. Due to complexity of the real human skin, a lot of effort has to be made to achieve realistic results in simulating virtual counterparts. Every year in major computer graphics conferences, a lot of work is presented based on advances in real-time rendering in 3D graphics and games (e.g. [1] and [2]).

1.2 Lighting and Shading

As far as the interaction between the light source and the object is concerned, one of the most used methods to light an object is the Blinn-Phong illumination model [3]. It is a method based on Phong illumination model [4]. The difference between these 2 methods is that Blinn-Phong doesn't have to recalculate the reflected vector of the light to the surface over and over again. Instead of that, it uses the halfway vector between viewer and the light source. The halfway vector is calculated by normalizing the sum of View vector and Light Vector (Figure 1). Then, we calculate the amount of specular reflection using the object's shininess, which depends on the material.

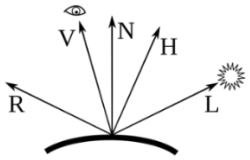

Figure 1: Vectors in Blinn-Phong[1]

But how is the normal obtained in order to compute each pixel's color? The most common method, is Phong shading [4], which improves Gouraud shading [5]. Both are interpolation techniques but they differ in the entity they interpolate. Gouraud Shading method uses the normal of each vertex to compute the color of that vertex and then interpolates the resulting color for each pixel covered by the triangle. On the other hand, Phong Shading technique first interpolates the normal of the vertex for each pixel and then computes the pixel's color separately based on the

[1] Image: http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model

interpolated normal. Everyone agrees with the fact that Phong Shading works much better and even if it is a bit slower, it is preferred over Gouraud's method.

Although, Blinn-Phong illumination model is an algorithm first used in 1975, it is still highly used. Even though it's easily implemented, it produces plausible results. Another advantage of using this algorithm is the fact that it can be fully implemented in fragment shader, while GPU interpolates the normals across the surface.

1.3 Shadowing

Before start talking about shadowing the scene with a virtual character, we have to consider the type of the light sources we are going to use. There are three basic types of lights that determine the behavior of the shadowing assuming that the light is depicted as a point. The first one is the Directional light, which simulates the behavior of the sun and is the easiest to compute. We don't have to specify position for this light because it is considered to be infinitely far away. The emitted rays are parallel and they are not affected by distance attenuation. The second type is the Point light, which (as the name explains) is just a point emitting in all directions (Omni Light). It can be thought as the light bulbs we use in our rooms. The third and the most complicated type of light is the spot light. They have position and they emit light in a specific direction. The closer the lighted point is to the central ray, the brighter it becomes. The area illuminated by the Spot light looks like a cone, the base of the cone being the far plane.

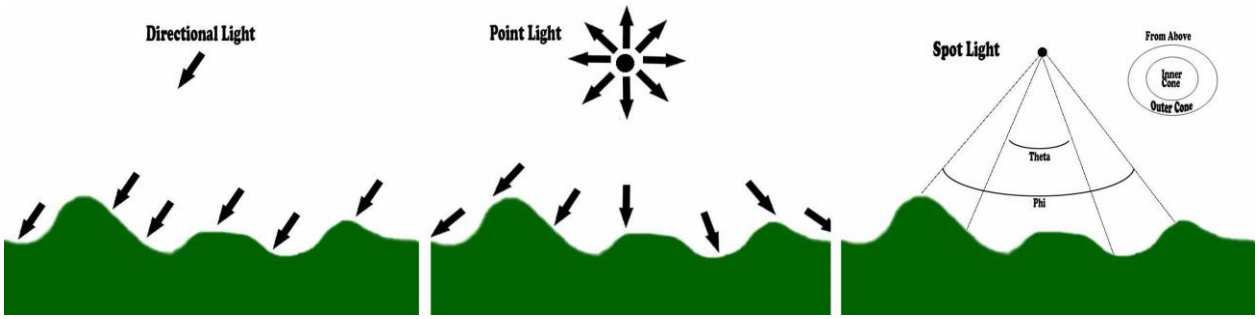

Figure 2: Light Types[2]

The most preferred technique that is used to shadow the scene is called shadow mapping [6], which is a 2-pass algorithm. In the first pass we store the light's depth buffer in a texture using its view and projection matrix. The depth texture is usually in higher resolution than the application's viewport. In the second pass we multiply each point visible from user's camera, with the light's view and projection matrix. According to the depth difference between the texel's depth value and the light's depth value obtained by the user camera, we can determine whether or not a point is shadowed.

[2] Image: http://www.toymaker.info/Games/html/lighting.html

Shadow Mapping can cause a lot of artifacts during real-time rendering. That's why there is a wide range of implementations to choose based on each application needs. Algorithms like Parallel-split shadow maps (PSSMs [7] & [8]) and Cascaded Shadow Mapping (CSM [9]) tend to split the view frustum into multiple layers storing each one of them in a different depth buffer [10] (in some cases with different resolution). Then, depending on the point's position relatively with the light, the corresponding buffer is picked in order to check for visibility. This group of techniques is used to produce shadows for large scale environments. Another technique that is commonly used is called Percentage-Closer Filtering [11], which is partially handled by the GPU. It is used to solve aliasing problems, which caused by illuminating an area almost parallel to the emitting rays. In this technique, we use samples around the shadowed pixel to calculate a shadow percentage.

Figure 3: PSSM (left)[3] & CSM (right)

1.4 Ambient Occlusion

In the real world, light tends to reach surfaces from all directions. At day, even if there aren't any light bulbs turned on around, our room is still enlightened. To achieve highly realistic scenes, we have to simulate global illumination without the use of any light source. In computer graphics, this natural phenomenon is encapsulated in ambient occlusion, which approximates the way indirect lighting works. Ambient occlusion is related only by neighboring geometries, which makes it an effect totally independent by viewer's position relatively with the object.

[3] PSSM Image: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch10.html

Figure 4: Ambient Occlusion in Uncharted 2 game[4]

Ambient Occlusion can be used either for static or dynamic scenes. The most accurate results are produced by ray-tracing techniques. The problem with these techniques is that they run in CPU and also they are so slow that makes them unable to be used in real time applications. The only exception is in case we have a static scene and we can afford to run a ray tracing algorithm to calculate the unmodified ambient occlusion for our scene. We can also obtain this information by reading textures, which can be created either by hand (on a paint tool) or by precomputing the ambient occlusion and storing it in these textures [12]. But, if someone uses deformable skeletal-based animated models, an alternative solution will be needed for the production of ambient occlusion.

In [13] the state of the art in ambient occlusion techniques is described. We will focus in those that not only can be used in dynamic scenes, but also they can be supported in real time applications. Bunnell's approach [14] gives efficient results for animated characters with the cost of performance as the number of passes for higher quality increases. The techniques mentioned in [15] and [16] precompute the ambient occlusion for several poses of the character. As the animation runs, an interpolation method decides for the final occlusion. We continue the discussion by focusing on Screen Space implementations. This category of algorithms is the most commonly used, since they run fully in GPUs, giving great results in speed issues. Even though they lack of neighboring geometry information, they increase the realism of the scene. When compared with ray tracing algorithms, users can hardly distinguish the difference. Screen Space algorithms don't need any preprocessing and also they are independent from scene's complexity.

[4] Image: http://fallenexile.blogspot.gr/2011/01/hut-interior-rough-modelling-wireframe_21.html

The next series of algorithms are based on a technique used in Crysis 2 [17], which first introduced the comparison of the depth buffer for the Ambient Occlusion production. Each pixel's occlusion value is depended on how many neighbor pixels are closer to camera (have less depth). A very influential technique is described in [18]. According to this, each surface presented by a pixel, will be reconstructed as a sphere in world space. These spheres will be used to calculate the subtended surface area. Another group of techniques uses a normal-oriented hemisphere with its center place on each pixel. Samples, whose depth values will be compared, have to belong inside hemisphere's volume. The method described in [19] is based on this approach. Initially, we compute the view space position of each pixel. Then, we pick a number of vector offsets and we add them separately to main pixel's view space position, resulting in many different positions in the same space. After that, we create the set of pixel samples by projecting each one these positions to screen space. For artifacts to be dealt, we have to either choose different sample offsets for each pixel or we can rotate the hemisphere containing the samples. A similar approach, which also uses hemisphere, but works directly in screen space is discussed in [20]. Instead of choosing samples, a number of directions are picked for each pixel in image space. For each direction, the horizon angle is computing by sampling the depth of pixels along the line. The higher is the horizon angle, the more it contributes to ambient occlusion.

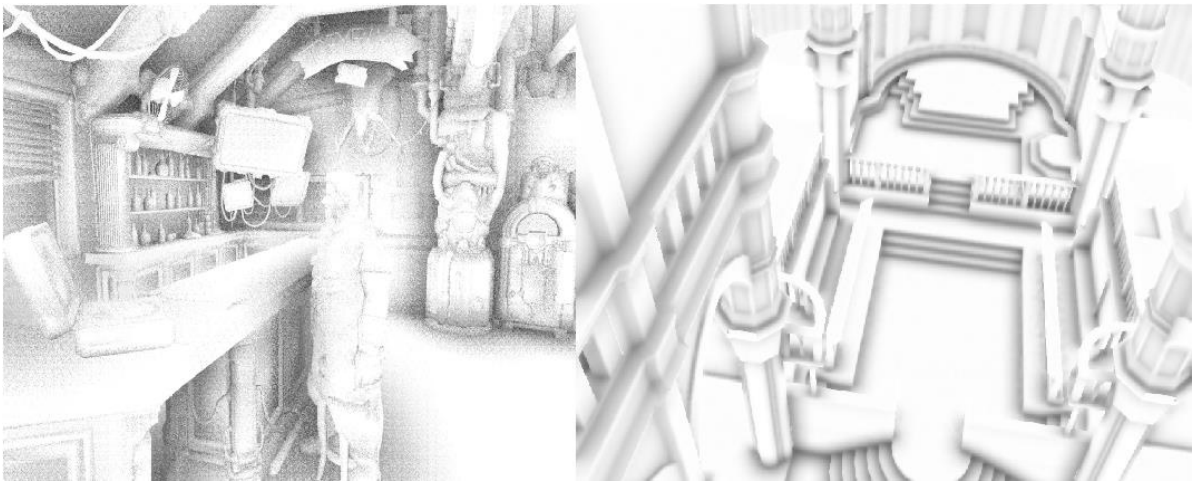

Figure 5: Starcraft AO (left) & Horizon-Based AO(right)

The previous algorithms we have discussed are applied to a single resolution. In [21], the use of multiple image resolutions is introduced. This method computes ambient occlusion in multiple resolutions and combines them to obtain the final result. Another difference to the previous screen space approaches is that it doesn't have to use random sampling. All it needs is a small sampling kernel at every resolution. This technique first renders the scene to a g-buffer at the highest resolution. The g-buffer is then downsampled multiple times. The ambient occlusion is calculated for each pixel in each resolution by sampling the neighboring pixels. Finally, all resolutions take part in the calculation of each pixel's ambient occlusion. Before combined, each resolution other than the highest has to pass through bilateral upsampling.

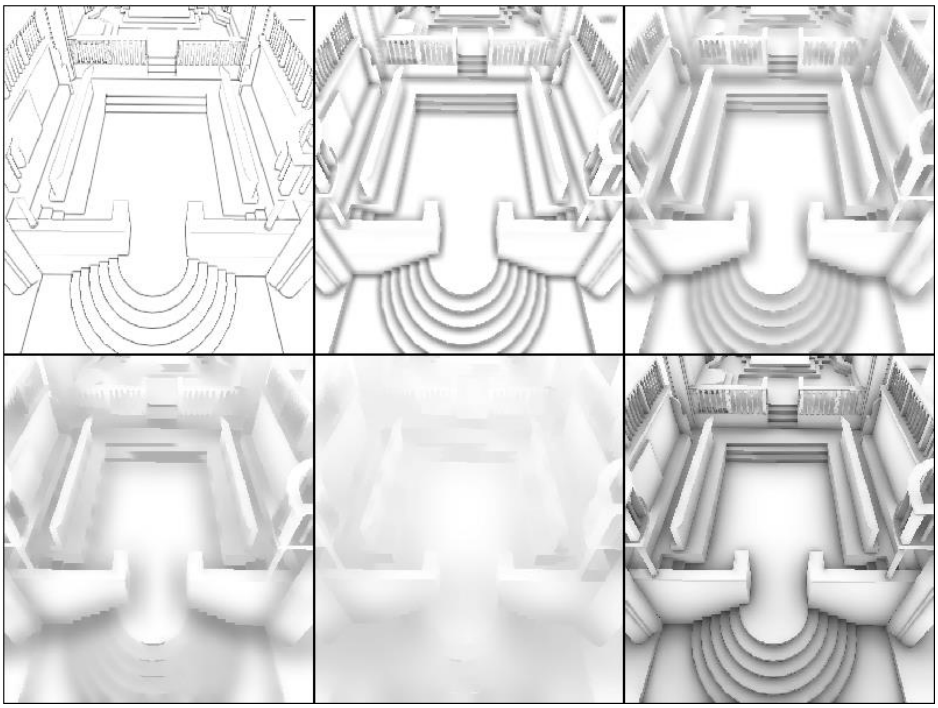

Figure 6: Results of Multi-Resolution Screen-Space Ambient Occlusion technique. The first 5 images depict the Ambient Occlusion values computed at 5 different resolutions. The sixth image is the final result.

Last but not least, the volumetric methods produce results as good as the previous methods mentioned. According to volumetric approaches, the ambient occlusion is calculated using the blocked volume in a sphere around the sample. In Volumetric Obscurance [22], instead of sampling points and comparing their depths, line samples are used and the ambient occlusion is computed by taking into account the visible portion of each line. In this category also belong the techniques Volumetric Ambient Occlusion [23] and Volumetric Ambient Occlusion for Volumetric Models [24]. They work based on how big portion of the tangent sphere of the surface belongs to the set of occluded points.

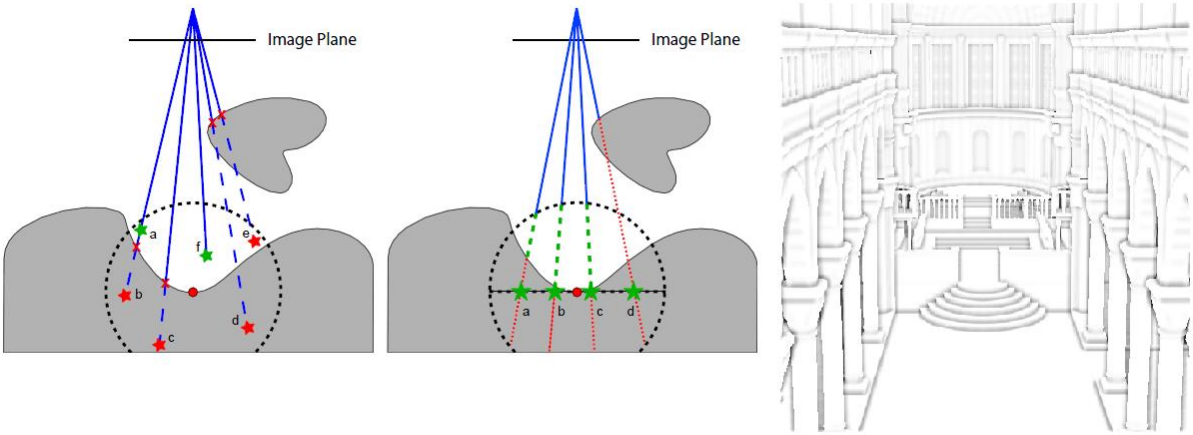

Figure 7: Screen-Space AO samples points (left), Volumetric Obscurance samples lines (middle), Volumetric Obscurance result (right)

1.5 Virtual Skin Rendering

Before we mention the state of the art on skin rendering, we have to discuss about translucency and transparency. Transparent are the objects that have the physical property of allowing the light to pass through without being scattered. Translucent are the objects, which allow the light to pass through, but partially. If we look through a transparent object we can see what's on the other side, but if we do the same using a translucent object, we will only see the glowness. The opposite of transparency/translucency is the opaqueness. Focusing on translucency, some objects that have this property are tree leaves, marble, wax or skin. Subsurface scattering is the phenomenon according to which the light enters a surface, scatters several times and then exits at a different point. The number of times the light will be scattered is depended on the material. In order for skin realistic effects to be created, this effect has to be simulated.

Figure 8: Transparent glass[5], translucent wax[6] and translucent tree leaves[7]

Eugene d'Eon and David Luebke have made an important contribution in [25] explaining all the aspects of light's interaction with skin. The skin in general and face in particular, is one of the hardest materials to render as far as realism is concerned. A face has a set of easily distinguishable characteristics, which consists of wrinkles, pores, freckles, scars and so on. Trying to work only with these, the skin will seem dull, like an object on the table, without inner complexity (under skin). What makes the skin differ, is the way it interacts with lighting, which happens through reflectance. The 2 types of skin reflectance are surface reflectance and subsurface reflectance. According to [26] only 6% of the incident light reflects directly. This happens due to the topmost oily layer of human skin [27]. An accurate method to implement surface reflectance is using Kelemen/Szirmay-Kalos [28] model, along with Fresnel Reflectance and Beckmann microfacet distribution function.

The other type of reflectance, which affects skin, is the subsurface reflectance. This is the most complex effect that occurs to skin. It describes the light's behavior after its entrance to subsurface layers. One portion of the light will be absorbed and another will be scattered many

[5] image: http://ilhammoutaharik.files.wordpress.com/2012/03/verre03.jpg
[6] Image: http://nutrivize.com/blog/wp-content/uploads/2012/05/commons.jpg
[7] image: http://img835.imageshack.us/img835/9946/candless.jpg

times before it exits from a neighboring area. If the skin area is thin, the light will be able to pass through. One of the characteristics that make skin hard to render, is the fact that it consists of many layers with different properties each. The more layers we take into account, the more realistic results our system will produce [29]. We have to use an appropriate *Diffusion Profile*, which approximates the light's attenuation while propagating. Each color has its own diffusion profile. In Figure 9, we can observe light's behavior by striking a surface with a laser beam. As we can notice green and blue are affected by a stronger attenuation than red (which is the color of blood).

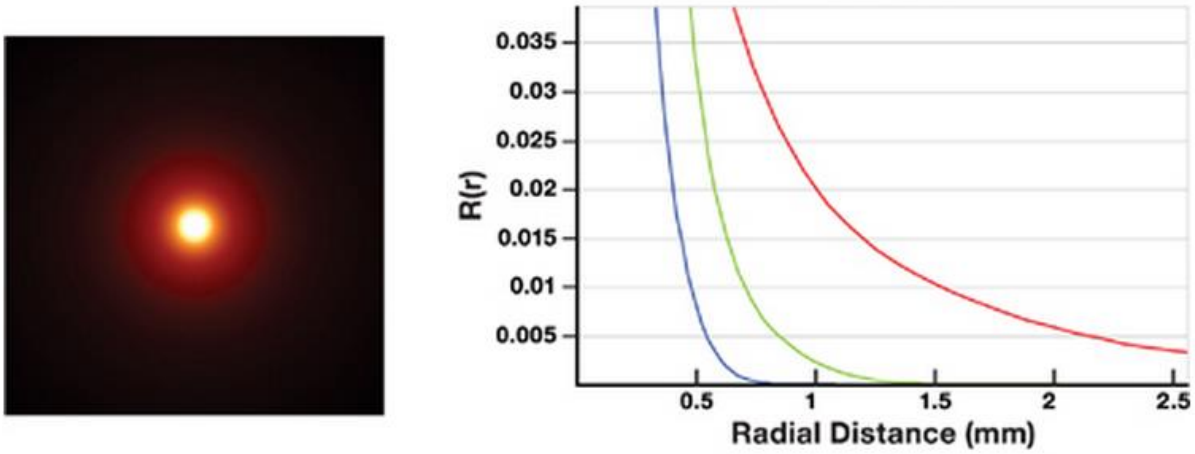

Figure 9: Light's attenuation after hitting a surface with a light ray

A way to express light's transmission is to use a sum of six Gaussians with different parameters in each one. Even though the 6 circular areas described by Gaussians are concentric, they don't use the same radius. The reason why we need different weights for red, green and blue, is because the 3 wavelengths don't attenuate with the same way. The parameters that are used for the Gaussians can be seen below. Two are basically the methods we use to simulate the subsurface scattering effect: Texture Space and Screen Space.

	Variance (mm^2)	Blur Weights Red	Green	Blue
	0.0064	0.233	0.455	0.649
	0.0484	0.100	0.336	0.344
	0.187	0.118	0.198	0
	0.567	0.113	0.007	0.007
	1.99	0.358	0.004	0
	7.41	0.078	0	0

Figure 10: Gaussian parameters

The Texture Space Diffusion for offline, virtual character skin rendering was first introduced in The Matrix Reloaded [30]. The basic idea is to render the geometry's irradiance in a texture. The scattering of light is implemented by blurring this texture. As we've mentioned before, an accurate method to express light's diffusion is to use 6 Gaussians. Embedding this method to Texture Space Diffusion, we need to blur the irradiance texture 6 times separably. The blurring will be applied according to a Gaussian function using the different parameters in Figure 10. Horizontal texels will be blurred apart from the vertical ones. An important thing to mention here is that the horizontal and the vertical length has to be the same for the Gaussian convolution to be correct. Before we render the initial irradiance texture, a correction for UV distortion is probably necessary. So, starting from the same irradiance texture, in the end we will have 6 different blurred textures. The final result (Figure 11) will be the linear combination of these textures. A huge problem of Texture Space Diffusion implementation for subsurface scattering is that it doesn't scale well in a scene with multiple virtual characters.

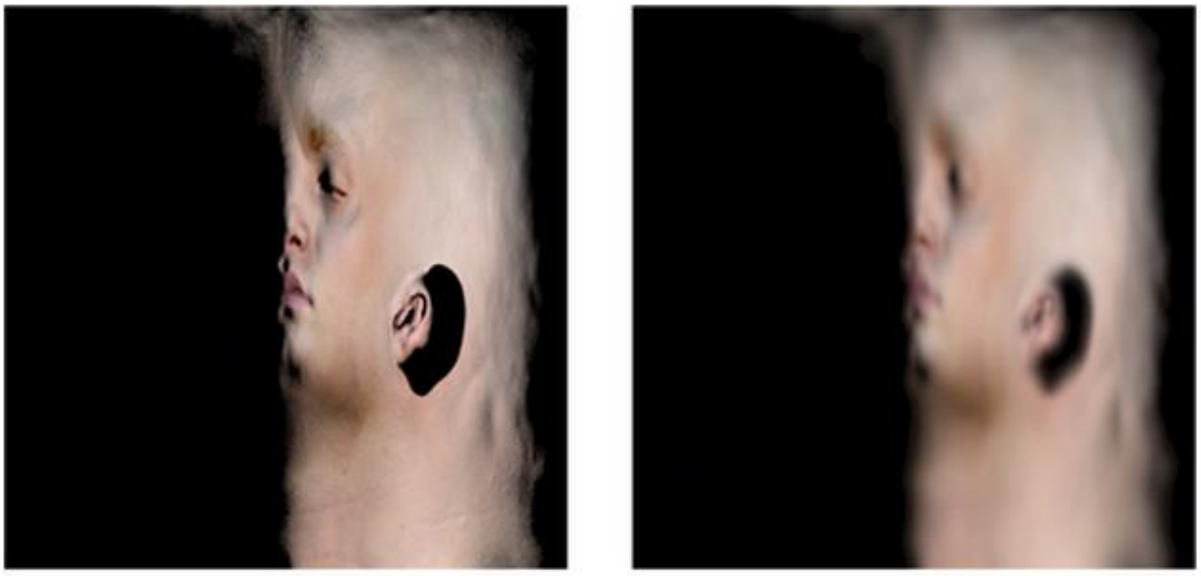

Figure 11: Irradiance texture (left) and irradiance texture after applying a gaussian convolution.

The second method to simulate the subsurface effect is described in [31], which is based on [32] and is implemented in Screen Space. This method increases the rendering performance, compared to Texture Space Diffusion, but with the cost of accuracy. The basic difference is that it doesn't need 5 more separable passes to blur the irradiance texture. Instead of that, it uses a bidimensional convolution on the final result as introduced in [32]. Another difference is that it scales a lot better in scenes with more objects. Textures Space Diffusion on the other hand has to render the irradiance texture and blur it once for each object. As we have already mentioned for Screen Space techniques, they are view depended, losing information about scene's geometry. In this case, the blur is not applied correctly for high-curvature features. In Figure 12 we can see the difference in convolution between the Texture Space and Screen Space.

Figure 12: Convolution comparison in Texture Space Diffusion (upper) and Screen Space (bottom)

Simulating the subsurface scattering effect with convolution is enough for thick surfaces. But, if we want to apply this effect in thin surfaces, we have to extend the methods discussed above. The difference between thin and thick surfaces is that if light ray hits a thin surface, it might pass through the skin and be emitted from the other side. Convolution can't simulate this effect. To deal this problem in Texture Space Diffusion, we use a solution which is based on Translucent Shadow Mapping [33]. While light renders the shadow map, instead of storing only the depth, we also have to store the UV coordinates for the irradiance texture for each pixel rendered. This information will be used later when the user's camera will look the opposite side of the thin surface. Knowing the irradiance and the depth the light traveled, we can approximately compute light's emittance power. The problem with this method is that it cannot be used in Jimenez' Screen Space technique for subsurface scattering, because it doesn't use irradiance textures. Also, in Screen Space we don't know the geometry of the opposite side of the surface the user is watching. Jimenez explains in [34], which is based on [35], that the normals of the pixels suffice to calculate light's transmittance. Assumes that the normal of the opposite side, is the opposite of each pixel's normal (No = -Np). Using this assumption, he calculates the irradiance on the other side of what user is watching.

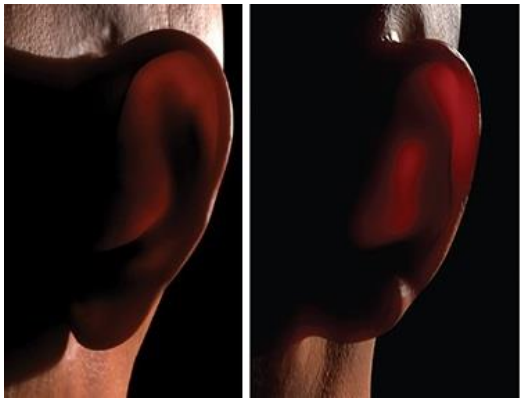

Figure 13: Translucency in thin surface for Texture Space Diffusion (left) and Screen Space (right)

1.6 Environment Mapping

Environment Mapping is a popular Image-Based rendering Technique which is extensively explained in [36]. We use it to simulate the space in which our object exists and being influenced by its surroundings. This technique assumes that the object's surroundings are infinitely far away and they can be depicted as an omnidirectional image known as Environment map. The reason why we use this method is to approximate in real time the way the environment affects the object, without any ray tracing algorithm. Ideally, each object in the scene should have its own environment map, but in practice the objects share environment maps without affecting the visual results.

All recent GPUs support cube map, which is a type of texture used for the environment's depiction as an omnidirectional image. It consists of 6 images, which are placed (sometimes virtually) around the object forming a cube. The most common use of environment mapping is to create reflective objects. It is implemented using the incident and reflected rays of a point. We calculate the reflected vector by taking into account the vertex' normal and the "viewer to vertex" vector. Then, we use the part of texture cube map, which intersects with the reflected vector.

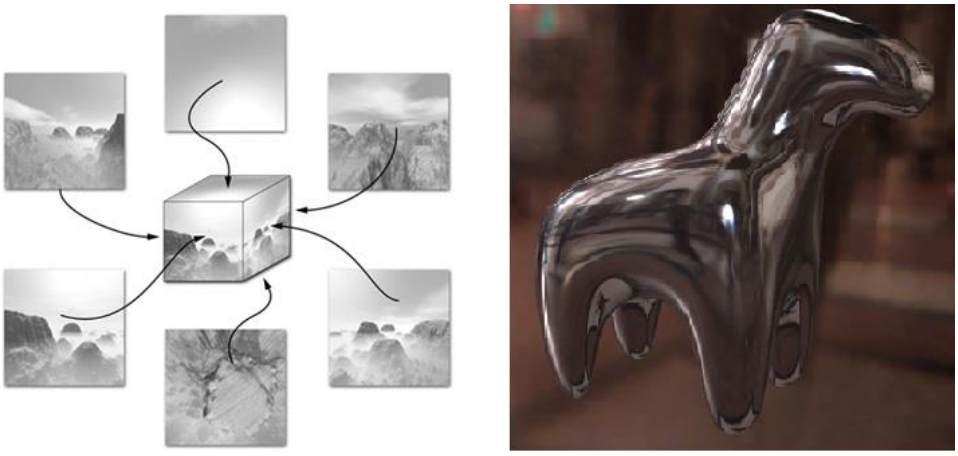

Figure 14: Texture Cube map (left), reflective object (right)

Texture Cube Maps are also used for a technique called Image Based Lighting (IBL), which is an approximation of the light diffusion in the environment. We can use it to simulate a room exposed to lighting with a virtual character inside it lit with natural light. Debevec [37] has done an amazing research calculating the portion of light that is diffused in a surrounding space. Using light probes in different environments, he created omnidirectional High Dynamic Range Images. Based on his algorithm, we can calculate the light's diffusion in a room by talking into account only the environment's omnidirectional image. HDRShop is a tool developed at the USC Institute for Creative Technologies that can create light diffusion images using Light Probe images in HDR format. Then, the light diffusion images can used to illuminate our objects without the presence of any light source. To calculate the irradiance, we have to create a Texture Cube Map from the light diffusion image. The normal vector of each point will show how much light falls on that point.

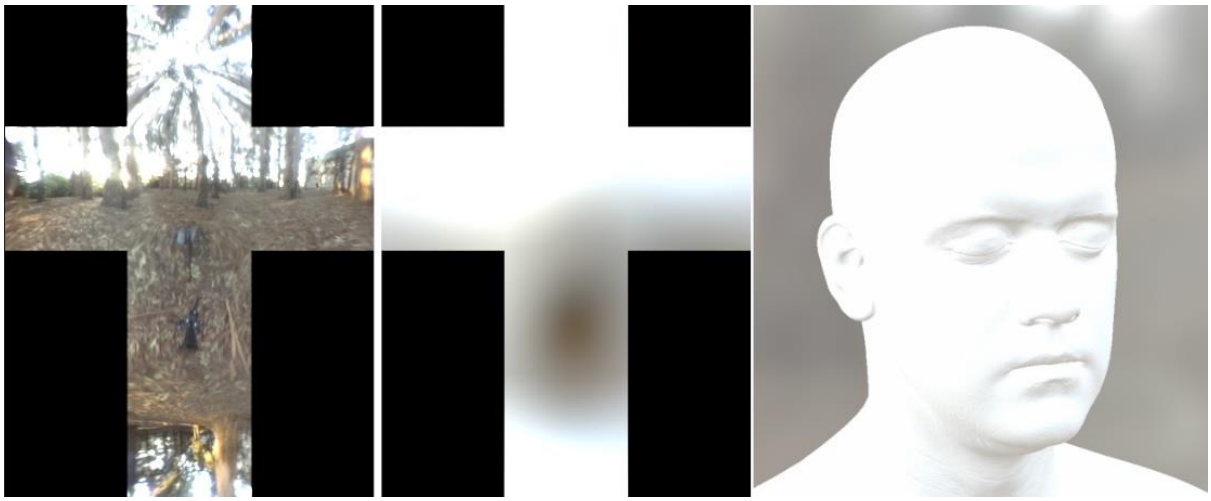

Figure 15: Environment Map (left), irradiance environment map (middle), irradiance as applied to a model (right)

Both reflection and light diffusion environment mapping are techniques that not only can be used in real-time applications, but also they can be supported in skeletal-based animated characters. Reflected vector can be calculated very easily in real time even for scenes with multiple animated objects. For light diffusion, the fastest way to produce the irradiance environment map is by using Ramamoorthi's technique [38], which is based on spherical harmonic convolution. Even if it improves the brute-force algorithm, it is still too slow to be applied in real-tile systems. The best solution to this problem is to precompute it and use it for Image-Based Lighting purposes. Even if we use a static texture cube map for dynamic scenes, the users will not notice the minor difference. IBL techniques have been employed in real-time, articulated virtual characters as shown in [39] and [40].

1.7 Rendering Equation

In computer graphics, we need a way to describe the complete, physical-based illumination of a virtual scene. To do that, we used the Rendering Equation, which generalizes how the light interacts with every surface in a physically-correct manner. The Rendering Equation was proposed by Kajiya in 1986 (Equation 1). It has been known long before that in thermal engineering, but Kajiya was the first to apply this equation to Computer Graphics. Even though it's used as an approach to light's behavior, it was not meant to model effects as subsurface scattering, fluorescence or phosphorescence in its first formulation. In this thesis, we will use the rendering equation as the basis from which our real-time Screen Space implementations will be derived. We will refer to Kautz' work [41] for the appropriate formulation of rendering equation for each of the effects we are going to implement in our system. So, it is basically a Friedholm integral equation of the second kind, describing all the light exchange in any point in a scene:

$$L_o(x,\upsilon) = L_e(x,\upsilon) + \int_\Omega f_r(x, I_\omega, \upsilon_\omega) L_{in}(x, I)(I \cdot n) dI$$

Equation 1: Rendering Equation

x	Position
υ	Viewing Direction (global)
υ_ω	Viewing Direction (local)
I	Light Direction (global)
I_ω	Light Direction (local)
fr	Bidirectional Reflectance Distribution Function
L_o	Reflected Exit Radiance
L_{in}	Incident Radiance
L_e	Emitted Radiance

Table 1: Rendering Equation Variables

The bidirectional reflectance distribution function describes how light incident on a surface is reflected. It is defined as the ratio of the reflected radiance L_o leaving position x towards direction ω_o and the irradiance arriving in x from direction ω_i. We consider a hemisphere positioned on X from which we will retrieve all possible directions ω_i and ω_o.

$$f_r(x, I_\omega, \upsilon_\omega) = \frac{dL_o(x,\omega_o)}{L_{in}(x,\omega_i)\cos\theta_i d\omega_i}$$

Equation 2: Bidirectional Reflectance Distribution Function

The BRDF can be extended to take into account the refraction and transmission. Then, it's called bidirectional scattering distribution function BSDF. In that case instead of hemisphere, we consider a sphere to be placed on X and from which we will obtain ω_i and ω_o. In this thesis we will use the BRDF function based on some assumptions. First of all, we consider our materials are isotropic. That means the reflection doesn't change when the surface is rotated. In order to describe ω_i and ω_o, we need only 2 angles θ_ι and θ_o. Other than that, we assume that the reflected light exits the surface at the same point it hit it.

1.8 Collada Format

In order to implement all the techniques and algorithms that we mentioned previously, we need a file format that supports skeletal-based animated models. Collada is a file format that can be freely supported by any interactive 3D application [42]. It can be processed according to XML-based schema. It supports not only a wide range of features related to modeling and animation, but also various physical attributes that can affect the visualization of the scene. A lot of tools, such as 3DS Max and Maya, that focus on 3D modeling and animation support Collada format, although in some cases a plugin is necessary. Collada is also used in 3D game engines such us CryEngine 2, Unreal Engine and Torque 3D. In 27 March 2013, COLLADA 1.5.0 has been published as an official ISO standard, which increases its reliability for 3D asset interchange between applications. Another feature that makes Collada format being more desired is the fact that most of 3D printers support it. And even if some don't, it can easily be converted in other formats.

1.9 Game Engines

A 3D game engine is a software framework that can be used by developers to create 3D games. The reason developers prefer to use a game engine than starting coding from scratch is because the engines provide a generic functionality. Game engines consist of several components. Each one of them has its own role, which cover any of the following aspects[8]: 3D rendering, physics, collision detection, sound, scripting, animation, networking, streaming, memory management, threading, localization support, and a scene graph. An important feature of game engines is that they can be reused by different developers to create different games. According to [43], game engine are differentiated by the following parameters: cost, features, ease of use, support, skill required, learning curve, interface and plug-ins. Table 2 shows a comparison between some properties of the most popular game engines.

[8] List taken from: http://en.wikipedia.org/wiki/Game_engine

Engine Name	Cross Platform	Physics Engine	Destructible Objects	Availability
CryEngine 3	No	Custom	Yes	1.200.000$
Unreal (UDK)	No	PhysX	Yes	99$ + 25% of royalties
Unity	Yes	PhysX	No	1.500$
Unigine	Yes	Custom	Yes	~50.000$
Hero Engine	No	PhysX	No	99$/Year
Esethel Engine	Yes	PhysX	Yes	199$

Table 2: 3D Game Engine Comparison[9]

[9] Comparison Table : http://www.esenthel.com/?id=compare

2 Area and Spot-based lighting

2.1 Spot Lighting

We use 2 different ways to illuminate the virtual character in our system, via spot lighting and image based lighting. In spot lighting we have a number of light sources illuminating our scene. For this type of lighting, we will use the following formulation of the rendering equation:

$$L_o(x,v) = k_a L_a + \sum_{j=0}^{n} f_r(x, I_\omega, v_\omega) \frac{I_j}{r^2} V(x,I)(I \cdot n)$$

where L_a is the emitted radiance of the scene and k_a is its scale factor. The term V indicates the visibility of point x towards I direction. This equation supports n light sources with I_j direction and r distance from point x.

Spot Lighting is the most complicated type of 0-area light compared to directional and point light. The main attributes that describe a spot light are color, position, direction and attenuation. Light's emitted color will affect the color of the surface it will fall. Depending on the position and the direction of the light source, different parts of the object will be illuminated. The area illuminated by the Spot light looks like a cone and the base of the cone is the far plane Figure 16. The points that reside out of the cone are not affected at all. To ensure this functionality we have to calculate the angle between the vectors: light-direction and light-to-point. If the angle is higher than the falloff angle, the point will not be affected by the light. We also have to take into account the attenuation due to angle, because the higher the angle, the darker the point becomes. We can also use attenuation depending on how close the point is to the light, but in our system we avoid this factor. The GLSL [44]code below describes this behavior:

```
float spot_cos = dot(light_direction, light_to_point);
if (spot_cos > falloff_cos) {
  //linear attenuation
  angle_attenuation = (spot_cos - falloff_cos)/(1.0 - falloff_cos);
  //implement lighting
}
```

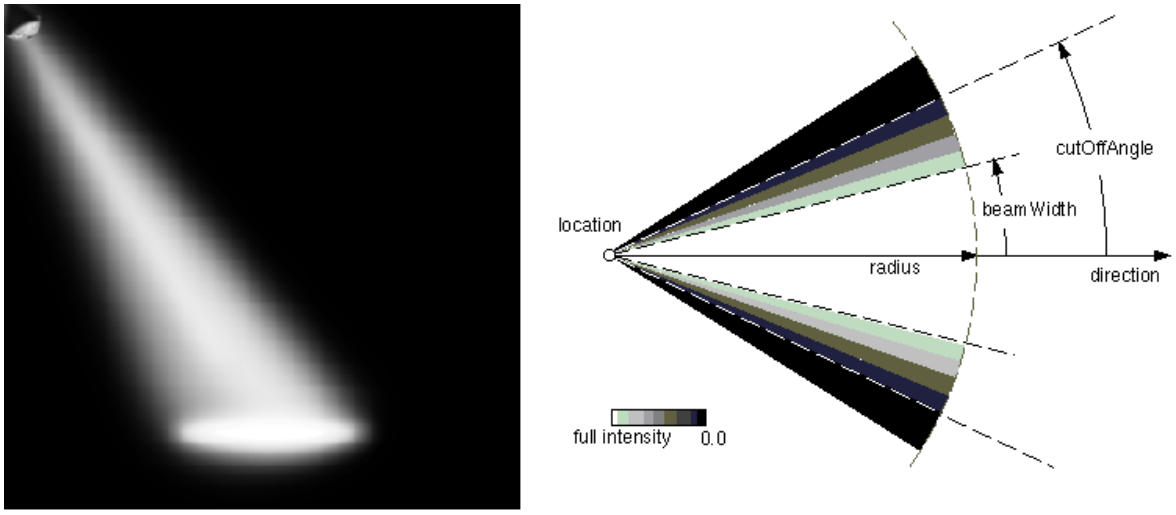

Figure 16: Spotligting[10]

In order to simulate the lighting effect on any object we have to take into account the diffuse reflection and the specular reflection. We will refer to diffuse reflection as diffuse lighting. When a light ray hits a surface, it reflects in all directions. According to Lambert's cosine law, the luminous intensity of the reflected rays varies according to the angle between the observer's line of sight and the surface normal. In our system, we will consider all surfaces to be ideal diffusely reflecting, which means the reflected light rays have the same intensity in all directions. In diffuse lighting, the object's brightness is depending on the surface orientation. The more the surface is faced towards the light source, the brighter is appears. The orientation of the surface is described with its normal. The diffuse lighting is implemented as follows:

```
diffuse = light_color * angle_attenuation * max(dot(normal, -light_dir),0.0) * object_color ;
```

In the previous GLSL code snippet, we use object color to describe the reflected light's color. The reason why we implement it this way is because the colored objects tend to absorb a portion of the incoming energy, changing the light's wavelength. This phenomenon is known as albedo and it defines the ratio of the reflected radiation from the surface, to the incident radiation upon the surface (reflected/incident) [28]. Black surfaces absorb all the incoming power while white surfaces absorb none. That's why even though the light's color is white, the reflected light from the skin has the same color as the skin.

[10] Image: http://www.web3d.org/files/specifications/19775-1/V3.2/Part01/components/lighting.html

2.2 Specular Reflection

Another parameter that contributes to the way the surface is appeared to the user, is the specular reflection. It is the effect that simulates the bright spot of light on glossy surfaces. The specular highlight reflects the color of the light source instead of the object color. In human skin, this happens due to the topmost oily layer [27]. For an ideal reflection, the angle between the incident light ray and the normal is equal with the angle between the reflected ray and the normal. To implement the specular reflection on skin, we have to use a distribution function based on microfasets. We assume that surfaces that are not smooth are composed of many tiny facets. Each one of them has its own normal. The rougher the object is , the more these normals differ from a smooth surface. In order to determine the specularity of the surface, we have to calculate the angle between the halfway vector and the normal for each of the facets. In our system, to simulate the specularity of skin, we use Beckmann microfacet distribution function. Figure 17 shows the precomputed result of the distribution function as taken from [25]. The texture sampling is implemented with dot(normal, halfway) as U and roughness as V coordinates [0,1].

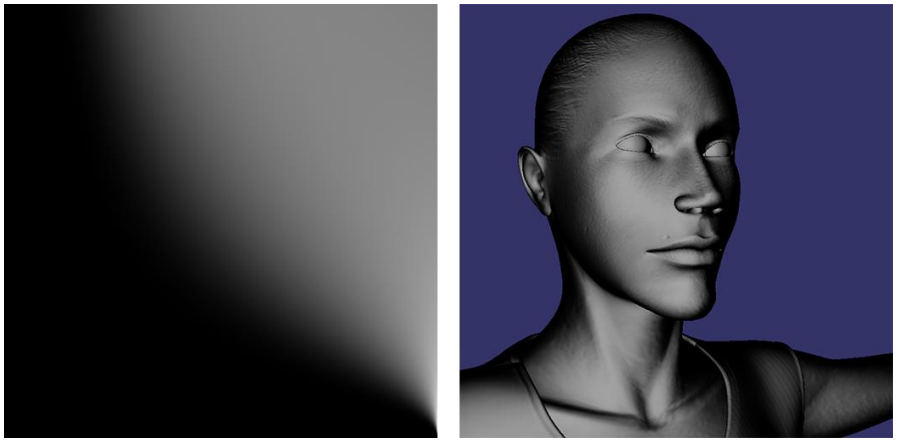

Figure 17: Precomputed Beckmann distribution (left) as applied to Alyson (right)

To simulate properly the specular reflectance, we need a technique that not only applies persuasively the Bidirectional Reflectance Distribution Function (BRDF) on our model, but also it has to be able to run efficiently in real time for hierarchical models such as virtual characters. We used the Kelemen/Szirmay-Kalos Specular BRDF [28], which is derived by the reflection density. That is the probability of a photon to leave the surface at a differential solid angle around v_ω given it comes from I_ω:

$$\mathrm{a}(v) = \int_\Omega f_r(I_\omega, v_\omega) \cos\theta' \, dI$$

where a is the albedo and θ' is the angle between I_ω and normal vector. The BRDF f_r is the reflection density divided by the cosine of the outgoing angle θ:

$$f_r(I_\omega, \upsilon_\omega) = \frac{w(I_\omega,\upsilon_\omega)}{\cos\theta}$$

The reason why we use θ' instead of θ in the rendering equation formulation is because of the symmetry of BRDF for $I_\omega \; and \; \upsilon_\omega$ directions.

In our system, in order to capture increased specularity at grazing angles, we used Fresnel reflectance. According to this theory, as the angle between the normal and the incident light ray becomes greater, the user tends to receive more reflected light. For this effect, we adopted the implementation from [25] (GLSL code below).

Specular Reflection:

```
float getSpecular(vec3 point_to_light, vec3 normal)

  vec3 halfway = point_to_light + point_to_viewer;
  vec3 halfway_normalized = normalize(halfway);

  //Fresnel reflectance calculation
  float base = 1.0 - dot( point_to_viewer, halfway_normalized );
  float exponential = pow( base, 5.0 );
  float fresnel_reflectance =
    exponential + 0.028 * ( 1.0 -   exponential );

  float normal_dot_half = max(dot(normal, half_normalized),0.0);
  float ph = pow( 2.0 * texture2D(beckmann_texture,
    vec2(normal_dot_half, roughness)).r, 10.0 );
  float mix_f = mix(0.25, fresnel_reflectance, specular_fresnel);
  float kspec = max( ph * mix_f /dot(halfway,halfway), 0 );

}
```

At this point we can either add specular reflection to the diffuse or render it in an alternative buffer. The reason why we have to do this is because Specular Reflection must contribute to the surface color after the calculation of Subsurface Scattering. The issue behind this act will be discussed in the Subsurface Scattering Chapter. Figure 18 shows the results of diffuse light and specular reflection on Alyson with different roughness values.

Figure 18: Diffuse and Specular reflection on Alyson with Roughness 0.3 (left) and 0.4 (right)

2.3 Image Based Lighting

Illuminating the virtual character with just spot lights is not enough to create a realistic depiction. As mentioned before, one of the factors that contribute to diffuse lighting is the angle between the light direction and the surface normal. The greater the angle, the less bright the object appears to be. If the angle is greater than 90 degrees, the surface is not illuminated at all. To increase the realism of our scene, we need to take into account the lighting that's received to the object from the environment. To do so, we employed a technique called Image Based Lighting. This technique is based on Environment Mapping, according to which we use an omnidirectional image to depict the surrounding space. In order to compute the light diffusion in the environment, in which our objects exist, we will use that omnidirectional image. This method will affect our skeletal-based, multi-hierarchical scene by illuminating even the parts that are not visible to any spot light. The diffuse lighting that is received from the environment to each position x is described by the following formula:

$$L_{diffuse}(x;n) = \frac{k_d}{\pi}\int_{\Omega} L_{in}(\mathrm{I})(n \cdot I)dI$$

where n is the normal vector for x position and k_d is the diffuse coefficient. As we will explain later, the illumination will be retrieved from the irradiance map. The term $L_{in}(\mathrm{I})$ refers to the sampling value of this texture. The first thing we have to decide is in which environment we will place our scene. In our system we use Galileo's tomb, which is an environment map in cross format taken from [45]. It's an omnidirectional image, which consists of 6 other images. These images will be used as different sides of the same cube called Skybox. The skybox will surround Alyson giving the delusion that she exists in a real room. Its center will be placed in the same position with user camera. Even though we rotate the camera around Alyson, she will be placed inside the skybox cube. For high environment quality, the images we use are in HDR format.

But how will the environment map affect our character surfaces? Based on the Environment Map, we have created the Irradiance Environment Map, which approximates the light diffusion in the scene. To do that, we have used a tool named HDRShop, to apply a technique described by Ramamoorthi [38] in our environment map. It takes as input an HDR image in Latitude/Longitude format and creates the diffusion convolution image. This method uses spherical harmonics and it is based on the fact that irradiance is insensitive to high frequencies in the lighting. The diffusion convolution image will be in Latitude/Longitude format too. Then we convert the image to cross format in order to retrieve the 6 sides of environment map and create a Texture Cube map. It will be used by our shader code to calculate the illumination of the model according to room's lighting. At this point we have to pay attention to the fact that each texel in HDR image is a 32-bit floating point. That means it can have any float value. What we need is a tone mapping function (Figure 20), which converts that float to a value in the range [0, 1]. It is performed in fragment shader in which vertex normals have been interpolated for each pixel. Each pixel's normal will be used to sample the irradiance texture cube map. The following code performs the illumination on our virtual human model (Alyson) based on environment image-based lighting. In Figure 19 we can see the environment map and the irradiance map that is used to illuminate our virtual character and in Figure 20 we can see the portion of light that Alyson receives from the environment based on different exposure values.

```
vec3 iradiance_color = textureCube(irradiance_tex, normal).rgb;
vec3 irradiance =  (1.0 - exp2(-iradiance_color * exposure));
```

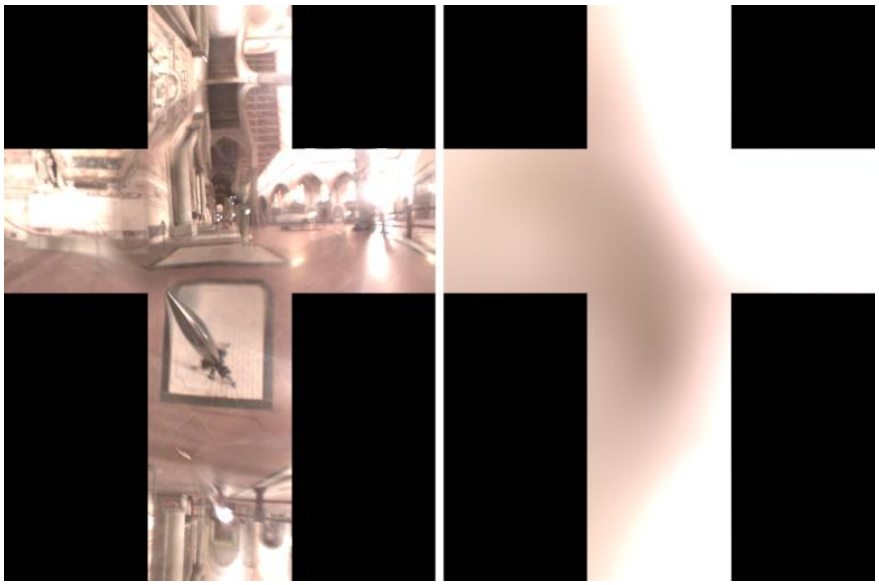

Figure 19: Environment Map (left) and Irradiance Environment Map (right) in cross format

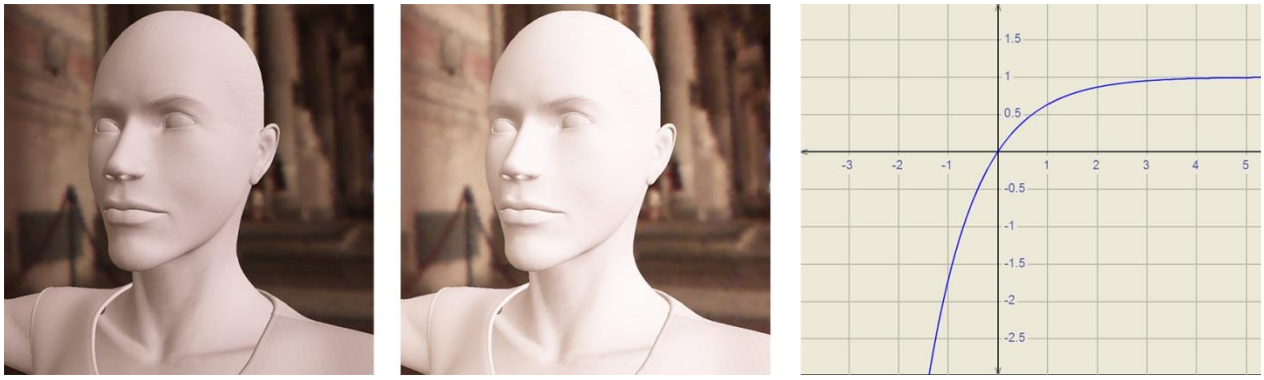

Figure 20: Alyson as illuminated by the environment with exposure 1 (left) and 2 (middle). Tone mapping function (right).

3 Screen-Space Shadowing

In this thesis we have used 2 methods for shadowing our virtual character. The first one is called shadow mapping and produces high-frequency and real-time shadows based on the light sources. The second one is called Ambient Occlusion and creates low-frequency self-shadows based on the neighboring geometries of the shadowed point on the virtual character. The reason why we have used these two techniques is because they increase Alyson's realism and reuse the rendering results of previous passes that are needed for other effects.

3.1 Shadow Mapping

There are currently two prominent methods for producing high-frequency, real-time shadows for virtual characters. The first one is called shadow mapping and the second one shadow volumes. In our system, we have used the shadow mapping method, as it can be employed efficiently in screen-space and coupled well with the ambient occlusion, low frequency shadowing technique. The basic principle in shadowing is to darken the parts of the scene that are not visible by light. The parameters of the technique are related to the type of light (directional, point or spot light). The most important parameters for each light are the view matrix, the projection matrix, the near plane, the far plane and the viewport's width and height. In our system they are encapsulated in a Camera structure. That means if we want to place a light in position A, looking at position B, we have to create a camera in position A, looking at B. Only the parts of the scene that are visible by the camera will be illuminated by the light. Point light and Spot lights use Perspective projection matrix, while directional light uses orthographic projection. We also have to mention the fact that as long as Point light is looking towards all directions, we have to use 6 cameras each one looking at a different side of a virtual cube with center the light position. With that way, the space around the point light's vision will be completely covered.

3.1.1 Implementation

Shadow mapping is a multi-pass technique. That means before we produce the final result, we have to collect some information about our scene. In this case the information we want is stored in a depth buffer. Every light's camera renders the depth in a separate buffer that will be used later. The problem here is that the more lights we use, the more passed we need. If we don't work carefully with our light needs, we will probably affect the performance due to number of passes. Each camera rendering is taking place in a different pass. That means if we use 1 point light, the scene will be processed by 6 passes (one for each camera). That's why we prefer to use spot lights in our system. The code below shows the implementation of the Light Depth Pass. A very important thing that we have to take into account is the fact that after projection, the position's depth value (z coordinate) is not linear. We fix this by calculating the linear depth in the vertex shader. So instead of gl_FragCoord.z in fragment shader, we use the interpolated linear depth value. At this point we

have to mention that in our system, we use 24 bits for each texel in the depth buffer instead of 32 that is the linear_depth value. That means there will be precision problems that we have to solve later on the main pass. The vertex and the fragment shader below handles the Light Depth Pass.

Having produced the depth buffers, now it's time to use them in the main pass where the shadow mapping is implemented. Here, each vertex is passed through N+ 1 different projections, where N is the number of cameras needed by our lights. To simplify it, let's say that N is 1, which means we use only 1 light. For each vertex visible from the user's camera, we have to decide whether or not this vertex is visible from the light's camera. First, we will sample the texture based on XY window coordinates produced by the light's view and projection matrix. Then, we compare the texel's value with the Z coordinate of the vertex after passing it from light's view and projection matrix (let's call it shadow coordinate). If the texel's value is lower that means the vertex is behind another geometry and therefore shadowed. Keep in mind that the depth value that we stored to depth buffer in the previous pass is linear in range [0, 1]. Our depth comparison is based on eye coordinates. So we have to retrieve the value from depth buffer and convert it into eye coordinates. The code below shows a simplified version of our shadow mapping implementation. There are a lot of problems that need work before the shadowing results seem realistic. In the following paragraphs we will discuss those problems and how we solve them in our system.

Vertex Shader In Light Depth Pass:

```
vec4 position_attribute;
uniform float light_near;
uniform float light_far;
uniform mat4 light_projection;
uniform mat4 light_view;
varying float linear_depth;

void main() {
  float depth_in_eye_space =
      (model_view_matrix * position_attribute).z;
  linear_depth = (-depth_in_eye_space - near)/(far - near);
  gl_Position = light_projection * light_view * position_attribute ;
}
```

Fragment Shader In Light Depth Pass:

```
varying float linear_depth;

void main(){
  gl_FragDepth = linear_depth;
}
```

Vertex Shader for shadow mapping in Main Pass:

```
attribute vec4 position_attribute;
uniform mat4 light_projection;
uniform mat4 light_view;
varying vec4 shadow_coords;

void main() {
  shadow_coords  =  light_projection * light_view * position_attribute;
  shadow_coords = shadow_coords / shadow_coords.w;
  shadow_coords.xy = shadow_coords.xy / 2.0 + 0.5 * shadow_coords.w;
  shadow_coords.z = (light_view * position_attribute).z;
  gl_Position = gl_ModelViewProjectionMatrix * position_attribute;
}
```

Fragment Shader for shadow mapping in Main Pass:

```
uniform sampler2D light_depth_texture;
uniform float light_near;
uniform float light_far;
varying vec4 shadow_coords;

int main(){
  float shadowmap_depth =
    texture2D(light_depth_texture,  shadow_coords.xy).r;
  float shadowmap_eye_depth =
    -(shadowmap_depth * (light_far-light_near) + light_near);
  if ( shadow_coords.z < shadowmap_eye_depth ) {
    //the pixel is in shadow
  }
}
```

3.1.2 Shadow Acne

First of all, in the following image you notice how error-prone the simplified version is. Even the points that are directly illuminated by the light source appear to be in shadow. This happens due to the precision problem, called shadow acne, that caused by using 24-bit depth buffers. For example, even when in the light depth pass we store the value -1.33333, in the main pass we retrieve a value with lower precision from the texture, let's say -1.333 (we lose the 2 last digits). Although the difference in this value is small, it causes wrong comparison result when we have to decide whether or not the pixel is shadowed. To fix this, we have to add a bias in shadow_coords.z to cover the depth difference. At this point we have to mention that when a surface is almost parallel to the light rays, we need higher bias. So we modify the code above and we recalculate the bias. The value depth_bias is adjustable by the user.

Bias computation:

```
float cos_theta = clamp(dot(point_to_light, normal),0,1);
float tan = tan(acos(cos_theta));
bias = bias*tan;
bias = clamp(bias, depth_bias, 2 * depth_bias  );
if ( shadow_coords.z + bias < shadowmap_eye_depth ) {
  //the pixel is in shadow
}
```

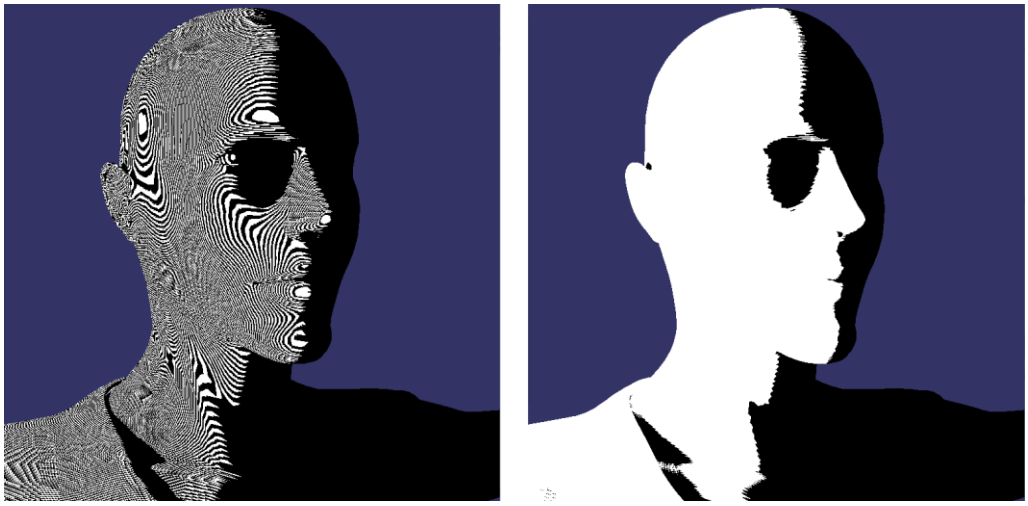

Figure 21: Shadow Mapping without bias (left) and with 0.3 bias (right)

3.1.3 Perspective Aliasing and Projective Aliasing

Two other problems that seem to strongly affect the result in our shadow mapping are called Perspective Aliasing and Projective Aliasing. The first one is caused due to the fact that the farther an illuminating point is from the light, the less accurate is the shadowing. The second one is caused when the illuminated surface is almost parallel to the light rays. In our system we solve these 2 problems by increasing the resolution of the depth buffer by 4 times the resolution of the viewport of the main camera. Also as far as the projective aliasing is concerned, we have discussed previously that we can increase the bias based on the angle of normal and light direction vectors.

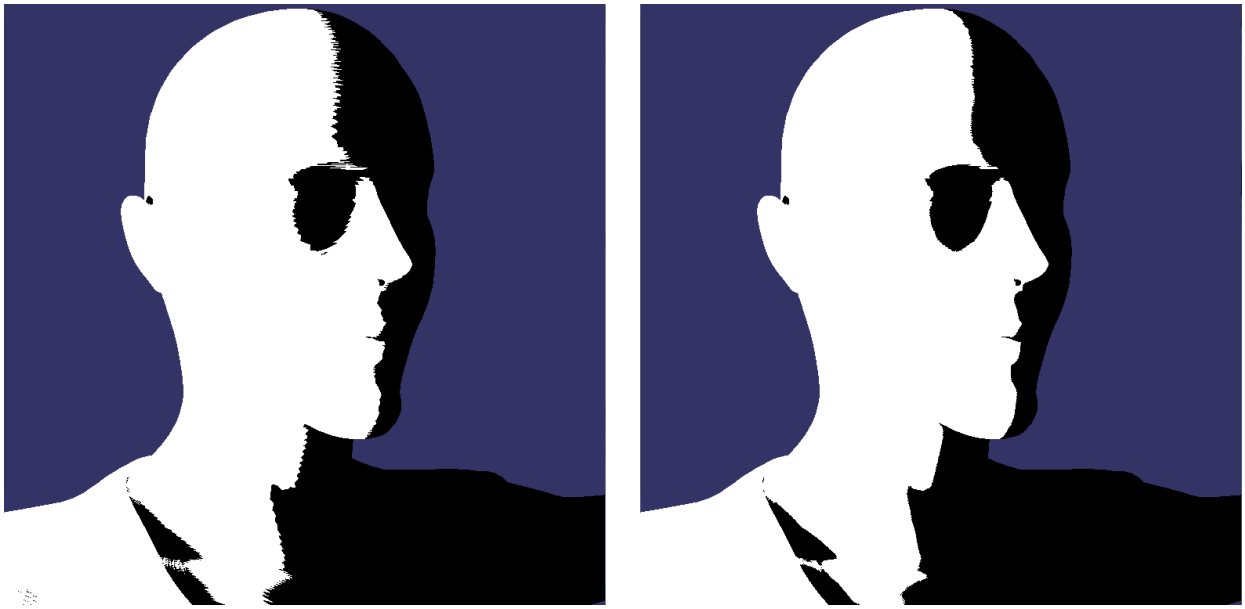

Figure 22: Shadow Mapping using the same Depth buffer resolution as the viewport (left) and 4 times higher than the viewport (right)

3.1.4 Sampling

Even though we solved a lot of issues based on the techniques we mentioned before, we still have to deal with high-frequency shadows. As it is now, each point either is shadowed or not (0 or 1). The color interpolation between the illuminated points and those that are in shadows must not be so steep. That's why we gather shadowing information from a lot of samples. According to this, we decide how strong the shadow is by sampling the neighboring texels as well. The more pixels succeed the depth comparison, the stronger the shadow is for the main pixel. We use 8 random samples generated randomly based on a circle with range 1. Of course, we let the user to decide the actual distance of the neighboring texels that will be retrieved from the shadow map. We also pick different samples for each pixel for better result. The code below shows how each sample's depth comparison result contributes to the shadow.

Sampling in Shadow Mapping:

```
for (int i=0;i<samples;i++){
  index = getRandomIndex();
  shadowmap_depth = texture2D(light_depth_texture, shadow_coords.xy
    + shadow_radius*sample_kernel[index].xy/sample_spread ).r;
  shadowmap_eye_depth = -(shadowmap_depth * (light_far-light_near)
    + light_near);
  if ( shadow_coords.z + bias < shadowmap_eye_depth ){
    visibility-=1.0/(samples);
  }
}
```

Figure 23: Shadow Mapping comparison when we use standard samples for each pixel (left) and random samples for each pixel (right)

Figure 24: Diffuse Lighting and Shadow Mapping as applied to Alyson.

3.2 Screen-Space Ambient Occlusion

As mentioned in a previous section, ambient occlusion is the simulation of the global illumination phenomenon. It is a necessary technique that has to be implemented in order to increase the realism of our scene while depicting low frequency, slowly varying and soft self-shadows. According to this, the objects in the environment are shadowed based on the neighboring objects instead of the lack of direct lighting. The more objects are close to each other, the more shadowed they appear. The Ambient Occlusion is expressed by the following equation:

$$L_{amb}(x) = k_a L_a \int_{\Omega} V(x, I)(I \cdot n) dI$$

where V describes the Visibility of point x towards I direction, L_a is the emitted radiance of the scene and k_a is its scale factor.

Ambient Occlusion is independent of the viewer's position relatively with the objects. Ambient Occlusion can be implemented either for static or dynamic scenes. The most accurate results are produced by ray-tracing algorithms, which take into account the occluders around each object. The problem with them is that they can't be used in real-time systems due to performance issues. In our system, we have implemented a Screen Space Ambient Occlusion technique that not only can be used in real-time applications, but also it gives realistic results similar to those produced by ray-tracing algorithms, without any preprocessing.

The Screen Space Ambient Occlusion algorithm we have implemented in our system is based on [17] and it requires 2 passes. It runs fully in fragment shader, which on the first hand it makes it able to run in real time even on large scale systems, but on the other hand it loses a lot of information about neighboring geometries. The results for this algorithm are depended of viewer's position relatively to the user camera. In order to compute the ambient occlusion for each pixel, we compare the depth of the main pixel to the depth of the neighboring pixels. The more neighboring pixels are closer to the camera than the main pixel, the higher is the ambient occlusion value.

3.2.1 Samples and Performance Issues

Before we discuss the implementation of this technique, we have to talk about some issues that affect the performance. The most important factor is the number of samples we have to use in order to achieve high ambient occlusion accuracy. It is expected that as we increase the number of samples, the fragment shader execution becomes slower. In our system, the number of samples that are used for the ambient occlusion is adjustable. We have noticed that the most efficient results of the ratio accuracy/execution_time, appear when we use a number of samples between 32 and 64. If this number goes higher than 128, we have a huge drop of frame rate and the difference in accuracy is not very noticeable. In Figure 25, we can see how the ambient occlusion result becomes clearer as we increase the number of samples.

Figure 25: SSAO using normal oriented hemisphere with 8 samples (left), 16 samples (middle) and 32 samples (right)

Another thing that we have to take into account is how we choose the neighboring samples. The simplest way is to pick some pixels randomly around the main pixel and compare their depths. The problem here is that it's now accurate as far as the distance from the main pixel is concerned. Let's say that there are 2 pixels between the main pixel (pixel A) and the sample (pixel B). The distance between A and B is greater when the object is far from the main camera than when it is closer due to perspective projection. In our system, we choose a different approach by picking samples from a virtual hemisphere, called sample kernel. We consider its center to be in the same origin with the pixel, but in eye space instead of window space.

Other than the origin, we also have to consider the orientation of the hemisphere. The easiest and fastest approach is for it to look at the user camera. That causes problems when the pixel, whose we want to compute the ambient occlusion belongs to a curved surface, because some of the samples will be at the inner part of the geometry. To increase the accuracy of this technique, we orient the hemisphere towards the same direction as the normal (Figure 26). That method causes all of the samples to be accessible to user's point of view. Last but not least, in order to avoid the gradually change of ambient occlusion values between neighboring pixels, we have to randomly rotate the hemisphere around the normal for each pixel. We use a 64x64 color noise texture for the generation of a random vector, which will control that rotation.

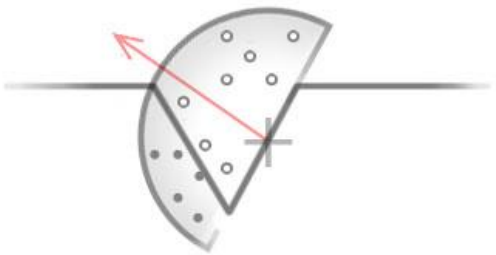

Figure 26: Normal Oriented Hemisphere

3.2.2 Implementation

As we have mentioned before, Screen Space Ambient Occlusion (SSAO) is a 2 pass deferred shading technique. That means in the first pass we have to gather information about our scene and in the second pass we retrieve that information to use it for our effect's implementation. The reason we can't implement it in one pass is because in SSAO each pixel's occlusion value requires information about the neighboring pixels' depths. During each fragment shader's execution, we don't have information for other pixels. So we store the depth values for each pixel in the first pass and in the second pass we can freely access them for any pixel.

During the first pass we render to a Depth Buffer in which each pixel stores its depth value from user's point of view. The depth buffer has 24-bits precision and each value stored is in the range [0, 1]. This depth value has to be linear for the calculation of Ambient Occlusion, which means we manually have to produce it by taking into account the near and far plane and also the distance from camera in eye space for each vertex. Then we pass vertex' linear depth in fragment shader where we store it in depth buffer. It is exactly the same procedure we used for the first pass of shadow mapping.

The calculation of Ambient Occlusion for each pixel will take place in the fragment shader of the main pass (second pass). We will use the occluder term when we want to refer to geometries that appear to be in front of the point described by the main pixel. Theoretically, every sample pixel that has lower depth, is considered to be part of an occluder and it will contribute to occlusion. In the rest of this section we will explain extensively how this technique is implemented in our system. Our method is based on [19] and [46]. The algorithm of this method is described below.

Our Screen Space Ambient Occlusion algorithm:

1. Generate Sample Kernel (Hemisphere)
2. For each pixel:
    2.1. Retrieve the current processing pixel
    2.2. Retrieve its stored Depth Value from the depth buffer
    2.3. Calculate the pixel position in view space based on its window space
    2.4. Create the rotation hemisphere matrix based on a noise texture
    2.5. For each sample:
        2.5.1. Calculate sample's position in view space
        2.5.2. Calculate sample's window coordinates based on its view space position
        2.5.3. Retrieve the sample's depth value from the depth buffer
        2.5.4. Compare the depths of main pixel and sample
        2.5.5. Use depth bias to fix the depth precision
        2.5.6. Calculate occluder's contribution based on the number of samples

First of all, we initialize our sample kernel during system's startup. Its initialization is pretty straight forward. Each sample is consists of 3 float structure for X, Y and Z coordinates. We call this structure vec3. To generate the x and y values, we use a random float function in the range [-1, 1]. We want x and y to be anywhere in the hemisphere with pixel's position as the center. But for the z

value of the sample, we get a random float in the range [0, 1], because the hemisphere has to be oriented towards user's camera. We store the generated samples in a vec3 uniform array of 256 size, because that will be the max number of samples that we might use. These samples will be constant during execution. The hemisphere's radius is adjustable by user through GUI. In Figure 27, notice that as we increase the radius of the hemisphere, the ambient occlusion effect is becomes tenser.

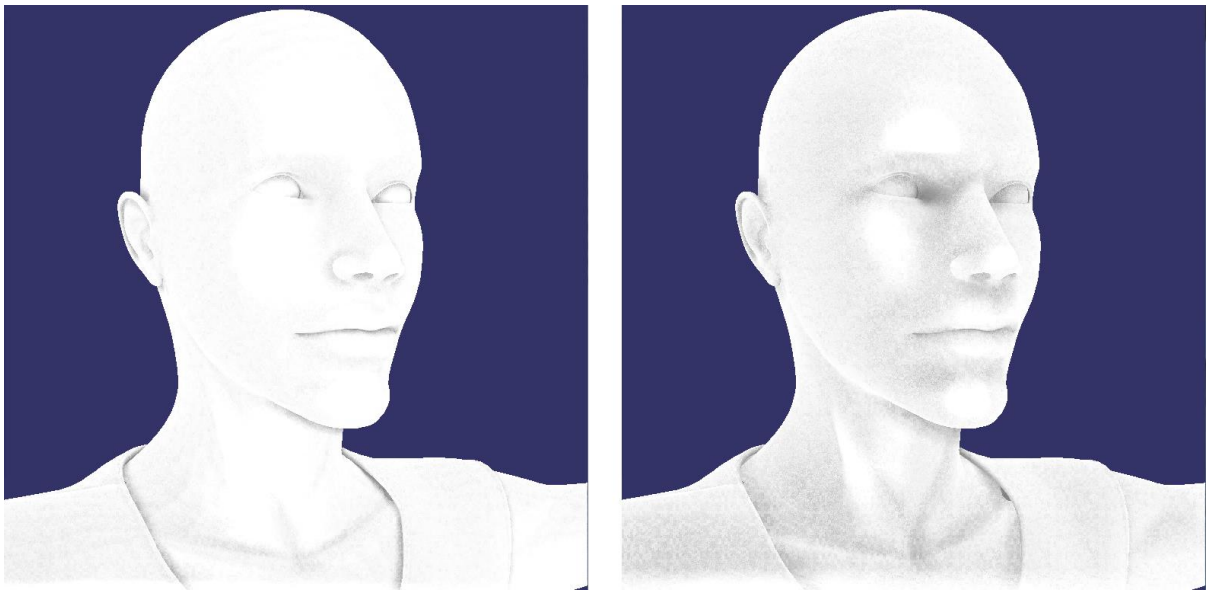

Figure 27: Hemisphere with 64 samples and radius 0.2 (left) or 0.5 (right)

Generating the sample kernel is the only preprocessing step that is needed for the screen space ambient occlusion technique and it's almost instant. When the rendering starts and the fragment shader of the main pass is being executed, we have to calculate the occlusion value for each pixel. First of all, we have to retrieve the current pixel's X and Y texture coordinates in range [0, 1]. To do this we use we use the window coordinates of the pixel in range [window_width, window_height] and also the window's width and height. Next, we use the texture coordinates to retrieve pixel's depth value from the depth buffer by sampling the corresponding texture. Then, we reconstruct the pixel's position in view space to use it later along with the samples. The code below uses pixel's window coordinates to create its view space position. It requires the pixel's linear depth, which is obtained by the depth buffer. First, it generates the normalized device coordinates using the window coordinates. After that, it creates the clipping space position by multiplying with W, which is the inverse step of perspective divide. In the end, we use the Inverse Projection Matrix to go back to View Space.

From Window to View Space:

```
vec4 fromWindowToView(vec2 texcoords, float window_depth){
      //Convert window coordinates to normalized device coordinates
      float ndc_x = texcoords.x * 2.0 - 1.0;
      float ndc_y = texcoords.y * 2.0 - 1.0;
      float ndc_z = window_depth * 2.0 - 1.0;

      //Convert NDC coordinates to clipping coordinates
      vec4 clip_pos = vec4(ndc_x, ndc_y, ndc_z, 1.0)/gl_FragCoord.w;

      //Convert Clipping coordinates to view space coordinates
      vec4 view_space_pos = gl_ProjectionMatrixInverse * clip_pos;
      return view_space_pos;
}
```

This matrix will be used to place each sample to a tangent space described by a normal oriented hemisphere. The view space position of the main pixel will be the origin for this tangent space. To create the tangent space we need 3 axes. We already know that its Z axis will be the pixel's normal. So we have to create the X and the Y axis. Instead of X and Y axes we will call them tangent and bitangent. In our system, we support 2 different approaches to create those vectors. The difference between them is how the tangent vector is created, knowing that bitangent is always the result of cross product between normal and tangent. In the first way, we create the temporary vector: vec3(normal.x, -normal.y, normal.z). The tangent is generated as the result of the cross product between the temp vector and the normal. This method causes the ambient occlusion values to gradually change between neighboring pixels.

M**atrix for sample placement in hemisphere:**

```
mat3 getTBN(vec3 normal){
        vec3 temp_vec = vec3(normal.x, -normal.y, normal.z);
        vec3 tangent = normalize(cross(temp_vec, normal));
        vec3 bitangent = normalize(cross(normal, tangent));
        mat3 tbn = mat3(tangent, bitangent, normal);
        return tbn;
}
```

The other method to create the rotation hemisphere matrix, is to create the tangent vector randomly. This method is more preferable, because the ambient occlusion difference from pixel to pixel is softer. For each pixel we need a random vector. Instead of calculating it during execution, we use a noise texture 64x64 (Figure 28), from which we retrieve random x, y values, creating the vector vec3(x,y,0). Based on Gram–Schmidt's process we are able to produce the tangent vector, using the random vector and the normal. Then, we create the bitangent by applying cross product on normal and tangent. We have to mention that normal, tangent and bitangent vectors have to be orthonormal, which means they are orthogonal and unit vectors. The code block below shows the second approach for the rotation hemisphere matrix.

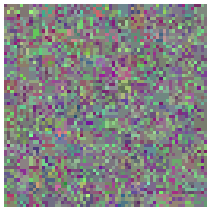

Figure 28: Noise Texture[11]

Matrix for sample placement in a randomly rotated hemisphere:

```
mat3 getRotatedTBN(vec3 normal){
        vec3 rvec = normalize(getRandomVectorFromNoise());
        vec3 tangent = normalize(rvec - normal * dot(normal, rvec));
        vec3 bitangent = cross(normal, tangent);
        mat3 tbn = mat3(tangent, bitangent, normal);
        //tbn = transpose(tbn);
        //gl_FragColor = vec4(normal,1);
        return tbn;
}
```

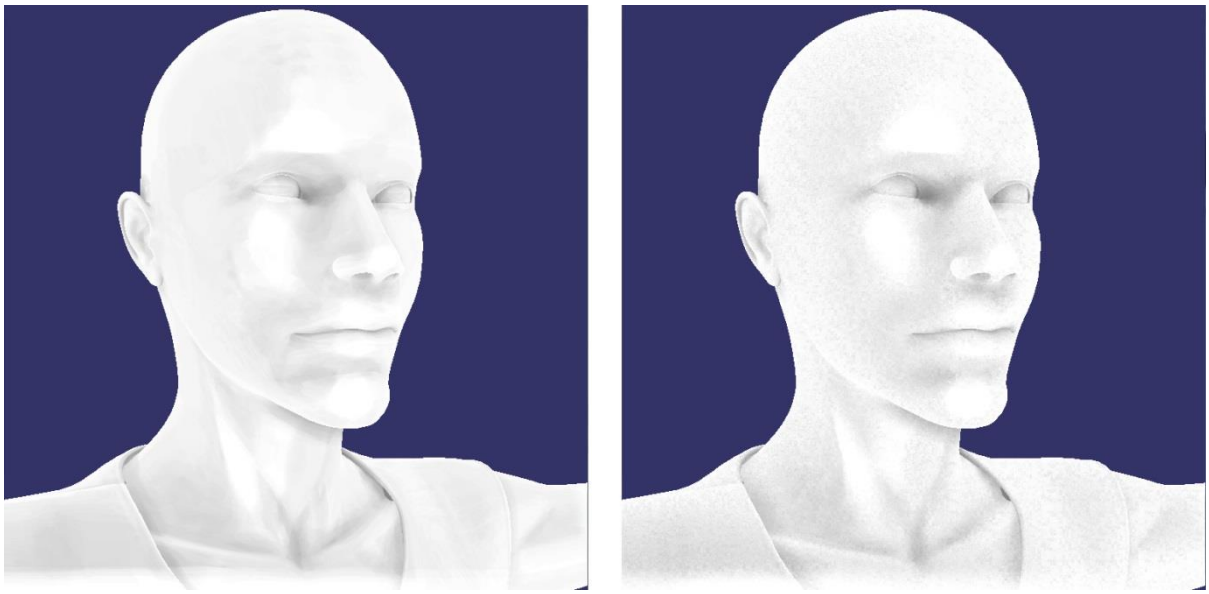

Figure 29: Using the matrix for sample placement in hemisphere (left) and using the matrix for sample placement in a randomly rotated Hemisphere (right)

At this point, we can start iterating samples. Initially, each sample describes its distance from the main pixel in view space. First of all, we apply change of basis by multiplying them with the TBN matrix. Now each sample has coordinates in main pixel's tangent space. After that we add the origin to the sample. At this point, the origin is the main pixel's view space position. The result of this

[11] Image: http://www.gamerendering.com/2009/01/14/ssao/

procedure is to obtain samples in view space. Then, in order to find the occluder corresponded to each sample, we have to convert them to window coordinates and sample the depth buffer as produced in the previous pass. For each depth value lower than the main pixel's depth, we increase an occluder counter, which indicates the number of different geometries that cover the main pixel. The more occluders exist in front of that pixel, the shadowed it appears. In the following getAmbientOcclusion function, we use a bias variable to correct the errors caused by 24-bit precision of the depth buffer. In the following image, we present Alyson's ambient occlusion, as produced by our system.

Ambient Occlusion Implementation in the main pass:

```
vec3 fromViewToWindow(vec4 view_space_pos){
  vec4 clip_pos = gl_ProjectionMatrix * view_space_pos;
  vec4 ndc = clip_pos/clip_pos.w;
  vec3 win_coords = ndc.xyz/2.0 + 0.5;
  return win_coords;
}


float getAmbientOcclusion(){
  vec2 texcoords = vec2(
    (gl_FragCoord.x - 0.5)/(window_width-1.0),
    (gl_FragCoord.y - 0.5)/(window_height-1.0)
  );

  float frag_depth =
    texture2D( camera_depth_texture, texcoords ).r;
  float bias = 0.0003;
  mat3 tbn = getRotatedTBN(eye_normal);
  vec4 origin = fromWindowToView( texcoords, frag_depth);
  float occlusion = 0.0;
  int num_of_occluders =0;
  for (int i = 0; i < num_of_samples; ++i) {
    vec4 sample_in_view =
       origin + vec4(tbn*sample_kernel[i]*hemisphere_radius,0);
    vec3 sample_in_window = fromViewToWindow(sample_in_view);
    float sample_depth =
       texture2D(camera_depth_texture, sample_in_window.xy).r
       + bias ;
    if (sample_depth < frag_depth) {
       occlusion += 1.0;
       ++num_of_occluders;
    }
  }

  float visibility = 1.0 - occlusion/num_of_samples
    * occlusion_contrubition;
  return visibility;
}
```

3.2.3 Ambient Occlusion Implementation Novelty

Knowing the ambient occlusion effect is highly dependent on neighboring geometry, we have implemented Ambient Occlusion in a way that not only the intensity but also the hemisphere radius of the effect for each point can be modified. By doing so, we can preserve the accuracy of the real-time rendering results, even if we have to replace our virtual character with another, which might have different dimensions. Of course even if we use multiple objects in the same scene, each one of them can have its own hemisphere radius which describes how close the samples are to the main pixel. That way we eliminate the issue of Screen Space Ambient Occlusion to be dependent on virtual character's dimensions.

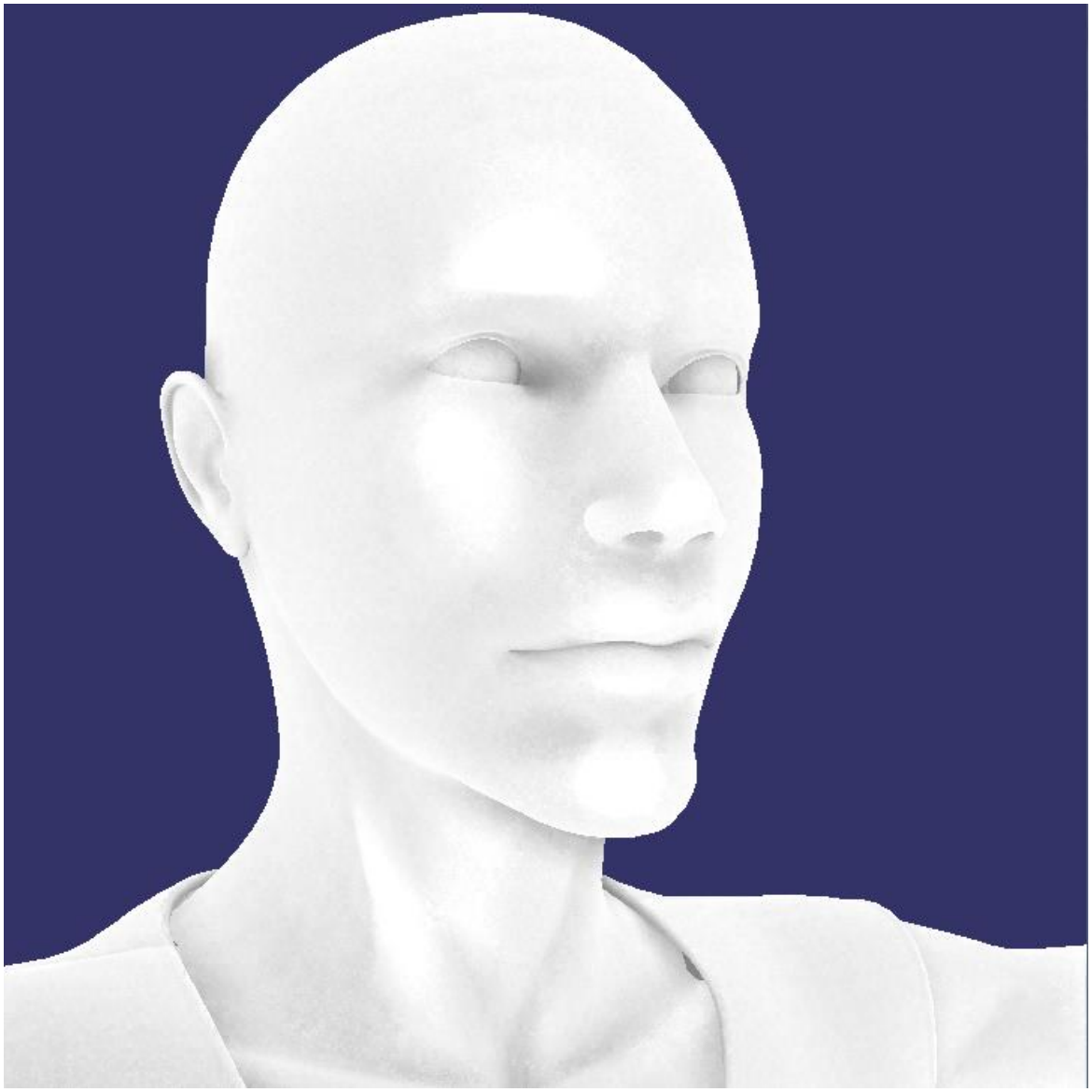

Figure 30: Screen Space Ambient Occlusion on Alyson's head with 128 samples and randomly rotated hemisphere.

Figure 31: Screen Space Ambient Occlusion on Alyson's hands and legs with 128 samples and randomly rotated hemisphere.

4 Subsurface Scattering for dynamic surfaces

There are 2 kinds of reflectance that take place when a light ray interacts with human skin. The first one is the surface reflectance, which causes a small percentage (6%) of the light to be reflected directly without being colored. It occurs due to the topmost oily layer of skin and it can be modeled using a specular reflection function. This type of reflectance has been discussed and its implementation has been explained extensively in the lighting section. The other kind of interaction between the light and the skin is the subsurface reflectance (subsurface scattering). This effect occurs due to the fact that human skin is a translucent material. Translucent objects allow the light to pass through but with a high degree of absorption. The Subsurface Scattering effect is described in terms of Bidirectional Scattering Surface Reflectance Distribution Function (BSSRDF) [47], which relates the outgoing radiance L_o at the point x_o in direction ω_o to incident radiance at the point x_i from direction ω_i:

$$S_d(x_i, \omega_i; , x_o, \omega_o) = \frac{1}{\pi} F_t(x_i, \omega_i) R(\|x_i - x_o\|_2) F_t(x_o, \omega_o)$$

where F_t is the Fresnel transmittance function and R is the diffusion profile. We can convert the BSSRDF into BRDF if we consider $x_o = x_i$. The outgoing radiance can be computed by using the equation:

$$L_o(x, v) = \int_A \int_\Omega S_d(x_i, \omega_i; , x_o, \omega_o) L_{in}(x, I)(I \cdot n) dI$$

where A is the area affected by subsurface scattering.

Subsurface Scattering describes the way light behaves after it enters the skin. While it travels through skin, it is either absorbed partially or scattered many times before it exits a neighboring area. In order for virtual human faces to be rendered realistically, we have to simulate this effect. In this section we will discuss about our implementation of subsurface scattering (SSS), which can be supported by high-detailed skeletal-based animated models. We use Jimenez' [31] implementation of approximating light's scattering underneath the surface of a translucent material. It is based on [32], which is an optimized approach of d'Eon's [25] method. To simulate the subsurface scattering effect for thin parts of human face (ears, nostrils) we have to use a different technique, which first introduced by Jimenez [34] and it is based on [25] and [35] .

What makes virtual skin rendering difficult to simulate is the fact that it consists of multiple translucent layers. Each one of them has a large variety of properties and interacts with the light in a different manner. The more layers we take into account, the more realistic results our system will produce. In our system we will assume that skin consists of 3 layers of translucent material. The mathematical derivation of subsurface scattering theory is a fairly complex analysis and it is described in depth in [25]. In this section we will discuss only the real time implementation of this effect.

4.1 Diffusion Profile

First of all, in our system we consider that light consists of Red, Green and Blue. Each color of light attenuates differently while propagating underneath the skin surface. We will refer to each color's attenuation as Diffusion Profile. As mentioned before, the Diffusion Profile describes the result of the following experiment. We hit a surface with a laser beam and we observe how the light behaves. A glow around the center point where the beam strikes will be created (Figure 32). This happens because a significant amount of light passes the surface, being scattered beneath and in the end comes out from a different point close to the center. While the light is being transmitted in the skin, it is absorbed partially. As we can notice in the diagram below, green and blue are affected by a stronger attenuation than red (which is the color of blood).

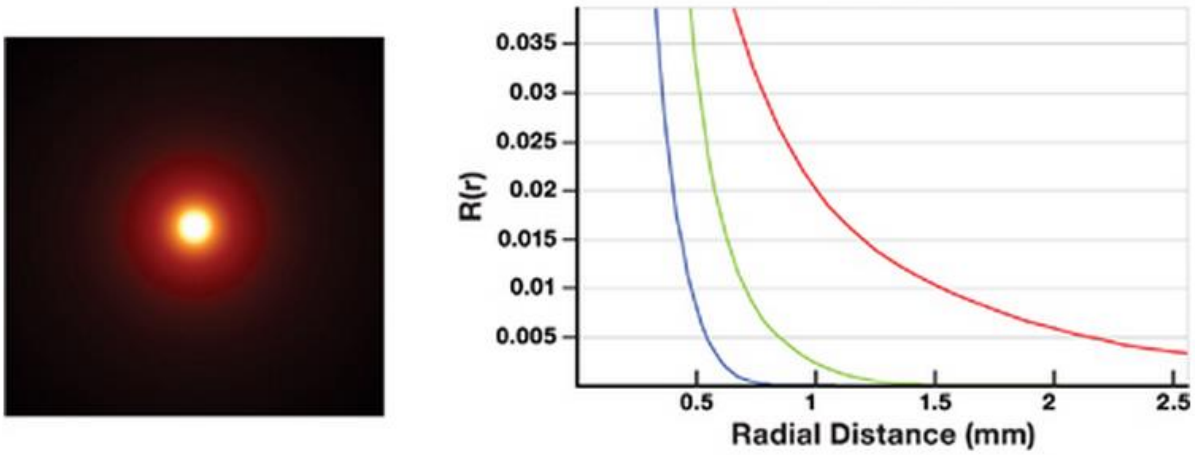

Figure 32: Light's attenuation after hitting a surface with a light ray.

After describing the role of diffusion profile, we wonder what it looks like. The diffusion profile we use in our system is based on Donner's and Jensen's work [29], in which they present a three-layer skin model for the purposes of subsurface scattering. The diffusion profile we use is a sum of six Gaussians with different parameters in each one. They are simultaneously separable and radially symmetric and when they convolve, they produce new Gaussians. Each of the Gaussians describes a circular area with different radius. The reason why we use 6 of them is because that many are needed to accurately match the three-layer model for skin. The diffusion profile is computed as:

$$\mathrm{R}(r) \approx \sum_{i=1}^{k} w_i G(v_i, r) \qquad \mathrm{G}(v, r) = \frac{1}{2\pi v} e^{\frac{-r^2}{(2v)}}$$

where G is the Gaussian, w_i is each color's weight for the current Gaussian v_i is the radius for the circular area covered by each Gaussian and r is the radial distance based on which we will calculate the diffusion profile.

The parameters that are used for each Gaussian can be seen in Figure 33. The reason why we need different weights for red, green and blue, is because the 3 colors are not absorbed by the skin the same. It is also noticeable that the weights for each profile sum to 1. The Diffusion Profile method will be used later in the Subsurface Scattering implementation for thin surfaces.

	Variance (mm^2)	Blur Weights Red	Green	Blue
	0.0064	0.233	0.455	0.649
	0.0484	0.100	0.336	0.344
	0.187	0.118	0.198	0
	0.567	0.113	0.007	0.007
	1.99	0.358	0.004	0
	7.41	0.078	0	0

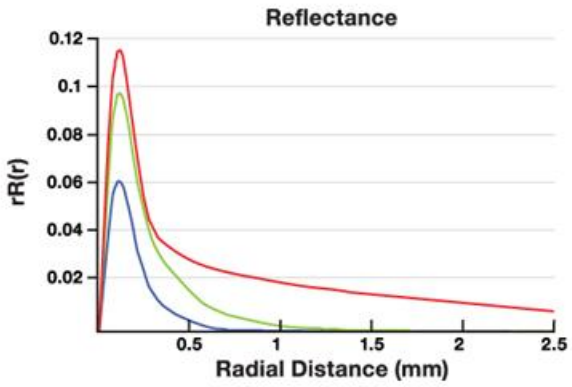

Figure 33: Gaussian Parameters (left), plot of Sum-of-Gaussians (right).

4.2 Implementation

The main use of Subsurface Scattering (SSS) algorithm in a system is to blur the high-frequency details in human skin. To implement it in our system, we use Jimenez' method [31], which is based on the idea of performing the diffusion approximation in screen space. The reason why we prefer this method over the Texture Space one (which discussed in State of the Art section), is because it eliminates a lot of problems that affect the real-time performance. The most important problem of Texture Space method was the fact that it doesn't use the vertex shader to transform object coordinates to clip coordinates. Instead of that, it is used to assign to each vertex a pair of UV coordinates on the unfolded mesh. That means we lose backface culling and view frustum clipping. Texture Space method was applying a convolution even in those parts of the human head that would not be rendered. Another problem of the Texture Space method is that it doesn't scale well because every object in the scene needs its own irradiance map. Apart from that, adjacent points in 3D world, which should be convoluted, may not be adjacent in texture space, causing errors in diffusion.

Texture Space Diffusion could not be used easily in a game, where real time is needed. The Screen Space method not only runs in real time, but it also scales well. The main problem with this method is the fact that, we lack information about scene's geometry. In our case, that means we tend to convolute parts that are adjacent in 2D but not in 3D world. Even though we use some techniques to minimize the errors, the problems are still present.

As we mentioned earlier we implement Jimenez' method for subsurface scattering. In this method the diffusion profile is applied directly to the image with the face. Two passes are needed in order to apply an horizontal and a vertical convolution. We don't sample all the texels in a straight line. Instead of that we use only 17 jittered samples retrieved from Jimenez' method. At this point

we have to mention that the number of samples is dependent to the resolution. The higher the resolution of our output image, the more samples we have to use in order to keep the final result undistorted. In our system, we render with 1024x768 resolution, so 17 samples are enough. Each sample has its own red, green and blue weights, which describe the attenuation for each of the light's colors. They also contain information of how far they are from the main pixel, which is the first sample. Eight samples are used for each side of the current direction (horizontal or vertical). The samples are generated based on Hable's work [32].

We implement the Subsurface Scattering in 2 passes in fragment shader. The first pass takes as input the depth buffer of the main camera and a rendered image of the face without sss. It applies the horizontal convolution using 17 jittered samples. Then, it renders the result to a texture which will be using as input in the second pass, along with the depth buffer. The second pass will convolute the image vertically. Its result will be the final in the whole procedure and it will render to screen. Before we present our code, we have to mention the fact that the subsurface scattering effect has to be applied only on skin surfaces, leaving hair and cloths unaffected. That's why we use an 1-bit per color texture indicating with white color those body parts that are skin (Figure 34).

Jittered Samples:

```
Vec4(0.536343,   0.624624,    0.748867,      0)
Vec4(0.00317394,0.000134823, 3.77269e-005,  -2)
Vec4(0.0100386, 0.000914679, 0.000275702,   -1.53125)
Vec4(0.0144609, 0.00317269,  0.00106399,    -1.125)
Vec4(0.0216301, 0.00794618,  0.00376991,    -0.78125)
Vec4(0.0347317, 0.0151085,   0.00871983,    -0.5)
Vec4(0.0571056, 0.0287432,   0.0172844,     -0.28125)
Vec4(0.0582416, 0.0659959,   0.0411329,     -0.125)
Vec4(0.0324462, 0.0656718,   0.0532821,     -0.03125)
Vec4(0.0324462, 0.0656718,   0.0532821,     0.03125)
Vec4(0.0582416, 0.0659959,   0.0411329,     0.125)
Vec4(0.0571056, 0.0287432,   0.0172844,     0.28125)
Vec4(0.0347317, 0.0151085,   0.00871983,    0.5)
Vec4(0.0216301, 0.00794618,  0.00376991,    0.78125)
Vec4(0.0144609, 0.00317269,  0.00106399,    1.125)
Vec4(0.0100386, 0.000914679, 0.000275702,   1.53125)
Vec4(0.00317394,0.000134823, 3.77269e-005, 2)
```

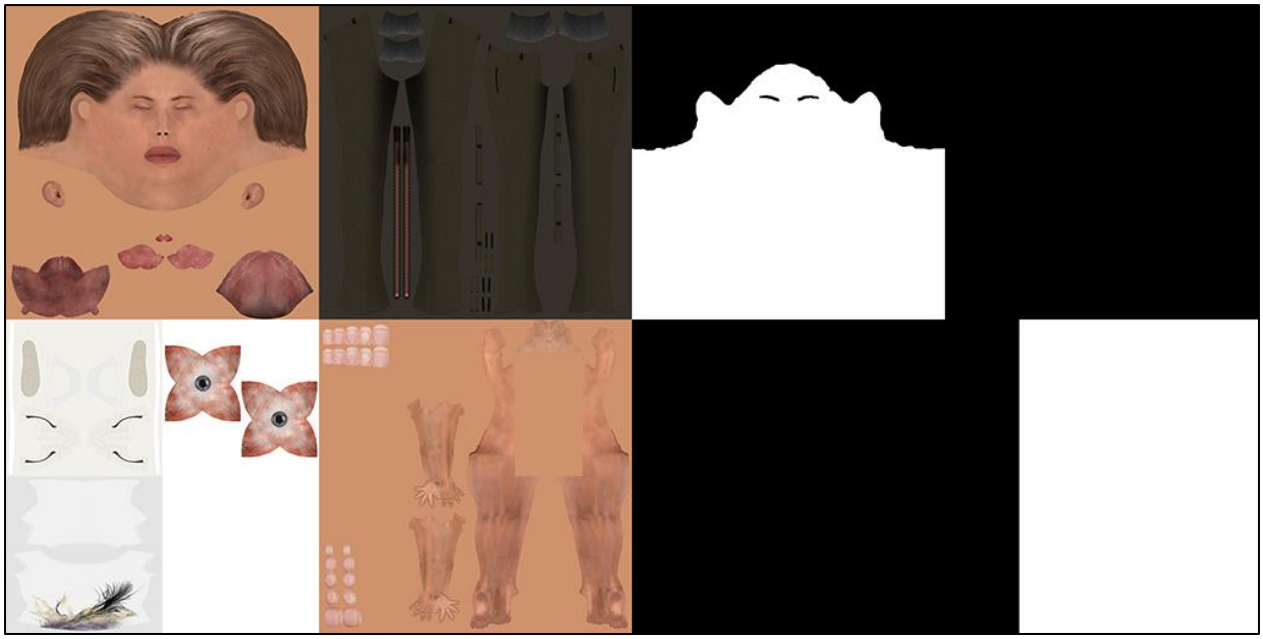

Figure 34: Diffuse texture (left) and Skin texture (right)

Another thing that has to be considered is that the convolution has to be applied only on points that are close to each other in 3D world. This condition is not always true for adjacent points in a 2D image. That's why we need the depth buffer in both passes. By comparing the depths between each sample and the main pixel, we can reduce the error of using non-adjacent points in convolution. We will take into account only those samples that the depth difference doesn't exceed a threshold. In case the depth difference is greater than the threshold, we will use the main pixel's color along with the samples weights for the convolution. This part differs from Jimenez' method, because he interpolates the colors of the sample and the main pixel based on their distance. The following code shows our implementation for the subsurface scattering effect. The variable scale_separable modifies the area that is affect by the light striking at any point and it can be adjusted by user through GUI. Increasing its value means the light travels farther under the skin surface. At this point we also have to add the specular reflection calculated in the main pass and rendered in a separate texture. The reason why we had to do this is because Specular Reflection must be applied after the 2D convolution of Subsurface Scattering on the surface color. By doing so, we avoid artifacts created by the specular highlight on virtual character's surface (Figure X). In **Error! eference source not found.** we compare the rendering results of Alyson's head without and with subsurface scattering. Notice how the high-frequency details in human skin are blurred

Figure 35: Specular Reflection applied on surface before (left) and after (right) Subsurface Scattering

Convolution Passes:

```
void main(){
  float depth_threshold = 0.3;
  float kernel_range = 2; //kernels range from -2 to 2.
  vec2 texcoords = vec2(
   (gl_FragCoord.x - 0.5)/(window_width-1.0),
   (gl_FragCoord.y - 0.5)/(window_height-1.0)
  );

  float depth = texture2D(camera_depth_texture, texcoords).r;
  vec4 color = texture2D(color_texture, texcoords);
  vec3 final_color = color.rgb;
  bool is_skin=texture2D(skin_texture, gl_TexCoord[1].xy).r == 1.0;
  vec2 final_step = vec2(0.0,0.0);
  if (is_skin){
    final_color = final_color * sample_kernel[0].rgb;
    final_step = scale_separable * blur_dir
        * 0.0025 * 1.0/depth * 1.0/kernel_range;
    float eye_depth = -(depth * (far-near) + near);
    for (int i = 1; i < 17; i++) {
      vec2 offset = texcoords + sample_kernel[i].a * final_step;
      vec4 sample_color = texture2D(color_texture, offset);
      float sample_depth =
         texture2D(camera_depth_texture, offset).r;
      float sample_eye_depth = -(sample_depth * (far-near) + near);
      if (abs(eye_depth - sample_eye_depth) < depth_threshold)
        final_color.rgb += sample_color.rgb * sample_kernel[i].rgb ;
      else
        final_color.rgb += color.rgb * sample_kernel[i].rgb;
      }
  }
  if (blur_dir.y == 1)
      final_color += texture2D(specular_texture, texcoords).rgb;
  gl_FragData[0] = vec4(final_color, 1);
}
```

Figure 36: Alyson's head as rendered without Subsurface Scattering (left) and with Subsurface Scattering (right).

4.3 Light transmission through thin surfaces

The simulation of subsurface scattering effect with convolution gives great results. The light travels underneath the skin surface affecting only the neighboring areas, because it is fully attenuated after some distance. Although, this method simulates the effect realistically for thick surfaces, it doesn't suffice for thin surfaces. When a light ray hits a thin surface, it might pass through the skin and be emitted from the other side. In real life this effect can be observed in ears, nostrils or any other thin part of skin (Figure 37).

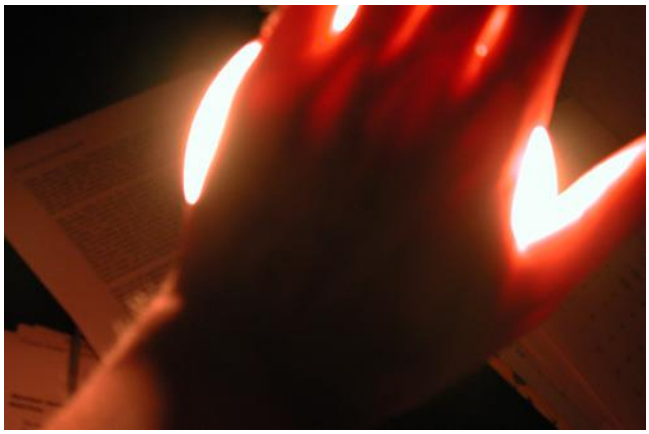

Figure 37: Light transmission through hand[12]

[12] Image: http://www.neilblevins.com/cg_education/translucency/tut28_fig02.jpg

In this thesis, we implement this behavior in screen space. The greatest problem of screen space algorithms is the lack of information about geometry other than what the user can see. In this case we don't know how the thin surfaces are shaped from the other side. What we need to know is the irradiance of a surface that is not observable by the user. Jimenez' solves this problem in [34], even if he uses a screen space implementation. First of all in order to calculate the irradiance at the back of a surface, we need the surface's normal. The key concept to Jimenez' method is to assume that the normal at the back of an object is the reversed of the current pixel normal. Certainly, this is just an approximation, which reduces the accuracy of the effect. This assumption solves our problem and now we can proceed to compute the irradiance.

Another matter that we have to deal with is the calculation of the distance the light travels before it exits from the other side. The reason why we need this is to know how much attenuated it will be. Based on the distance, we can compute light's emittance color. To compute the distance the light travelled, we use Green's method as described in [35]. According to this method, the distance is equal to the depth difference of entrance and exitance points of light. Both of these depth values are based on light's point of view. We have already explained in our shadowing technique implementation how the depth map is created in the Light Depth pass (first pass). We will use the same 24-bit depth buffer in order to retrieve the linear depth of what the light's camera can see. Except from that we need the linear depth of what user can see. The depth values are in the range [0, 1] as retrieved from the depth buffer. A problem that appears here is that the depth values are dependent on near and far plane. Before we subtract the depths to calculate the distance the light travelled, we need to convert them in eye space. With that way our system's implementation of subsurface scattering won't be dependent on each camera's near and far plane. The figure below shows that the depths di1, do1, di2, do2 are from light's point of view and they will be used to calculate the distances s1 and s2.

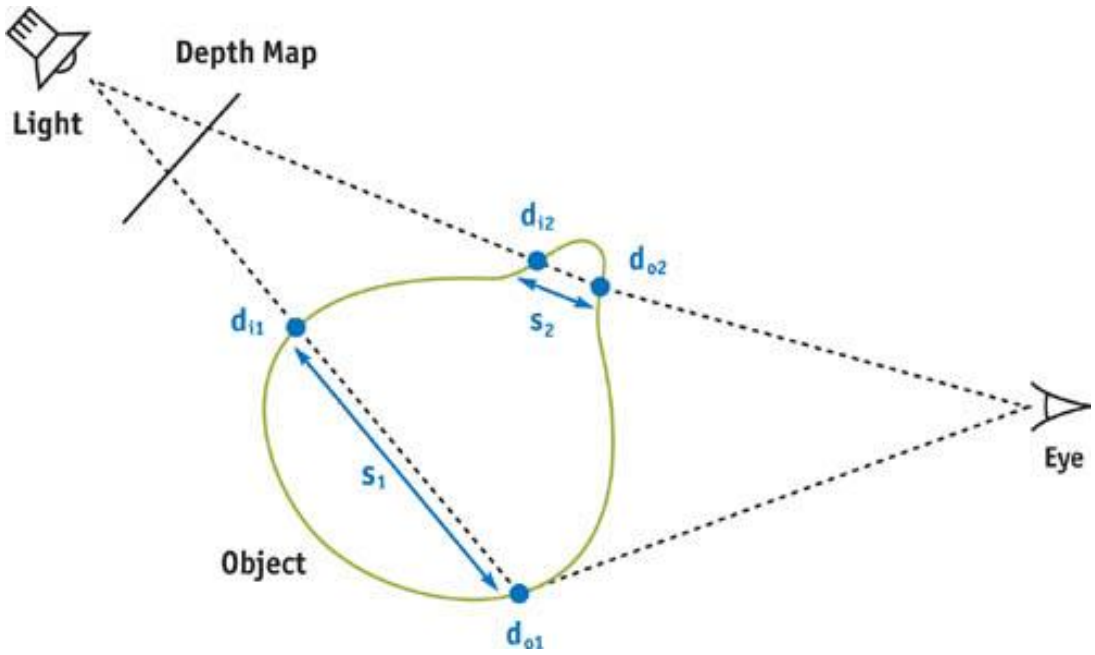

Figure 38: Calculating the distance light has traveled through from light's point of view

Based on the distance between entrance and exitance points of light, we can calculate attenuated light's color. There are 2 ways to do that. The first one discussed in Green's approach, in which the distance is used to sample an 1D Texture like the one in Figure 39. As the distances increases, we sample texels closer to the right edge of the texture. The other method is to use the diffusion profile that described above in this section. As mentioned above, each color of light attenuates differently, while propagating underneath the skin surface. In our system, we implement the second method, which uses the diffusion profile. Its implementation is shown in the GLSL code "Diffusion Profile" below, where dd = - distance_light_travelled.

Figure 39: Distance to color mapping based on 1D Texture.

Diffusion Profile:

```
vec3 getProfile(float dd){
  return   vec3(0.233,       0.455,      0.649) * exp(dd / 0.0064) +
           vec3(0.1,         0.336,      0.344) * exp(dd / 0.0484) +
           vec3(0.118,       0.198,      0.0)   * exp(dd / 0.187)  +
           vec3(0.113,       0.007,      0.007) * exp(dd / 0.567)  +
           vec3(0.358,       0.004,      0.0)   * exp(dd / 1.99)   +
           vec3(0.078,       0.0,        0.0)   * exp(dd / 7.41);
}
```

As happened with shadow mapping, depth buffer produces some problems here as well. This time, they are caused due to shadow map's resolution instead of the 24-bit precision. The problem is that some artifacts appear around the projection's edges, where the depth distance between 2 adjacent pixels can be huge. A prompt fix is to use shadow maps with higher resolution, as we already do to fix shadow mapping issues. But even with 4 times higher resolution that screen's resolution, we still get the annoying artifacts. There are 2 ways to deal with this. The first one is mentioned in Green's SSS approach [35], in which he grows the vertices towards normal when the shadow map is rendered. Jimenez' on the other hand prefers to shrink the vertices in normal direction when creating the shadow coordinates for a vertex. They are the window coordinates from light's point of view. Keep in mind that the distance the light traveled will be calculated based on eye coordinates of light's camera. So, in the end, X and Y shadow coordinates will be in window space and Z coordinate will be in eye space. Figure 40 shows the difference between shrinked and non-shrinked vertices. The two following code snippets show how we have implemented the vertex

shrinking in GLSL vertex shader and light's transmission in GLSL fragment shader. The variable shrinked_bias is configurable by the user and it handles the amount of vertex. Figure 41 and Figure 42 show the subsurface scattering results along with light's transmission through thin skin in our rendering framework.

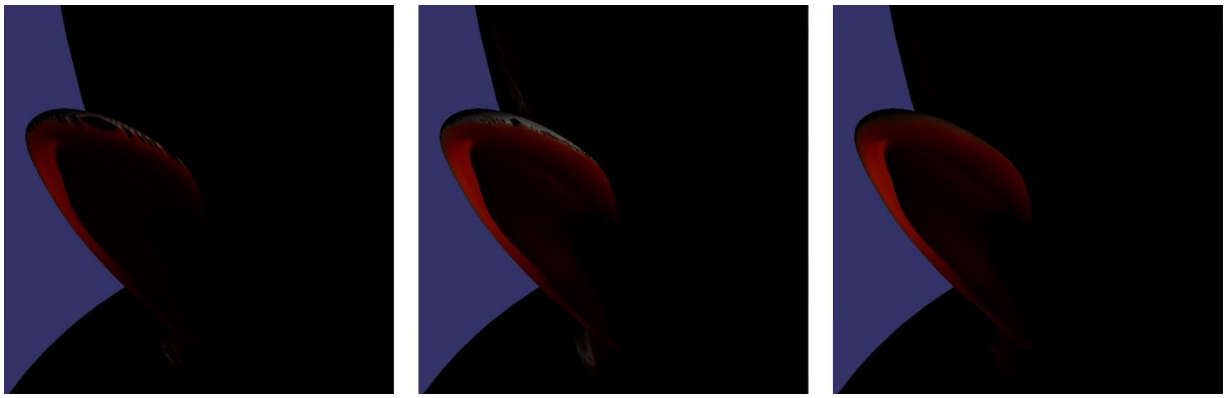

Figure 40: Light transmission through Alyson's ear with low resolution depth map and no vertex shrinking (left), with high resolution depth map and no vertex shrinking (middle) and with high resolution depth map and vertex shrinking (right).

Vertex Shrinking in GLSL vertex shader:

```
vec4 shrinked_vertex = vec4(position_attribute.xyz - shrinked_bias *
     normalize(normal_attribute.xyz), 1);

shadow_coords  =  light_projection * light_view * shrinked_vertex;
shadow_coords = shadow_coords / shadow_coords.w;
shadow_coords.xy = shadow_coords.xy / 2.0 +
     0.5 * shadow_coords.w; light_near, light_far);
shadow_coords.z = (light_view * shrinked_vertex).z;
```

Light Transmission in GLSL fragment shader:

```
vec3 getTransmission(){
  vec4 shadowmap_depth =
    texture2D(light_depth_texture, shadow_coords.xy).r;
  float eye_depth =
    -(shadowmap_depth * (light_far-light_near) + light_near);
  float depth_diff =
    scale_transmittance *abs(shadow_coords.z - eye_depth);
  vec3 profile = getProfile( -depth_diff * depth_diff  );
  float irrandiance =
    clamp(0.3 + dot(light_dir, world_normal) , 0.0, 1.0);
  vec3  transmission = profile * irrandiance;
  return transmission;
}
```

4.4 Subsurface Scattering Implementation Novelties

As we have mentioned in this section, we used Jimenez Screen Space method to simulate the subsurface scattering effect in our rendering framework. The reason why we preferred his method is because it runs in real-time, while produces high accuracy results based on comparison with Ground Truth images. But even though this implementation is highly efficient, we managed to find some parts of the code that can be improved to offer even better results. The first one appears in the vertex shrinking part which is used to avoid projection errors on the edges of the dynamic surfaces. In Jimenez' demo, it is implemented in the fragment shader of the main pass while we in our system it is implemented in the Vertex Shader of the same pass. We made this decision due to the fact the vertex shader is executed less times than the fragment shader for our virtual character. That way we pass to the fragment shader the coordinates produced based on light's view and projection matrices using a vec4 varying. After that they are available for the calculation of light's transmittance through thin skin.

The next difference with Jimenez' method appears again in the fragment shader but this time in separable passes. If the retrieved jittered sample has high depth difference with the main pixel, he interpolates the sample's color with the main pixel's color based on that depth difference. In case these 2 pixels are too far, he uses the main pixel instead of the sample. In our implementation we have defined an depth difference threshold. Every sample that exceeds that distance, it doesn't contribute to the 2D convolution of the subsurface scattering. That way we avoid the interpolation in the fragment shaders of the two Separable passes.

Apart from these 2 differences on SSS implementation, we also have modified the way user adjusts the width that SSS affects the virtual skin area. In Jimenez' method, a common modifier is used for both 2D screen convolution in separable pass and light's transmittance through thin skin in main pass. In our system, we use 2 separate modifiable parameters which allow user to handle the way light passes through skin in a different manner than the way light enters and exits from a neighboring area. This technique can also be used to fix visual errors caused by geometries in body parts that normally should allow light to pass through but they don’t, without affecting the Subsurface Scattering 2D convolution on Screen Space.

Figure 41: Alyson Head rendered with SSS. Light is placed behind the head.

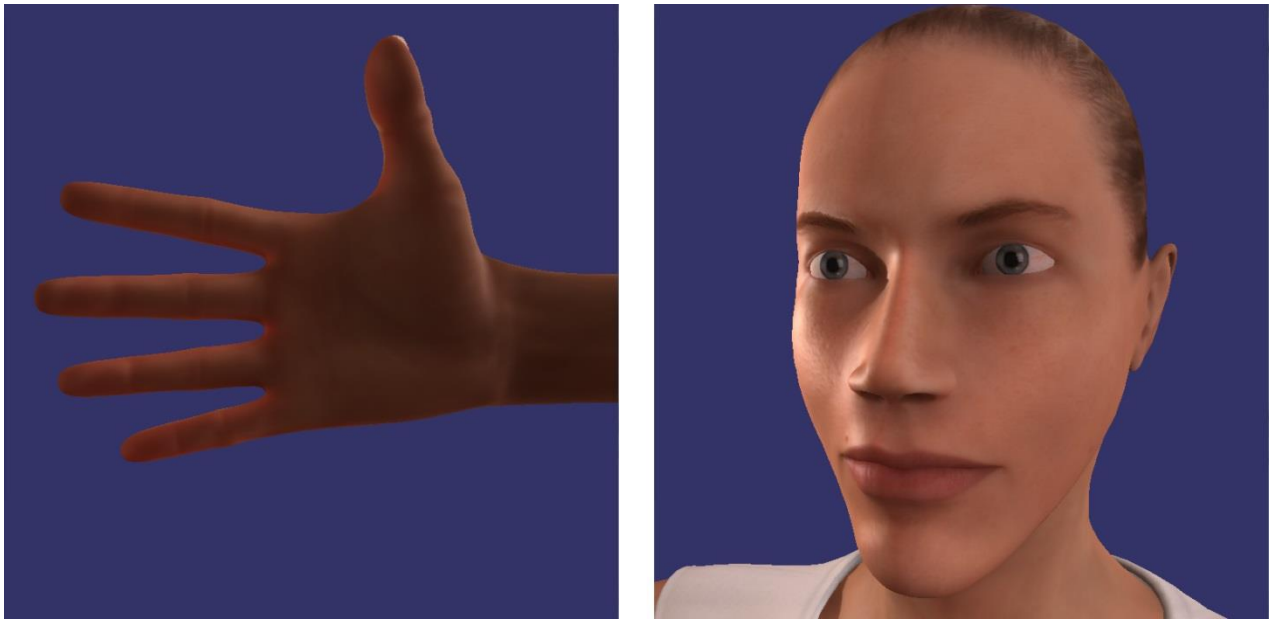

Figure 42: Alyson Hand and head. Light is placed above right hand.

5 Normal Mapping for Multi-Surface Geometry

5.1 Primitives

In mathematics, the normal is defined as the perpendicular vector to an object such as line or surface. In 2D geometries such as line, the normal at each point of this line is the perpendicular vector to the tangent line at that point. But normals are also used for 3D surfaces, in which the normal at each point is the perpendicular vector to the tangent plane at that point. In computer graphics we use the term normal to refer to the perpendicular vector at each vertex. But how can a vector be perpendicular to a vertex? In order to explain why we need normals in a system and how they are created, first we have to explain the structure of a 3D object.

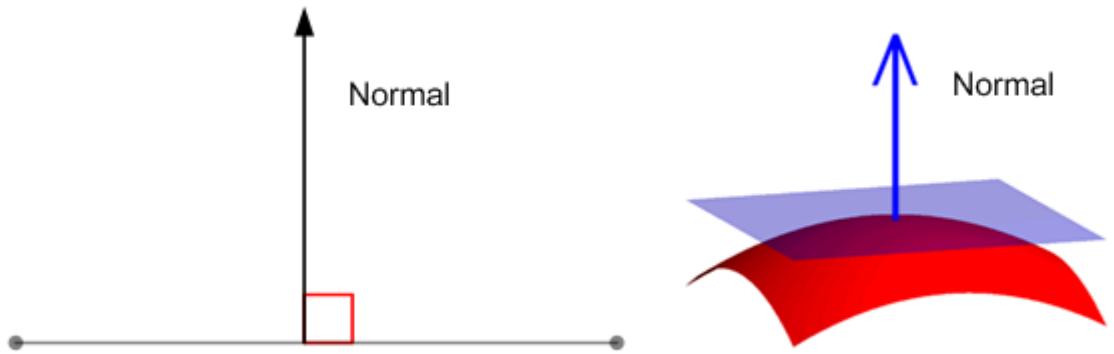

Figure 43: Line Normal (left) and Surface Normal (right).[13]

Every rendering API such as OpenGL provide geometrical objects called primitives. The primitives that are supported by OpenGL are the following: Points, Lines, Line Strip, Line Loop, Triangles, Triangle Strip, Triangle Fan, Quads, Quad Strip and Polygon. Every single 3D object in OpenGL is constructed using any type of these primitives. But no matter which primitive we pick (other than points, lines, line strip and line loop), GPU will split it to triangles. The reason is that triangles have advantages that make them easier to be handled, such as they are always planar and convex. That's the reason why modern GPUs work only with triangles on hardware level. Not because they can't be designed to process quads, but it is more efficient to process triangles as fast as possible and emulate all other primitives with just triangles. The complete list of primitives can be found in [48].

[13] Image (left): http://back2basic.phatcode.net/images/line_normal.png
Image (right): http://en.wikipedia.org/wiki/Normal_(geometry)

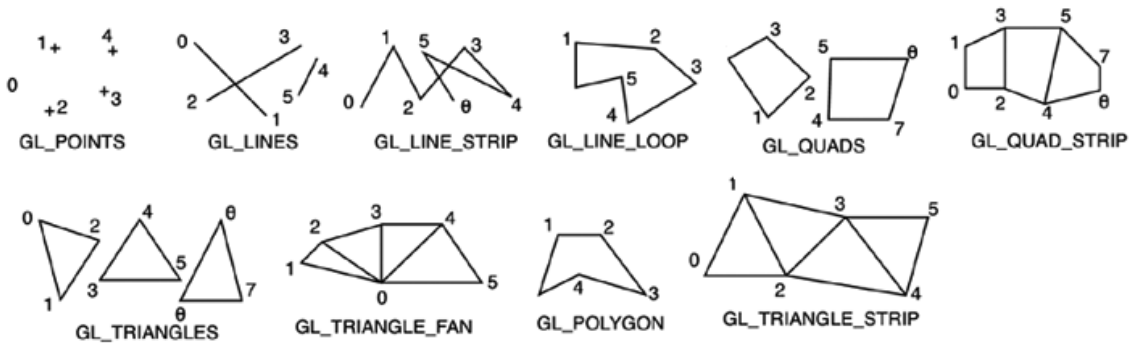

Figure 44: OpenGL Primitives

First of all, we have to clarify that normals are just invisible vectors, which will help us to implement our algorithms. In our system, normals will be used for lighting and shading purposes. Depending on how these normals are created, we get different results as we apply the algorithms on the model. To explain how the normals in our system are created, we will use a cube as model example. We assume that the cube is constructed of 6 quads, 1 for each side. Each quad consists of 4 different vertices. That means there are 24 vertices and every 3 of them have the exact same position in object space. As we mentioned before, every single vertex has its own normal. Each cube side has a different perpendicular vector, which will be used as normal for the 4 vertices of this side. Based on these factors, even though there are vertices that have the same position, the have different normals.

If we don’t eliminate this problem, it will affect the lighting and shading algorithms giving wrong results. What we want is that every 3 vertices that have the same position must have the same normal as well. We call these normals unified. So what happens when the normals aren't unified? Lighting algorithms implemented in Vertex shader affect each vertex' color based on its normal. If the normals are not unified, even though some vertices have the same position, they won't have the same color. This will affect the color interpolation from Vertex to Fragment shader. The same principles apply to the lighting algorithms that are implemented in fragment shader. The only difference is that instead of interpolating the color, we interpolate the normal. The following images are produces from 3ds Max and show the difference between unified and non-unified normal in 3 different meshes.

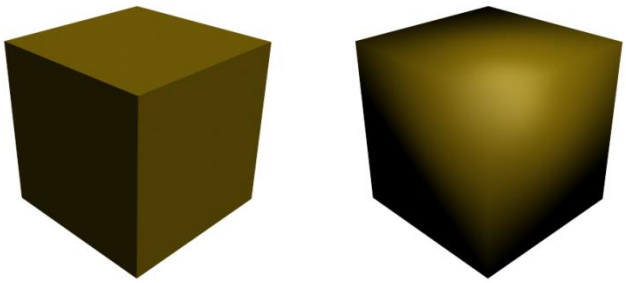

Figure 45: Comparison between non-unified and unified normals in a box

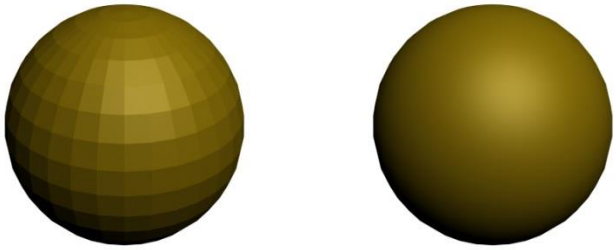

Figure 46: Comparison between non-unified and unified normals in a sphere

Figure 47: Comparison between non-unified and unified normals in Alyson

5.2 Bumpiness

Every single of the algorithms we have discussed until now, require a normal vector to be assigned for each of the vertices in the illuminated object. The rendering results of these algorithms are closely dependent on the normals. If the normals are not calculated correctly, they will cause errors to the final image. If we look carefully to the previous image, in which Alyson is rendered with unified normals, we can notice how plastic her face appears. It is perfectly smooth without any physical characteristic. Well, in order for our techniques to produce realistic images, we have to take into account the true nature of human face. A realistic face model must include wrinkles, pores, freckles, hair follicles and scars. And not just the face, we also have to consider the rest of the body, the cloths and the shoes. We will call bumpiness the set of these characteristics. So, the question here is how can we make Alyson to be more realistic giving emphasis to her surface particularity or else how can we add bumpiness to her.

One solution to this matter is to simply increase the model details. Well, it's not that simple, because if we really want to make these characteristic more tense, we have to spend dozens of hours in front of the modeling tools. No one uses this method so neither do we. Keep in mind that it doesn’t matter how detailed the model is but how it appears to the user. To change the way our model is rendered, all we have to do is to modify model's normals to simulate the surface characteristics we mentioned before (wrinkles, scars, etc). The algorithms we use, which are implemented in fragment shader, decide the color of pixel based on its interpolated normal vector. To manage to modify each vertex' normal, we use a technique called Normal Mapping.

In order to be able to use this technique, we have to make two preparations: define the tangent space and create the normal map for our model. To define the tangent space each vertex also needs the tangent vector, in addition to normal. Tangent is a perpendicular vector to normal, which is created based on the texture coordinates. By calculating the cross product of these 2 vectors, we create a third vector called bitangent. Tangent, bitangent and normal are perpendicular to each other. These 3 vectors are used as a basis to define the tangent space for each vertex. Basis is a set of independent vectors that describe a coordinate system. In this case the tangent space is a three-dimensional coordinate system with origin the position of vertex. The difference between the tangent space and the world space other than the position of the origin, is that their axes don’t have the same direction. Based on the tangent space we define a 3x3 matrix called TBN. In order to convert a point or a vector from world space to tangent space, we have to multiply the point's or vector's coordinates with the TBN matrix.

Before we discuss how we use the TBN matrix, we need to talk about the second preparation for Normal Mapping. So, in order to add bumpiness to our model, another thing we have to do is to create the normal map. We use the NVIDIA texture tool, which is a plug-in for Photoshop. It takes the diffuse texture of a model as input and creates a normal map for the same model based on that diffuse texture. Figure 48 shows the tool's interface. To create the normal map for Alyson we use 4 samples, average RGB height source and we set the scale to 8. In case we need more intense bumpiness, we can increase the scale value. In Figure 49 you can see the normal map we use in our system for Alyson's bumpiness.

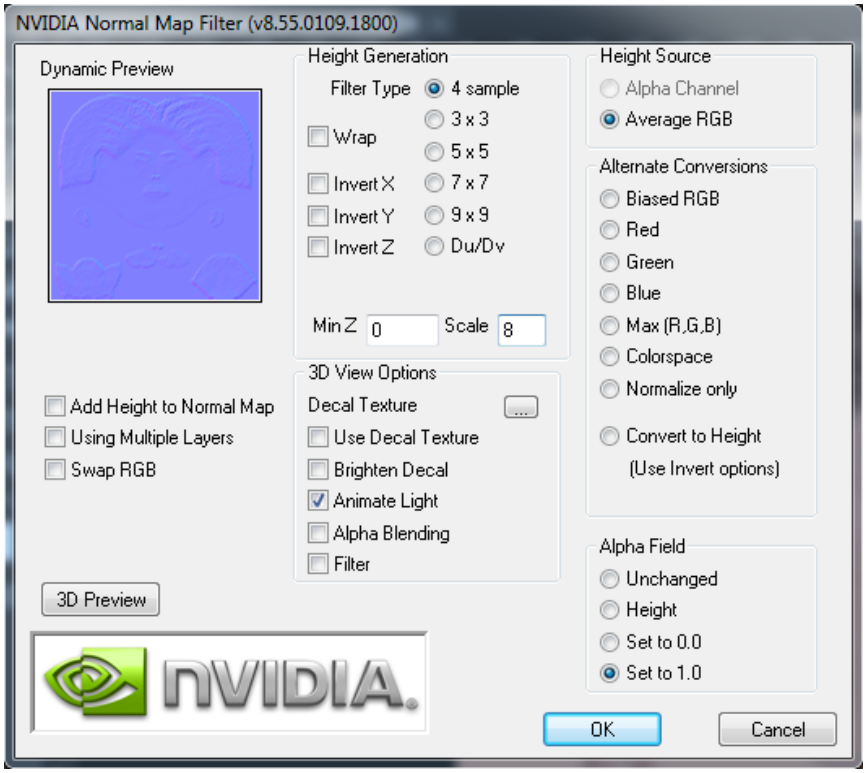

Figure 48: NVIDIA texture tool plug-in for Photoshop

Figure 49: Normal Map of Alyson's head

But how is the normal map used to apply bumpiness in a model? First of all, you can see that the normal map is a texture, in which each pixel is described by the RGB color model. Using each vertex' texture coordinates, we sample the normal map to get the normal for that vertex. Each pixel channel has value between 0 and 1. To convert the retrieved color into vector we multiply each pixel's channel with 2 and we subtract 1. So vector.xyz = normal_color.xyz * 2 - 1. The red, green and blue colors describe the x, y and z coordinates for the vector. Basically, this newly created vector describes the difference from the original normal. So, if the retrieved color is the R:0.5, G:0.5, B:1.0, it corresponds to the vector with coordinates (0,0,1). If we multiply this vector with the TBN, we will get the vertex' normal without any modification. The reason why a bluish color prevails in the normal map is because the bumpiness in human face is created by applying a small modification to each vertex' normal. Those colors that are close to red, tend to change normals direction towards tangent vector, which is the X axis in the tangent space. The same happens for green color and bitangent vector (Y axis). The code below shows the implementation of Normal Mapping technique in our system. We use a Bumpiness slider to let the user configure the level of bumpiness to be applied in Alyson as shown in Figure 50.

Bumpiness implementation in fragment shader of the Main Pass:

```
vec3 normal_color = texture2D(normalmap_texture, gl_TexCoord[1].xy).rgb;
vec3 normal_vec = normal_color * 2.0 - 1.0;
normal_vec = mix( vec3(0,0,1), normalize(normal_vec),bumpiness);
vec3 bitangent = cross(world_normal, tangent);
mat3 tbn = mat3(tangent, bitangent, world_normal);
vec3 bump_normal = normalize(tbn * normal_vec);
```

Figure 50: Alyson without bumpiness (left) and Alyson with bumpiness 0.8 (right)

6 System implementation

In our system, we have integrated in real-time and in a physically principled manner the effects that are produced when natural light interacts with the human skin. These effects have not been integrated before in skeletal-based, multi-surface, animated and deformable virtual characters. For this purpose, we use a virtual human model, on which we apply the effects already described in detail in previous chapters: *Ambient Occlusion, Shadow Mapping, Image-Based Lighting, Specular Surface Reflectance, Subsurface Scattering and Environment Mapping*. We have used PoserPro2012 to generate the human model (Alyson) in Collada format and 3ds Max to modify its geometry to make it match with our needs. Our system is based on OpenSceneGraph, which is an open-source, multi-pratform 3D Graphics toolkit that uses OpenGL as 3D rendering API. The C++ code is protable but for the purposes of this thesis is written on Visual Studio 9, because it offers a wide range of facilities that increase our programming productivity. Alyson's rigging, skinning and animation is created using 3ds Max 2011 along with OpenCollada plug-in, which allows us to import and export models in Collada format. The textures are generated from PoserPro2012 along with the model. Each different body part of the virtual human body uses its own texture. We used Photoshop to merge the textures into a single texture-atlas and 3ds Max to modify the texture coordinates that will match with the new single texture. We have handled the real-time modification parameters for each effect using AntTweakBar GUI toolkit [49]. In this section we will explain how our system works referring to these aspects.

6.1 OpenSceneGraph

To develop our system and implement the algorithms for realistic effects, we have used OpenSceneGraph [50]. It is an open source high performance 3D graphics toolkit, used by application developers in fields such as visual simulation, games, virtual reality, scientific visualization and modeling. The project started in 1999 by Dun Burns and Robert Osfield. It was initially part of Hang Gliding Simulator developed by Dun Burns. Together they open sourced the software and begun improving it by refactoring the existing codebase and embracing design patterns. Written entirely in Standard C++ and OpenGL it runs on all Windows platforms, OSX, GNU/Linux, IRIX, Solaris, HP-Ux, AIX and FreeBSD operating systems. The OpenSceneGraph is now well established as the world leading scene graph technology, used widely in the vis-sim, space, scientific, oil-gas, games and virtual reality industries.

6.2 Real-Time Character Rendering Implementation

To assemble our 3D scene we use the scene graph approach OSG provides us. Based on this approach, the 3D world is represented as a graph of nodes. Each object in that graph can be either a subgraph containing other nodes or a leaf node. Leaf nodes, which are called Geodes, are the

drawable objects that will be rendered in each frame with their own properties. The main objects that constitute our scene are the RenderEffect, the Skybox and the Interface.

First of all, in Figure 51 we present the Class Diagram of our code showing the connection between classes. ViewManager is the Component that initializes the application's viewport and the OpenGL Graphics Context. It handles the scene graph, which is the container of the objects that will be rendered. ViewManager also initializes the camera manipulator and adds the event handlers in the main application's window. Three Objects are rendered at any time. The first is the RenderEffect class, which contains the model, on which we apply the realistic effects. The second one is the SkyBox which simulates the surrounding space of our model. The third one is the Interface Window, which allows the user to handle the effect parameters. In this section we will discuss the purpose and how we use each one of these Objects.

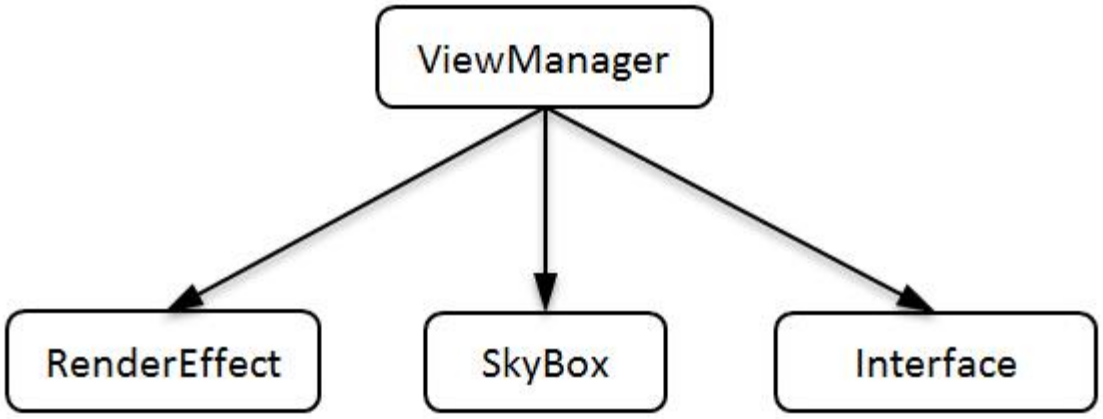

Figure 51: Simple Version of our system's scene graph

6.2.1 Render Effect

The RenderEffect component is responsible to simulate the behavior of the interaction between light and model. It has this name due to consistency reasons because it extends the Effect class provided by OSG. Each Effect class can contain any number of techniques. In our case, we use only the RenderTechnique, which is a 5-pass technique. In these 5 passes we implement the interaction between light and model. Each pass extends a class named RTTData (render-to-texture data), which contains one camera and the output texture. Each camera instead of rendering to the screen, it will render to a frame buffer object (FBO). OpenGL provides us with Frame Buffer Objects through an extension. Using FBOs we are able to do off-screen rendering, including rendering to a texture. We can capture images that would normally be drawn to the screen. They can be used to implement a large variety of image filters, and post-processing effects. Two are the main reasons for using FBOs:

1. The captured image can be passed through fragment shaders to add a post-processing effect such as blurring or bloom effect.
2. We can use many cameras to capture the same scene from different positions. This is how we can simulate televisions or mirrors in a virtual world.

In a multi-pass effect, in order to use the rendered image to the next pass, we must have a texture attached to the Frame Buffer Object. That way, after each frame, the texture will contain the rendered image. We have to be careful here, because the texture must use the same format as the buffer. Figure 52 depicts the Class Diagram of the RenderEffect Component, which is responsible to apply the effects we have implemented to the human model.

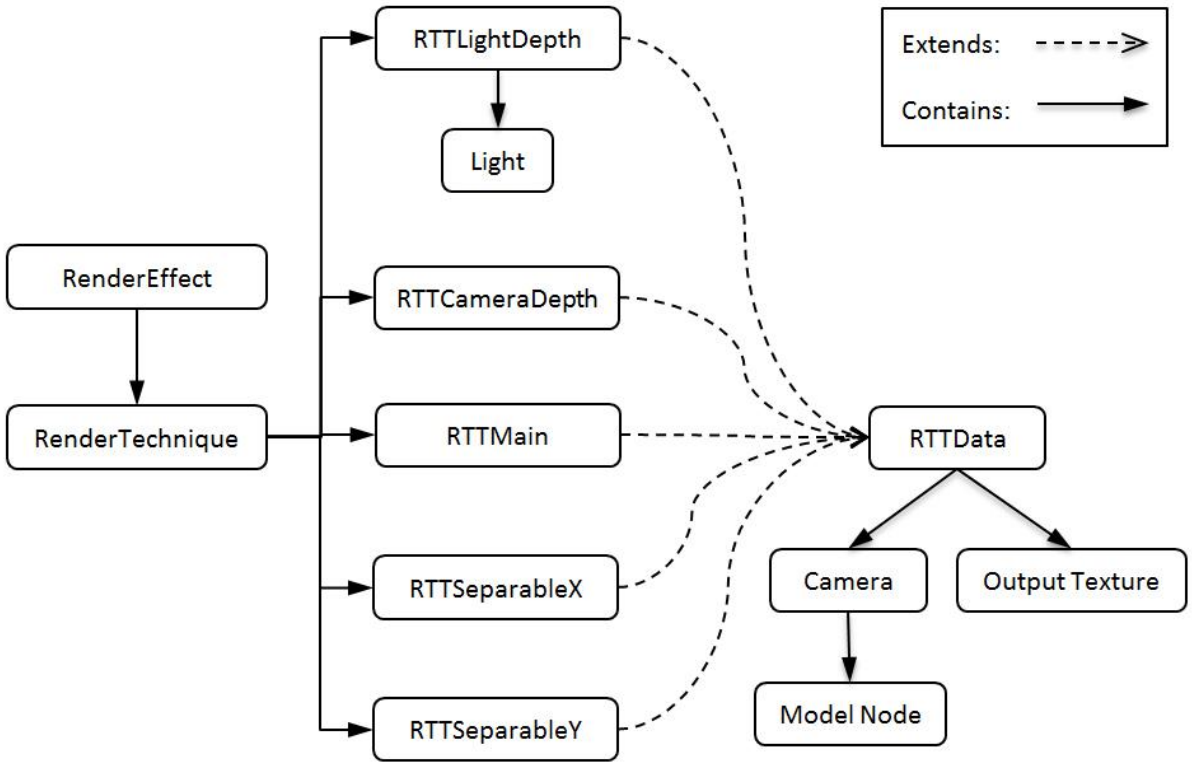

Figure 52: RenderEffect Class Diagram

The RenderTechnique contains the 5 passes that will be executed sequentially. For all of these passes we need 5 cameras and 4 output textures attached to frame buffer objects. The reason why we need fewer textures than the number of cameras is because the last camera will render to screen instead of the Frame Buffer Object. Each camera is behaved as a separate scene graph containing only the human model node that will be rendered. That means the 5 cameras contain a reference to the same human model. The Call Graph below shows the Inputs and Outputs of each pass, as well as the execution sequence.

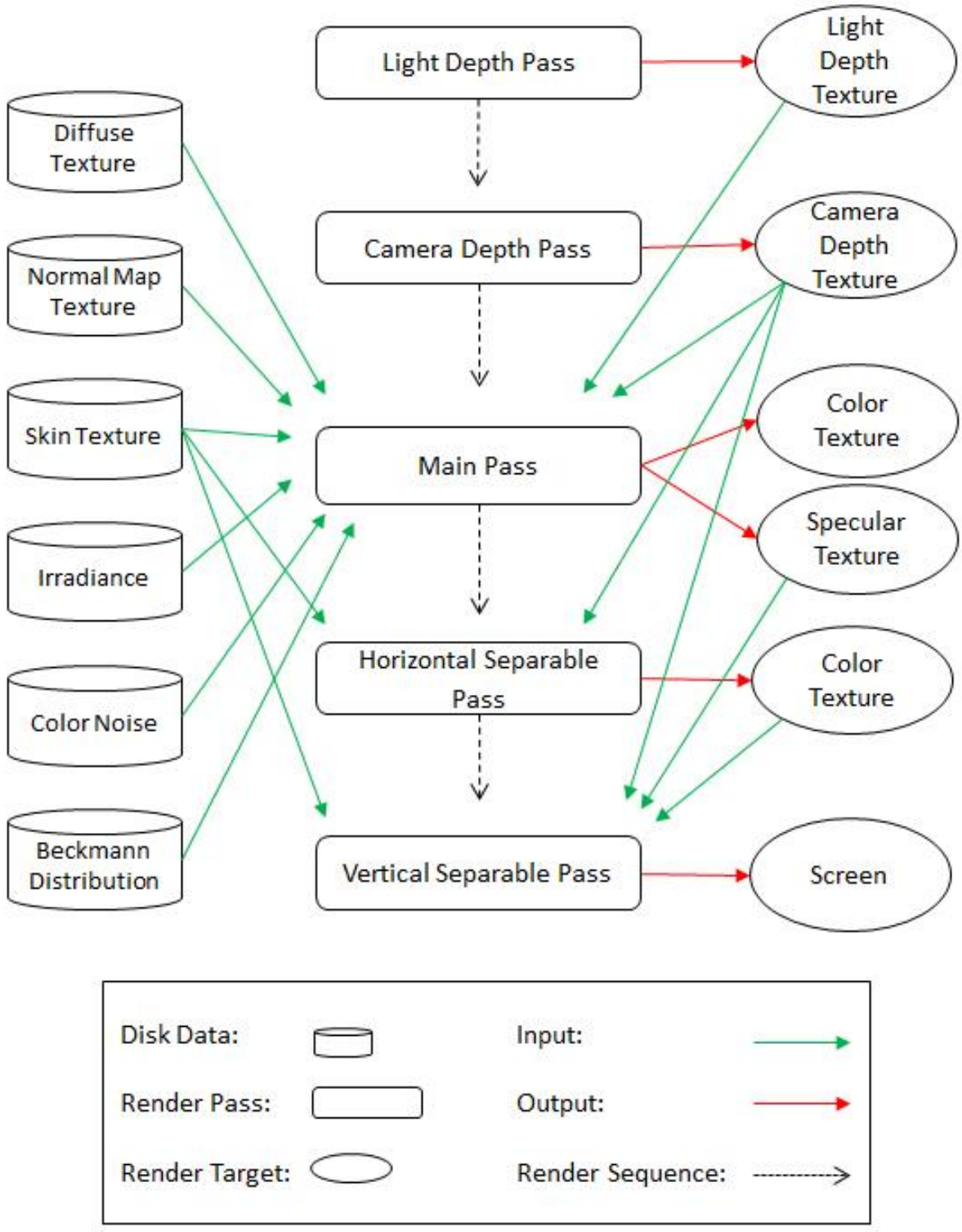

Figure 53: Rendering Technique's Call Graph

The purpose of each pass is described below:

1. **Light Depth Pass**: Stores the scene's depth from Light's point of view. The depth has to be linear and the render target will be a 24-bit depth buffer. The Camera of this pass will use the light's view and projection matrices. The texture result of this pass will be used in the main pass. The 2 effects that need this buffer are the shadow mapping and the light's transmission through thin surfaces. Figure 54 shows the depth buffer of a light positioned close to Alyson, lighting her head.

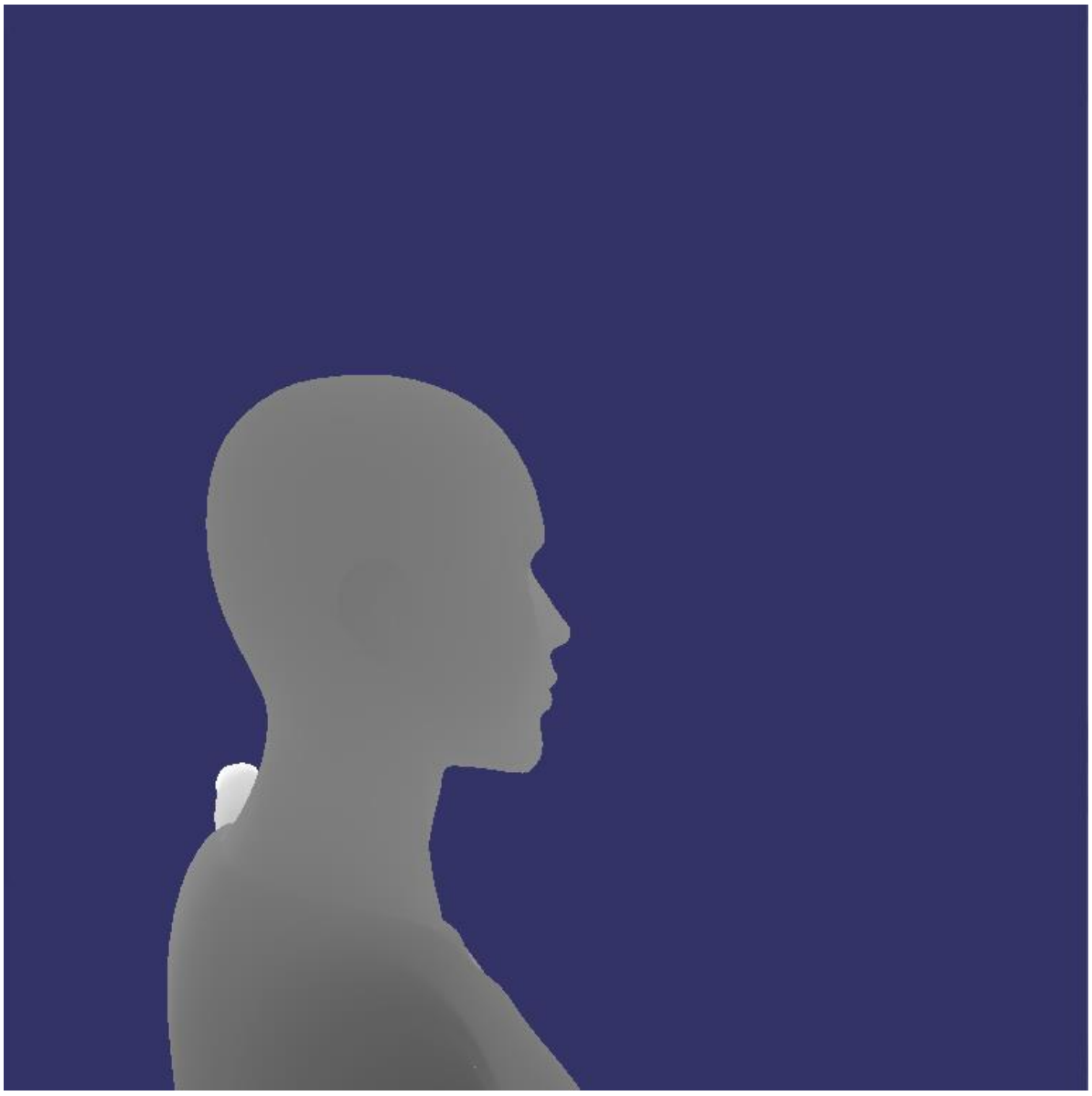

Figure 54: Captured image after Light Depth Pass

2. **User Camera Depth Pass**: Stores the scene's depth from User's point of view. It works in the same manner as Light Depth pass, which means the depth has to be linear and the render target will be a 24-bit depth buffer. The only difference is that the view and the projection matrices have to be retrieved from user's camera instead of light's camera. The texture result of this pass will be used in 2 effects. The first one is the Ambient Occlusion and we need scene's depth to calculate the occlusion value of each point. The second usage will be in subsurface scattering to avoid convoluting pixels with high depth difference. Figure 55 shows the depth buffer of user's camera. The reason why it seems black is because it is very close to the camera.

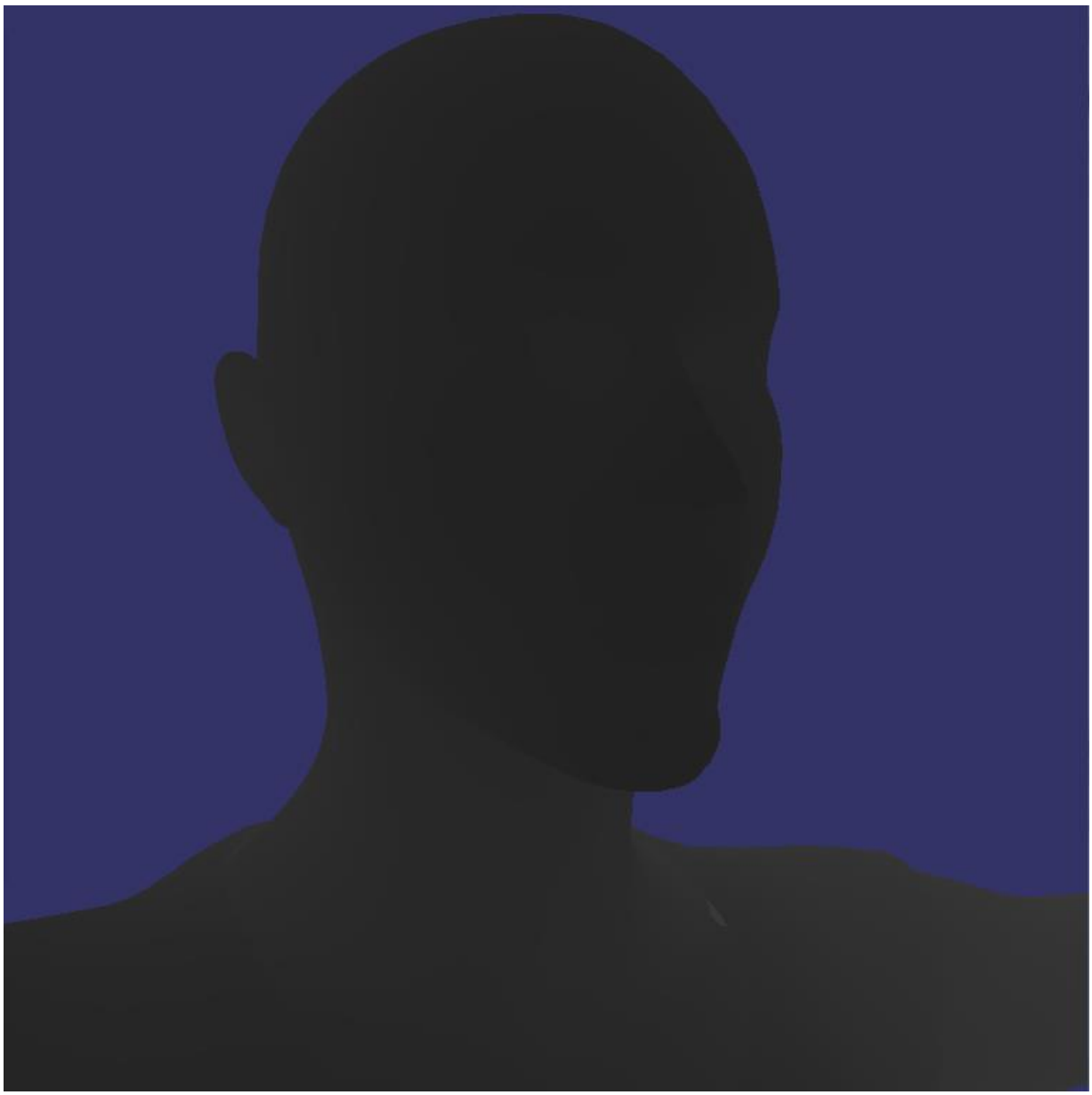

Figure 55: Captured image after Camera Depth Pass

3. **Main Pass**: As the name says, this is the main pass in which we apply most of the effects that are produced on interaction between light and model. First of all, the object is illuminated from the spot light according to Diffuse Lighting. To do that we use Alyson's diffuse texture. Apart from that, the object is also illuminated by the environment. We use an HDRI irradiance map retrieved from the disk, which approximates the light's diffusion in a room. A tone mapping function is also necessary to restrict the value retrieved from irradiance map between 0 and 1. The model's surface that isn't visible from the light will be shadowed. To apply the shadow mapping, we use the Light's depth map that was generated in a previous pass. We also take into account the Ambient Occlusion effect, which darkens those parts that are hidden by the neighboring geometries. To calculate it, we retrieve a color noise image from the disk and also the depth map of user's camera produced in the previous pass. Other than that we apply the specular reflection on model's surface based on Beckmann Distribution Function. Last but not least, we approximate the light's transmission through thin skin surfaces by using a diffusion profile, which describes the way the light attenuates, while propagating underneath the skin. To apply light's transmission, we use a Skin texture that indicates which part of surface is skin. For each of the above effects, we use normals computed by the Normal Mapping technique which uses Alyson's Normal Map Texture. We capture the rendered image of this pass and we store it in a Color Buffer to be used in the next pass.

Figure 56: Captured image after Main Pass

4. **Horizontal and Vertical Separable Passes**: We will discuss the last 2 passes together. Here, we approximate the subsurface scattering effect by convoluting the image produced by the previous pass. We use 17 jittered samples forming a Gaussian to blur the image first horizontally and then vertically. Because of Gaussian blur's properties, we can apply it to a two-dimensional image as two separate one-dimensional calculations. We have to blur the image first horizontally and use the result to blur it again vertically. Even though we need 2 passes instead of one, the calculation cost is much cheaper. If we use the two-dimensional convolution in the same pass, the calculation is performed in O(17 * 17 * ImageWidth * ImageHeight), which is more expensive than O(17 * ImageWidth * ImageHeight) + O(17 * ImageWidth * ImageHeight) that is needed to convolve it in 2 different passes. The Subsurface Scattering effect will be applied only in skin surfaces. That's why we need the Alyson's Skin Texture. The last thing that we have to take into account is to avoid blurring those pixels that have high depth difference and for that we need User's Camera depth map produces in a previous pass.

Figure 57: Captured image after both separable passes

6.2.2 SkyBox

The SkyBox component is responsible to simulate the environment. It is a cube, in which the human model exists and its center is always the position of user's camera. No matter where we move with the camera, we will always be inside that environment. The Skybox is a node in the scene graph containing 6 quads, one for each side of the cube. To create the cube, an Environment Map in cross format is required. The reason why we use cross format is because we can easily extract the 6 sides of the cube by picking subimages. The environment map has to depict the surrounding space as an omnidirectional image.

To create the skybox node, first of all we create 6 quads positioned correctly to form a cube. These quads have to be far enough to cover the model completely. Each quad uses a different texture, which is extracted from the environment map. In order for the cube to create the surrounding space, the environment map must depict the sides correctly. The following image shows how the sides must be placed to form the cross format image. To depict the skybox in the screen, we render the 6 textured quads. In our system we use HDRI images as environment maps. That means we have to use the same tone mapping function we used in irradiance map.

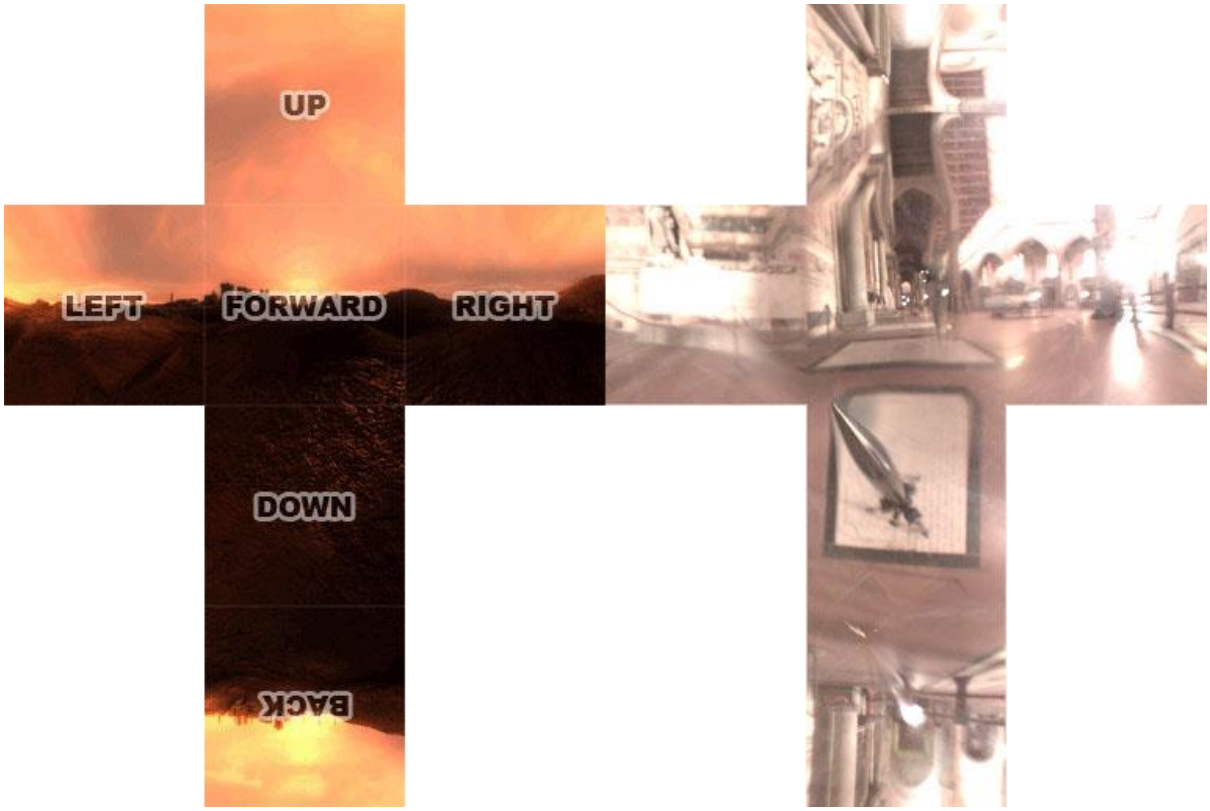

Figure 58: Cross Format Environment Map

Figure 59: Alyson inside environment

6.2.3 Interface Window

AntTweakBar [49] is a small and easy-to-use C/C++ library that allows programmers to quickly add a light and intuitive graphical user interface into graphic applications based on OpenGL, DirectX 9, DirectX 10 or DirectX 11 to interactively tweak parameters on-screen. As long as OpenSceneGraph uses OpenGL as 3D Rendering API, AntTweakBar can be embedded in our system. The reason why we use it over other GUI toolkits is because not only it can be installed easily, but also it has a great configurable design. Furthermore, the fact that we have access to its code allowing us to modify it any time is a huge plus. The main purpose of AntTweakBar in our system is to modify the parameters of the effects we apply to the human model. So, we use uniforms that are linked with AntTweakBar

controls, giving the user the capability to easily adjust the result of the effects. The purpose of each control is the following:

- **Camera**
  - Camera Enabled: Enables or disables camera's movement. If the camera is disabled, it is locked to its current position.
  - Camera View: Moves the camera to a predefined position chosen from a set of standard positions. It is used to extract results with the camera in a specific position.
  - Use Animation: This control specifies whether or not we want to use the animated positions, normals or tangents instead of the static ones.

- **Screen Space Ambient Occlusion**
  - Samples: Specifies the number of samples to be used for the Ambient Occlusion calculation.
  - Occlusion Intensity: Modifies the intensity of the Ambient Occlusion effect. If its value is 0, the human model will not be affected by Ambient Occlusion.
  - Hemisphere Radius: Modifies the radius of the hemisphere from which we retrieve the samples.

- **Skin**
  - Scale Transmittance: Specifies the intensity of light while it travels through thin surfaces.
  - Exposure: This parameter determines the portion of light the human model receives from the environment.
  - Roughness: Defines how rough the skin surface is. It is used as the Y texture coordinate for the Beckmann Distribution Image.
  - Specular Fresnel: This control is used as interpolation value between 0.25 and Fresnel reflection.
  - Specular Intensity: Modifies the intensity of the specular reflection.
  - Bumpiness: This parameter is used as interpolation value between Vertex' static normal and the normal retrieved from Normal Map.
  - Scale Convolution: Specifies the distance the light's travels in Subsurface Scattering effect.
  - Shrink Bias: Determines how shrinked each vertex will be towards its normal. We need this parameter to avoid artifacts appeared in Subsurface Scattering.

- **Shadow Mapping**
  - Depth Bias: This parameter is added to the projected depth value of each pixel, which is produced based on light's view and projection matrix. It is used to fix the precision problems of the depth buffer.
  - Shadow Radius: With this control we can adjust the distance of the neighboring samples that are used for the shadow mapping technique.

- **Light**
  - Color: Modifies light's intensity in a range between 0 and 1.
  - X, Y, Z: Using these 3 controls we can change light's position.
  - Position: Moves the light to a predefined position chosen from a set of standard positions.
  - Look At: Similar with position control, but in this one we specify light's "look at" direction. The direction of the light can be any of the 3 axes.

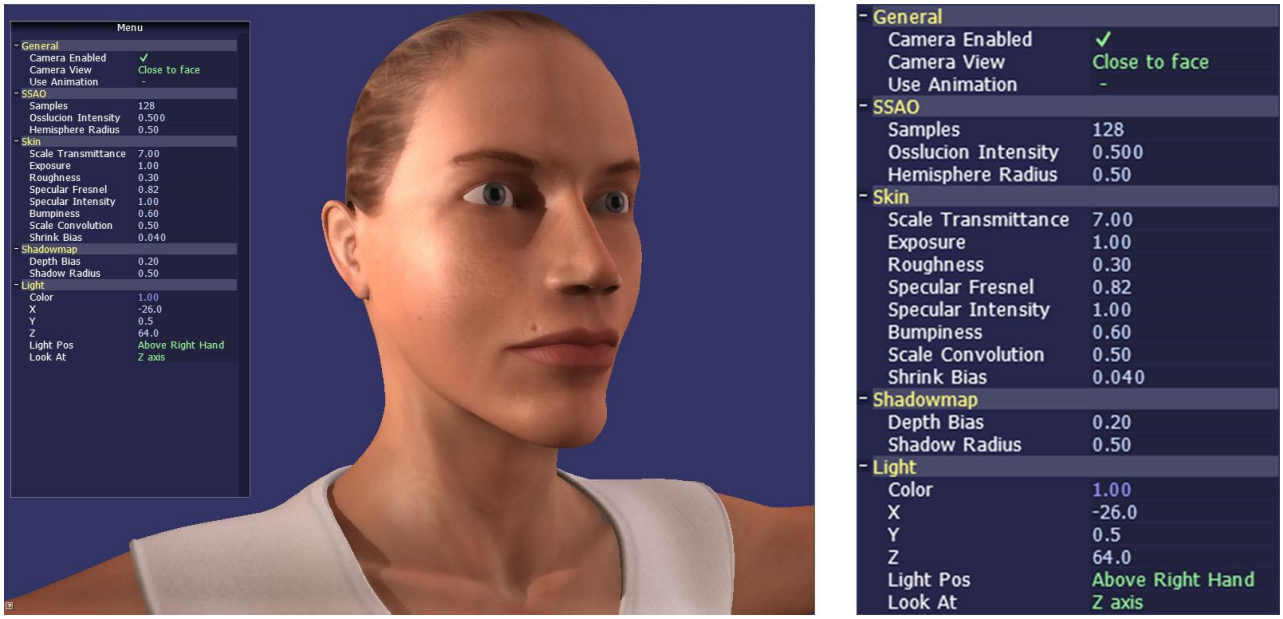

Figure 60: AntTweakBar

6.3 GLSL Debug Methods

To write the shaders for our effects' implementation was by far the hardest part of this project. The reason is because there are no obvious ways to debug the code. In most cases we are not sure whether or not the visual results are correct. We can't just assume their correctness if they are close to the expected ones. The most important difference is the fact that GLSL instead of CPU. When we use Visual Studio to write a program that is supposed to run on CPU, we can either use the step-by-step debugging or even print the values for our variables. In GLSL we don't have this capability. In this section we will discuss the methods we used to debug the GLSL code in our system.

6.3.1 Test Depth Precision

As we have mentioned before, when we render to textures, we cannot preserve the precision that we have in the float values used in the GLSL code. Usually each pixel in a depth buffer

is 8 bit. In most of cases, we try to store in these 8 bits values that are 32 bits. We have mentioned in a previous section that to solve this problem we use depth buffers with higher precision (24 or 32 bits). So, when we try to retrieve a pixel value from any high-precision depth buffer, we have to be sure that this value preserves its precision. To test this case, we have created a method that checks which pixels have the same depths. In a low precision buffer there will be more pixels with equal depths than in a high precision buffer. Each time a fragment shader is executed, we check if a pixel has equal depth value with a pixel in fixed position. In our case the fixed position will be the middle of the screen (width / 2 and height / 2). Figure 61 shows the pixels with the same depth with the fixed position, for 8-bit and 16-bit precision.

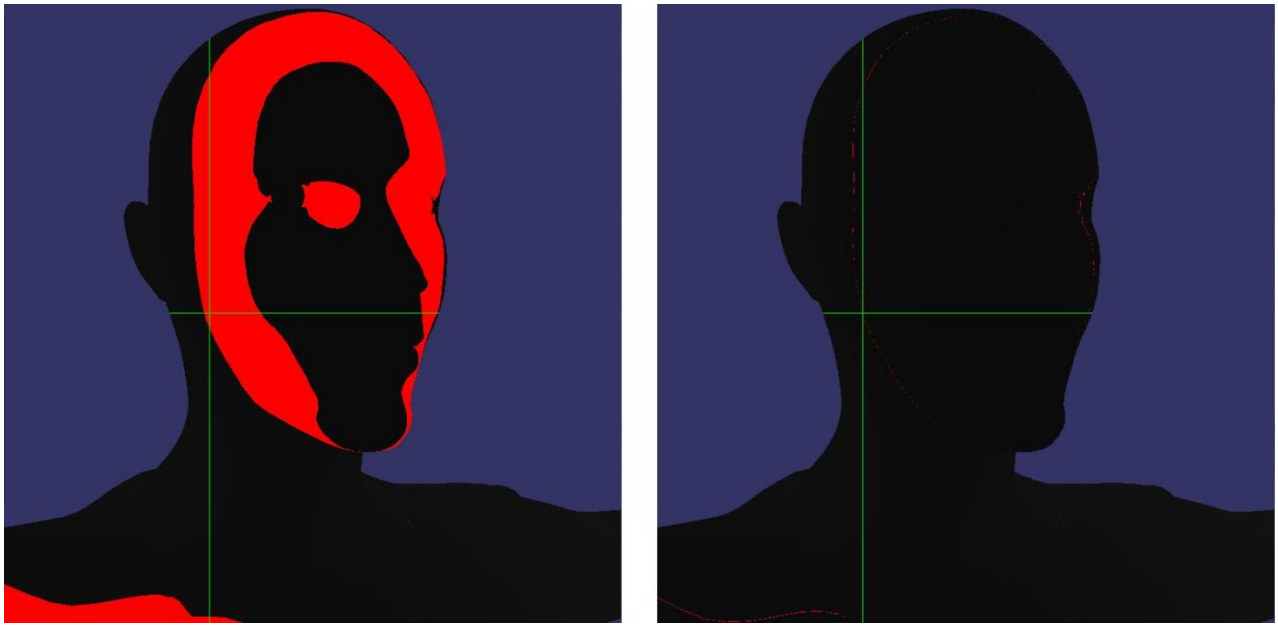

Figure 61: 8-bits (left) and 16-bit depth buffer precision. The red area indicates the pixels that have the same depth.

6.3.2 Test Depth Difference

In Subsurface Scattering the light travels beneath the skin before it completely attenuates after some distance. To simulate this effect we blur neighboring samples according to diffusion profiles. The problem here is that 2 pixels that are close in screen space might not be close in 3D world. We want to use for blurring only those samples that are close in 3D world because that means the light can reach that distance. So, for each neighboring sample we use, we compare its depth value with the depth value of the main pixel. If it higher that a threshold, it won't be taken into account for the convolution. Figure 62 shows how we apply this test method on the ear area, where 2 adjacent pixels might have large depth difference. The pixels in the red area don't exceed the threshold value.

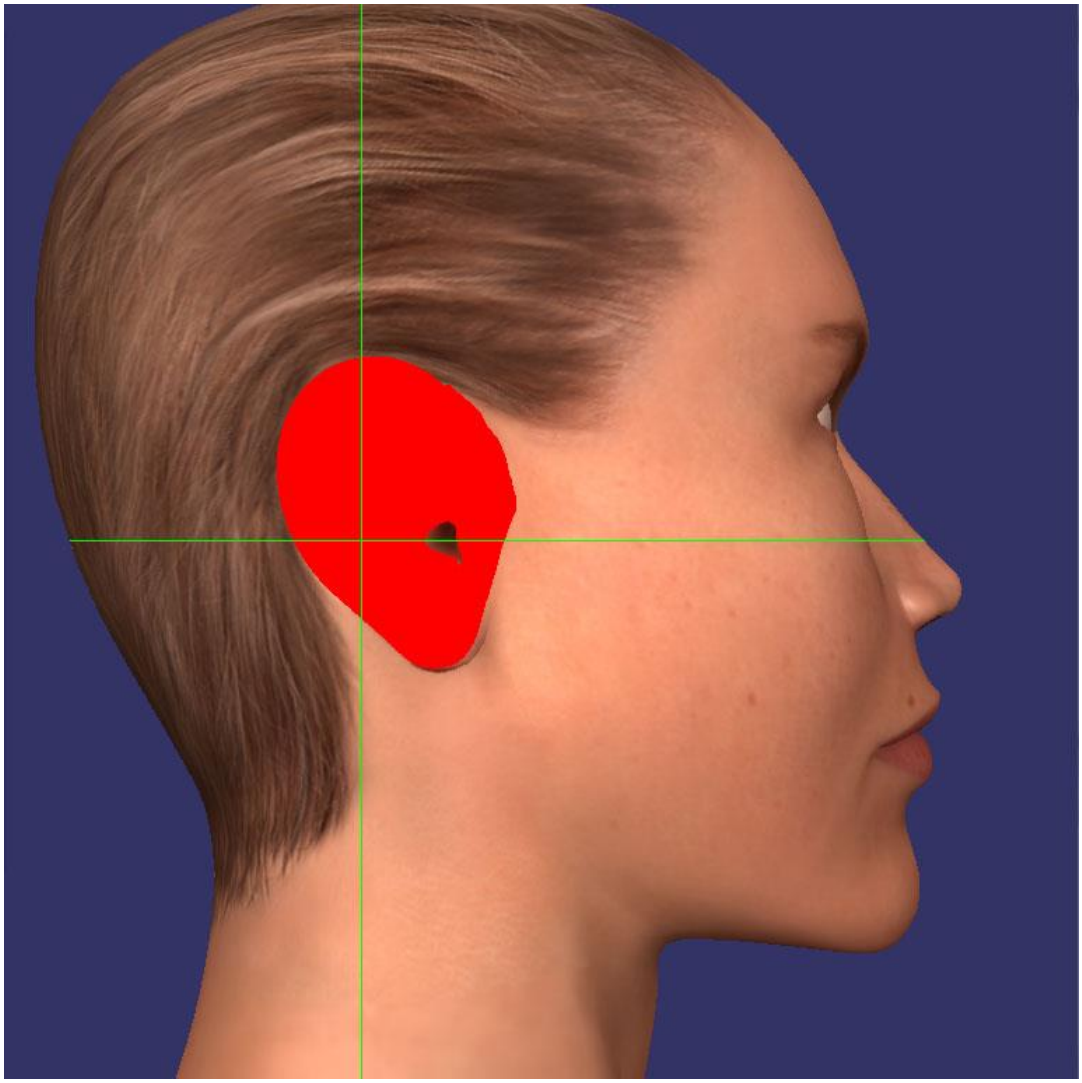

Figure 62: The depth difference in the red area doesn't exceed the threshold value.

6.3.3 Test Convolution Width

The last debug method is used in Subsurface scattering convolution as well. As we have mentioned before, the convolution is calculated in Screen Space and we want to be sure that the samples we use are close to each other in the 3D world. But the 3D world distance between 2 pixels might differ according to the distance between the user camera and the model. The closer the camera comes to the model, the 3D world distance between 2 pixels becomes shorter. So during the blurring, we need to take into account the distance between the camera and the model to avoid blurring pixels that even though they are close in screen space level, they are far in 3D world. In Figure 63 below we show how the blurring area (red) is being modified according to the distance between camera and model. The red circle covers the same area in 3D world, independently the distance between the camera and the model.

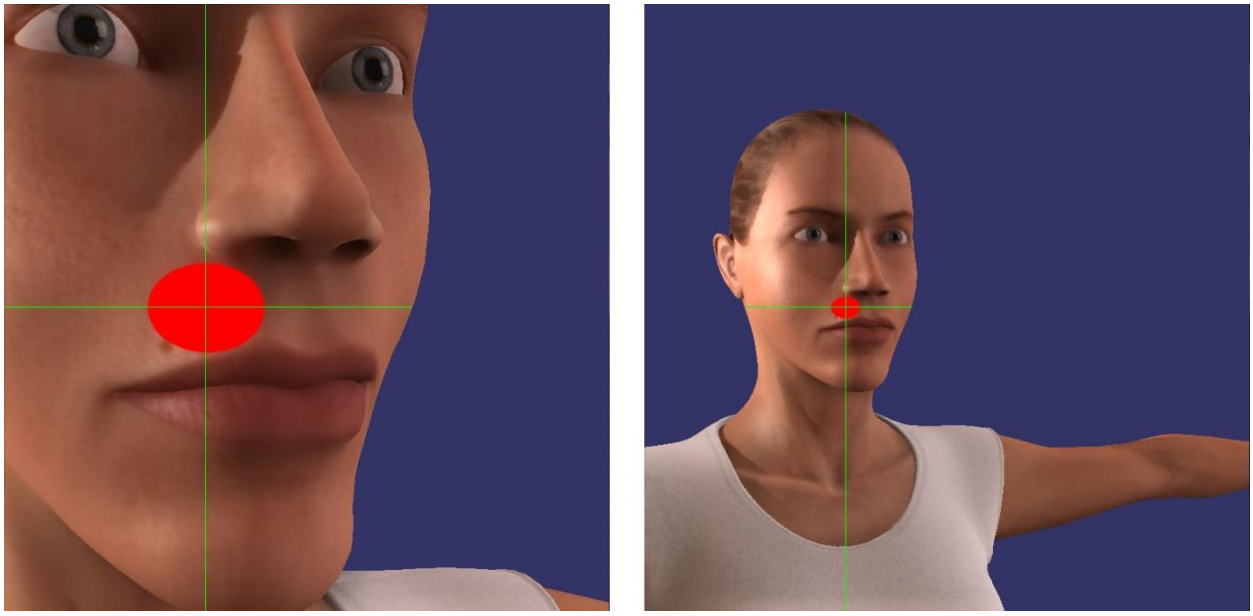

Figure 63: No matter if we are close or far from the model, the convolution will cover the same area.

6.4 Animation Implementation

Animated Models are important in a 3D Virtual World as they enhance scene's realism. In our system, each algorithm we have implemented runs on Alyson. Even though initially she was a static model as produced by PoserPro 2012 [51], we managed to convert her into a skeletal-based deformable animated model. What we had to do is to take care of the rigging, the skinning and to run the animation in OpenSceneGraph using shaders. In this section we will discuss the procedure for adding animation to Alyson and the issues we had to deal with.

Starting with PoserPro, we extracted Alyson model as Collada format so we can import it in 3ds max with the OpenCollada Plugin. First of all we had to take care of the rigging, which refers to the creation of bones that Alyson will use for animation. So using 3ds max, we created a biped and we configured it to fit inside Alyson's body. We also had to reduce the number of bones, she uses due to limitations in the vertex shader attributes (we will discuss these issues later). After that, we had to bind the Alyson's mesh on the bones. This process is called skinning and it's responsible to associate the model vertices with the bones. We have to bind each vertex with at least one bone. In case a vertex is animated according to two or more bones, we have to define how much of each bone's animation affects the vertex. We do that by adjusting the weight values. Each vertex that is affected by any bone contains a weight value which refers to that bone. So after completing the rigging and the skinning, we export Alyson as Collada Format.

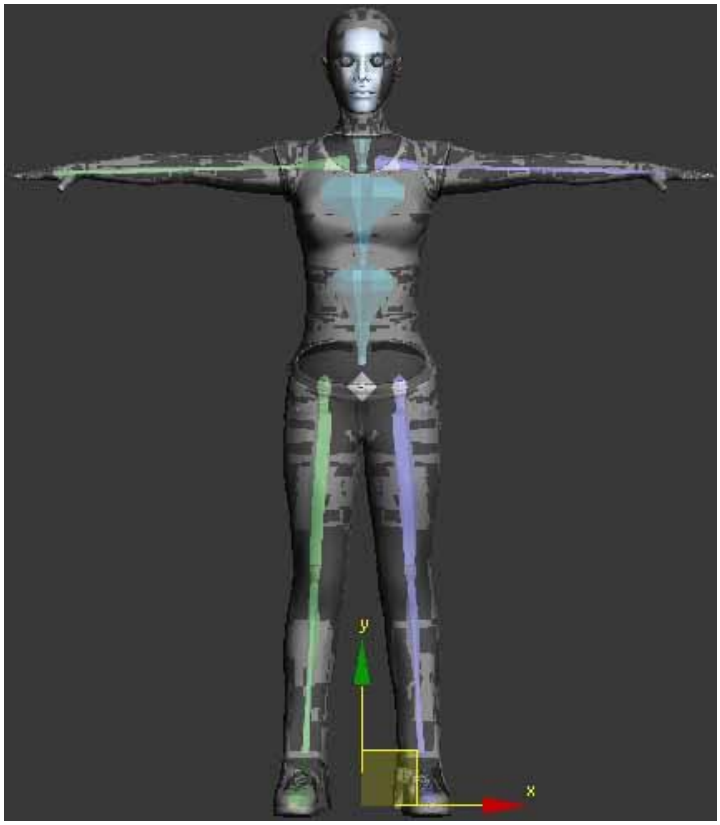

Figure 64: Rigging on Alyson

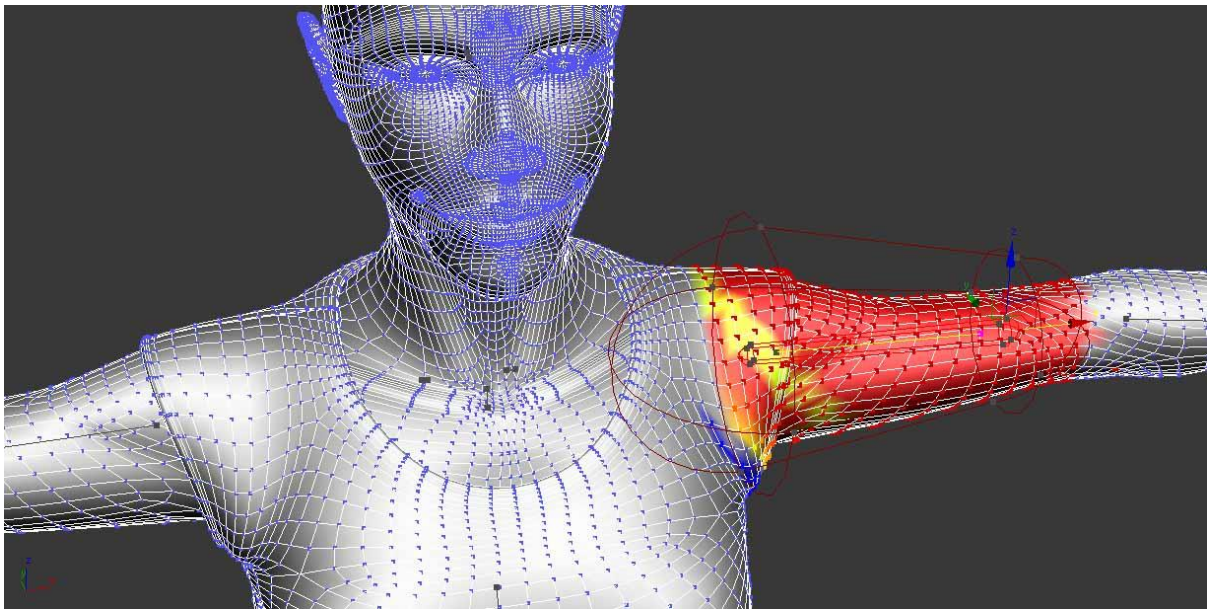

Figure 65: Skinning on Alyson. The colors around the arm indicate the weight values of the corresponding vertices to the selected bone.

OpenScreneGraph supports animation by using a specific set of classes. Some of these classes are Bone, Skeleton, Animation, RigGeometry, RigTransform and BasicAnimationManager.

The model node needs to use all of these classes in order for the animation to be able to run. The good thing is that when we load a model in Collada format that supports animation, OpenSceneGraph takes care of creating these classes be loading the data. The only thing we have to do here manually is to set the correct update callback in the model that allows us to run or stop the animation.

The model is being animated by moving its bones. To do that there we have to retrieve from the bones the matrices that handle the animation and multiply them with the model vertices. As mentioned in skinning, each vertex is attached to one or more bones. The component that is responsible for the matrix multiplication is the RigTransform. OpenScreneGraph has two implementations of this class. The first one is the RigTransformSoftware class, which is loaded by default when the model supports skeletal-based animation. This class takes care of the matrix multiplication between the matrices stored in bones and the model vertices. The fact that these matrices multiplications run on CPU makes the animation on high-detailed models unbearably slow. There is no way a model with more than 100 or 200 thousands vertices to run in real time. Using this implementation, Alyson's animation runs with only 5 frames per second, which is far away from real time.

What we need to do in order to solve this problem is to apply the matrix multiplications on GPU. But in order to be able to do that we need to use the second implementation of RigTransform, which is RigTransformHardware. This class passes in the shader program a matrix palette, which is a uniform array that contains all the matrices retrieved from the bones. Apart from that, this class uses vertex attributes to indicate which matrices from the palette each vertex needs, as well as the weights these matrices affect the vertex. These attributes will be called Bone Attributes and in each one of them we store data for 2 bones (1 matrix index and 1 weight for each bone). Now, during each vertex shader execution, we parse the Bone Attributes to retrieve the matrices each vertex needs. After obtaining the matrices that correspond to the vertex' animation, we apply the matrix multiplication on the other vertex attributes that need to be animated. In our case, the attributes that have to be animated along with the model's geometry are the position, the normal and the tangent vectors.

Even though the class RigTransformHardware offers the functionality we need, it is not enough for to render high-detailed models. The problem lies in the fact that this class uses only 4 Bone Attributes to store matrix indices and weights. That means each vertex cannot rely on more than 8 bones, because on each Bone Attribute we store data for 2 bones. So, in a high-detailed, there will be bones that will fail to affect the vertices they are connected with. We had to make 2 actions to fix this problem. The first one was to implement our own RigTransformHardware class, in such a way that it will be able to use more attributes. At this point we have to mention that we are limited by the hardware that provides us with only 16 attributes. So, as long as we already need 5 attributes for position, color, normal, tangent and texture coordinates, we can allow our RigTransformHardware implementation to use only 11 attributes, which corresponds to 22 bones. The second action we had to make to fix this problem with the number of attributes, was to lower our model needs for bones. That totally affected the animation, which in some parts appears a bit broken (Figure 66). The code below is a part of each vertex shader in which we multiply the position, the normal and the tangent which a matrix that was retrieved from the matrix palette.

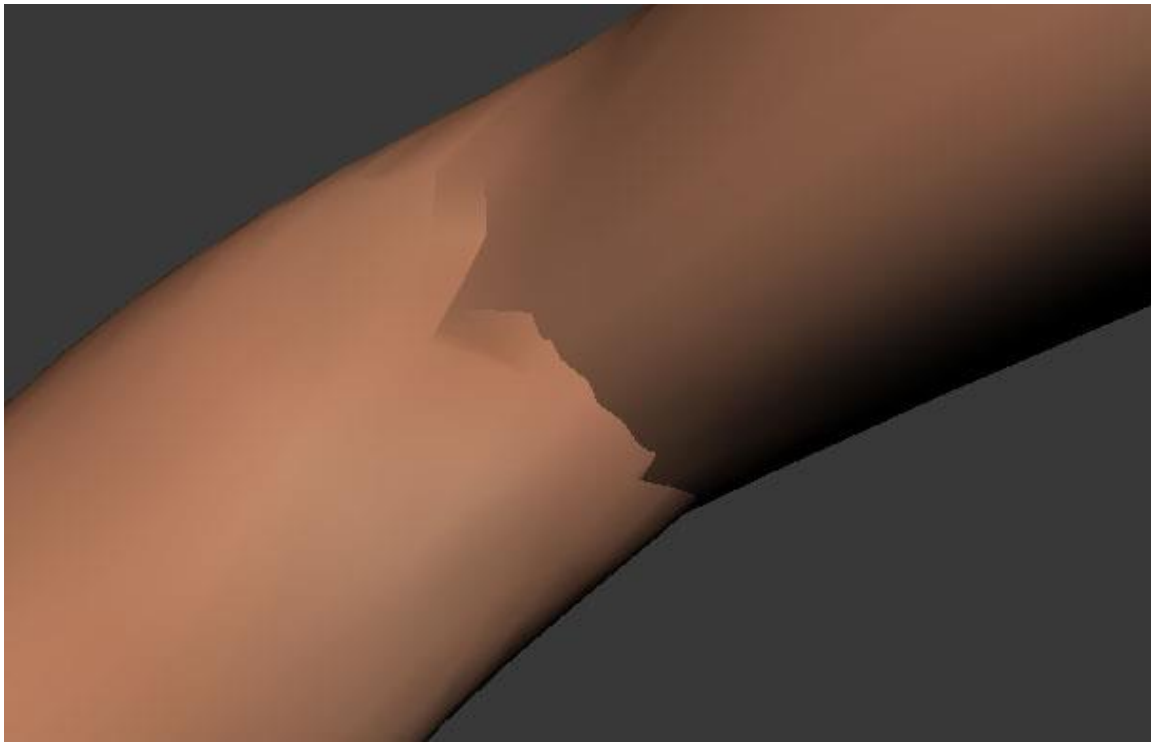

Figure 66: Slightly broken skin

Animation in Vertex Shader:

```
int matrixIndex;
float matrixWeight;
for (int i = 0; i < 2; i++)     {
   matrixIndex =  int(boneWeight[0]);
   matrixWeight = boneWeight[1];
   mat4 matrix = matrixPalette[matrixIndex];
   animated_position += matrixWeight * (matrix * position_attribute);
   mat3 normal_tangent_matrix = mat3(matrix);
   animated_normal +=
   matrixWeight * (normal_tangent_matrix * normal_attribute );
   animated_tangent +=
   matrixWeight *(normal_tangent_matrix * tangent_attribute );
   boneWeight = boneWeight.zwxy;
}
```

7 Results and Conclusions

7.1 Comparison with Offline Ground Truth

The effects that we support in our project are implemented in Screen Space. As we have mentioned, these Screen Space techniques run faster than the respective ray tracing ones, making them able to be used in real time rendering systems. But how much can the Screen Space techniques match the realism produced by ray tracing techniques? No matter how fast a rendering technique is, it's of no use if the visual results are not correct. In order to test the visual accuracy of our effects, we have to compare them with Ground Truth images. For this purpose we use 3D Studio Max to create our scene with the same camera and light position. To render the scene in 3ds max we use the offline Mental Ray renderer, because it supports ray tracing and the generated images are depicted with great realism. In this section, we compare the images produced by our effect with those produced by the Mental Ray Renderer of 3ds max. We consider the image generated by 3ds max to be the Ground Truth. That way we will be able to evaluate our results based on rendering time and accuracy criteria. We picked the most challenging effects in our system based on the number of passes and the number of samples needed by each one of them.

7.1.1 Shadow Mapping

To produce shadows in our system we use a 2-pass technique called Shadow Mapping. This will be the first effect we compare with Ground Truth. Starting with a scene in 3ds max containing only the human body, we place a light in exactly the same position we do in our system. As shows in Figure 67, the light is position to the right of Alyson. We also use the same type of light, which is spot light. We enable the producing of shadows and set the shadow type to Ray Traced. To compare effectively the results with our system, we need to "watch" the scene from the same point of view. So we create a camera object in 3ds max using the same position, direction, field of view, near and far plane with the main camera of our system. We choose to use that camera as viewport and finally we proceed with the rendering.

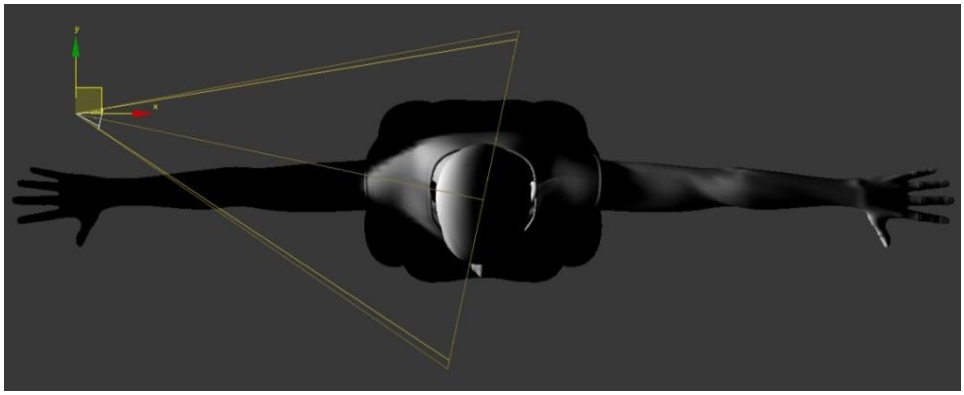

Figure 67: Light Position for Shadow Mapping

Figure 68 shows the comparison between the results of our method and 3ds max. The 2 images are almost identical meaning that we have achieved our purpose to produce a shadowed human body with accuracy close to the ray tracing one. The main difference in ray tracing algorithm is the fact that light can reach even in surfaces that are not directly visible to it. Another thing that we can notice is that our method produces high-frequency details. For example, if you look closely the neck (Figure 69), you will notice some surfaces that are half exposed to light, are appeared shadowed. We tried to remove those high-frequencies by gathering shadowing information from the surrounding pixels as well. The problem here was that if we increase the sampling radius too much, even those pixels that are directly exposed to light, would appear partially in shadow. Other than that our system produces very accurate results that are very close to the real world. With that said, someone would be suspect with the fact that these 2 methods would have almost the same rendering time as well. Even though we have rendered these 2 images in the same PC, our method runs 600 times faster. The 3ds max takes 2 seconds to produce 1 frame, while our system runs with 310 frames per second (3 milliseconds per frame) in this 2-pass technique. With that said, shadow mapping technique is clearly preferable in a 3D real-time rendering system.

Figure 68: Shadow Mapping comparison between Screen Space (left) and Ray Tracing (right)

Figure 69: Alyson Neck for Screen Space (left) and Ray Tracing (right)

7.1.2 Ambient Occlusion

Ambient Occlusion is a global illumination phenomenon that is dependent on the neighboring geometries instead of the light sources. This effect measures the accessibility and shades each point based on how easy it is for a ray to hit on that point. To measure the occlusion value of a point using Ray Tracing, we cast a number of rays in different directions. The more rays hit on neighboring geometries, the greater occlusion value this point gets. In Screen-Space implementation, this effect is calculated by sampling neighboring pixels and comparing their depth values with the main pixel. The occlusion value here is increased by the number of sample pixels that are closer to the camera than the main pixel. Again, we will use 3ds max to generate the ground truth and compare it with our system.

To produce the Ambient Occlusion in 3ds max, we have to create a mental ray material and apply it to our object. There are a lot of parameters that can be adjusted here in order to increase the effect's quality. First of all, we can increase the number of samples used for every single point. These samples will be used to cast rays in different directions. Apart from that, we can also configure the max distance that will be taken into account for each ray casted. If a ray passes that distance, which means it doesn't collide to other geometries, then that ray will not contribute to occlusion. We might as well disable the light source we used for shadow mapping, because this technique is not affected by lighting. To observe the results from the same point of view as we did in shadow mapping, we will use the same camera to render our images.

We have implemented Ambient Occlusion in a 2-pass technique, which requires the depth value for each pixel used as sample. Figure 70 shows the 2 rendering results in 128 samples from both our system and 3ds max. The first thing we notice in both images is that those corners whose accessibility is limited by neighboring geometries appear black. This will not change even if we place light sources illuminating directly the black surfaces. Focusing in the image produced by our system,

we can see that some surfaces appear to be grey, even if they are wide open to environment. This happens because the head is curved and even if we use a normal oriented hemisphere for each rendered point, there are still some surrounding pixels with less depth than the main pixel. But even though our image is not perfect, the visual result is very close to the ground truth. This method is used by a wide range of game engines or other 3D real-time rendering systems to simulate the ambient occlusion effect. But not only the visual result similar to ground truth, our method manages to run 300-600 times faster depending on the number of samples we use. The following table compares the rendering time required of Screen Space and Ray Tracing implementations for different number of samples.

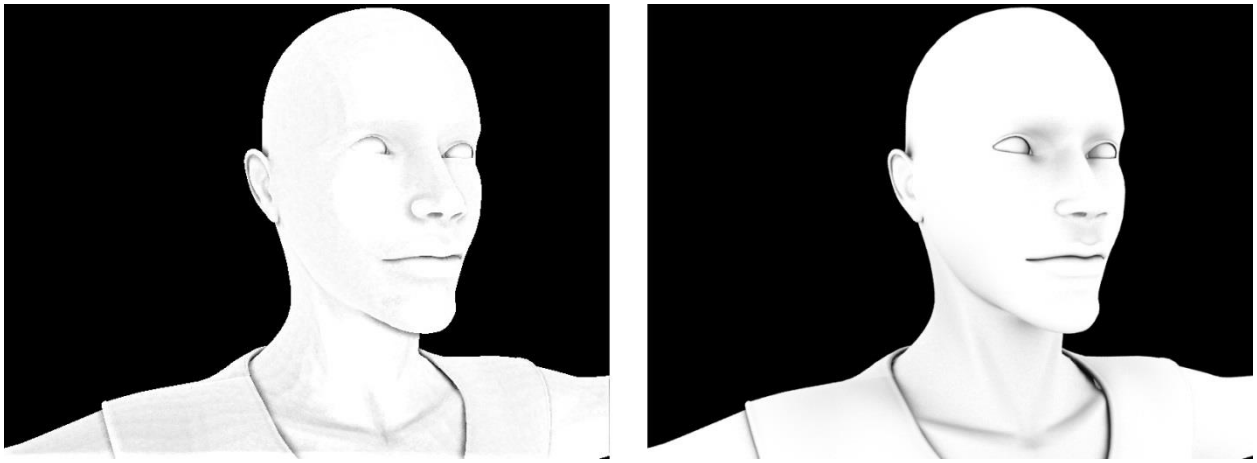

Figure 70: Ambient Occlusion comparison for Screen Space (left) and Ray Tracing (right)

Samples Algorithm	16	32	64	128	256
Screen Space	3ms (330 fps)	3.7ms (270 fps)	5.2ms (190 fps)	8.3ms (120 fps)	14.2ms (70 fps)
Ray Tracing	1s	2s	3s	5s	9s

Table 3: Ambient Occlusion Rendering Time comparison between Screen Space and Ray Tracing

7.1.3 Subsurface scattering

Subsurface Scattering is the effect that describes the light's behavior after it enters the human skin. This phenomenon is taking place because skin is a translucent material and allows the light to travel through. While it propagates underneath the skin surface, it is either absorbed or scattered many times before it exits a neighboring area. In our system, we have implemented the subsurface scattering effect to increase the realism of our human model. In this section, 2 experiments will take place. In the first one, we will compare the images produced by our system

and 3ds max for SSS effect in thick surfaces, where the light is either absorbed completely or it exits from a neighboring area. In the second experiment we will study this effect for thin surfaces where the light passes through and is emitted from the other side.

To implement the subsurface scattering effect in our system, we need 5 passes. In the first two we store information in textures that are going to be used in the main pass, which is the third one. In the fourth pass we apply a horizontal convolution on image in order to blur the high-frequencies and in the fifth pass we blur the image vertically. This effect can be produced in 3ds max by using a mental ray material. There are 2 materials that simulate the SSS, subsurface scattering fast material and subsurface scattering fast skin. In the first one, the object is considered to consist of only 1 layer of translucent material and in the second one; multiple layers can be used as happens with skin. We will use the second material. In the following image we compare the SSS in our system and in 3ds max. To understand how the SSS affects the skin, we have to also present how the model was. We can see that the visual results between our method and 3ds max are very similar. Both methods are blurring the image with a red tone color. Again, we have a huge difference in rendering speed. Our system produces this result in 6ms (160 fps) and 3ds max in 2 seconds.

Figure 71: Alyson rendered by 3ds max without SSS (left) and with SSS (right)

Figure 72: Alyson rendered by our System without SSS (left) and with SSS (right)

In the second experiment, we will compare the results for light's transmission through thin body parts. To conduct this experiment, we have to change position of the light source and place it behind the head to observe how light passes through ear. The light won't be able to be absorbed completely and it will be emitted from the other side with different color due to its attenuation. In Figure 73 we compare this behavior between our system and 3ds max. It is noticeable that the results look alike. The red color is the result of light's attenuation, which is approximated by the diffusion profile that we use. Skin tends to absorb blue and green colors more than red. This image is rendered by 3ds max in 8 seconds, while we need only 4ms (250 fps).

Figure 73: Light's transmission through thin skin as rendered by our method (left) and by 3dsmax using ray tracing (right)

7.2 Alyson as rendered with our integrated rendering framework

In Figure 74, Figure 75 and Figure 76 you can see the rendering results of the physically-principled effects we have implemented as applied to the virtual character used by our rendering framework. In each one of these three cases, we render Alyson employing all of the effects we have analyzed thoroughly in this thesis: *Spot-based Lighting, Ambient Occlusion, Shadow Mapping, Image-Based Lighting, Specular Surface Reflectance, Subsurface Scattering and Environment Mapping*.

Figure 74: Alyson Head

Figure 75: Alyson Full Body

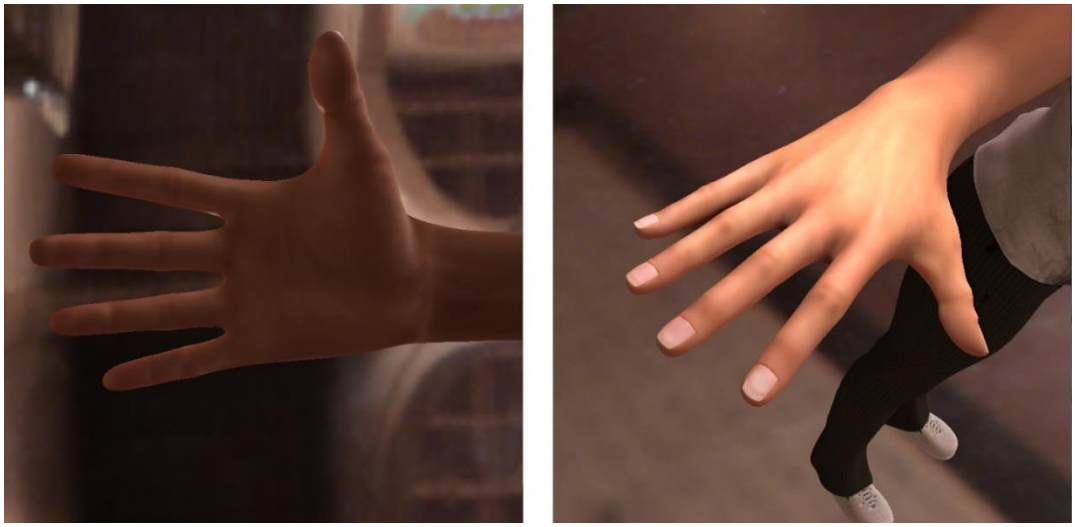

Figure 76: Alyson Hand

7.3 Our rendering framework integrated in other 3D rendering systems

In this rendering framework, we have implemented a multi-pass technique that applies a number of effects on a human animated model with great speed and high accuracy. But at this point, we are wondering whether or not this technique is able to be adopted by any other virtual world. If the algorithms we have implemented cannot be supported by other systems, then they lose their value no matter the speed or accuracy. So, we have created 3 groups of conventions that if fulfilled, the rendering technique can be used in any game engine or other 3D rendering system. Figure 77 shows these conventions.

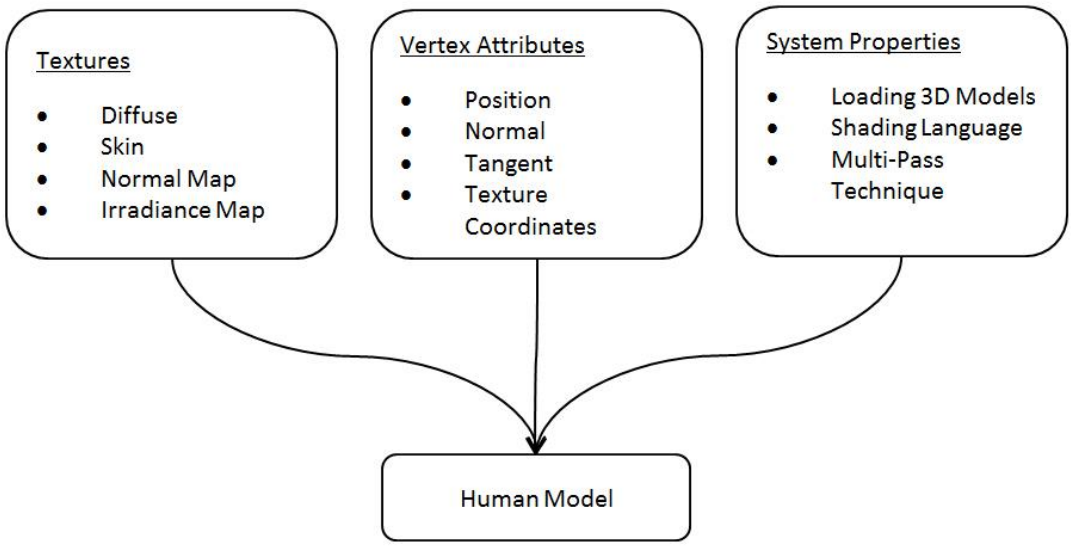

Figure 77: Conventions required by rendering technique in order to be supported by other systems.

First of all, as long as the depicted model is a human body, we need a set to textures describing its appearance. The diffuse texture contains the model's color, the skin texture shows which part of human body is skin and the normal map shows how the normals will be modified to apply bumpiness on the model. Apart from that, an Irradiance Map is required, which indicates how the light is diffused inside the room. Other than textures, we also need a set of vertex attributes. They are the data that differ from vertex to vertex. So, our algorithms need the position, the normal vector, the tangent vector, which has to be perpendicular to normal, and also the texture coordinates referring to the previously mentioned textures. The vertex position can be either pre-animated or static. In case it is static, the bone matrices must be accessible for the animation to be applied on the model. Last but not least, the rendering on that system has to be implemented in a shading language that supports multi-pass techniques, which can use textures as inputs or outputs. Figure 78 shows Alyson integration in Unigine [51].

Figure 78: Alyson in Unigine.

7.4 Conclusions and Future work

The main purpose of this master thesis was to prove that a series of physically principled lighting and shading effects can be applied on a skeletal based deformable animated virtual human character and executed with high-accuracy in real time. The reason why such a project is necessary is because even though there are similar systems, they either apply only a subset of the effects we aim to implement or they are not supported for animated full virtual human characters. Jimenez in 2012 presented at SIGGRAPH a set of realistic effects that run in real-time but only on a static virtual human head model. In 2013 NVIDIA presented the same set of algorithms but they used an animated virtual human head instead of static utilizing latest cluster of high-end dual GPU configuration. Unlike these 2 examples, our system not only applies a set of dynamically calculated realistic effects in real time using modern commodity hardware, but also it supports virtual animated full body human characters. **To the best knowledge of the authors there is no similar rendering framework in the bibliography.**

Therefore, we have created a rendering framework, which applies a series of physically principled lighting and shading effects in real-time to a skeletal-based deformable animated virtual human character. The effects we have taken into consideration are the following: Ambient Occlusion, Shadow Mapping, Image-Based Lighting, Specular Surface Reflectance, Subsurface Scattering and Environment Mapping. In order to produce the appropriate results, some of them are selectively applied only to human skin. All of these effects have been implemented based on state-of-the-art Screen Space algorithms in a 5-pass technique that runs exclusively on GPU. Each pass renders to an output image that can be used as input to a subsequent pass. The reason why we need multi-ass techniques is because each one of these images contains fragment data that cannot be obtained in a single pass. Of course, more than one passes might use an output image rendered in a previous pass. That way we don't have to recalculate the common data for multiple effects that might be implemented in a different pass. This method offers great advantages to a 3D rendering system, such as scalability and efficient real-time rendering execution.

But even though our visual results are very close to the ones produced by ray tracing algorithms, there are still some aspects that can be improved. First of all, we could use more than one light source. Although the number of passes will be increased, by combining effects such as shadow mapping from multiple light sources, we can create more realistic scenes. Apart from that, we could also use area lights instead of point lights. Area lights are light sources that occupy space in 2 or 3 dimensions. The benefit would be the generation of low-frequency instead of high-frequency shadows. We could also use different specular reflection functions based on each geometry's material. As it now, only skin surfaces are subject to specular reflection. Last but not least, the scene could be more complex than only the human model. Placing Alyson in an actual modeled environment would make her look more realistic.

Other than the already existing effects we use, we could increase our scene's realism by implementing more effects that take place in the real world. Certainly, one of them is the depth of field. According to this, a distance is defined between a near and a far point in which objects appear sharp. The farther an object is from the far point or the closer it is from the near point, the more blurred it appears. Depth of field is a post-processing effect that needs to be implemented in a

separate pass after the last 2 subsurface scattering passes. It simulates the behavior of blurring the surrounding space when our eyes focus on a specific object. Apart from that, we could also increase the visual results of the rendered images by implementing the bloom effect. As with depth of field, bloom is a post-processing effect that is applied upon the rendered image after the subsurface scattering. It reproduces imaging artifacts of real-world cameras creating the illusion that the scene is captured with a real device. There are many more post-processing effects that we could add in our system to increase the realism, but we have to be careful with the fact that they decrease the rendering execution time as well.

8 Bibliography

[1] Hable and Dog, "Uncharted 2 Character lighting and shading," in *SIGGRAPH*, 2010.

[2] White and Barre-Brisebois, "Five Rendering Ideas from Battlefield 3 and Need for Speed: The Run," in *SIGGRAPH*, 2011.

[3] Blinn, "Models of Light Reflection for Computer Synthesized Pictures," 1977.

[4] Phong, "Illumination for Computer Generated Pictures," 1975.

[5] Gouraud, "Continuous Shading of Cruved Surfaces," 1971.

[6] William, "Casting Curved Shadows on Curved Surfaces," 1978.

[7] Zhang, Xu, Sun and Lee, "Parallel Split Shadow Maps For Large Scale Environments," 2006.

[8] Zhang, Sun, Xu and Lee, "Hardware-Accelerated Parallel-Split Shadow Maps," 2007.

[9] Dimitrov, "Cascaded Shadow Maps," 2007.

[10] Tadamura, "Rendering Optimal Solar Shadows with Plural Sunlight Depth Buffers," 2001.

[11] Reeves, "Rendering Antialiased Shadows with Depth Maps," 1987.

[12] Pharr and Green, "Chapter 17 of GPU Gems: Ambient Occlusion," 2004.

[13] Ritschel, Dachsbacher, Grosch and Kautz, "The State of the Art in Interactive Global Illumination," 2011.

[14] Bunnell, "Dynamic Ambient Occlusion GPU Gems," 2005 .

[15] Kontkanen and Aila, "Ambient Occlusion for Animated Characters," 2006 .

[16] Kirk and Arikan, "Real-Time Ambient Occlusion for Dynamic Character Skins," 2007 .

[17] Mittring, "Finding NextGen CryEngine2," 2007 .

[18] Shanmugam and Arikan, "Hardware Accelerated Ambient Occlusion Techniques On GPUs," 2007 .

[19] Filion and McNaughton, "StarCraftII Effects and Techniques," 2008 .

[20] Bavoil and Sainz, "Image-Space Horizon-Based Ambient Occlusion," 2008 .

[21] Hoang and Low, "Multi-Resolution Screen-Space Ambient Occlusion," 2010 .

[22] Loos and Sloan, "Volumetric Obscurance," 2010 .

[23] Szirmay-Kalos, Umenhoffer, Toth and Sbert, "Volumetric Ambient Occlusion," 2010.

[24] Ruiz, Szirmay-Kalos, Tamas and Umenhoffer, "Volumetric Ambient Occlusion for Volumetric Models," 2010.

[25] d'Eon and Luebke, "Chapter 14 of GPU Gems 3: Advanced Techniques for Realistic Real-Time Skin Rendering," 2007.

[26] Tuchin and Valery, "Light Scattering Methods and Instruments for Medical Diagnosis (in Tissue Optics)," 2000.

[27] Donner and Jensen, "A Spectral BSSRDF for Shading Human Skin," 2006 .

[28] Kelemen and Szirmay-Kalos, "A Microfacet Based Coupled Specular-Matte BRDF Model with Importance Sampling," 2001.

[29] Donner and Jensen, "Light Diffusion in Multi-Layered Translucent Materials," 2005.

[30] Borshukov, "Realistic Human Face Rendering for The Matrix Reloaded," 2003.

[31] Jimenez, "Screen-Space Perceptual Rendering of Human Skin," 2009 .

[32] Hable and Borshukov, "Fast skin shading (Shader X7 chapter 2.4)," 2009.

[33] D. Stamminger, "Translucent Shadow Maps," 2004.

[34] Jimenez, "Real-Time Realistic Skin Translucency," 2010.

[35] Green, "Real-Time Approximations to Subsurface Scattering (Chapter 16 of GPU Gems)," 2004.

[36] NVIDIA, "Chapter 7 of The CG Tutorial: Environment Mapping Techniques," 2003.

[37] Debevec, "Rendering Synthetic Objects into Real Scenes," 1998.

[38] Ramamoorthi, "An Efficient Representation for Irradiance Env Maps," 2001.

[39] Egges, Papagiannakis and Magnenat-Thalmann, "Presence and Interaction in Mixed Reality Environments," in *The Visual Compute*, 2007, p. 317–333.

[40] Papagiannakis, Foni and Magnenat-Thalmann, "Practical Precomputed Radiance Transfer for Mixed Reality," in *Virtual Systems and Multimedia*, 2005, p. 189–199.

[41] Kautz, "Hardware Lighting and Shading," 2004 .

[42] Barnes and Finch, *COLLADA – Digital Asset Schema Release 1.4.1 Specification (2nd Edition),*

2008.

[43] Baba, "An Overview of Parameters of Game Engine," 2007.

[44] "GLSL," [Online]. Available: http://www.opengl.org/documentation/glsl/.

[45] Debevec, "Light Probe Image Gallery," [Online]. Available: http://www.pauldebevec.com/Probes/.

[46] Chapman, "john-chapman-graphics," [Online]. Available: http://john-chapman-graphics.blogspot.co.uk/2013/01/ssao-tutorial.html.

[47] Jensen, "A Practical Model for Subsurface Light Transport," 2001.

[48] "OpenGL Primitives," [Online]. Available: http://www.yaldex.com/game-programming/0131020099_app02lev1sec3.html.

[49] "AntTweakBar," [Online]. Available: http://anttweakbar.sourceforge.net/doc/.

[50] "OpenSceneGraph," [Online]. Available: http://www.openscenegraph.org/.

[51] "PoserPro," [Online]. Available: http://poser.smithmicro.com/poser.html.

[52] Tato, Papanikolaou and Papagiannakis, "From Real to Virtual Rapid Architectural Prototyping," 2012.