NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

MSc THESIS

Real-Time Hair Rendering Using Hardware Tessellation

Panteleimon D. Kanellis

Supervisor: Theoharis Theoharis, Professor

ATHENS
SEPTEMBER 2023

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Απόδοση Μαλλιών σε Πραγματικό Χρόνο με Χρήση Tessellation

Παντελεήμων Δ. Κανέλλης

Επιβλέπων: Θεοχάρης Θεοχάρης, Καθηγητής

ΑΘΗΝΑ
ΣΕΠΤΕΜΒΡΙΟΣ 2023

MSc THESIS

Real-Time Hair Rendering Using Hardware Tessellation

Panteleimon D.Kanellis

S.N.: 1115201600055

Supervisor: Theoharis Theoharis, Professor

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Απόδοση Μαλλιών σε Πραγματικό Χρόνο με Χρήση Tessellation

Παντελεήμων Δ. Κανέλλης

Α.Μ.: 1115201600055

Επιβλέπων: Θεοχάρης Θεοχάρης, Καθηγητής

ABSTRACT

Real-time hair rendering has been a challenging topic in computer graphics for many years. On one hand, traditional techniques that involve generating polygonal layers (hair cards), are suitable for real time applications, but they cannot accurately capture physical properties such as lighting or shadows. On the other hand, rendering hair as individual lines can simulate physical effects much more accurately, but it is also more expensive to render. This document sets two goals. The first is to provide a deep technical understanding of hair rendering as individual lines, with the help of hardware tessellation in modern OpenGL. The entire pipeline is explained from pre-processing to create a simplified growth mesh for the existing hairstyle, to correctly setting up appropriate buffers for smooth curves and line generation, to hair lighting and shadowing. A github repository [1] with all the source code is accompanied with this document. The second goal is to create the hair rendering framework for GPUs with limited capabilities. More specifically, the challenge is to create a real-time framework for low end GPUs from as far as 8 years ago. All the measurements were performed on an Nvidia GT 920M GPU with 2GB of virtual memory.

SUBJECT AREA: Graphics Programming

KEYWORDS: hair rendering, OpenGL, tessellation, lighting, shading

ΠΕΡΙΛΗΨΗ

Η απόδοση μαλλιών σε πραγματικό χρόνο αποτελεί ένα δύσκολο θέμα στα γραφικά υπολογιστών για πολλά χρόνια. Από τη μια πλευρά, παραδοσιακές τεχνικές όπως η δημιουργία πολυγώνων (ή κάρτες μαλλιών), είναι κατάλληλες για εφαρμογές πραγματικού χρόνου, αλλά δεν μπορούν να αποτυπώσουν με ακρίβεια φυσικές ιδιότητες όπως το φωτισμό ή τη σκίαση. Από την άλλη πλευρά, η απόδοση των μαλλιών ως μεμονωμένες γραμμές μπορεί να προσομοιώσει τα φυσικά εφέ με πολύ μεγαλύτερη ακρίβεια, αλλά ο υπολογιστικός φόρτος είναι μεγαλύτερος. Το παρόν έγγραφο θέτει δύο στόχους. Ο πρώτος στόχος είναι η παροχή μιας βαθιάς τεχνικής κατανόησης της απόδοσης μαλλιών ως μεμονωμένες γραμμές, με τη βοήθεια της τεχνολογίας tessellation (ψηφίδωσης) σε σύγχρονη OpenGL. Εξηγείται ολόκληρη η διαδικασία, από την προεπεξεργασία για τη δημιουργία ενός απλοποιημένου πολυγωνικού σχήματος για το υπάρχον στυλ μαλλιών, για τη σωστή δημιουργία των κατάλληλων καταχωρητών για ομαλές καμπύλες και δημιουργία γραμμών, μέχρι το φωτισμό και τη σκίαση των μαλλιών. Το παρών έγγραφο συνοδεύεται με όλο τον πηγαίο κώδικα σε μορφή ενός αποθετηρίου github [1]. Ο δεύτερος στόχος είναι να εφαρμοστεί η απόδοση μαλλιών σε κάρτες γραφικών με περιορισμένες δυνατότητες. Πιο συγκεκριμένα, τίθεται η πρόκληση της ανταπόκρισης της εφαρμογής σε πραγματικό χρόνο, για κάρτα οκταετίας. Όλες οι μετρήσεις πραγματοποιήθηκαν με χρήση της κάρτας Nvidia GT 920M με 2GB εικονικής μνήμης.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Προγραμματισμός Γραφικών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: απόδοση μαλλιών, OpenGL,
ψηφίδωση, φωτισμός, σκίαση

CONTENTS

1. INTRODUCTION

Hair and fur rendering for real time applications is challenging due to the large amount of thin strands that need to be rendered. For reference, the average person has around 100,000 - 150,000 hair follicles. Traditional techniques involve rendering a patch of hair strands as a collection of polygons. These polygons also referred to as hair cards (Figure 1.1). This is fairly efficient in real time, however hair cards cannot accurately capture physical properties such as lighting or shadows between strands. Also, the simulation of hair cards is less realistic. Moreover, the process of creating hair cards is quite tedious and requires a lot of time from artists. On the other hand, rendering each individual strand that can have multiple segments, and calculating the lighting and shadowing amount that is caused by all the other hair strands can become a difficult problem to solve, but it can provide much more realistic results than hair cards. The problem of rendering each line becomes even bigger for hairstyles with long hair, or for many lights in the scene. This document builds a rendering framework for hair rendering in modern OpenGL from scratch. The main contributions of this document are:

- Examines on a deep technical level the process of rendering hair as strands with hardware tessellation. Every single step, regardless if it involves CPU or GPU code, is thoroughly explained with easy-to-follow code snippets.
- Manages to create a strand based hair renderer efficiently, even in an old, low-end graphics card. This is the main constraint that is set in this document. The framework needs to be responsive (30+ frames), while also providing realistic results.

The structure of the document is as follows:

Chapter 2 presents related work that can go beyond the scope of hair rendering. In chapter 3, a simple hairstyle consisting of 10,000 hair strands and 16 segments for each strand is rendered by sending each individual line to the GPU, without additional lighting or shading. The disadvantage of this approach is that it is quite expensive. Chapter 5 tries to solve this problem, by first selecting a subset of the 10,000 strands and creating a polygon from the roots of the subset. The resulting subset is referred to as ”guides”, because they will be used to guide the pipeline into creating more hair to fill the rest of the scalp later on. In this chapter, the guides are rendered as camera facing quads instead of lines, which will help to assign properties to each strand, such as width and texture coordinates. Chapter 6 describes the tessellation pipeline in OpenGL. In the context of hair rendering, there are two uses of tessellation. The first is to create additional points for each strand, effectively turning a jagged line into a smooth curve. The second use is to fill the entire head with more strands on the GPU instead of sending each hair strand from the CPU to the GPU. Chapter 7 introduces two lighting techniques that are specifically tailored for hair rendering. In chapter 8, two of the most popular shadowing techniques are explored that are not exclusive to hair rendering and can be applied to any sort of geometry. Finally, chapter 9 presents observations and areas of the implementation that can be improved in the future.

Figure 1.1: Traditional hair rendering techniques involve rendering the hairstyle as polygonal cards.

2. RELATED WORK & BACKGROUND

In terms of representing hair as a file, Yuksel [2] has created the HairFile class, written in C++. This data structure holds valuable properties of hair such as color, thickness, hair count and the entire array of 3D points from the hairstyle. The data that holds these properties has the ".hair" file extension. Additional functions can read this file and fill the HairFile class. Tariq et al. [3] utilized the power of hardware tessellation to effectively render and simulate many hair strands from a small subset of guide hairs in real time. Many of the techniques explored in this document have Tariq's work as a reference point. Disney's contribution [4] in this topic was creating an artist friendly hair rendering system for fast iterations, that lets the user change the physical properties of hair with ease. Even though their work did not include real-time rendering as a priority, it is important to note that they rendered a small subset of guide hair, and they used the guide hair in order to increase the number of strands on the fly. This technique, as well as similar lighting models are also explored in this document. Kajiya & Kay [5] presented the earliest lighting model for rendering hair. In theory it is similar to the Phong model, since it is combined of ambient, diffuse and specular components. For the diffuse component, the light scatters equally in all directions. The specular component is calculated based on the orientation of the surface and the viewing direction. Marschner et al. [6] presented a more accurate lighting model for hair. The observation was that hair have primary and secondary highlights. Treating the hair as very thin cylinders, Marscher's lighting model can more accurately capture the physical properties of light scattering in hair, but it is slightly more complex than Kajiya & Kay's model. Scheuermann [7] built on top of that by replacing the tangent with a randomly shifted tangent, in order to simulate the offset in the specular highlight. Reeves et al. [8] introduced the shadow mapping technique with filtering, which is widely used in real time graphics applications ever since. This technique generates a depth map in a separate pass and using this texture in the main rendering pass makes it easy to determine if an object is in shadow or not. An extra step is needed to avoid common issues such as shadow acne. Percentage Closest Filtering (PCF) is used to minimize this issue and simulate soft shadows. Donnelly et al. [9] improved upon the original PCF shadows, by introducing Variance Shadow Maps (VSM). This technique calculates the variance of depth values within a shadow map, and estimates the amount of shadowing based on the variance texture. As a result, VSM can achieve smoother and more realistic shadows than PCF.

3. BRUTE FORCE RENDERING APPROACH

The goal of this chapter is to introduce the hair rendering framework in OpenGL. More specifically, this chapter explores how the attributes of a specific hairstyle are stored. A hairstyle can be described by the .hair file extension. After reading the file and filling appropriate data structures, the hair sample is rendered by passing each segment separately. A segment consists of two 3-dimensional points that make one strand of hair. This approach is described as ”brute force”, because the CPU passes every single strand of hair to the GPU one by one, which creates a performance degradation.

3.1. The Hair Rendering Framework

Before any hair rendering implementations are explored, it is important to set up the foundations of the framework. The rendering environment is written in OpenGL and C++, using GLFW library to create a window on the screen, as well as the GLEW library for OpenGL core and extension functionalities.

The file that contains the main function is responsible for setting up the window and its parameters, as well as the camera. The rendering environment is interactive, meaning that the user can move around the scene with the ”WASD” keys, and change the camera rotation with the mouse.

The main rendering loop simply calls a display function with all the necessary shaders as parameters. In order to render the hairstyle, a 3D head model is rendered first. The framework uses the Assimp library to load .obj files. If a model consists of several meshes, each one of them are drawn to the screen sequentially. The model, view and projection matrices are passed to the shaders of the head, and a simple phong model is set up to increase the realism. A single light source that acts as a point light, is rotating around the model. Figure 3.1 shows the rendering of just the head model.

Figure 3.1: Rendered 3D head model. A point light source illuminates the object.

3.2. Rendering .hair files

Now that a rendering environment is set up, it is time to render the hairstyle. This implementation reads .hair files and fills data structures before sending the vertices to the rendering pipeline. A .hair file is described with the following most important attributes:

- **hair count**: Total number of hair strands
- **point count**: Total number of hair points (in XYZ triplets)
- **segments**: Default number of segments for each strand
- **thickness**: Default thickness for each strand
- **transparency**: Default transparency for each strand
- **color**: Default color for each strand

The above parameters describe the header of the .hair file. Additional structures are needed to store the attributes of each individual strand:

- **point array**: XYZ coordinates of each individual point of the hairstyle
- **segment array**: Number of segments per strand. A segment consists of two points.

The definition of the HairFile class of the external cyHairFile code can be defined with the above arrays and the header information.

The Hair class that is part of the implementation of the hair rendering framework, consists of one instance of HairFile object, a vertex array object (VAO) and four vertex buffer objects (VBO). The four VBOs represent the point,color,transparency and thickness arrays. The main function reads a .hair file and the Hair class is responsible for filling the arrays.

The reason why this technique is called ”brute force”, is because the CPU will send vertex data to the rendering pipeline for every single strand of hair. This can be a huge performance bottleneck for any type of hairstyle. For example, the framework renders a hairstyle consisting of 10,000 strands and 16 segments per strand. The total size of vertices that need to be passed to the vertex shader are 160,000. The pseudocode below describes the draw function that gets called by the main rendering loop.

```
pointIndex = 0
segments = hairfile.segments
for (hairIndex = 0; hairIndex < hairCount; hairIndex++) {
        glLineWidth(hairfile.thickness)
        glDrawArrays(GL_LINE_STRIP, pointIndex, segments + 1)
        pointIndex += segments + 1
}
```

First, the number of segments for each strand are accessed. Inside a loop, one strand of hair is sent to the GPU until all the segments are rendered. From the previous example, this code would send 16 XYZ triplets per loop. The loop would be completed after 10,000 iterations. This procedure happens per frame. Figure 3.2 shows the rendered hairsytle, along with a frame-pre-second counter at the top. The vertex and fragment shaders are pass-through shaders. It is important to note that this result does not contain any lighting or shadow calculations. Even without any complex rendering, the performance can barely stay above 50 fps.

Figure 3.2: Rendering a hairstyle by sending every strand to the GPU. This is a performance bottleneck

4. SETTING UP THE HAIR GROWTH MESH

From the previous chapter, it is evident that rendering each individual strand per frame is not optimal. Tariq [3] has shown that there is a more effective way to render dense and thin geometry. Instead of sending every strand segment to the GPU in a loop, the framework can render the entire hairstyle with a single draw call. First, a subset of all the hair must be selected from the original hairstyle. The triangle mesh that will be created from the roots of the subset will be called growth mesh. The resulting hair strands that are selected as a subset are called guide strands. They will be used in a later chapter to fill the entire scalp with strands that will be rendered on the fly. The first section of this chapter describes the process of outputting a point cloud including all the vertices at the root of each strand. The second section creates the growth mesh by connecting the point cloud into a triangle mesh and simplifying it. Both of these steps are pre-processing steps, and do not have any burden on rendering performance.

4.1. Exporting Hair Root Vertices

In order to create the growth mesh, the OpenGL framework extracts the root vertices of the hairstyle and outputs them in a text file. The pseudocode code below implements this process.

```
void WriteRootOutput() {

        pointCount = hairfile.GetHeader().point_count;
        points = hairfile.GetPointsArray();

        rootFile.open("roots.txt");

        // Segments for each hair
        segments = hairfile.GetSegmentsArray();

        if (segments) segment_stride = segments[0];
        else segment_stride = hairfile.GetHeader().d_segments + 1;

        hairCount = hairfile.GetHeader().hair_count;
        roots = float_array[ hairCount* 3];
        rootIndex = 0;

        for (int i = 0; i < pointCount*3; i += 3*segment_stride) {
                x = points[i];
                y = points[i + 1];
                z = points[i + 2];

                roots[rootIndex] = x;
                roots[rootIndex + 1] = y;
                roots[rootIndex + 2] = z;
                rootIndex += 3;

                rootFile.write(x y z);
        }

        rootFile.close();
```

```
}
```

First, the number of total points is accessed and the empty file is created. The framework assumes that the hair file has a fixed amount of segments for every strand. The loop passes through all the points and only writes the first three vertices of each segment to the root file. Each coordinate is separated by space.

4.2. Creating the Growth Mesh

Now that there is a representation of the root vertices in text form, the growth mesh can be created by first connecting the point cloud in a triangle mesh. MeshLab was used to import the text file that was created in the previous chapter. Figure 4.1 shows the imported point cloud.

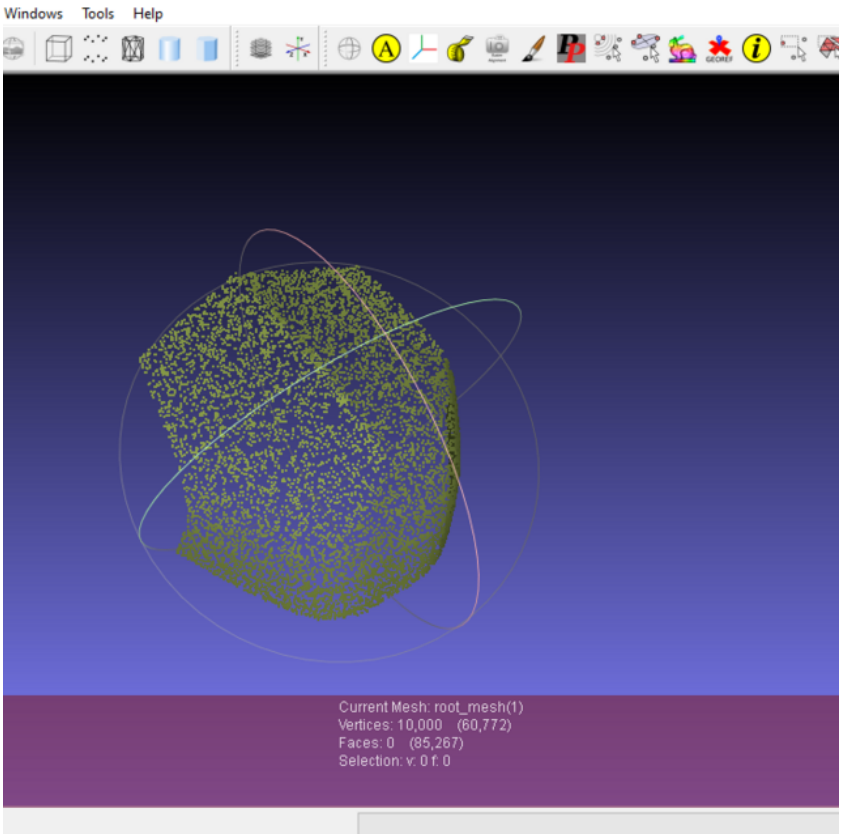

Figure 4.1: Imported root vertices to MeshLab

The pre-processing stage continues by connecting the point cloud into a triangle mesh. In order to achieve this, the Ball Pivoting Algorithm (BPA) was executed. According to BPA, a ball with a user defined radius is used. If the ball touches three points of the cloud, then a triangle is created. Traversing the entire point cloud, a triangle mesh can be created. MeshLab already provides this algorithm under the Filter tab. Figure 4.2 shows the result of BPA. An additional step is made to cover potential holes that BPA has left. After that, the mesh is ready to be simplified. The number of vertices chosen was 116. Figure 4.3 shows the result. This is not the final form of the growth mesh, as an additonal modification must be done. The mesh was edited to create a hairline, as well as increase the number of vertices in some areas that are needed. The final growth mesh, which was edited in Blender, is depicted in Figure 4.4 and consists of 418 vertices.

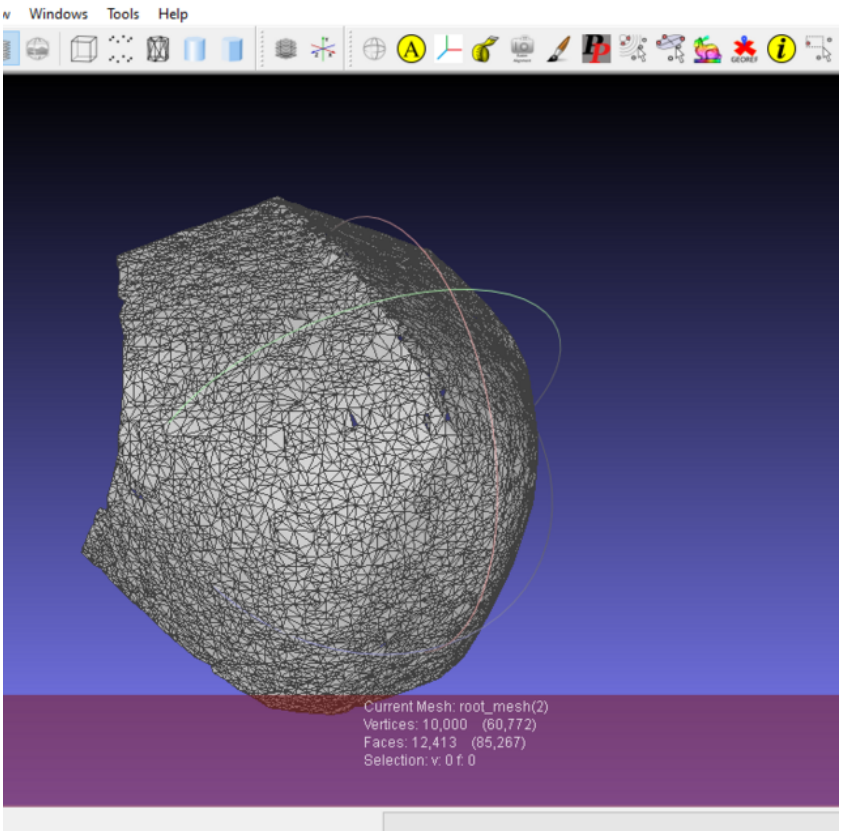

Figure 4.2: Ball Pivoting Algorithm (BPA) in MeshLab

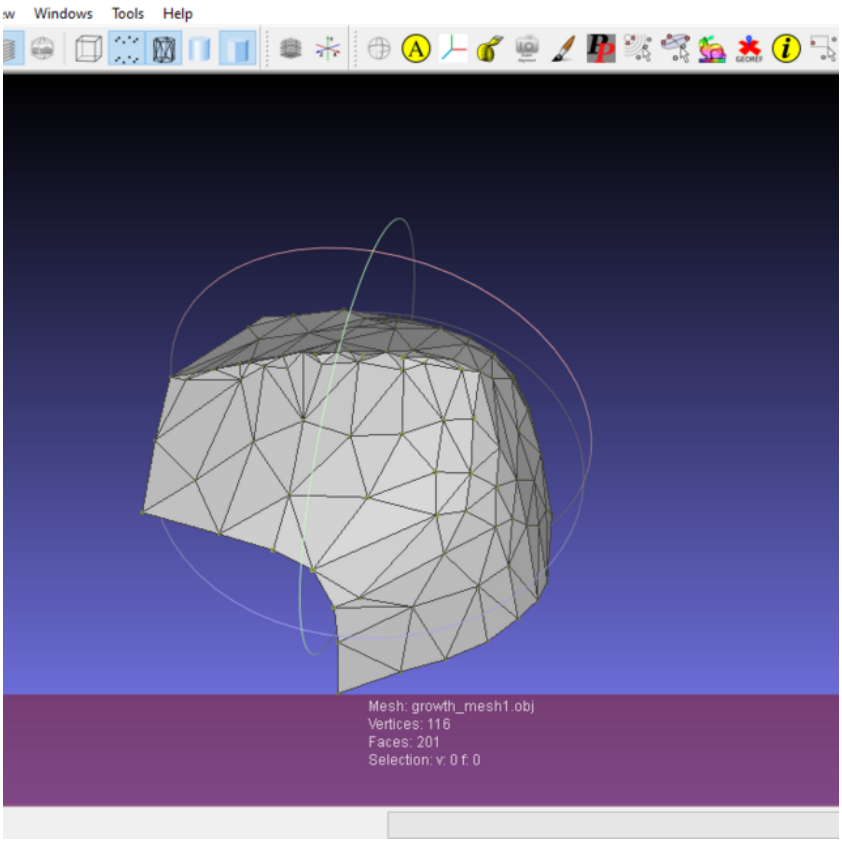

Figure 4.3: Simplified growth mesh in MeshLab

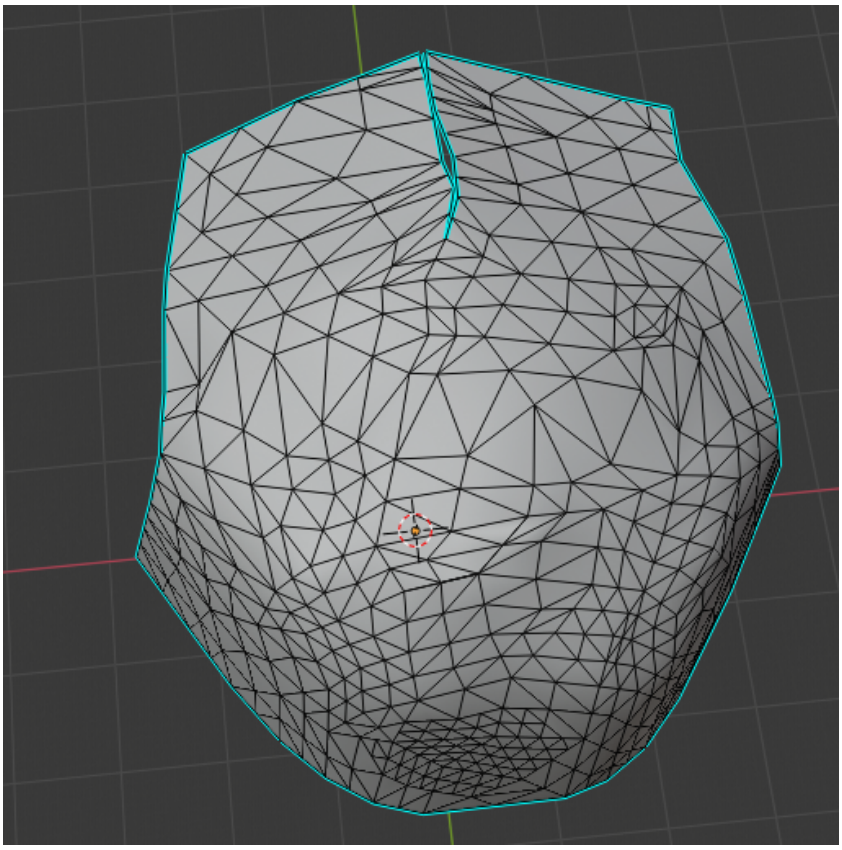

Figure 4.4: Added hairline and increased the number of vertices in Blender

5. RENDERING HAIR GUIDES

This chapter continues the development of the OpenGL framework, after having acquired the growth mesh from the previous chapter. The goal is to render only a small subset of strands instead of the entire hairstyle. The number of guide strands are the same as the number of vertices of the growth mesh. The previous chapter created a growth mesh with 418 vertices, so that means that 418 guide strands will be rendered. The first section of this chapter selects the strands that are closest to the vertices of the growth mesh, thus creating the guide strand subset. The second chapter renders the guide strands not as lines, but as camera facing quads. Multiple reasons are presented as to why rendering triangles instead of lines is more appropriate for hair rendering.

5.1. Choosing Hair Guides

The exported format of the growth mesh is in .obj format. Assimp is used to load the final version of the growth mesh into the rendering framework. The Hair class that was introduced in chapter 3 could be used to describe the guide strands. However, a separate class is made to represent the guide hair, because additional functionalities that are exclusive to guide strands must be defined. The most important attributes of the Guides class are:

- **growth mesh points**: The xyz coordinates of the growth mesh vertices.
- **guide points**: The selected points from the hairstyle.
- **per vertex attributes**: color, thickness and transparency values for each guide point. There's also a segment index that signifies whether there is a new strand.

The first thing that needs to be done is to write the vertices of the growth mesh in a separate array. When the growth mesh .obj file is loaded with Assimp, many of the vertices will be repeated, because indices will be repeated inside the file in order to draw the mesh. Since the goal here is to extract only the individual vertices and not render the growth mesh, a function must be created to remove vertex repetition. The following pseudocode fills an array of 3D points from the vertices of the growth mesh, while also checking for duplicates and rejecting them.

```
void FillPoints() {

        map<<float, float, float>, int> check_duplicates;

        count = 0;

        for (i = 0; i < total_points; i++) {

                x = all_vertices[i].Position.x;
                y = all_vertices[i].Position.y;
                z = all_vertices[i].Position.z;

                findVertex = check_duplicates.find(x, y, z);
                end = check_duplicates.end();

                isDuplicate = findVertex != end;
```

```
                if (isDuplicate) continue;

                check_duplicates[make_tuple(x, y, z)] = count;

                growth_mesh_points[count] = x;
                growth_mesh_points[count + 1] = y;
                growth_mesh_points[count + 2] = z;

                count += 3;
        }

}
```

First, a dictionary is created with xyz coordinates as keys and an integer as values. If a vertex does not exist in the dictionary, it is added and the vertex is added to the point array. Else, it is rejected.

Since, the discrete vertices of the growth mesh are saved in an array, it is time to select the strands that are nearest to each vertex of the array. Essentially, the distance of each root of the entire hairstyle to every xyz coordinate of the growth mesh vertices is calculated. The minimum distance will be used to pick the nearest strand. This is a standard nearest neighbor algorithm, where each query is compared with all of the items in the dataset. In order to make sure that all the strands have been compared and that only discrete strands have been picked, an additional array is needed. The pseudocode below describes the process of selecting the nearest neighboring hair strand for every vertex of the growth mesh. Also, each selected strand is offsetted so that the root sits exactly at the vertex.

```
void SelectGuidesFromHairfile() {

  total_point_array_size = 0;

  hair_points = hair.GetAllHair();
  seg_count = 0;

  //for each vertex in the growth mesh
  for (int i = 0; i < 3*total_points; i+=3) {

    min_dist = INT_MAX;
    min_index = 0;

    x_growth = growth_mesh_points[i];
    y_growth = growth_mesh_points[i + 1];
    z_growth = growth_mesh_points[i + 2];

    min_ammountX, min_ammountY, min_ammountZ = 0.0;

    //for each hair root
    for (int j = 0; j < 3 * hairCount; j+=3) {

      // Ignore already picked lines
      if (already_picked_line[j / 3] == 1) continue;

      x_root = roots[j];
      y_root = roots[j + 1];
      z_root = roots[j + 2];

      distance = CalculateDistanceAndAmount();
```

```
      if (distance < min_dist) {
        min_dist = distance;
        min_index = j;
      }

    }
    already_picked_line[min_index / 3] = 1;

    // NN -> Nearest Neighbor
    int NN_segments = line_segments[min_index/3];

    min_indices[seg_count] = min_index / 3;
    line = new array[3*NN_segments];

    nearest_segments[seg_count] = NN_segments;
    ++seg_count;

    start_line = 0;

    // The min_index root is the nearest to the i-th vertex
    for (k = 0; k < min_index/3; k++) {
        start_line += line_segments[k];
    }

    // for each x,y,z coordinate
    start_line = 3 * start_line;
    total_point_array_size += NN_segments;

    count = 0;

    for (g = start_line; g < start_line + 3*NN_segments; g+=3) {

      // Moving each point in order to sit
      // exactly at the vertex of the growth mesh
      line[count] = hair_points[g] + min_ammountX;
      line[count + 1] = hair_points[g + 1] + min_ammountY;
      line[count + 2] = hair_points[g + 2] + min_ammountZ;
    }

    guide_points.add(line);
  }

}
```

For each vertex in the growth mesh, the function scans the entire list of hair strands. From them, only the roots are chosen. The root vertex that has the minimum distance from the current growth mesh vertex, is selected and thus belongs to the vertex. The selected root is marked as ”already picked”, inside an array. Starting from the root of the nearest strand, every point is moved in 3D space, so that the root sits exactly at the top of the growth mesh vertex. The process of selecting the nearest root is a pre-processing step and doesn’t burden the real-time performance.

5.2. Rendering Camera Facing Quads

The guide strand dataset that was completed in the previous section, is now ready to be rendered. Keeping in mind that only one draw call is optimal, a new issue arises: How can the strands be separated from one another if only one draw call is used? In chapter 3, the

for loop was enough to distinguish between different strands, because every draw call was assigned to each strand. If the guide strands are rendered as is, the connecting segment at the end of one strand to the start of the next will be visible. Figure 5.1 describes this problem. S1 and S2 represent two guide strands, where S1 precedes S2. The connecting edge S12 must not be visible in the rendered image. The solution is to provide a buffer to the guide shader that explicitly states the width of each individual segment of every strand. The width of the segment that starts at the end of one strand and ends at the start of the next strand, will be zero.

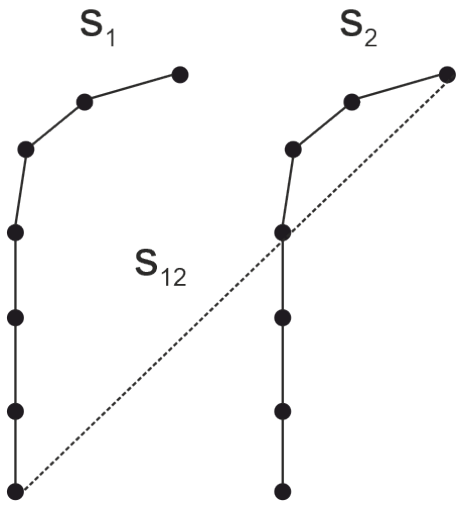

Figure 5.1: Example of two guide strands. The connecting segment S12 must be invisible.

Even though this could solve the aforementioned issue, the framework so far renders the hair as 2D lines. Calling the draw command with the primitive type ”GL_LINES”, it is expected to pass a line width as a parameter. Since every strand is rendered per iteration, only a constant number of width per strand is passed.

An alternative way to rendering hair as 2D lines, is to render them as camera facing quads. This technique, also known as ”billboarding”, extends a line into a quad that constantly faces the viewer. In order to render each segment of the hair as two triangles, they must be expanded in the shader pipeline. Even though this expansion can happen in the vertex shader, there is a more convenient way. The geometry shader can easily modify the vertices from the vertex shader. The advantages of rendering hair as quads are:

- **real life width**: Each segment can have its own thickness value. Human hair have thick roots and thin tips. The thickness will be especially useful in eliminating the undesirable connecting segments across strands.
- **texture coordinates**: Since the framework renders triangles, textures can be applied in the fragment shader, that will give a more realistic appearance, instead of a linear color that 2D lines demand. The texture coordinates can be generated in the geometry shader.

The code snippet below introduces the geometry shader pass. Figure 5.2 shows the process of creating two triangles from a line segment.

```
void main(){
```

```
        vec4 p1 = gs_in[0].point;
        vec4 p2 = gs_in[1].point;

        vec3 tangent = p2.xyz - p1.xyz;
        tangent = normalize(tangent);

        vec3 eyeVec = p1.xyz;
        vec3 side_vector = normalize(cross(eyeVec, tangent));

        float radius1 = gs_in[0].thickness * 0.05;
        float radius2 = gs_in[0].thickness * 0.05;

        texCoord = vec2(0.0, 0.0);

        vec3 Pos;

        Pos = p1.xyz + ((side_vector + tangent) * radius1);
        texCoord.x = 0.0;
        gl_Position =  projection *  vec4(Pos, 1.0);
        EmitVertex();

        Pos = p1.xyz - ((side_vector - tangent) * radius1);
        texCoord.x = 1.0;
        gl_Position =  projection * vec4(Pos, 1.0);
        EmitVertex();

        Pos = p2.xyz + ((side_vector + tangent) * radius2);
        texCoord.x = 0.0;
        gl_Position =  projection * vec4(Pos, 1.0);
        EmitVertex();

        Pos = p2.xyz - ((side_vector - tangent) * radius2);
        texCoord.x = 1.0;
        gl_Position = projection * vec4(Pos, 1.0);
        EmitVertex();

        EndPrimitive();
}
```

The geometry shader starts by receiving the input line segment from the vertex shader in view space. It accepts lines as input and outputs a triangle strip consisting of four vertices. In order to transform the line segment into a quad, the code snippet first calculates the tangent of the segment. Since the two points are already transformed into view space from the vertex shader, the eye vector is just the first vertex of the line. The side vector is the cross product of the the eye vector and the tangent, and is perpendicular to the two. Each point in the line will be expanded twice, once for the left and once for the right vertex of the quad. In this stage, the thickness of each segment is taken into account. For the final segment of the hair strand, the thickness value of the second vertex will be zero. Texture coordinates are also calculated for each emitted vertex.

The disadvantage of rendering each strand as a set of two camera facing triangles, is that lines are generally cheaper to render compared to triangles. An additional shader is introduced to the rendering pipeline, which will have an impact on the performance. However, the performance degradation is very small. Figure 5.3 shows the rendered subset of the hair guides as camera facing quads.

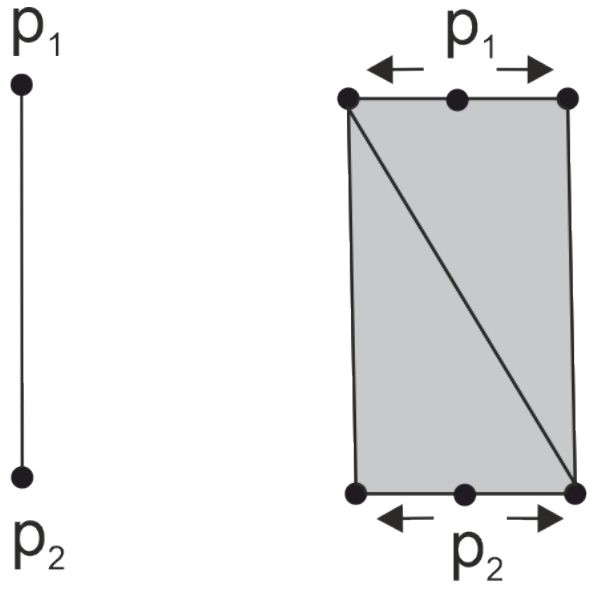

Figure 5.2: The process of expanding a line into a quad. The amount of expansion depends on the thickness of the segment.

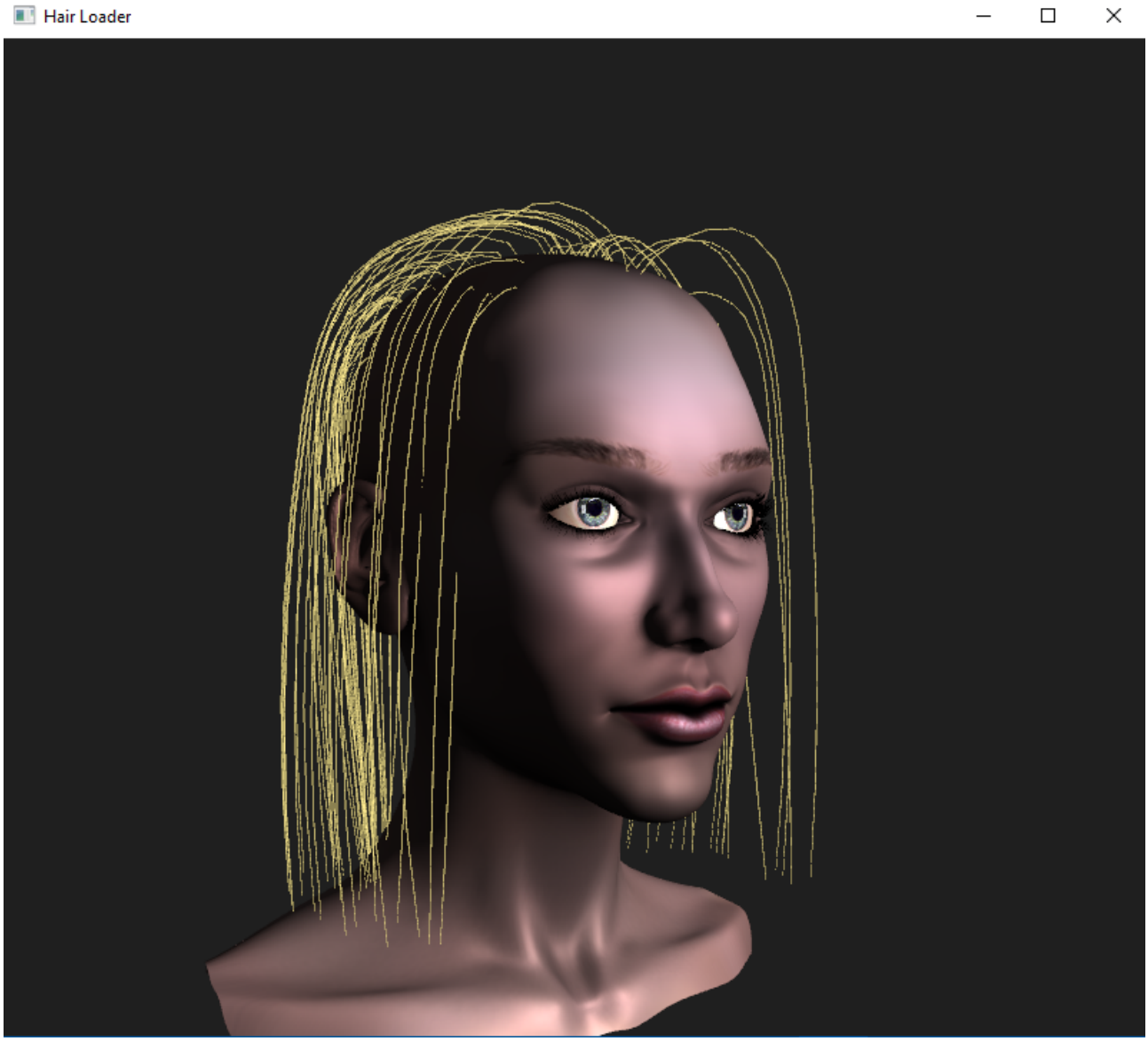

Figure 5.3: Rendered guide strands as camera facing quads, using the geometry shader.

6. LINE TESSELLATION WITH OPENGL

The guide hair have been defined in the previous chapter, and have been rendered as quads with segment separation. This chapter will describe the process of using the guides in order to fill the entire scalp with additional hair, as well as create smooth curves for each strand. These two processes must happen on-the-fly in the GPU for better performance. There is a powerful step in the graphics pipeline: the tessellation. It is important to define tessellation and how it can be used not only for its most widespread use which is triangles, but for a more specialized use, such as lines.

Tessellation was first introduced as an additional (and optional) vertex processing stage in the graphics pipeline in 2010 for OpenGL. Tessellation is divided in three stages, two of which are programmable. These three stages are defined before the optional geometry shader and after the vertex shader. Figure 6.1 shows the complete pipeline in OpenGL.

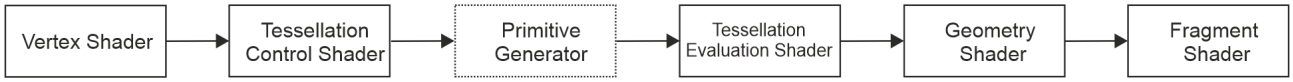

Figure 6.1: Complete pipeline in OpenGL. The Primitive Generator stage is non programmable.

Generally, tessellation is used to subdivide a user defined patch. For example, if the user defines a patch of 3 vertices, each triangle of the mesh will be subdivided. The tessellation stages are described below.

- **Tessellation Control Shader:** This stage is responsible for setting the tessellation parameters such as the amount of tessellation, as well as passing potential data fields to the next stage.
- **Primitive Generator:** The primitive generator is a fixed stage and cannot be controlled by the programmer. Depending on the amount of tessellation, a domain is subdivided, not the actual patch. Barycentric coordinates are generated based on the primitive type. For triangles, there will be U,V and W coordinates, and the domain will be a triangle domain.
- **Tessellation Evaluation Shader:** In this stage, the generated barycentric coordinates are processed and mapped to the actual geometry.

A widespread application of tessellation is terrain generation with dynamic level of detail. In that case, the amount of triangle subdivision can fluctuate depending on the distance from the camera to the geometry. Using tessellation for lines, the primitive generator will handle the normalized coordinates differently than triangles. In case of isolines, only two coordinates will be generated. The first one determines the distance between generated vertices of the same line, and the second coordinate defines which line is generated. Regarding hair rendering, the first coordinate will be used for creating additional points in the line for smooth curves, while the second coordinate will be used to generate additional lines across the scalp.

The chapter continues by exploring two cases of line tessellation. First, the technique that was used to create smooth curves is explained from a theoretical perspective. After that, a more practical approach is taken in order to efficiently send data to the GPU in such a way

that makes the smooth curve generation possible. Next, the code for the tessellation shaders is introduced and explained. The second use of tessellation is later explained and a strategy is created to generate additional hair. Having this strategy in mind, the last section of this chapter presents the complete tessellation evaluation shader code as well as performance measurements and comparisons.

6.1. Cubic B-Spline Curves

In mathematics, there are many ways to express a curve. One of the most popular ones is to use Bezier [10] curves. In that case, one could simply implement Bezier curves per 4 control points. A control point is a vertex in the initial, pre-tessellated hair strand. Since 4 control points were chosen, the Bezier curve will be cubic, which means that a polynomial of degree 3 must be calculated inside the shader. The major disadvantage in using Bezier curves is that there are discontinuities for every 4 vertices, as shown in Figure 6.2. One way to solve this would be to create a Bezier polynomial with a degree equal to the number of segments per strand. In case of the selected hairstyle, the shader would need to calculate 15 coefficients of degree 15. This would be extremely inefficient for real time rendering.

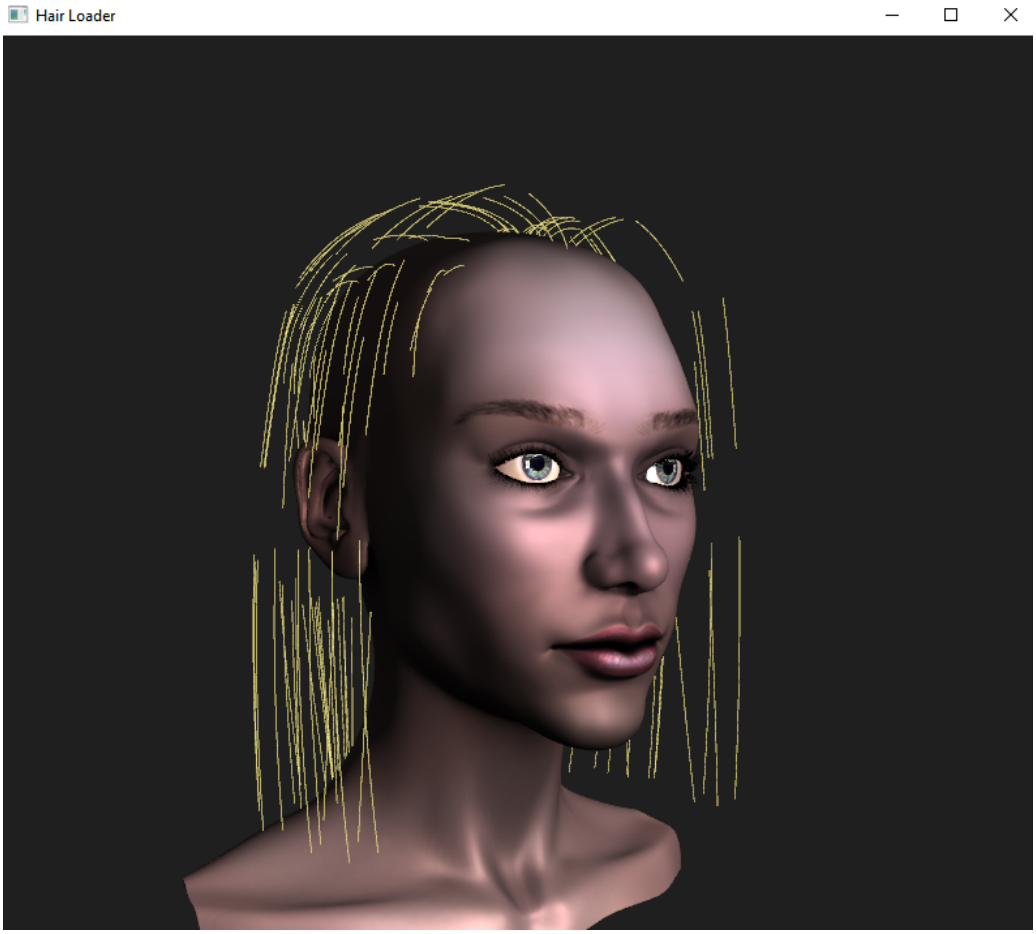

Figure 6.2: Bezier curves suffer from disconuities between adjacent segments.

There is a much more convenient way of expressing splines, one that guarantees line continuity between segments. This technique is called basis splines or b-splines [11]. A basis spline of order n can be written as a piecewise polynomial of multiple Bezier curves. A cubic

b-spline of 4 control points has degree 3 and consist of one Bezier curve. The number of control points are independent of the degree of the b-spline. In general:

> A b-spline of degree k and $n+1$ control points, is a collection of $n-k+1$ Bezier curves.

The main difference is that b-splines ensure continuity between segments, by using the knot vector. The knot vector is the collection of points that join two Bezier curves together. The following statement is true for the knots:

> A k order b-spline has at most C^{k-2} continuity at the knots.

For a cubic b-spline, there is at most C^1 continuity, meaning that the joining parts of the spline are connected and smooth. Mathematically, the knot vector is defined as a collection of breaking points with ascending order:
$T = (t_0, t_1, ..., t_m)$, where $t_0 \leq t_1 \leq ... \leq t_m$

The b-spline basis function, given a knot vector T, is written as $N_{i,k}(t)$ and defined as:

$$N_{i,1}(t) = \begin{cases} 1, & \text{if } t_i \leq t \leq t_{i+1}. \\ 0, & \text{otherwise.} \end{cases} \tag{6.1}$$

for $k = 1$, and:

$$N_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i-1}} N_{i+1,k-1}(t) \tag{6.2}$$

for $k > 1$ and $i = 0, 1, ..., n$, where i defines the index of the control points $P_0, P_1, ..., P_n$.

For a cubic b-spline with four control points, using the equations 6.1 and 6.2, the following basis functions can be derived:

$B_0(t) = \frac{-t^3+3t^2+1}{6}$

$B_1(t) = \frac{3t^3-6t^2+4}{6}$

$B_2(t) = \frac{-3t^3+3t^2+3t+1}{6}$

$B_3(t) = \frac{t^3}{6}$

A good property of cubic b-splines with four control points, is that the coefficients of the basis functions could be written as a 4x4 matrix.

6.1.1. Sending Data to the GPU

The previous chapter explained how the cubic b-splines ensure continuity between two segments. However, there is vertex repetition in order to achieve this. For example, suppose that twelve vertices consist of the initial control points of one strand, $P_0, P_1, P_2, ..., P_{11}$. If four control points are chosen each time, the order that they must be sent to the GPU is:

$P_0P_1P_2P_3$
$P_1P_2P_3P_4$

$P_2P_3P_4P_5$
...
$P_8P_9P_{10}P_{11}$

The way that the control points were sent before the tessellation stage were sequential without any repetition. In order to implement cubic b-splines, repeated vertices must be sent to the pipeline. To achieve this, indices must be introduced. The same way that an .obj file defines the order of the vertices to create triangles for example, a specific order must be created for the control points in order to permit vertices to be used more than once. The pseudocode below creates indices for each strand of the guide subset:

```
void Fill_Indices(total_points) {
        count = 0;

        p0 = 0;
        p1 = 1;
        p2 = 2;
        p3 = 3;

        for (int i = 0; i < total_points ; i++) {
                for (int j = 0; j < nearest_segments[i]-1; j++) {
                        // special treatment for first and last
                        //vertex of the strand
                        if (j == 0) {
                                indices.push_back(p0);
                                indices.push_back(p0);
                                indices.push_back(p1);
                                indices.push_back(p2);

                                --p0; --p1; --p2; --p3;
                        }
                        else if (j == nearest_segments[i]-2) {
                                indices.push_back(p0);
                                indices.push_back(p1);
                                indices.push_back(p2);
                                indices.push_back(p2);
                                p0+=2; p1+=2; p2+=2; p3+=2;
                        }
                        else {
                                indices.push_back(p0);
                                indices.push_back(p1);
                                indices.push_back(p2);
                                indices.push_back(p3);

                        }
                        ++count; ++p0; ++p1; ++p2; ++p3;
                }
        }

}
```

For each vertex in the strand of the current guide, a new vector that stores the order of each vertex is created and filled. There needs to be special treatment for the first and last vertex of the strand. More specifically, if the first vertex of the strand is chosen, the first control point will be repeated twice, because the first point does not have a predecessor. The index counters will be subtracted and added by one after the if case, so that they will be restored to their initial state. Similarly, if it is the last vertex of the strand, the last control point will be

repeated twice, because the last point does not have a successor. In any other case, the indices will be used in order.

In the setup stage before the draw call, an index buffer (or EBO in OpenGL) must be created to hold the indices of the guides. The draw call will be changed from chapter 3 to:

```
void Guides::Draw() {

        glBindVertexArray(VAO);

        glPatchParameteri(GL_PATCH_VERTICES, 4);
        glDrawElements(GL_PATCHES, (GLsizei)indices.size(),
                        GL_UNSIGNED_INT, 0);

        glBindVertexArray(0);
}
```

Here, a patch of four vertices is defined, and the glDrawArrays is changed to glDrawElements, with the size of the indices vector.

6.1.2. The Tessellation Shaders

Now that the indices are set up, it is time to introduce the tessellation control and evaluation shaders. The tessellation control shader is mostly pass-through, and the amount of tessellation is decided in this stage:

```
#version 430

layout (vertices = 4) out;

float uOuter0 = 20.0;

// number of spline segments
float uOuter1 = 3.0;

in VS_OUT{

        float thickness;
        vec3 point;
        vec3 normal;

}vs_out[];

out CS_OUT{

        float thickness;
        vec3 point;
        vec3 normal;

} cs_out[];


void main( )
{

cs_out[gl_InvocationID].point = vs_out[gl_InvocationID].point;
```

```
cs_out[gl_InvocationID].thickness = vs_out[gl_InvocationID ].thickness;
cs_out[gl_InvocationID].normal = vs_out[gl_InvocationID].normal;

        gl_TessLevelOuter[0] = uOuter0;
        gl_TessLevelOuter[1] = uOuter1;
}
```

The first tessellation level defines the amount of additional hair that will be created. The current section will only focus on the second parameter which is the number of extra points that will be created per patch.

The tessellation evaluation shader below, takes into account the first barycentric coordinate from the tessellator, which defines the position of the subdivided primitive patch:

```
    #version 430
layout( isolines, equal_spacing) in;

uniform mat4 view;
uniform mat4 model;

in CS_OUT{
        float thickness;
        vec4 point;
        vec3 Normal;
}es_in[];

out ES_OUT{
        float thickness;
        vec4 point;
        vec3 Normal;
} es_out;

vec4 B_spline(vec3 p0, vec3 p1, vec3 p2, vec3 p3,
                float tessCoordX){

        float u = clamp(tessCoordX, 0.0, 1.0);

float b0 = (-1.0*u*u*u + 3.0*u*u - 3.0*u + 1.0) * (1.0/6.0);
float b1 = (3.0*u*u*u - 6.0*u*u + 4.0 ) * (1.0/6.0);
float b2 = (-3.0*u*u*u + 3.0*u*u + 3.0*u + 1.0) * (1.0/6.0);
float b3 = (u*u*u) * (1.0/6.0);

        return vec4((b0*p0 + b1*p1 + b2*p2 + b3*p3) , 1.0);
}

void main( )
{
        vec3 p0 = es_in[0].point;
        vec3 p1 = es_in[1].point;
        vec3 p2 = es_in[2].point;
        vec3 p3 = es_in[3].point;

        vec4 position = B_spline(p0,p1,p2,p3,gl_TessCoord.x);
        es_out.point = position;
}
```

In the simplified code above, the four control points are accessed and passed into a function that computes the position of the new point, using the basis functions of chapter 6.1. The t parameter is the gl_TessCoord.x. Figure 6.3 shows the result of tessellating the guide

strands into smooth cubic b-splines.

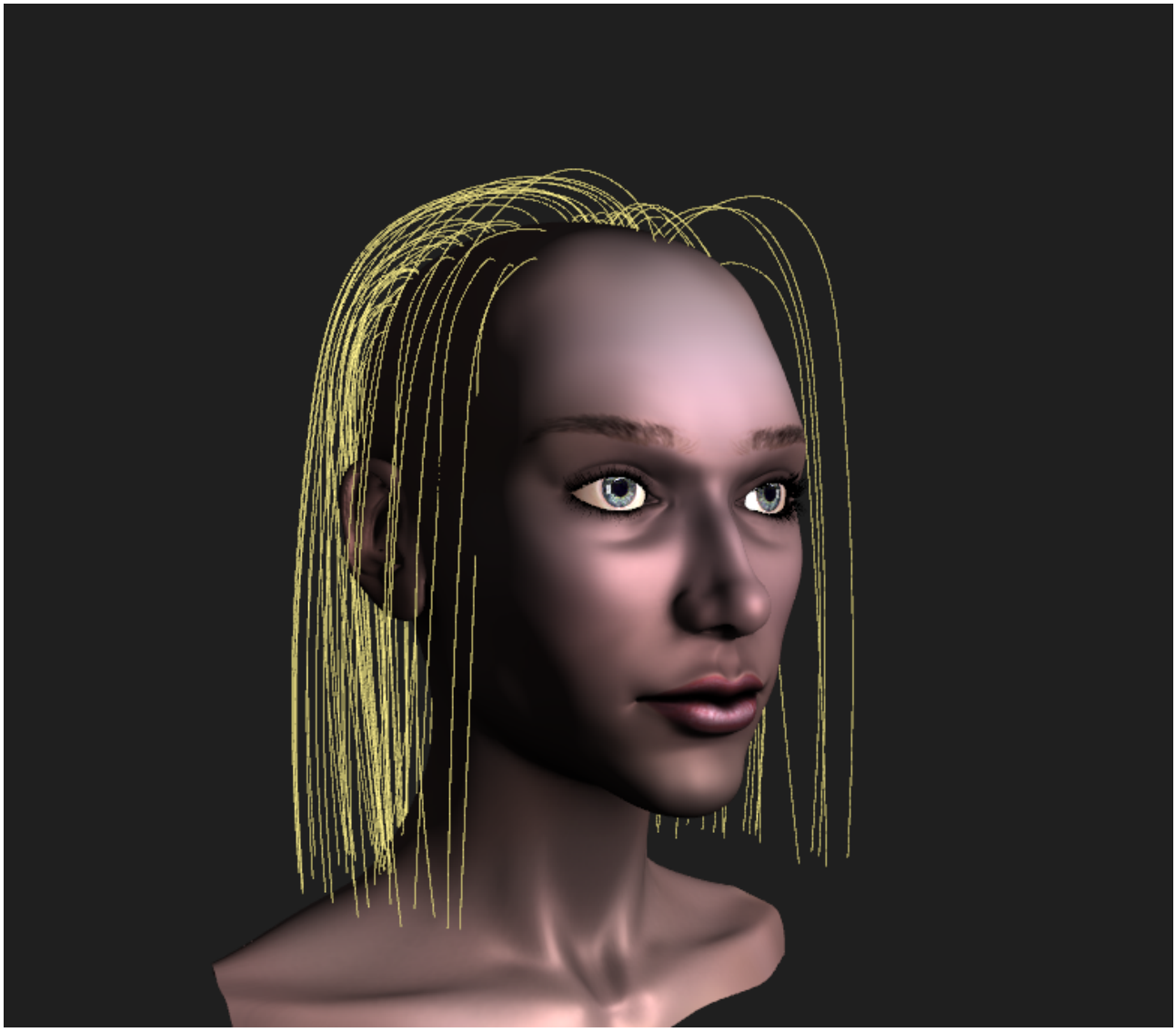

Figure 6.3: Rendering the guide strands as cubic b-splines using hardware tessellation.

6.2. Rendering Additional Hair

As it has been already explained in the previous chapter, the second barycentric coordinate from the tessellator describes which line was generated from the currently selected guide strand. This newly generated line must be placed randomly along the surface of the growth mesh. One elegant solution is to place the new strand in a random position along a circle with the root of the guide as centre and a predefined radius. Figure 6.4 depicts this interpolation method.

The green spots of Figure 6.4 show the random positions of the generated copies of the guide strand. This technique is also called ”single-strand interpolation”, because the additional hair depend only on one master guide strand.

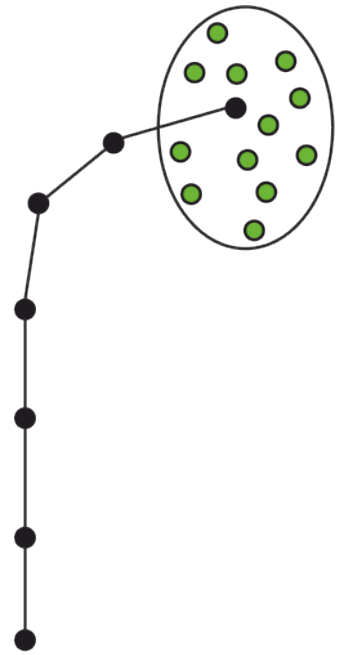

Figure 6.4: Interpolating newly generated strands randomly inside a circle defined by the guide strand.

6.2.1. Expanding the Tessellation Shaders

This section completes the tessellation evaluation shader, by utilizing the second coordinate of the tessellation stage in order to interpolate hair that are generated on the fly. Below is the complete code, assuming that the cubic b-spline step has already been implemented:

```
// ... same as before ...

float rand( vec2 p )
{
    return fract(sin(dot(p,vec2(12.9898,78.233)))
                 * 43758.5453);
}

void main( )
{
        vec3 p0 = es_in[0].point;
        vec3 p1 = es_in[1].point;
        vec3 p2 = es_in[2].point;
        vec3 p3 = es_in[3].point;

        vec4 position = B_spline(p0,p1,p2,p3,gl_TessCoord.x);

        vec3 norm = normalize(vec3(-es_in[0].normal.x,
               es_in[0].normal.y,0.f));

        vec3 norm2 = cross(es_in[0].normal,norm);



        // Single strand interpolation
    vec4 finalPosition = position + vec4(r * cos(theta)*norm
           + r * sin(theta)*norm2,0.0) ;

         // Apply noise to offset position.
```

```
        float noise_ = rand(vec2(gl_TessCoord.y,1.0));


        finalPosition.x += noise_ *  0.5;
        finalPosition.y += noise_ * 0.5;
        finalPosition.z += noise_ * 0.5;


es_out.point =  view * model * vec4(finalPosition.xyz,1.0);
}
```

First, the curve is smoothed using cubic b-splines, as explained by the previous section. After that, two quantities are calculated. The first one chooses only the two coordinates from the point's normal and the second one is the cross product between the normal vector of the point and the previously computed quantity. These two vectors will be used to determine the position of the new strand, which is calculated with spherical coordinates. A random noise is also applied to the position, in order to make the hair appear less uniform. Figure 6.5 depicts the final result, while Figure 6.6 shows the performance difference from rendering the hair as described in chapter 3, versus generating hair on the fly. With this technique, roughly 200K additional vertices can be computed with performance gains!

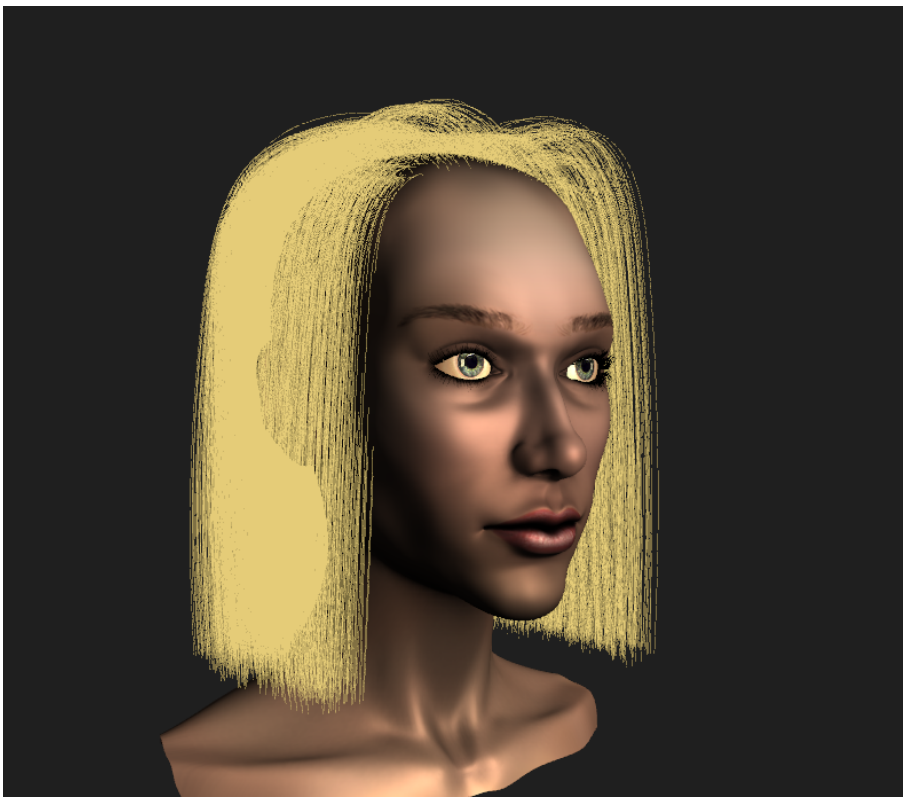

Figure 6.5: Final tessellated result. Smooth curves and on the fly hair generation.

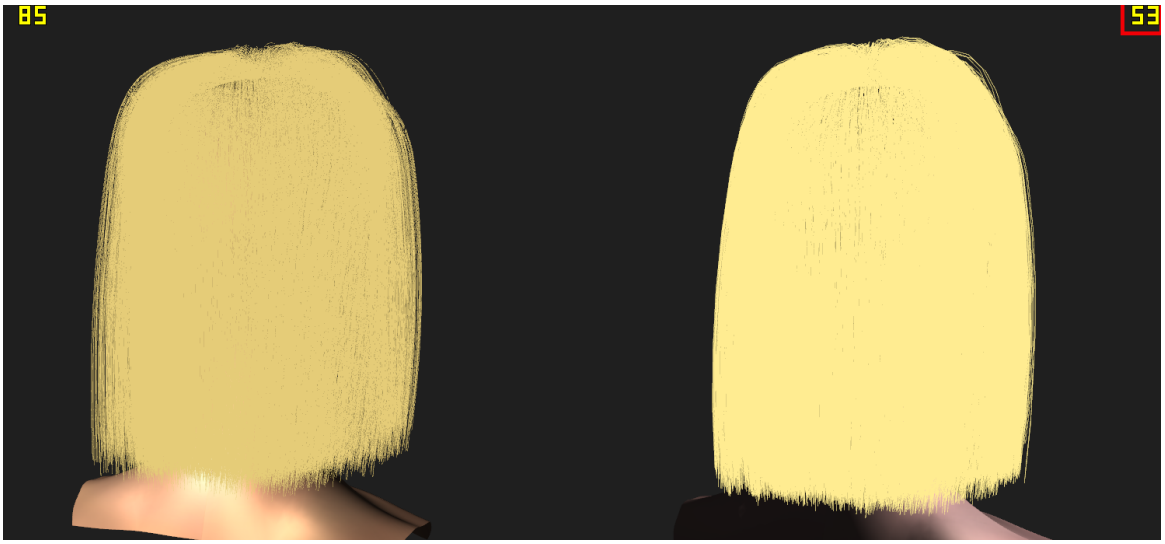

Figure 6.6: Performance comparison. Brute force technique on the right and tessellation on the left. Notice the huge performance advantage.

7. LIGHTING MODELS

After the tessellation stage has been completed, the document proceeds with exploring how light reacts with hair fibers. Two of the most popular lighting models are described, each with its own advantages and drawbacks. The pixel shader functions for each lighting method are presented, and performance measurements are derived. In the current framework, only one point light source is implemented.

7.1. Kajiya-Kay Model

In 1989, Kajiya&Kay proposed a simple lighting model for hair rendering. This model conists of two components: diffuse and specular.

The diffuse component accounts for the light that scatters within the hair volume and reflects to every direction equally. The diffuse reflection depends on the non negative dot product of the hair normal and the view vector.

The specular component captures the primary highlights of hair when illuminated. The specular reflection can be calculated by the Phong model, by using the dot product of the hair normal and the half vector, raised to a power which is referred as ”shininess”.

The pixel shader function below calculates the diffuse and specular component based on the Kajiya&Kay model.

```
vec3 KajiyaKay() {

    vec3 L = normalize(lightPos - posWorld.xyz);
    vec3 E = normalize(camera_position - posWorld.xyz);
    vec3 H = normalize(L + E);

    float NdotL = max(dot(normalize(Normal), L), 0.0);
    float NdotH = max(dot(normalize(Normal), H), 0.0);

    float specular = pow(NdotH, 7.0) ;
    vec3 hairColor = texture(hairColorTexture,(texCoord)).rgb;

    vec3 diffuse =  hairColor * NdotL;
    vec3 ambient = vec3(0.1) * hairColor;

    vec3 finalColor = ambient + diffuse + specular;
    return finalColor;
}
```

The ambient term simulates indirect illumination from hair-hair interaction, when no light is facing the hairstyle. Figure 7.1 shows the rendered result, as well as performance measurement. The Kajiya&Kay model is reasonably cheap to compute.

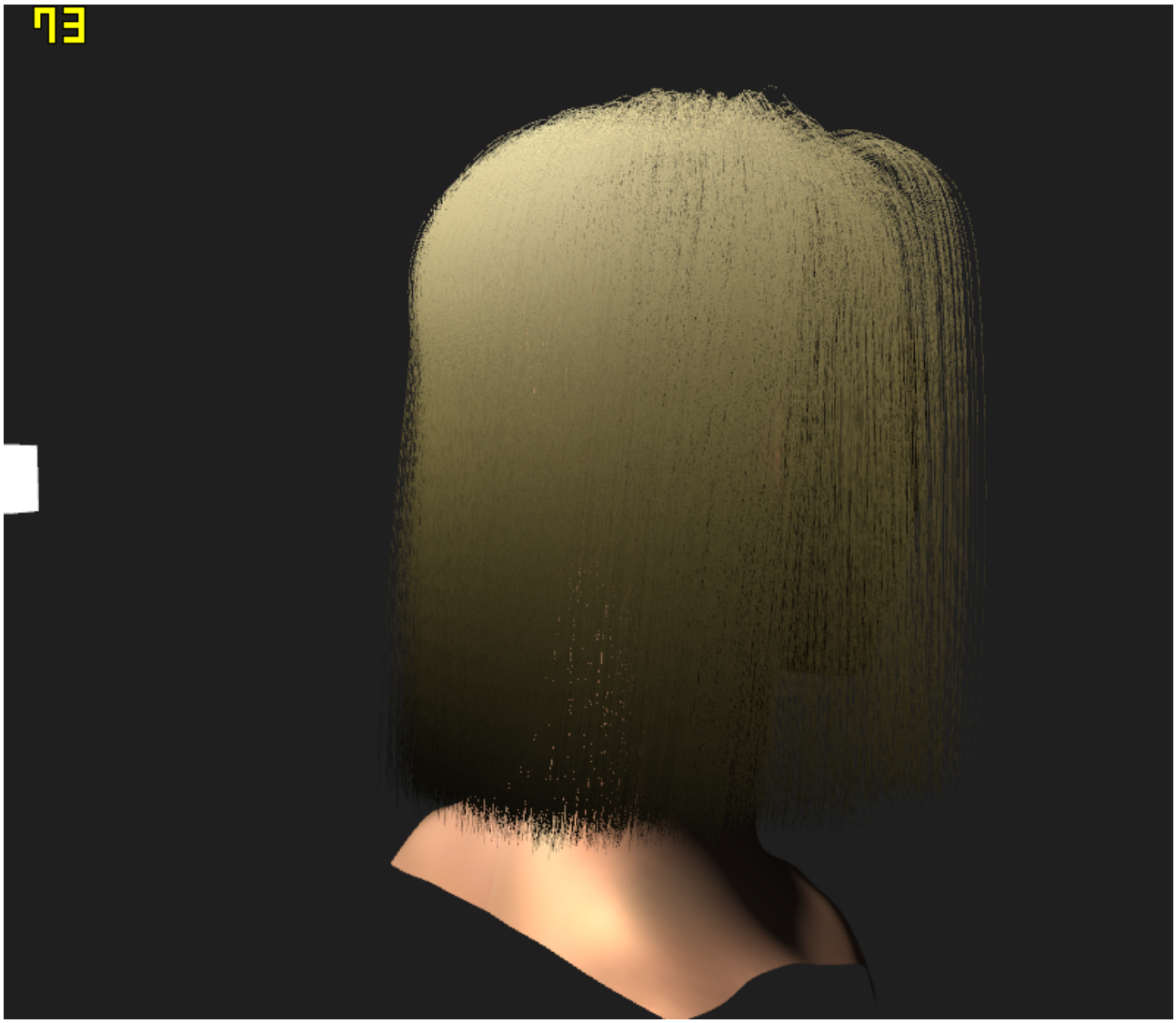

Figure 7.1: Kajiya&Kay lighting model. The primary specular highlights are visible.

7.2. Marschner Model

Even though the previous lighting model is cheap to calculate, it does not look as realistic as Marschner's model. Marschner et al. [6] observed the physical properties of hair scattering, realizing that there are also secondary specular highlights when light interacts with hair. A model was developed that treats each hair fibre as a thin cylinder, where light not only reflects, but also refracts inside the fibre. Figure 7.2 depicts the process of light scattering in a single fiber. In this model, there are three components that describe the behavior of the light:

- **Reflection (R):** The reflection component accounts for the primary specular highlight of the hair, as the light is immediately reflected.
- **Transmission (TT):** The transmission component exists due to the fact that hair fibers consist of translucent material. One property of light interaction with such materials, is that a portion of light exits the fibre with a specific angle.
- **Transmission-Reflection-Transmission (TRT):** The second property of translucent materials, is that a portion of light will reflect at the inner border of the fibre and will refract in an angle. Visually, this component gives the secondary specular highlight.

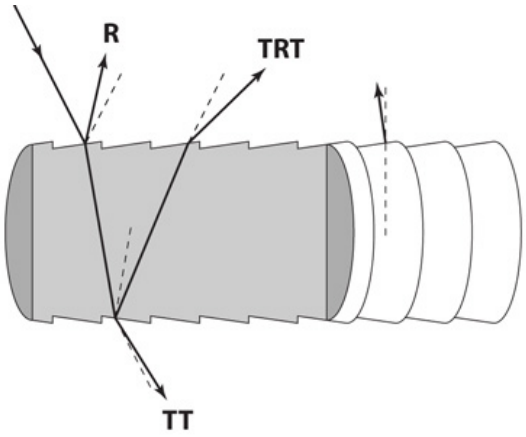

Figure 7.2: Light scattering in hair fibre. Three components are presented: reflection, refraction and transmission

The function below calculates the parameters for the Marschner model.

```
vec3 Marschner(){

    float shift = texture(specularTexture,texCoord).r;
    vec3 N = shiftedTangent(Tangent,Normal,shift);

    vec3 lightDir = normalize(lightPos - posWorld.xyz);
    vec3 viewDir = normalize(camera_position - posWorld.xyz);

    vec3 V = normalize(viewDir);
    vec3 L = normalize(lightDir);

    float NdotV = max(dot(N, V),0.0) ;
    float NdotL = max(dot(N, L),0.0);

    vec3 hairColor = texture(hairColorTexture,(texCoord)).rgb;

    vec3 diffuse = hairColor * NdotL ;

    float primaryHighlightIntensity = 0.3;
    float secondaryHighlightIntensity = 0.38;

    float theta = asin(NdotL) + asin(NdotV);

    vec3 term1 = normalize(L-NdotL*N);
    vec3 term2 = normalize(V-N*NdotV);
    float cosPhi = dot(term1,term2);

    vec3 R_ = vec3(clamp(primaryHighlightIntensity *
        pow( abs(cos(theta - 64.0)) , 2.0) , 0.0, 1.0)) ;

    vec3 TRT = vec3(clamp(secondaryHighlightIntensity *
        pow( abs(cos(theta - 64.0)), 128.0 ),0.0, 1.0));

    float rimIntensity = 0.1;
    vec3 TT = vec3(clamp(rimIntensity * max(0.0, cosPhi) *
        pow( cos(theta - 2.0) , 2.0 ), 0.0, 1.0));

    vec3 specular =  R_ + TRT + TT;

    float ambientStrength = 0.1;
```

```
    vec3 ambient = hairColor * ambientStrength;

    vec3 color = ambient + diffuse + specular;
    return clamp(color * lightColor,0.0,1.0);
}
```

First, the normal vector of the point is slightly shifted in order to create a variation that human hair naturally have. The function that shifts the normal with the help of the tangent vector is very simple and can be defined as:

```
vec3 shiftedTangent(vec3 T, vec3 N, float shift){
        vec3 shiftedT = T + shift * N;
        return normalize(shiftedT);
}
```

The tangent vector is calculated in the geometry shader as the model-space coordinates of the tangent that was used to expand the line:

```
Tangent = (model*vec4(tangent,1.0)).xyz;
```

Continuing with the Marschner model, the view and light directions are calculated, as well as their angles to the normal vector. The diffuse component is the same as the Kajiya&Kay model. The specular component consists of the three components described earlier. The spherical coordinates θ and ϕ depend on the angle of the viewing direction and light direction. The primary specular highlight is calculated as the power of the cosine of θ decreased by a user-defined amount. The smaller the exponent, the wider the specular lobe is. The secondary specular highlight is calculated the same way as the primary highlight, with the main difference being the exponent number. The higher the exponent, the narrower the specular lobe will be, resulting in sharp secondary reflections. Finally, the transmission component can be calculated similarly, but transmission also depends on the cosine of ϕ. The total light contribution from ambient,diffuse and specular is scaled by the light's color and returned to the caller. Figure 7.3 shows the result of Marschner's lighting model. The quality is better than the previous model, albeit with a slight performance degradation.

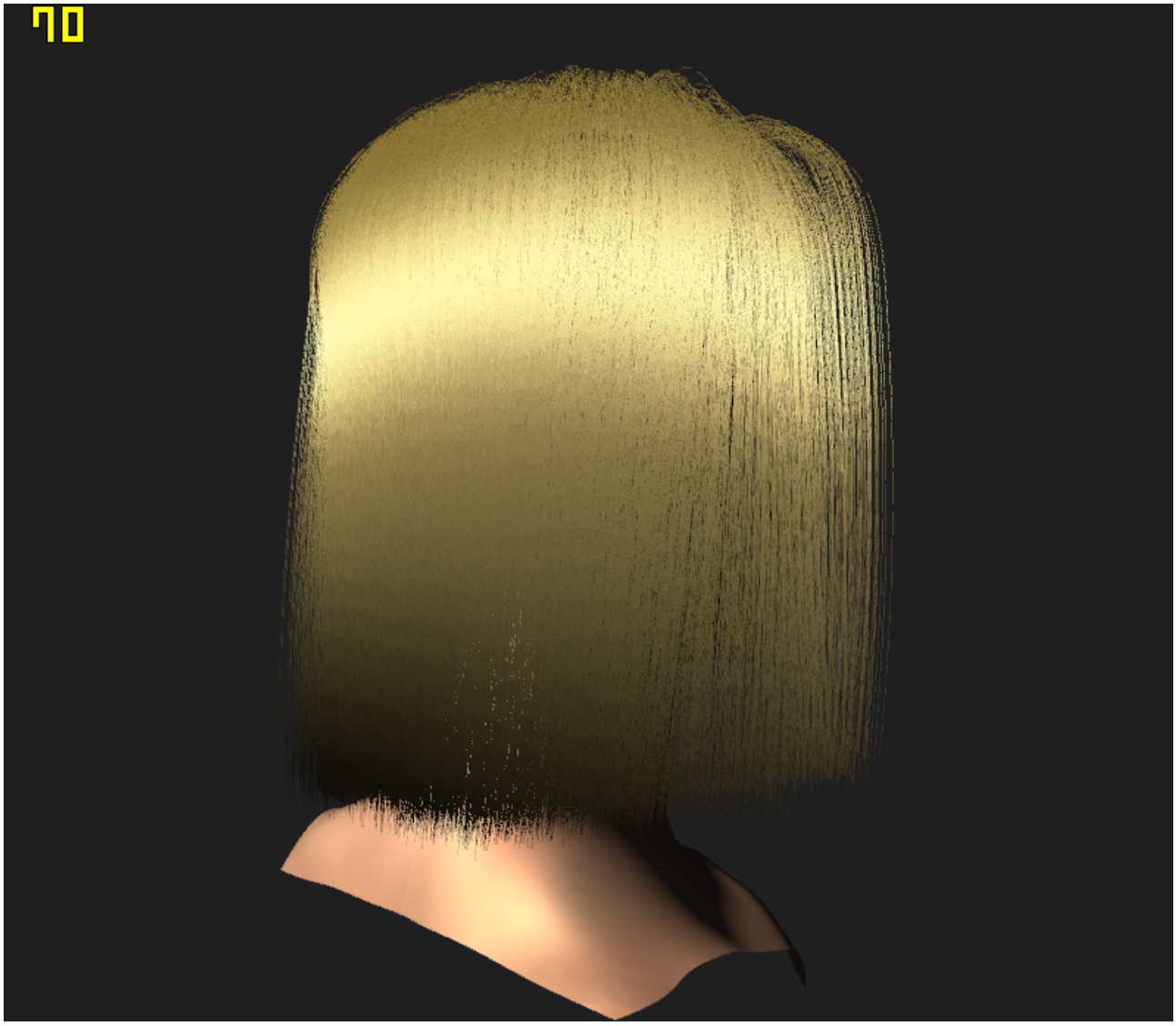

Figure 7.3: Marschner's lighting model without shadows. Notice the secondary specular highlights.

8. HAIR SHADING

The final step of the framework is to apply shadows to the hairstyle. In traditional rasterized computer graphics, direct shadows can be computed by first generating a depth map. This process simply renders the scene from the light's perspective, and stores the Z coordinate of the closest geometry. Afterwards, in the main render pass, the world position of the object is transformed into light space, and the result is compared with the depth texture. If the current depth is greater than that of the depth texture, then the fragment is in shadow.

The depth texture can be generated as any other texture in OpenGL, with the main difference being that the texture type must be defined, by using the GL_DEPTH_COMPONENT specifier. In order to transform the vertices into the light's point of view, the light space matrix can be generated as:

```
lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f,
                  nearPlane, farPlane);

lightProjection[2][2] *= -1.0f; // Flip the z-axis

lightView = glm::lookAt(lightPos, glm::vec3(0.0f),
           glm::vec3(0.0, 1.0, 0.0));

lightSpaceMatrix = lightProjection * lightView * light_mMat;
```

The light space matrix is the product of three matrices. The light model matrix simply holds the position of the light. The light view matrix can be computed by using the lookAt() function of the light's position and the positive Y axis. The light projection matrix describes an orthographic projection with predefined planes. Also, the Z axis of the projection matrix is flipped in order to have correct depth values.

The depth texture generation needs a new set of shaders. The scene will be rendered first with these new shaders. It is important to capture as much detail in the depth map as possible. Since the framework uses only a very small subset of strands as guides and generates additional hair in the tessellation shader, this stage should also be included in the depth map generation. The depth shaders are very similar to the shaders introduced earlier, only much more simplified. In the geometry shader, the generated vertices are multiplied with the light projection matrix. The fragment shader simply passes the depth value as follows:

```
void main()
{
        gl_FragDepth =  gl_FragCoord.z ;
}
```

In the tessellation control shader of the depth pass, the same number of generated lines as that of the main render pass is recommended to be used, in order to have accurate shadows. However, the framework can get away with detail regarding the number of segments in the line. A depth texture can be generated without a smooth tessellated line. Figure 8.1 shows the depth map from the light's point of view, assuming the light is positioned behind the head.

Using a binary decision to determine a shadowed pixel, can result in undesired visual artifacts such as shadow acne. Two of the most popular techniques to reduce such artifacts are presented in the sections below.

Figure 8.1: Depth texture from the light's perspective.

8.1. Percentage Closest Filtering

In order to have high quality shadows and avoid pixelated shadows, a shadow map with high resolution must be generated. However, this is very costly, especially in the context of hair shadowing. An alternative way is to filter the shadow map, creating soft edges. The Percentage Closest Filtering (PCF) algorithm works by sampling multiple locations within the shadow map for each pixel and blending the results together. The percentage of shadowed samples is then used to determine the final shadow intensity at the pixel.

In the main render pass, after the depth map has been generated, a texture fetch is applied to determine the value that will be compared to the current depth. Using a neighborhood around the fetched texel with a user defined radius, the algorithm decides whether the neighboring pixel is in shadow or not. The total percentage of shadow in the original pixel is equal to the number of shadowed samples divided by the total number of samples.

The GLSL function below implements the PCF algorithm.

```
float PCFShadowCalculation()
{

    // perform perspective divide
    vec4 sCoord = lightSpaceMatrix * posWorld;
    vec3 projCoords = sCoord.xyz / sCoord.w;

    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
```

```
    // get closest depth value from light's perspective
    float closestDepth = texture(shadowMap, projCoords.xy).r;

    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;

// calculate bias
    vec3 normal = normalize(Normal);
    vec3 lightDir = normalize(lightPos - posWorld.xyz);
    float bias = max(0.05*(1.0 - dot(normal,lightDir)),0.005);

    float shadow = 0.0;
    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
    for(int x = -PCFKernel; x <= PCFKernel; ++x)
    {
        for(int y = -PCFKernel; y <= PCFKernel; ++y)
        {
            float pcfDepth = texture(shadowMap,
            projCoords.xy + vec2(x, y) * texelSize).r;
            shadow += pcfDepth;
        }
    }
    shadow /= float(pow((PCFKernel * 2) + 1 , 2));

    float visibility = 1.0;

    if (currentDepth > shadow + bias) {
        float shadowDist = currentDepth - shadow;
        float maxShadowDist = 0.1;
        visibility = 1.0-smoothstep(0.0,maxShadowDist,shadowDist);
    }

    return visibility;
}
```

First, the world position needs to be transformed into light space and perform a perspective division. The projection coordinates are normalized and used to fetch the closest depth value. A small bias value is calculated to be used in the shadow decision making process later. In this implementation, the PCFKernel value is chosen to be three, as it offers a good balance between performance and quality. The shadow percentage is calculated, as well as the final visibility term. This function can be called inside each lighting model. For example, for the Marschner model, the shadow calculation can change the function defined in chapter 7 as:

```
vec3 Marschner(){
    ... same as before ...

    float shadow = 1.0 - PCFShadowCalculation();

    vec3 color = ambient + shadow * (diffuse + specular);
    return clamp(color * lightColor,0.0,1.0);
}
```

A similar approach can be done for the Kajiya&Kay function. Figures 8.2 and 8.3 , show the result of PCF for the two lighting models. The framerate manages to stay above 30 fps.

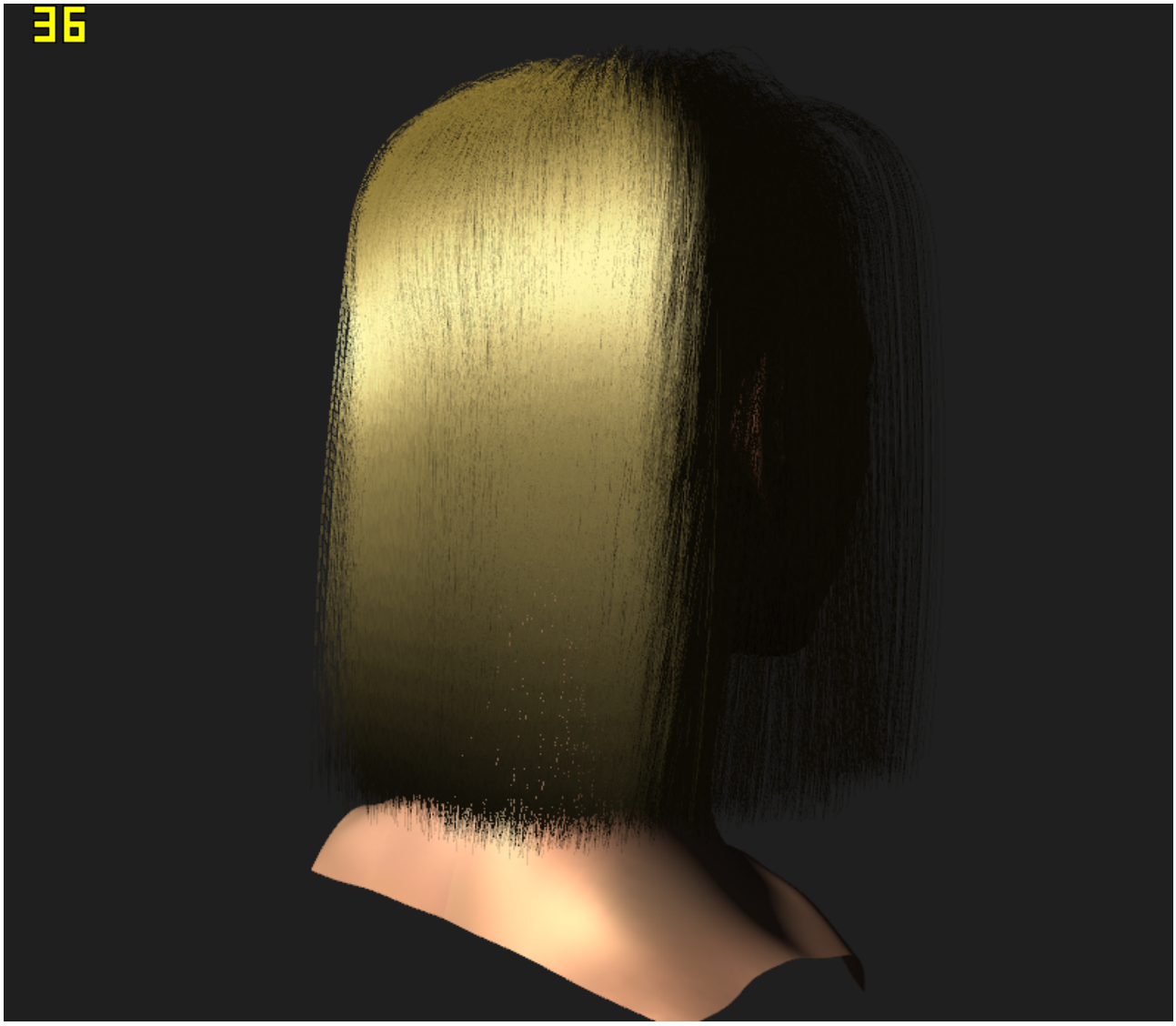

Figure 8.2: Marschner's lighting model with PCF shadow calculation.

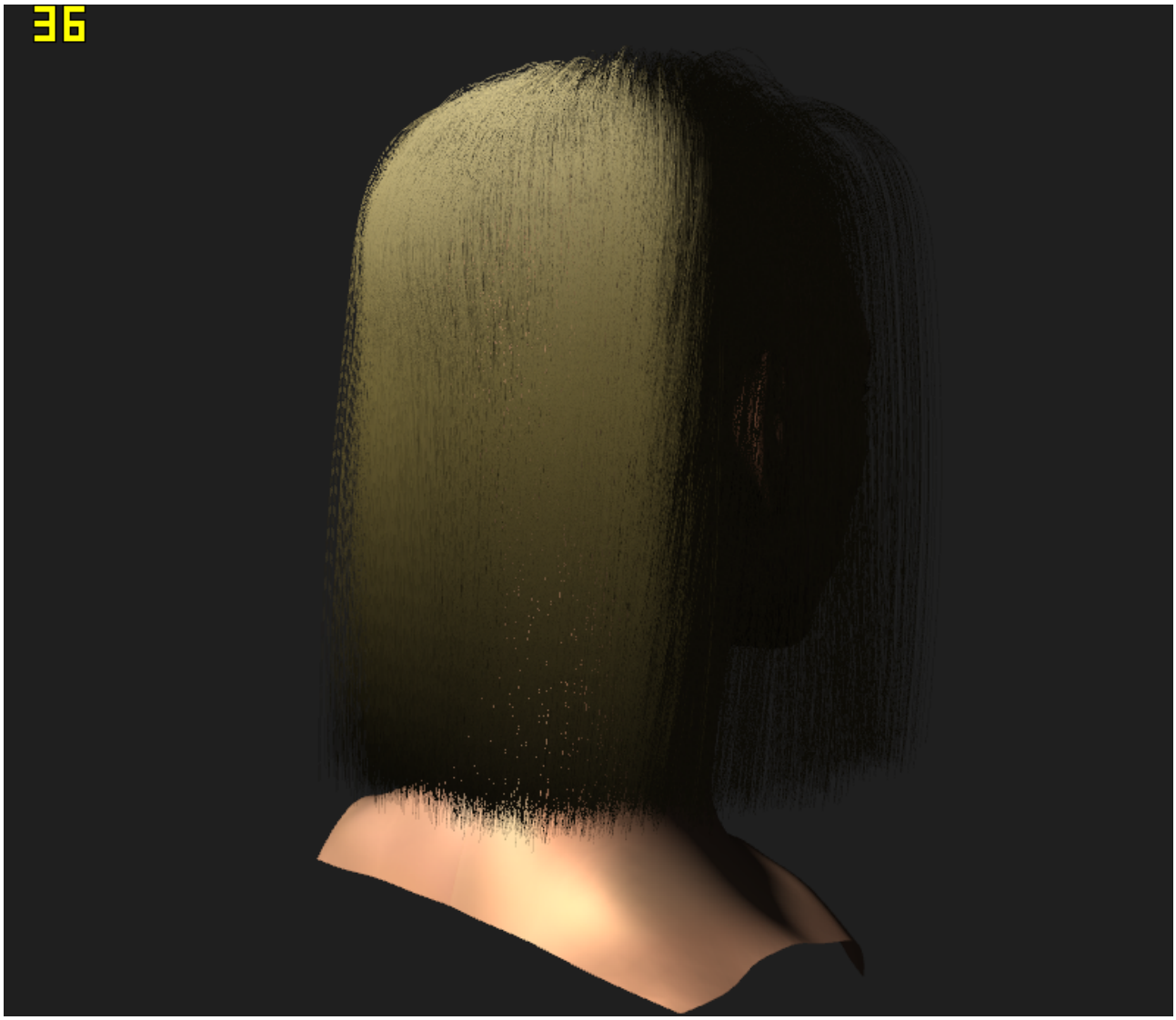

Figure 8.3: Kajiya&Kay lighting model with PCF shadow calculation.

8.2. Variance Shadow Mapping

Another popular technique for shadow mapping is Variance Shadow Map (VSM).The main feature of VSM is introducing probabilistic sampling and filtering techniques. Instead of storing just the depth information in the shadow map, VSM stores the mean depth and the squared depth. These values are used to calculate the mean and variance of the depths within a certain area of the shadow map, typically using a Gaussian filter. During the final rendering pass, when testing if a fragment is shadowed or not, multiple samples are taken from the shadow map within a region around the current fragment. These samples are then used to calculate the mean and variance of the depths. The final decision on whether a fragment is shadowed or not is made by comparing the depth of the fragment with the mean and variance values obtained from the shadow map samples. The comparison takes into account the uncertainty or variation in the depths, resulting in smoother transitions between shadows and illuminated areas.

The first step is to render the hairstyle again after the depth pass. This time, the depth texture will be input to the VSM pass fragment shader. Every other shader is exactly the same as the depth pass. The fragment shader of VSM can be described by the code below:

```
void main()
{
        vec4 sCoord = lightSpaceMatrix * posWorld;
    vec3 shadowCoord = sCoord.xyz / sCoord.w;
        shadowCoord = shadowCoord * 0.5 + 0.5;

    float depth = texture(shadowMap, shadowCoord.xy).r;

    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
    float dx = dFdx(depth) * texelSize.x;
    float dy = dFdy(depth) * texelSize.y;

    float filteredDepth = depth;
    float filteredVariance = depth*depth + (dx*dx + dy*dy) / 4.0;

    for (int x = -1; x <= 1; ++x)
    {
        for (int y = -1; y <= 1; ++y)
        {
            vec2 texCoord = shadowCoord.xy +
                    vec2(float(x), float(y)) * texelSize;
            float depthSample = texture(shadowMap, texCoord.xy).r;

            dx = dFdx(depthSample) * texelSize.x;
            dy = dFdy(depthSample) * texelSize.y;
            float varianceSample = depthSample*depthSample +
                    (dx*dx + dy*dy) / 4.0;

            filteredDepth = min(filteredDepth, depthSample);
            filteredVariance=min(filteredVariance,varianceSample);
        }
    }
        variance = vec2(filteredDepth,filteredVariance);
}
```

A new texture is generated that stores the filtered depth in the first channel and the filtered variance in the second. The filtered variance is calculated by finding the partial derivatives

across the X and Y coordinates of the depth texture, and selecting the minimum result for every sample iteration. Figure 8.4 shows the variance shadow map, assuming the light is located behind the hairstyle.

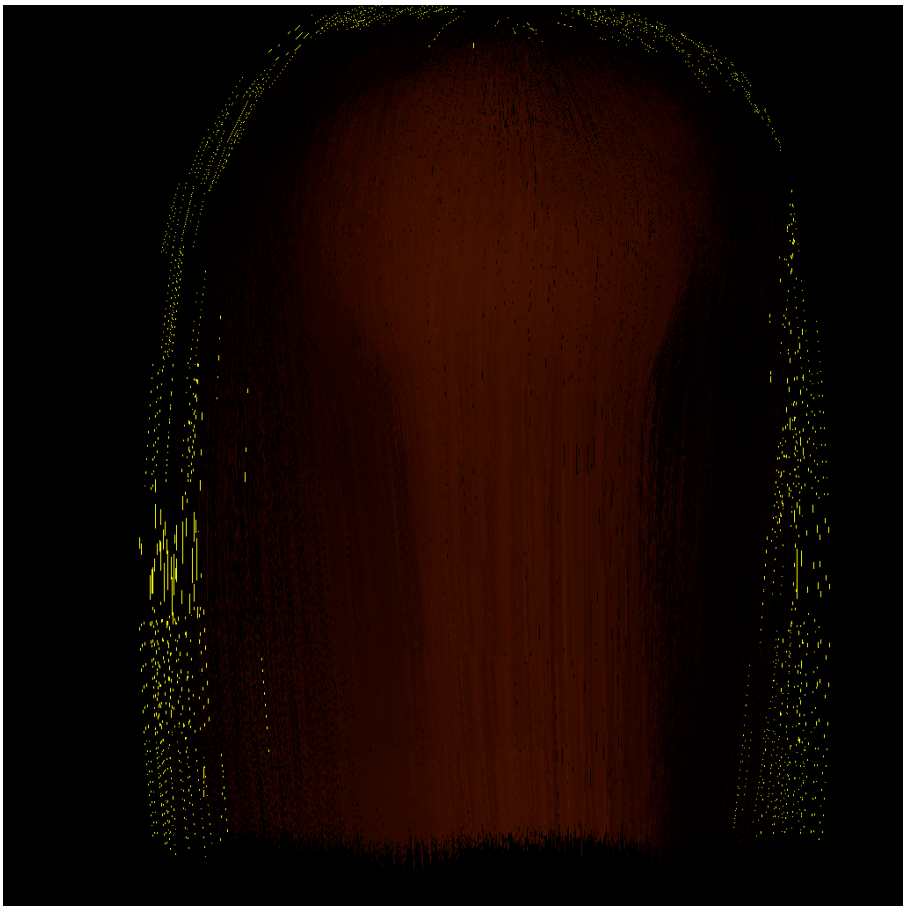

Figure 8.4: Variance Shadow Map from the light's perspective

In the main rendering pass, the variance map can be accessed, and by computing the Chebyshev upper bound, the maximum variance is returned as the percentage of shadow. The following function implements the VSM shadow calculation in the main render pass:

```
float ChebyshevUpperBound(vec2 Moments, float t) {
        // One-tailed inequality valid if t > Moments.x
        float p = (t <= Moments.x) ? 1.0f : 0.0f;
        // Compute variance.
        float Variance = Moments.y - (Moments.x * Moments.x);

        float g_MinVariance      = 0.0007;

        Variance = max(Variance, g_MinVariance);
         // Compute probabilistic upper bound.
        float d = t - Moments.x;
         float p_max = Variance / (Variance + d*d);
         return max(p, p_max);
}

float VSMShadowContribution() {
   vec4 sCoord = lightSpaceMatrix * posWorld;
   vec3 projCoords = sCoord.xyz / sCoord.w;
   // transform to [0,1] range
```

```
    projCoords = projCoords * 0.5 + 0.5;

    // Read the moments from the variance shadow map.
    vec2 Moments = texture(varianceMap, projCoords.xy).xy;
    // Compute the Chebyshev upper bound.
    return ChebyshevUpperBound(Moments, projCoords.z);
}
```

The two values of the variance shadow map are also referred to as ”moments” in the terminology. By tweaking the minimum variance value in the Chebyshev function, the scope of the shadow can be changed. Figures 8.5 and 8.6 show the final result of VSM with the two aforementioned lighting models.

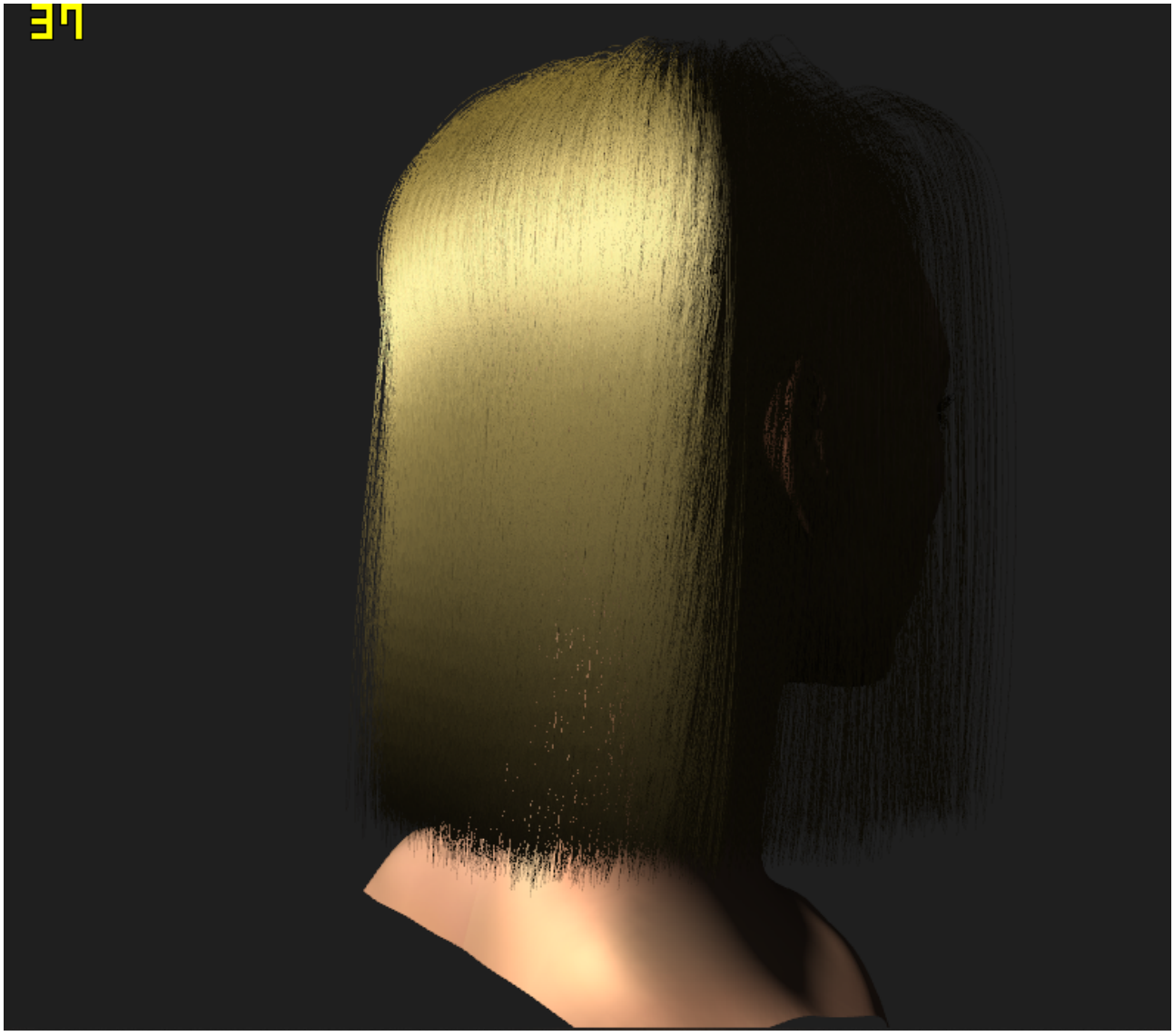

Figure 8.5: Marschner’s lighting model with VSM shadow calculation.

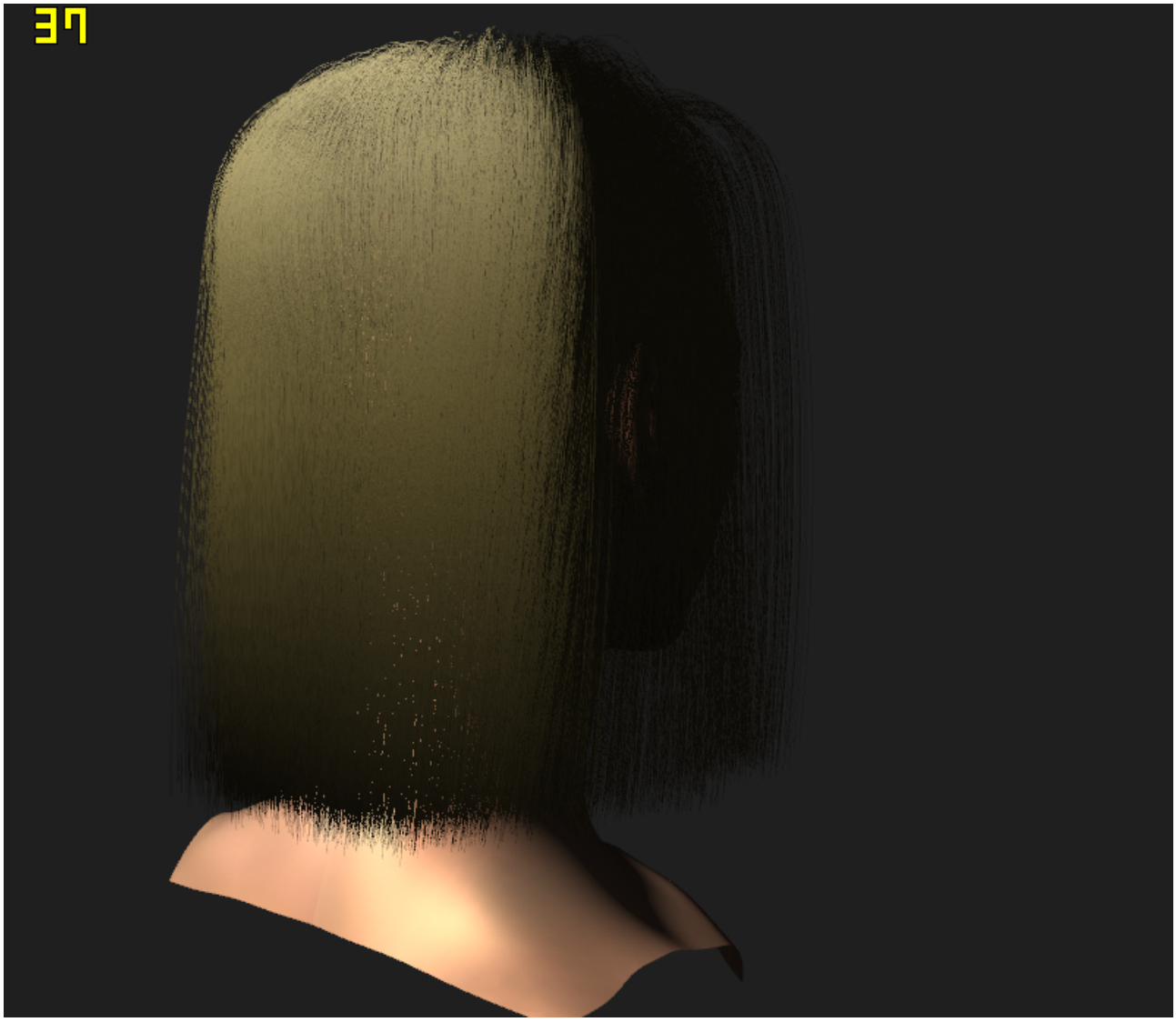

Figure 8.6: Kajiya&Kay lighting model with VSM shadow calculation.

9. DISCUSSION AND FUTURE WORK

In this document, the process of rendering hair on the GPU efficiently was thoroughly described, as well as algorithms for realistic lighting and shadowing. The framerate manages to stay real-time for an 8 year old low-level laptop GPU. Additional optimizations could have reduced the rendering time even lower, but since the initial goal was achieved, they will only be described as a potential future work. These optimizations contain:

- **Dynamic LOD**: The tessellation control shader can parametrize the amount of tessellation depending on the camera distance. The further away the camera is from a point in the strand, the less hair will be generated and there will be less detail in the B-spline calculation.
- **Lower Detail Shadow Maps**: The shadow map can contain less tessellation amount than rendering every single strand as in the main rendering pass, if the application can allow for some visual degradation.
- **Culling**: Back face culling can be applied to save performance. However, this must be used with caution in case of hair strands that are partially visible.

The real-time achievement of hair rendering can be especially important for mobile GPUs, where resources are limited. For example, a video game consists of tens of thousands of polygons per frame. Adding to that the calculations for phenomena such as weather, physics, and artificial intelligence, the rendering budget becomes more and more limited. A hair rendering system presented by this document can be used for a video game, if additional measures are taken to ensure a higher framerate, such as the above steps. It is important to note that for old and limited GPUs, it may be possible to have a strand-based renderer only for the protagonist of the video game. Since most of the time the camera will mostly be located in a greater distance than the images of this document, the framerate will be higher. Another use for effective real time hair rendering is VR/AR applications. Performance in VR games is of high importance, and a cheap way to render realistic hairstyles can be proven very useful. Since VR games typically require a strong GPU, the performance of a strand based renderer will be higher than the one implemented in this document. As a result, multiple characters can have realistic hair in real time!

Additional future work can be made to explore more realistic shadowing techniques. More specifically, Deep Opacity Maps [12] is a very well known technique for hair rendering, producing realistic shadows in real time. Ray tracing can bring real-life accuracy in terms of shadows, but this solution ignores the constraint for low-end GPUs that was set by this document.

9. REFERENCES

[1] GitHub Repository. *https://github.com/pantelis-kan/HairLoader*.

[2] cyCodeBase Hair Class Website. *http://www.cemyuksel.com/cyCodeBase/code.html#cyHair*.

[3] Sarah Tariq and Louis Bavoil. Real time hair simulation and rendering on the gpu. In *ACM SIGGRAPH 2008 Talks*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery.

[4] Iman Sadeghi, Heather Pritchett, Henrik Wann Jensen, and Rasmus Tamstorf. An artist friendly hair shading system. *ACM SIGGRAPH 2010 papers*, 2010.

[5] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. 23(3):271–280, jul 1989.

[6] Stephen R. Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, and Pat Hanrahan. Light scattering from human hair fibers. *ACM Trans. Graph.*, 22(3):780–791, jul 2003.

[7] Thorsten Scheuermann. Practical real-time hair rendering and shading. In *ACM SIGGRAPH 2004 Sketches*, SIGGRAPH '04, page 147, New York, NY, USA, 2004. Association for Computing Machinery.

[8] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *SIGGRAPH Comput. Graph.*, 21(4):283–291, aug 1987.

[9] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, page 161–165, New York, NY, USA, 2006. Association for Computing Machinery.

[10] Rasterization of Parametric Curves using Tessellation Shaders in GLSL. *https://computeranimations.wordpress.com/2015/03/16/rasterization-of-parametric-curves-using-tessellation-shaders-in-glsl/*.

[11] Patrikalakis-Maekawa-Cho, B-Splines. *https://web.mit.edu/hyperbook/Patrikalakis-Maekawa-Cho/node16.html*.

[12] Cem Yuksel and John Keyser. Deep Opacity Maps. *Computer Graphics Forum*, 2008.