Cartoon Hair Simulation

by

Timothy Costigan, BAI

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2013

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Timothy Costigan

August 28, 2013

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Timothy Costigan

August 28, 2013

Acknowledgments

I wish to thank my supervisor Rachel McDonnell for her valuable input.

Timothy Costigan

University of Dublin, Trinity College
September 2013

Cartoon Hair Simulation

Timothy Costigan
University of Dublin, Trinity College, 2013

Supervisor: Rachel McDonnell

The purpose of this project was to implement a real-time cartoon hair simulation pipeline for potential use in games and other interactive applications. The overall goal was for the hair to be appealing and to successfully emulate a cartoon artistic style.

The hair was composed of guide strands which were designed in a 3D modelling program and then used to generate a base mesh around each hair which could then have artistic stylisations applied. Depth dependent cartoon shading, back lighting and custom specular effects were implemented using an extended cartoon shader method where the x axis is indexed by light intensity and the y axis by a custom metric such as surface orientation or specular value. Surface hatching was applied using either viewport projection or surface projection. The hatching textures were generated in a custom program to ensure uniformly distributed strokes and tonal, border and depth coherency. The hair silhouettes were outlined using either an image-space method based on Sobel and a dilation pass or an object-space method which identified silhouette edges as line segments bordering front and back facing triangles. The object-space

silhouettes were used to generate camera facing quadrilateral strips which could have custom stroke textures applied.

The final product was tested for performance and user acceptance. Real-time performance was found to be achievable on modest hardware using the default settings, and users in general found the hair appealing although scores for effectively emulating the cartoon style were more neutral.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Hair simulation and rendering has been a popular research topic in the computer animation and graphics communities for over 20 years. However, most implementations over this period were strictly offline and required expensive custom hardware. In terms of applications, hair simulation would have most commonly been seen in large scale computer animated films where the studio could afford vast arrays of specialised computer hardware to perform the simulation and rendering.

In the last few years things have changed, with general purpose computers having become much more prevalent, cheaper and significantly more powerful. Thanks to the advent of multi-core processors, programmable graphics pipelines and the ability for graphics cards to perform large scale distributed general purpose computations, hair simulation has become possible in real-time on consumer level hardware.

Previously, in real-time applications when hair was required, a simple static mesh in the shape of a hair style would have to suffice. At most, a pony tail might move but this would have been animated through traditional key framing or bone skinning and certainly would not be simulated on a strand by strand level. Today in contrast, dynamic hair simulations are even starting to appear in computer games.

Graphical fidelity in video games has also improved significantly over the same period. Twenty years ago most game consoles could barely display primitive 3D graphics at playable frame rates while today some games can render at high frame rates, visuals that could be described as photo-realistic.

It is now expected for all modern games to look great and this can be problem for

smaller developers who cannot compete in terms of realistic graphics. Many studios as a result are instead now looking towards abstract cartoony art styles to help distinguish their game from the competition for example Crackdown™, Borderlands™and Okami™.

There is unfortunately very little research into dynamic cartoon hair. This will become a problem in future as dynamic hair becomes more and more common. The goal of this project is therefore to add to the currently small body of cartoon hair research by designing and implementing a cartoon hair simulation and rendering model along with its associated production pipeline.

This dissertation report will detail the entire process of implementing the project and its stated goals. The state of the art chapter will cover the investigation of the current hair simulation and rendering research. The design chapter will discuss the planning for the project application based on the information gleaned from research. The implementation chapter will discuss the details of enacting the design, and finally the evaluation and conclusion chapters will cover the assessment and discussion over whether or not this project's goals were actually achieved.

Chapter 2

State-of-the-art

In this chapter, we will review the various techniques for hair simulation and rendering that have been developed. While realistic methods are touched upon, emphasis is placed more on abstract techniques as the dissertation is concerned primarily with cartoon hair simulation.

The main difficulty with hair simulation is not the accurate recreation of strand behaviour (although this is by no means trivial) but rather the scale of the problem. The average human head can have over 100 thousand strands of hair [2]. If we naively attempted to simulate each and every strand including collisions, it could be very expensive to compute.

In terms of rendering, most research has been concerned with realistic hair, but there is an alternative branch of study known as NPR or Non-Photo Realistic rendering where the object in question is deliberately drawn in an unrealistic or abstract style. NPR research to date has not placed a large emphasis on hair.

2.1 Strand Dynamics

In terms of modelling individual strands, there is no real consensus on what is the best technique [3]. Numerous methods have been developed, all of which have their own advantages and disadvantages depending on the level of control we desire, the shape of the hair, the available hardware and so on.

2.1.1 Mass Spring Systems

Mass spring systems, first implemented by Rosenblum *et al.* [4] were one of the earliest hair simulation methods. In this technique, each hair strand is modelled as a set of particles which are connected using springs with one translational and two angular degrees of freedom. Mass spring systems are very simple to create and are easily constrained; however, there are certain limitations to a basic implementation.

Real hair for example, typically exhibits non-linear elastic behaviour and will retain its length when stretched but springs do not [3]. In order to emulate the rigid nature of hair using springs, the stiffness values must be increased. Very stiff springs will however, induce numerical instability if the simulation time step is too large (when using explicit numerical integration).

The bending of the hair can be restricted through the use of springs at each joint, but torsion is normally not accounted for, and in certain situations the hair could collapse to zero volume which would cause the simulation to fail [2].

These limitations can be alleviated somewhat through an alternative spring configuration proposed by Selle *et al.* [2]. In their method, instead of a simple chain of springs, a tetrahedral structure is used (referred to as 'tetrahedral spring configuration'). This structure is composed of edge springs, bending springs and torsion springs which constrain the stretching, bending and twisting of the hair segments (fig 2.1).

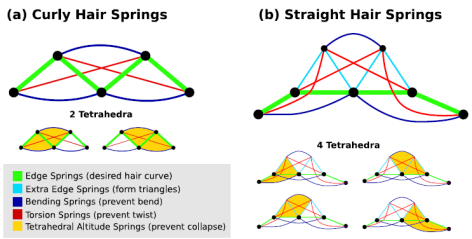


Figure 2.1: The tetrahedral spring configuration [2].

The tetrahedral configuration reduces stretching of the hair segments and prevents the hair collapsing to zero volume as the shape guarantees that no spring will reach an illegal state [2]. This method works for curly hair without modification, but for straight hair additional particles need to be created to attach the springs or else some springs will be initialised at zero volume and the simulation will fail.

In addition to tetrahedral springs, the length of the hair segments can also be preserved through the application of strain limiting [2]. Instead of increasing stiffness, strain limiting involves iterating through the hair's edge springs every cycle and applying velocity impulses to the particles furthest from the root to restore their length.

Overall, mass spring systems remain one of the more popular methods for hair simulation, having been used in films such as Pixar's Monsters Inc for fur simulation [5] and for key strand dynamics in paper's such as Noble *et al.* [6] and Shin *et al.* [7]. The main attraction seems to be the ease of constraining the springs as by altering the parameters for the hair's edge, bending and torsion springs, a greater degree of artistic control can be realised than with other methods.

2.1.2 1D Projective Equations

While springs can be easily used to model hair strands, they are not without their issues. Stiff springs, complex time integration and costly strain limiting are often used to overcome their various flaws. This can limit the number of possible strands, for example Selle *et al.* [2] was only able to reach about 10 thousand strands out of a stated goal of 100 thousand.

There is however, an alternative method using one dimensional projective equations that preserves the hair's length and is numerically stable. This method was first used by Anjyo *et al.* [8] and is based on the theory for the deformation of cantilever beams to model hair as a chain of rigid segments instead of loosely connected particles (fig 2.2). In 1992, it was possible to generate 20 thousand hair strands with this method.

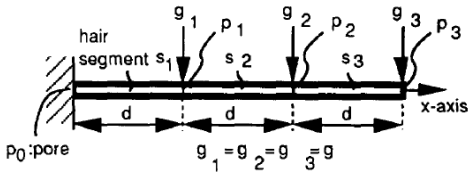


Figure 2.2: A cantilever beam [8].

The dynamics of the hair strands are implemented through the application of what is described as a 'pseudo-force field' [8]. The degree of bending the hair can undergo is controlled by adjusting the stiffness values of the cantilever beam equations. The hair

motion can also be made more realistic by introducing some uniform random numbers to the dynamics calculations to cause the hair to fluctuate subtly.

Torsion is however, not considered in this method, making it less desirable for simulating 3D hair motion [2, 8]. Additionally, only straight hair can be implemented and unlike mass spring systems, it is not trivial to apply constraints [2]. These restrictions, despite the low cost of the simulation may explain the technique's lack of popularity.

2.1.3 Rigid Multi-Body Serial Chains

If 3D motion is required, there is an alternative to one dimensional projective equations, that is numerically stable, length preserving and models torsional motion. This is the method of rigid multi-body serial chains, first proposed by Featherstone [9] for use in robot dynamics. Hair strands are modelled as chains of rigid segments connected by rotational and hinge joints with two degrees of rotational freedom and no translation [5].

The joint rotations are calculated using an exponential map, and by calculating the product of all proceeding exponential maps we can find the position of any hair vertex [5]. Each joint also has an associated actuator force in the form of a damped hinge. This actuator force helps model hair's naturally anisotropic (directionally dependent) behaviour and its torsional and bending rigidity. When coupled with breakable static springs, the actuator force can also help preserve the hair's shape after slight movement.

Unlike mass springs or one dimensional projective equations, rigid multi-body serial chains are capable of modelling not just simple hair but also complex arrangements like braids (fig 2.3) [5]. The method is however, a lot more costly to compute and is therefore better suited for simulating guide strands in a sparse hair model.

2.1.4 Dynamic Super Helices

Mass spring systems, one dimensional projective equations and rigid multi-body serial chains all attempt to produce hair that looks convincing regardless of whether or not the hair actually behaves realistically. The method of super helices is different however, as it tries to replicate the realistic properties of hair [10].

This method is based upon Kirchoff's equations for dynamic rods and models strands as piecewise helical rods animated using Lagrangian mechanics (fig 2.4) [10].

Figure 2.3: Rigid multi-body serial chains simulating a hair braid [5].

Using this technique, the non-linear properties of hair can be accurately simulated and the animator can instead rely on real life behaviour when styling the hair.

This method is numerically stable and produces very realistic results; however, it is not often used in real-time as it is very expensive computationally [10]. Only around 10 strands can be simulated in real-time on a 3 GHz Pentium 4 with a time complexity of $O(n^2)$.

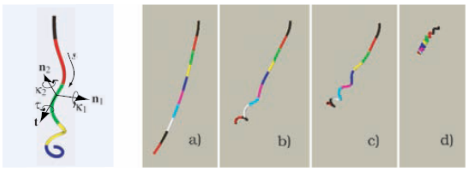


Figure 2.4: Super-helix [10].

2.1.5 Animator Driven

The above methods are best suited for realistic hair but cartoon hair is different. Cartoon hair tends to be clumpy and may actually move in physically impossible ways [11]. In order to replicate such motions, we must instead allow an animator to predefine the motions. Sugisaki *et al.* [11] proposes a technique for animating cartoon hair using

a database of motion impulses.

These impulses are created by a skilled animator to move the hair strands into whatever sequence of shapes they desire [11]. These impulses are then placed in the database along with an associated image of the hair. Hair animations can then be created by drawing a number of hair sketches and comparing them with the database images through angle and projection comparison (fig 2.5).

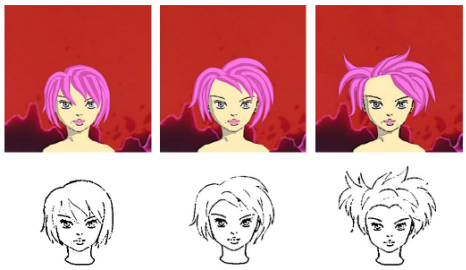


Figure 2.5: Matching hair motions using sketches [11].

Each database entry has an associated camera position and can store around 5 frames of impulse motions, so the strands are interpolated between the frames to ensure smooth movement [11]. This method produces good cartoon hair animations but, as it uses normal drawings, it is only suitable for 2D animations. A skilled animator is also required for generating the impulses but with little hope for division of labour as additional animators could introduce an unpleasant mixture of styles.

2.2 Hair Style

In addition to the dynamics of a single strand, we must also consider and simulate the properties of the entire hair style. Some hair simulations completely ignore the global properties and just simulate every strand in isolation, some use global methods on top of strand simulation and others ignore individual strands entirely and perform only a global simulation.

2.2.1 Fluid Based

Hair is composed of solid strands and exhibits rigid behaviour when stretched but when exposed to lateral shearing reacts more like a fluid [12]. This behaviour is supported by the theory of fluid dynamics as it covers not only liquids and gases but also solids (when they are considered under elastic theory). It is upon this fact that Hadap *et al.* [12] based their global hair method.

The hair strands are linked to fluid particles (fig 2.6) and hair-hair interactions are implemented simply through the influence of pressure [12]. The pressure density is affected by the number of hair strands in that section of hair fluid. When the strands move together, the fluid is compressed causing pressure to increase and the strands are forced apart.

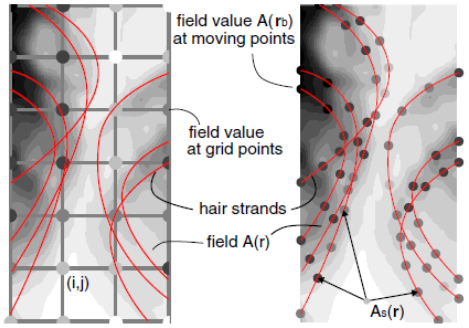


Figure 2.6: Hair particles immersed in a fluid simulation [12].

In addition to simulating the global properties of hair, this method has the advantage of a $O(kn)$ time complexity in comparison to a brute force $O(n^2)$ implementation (where n is the number of particles, and k is the number of surrounding particles) [12]. The reliance on an underlying strand simulation does however, impact the performance of the method.

2.2.2 Particle Based

An alternative to using fluid dynamics is the particle based method from Bando *et al.* [13]. This method while inspired by fluid dynamics drops the individual strand dynamics and instead treats the strands as loosely connected particles (fig 2.7). These

unordered particles are connected to their neighbours by temporary springs which can break if the particles are moved far enough apart.

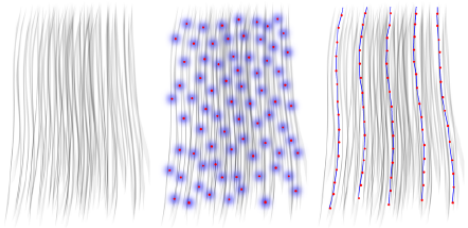


Figure 2.7: Generated hair strands from loosely connected particles [13].

In the fluid dynamics method, each hair particle was associated with a specific strand, but in this method particles can be viewed instead as sampling points in a hair volume [13]. Hair-hair interactions are modelled by implementing attraction and repulsion between particles to preserve density. This method is faster than Hadap *et al.* [12] but does pose a greater rendering challenge as there are no explicit strands.

2.2.3 Lattice Based

The fluid dynamics and particle based methods are better suited for an offline environment due to the number of strands and particles involved. A technique better suited to real-time applications is the lattice based method [14].

This method, similar to the loosely connected particles, forgoes the individual strand simulation and instead treats the hair as a cubic lattice grid fitted around the hair mesh (fig 2.8) [14]. This lattice does not require any information on the structure of the hair unlike other methods and therefore can be applied to an arbitrary mesh of any style.

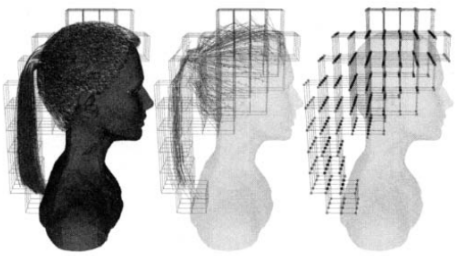


Figure 2.8: Lattice fitted around arbitrary hair mesh [14].

This technique ignores hair-hair collisions as the deformation of the lattice does not affect the relative positions of the neighbouring hairs except in the unlikely event of extreme motion [14]. Hair-body collisions are still considered though and are implemented using sixth-order polynomial metaballs which unambiguously stop collisions.

The main advantage of using a cubic lattice grid, in addition to being able to use arbitrary hair meshes, is that positions within the volume can be found and interpolated quickly, making this method well suited for real-time [14]. Even on relatively modest hardware (3GHz Pentium 4, NVIDIA Quadro4 980 XGL) interactive frame rates can be achieved (21.4ms per frame) with a 10x10x10 lattice, 200 attachments, 800 springs and 7 metaballs. The big disadvantage of the technique is that the motion is not particularly realistic as the strands do not move relative to their neighbours.

2.2.4 Triangle-Strip Based

Lattice based methods, despite their real-time performance may not be ideal if more realistic hair motion is required. Fluid and particle based techniques may also be too expensive for applications such as games where multiple characters will need simulated hair. A sparse hair model supported by a triangle-strip based system might be a more appropriate solution [5].

This method simulates the individual strands with a small number of guide strands with the additional strands being interpolated [5]. The hair-hair interactions are implemented by approximating a dense hair volume by constructing triangle strips between pairs of guide strands whose root tips are within a threshold of each other (fig 2.9). These strips can break if the strands become separated or if the strips become too elongated.

The unbroken triangle strips that remain are placed in an octree data structure and are used for detecting hair segment and hair vertex-face collisions [5]. Hair vertex-face collisions are detected when the vertex penetrates the face, and hair segment collisions are detected when the distance between them drops below a threshold value. Collisions are prevented through the application of a strong damped force.

This method is good for real-time systems, and with the addition of an orientation check during collisions (applying a weaker or stronger force depending on the angle), we can even replicate the anisotropic properties of hair [5].

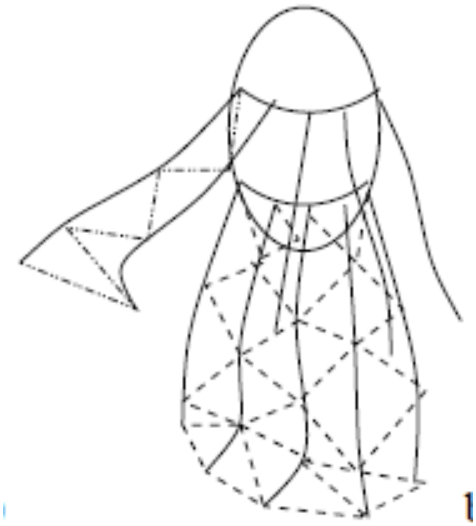


Figure 2.9: Triangle strip based hair [5].

2.2.5 Hair Strips

Similar to triangle strips, hair strips can be used to simulate hair by treating clumps of strands as a single strip [15, 16]. Each strip is composed of a parametric surface animated by a number of control points. Hair-hair collisions are ignored and instead strips are kept apart by binding them to their neighbours by a number of springs (fig 2.10).

This technique is good for real-time performance as the strip resolution can be easily scaled to balance quality and performance [15, 16]. By replacing hundreds of strands with a single strip, we can easily give the illusion of a full head of hair with as little as 100 strips.

The strips can also be easily animated using predefined key frames, a desirable feature for cartoon hair [15, 16]. However, the downsides of hair strips are that only straight hair can be simulated, by using 2D surfaces the hair can appear flat and without volume, and the strips cannot separate into smaller strips.

2.2.6 Wisps

Another method which exploits the grouping of hair strands, like hair strips, is wisps [3]. There are varying implementations of wisps, but in general they are (in contrast

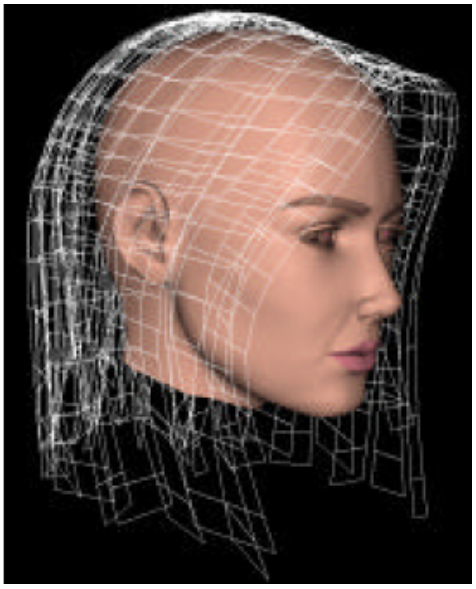


Figure 2.10: Hair strips [16].

to 2D hair strips) volumetric structures deformed by a skeleton curve with a number of instanced hair strands distributed within them.

The dynamics of the wisps can be simulated using traditional hair simulation methods such as mass spring systems [3]. Length preservation is not as big a concern as the wisp's curve is an average and will change length slightly anyway. Hair-hair collisions are minimised by only considering collisions between wisps of different orientation and ignoring strands within wisps.

Standard wisps share the limitation with hair strips that the hair clumps are not able to split and merge as they naturally would [17]. However, wisps can be extended with a tree structure to form adaptive wisp trees. The clumping and splitting of hair is implemented by breaking single branches into multiple branches when motion goes above some threshold and merging them when their motions become too similar.

Adaptive wisp trees are effective for fast motion, but due to their clustered appearance, they might be unsuitable for smooth hair [17]. Another possible disadvantage of this method is that it was not originally meant for real-time use and while modern hardware may allow it, it might not be viable for games etc.

2.3 Strand Rendering

Simulating the individual and global behaviour of hair is only part of the problem of hair animation. Once the motion has been calculated it must then be visualised and as with the various simulation methods, there is no definitive way to render hair. Certain techniques are better for realistic or abstract hair and depending on the underlying simulation model, we may be restricted to a particular method anyway.

2.3.1 Explicit Strands

Perhaps the most intuitive method is to render the strands individually as that is how we view hair normally. However, individual strands can be infinitesimally thin (less that the width of a pixel) and can create serious visual artefacts as a result [3]. Most lighting models are also dependent on surface normals for determining light intensity, which are undefined for 3D line segments [7, 18].

The issues with thin hair can be resolved by increasing the number of samples per pixel although at the cost of computation time [3]. The lack of surface normals is a more difficult problem to solve. Petrovic *et al.* [18] takes advantage of the solid appearance of hair to generate normals by producing a volumetric isosurface from the hair and using its surface to determine the normals. The normals are found by projecting the strand vertices onto the surface or by checking the signed distance field gradient used to generate the isosurface (faster).

Overall, explicit strand rendering can be expensive but is effective for realistic hair (fig 2.11). It is not ideal for stylised hair however, as features such as hard shading, hatching, strokes or silhouettes are difficult or impossible to apply to a thin strand.

2.3.2 Billboard Particles

A method perhaps better suited to abstract hair is the use of billboard particles [7]. These are particles, typically in the form of a quad which always face the camera. This technique does not require strands to be simulated individually and can be used with loosely connected particle systems and more traditional methods.

Billboard particles are usually used in conjunction with a sparse rather than dense hair model as the appearance of volume and clumping can be given by scaling the

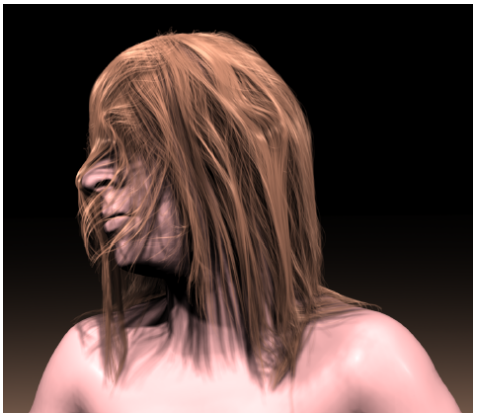


Figure 2.11: Individual strand rendering [2].

particles [7]. Shin *et al.* [7] uses a shifted sine wave to give a tapered shape to the hair clumps, but any function can be used (fig 2.12).

Figure 2.12: Billboard particle hair rendering [7].

This method is ideal for cartoon rendering as the hair produced has a clumpy, flat shaded appearance, and effects such as stylised specular highlights and silhouette edges can be easily added [6].

2.3.3 Volumetric Methods

Volumetric methods can be used to help generate surface normals in explicit strand rendering; however, they can also be used to generate a surface for rendering as well.

While probably not a good idea for realistic rendering, volumetric methods can be great for cartoon hair where a solid clumpy appearance might be desirable. The most popular method for producing a volume is the marching cubes algorithm [19].

Marching cubes takes as an input: a 3D scalar field, generated from the hair mesh and converts it into a 3D surface [19]. This surface is created by converting the scalar field into a cubic grid and marching through it cube by cube. Geometry is identified at the grid locations by comparing field values at each cube vertex to a threshold value. Vertex values at or above this threshold are marked and surface vertices are identified as being in between marked and unmarked vertices.

There are 256 possible vertex configurations, but as there are only about 15 unique variants (depending on the implementation), a significant speed up can be achieved using a lookup table [19]. Marching cubes can be reasonably fast as it makes extensive use of lookup tables and can be run in parallel due to the independent nature of each cube. However, small details can be lost if the resolution is too low, and many redundant triangles can be present.

There is an alternative method known as marching tetrahedrons based on a similar principle, but this is actually less parallelizable and less accurate [20]. A better alternative would be ellipsoidal sweeping [21]. This method works by first interpolating a number of ellipsoids of varying scale across a curve (hair strand in this case) and then sweeping an ellipsoid along this curve to generate a continuous base surface. This technique is far simpler than marching cubes and, when paired with a displacement map, the base surface can preserve fine details that marching cubes would otherwise lose.

2.3.4 NURBS Surfaces

For methods such as hair strips, most of the methods discussed for strand rendering would not be applicable. We would instead have to render the strips as NURBS surfaces whose control points are bound to a key strand to deform it in line with the position of the hair [16].

The entire hair style could in fact be treated as a single NURBS surface, giving the hair a solid appearance that would be ideal for cartoon hair (fig 2.13) [6]. A downside of using NURBS surfaces is that the hair may look too solid with no gaps

where appropriate, so alpha texturing may be necessary.

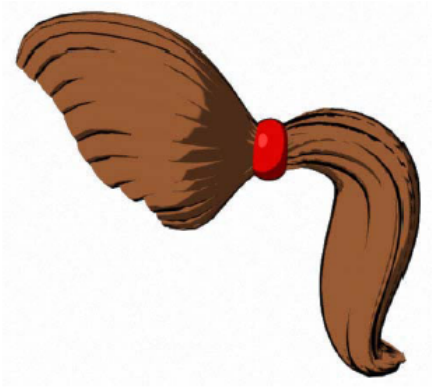


Figure 2.13: NURBS surface hair [6].

2.4 Cartoon Stylisation

Once the simulation and rendering methods have been decided, the final step for animated hair is figuring out how to light it and in the case of cartoon hair, how to add stylistic effects such as silhouettes, hatching and strokes.

2.4.1 Cartoon Shading

Traditionally, cartoons are not typically shaded with a smooth colour gradient but rather use flat shading with sometimes only one colour to represent areas in light and another representing areas in shadow [22]. In computer graphics, this effect can be replicated by calculating the cosine of the angle between the light direction and the relevant surface normal and using the result to index into a 1D texture containing a quantised colour gradient (fig 2.14).

By increasing the size of the 1D texture, additional effects can be implemented such as smoother transitions between colours, stylistic shadows or double contour lines [23]. If another dimension was added, making it a 2D texture, effects using additional metrics could be added [24]. For example, if the second axis was indexed by the distance of the object from the camera, we could have different colours at different depths.

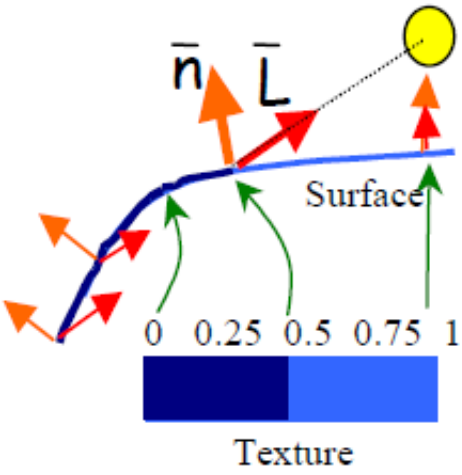


Figure 2.14: Using a 1D texture for lighting equations [22].

This kind of lighting is ideal for replicating a cartoonish style; however, there can be harsh transitions at the boundary of two colours thanks to the heavily quantised palette [22]. It is advisable to either use texture filtering or a smooth gradient between flat colours to minimise this problem.

2.4.2 Specular

If the lighting of the hair was limited to just the quantised cartoon shading above, the hair would only have a diffuse term and would exhibit a matte or dull appearance. Hair, on the contrary, is usually shiny with noticeable specular highlights where the light hits the surface at particular angles.

This specular effect is usually implemented through a specular term in the lighting equation based on one of many models. Examples include relatively simple models such the Phong reflection model, the Blinn-Phong shading model or more elaborate models such as the Cook-Torrance model [25, 26, 27].

The Phong and Blinn-Phong techniques use the dot product of two vectors to the power of some shininess value to determine the specular response [25, 26]. The response of both methods can be altered by using different values for the shininess parameter but this only really changes the size of the highlights, not their appearance or where they occur.

The only differences between the two models are the vectors used in the dot product calculation. The basic Phong model uses the reflection and view vectors while the Blinn-Phong uses the normal and half-angle vectors [25, 26]. The response is similar but the Blinn-Phong model is technically more correct through its use of the half angle vector as it is at the mid-point between the surface and its normal that specularity is at its highest.

The Cook-Torrance model goes further towards realism by including a number of additional parameters to allow greater control over the surface type being simulated [27]. These parameters are: the slope distribution, the Fresnel term and the geometric attenuation factor. The slope distribution helps control the apparent smoothness or roughness of the surface by defining the orientation of the microfacets that compose the surface. The Fresnel term controls how the light is reflected by each microfacet and finally, the attenuation factor determines the shadowing and overlap of the microfacets.

The Cook-Torrance model is much more realistic than the Phong or Blinn-Phong models but it is also much more complex, and its realism is not really desirable for an abstract rendering. Cartoon hair does not usually conform to realistic specular behaviour and it is therefore more important to have artistic control over the response rather than accuracy.

A better specular method for cartoon hair would be the 'X-Toon' method from Barla *et al.* [24]. This technique is derived from the Phong model but instead of using the dot product result to directly alter the colour value, it is instead used to index the y axis of a 2D texture similar to figure 2.15. The x axis is indexed by the same light intensity value used to index the quantised cartoon texture. This method allows the specular response to be completely customised. For example, the sharpness of the highlights can be adjusted by smoothing the transition, their size can be changed by altering the texture or the shininess value and the location can be moved arbitrarily.

An alternative to the 2D texture method is the specular sampling technique from Shin *et al.* [7]. This method, unlike the 2D texture model, does not use any Phong model derivatives for the specular term but rather uses its own specular term given by:

$$specular = K_s * lightColour * (max(L \cdot W, 0))^{shininess}$$
$$W = N * \omega + T * (1 - \omega)$$

where K_s is the specular strength, L is the light direction, N is the normal vector,

Figure 2.15: Example of using 2D textures for specular highlights [28].

T is the tangent vector, W is a weighted factor and ω is a weight value. The weight value can be used to move the specular highlights up or down the hair with a low value moving it down and a high value moving it up (between 0 and 1).

Unlike the other methods discussed, the specular response is not calculated at the same time as the diffuse response on the graphics card but rather is determined beforehand on the processor [7]. This method works by using the hair particles as specular sampling points. The particles that meet some predefined thresholds are marked and then formed into groups. These groups are then used to produce quadrilateral primitives upon which a custom highlight stroke texture is applied.

This method allows for more direct artistic stylisations than the 2D texture method, but it is designed for use with flat hair rendered using billboard particles and would probably not work well with 3D hair strands where depth testing would become a concern. This technique also loses the speed advantage that comes from performing the specular calculations on the graphics card. The method could potentially be implemented on a graphics card with geometry shader cores but not all computers have these included yet.

2.4.3 Strokes

When drawing normally, your pen, pencil or brush will usually leave a distinctive stroke mark upon the paper. Many methods exist for emulating this effect and by applying them to our cartoon hair model we can potentially enhance its believability.

The simplest way to render strokes is to just use 2D textures of various premade strokes which vary depending on how light or dark the surface will be [29]. These textures can be applied in two ways. The first and easiest way is to project the textures onto the screen based on the lighting values of the underlying model. This method produces a flat and coherent stroke texture as it is not tied to the model's geometry but is not suited for moving objects as the strokes remain stationary, causing what is known as the 'shower door' effect [22].

The second method is to project the textures onto the model [30]. This method works by projecting the stroke textures onto each of the model's triangles and is better suited to animated models as it is temporally coherent. This technique does however, have problems with spatial coherence between triangles as the textures will appear faceted without some form of blending.

Simple blending schemes can give smooth results but will still contain discontinuities and temporal artefacts [30]. Praun *et al.* [30] gives a number of methods that can resolve such issues. For example, single-pass 6-way blending which calculates blend ratios for 6 textures and interpolates them across the triangle or triple-pass 2-way blending which can use a higher number of textures by operating over a number of passes. These methods are a little dated however, as modern graphics hardware can utilise more textures simultaneously but the principle remains relevant.

There are a number of ways to generate the 2D stroke textures, normally they can be created by hand but this is rather tedious, or they can be generated randomly but this can result in clustering as the distribution is not likely to be uniform [30]. Stroke coherence between the different tones of texture can also be an issue.

These distribution and coherence problems can be fixed using the tonal art map method [30]. This method solves the stroke coherence problem by ensuring all strokes present in lighter tone maps are also present in darker tone maps. The distribution problem is resolved by looking at a number of random candidate strokes before adding one and choosing the one that has the greatest effect on the average tone. This method also maintains tone coherence at different zoom levels by creating custom mipmap images where the stroke number is decreased but the screen pixel width is kept constant.

Instead of using 2D textures, a more elaborate system using realistic brush modelling can be used [31]. In this system the scene is broken up into a number of regions which are then customised with perceptual dimension parameters such as density,

coarseness or hue contrast. These regions then have customised brush strokes applied to them to replicate almost any artistic style imaginable, but at a rendering time of around 10-20 minutes per frame, this is not suitable for real-time.

A particle based system using 'geograftals' might be better for interactive applications [1]. Geograftals are customisable triangle strip structures with line strips along their edges for silhouettes. These geograftals are placed along all the surfaces of the model and are scaled based on their orientation with respect to the viewer, with those oriented perpendicular being largest (fig 2.16).

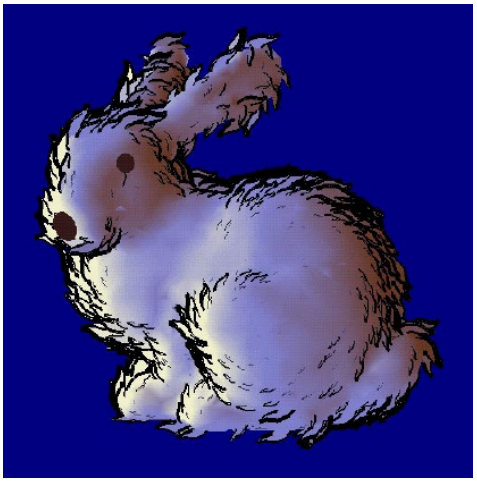


Figure 2.16: Stanford bunny using geograftals [1].

By altering the shape, colour and silhouette of these geograftals, many art styles such as pointillism, oils and pen can be replicated at interactive frame rates [1]. The number of geograftals can be rather sparse however, resulting in visible holes if colour is used, so a base colour for the model is desirable.

2.4.4 Outlines

In cartoons, object silhouettes are typically outlined to make them stand out, and creases are normally highlighted to represent surface details that may otherwise be hidden with cartoon shading. These features can be replicated for simulated cartoon hair using image-space or objects-space methods.

Image-space methods are the simplest to apply and can be implemented with a Canny or Sobel edge detection filter applied to the colour buffer (fig 2.17) [32]. Canny

is normally considered a superior edge detection algorithm thanks to its noise tolerance but Sobel is more commonly used due to its simplicity.

Figure 2.17: Example of image-space edge detection [32].

The edge detection can be applied to the colour buffer directly, but in order to ensure overlapping objects do not merge it is a good idea to use an ID buffer [32]. An ID buffer is just a image buffer where each object is assigned a unique identifying colour. An ID buffer helps the edge detector distinguish between objects but does prevent internal edges such as creases from being detected. A second edge detection pass may be required if internal edges are desired.

Object-space methods on the other hand are more difficult to implement but allow for potentially nicer and more controllable silhouettes (fig 2.18). These methods usually identify silhouettes by looking for the edges between back and front facing triangles in the object [22].

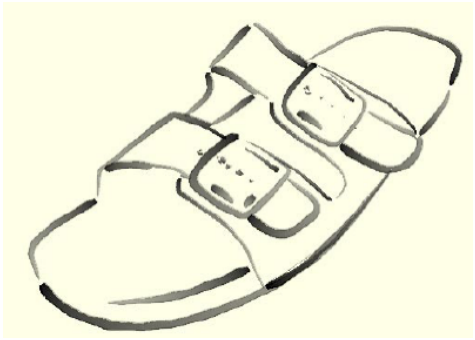


Figure 2.18: Example of object-space silhouettes [34].

There are many ways to find object-space edges, Lake *et al.* [22] stores all of an

object's edges in a hash table using the sum of the edge vertex indices as the hash key. Edges with only one instance in the hash table are marked as borders and will be outlined, edges between front and back facing triangles (determined by surface normals) are silhouettes, and edges whose triangles are above a threshold angle apart are creases. This method does however, have temporal coherence problems as it does not take edge ordering into consideration.

Markosian *et al.* [33] proposes an alternative method utilising randomised silhouette detection. This method works by checking a small number of edges randomly until a silhouette is identified and then following all adjoining silhouette edges until the first edge is reached again. In order to speed up the process, the previous frame's silhouette position is also used to help inform where to first start looking. This method produces an ordered edge chain but is restricted to closed meshes.

Once the silhouettes, creases and borders are identified, they must be highlighted. Perhaps the easiest way to do so would be to draw a thick line over the edge but this limits the style of the edge. A better method is to use the silhouette edges to generate a quadrilateral strip oriented towards the viewer which allows us to apply stroke textures, increase stroke width and so on [22]. In order to ensure the quads are always facing the viewer and are of the same width, it is also necessary to define them in screen-space coordinates [34].

Quadrilateral edges do however, introduce a new host of problems, in particular strokes penetrating the polygon's surface, strokes overlapping and sharp angles [22]. The penetration and overlapping issues are due to the depth buffer getting the ordering wrong as the strokes do not have reliable depth values as they are defined in screen-space [34]. The solution is to drop the depth buffer all together and instead use an ID buffer to determine visibility [35]. The sharp angles can be fixed by increasing the quad width at corners, relative to the angle [34].

Edge detection can have a problem with temporal coherence as the edges change length or position over time [35]. Temporal coherence can be increased by either linking the position of the strokes to the underlying geometry of the model (not really possible for image-space methods) or by using the previous frame's silhouettes to interpolate the current frame's silhouette parameters. This can lead to discontinuities forming though, as strokes break apart, so a regular healing step should be applied to rejoin these edges.

Chapter 3

Design

The overall goal of this project is to produce a real-time cartoon hair simulation and rendering pipeline. In order to do so, the system must first be designed.

The design process consists of a number of major decisions. These are:

- What simulation model and hair density will we use?
- How will we define the initial strands?
- What form will the rendered strands take?
- What artistic stylisations will be applied?
- What languages, tools and libraries will be used to ease development?
- How will the project be controlled?

In this chapter, we will look at the choices that were made for each of these decisions.

3.1 Simulation Model

The choice of hair simulation model is relatively simple given the circumstances. Most of the hair simulation methods discussed in the state of the art strive for realism but as we are implementing an abstract hair simulation we are not concerned with such behaviour. It is instead, more desirable to be able to shape the hair into wild and unrealistic styles and to blend the dynamic simulation with predefined animations

along the lines of Noble *et al.* [6] and Sugisaki *et al.* [11]. The only real choice given the amount of constraints required is to use a mass spring model.

In order to reduce the difficulty of implementation and to increase the stability of the hair simulation, the Bullet Physics engine is used instead of custom code to perform physics calculations [36]. We use Bullet Physics to create the hair strand spring chains and also to keep track of hair-body collisions. Hair-hair collisions are not implemented as Bullet does not support collisions between thin strands, and other methods that were investigated were found to be too costly to implement in real-time.

The spring chains are structured according to Selle *et al.* [2] which uses additional structured springs to resolve length, torsion and bending issues that would otherwise be present. As pointed out in the state of the art, hair strands based upon spring chains have a tendency to stretch or compress due to the elastic nature of springs. No measures to prevent this effect are implemented as it should not be noticeable with an already unrealistic hair style, and Bullet may have some in-built techniques to combat it already.

In addition to realistic motion, artists may, for effect, make cartoon hair move in an exaggerated and perhaps impossible way. This effect requires artistic input and may violate the laws of physics so it obviously cannot be implemented using the physics engine. The project application therefore uses a system of blending springs attaching all hair particles to anchor points in order to add predefined animation, similar to Noble *et al.* [6]. These anchors can be animated and by altering the blending spring stiffness it is possible to adjust the hairs adherence to either the dynamic or predefined animations. These anchor points when not animated also have the added benefit of maintaining the hair style shape.

A sparse hair model of hundreds as opposed to hundreds of thousands of strands similar to Shin *et al.* [7] is used. This low hair density is used for a number of reasons. A dense hair model is better suited to realistic hair but not cartoon hair as cartoon hair is normally clumpy. In this project, each hair strand is a guide strand for an entire hair clump. Dense hair models are also much harder to implement in real-time so a sparse hair model is better suited for this application.

3.2 Hair Styling

Ideally the project application would be completely automated and would require no artistic input; however, even relatively automatic hair generation schemes such as the method used by Yukesel *et al.* [37] still require the input of at least a base hair mesh from an experienced artist. In general, you have a choice between defining the simulated strands and having the rendered surface being automatically generated as in Shin *et al.* [7] or the other way around as in Noble *et al.* [6]. Creating the rendered surface first does give greater artistic control over the final result but it also makes the project dependent on a skilled modeller and animator.

Considering the above, this project opts for a more intuitive option and uses the 3D modelling program Blender [38] to create the initial shape for the simulated hair strands, allowing the rendered strands to be programmatically generated later. Blender has an in-built hair particle system which allows the number, length and distribution of strands to be defined and the result can then be combed or cut naturally using Blender's styling tools. The hair particles can then be sent to the project application using a custom XML exporter.

Blender also uses Bullet Physics as its underlying physics engine and while no physics simulation for this project takes place in Blender, it does mean that the structure of the hair strands is close to what the project application expects. This simplifies the exporting process as once we extract the hair particle positions we can almost directly input them into the project application to create the simulation. Blender does not however, have a hair exporter so a custom one has to be written.

3.3 Hair Rendering

The most straightforward way to render the simulated strands is as explicit line segments but this would not work well for this project. This is because we want to add additional stylisations to the hair which require a larger surface area than a line segment can provide. An additional issue prohibiting the use of explicit strands is the sparse hair model as without additional strand interpolation the hair would appear very thin. Line segments also lack an easily definable surface normal (vital for most lighting equations) which makes shading more difficult.

The project instead uses a sort of hybrid method between the NURBS and billboard particle methods inspired by the ellipsoidal sweeping technique in Hyun *et al.* [21] to create automatically, a 3D base mesh around each strand. These base meshes are scaled according to a curve similar to Shin *et al.* [7] to give each strand the tapered appearance of cartoon hair. By treating each strand as a guide for a clump of hair, we can also drastically reduce the amount of strands required to numbers that can easily achieve real-time performance.

The base meshes (unlike 3D line segments) are composed of triangles and therefore have surface normals (essential for lighting) defined. These normals are however, only defined for the triangle faces and can result in the strands taking on a faceted appearance if lighting was to abruptly change at the triangle boundaries. Therefore, we use vertex normals instead by averaging the surrounding surface normals to avoid this issue.

3.4 Artistic Stylisations

In the state of the art, it was highlighted that most hair rendering research has been concerned with realistic and not cartoon hair. Cartoon hair has a significantly smaller body of research associated with it and the stylisations applied in the few cartoon hair papers that could be found were relatively simplistic.

This paper therefore, in addition to the basic effects found in the cartoon hair papers, also attempts to implement more advanced effects from general NPR non-photo realistic papers that do not appear to have been applied to hair before. This project is, in a way, a sort of evaluation of their viability as cartoon hair effects.

The basic effects that are implemented are quantised cartoon shading texture gradients, Blinn specular highlights and image-space silhouette edge detection. The more advanced techniques are object-space silhouette detection, surface hatching and extended cartoon shading textures which use an additional axis to implement interesting effects such as custom specular responses.

The basic shading is a hard shading technique based on Lake *et al.* [22] which normally uses a 1D texture for a quantised colour gradient (appropriate for cartoons) that is indexed by light intensity. This project actually extends the 1D texture method to instead use a 2D texture through a technique inspired by Barla *et al.* [24].

This 2D extension also referred to as 'X-toon' shading, similar to Lake *et al.* [22] indexes the x axis of the texture using the dot product of the light direction and surface normal [24]. The y axis however, can be indexed by any number of custom metrics such as surface orientation or distance from the camera, allowing effects such as depth of field, specular highlights and back lighting to be applied.

In addition to various texture based specular methods, this project also attempts to implement the specular highlight method from Shin *et al.* [7] based on grouping specular points. This method samples specular values across the hair strands and joins together points that meet certain criteria to form textured quadrilateral strips which can be used for custom highlights. As will be seen in the implementation chapter, this method is not ideal for 3D strands or moving hair and has mainly been included for comparison with the other specular techniques being implemented.

Entities in cartoons commonly have an outline distinguishing them from the rest of the scene. This project implements two different silhouette outlining methods to emulate this effect. The first is an image-space based system which uses edge detection and a dilation step based on Redmond *et al.* [32] and similar to a method used in Shin *et al.* [7].

The second method is an object-space method from Lake *et al.* [22] which marks edges as silhouettes when they neighbour a front and back facing triangle. These edges are then sorted and using methods from Kalnin's *et al.* [35] and Northrup *et al.* [34] are converted into a number of quadrilateral strips which can be textured with custom strokes. It should be noted that object-space silhouette methods do not appear to have been used with cartoon hair before.

This project also implements two hatching techniques to produce strokes on the surface of the strands. The first is a viewport projection method from Lake *et al.* [22] which projects stroke textures of varying tone onto the screen based on the lighting intensity of the associated geometry. This method is included mainly for comparison as it looks unusual when applied to dynamic objects as the strokes remain static, relative to the screen and not the strands.

The second method takes advantage of the hair mesh triangles to apply blended stroke textures based on Praun *et al.* [30]. This method is temporally coherent as it is based on the underlying strand geometry, and by using custom mipmap levels, the stroke widths can be kept the same across various depths.

3.5 Languages, Tools and Libraries

Three computer languages are used in the development of the project. C++ is used for programming the main application as the libraries we use are also written in C++. Python is used for writing user tests for the evaluation portion of the project and also for Blender addons, and GLSL is used for programming the shader units for advanced rendering effects.

In terms of tools, the 3D modelling and animation program Blender is used for designing the initial simulated hair strands. Blender is used mainly for its in-built hair particle system. The image editor GIMP is used for designing stroke textures as it has a selection of realistic brush effects included [39]. GIMP is also used for compiling the final hatching textures into a format Ogre can read. Finally, GitHub is used for source control and issue tracking [40].

In terms of libraries, Ogre is used for rendering as it has a number of very useful convenience features such as material and compositor scripting, a detailed error logging system and extensive documentation [41]. The user interface for the project application is implemented using the library CEGUI which has its own external layout editor [42]. OpenCV is also used in conjunction with Python for video playback during user testing, and as already mentioned Bullet Physics is used for the physical simulation [43]. TinyXML-2, a lightweight XML parser is used for reading the hair strand and collision shape XML files produced in Blender [44].

3.6 Controls

The hair and character animations used by the project application are set at build time. Various default parameters for the hair can be also set as pre-processor directives as well but these can be modified at run-time also. The resolution and a few rendering options can also be altered at start-up using the default Ogre settings prompt before fully loading into the application.

The camera remains always focused on the hair, but you can rotate around the head using the directional, page up and page down keys. By pressing M, the run-time hair parameters user interface can be toggled on or off. The design for this user interface can be seen in figure 3.1.

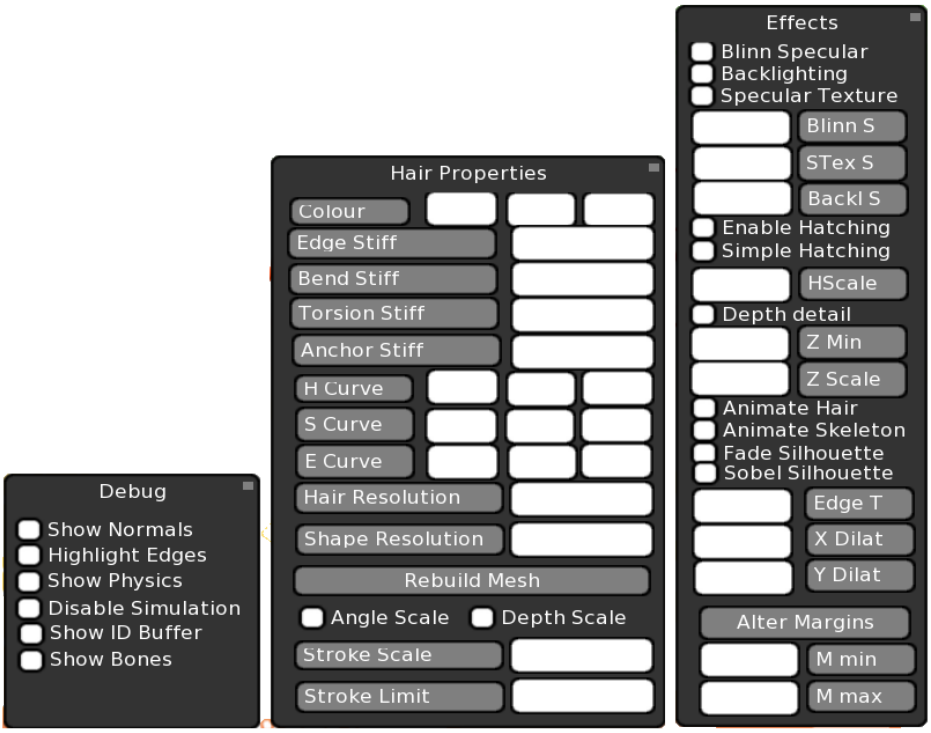


Figure 3.1: The final GUI.

The user interface consists of three menu windows: a *debug* menu for investigating bugs, a *hair properties* menu for adjusting aspects of the hair simulation and the rendering base mesh, and an *effects* menu for controlling some of the more advanced shading features.

Most of the options visible in figure 3.1 are self-explanatory; however, some might be a little obscure so we will explain them.

The H, S and E curve options control the quadratic equations used by the base mesh scaling, anchor stiffness scaling and finally, silhouette fading techniques respectively. They each have three parameters which represent the a, b and c values in the quadratic equation $value = a * x^2 + b * x + c$ where x represents the point along the strand or silhouette (between 0 and 1).

The *hair resolution* option alters the number of hair spline curve sampling points used when generating the base strand mesh, and the *shape resolution* adjusts the number of vertices created per sampling point. The mesh must be rebuilt, after adjusting these values, by pressing the *rebuild mesh* button for any changes to take effect.

The *angle scale*, *depth scale* and *stroke scale* options can be more or less ignored as they are overpowered by the *stroke limit* option which controls how thick object-space silhouettes can get.

The *simple hatching* option switches the hatching technique from strand texturing to viewport texturing, and the size of the hatching strokes can be adjusted by varying *HScale* or hatching scale value. The depth detail effect where the cartoon texture varies at different depth levels is controlled by the *Z Min* and *Z Scale* parameters. *Z min* determines the depth value that represents the lower end of the specular texture y axis and this multiplied by the *Z Scale* parameter represents the upper end.

The point at which the hair strands collide with the head can be adjusted with the *alter margins* button and the *M min* and *M max* values. If *M min* and *M max* differ, strands are randomly assigned collision margins in between the two values.

Finally, the thickness of the silhouettes when using image-space edge detection can be adjusted by changing the *X* and *Y Dilation* options. The *Edge Threshold* value determines the threshold between colours necessary for an edge to be identified. If the edge detection doesn't appear to be working you can experiment with the *Edge Threshold* parameter.

Chapter 4

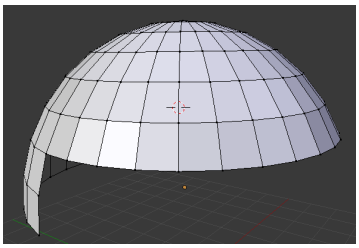
Implementation

In this chapter, we will discuss the specific details of the implementation of the project application based upon the design laid out in the previous chapter. The implementation will be discussed in the same order as the stages of the production pipeline.

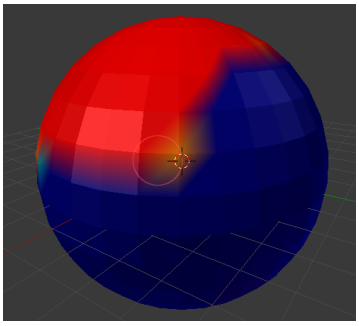
The hair design phase is discussed first, this is where the hair style is created in Blender along with the head collision shape and ultimately exported in XML for use in the project application. The simulation phase is next, this is where the hair style XML file is used to create the simulated strand structure and animated anchor particles. Finally the rendering phase is discussed, where the simulated strands are converted into a 3D base mesh and cartoon stylisations are applied.
The entire implementation can be found on the attached cd.

4.1 Hair Design

Before any simulation or rendering can be performed, the hair structure to be simulated must first be created. This step is performed by an artist using the in-built Blender hair particle system. In order to create the hair we wish to style, we must first create a hair particle emitter in Blender; however, it needs to be attached to a surface in the same way our hair is attached to our scalp. An entire object can be designated as a scalp, like the object in figure 4.1a or just part of an object can be used by painting the areas we would like to use with Blender's weight painting system (fig 4.1b). The hair in this project uses the former method.



(a) Basic scalp model.



(b) Weight painted scalp.

Figure 4.1: The two ways to create a scalp for the hair emitter.

Once the hair emitter is applied to the scalp, strands grow perpendicularly from the surface as in figure 4.2. The global parameters such as the number of strands, the particle resolution, the distribution and the length must be set at this point before any styling can been performed. Due to the design of Blender's hair particle system, if we begin styling before setting the global parameters, they will be locked and become read-only. The styling will need to be completely discarded if we need to change any parameters afterwards.

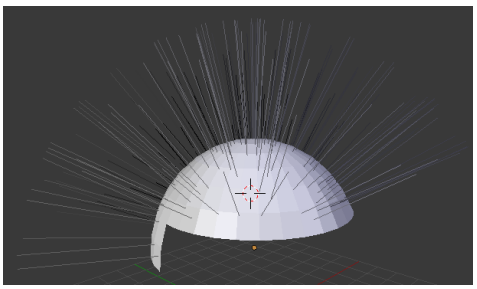


Figure 4.2: Unstyled hair.

When the parameters have been set, the hair can then be styled using Blender's particle tools for combing, cutting, adding and subtracting strands (fig 4.3). If the artist is happy with the resulting hair shape, they can then export the hair to XML for parsing in the project application. Anchor particle animations whose implementation will be detailed later can be defined at this point. These animations are created by reusing the same hair emitter with the same global parameters to produce multiple hair strand XML files with each file acting as an animation key frame. These files are

all passed to the project application where the first one is used as the base hair style and all subsequent files are used to interpolate the anchor movements.

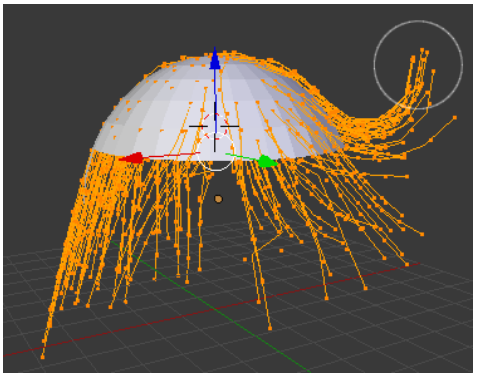


Figure 4.3: Styling the hair.

In addition to defining the hair structure, the collision shape for the head is also specified at this point. This is the structure that is used to represent the head in the physics engine allowing the strands to interact with it rather than pass right through. This can be done programmatically through various helper functions in Bullet Physics, but it is generally more accurate and efficient to do it by hand. Once again Blender is used to create the shape by creating and grouping sphere primitives which are then saved to an XML file for use in the project application (appendix fig 1).

The hair particle system in Blender is not designed for exporting to an external physics engine and as a result no hair exporter of any kind is bundled with Blender. There is also no exporter for the head collision shapes defined using sphere primitives as this project is essentially using modelling tools for unanticipated applications. These exporters must therefore be custom made.

Exporter scripts can be written in Blender using the fully featured Python scripting language. Writing the collision shape exporter is relative trivial, the Blender API (Application Programming Interface) gives access to the objects currently selected so if the artist selects all the spheres they wish the collision shape to be composed of, we can access them easily through Python. The radii and positions of the spheres can then be written out to an XML file using Python's standard XML library.

The exporter for the hair strands is not as simple. As previously stated, the hair particle system does not appear to be intended for use outside of Blender and it would seem that the Blender API for accessing the particles is very minimal. We are able to

select the particle system as a whole, but we are unable to access individual particle positions.

A work-around using the few Python accessible particle system functions is used to export the hair. First, the hair is duplicated so that any modifications do not affect any future attempts at styling. A Blender function is then used which allows you to convert a hair particle system into a static mesh composed of line segments. The API for accessing standard meshes is far more developed, thankfully. The hair mesh is however, still useless to us in this form as the strands are indistinguishable from each other. Luckily, Blender has another in-built function which separates meshes according to their loose parts which gives us the strands as individual meshes.

The hair is finally exported by iterating through each of these strand meshes and outputting their particle positions to an XML file which also maintains the relationship of each particle to its parent strand. Hair is typically very thin, but Bullet Physics does not appear to handle millimetre level collisions very well. In order to overcome this problem, the hair is scaled up so that it may in fact be several metres across. We keep the simulation as accurate as possible by sending the scale factor with the hair strand XML file and then using it to scale the simulation accordingly. Once the hair has been exported, the meshes are deleted leaving the original particle system.

4.2 Hair Simulation

4.2.1 Importing Hair

The collision shape and hair strand XML files are read into the project application using the C++ library TinyXML-2.

The collision shape file is used to generate a rigid body using a Bullet *btMultiSphere* shape. This rigid body is then attached to the head bone of the animated character so that it will move along with any animations. In early versions of the application the head collision shape was generated automatically using the head vertices to generate a convex hull shape. This method was expensive to use due to the irregular shape and was difficult to implement with a full character body as convex hulls can not be animated (only a head was used in that version).

The hair strand file is used to recreate the same hair structure as defined in Blender.

The XML file contains objects, for each strand that was present in Blender, which contain all the particle positions. The strands are easily recreated by iterating through each strand object, passing the particle positions to Bullet to generate soft body nodes and then linking them to the previous node in the strand.

The strands at this point however, are not ready for use as they are not bound to the character's head yet and will fall away if simulated. It is therefore necessary to bind them to the head by making the root particle of each strand static so that it does not take part in the simulation and instead holds the other particles up. This is done by setting the mass of the particle to zero. In order to have the hair move with any character animation, the position of the head bone of the character skeleton is tracked and any transformations are directly applied to the root hair nodes.

4.2.2 Structured Strands

As stated in the state of the art, simple mass spring systems have no ability to constrain torsional or bending motion unlike real hair. In order to resolve this and to produce more style maintaining constraints, we create additional hair particles perpendicular to the hair segments. We then attach springs between these new particles and the existing ones to create a more structured strand.

The structured layout is based on the tetrahedral spring configuration introduced by Selle *et al.* [2] and visible in figure 2.1. There is one main difference in this implementation however, as the additional springs are not applied to a straight strand which is then later styled but rather the springs are attached to the already styled strand. This means that the default shape of the strand is not necessarily straight and it might in fact resist being straightened in some cases (fig 4.4).

Different stiffness values can be applied to the edge, torsional or bending springs. These stiffness values can be set between 0 and 1 (with 0 being completely loose and 1 being almost rigid). By adjusting these values we can have strands seemingly composed of elastic or almost rigid hair segments, hair that bends easily or hair that stubbornly remains styled and so on.

The structural springs are added to the already deformed hair to help maintain the hair style as designed. This can however, lead to unpleasant hair movement if the strand is in a particularly odd position so some experimentation with hair styles and

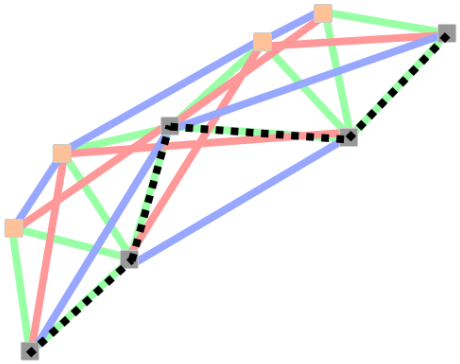


Figure 4.4: An example of the structural springs applied to an already deformed strand. The new structural particles are orange, the bending springs are blue, the edge springs are green and the torsion springs are red. The original strand is represented by the dotted line.

spring stiffness values may be required.

4.2.3 User Defined Hair Motion

As mentioned in the state of the art, some hair motion in many traditional cartoons can not be replicated using just a physics simulation. For dramatic effect, the artist may have exaggerated the motion or made it downright impossible to reproduce in real life. In order to support such a feature, we allow the user to pass several hair style files created in Blender to the project application and use them as motion keyframes.

This feature is implemented using additional anchor nodes bound to the ordinary hair particles by 'blending' springs. By altering the stiffness of the blending springs, we can control the degree of adherence to the predefined or dynamic animations. The stiffer the spring, the closer the correspondence to the predefined animation. The stiffness can also either be set uniformly along the length of the strand or varied using a quadratic curve. This is similar to Noble *et al.* [6].

The animation data is specified by passing to the project application an XML file containing a list of different hair shape XML files creating in Blender. These must derived from the same base hair or otherwise strand differences will produce visual artefacts or more likely cause the program to crash.

As each hair strand file is based on the same base hair, an instance of every hair

particle can be found in the same part of each file. By altering the hair style in each file, we can have different key frame positions for each particle and interpolate between them to move the anchors. We do not linearly interpolate between these positions however, as that would result in a mechanical looking animation. The particle positions are instead used to create a spline curve using Ogre's SimpleSpline class which we can then smoothly interpolate across.

4.2.4 Character Motion

For demonstration purposes, the simulated hair is also parented with the head of an animated character. Modelling and animating a custom character for this task would be outside the scope of the project so a sample ninja character bundled with Ogre is used.

In order to parent the hair to the head, we first find the position of the head bone in the character's skeleton. The bone's position is however, given in local skeleton coordinates relative to the root bone rather than world-space coordinates needed to position the hair. The world-space coordinates are obtained by finding the bone's position relative to the root bone and then the root bone's position relative to the scene's origin.

The hair is also initially located at the origin of the scene so it must be translated to the bone's position and then rotated to match its orientation. Once the hair is in place, only the anchor and root particles need to be directly transformed from then on, allowing the strands to otherwise move freely.

4.3 Rendering

4.3.1 Strand Smoothing

The number of segments for each simulated strand is determined by the hair resolution in the XML files imported from Blender. It may be desirable to have a low resolution for each strand to reduce the computational burden of simulating the hair. The lower the resolution however, the less smooth the hair appears, making the strands seem more mechanical than organic.

The 3D strand mesh is generated from the simulated hair strands so in order to reduce the effect the simulation resolution has on the rendered result, we do not use the simulated strands directly but rather use them to create smooth spline curves using the Ogre SimpleSpline class. By using a spline instead of the hair particles directly, we are less likely to see sharp angles and can have a rendered strand of a far higher resolution by simply increasing the number of sample points along the spine.

4.3.2 Base Mesh

The most intuitive way to render the hair is as individual strands but not only is this expensive for a large body of hair but also if we wish to apply artistic stylisations, we will not have much of a canvas to work with. Strand silhouettes, hatching textures and cartoon shading can not be applied to a sub-pixel thin surface.

In order to apply these effects, a base mesh must be created from the strand. We do this by creating a simple shape approximating a circle such as a hexagon. This shape is then interpolated across the strand spline curve, and at each sampling point, we use the shape oriented with the strand segment to generate a number of vertices.

The generated vertices are then linked with the vertices generated at the previous sampling point. The shape is also scaled according to a quadratic curve as it moves along the strand to give the mesh a tapered appearance (appendix fig 2). As the base mesh is generated by sweeping a flat shape across the strand, the ends of the strand are left open so additional vertices must be generated for the top and bottom and joined to the surrounding vertices to close the mesh.

The triangular structure for the strand is however, not generated at this point but rather individual vertices are created one by one in a spiral as in figure 4.5. Only one instance of each vertex is generated as well, even if shared by multiple triangles, and if this structure was to be passed directly to Ogre to be rendered, it would take the form of a mass of disjoint triangles.

In order to create the triangular structure, a list of indices for the vertices we created in the order necessary to create the desired triangles is generated (fig 4.6). When rendering the strand, we iterate through this index list and use the resulting indices to access the correct vertices in the vertex list. This process could be greatly simplified by just generating the triangles from the beginning and dropping the indices

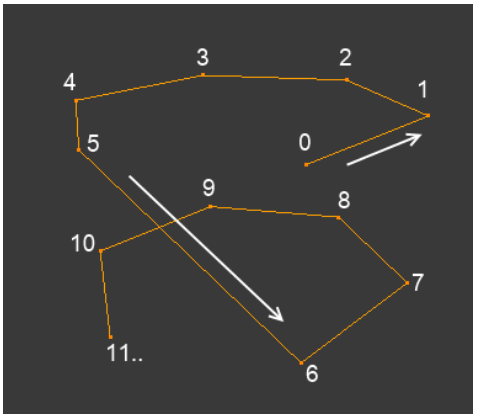


Figure 4.5: The vertices being generated, the numbers are their indices.

altogether; however, this would significantly increase the storage needed as an index only requires a single integer while a vertex needs three float values for its x, y and z coordinates, and we would need multiple identical copies of each vertex.

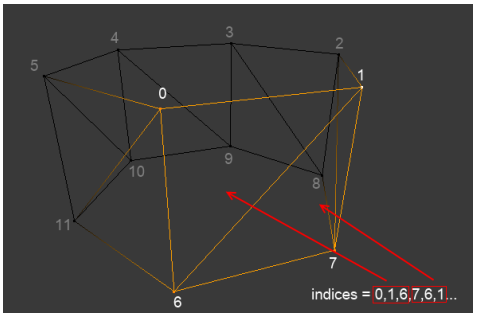


Figure 4.6: The triangle indices being generated.

Using the simpler approach would also complicate the generation of the normal vectors necessary for performing lighting equations on the mesh. The normals are manually generated per vertex by averaging the surface normals of the neighbouring triangles of each vertex. If indices were not used, this averaging would have to be performed essentially multiple times for each unique vertex with the same result every time.

4.3.3 Cartoon Shading

The process involved in adding a basic cartoon shading effect to the strand surface is relatively simple. We first determine the light intensity based on the angle at which the light is hitting the surface which is given by the equation [22]:

$$intensity = max(dot(\vec{N}, \vec{L}), 0.0)$$

where $\vec{N}$ is the surface normal and $\vec{L}$ is the light direction. A result of 0 (if the light is parallel) means no light is reaching the surface, and a result of 1 (if perpendicular) means the maximum amount of light is present.

Cartoons usually do not have a smooth lighting gradient but instead rather sharply transition between a limited set of tonal values [22]. We can emulate this effect by not applying the intensity directly to the lighting value but by rather using it to index into a texture. By using a quantised gradient in the texture we can reproduce the limited colour palette of a traditional cartoon, and we can also completely customise the lighting response by altering the texture. The indexed value is applied by multiplying the base hair colour by it.

If we are only concerned with lighting intensity, we can just use a 1D texture (fig 4.7) for the shading value, but if we use a 2D texture with the x axis representing intensity, we can implement a whole host of effects by assigning custom metrics to the y axis. In this project, we use 2D textures to implement effects such as depth dependent detail, back lighting and specular highlights.

Figure 4.7: 1D cartoon texture.

4.3.4 Varying Detail with Depth

Looking at a photo of a landscape, you may notice that the foreground is more vibrant while a mountain in the distance might appear more washed out. We can replicate this background fading effect in our cartoon scene to emphasise the nearest strands. We implement this effect by extending the cartoon texture to 2D as in the appendix (figure 3) and using the equation:

$$D = 1 - log(z/z_{min})/log(z_{max}/z_{min})$$

to index the y axis [24]. Background objects can be washed out by fading the 2D texture gradient towards the upper y axis. The distance response can be customised by changing the minimum and maximum z values which determine the lower and upper bounds of the y axis.

4.3.5 Back Lighting

The extended cartoon texture method can do a lot more than vary the colours at depth. By replacing the y axis with the equation:

$$y = \left|\vec{N} \cdot \vec{V}\right|^r, r \geq 0$$

where $\vec{N}$ is the normal vector, $\vec{V}$ is the view direction and r is a user definable value to alter the intensity of the effect, we can implement back lighting [28]. Back lighting (also known as contre-jour) is a photographic technique where the subject is illuminated from behind [45].

Figure 4.8: Example of the back lighting effect.

An example of the back lighting texture used in this project can be seen in the appendix (figure 5). The r parameter has no effect when the view direction is parallel or perpendicular with the normal vector as $\left|\vec{N} \cdot \vec{V}\right|$ will equal 1 and 0 respectively. The r parameter does however, affect the intermediate values and will cause the index to tend towards the upper y axis when $0 \geq r < 1$ and the lower y axis when $r > 1$. To

apply the effect, the back lighting is mixed with the cartoon shading by adding it to the diffuse colour value similar to the specular methods detailed below. This effect can be seen in figure 4.8.

4.3.6 Specular Highlights

Hair is not generally a dull matte material but rather a shiny one, and exhibits specular or reflective properties. In order to make the cartoon hair shine, we implement three different specular methods for comparison. The first method is a basic Blinn specular component which is simply added to the diffuse colour.

The Blinn specular value is given by equation:

$$specularity = (\vec{N} \cdot \vec{H})^{c_1}$$

where:

$$\vec{H} = \frac{\vec{L}+\vec{E}}{\|\vec{L}+\vec{E}\|}$$

and $\vec{N}$ is the normal vector, c_1 is the surface shininess, $\vec{L}$ is the light direction vector, and $\vec{E}$ is the eye or view direction vector. An example of this effect can be seen in figure 4.9.

Figure 4.9: Example of Blinn specular applied to the cartoon hair.

The second implementation is based on a method from Barla *et al.* [24]. This technique uses the extended 2D texture method mentioned earlier with the y axis being

indexed by the same specular equation from the Blinn model above. This method offers a far greater degree of artistic control as by altering the specular texture (appendix fig 4) we can specify the smoothness and width of the specular highlights in a simple and intuitive way. An example of this effect can be seen in figure 4.10.

Figure 4.10: Example of the 2D texture custom specular effect.

The final specular method is the specular sampling technique from Shin *et al.* [7]. This method is implemented by using the hair particles as sampling points for specular values. We choose some arbitrary threshold value and all points found to be above this are grouped together. Multiple groups may be formed if the points are too far apart. These groups are then used to generate a quadrilateral strip facing the camera which we apply a custom highlight texture to.

Unlike the other two techniques, this method is not accessible at run-time in the project application but rather has to be enabled at build-time by uncommenting a pre-processor directive. This is because the method was designed with 2D billboard particles in mind, not 3D strands. The generated quad strips are created in the centre of the strands and can not be seen without experimentation with depth biasing and ID buffers. Even with the work necessary to make the effect visible, this technique does not look correct with 3D strands.

4.3.7 Silhouettes

The hair's silhouette is outlined using one of two techniques. The first method is an image-space edge detection technique based on Redmond *et al.* [32] which is similar

to a method used on cartoon hair before in Shin *et al.* [7]. The second method is an object-space technique from the more general cartoon rendering paper Lake *et al.* [22] with some features from Northrup *et al.* [34] and does not appear to have been applied to hair before.

The image-space method is applied over three steps. First, an edge image is produced. This is generated by a Sobel shader program based on code derived from the OpenGL Superbible [46]. The Sobel shader works by applying two 3x3 kernels across the image, one for vertical and the other for horizontal edges. The pixel values under each cell of a kernel are multiplied by the cell value and then summed with the other 8 pixels. The vertical and horizontal kernel sums are then combined according to the following equation:

$$edge = \sqrt{(horizontalsum^2 + verticalsum^2)}$$

Normally the value of the above equation would be used as the colour for the detected edge but this is not suitable for cartoon hair as we have no real control over what colour that may be so the value is instead thresholded and set to a specific colour (either white or black in this case).

The edges in the edge image are at this stage only 1 pixel wide so the second step is to increase their width (fig 4.11a). This is done by a dilation shader that works by converting pixels to edges if they are within a certain distance of an existing edge (fig 4.11b). The distance can be user defined which allows for a custom edge thickness.

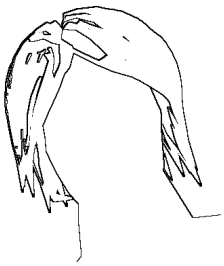
(a) The scene after an edge detection pass. (b) The edge image after a dilation pass.

Figure 4.11: The steps of the image-space silhouette method.

The edge detection and dilation passes are designed for use on 2D images so they can not be used directly in the scene but rather must be applied to the image buffer.

Conveniently, the image buffer can be easily obtained by using the in-built Ogre compositor.

The final step is to combine the dilated edge image with the final rendered scene in the compositor. It should be noted that this project does not use the original scene for the edge detection. If the original image were to be used directly, the boundaries between different lighting intensities and texture features would also be highlighted ruining the silhouette effect (fig 4.12a). The erroneous edges are eliminated by not using the original scene but by re-rendering it using a solid colour shader on the objects as in figure 4.12b.



(a) The edge image when Sobel is applied to the normally lit scene.

(b) The scene rendered using solid shading.

Figure 4.12: Using a solid shaded scene is essential for clean image-space edges.

There is however, a small problem with applying the solid shading as the compositor does allow the re-rendering of the scene but does not give direct access to materials. Ogre materials however, can have different schemes associated with them which the compositor can switch between. Schemes are an Ogre feature that allow you to specify alternative ways for rendering a material and are commonly used to create simpler materials for weaker computers without having to define a whole new material. In this case, we use the schemes to switch between fully textured and solid shaded materials. The problem with schemes is that the compositor expects every object in the scene to have the same schemes defined which can lead to a lot of repetition in the material scripts.

In order to remove the need to define the same schemes for every new material we create, the project instead uses a material listener which is associated with the scene and is called whenever a material does not have a scheme defined. When the listener encounters a request for a *solid* scheme it returns a generic solid material technique.

Once the compositor has done its work the final result should look something like the example in figure 4.13.

Figure 4.13: The final result of the Sobel based silhouette method.

The object-space edge detection works by iterating through all the triangles in the hair strand meshes and comparing them with their neighbours. If a front facing triangle is joined to a back facing triangle, the border edge in between is a silhouette (fig 4.14). The problem with this method is that the triangles are stored in a linear fashion for rendering with no regard for which triangles they neighbour.

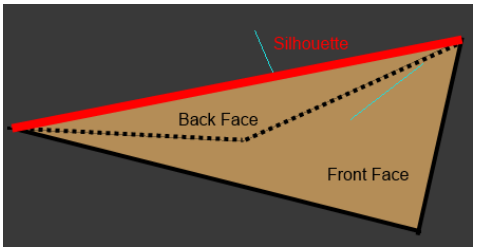


Figure 4.14: The edge joining a back and front facing triangle is a silhouette edge.

Before any detection can occur we must first generate a data structure containing these relationships. This structure is created by iterating through every triangle and breaking them up into their three edge segments. These edges along with their associated triangle are then input into a hash table using the sum of the edge vertex indices as the key. As all of the triangles neighbour three other triangles, we will eventually

encounter duplicate edges. When duplicates are found, we append the triangle of the duplicate edge to the existing hash table entry. Once all edges have been processed, we have a data structure containing every unique edge and their neighbouring triangles.

No vertex positions are used while generating the acceleration data structure, only the indices. As the indices do not change unless additional triangles are added, and they are shared by all strands (being used to index their unique vertex lists), we only have to calculate the relationships once.

This edge map is used for every strand, every update cycle by iterating through it and using the triangle indices associated with each edge entry to find the vertices of every pair of neighbouring triangles. The surface normals are then calculated and the dot product of the normals and the camera viewing direction is determined. If the dot product is less than or equal to 0 then the edge is a silhouette and is added to a silhouette list. The silhouette list can not be used to make a continuous silhouette edge yet, as the edges are stored in no particular order.

The unsorted silhouette list is sorted using a custom insertion sort algorithm. Here is how works: If the list is empty, the first element we attempt to add will be automatically accepted. The two indices of all subsequent entries however, will be compared with the first index of the top element and the second index of the bottom element. If the edge shares one of these indices, it will be added to either the top or bottom of the list accordingly. The entry's indices may also flipped around so that they are in the same order as the rest of the list. If no indices are shared, the entry is added to a temporary list. This temporary list is then checked after every new insertion attempt to see if the new entry has opened a position for it. Eventually every entry should be added to the list as each strand is a closed mesh so the silhouette should form a loop.

In order to create the final silhouette stroke we use a method from Northrup *et al.* [34]. The silhouette positions are converted to screen space by multiplying them by the camera's view and projection matrices like so:

$$screenPoint = projection * view * worldPoint$$

The reason we convert the points to screen space is to simplify making the silhouette strip face the camera as we can easily extrude new points parallel to the viewport this way without any complicated rotations. We should have something similar to figure 4.15 at this point.

Figure 4.15: The silhouette edge after sorting.

We generate the quad strip by iterating through the silhouette points and extruding along normals generated perpendicular to the normalised sum of the two edges sharing the current point (fig 4.16). A number of scaling techniques can be applied to these new points to adjust their appearance. They can be scaled based on the angle between the two edges to reduce sharp corners or based on depth to help maintain size at different zoom levels.

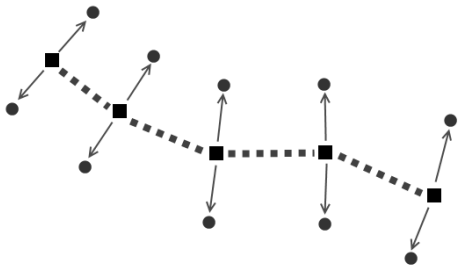


Figure 4.16: The silhouette edge after extruding additional points.

Once the new points have been generated, the original points are discarded and the new ones are converted to world space by applying the inverse of the original matrix operation. They are converted to world space rather than being left in screen-space because the default graphics pipeline expects them to be in that format. If we wanted to keep them in screen-space, we would need to write custom shader programs for every rendering step even if we didn't want to implement any fancy effects. These points are then passed to Ogre where texture coordinates are generated, and they are finally rendered as long triangle strip primitives (fig 4.17). The appearance of the stroke can be easily adjusted by specifying a custom texture to apply to the strip; also, by using the underlying strand geometry to generate the silhouette edges, this method is

naturally spatially coherent.

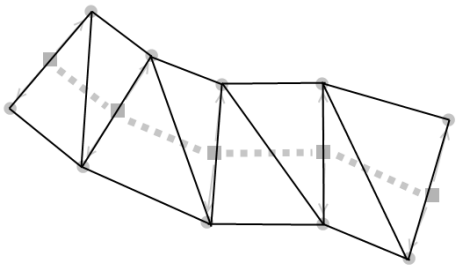


Figure 4.17: The silhouette edge being used to generate a triangle strip.

When rendering objects, the graphics card typically determines visibility by using a depth buffer which compares the z axis (relative to the camera) position of the vertices and in the event of an overlap renders the closest vertex. This method does not work for our 3D silhouettes as they were generated to face the viewport and might partially or completely penetrate the surface of the strands.

We therefore have to disable the depth buffer and instead perform visibility testing using an ID buffer method based on Northrup *et al.* [34]. The ID buffer is simply an image of the scene with everything but the hair blacked out and the strands rendered with unique solid colours and the silhouettes as pure white.

The ID buffer is passed to the silhouette shader where it can compare each fragment pixel to the same position in the ID buffer. The ID colour for the current pixel can be found by converting the fragment coordinates to screen space coordinates and using them to index the ID buffer like a texture [47].

Each silhouette strip is aware of the ID colour of the strand it was generated from. When rendering the strips, we compare each pixel with the same pixel in the ID buffer, if the colour is either white or the correct ID colour, we can draw the silhouette, otherwise we leave it transparent.

The appearance of the textured edges can be significantly altered by adjusting how the silhouette edges are drawn in the ID buffer. By changing a number of pre-processor directives when compiling the project application, the edges can be shaded pure white, they can be textured with a white version of the stroke texture or they can be omitted entirely.

If pure white is used, the parts of the stroke texture not on surface of the parent strand should be visible. However, as the depth buffer is not used, if two silhouettes

overlap one another, one may be visible through any transparent parts of the stroke even if normally hidden behind another object. Using a textured silhouette in the ID buffer should solve this problem as only non-transparent parts of the stroke will appear. Unfortunately, even though the ID buffer resolution is the same as the scene's image buffer resolution, the pixels do not always match up (perhaps due to floating point error) and this results in a patchy final result. Omitting the silhouette entirely ensures that no overlap issues will occur but also means that only half of the silhouette (the part overlapping the strand that generated it) will be visible.

The reason why the colour white is used to shade the silhouette in the ID buffer rather than the colour of the associated strand and why the overlapping silhouette issue can not be fully resolved, is that certain overlaps are actually desirable. If the same silhouette colour is not used, silhouettes that should otherwise be visible through transparent parts of an overlapping texture might be hidden. White in particular is used as it is highly unlikely to match an existing strand ID colour as it is the last colour to be issued and over 8000 strands would need to be created before then. This project only uses between 100 and 200 strands although if more strands were required the increment (introduced to reduce floating point error) between successive IDs could be reduced, giving us more strands before a white ID occurs.

Using an ID buffer instead of a depth buffer does lead to one rather significant problem. While the buffer can help us distinguish between different strands, it can not identify if the strand is overlapping itself. This would not be an issue if we were using the silhouette technique on rigid convex objects but with dynamic strands this means that we might see a silhouette that should otherwise be concealed. An example of this issue can be seen in figure 4.18. There is no simple way to prevent this issue but as the effect is most noticeable near the root, it can be alleviated by fading the silhouette as it goes towards the root [7].

In this project, we fade the intensity by iterating through the silhouette points and finding the nearest hair particle to that point. We then determine how far that point is down the length of the strand and assign an alpha value from 0 to 1 to the relevant silhouette points. These alpha values are then passed to the silhouette edge shader as part of the otherwise unused ID colour alpha channel and are used to fade the silhouette. The amount the silhouette is faded can be controlled by a quadratic curve. The final result should look something like figure 4.19.

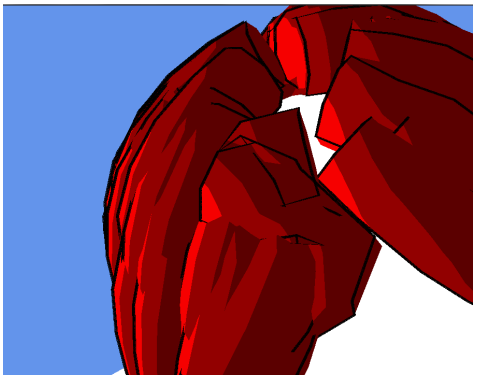


Figure 4.18: Some silhouettes can be seen overlapping themselves as no depth buffer is used.

Figure 4.19: The silhouette faded towards the strand roots.

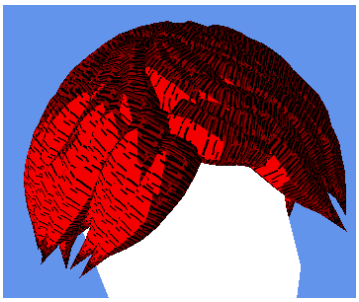
There is one final issue with this silhouette method. Ogre by default, renders objects with different associated materials and shaders, in different passes. As the depth buffer is not disabled for objects other than the silhouettes, this can cause problems with the rendering order. If the silhouettes are rendered first, then the hair and character will appear on top of them rather than under. This is fixed by separating the objects into strict rendering groups and assigning a rendering order using Ogre's *setRenderQueueAndPriority* function.

4.3.8 Hatching

In order to give the appearance of pencil, pen or brush strokes on the strand surface, a hatching effect is applied to the hair mesh. Two methods in particular are used to implement this effect, one derived from Lake *et al.* [22] and the other from Praun *et al.* [30].

Both methods apply a hatching texture based on the light intensity of the hair strand, but they differ in how they project the textures. The method by Lake *et al.* [22] projects the hatching onto the viewport, producing a flat effect that looks convincing as hand drawn when viewed with a static scene (fig 4.20a). If the strands are moving though, the strokes remain static relative to the viewport and the effect is ruined.

(a) Hatching projected onto the screen.



(b) Hatching used as a surface texture.

Figure 4.20: The two ways to apply hatching strokes.

The method by Praun *et al.* [30] on the other hand, projects the hatching onto the surface of the strands by using them as textures for the surface triangles (fig 4.20b). This method requires more work to appear coherent but works better for dynamic scenes. The textures in particular, need to be blended to ensure a smooth transition across the surface. Praun *et al.* [30] uses a technique known as single pass 6-way blending; however, the exact method used is rather out of date as it is concerned with multi-texturing limitations no longer present on modern graphics cards. This project uses an updated version of the technique instead which is derived from a student implementation [48].

The hatching textures can be created by hand using an image editor like GIMP but it can be difficult to make them tileable and coherent this way. A hatching texture

generator written for this project based on specifications given by Praun *et al.* [30] is used to create them instead.

The program works by taking a single stroke texture such as the one in the appendix (figure 6) and using it to generate a number of textures of different tonal values. The strokes are not applied randomly since they are not likely to be uniformly distributed. Instead, a number of potential candidate strokes are randomly generated [30]. The candidate with the greatest tonal contribution is the one chosen.

In order to ensure coherency when shifting between tonal values, the strokes of the lighter toned textures are nested within the darker toned textures [30]. The strokes when used with the projection based hatching keep the same thickness and position regardless of depth but with textured triangles, the thickness can change and the perceived stroke density can increase. The stroke density and thickness are kept coherent by generating custom mipmaps which drop strokes but increase stroke width at each mipmap level. The project uses this method to create 6 hatching textures with 3 mipmap levels each. Before we can use the mipmaps they must be first compiled in an image editor such as GIMP into dds format as this format allows for custom mipmaps and can be read by Ogre.

The process of compiling the textures into the required dds format is unfortunately rather tedious. No tools could be found to perform the process automatically so it must be performed manually. In order to create the files, the main texture must be initially loaded into GIMP and immediately saved as a dds file. When saving, automatic mipmap generation must selected. This gives us an image file with a number of hidden layers each half the size of the layer above it, these are the mipmaps. Our custom mipmaps must be then loaded into the current image and then used to replace the corresponding mipmap layer. We generate automatic mipmaps only to override them rather than create new layers because no clear way to do so could be found.

(a) Example of poor tiling. (b) Example of good tiling.

Figure 4.21: The ability for textures to tile is important.

If the hatching textures at this point are tiled, the borders between the textures will most likely not match up and we will see something like figure 4.21a. The hatching textures are made tileable by creating an additional stroke per candidate which is normally not visible. When a stroke is placed near the border of the hatching texture and overlaps the edge however, the extra stroke is made visible in position, the same distance from the opposite border, as can be seen in figure 4.22. This ensures that the strokes that are cut off at the border continue seamlessly into the next texture and the transition between them becomes difficult to spot. This effect can be seen in action in figure 4.21b.

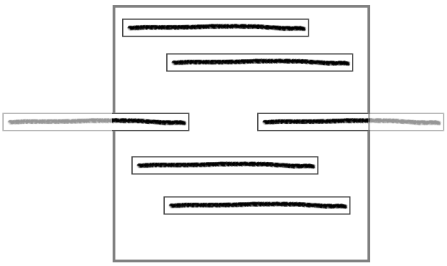


Figure 4.22: Strokes wrap around at the border with the help of an additional stroke that is otherwise hidden.

4.4 User Interface

The final part of the project application implementation is the user interface. This is the series of menus through which the user can interactively alter various parameters and view the results in real-time. Prior to the addition of these menus the only way to modify the hair was to alter a number of pre-processor directives and recompile the project. Recompiling is not problematic for a relatively small project but if used as part of a much larger code base, it could take a significant amount time.

The UI system is implemented using the CEGUI library which has native support for Ogre. The layout of the menus is defined using the CELayoutEditor. This program allows the user to create frame windows and populate them with checkboxes, text fields and buttons by simply dragging and dropping. The options, layout and style used by this project's interface are set according to the specifications laid out in the design

chapter.

The UI layout is saved to a .layout script which is then read into the project application and used by CEGUI to generate the interface. The GUI elements are then programmatically linked to event handlers which are called when a value changes or a button is pressed. These event handlers are passed pointers to functions created for this project that perform whatever action is desired when a specific event occurs.

Chapter 5

Evaluation

In this chapter, we conclude the development of the project application by discussing the various performance and acceptance tests carried out on the resulting program to determine its success at meeting the project's stated goals. The testing was composed of performance and user acceptance tests.

Performance testing consisted of measuring the hair exporter, the hatching texture generator and the project application for total completion time and frame rate (where appropriate) to see if the stated goal of real-time performance was achieved. In user testing, a number of volunteer participants watched a number of clips of the simulated hair with various effects applied and then rated them according to style and appeal.

All testing was performed on the development computer which was a Lenovo Thinkpad Edge E520 laptop with an Intel Core i3-2330M 2.20Ghz processor, 4 GB of RAM and running Windows 7 64-bit.

5.1 Performance Testing

5.1.1 Hair Exporter

The hair strand exporter was tested by measuring the completion time for export jobs of varying strand numbers. The process was found to be near instantaneous with around 100-150 strands, at a resolution of 5 segments per strand. The final XML file was also relatively small at about 40 KB.

Texture Size	Strokes	Candidates	Time to generate (min:sec)
256x256	78	10	0:18
512x512	78	10	1:05
1024x1024	78	10	4:00
512x512	36	10	0:30
512x512	120	10	1:36
512x512	78	1	0:13
512x512	78	20	2:00

Table 5.1: Hatching texture generation times at varying resolutions, stroke amounts and candidate numbers. The effect of varying texture size, number of strokes and finally, number of candidates was tested.

When applied to more dense hair, the response time started to become more noticeable. At 1000 strands, the completion time was around 3 seconds, and as expected the hair strand file increased 10 times in size to around 400 KB. The response time and file size became unmanageable when trying to replicate the 100,000 strands on the average human head. When tested, the process lasted more than 23 minutes before the task had to be ended prematurely, and the resulting file size could have been expected to be in the range of 40 megabytes.

From the above results, it can be seen that the exporter is acceptable for the sparse hair model used in this project; however, it is completely unsuitable for modelling realistic hair numbers.

5.1.2 Hatching Texture Generator

The hatching generator was tested by measuring the time from start to completion at different texture sizes, stroke amounts and potential candidate stroke (used for ensuring a uniform distribution) numbers for 6 hatching textures with 3 mipmap levels. The results can be seen in table 5.1.

By analysing the results, the following observations can be made. The texture size has the greatest influence on the completion time with more than 3 minutes added by increasing the final texture from 256x256 to 1024x1024. The texture size has such a great effect due to the way in which the program checks the tonal value of the texture after placing a stroke. The texture cannot be queried directly so it must be expensively copied to memory and then iterated through. The larger the texture, the longer the

process. The final application uses 512x512 as the default size as it appears to be a fair compromise between detail and speed.

The number of candidate strokes has a reasonably large effect on the performance of the generator. Using the same amount of final strokes, the final time can vary from 13 seconds to over 2 minutes depending on the number of candidate strokes. The obvious choice would be to lower the number of candidates but they are extremely important to the distribution of the strokes in the final image. It can be seen in the appendix (figure 7) that using just 1 candidate results in a sparser and less uniform image. The higher the number of candidates, the more uniform the final texture should appear, for example figure 8 in the appendix uses 10 candidates and figure 9 in the appendix uses 20.

The number of final strokes used has the least direct effect on performance as an additional 84 strokes only adds an extra minute to the final time although it should be noted that larger textures will require more strokes if they are not also scaled up accordingly. A higher number of strokes also means a higher number of candidates will need to be checked and for X strokes with Y candidates, a total of $X * Y$ expensive tonal measurements have to be made.

5.1.3 Project Application

The project application, being the main focus and by far the most detailed component of this undertaking, was the most difficult to test. There are a large number of possible variables so the project was not tested for every possible combination. Features were instead tested in isolation using the default parameters for all other variables. The default parameters unless otherwise stated were:

Resolution 1024x768

Strand Numbers 100

Hair Segments 5

Hair Sampling Points 8

Vertices Per Point 8

Screen Resolution	FPS
800x600	28
1024x768	28
1366x768	28

Table 5.2: Screen resolution effect on project application performance.

The effect of the viewport resolution was tested by measuring the average frame rate at different screen sizes. The highest resolution tried was 1366x768 as that was the size of the development computer's screen. The results are listed in table 5.2. The measurements show that the viewport resolution does not have a perceivable effect on the performance of the program.

The number of strands used has a very big effect on the frame rate of the project application. Using a small number of strands, the frame rate can reach into the hundreds but as we approach 150 strands the performance drops drastically (fig 5.1). This is likely due to the expensive base mesh creation and silhouette detection steps.

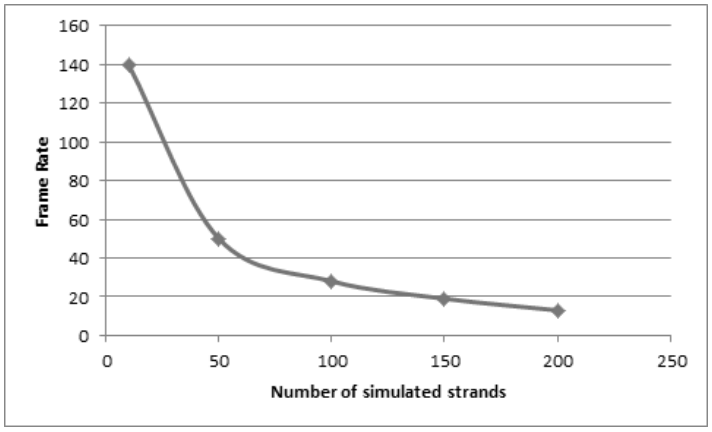


Figure 5.1: Frame rate performance for different numbers of simulated strands.

The base mesh generation appears to be a bigger drain on performance than the silhouette detection, based on the results in table 5.3. The hair and shape resolution give a large 9-12 frame rate increase when only slightly reduced while the silhouette detection schemes only vary the frame rate a small amount using the same settings.

The performance of the two silhouette detection schemes does seem to vary greatly when they are changed from their default parameters. The thickness of the quad strips can be increased with no change in frame rate but using the Sobel method, a larger

Hair Sampling Points	Vertices Per Point	Silhouette Type	FPS
6	4	object-space	37
8	8	object-space	28
32	32	object-space	3
6	4	image-space	44
8	8	image-space	32
32	32	image-space	4

Table 5.3: Performance with different hair and shape resolutions depending on the silhouette outline method used.

Dilation	Stroke Limit	FPS
1	-	32
12	-	23
-	0.1	28
-	1	28

Table 5.4: Performance when increasing stroke width. Dilation is used for image-space and stroke limit for object-space silhouette detection. A dilation value of 12 and a stroke limit value of 1 produce an edge of similar thickness.

dilation kernel needs to be applied. In table 5.4, it can be seen that the larger the dilation kernel, the greater the hit to the frame rate while a quad of the same thickness would experience no drop in performance.

Applying animation to the model does not seem to affect the frame rate much, according to table 5.5. There is no drop in performance when the hair is static or when it is being moved by the character's skeleton. There is a slight drop however, when the anchor particles are animated but this is still within real-time. The model appears to perform better when the anchor blending springs are stiffer as the frame rate dipped below 25 when the blending springs were released. The improved performance from stiffer blending springs is likely due to the number of collisions avoided by preventing the hair from moving too much.

5.2 User Testing

The effectiveness of the various visual and motion based techniques developed for this project were evaluated through a number of user tests. These tests consisted of a

Anchor Stiffness	Animated Skeleton	Animated Hair	FPS
0.01	no	no	28
0.01	yes	no	28
0.01	yes	yes	26
0	yes	yes	23-26
1	yes	yes	23-26

Table 5.5: Performance when predefined animations are added.

volunteer viewing 43 short clips and rating them based on their appeal and effectiveness in emulating a cartoon style. The target style was Japanese anime hair like in Shin *et al.* [7].

The clips contained examples of static or moving hair, hair with or without effects enabled and so on. Due to the large number of variables, all possible scenarios could not be included and rather than testing combined effects or specific parameter values, we instead tested effects in isolation with their parameters set to default or extreme values. The animation potential for the anchor particles was unfortunately not thoroughly tested. The anchor particle animation is more or less ignored due to the skill required to produce convincing animations, unless created by someone skilled in animating hair, the results would undoubtedly be poor.

In order to undertake these tests, university ethical approval was first sought and all volunteers filled in participant information and consent forms. The videos were shown to the user through the use of a Python script which used OpenCV to iterate through and play each video specified in a list. The user watched each video and answered the two prompts given afterwards. These prompts asked them to rate the appeal from 1 to 5 (1 being very appealing and 5 being very unappealing) and to rate the effectiveness replicating a cartoon style (1 being very effective and 5 being very ineffective). A total of 5 volunteers completed the user testing. It should be noted that due to short time frame, the participants for the user testing portion of the evaluation were drawn largely from family and friends.

In general the cartoon effect was liked; however, some interesting observations can be made. The basic Blinn effect was found to be more appealing than the 2D specular texture method and rated high for emulating the cartoon hair style (fig 5.2). This remained true for various highlight sizes. The specular texture method was still con-

sidered appealing although users much preferred smaller highlights.

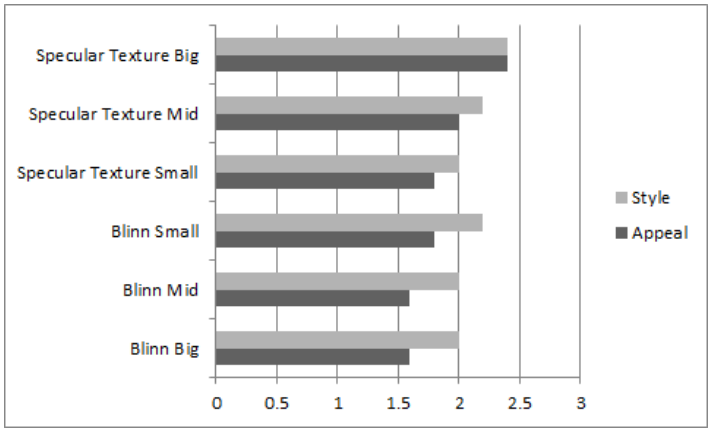


Figure 5.2: Scores for the two specular texture methods.

The back lighting effect was given rather negative ratings and actually detracted from the style (fig 5.3). In terms of silhouette techniques, the object-space method was higher rated than the image-space method. Users were also found to prefer thick outlines and surprisingly preferred the unfaded quad strips despite the overlapping near the hair root.

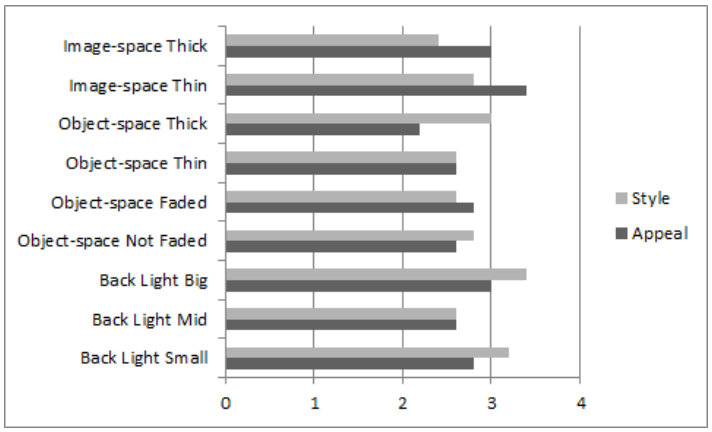


Figure 5.3: Image-space, object-space and back lighting scores.

The viewport and surface hatching methods were given rather similar, neutral ratings for appeal and style (fig 5.4). Users did not express a preference over large or small strokes with the viewport hatching, but they did prefer big strokes when used with the surface hatching.

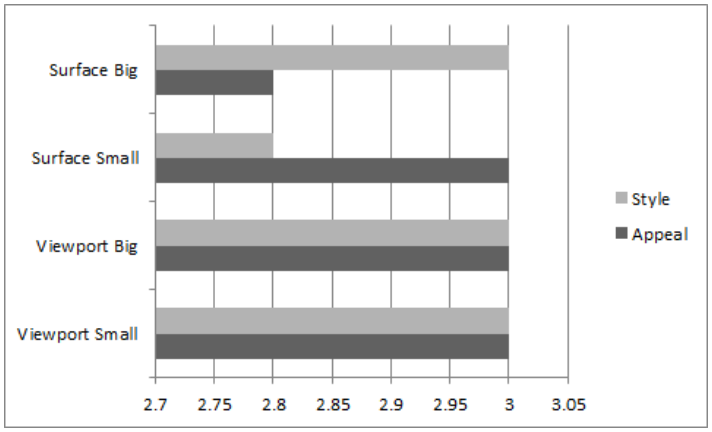


Figure 5.4: Scores for the hatching methods.

The cartoon shading with depth dependent detail was preferred over the effect without it (fig 5.5). As expected, the hair was found more appealing at higher resolutions although the shape resolution was preferred at the default 8 vertex resolution. High resolution hair and shapes combined were found even more appealing than the effects in isolation although there was no increase in scores for style. The hair was found most appealing and closest in terms of cartoon style when the collision margin was left untouched. High margins were found the most unappealing and the furthest away in style. The hair was preferred when it was moving with the skeleton and in terms of predefined animations, the anchor blending springs were preferred loose but not completely absent.

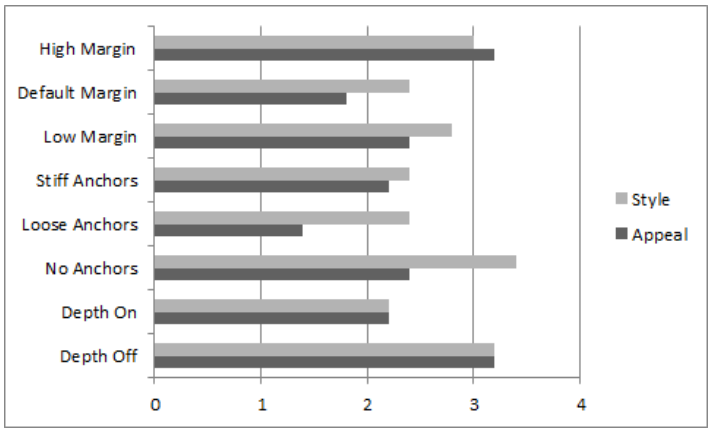


Figure 5.5: Scores for depth fading, anchor stiffness (loose means 0.01 stiffness, stiff means 1) and collision margins (low is 0.1, default is 0.25 and high is 1).

Chapter 6

Conclusions and future work

The stated objective to implement a real-time cartoon hair simulation and rendering pipeline has been achieved. A number of novel rendering features that do not seem to have been used with cartoon hair before have been implemented. These features included object-space silhouettes, surface hatching and extended 2D cartoon texturing effects such as custom specular responses, back ligthing and depth dependent diffuse shading.

In the process, a number of tools have also been developed such as the Blender hair exporter and the hatching texture generator, and of course the main project application has been implemented. The final project application is capable of taking in a number of hair strand files, converting them into a simulation model and finally, generating a 3D strand mesh for rendering upon which it can apply numerous stylisations with user configurable parameters.

The project application is capable of real-time performance on modest hardware using standard CPU programming for the simulation and mesh generation and shader programming for rendering. No specialised techniques such as general purpose graphics card programming or geometry shaders are necessary, meaning most computers should be able to run the program without needing powerful gaming quality graphics cards.

In terms of user acceptance, the results are positive but a little more mixed. People find the final hair appealing in general, but unexpectedly they prefer, in some cases, the more basic implementations of effects over their more complex variants. This is likely due to the limited 2D textures on which the techniques are based rather than

the methods themselves.

Plenty has been left for potential future work. More varieties of 2D textures could be created and tested and these would likely lead to an increase in the user appeal for the advanced rendering methods over the more basic variants.

The Ogre compositor system could be researched further. The way in which you enable and disable the compositor is not entirely clear in the Ogre documentation and as result, in the current application, once you apply and then remove the image-space silhouette effect you cannot re-enable it without restarting.

The simulation model could be significantly expanded. In particular, hair-hair collisions could be implemented as their absence was noted by some people during the user testing. Utilising more distributed programming techniques such as multi-threading or general purpose graphics card programming could also be investigated as it would likely lead to a vast increase in the segment resolution and the number of simulated strands.

Appendix

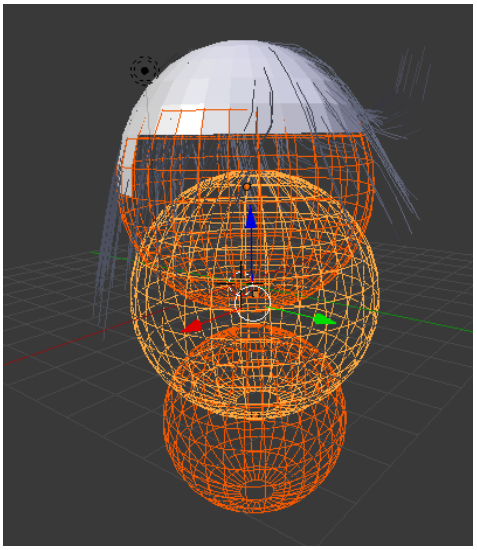


Figure 1: Approximating the head collision shape using spheres.

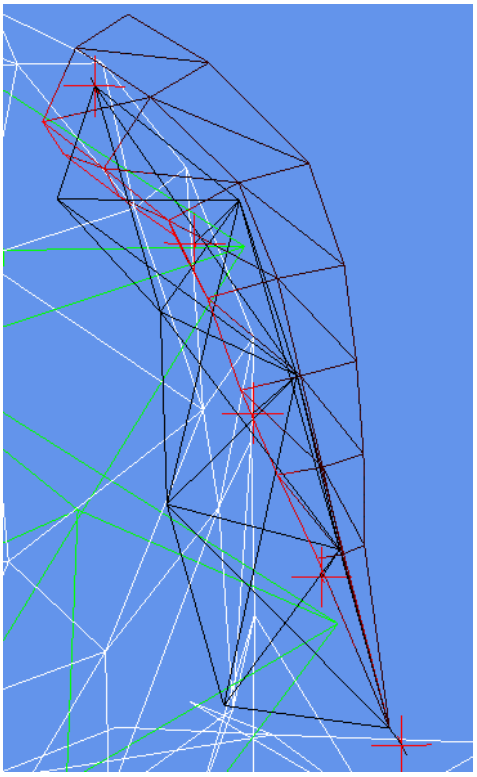


Figure 2: The base mesh generated around a strand.

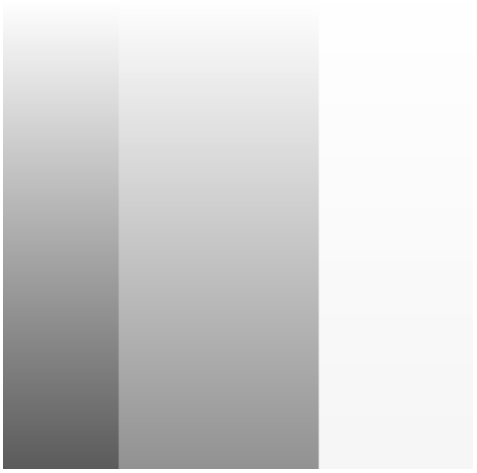


Figure 3: The 2D cartoon texture.

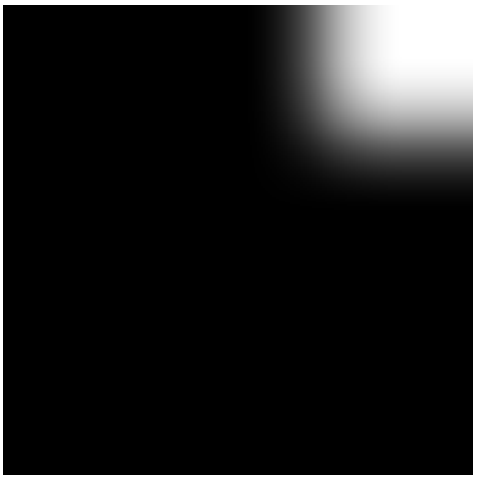


Figure 4: The 2D specular texture.

Figure 5: The 2D back lighting texture.

Figure 6: The stroke used to generate hatching textures in this project.

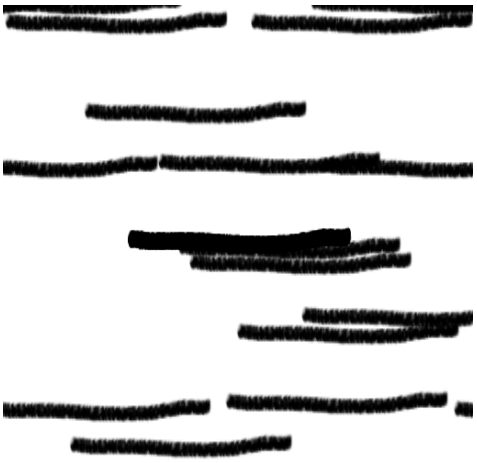


Figure 7: Hatching texture using 1 candidate stroke.

Figure 8: Hatching texture using 10 candidate strokes.

Figure 9: Hatching texture using 20 candidate strokes.

Bibliography

[1] Kaplan, M., Gooch, B., and Cohen, E. Interactive artistic rendering. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering* (New York, NY, USA, 2000), NPAR '00, ACM, pp. 67–74.

[2] Selle, A., Lentine, M., and Fedkiw, R. A mass spring model for hair simulation. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 64:1–64:11.

[3] Ward, K., Bertails, F., yong Kim, T., Marschner, S. R., paule Cani, M., and Lin, M. C. A survey on hair modeling: styling, simulation, and rendering. In *IEEE TRANSACTION ON VISUALIZATION AND COMPUTER GRAPHICS* (2006), pp. 213–234.

[4] Rosenblum, R. E., Carlson, W. E., and Tripp, E. Simulating the structure and dynamics of human hair: Modelling, rendering and animation. *The Journal of Visualization and Computer Animation 2*, 4 (1991), 141–148.

[5] Chang, J. T., Jin, J., and Yu, Y. A practical model for hair mutual interactions. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2002), SCA '02, ACM, pp. 73–80.

[6] Noble, P., and Tang, W. Modelling and animating cartoon hair with nurbs surfaces. In *Computer Graphics International, 2004. Proceedings* (2004), pp. 60–67.

[7] Shin, J., Haller, M., and Mukundan, R. A stylized cartoon hair renderer. In *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology* (New York, NY, USA, 2006), ACE '06, ACM.

[8] Anjyo, K.-i., Usami, Y., and Kurihara, T. A simple method for extracting the natural beauty of hair. *SIGGRAPH Comput. Graph. 26*, 2 (July 1992), 111–120.

[9] Featherstone, R. *Robot Dynamics Algorithm*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[10] Bertails, F., Audoly, B., Cani, M.-P., Querleux, B., Leroy, F., and Lévêque, J.-L. Super-helices for predicting the dynamics of natural hair. *ACM Trans. Graph. 25*, 3 (July 2006), 1180–1187.

[11] Sugisaki, E., and Yu, Y. Simulation-based cartoon hair animation. In *In 13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG05)* (2005), pp. 117–122.

[12] Hadap, S., and Magnenat-Thalmann, N. Modeling dynamic hair as a continuum. *Computer Graphics Forum 20*, 3 (2001), 329–338.

[13] Bando, Y., Chen, B.-Y., and Nishita, T. Animating hair with loosely connected particles. *Computer Graphics Forum 22*, 3 (2003), 411–418.

[14] Volino, P., and Magnenat-Thalmann, N. Animating complex hairstyles in real-time. In *Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2004), VRST '04, ACM, pp. 41–48.

[15] Koh, C., and Huang, Z. Real-time animation of human hair modeled in strips. In *Computer Animation and Simulation 2000*, N. Magnenat-Thalmann, D. Thalmann, and B. Arnaldi, Eds., Eurographics. Springer Vienna, 2000, pp. 101–110.

[16] Koh, C. K., and Huang, Z. A simple physics model to animate human hair modeled in 2d strips in real time. In *Proceedings of the Eurographic workshop on Computer animation and simulation* (New York, NY, USA, 2001), Springer-Verlag New York, Inc., pp. 127–138.

[17] Bertails, F., Kim, T.-Y., Cani, M.-P., and Neumann, U. Adaptive wisp tree: a multiresolution control structure for simulating dynamic clustering in hair motion. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2003), SCA '03, Eurographics Association, pp. 207–213.

[18] Petrovic, L., Henne, M., and Anderson, J. Volumetric methods for simulation and rendering of hair. *Pixar Technical Memo 06-08*

[19] Newman, T. S., and Yi, H. A survey of the marching cubes algorithm. *Computers & Graphics* (2006), 854–879.

[20] Lewiner, T., Lopes, H., Vieira, A. W., and Tavares, G. Efficient implementation of marching cubes cases with topological guarantees. *Journal of Graphics Tools 8*, 2 (december 2003), 1–15.

[21] Hyun, D.-E., Yoon, S.-H., Kim, M.-S., and Jüttler, B. Modeling and deformation of arms and legs based on ellipsoidal sweeping. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2003), PG '03, IEEE Computer Society, pp. 204–.

[22] Lake, A., Marshall, C., Harris, M., and Blackstein, M. Stylized rendering techniques for scalable real-time 3d animation. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering* (New York, NY, USA, 2000), NPAR '00, ACM, pp. 13–20.

[23] Spindler, M., Rber, N., Dhring, R., and Masuch, M. Enhanced Cartoon and Comic Rendering . In *Proceedings of Eurographics 2006, Short Papers* pp. 141–144.

[24] Barla, P., Thollot, J., and Markosian, L. X-toon: an extended toon shader. In *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering* (New York, NY, USA, 2006), NPAR '06, ACM, pp. 127–132.

[25] Phong, B. T. Illumination for computer generated pictures. *Commun. ACM 18*, 6 (June 1975), 311–317.

[26] Blinn, J. F. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph. 11*, 2 (July 1977), 192–198.

[27] Cook, R. L., and Torrance, K. E. A reflectance model for computer graphics. *ACM Trans. Graph. 1*, 1 (Jan. 1982), 7–24.

[28] Rusnell, B. X-toon: An extended toon shader, 2006. Accessed: 3/8/13, `http://brennanrusnell.com/projects/xtoon.php`.

[29] Buchin, K., and Walther, M. Real-time per-pixel rendering with stroke textures. In *In SCCG 03: Proceedings of the 19th spring conference on Computer graphics* (2003), ACM Press, pp. 125–129.

[30] Praun, E., Hoppe, H., Webb, M., and Finkelstein, A. Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 581–.

[31] Zhao, M., and Zhu, S.-C. Customizing painterly rendering styles using stroke processes. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering* (New York, NY, USA, 2011), NPAR '11, ACM, pp. 137–146.

[32] Redmond, N., and Dingliana, J. A Hybrid Approach to Real-Time Abstraction. In *Proceedings of Eurographics Ireland, 2009* (2009).

[33] Markosian, L. *Art-based modeling and rendering for computer graphics*. PhD thesis, Providence, RI, USA, 2000. AAI9987803.

[34] Northrup, J. D., and Markosian, L. Artistic silhouettes: a hybrid approach. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering* (New York, NY, USA, 2000), NPAR '00, ACM, pp. 31–37.

[35] Kalnins, R. D., Davidson, P. L., Markosian, L., and Finkelstein, A. Coherent stylized silhouettes. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, ACM, pp. 856–861.

[36] Bullet - game physics simulation. Accessed: 21/8/13, `http://www.bulletphysics.com/`.

[37] Yuksel, C., Schaefer, S., and Keyser, J. Hair meshes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2009) 28*, 5 (2009), 166:1–166:7.

[38] Blender 3d. Accessed: 21/8/13, `http://www.blender.org/`.

[39] Gimp - gnu image manipulation program. Accessed: 21/8/13, http://www.gimp.org/.

[40] Github. Accessed: 21/8/13, https://github.com/.

[41] Ogre - open source 3d graphics engine. Accessed: 21/8/13, http://www.ogre3d.org/.

[42] Crazy eddie's gui system for games (open source). Accessed: 21/8/13, http://www.cegui.org.uk/wiki/index.php/Main_Page.

[43] Opencv. Accessed: 21/8/13, http://opencv.org/.

[44] TinyXML-2. Accessed: 24/8/13, http://www.grinninglizard.com/tinyxml2/.

[45] Definition of contre-jour in english. Accessed: 3/8/13, http://oxforddictionaries.com/definition/english/contre--jour.

[46] Image processing with glsl shaders. Accessed: 22/8/13, http://www.blitzbasic.com/Community/posts.php?topic=85263.

[47] Phasersonkill - Screen Space Texture Coordinates. Accessed: 22/8/13, http://www.phasersonkill.com/?p=495.

[48] The term project report for computer graphics 2004: Real-time hatching, 2004. Accessed: 3/8/13, http://www.cmlab.csie.ntu.edu.tw/~daniel/projects/hatching/hatching.htm.