Utilizing Particle Systems for Strand Animation

Magnus Raunio, 900531-5131
Phone: 0760380227
Address: Häggvägen 15 A, Tyresö 13552
mraunio@kth.se

Kungliga Tekniska högskolan
Degree Project in Computer Science, First Level - DD143X
Supervisor: Michael Minock

May 20, 2012

Abstract

In this paper we look at earlier research into particle systems and particularly on how they have been used for creating strands. We use this knowledge to implement our own particle system to simulate strands, in order to understand the strengths and weaknesses of particle systems better. What we find is that particle systems give highly detailed images but that they are very costly to compute when the number of particles increase. As such, when implementing particle systems you should create different level of details of the particle system which limits the number of particles depending on the distance from the viewer as well as try to use approximative fast algorithms to compute the particles behaviours.

Contents

1 Introduction **3**
1.1 Overview of the document 3

2 Background **4**

3 Approach **5**

4 Implementation **6**
4.1 Particle System Structure 6
4.2 Particle System Rules 7
4.3 Presentation 9

5 Result **12**

6 Discussion **13**

7 Conclusions **15**

8 References **16**

1 Introduction

Particle systems is a pretty old technique that was first mentioned in a article by William T. Reeves [9]. The purpose being to model ”fuzzy” objects which do not have a specific shape but instead behave and change according to a pattern. The idea about these ”fuzzy” objects was created in order to create a flame effect for a movie but Reeves also mentions the ability to use the same technique in order to model more tangible objects like grass. As such; particle systems were one of the first approaches to creating strand models which is a research area that later has been further developed in order to create realistic fur or hair in animated movies.

The modern approach to model strands is usually more advanced than the way suggested by Reeves in ’83 since it can be quite costly to use a particle system where each particles lifespan represents a strand. But most of these methods are still at their essence based on particle systems which is why it’s interesting and required to understand the basics of particle systems if you wish to create realistic strand animations.

Because of this we will try to gain an in depth understanding of strand animation using particle systems by trying to implement a fast real time running particle system. We hope that this will highlight the strengths and weaknesses of the traditional approach to strand animations as well as give readers a good idea of how to implement their own system for strand animations. This is of interest mainly because these kinds of systems are being more and more used in games since the hardware exists to support these kinds of real time computations of large particle systems. When you have the ability to pre render the particle system, like for a movie, the issues of computation speed it not as important.

1.1 Overview of the document

In section 2 we will review earlier research into strand animation and particle systems as well as bring up other techniques for modelling strands. Section 3 contains information on the general structure of the strand implementation and what we aim to achieve with the implementation. The detailed description of the implementation is given in section 4. In section 5 we describe what the final implementation was able to achieve in comparison to the goals we set up in section 3. Section 6 evaluates the strengths and weaknesses of our implementation of a strand animation system and what you could do to improve the current implementation. Finally section 7 will contain information about the project as whole and summarise sections 5 and 6.

2 Background

As mentioned in section 1 strand animation using particle systems has been around since 1983 [9]. A particle system is structured such as there is a source for particles, called an emitter, which creates, updates and kills particles. Once a particle is created its life cycle begins and the particles attributes are updated by the emitter until the life cycle end and the particle is killed. When using particle systems for strand animation you allow a particles life cycle represent a strand. This is done by calculating the particles initial attributes for the the whole lifespan in the creation phase, so the time between the creation and destruction of the particle determines the strands length. In the update phase you then update all the points that compose the strands structure.

The computation of this can get quite expensive depending on which rules you apply and on the number of strands you create. Usually when trying to model strand animations you require thousands of strands to create a good image so a rule with time complexity $O(n^2)$ or greater will become very expensive. Collision between strands would fall within that time complexity and this is one of the problems that many strand animation techniques try to overcome since the lack of collision between strands can become very noticeable when the strands get longer and get larger reach.

A normal problem among strand animation is when you try to create long human hair where the interaction between the strands are essential to create a realistic look. There's mainly two techniques for trying to tackle this issue. The first is to reduce the amount of stands generated by the particle system and instead use the strand generated to create duplicates of them, also known as wisps [8]. The second technique used is to instead of each particle represent a individual strand the particles represent a continuum which behaves like a fluid or similar and you render the strands on this continuum [1].

When creating short strands you do not necessarily have to consider the interaction between strands since the reach is so short, but there exists other methods that are cheaper than particle systems for short strand animation. They do however not provide as much flexibility as particle systems does, but if you're interested in only creating a simple strand animation like fur the following reference is recommended [4].

3 Approach

What we aim to achieve with the strand animation is a realistic as possible animation of hair. The focus will be on the use of particle systems to create a realistic model of the hair movement and behaviour but we will also try and add features such as self shadowing of the hair and whatever else will give the final animation a realistic feel as possible. As such we the project is basically divided into two parts, modelling and presentation.

For the modelling part we will aim to achieve the following effects/rules for the hair behaviour:

Elasticity The strands should try to remain a constant length with allowance of some stretching.

Gravity The strands should fall towards the ground.

Emitter motion The strands should react to change in the emitters position and follow it in a realistic way.

Wind The strands should be able to react to a gust of wind and react differently depending on how the strand lies against the wind.

Collision The strands should be able to collide with object which are given as solid. At collision the strands should give way to the solid object. The strands will however not collide with each other because of the limitations of the approach to strand animation using particle systems which is given in section 2.

Stiffness The strands should have a certain amount of stiffness which would counteract the gravity effect which should make the strands bend towards the ground instead of fall.

After we've successfully implemented these rules we will focus on the presentation of the strands which will include the following:

Smoothness The strands should be smooth and not have jagged edges. This effect could also be achieved in the modelling part by sampling more points from the particles lifespan but this would increase the computation cost of rules since there's more points to calculate. The smoothing effect should therefore be cheap to compute and still give a good result.

Self Shadowing The strands should be self shadowing and this should create a more realistic image of the strands.

4 Implementation

The particle system will be implemented in C# and XNA 4.0 and we will try to give a top down view of the system in this section.

4.1 Particle System Structure

The particle system basically consists of Emitters which has a list of Strands which it can apply a set of EmitterRules on as described in figure 1. The Application class described only handles set up of windows, the initialization of Emitters and provides a loop for updating the rules/emitter and drawing the strands/emitter so it does not really handle anything of interest. Once a emitter has been initialized and none of the rules have been applied it will look like the image in figure 2.

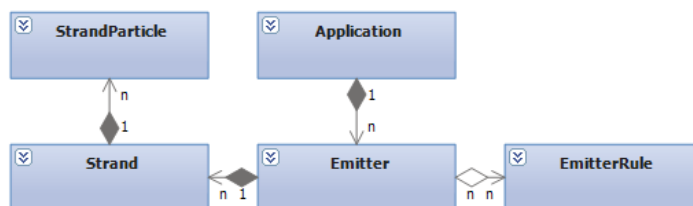

Figure 1: A overview of the particle system.

The Emitter class is initially based on the emitter structure by Jeff Lander [3] but has been modified in order to allow for strands instead of particles. We've modified the Emitter structure to allow for easier implementation of new rules by extracting the rules from the Emitter to a separate EmitterRule class. The Strand class is basically a list of StrandParticles which are created in the Strand constructor from the given parameters.

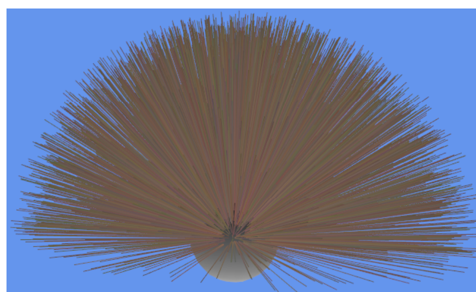

Figure 2: A view of the Emitter once all strands have been drawn but none of the rules are applied.

It's quite a simple structure and the difficulty we're faced with after this is creating subclasses of EmitterRule which makes the particle system come alive. We also add a Shape class which is used by the collision rule and also by the Emitter class to set the initial position and direction of strands.

4.2 Particle System Rules

All the rules for the particle systems are trying to update particles in a strand. We will use the notation p_i for the particle they are trying to update at the moment and $p_i^{'}$ for the same particle but with updated position. The i indicates the distance from the root particle which is attached to the emitter. Once all the rules are applied it will look something like figure 3.

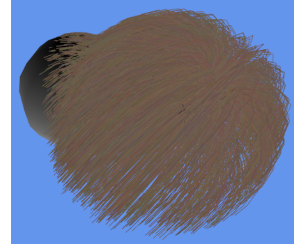

Figure 3: The result of all the particle system rules applied.

Elasticity For the elasticity rule we've used a known constraint rule [8]. The position p_i is the one being updated and p_{i-1} is the position which p_i is trying to stay close to. L is the distance the that the particles are trying to keep to each other.

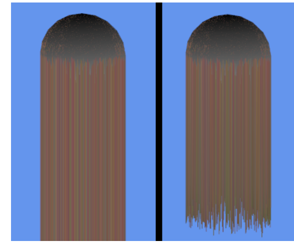

Figure 4: Gravity rule applied with elastic rule not applied to the left and elastic rule applied to the right.

$$p_i^{'} = p_i + (p_i - p_{i-1})\frac{L-|p_i-p_{i-1}|}{|p_i-p_{i-1}|}$$

The approximation ensures that the strands particles don't move too far from each other and allows a certain degree of freedom, a example of how the elasticity rule looks once applied can be seen in figure 4. The drawback of this approach is however if particle p_i and p_{i-1} would be in the same position it would cause a division by zero so we have to implement a special case for this when we don't move the particle at all since we don't know in which direction to move it.

Gravity The gravity rule is quite simple to implement and the result can be seen in figure 4. We have a vector g which determines the direction towards the centre of gravity as well as the strength of the pull and Δt is the time that has passed since the last update. We apply this rule on all the strand particles except the first one since it's in a fixed position by the emitter.

$$p_i^{'} = p_i + g\Delta t^2$$

Emitter motion To implement the emitter motion rule we simply have to apply the change in the emitters position to each strands root particle and the elasticity rule will make sure the hair follows the change in the emitters position naturally as seen in figure 5. This is the only rule that is applied to the root of the strand and which isn't applied to the other strand particles.

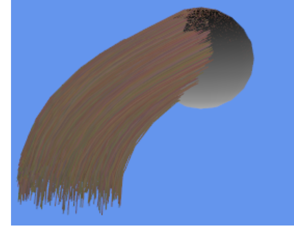

Figure 5: Hair moving to the right with gravity, elastic and motion rule applied

Wind The wind rule is somewhat expensive to calculate since we want the winds force on the strands to change depending on which angle the wind hits them, see figure 6 for the wind rule in action. We use a known formula [5] to calculate the impact of the wind on a particle where w is the vector which represents the wind, Δt is the time elapsed since the last update and d is the direction from p_{i-1} to p_i, i.e. $p_i - p_{i-1}$. We then change the value of w with small values over time to have the effect of the wind increasing and decreasing.

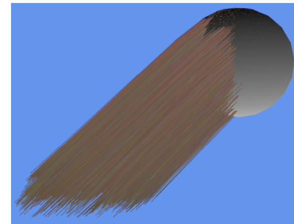

Figure 6: Hair with gravity, elasticy and a wind vector from the right

$$p_i^{'} = p_i + w\Delta t^2 \frac{|w \times d|}{|w|}$$

Collision For the collision rule, see figure 7, we simply check if any particle is inside a Shape object and the Shape object returns a vector with which to adjust the particle with so it ends up outside the Shape (We don't check the root particles though, since they are static except for them following the emitter). The only Shape objects we have implemented is spheres which makes the collision detection very simple since we only need to check the distance from the spheres origin, o, to the particle, p_i, and if the vectors length between these two points is less than the spheres radius, r, we adjust the particle with the following amount where d is the normalized vector $p_i - o$:

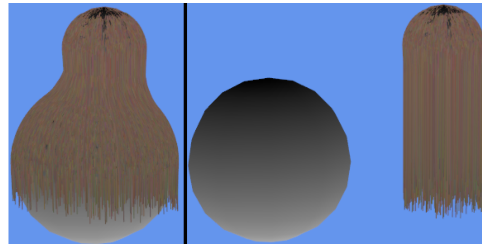

Figure 7: The hair from the emitter collides with the emitter sphere and a bigger sphere

$$p_i^{'} = p_i + (r - |p_i - o|) * d$$

Stiffness The stiffness rule, figure 8, we apply tries to make the strands go back to their original position as shown in figure 2 and it's stronger the closer the particle is to the root of the strand. We apply the rule on all particles except the root particle. The rule has a strength factor s that says how strongly the stiffness rule is applied on a strand of length l.

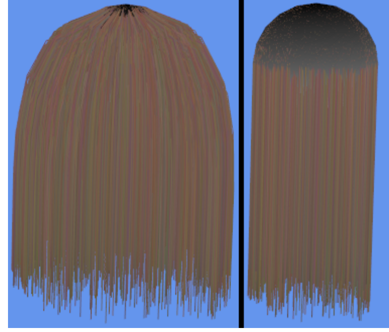

Figure 8: Hair with and without the stiffness rule applied while under the effect of the gravity and elasticity rule.

$$d_1 = p_{i-1} - p_{i-2}$$
$$d_2 = p_i - p_{i-1}$$
$$p_i^{'} = p_i + (l - i)s(d_1 - d_2)\Delta t^2$$

In case $i = 1$ then we set d_1 to the direction of the strand instead which is given by the emitter when the strand is created. There's also an exception to the rule, if the scalar $(l - i)s\Delta t^2$ is greater than 1 we simply set $p_i^{'} = p_i(d_1 - d_2)$. This is done in order to prevent the strands from swaying, since if they moved further than vector $d_1 - d_2$ they would in the next update have to move back a bit, which may result in them moving too far again and it may go back and forth like that.

4.3 Presentation

The presentation part of the implementation is where we discuss effect we apply on the strands to make them look better but which does not have to do with their behaviour which is controlled by the particle system.

Smoothness To smooth the jagged edges of a strand we implement Bézier curves to calculate points in between the strand particles. The difficulty in the implementation is to create good control points for the Bézier curve. A good approach is to calculate the tangents for a strand particle and use them to set the control points. To get the tangent [2] for a particle p_i we calculate vectors $v = p_i - p_{i-1}$ and $w = p_i - p_{i-2}$ and then calculate $u = v \times w$. Then we calculate $t_i = u \times (v + w)$ and get

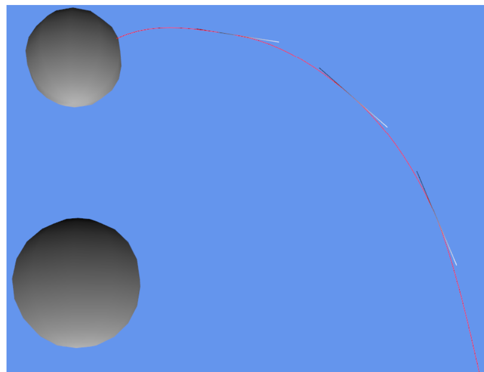

Figure 9: A four point strand that has been smoothened by Bézier curve.

the tangent t_i. We don't want control points to have too much effect though so we normalize the tangent and multiply it by a third of a strands segment length, l. So the control points between two points p_{i-1} and p_i are $p_{i-1} - t_{i-1}l/3$ and $p_i + t_il/3$.

If the points p_{i-1}, p_i and p_{i+1} are collinear however we can't calculate the cross product for v and u and we then simply set $t_i = \overline{0}$. When we implement this however it appears that if the points p_{i-1}, p_i and p_{i+1} are almost collinear but not quite the cross product of v and u becomes unstable, probably because of precision errors when calculating the cross product. So instead of checking if they are collinear we check if the vectors u and $v + w$ have a length greater than 0.1.

We also need to apply special cases for the first and last point since we can't calculate vectors v and w then. For the first point we set the tangent in the direction of the strand and for the last point we set the tangent to $\overline{0}$. In figure 9 you can see one smoothed strand (red) as well as the tangents (black and white) for each of the points that make out the strand.

Self Shadowing For self shadowing we first need to create a light source and have the hair reflect the light. This we do by applying a known model for achieving realistic light reflection on hair known as the Marschner reflectance model [6]. The model for this is quite long and is not directly related to particle systems so we won't go into detail how it works and recommend looking up the cited source instead. The result of the reflectance model can be seen in figure 10.

Figure 10: Result of the Marschner reflectance model

Now we're ready to move on to creating the self shadowing of the strands. We look up different techniques that have been applied before but they are too complex to implement in the time we have left of the project [6],[11],[10]. Instead we draw inspiration of what we have read and create a simple model which is fast to implement but does not achieve the same good results.

We create a depth map of all the strands which essentially is a texture which tells us what are the closest and furthest away positions per texel that the strands are being drawn at as seen from the light, see figure 11. This we can do on the GPU by creating a shader that calculates the vector from the light to the strands position which we normalizes. We then write the z value of this vector to the depth map textures red and alpha channel using a minimum respectively maximum blending function, and as such we get the

closest value in the red channel and the furthest away value in the alpha channel.

We can then use this depth map to create an interval in which all the strand lie, and depending on how close the strands are to the deepest position in comparison to the closest position we shadow the strand more. If r is the closest position to the light and a is the furthest away position and d is the position of the strand at that texel we then calculate x using the following formula:

$x = \frac{a-r}{d-r}$

The factor x is then a value in $[0,1]$ which tells us how deep down strand is with 0 being on the surface and 1being at the bottom. We then let the light diminish exponentially using the following formula:

Figure 11: A depth map of strands as seen from the light source

$y = \frac{e^x-1}{e-1}$

We then get a value y between $[0,1]$ which tells us how strong the shadow is. We simply multiply $1.0 - y$ by the light intensity/color before we apply it to the strands color. The result can be seen in figure 12

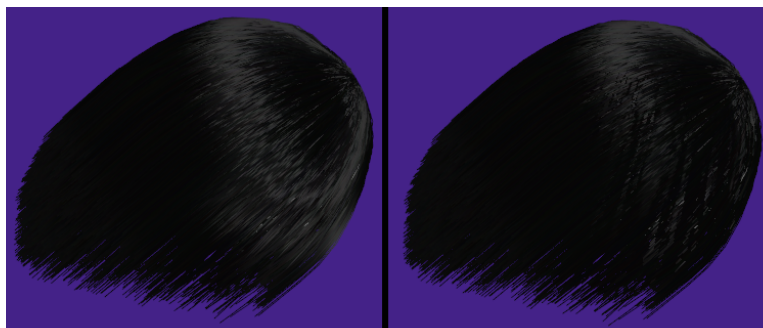

Figure 12: To the left unshadowed strands, to the right shadowed strands.

5 Result

We successfully implemented all the particle system rules we set out to do. The collision rule and stiffness rule may however be too simple since there exists some faults with them. Firstly the collision rule is point based and not segment based which makes it possible for strands to pass through objects if the segment length is great enough. The trade off is that its quite fast which allows for more strands to be animated. It would have been good to test the collision rule on different objects other than spheres as well but the reason we only implemented it for spheres now is that checking if a point is inside the sphere is very fast.

The stiffness rule suffers from that it's very sensitive to if the time steps, Δt, changes between updates which causes the strand to vibrate. By having a rule that is more based on a real phenomenon it could probably be made more stable. As long as the time steps between the updates are consistent it works quite well though.

The Béizer curve we implemented allows us to create smooth strands without having to have so many particles. Our implementation of it does however not look great when all three points after each other are collinear because of the problem mentioned before in section 4.3. When we check to see what performance impairing effect using Béizer curves has it takes about 30%-50% depending on how many extra segments we create which is very expensive. Maybe there's some way to improve our way of calculating Béizer curves but at this points it seems better to simply use more control points since the rules seems to be cheap to calculate in comparison.

The self shadowing of the strands is not ideal but it's better than nothing and at least it's fast, taking about 5% of the CPU time to create the depth map. The problems with the current model is that it does not handle thickness of the strands so the diminishing effect as the depth increases is not affected but the number of strands. It also can give jagged edges or shadow poorly if the strands are too far from the light source. As such the shadowing model should really be replaced and the best alternatives that we could find were either deep opacity maps [11] or a model by Erik Sintorn and Ulf Assarsson [10].

6 Discussion

Our implementation of a strand animation using a particle systems works quite well, but there's certain limitations to it, for example, in our implementation we render each strand as a line. That means that they don't really have any body and if you look up close on the emitter it becomes very visible. We could of course create a cylinder around each strand but this would be expensive to calculate. If we could have done it on the GPU it may have been fast enough, but we would need a geometry shader for that which XNA 4.0 does not support. The need for giving the strand a body is probably not worth the computation it would take even with a geometry shader unless you want to give the strand another shape than a cylinder though, since it would be something you wouldn't even notice in most simulations. If you'd do this you should probably introduce different level of detail so that you only render them with a body once they are close.

A geometry shader could also have been used to implement the Béizer curve on the GPU which would have eased the load on the CPU by about 50%. So we would recommend to use a library which supports geometry shaders if you would like to create strands using particle systems even if you're not thinking of giving them a body. You should probably here as well introduce different level of detail since it's unnecessary to smooth strands which are very far away from you.

Our particle system is quite limited in the number of particles it can create and render, at most we managed to render about 100000 particles with 10000 strands and still have an good frame rate around 25-30 fps. For comparison, using PhysX you can create a particle system using 840000 particles and 100000 strands [7] where the particle system is run on the GPU instead of CPU. So our implementation does not really have any value when it comes to real world application in games or animations since it does not utilize the hardware to the full extent like modern particle system engines does.

Our particle system serves it's purpose as a learning tool though and has helped us gain a better understanding of particle systems and strand animation but for practical application of particle systems you really need to utilize SDK libraries which takes advantage of the hardware power better. It can also be seen that once the particle system gets further away from the viewer the need for a high number of strands decreases as one can't tell very much difference between the rendered images, suggesting that it would be a good idea to try and limit the number of strands rendered once the particle system is further away from the viewer. This would free up the hardware for other tasks or more emitters, see figure 13 and figure 14 for comparisons.

Figure 13: To the left 1000 rendered strands, in the middle 5000 rendered strands, to the right 10000 rendered strands. Rendered a distance of 50 units from the camera.

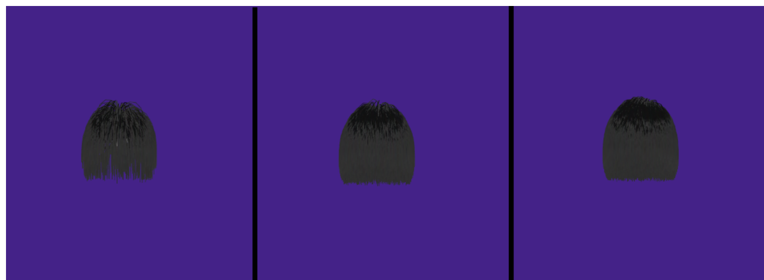

Figure 14: To the left 1000 rendered strands, in the middle 5000 rendered strands, to the right 10000 rendered strands. Rendered a distance of 150 units from the camera.

7 Conclusions

We manage to create a implementation of a particle system simulating strands which can create a decent image of moving hair/strands. It demonstrates the power of particle systems and how many simple rules together can create realistic effects as long as you create good rules for the system since it may otherwise spin out of control. We didn't manage to create smooth strands to a cheap cost though. To achieve this we should really have considered our choice of SDK libraries better and not gone with XNA 4.0 but perhaps instead OpenGL 3.2.

The shadowing effect we created was cheap and fast but not very good looking when the light source became distant so different techniques for shadowing is recommended. But this does not matter much, since the particle system is slow anyway when compared to modern particle system engines, and it can't really be used for anything other than learning purposes or simple simulations.

The most important lesson we've learned regarding particle system is probably though that you should try and limit the computation cost of the particle updates and also limit the number of particles. This can be done by when designing the rules that you perhaps don't always use the rules that give the most accurate representation of the real world but instead use approximations if they are faster. Since the more particles you can simulate the better looking particle system you can get it's very important that you keep the cost of the rules down, perhaps by using precalculated functions if the rules allow for it.

Another way of limiting the particle system cost is by introducing different level of detail depending on how close to the viewer the system is which limits the number of particles being drawn. This can be done either by varying the number of particles for each strand or the number of strands at a time. Best would probably be to vary the number of strands and not the particles that a strand consists of since removing strands should both be easier and not limit the freedom of the strands movements as much.

If it would be more easier to remove strand or particles from strands would probably depend on the implementation though. In out implementation each strand in a array of particles which would make it difficult to remove points on the strand. If we however were to develop the particle system on the GPU it may be easier to remove particles if you would store the strands such as all the root particles are in one texture and each successive layer is in different texture.

However you do it it would probably be a good idea to implement some kind of level of detail of particle systems if you wish to incorporate it into real time running systems.

8 References

[1] Yosuke Bandoy, Bing-Yu Chenz, and Tomoyuki Nishita. Animating hair with loosely connected particles. Technical report, The University of Tokyo, 2003.

[2] Paul Bourke. Piecewise cubic bézier curves. http://paulbourke.net/geometry/bezier/cubicbezier.html, March 2000. [Retrived 2012-03-17].

[3] Jeff Lander. The ocean spray in your face. Technical report, Game Developer, 1998.

[4] Jed Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. Technical report, Microsoft, 2001.

[5] Steve Lesser. Fast hair simulation and rendering using cuda and opengl. Technical report, Stanford University, 2011.

[6] Hubert Nguyen and William Donnelly. Chapter 23. hair animation and rendering in the nalu demo. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter23.html, April 2005. [Retrived 2012-04-07].

[7] NVIDIA. Gdc 2012: Tech demo walkthrough part 1 (cam) hd. http://www.gametrailers.com/video/gdc-2012-nvidia/727874, March 8 2012. [Retrived 2012-03-17].

[8] Masaki Oshita. Real-time hair simulation on gpu with a dynamic wisp model. Technical report, Kyushu Institute of Technology, 2007.

[9] William T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. Technical report, Lucasfilm Ltd, 1983.

[10] Erik Sintorn and Ulf Assarsson. Hair self shadowing and transparency depth ordering using occupancy maps. Technical report, Chalmers University of technology, 2009.

[11] Cem Yuksel and John Keyser. Deep opacity maps. *EUROGRAPHICS*, 27(2), 2008.